Software Thread Level Speculation for the Java Language and Virtual Machine Environment

Christopher J.F. Pickett Clark Verbrugge

School of Computer Science, McGill University
Montréal, Québec, Canada H3A 2A7
{cpicke,clump}@sable.mcgill.ca

Abstract. Thread level speculation (TLS) has shown great promise as a strategy for fine to medium grain automatic parallelisation, and in a hardware context techniques to ensure correct TLS behaviour are now well established. Software and virtual machine TLS designs, however, require adherence to high level language semantics, and this can impose many additional constraints on TLS behaviour, as well as open up new opportunities to exploit language-specific information. We present a detailed design for a Java-specific, software TLS system that operates at the bytecode level, and fully addresses the problems and requirements imposed by the Java language and VM environment. Using SableSpMT, our research TLS framework, we provide experimental data on the corresponding costs and benefits; we find that exceptions, GC, and dynamic class loading have only a small impact, but that concurrency, native methods, and memory model concerns do play an important role, as does an appropriate, language-specific runtime TLS support system. Full consideration of language and execution semantics is critical to correct and efficient execution of high level TLS designs, and our work here provides a baseline for future Java or Java virtual machine implementations.

1 Introduction

Thread level speculation (TLS), also known as speculative multithreading (SpMT), is a technique for automatic program parallelisation that has been investigated from a hardware perspective for several years, and current systems are capable of showing good speedups in simulation based studies [1, 2]. As a hardware problem, the issues of ensuring correctness under speculative execution have been well defined, and different rollback or synchronization approaches are sufficient to guarantee overall correct program behaviour. Software approaches to TLS, however, need to take into account the full source language semantics and behaviour to ensure correct and efficient execution, and in general this is not trivially ensured by low level hardware mechanisms.

In this paper we provide a detailed description of the requirements and performance impact of various high level aspects of Java TLS execution. We consider the full Java semantics, including all bytecode instructions, garbage collection (GC), synchronization, exceptions, native methods, dynamic class loading, and the new Java memory model [3]. These requirements are often dismissed or ignored in existing Java TLS work, but in fact are crucial to correct execution and can significantly affect performance.

Language and VM level speculation also produce design constraints due to efficiency concerns; for instance, Java programs tend to have frequent heap accesses, object allocations, and method calls. Our runtime TLS support system accomodates this behaviour, and we evaluate the relative importance of dependence buffering, stack buffering, return value prediction, speculative allocation, and priority queueing.

General purpose software and intermediate, VM level implementations of TLS are difficult goals, but have significant potential advantages, including the use of high level program information and the ability to run on existing multiprocessor hardware. Rather than describe a series of optimisations to eliminate previously characterized thread overheads [4], our work here is intended to provide a thorough Java TLS design and an understanding of the requirements and relative impact of high level language semantics.

1.1 Contributions

We make the following specific contributions:

- We provide a complete design for TLS at the level of Java bytecode. We modify existing instructions for speculative safety and introduce only two new bytecodes, `SPMT_FORK` and `SPMT_JOIN`. We also present software implementations of various runtime support components suitable for the Java virtual machine environment.
- We provide a detailed exposition of how high level Java language constructs and semantics affect TLS design. This includes object allocation, garbage collection, native methods, exceptions, synchronization, and the new Java memory model.
- We analyse the impact of high level safety considerations and the benefits derived from our runtime support components using an implementation of this design, the SableSpMT analysis framework [4].

In the following section we present related work on TLS and Java designs in that context. Then we describe our basic TLS threading model and provide an overview of SableSpMT in Section 3. Details of our design for Java TLS are described in Section 4, and intricacies of the Java language are considered in Section 5. Experimental analyses of both the impact of safety constraints and mechanisms that support Java TLS execution are given in Section 6. Finally, we conclude and discuss future work in Section 7.

2 Related Work

Thread level speculation has been the subject of hardware investigations for over a decade, and a variety of general purpose machines have been proposed and simulated [5–7]. These have also been tailored to specific speculation strategies; *loop level* speculation focusses on loop iterations [8], whereas *method level* speculation or *speculative method level parallelism* (SMLP) [9] speculates over method calls. SMLP has been identified as particularly appropriate for Java, given the relatively high density of method calls in Java programs, and simulation studies have shown quite good potential speedup [9]. The impact of frequent method calls was further explored and optimised by Hu *et al.* in their study of return value prediction [10].

Most current hardware designs could in fact be classified as hybrid hardware/software approaches since they rely to various extents on software assistance. Most commonly, compiler or runtime processing is required to help identify threads and insert appropriate TLS directives for the hardware [11, 12]. Jrpm makes further use of several code optimisations that reduce variable dependencies [1], and other recent designs such as STAMPede [2] and Mitosis [13] are based to a large degree on cooperative compiler and software help.

Speculative hardware, even with software support, largely obviates the consideration of high level language semantics: correct machine code execution implies correct program behaviour. Pure software architectures based on C or FORTRAN also have relatively straightforward mappings to speculative execution, and thus systems such as Softspec [14], thread pipelining [15], and others [16, 17] do not require a deep consideration of language semantics.

For Java stronger guarantees must be provided. In the context of designing JVM rollback for debugging purposes some similar semantic issues have been considered [18], but much less so for Java TLS. As part of their software thread partitioning strategy, Chen and Olukotun do discuss Java exceptions, GC, and synchronization requirements [1]. However, they do not consider class loading, native methods, or copying GC behaviour, and nor does their handling of speculative synchronization by simply ignoring it correctly enforce Java semantics. Pure Java source studies, such as the partially or fully hand-done examinations by Yoshizoe *et al.* [19] and Kazi and Lilja [20], focus on small execution

traces in a limited environment or rely on human input respectively. In the former case the environment is too constrained for Java language issues to arise. In the latter, exceptions, polymorphism, and GC are discussed, though not analysed, and assumptions about ahead-of-time whole program availability are contrary to Java's dynamic linking model. These are not *a priori* clearly insignificant differences; the effect of dynamic class loading in Java, for instance, has spawned a large number of non-trivial optimisation considerations [21], and despite Kazi and Lilja's dismissal of GC as unimportant for applications with small footprints, many Java applications *do* have large memory requirements [22, 23]. Differences and omissions such as these make it difficult to compare Java studies, and leave important practical implementation questions open; our work here is meant to help rectify this situation.

3 Background and System Overview

In our design for Java TLS we employ *speculative method level parallelism* (SMLP), as depicted in Figure 1. SMLP uses method callsites as fork points: the parent thread enters the method body, and the child thread begins execution at the first instruction past the callsite. When the parent returns from the call, then if there are no violations the child thread is committed and non-speculative execution continues where speculation stopped, otherwise the parent re-executes the child's body. SMLP accommodates Java's dense object-oriented method invocation structure, and has previously been demonstrated as a useful TLS paradigm for the language [1, 10].

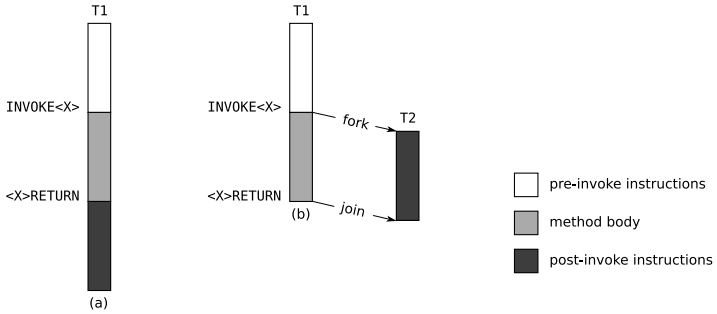

Fig. 1. (a) *Sequential execution of Java bytecode.* The target method of an `INVOKE<X>` instruction executes before the instructions following the return point. (b) *Speculative execution of Java bytecode under speculative method level parallelism (SMLP).* Upon reaching a method callsite, the non-speculative *parent* thread `T1` forks a speculative *child* thread `T2`. If the method is non-void, a predicted return value is pushed on `T2`'s Java operand stack. `T2` then continues past the return point in parallel with the execution of the method body, buffering main memory accesses. When `T1` returns from the call, it joins `T2`. If the actual return value matches the predicted return value, and there are no dependence violations between buffered reads and post-invoke values, `T2`'s buffered writes are committed and non-speculative execution jumps ahead to where `T2` left off, yielding speedup. If there *are* dependence violations or the prediction is incorrect, `T2` is simply aborted.

An overview of the SableSpMT execution environment and Java TLS analysis framework [4] is shown in Figure 2. SableSpMT is an extension of the "switch" bytecode interpreter in SableVM [24], a Free / open source software Java virtual machine. SableVM adheres to the JVM Specification [25], and is capable of running Eclipse and other large, complex programs. Static analysis with Soot [26] occurs ahead-of-time, and SableSpMT uses the results to prepare special speculative *code arrays* for Java methods from their non-speculative equivalents in SableVM; code arrays are generated from Java bytecode, and are contiguous sequences of word-sized instructions and instruction operands representing method bodies. SableSpMT forks and joins child threads at runtime, and these

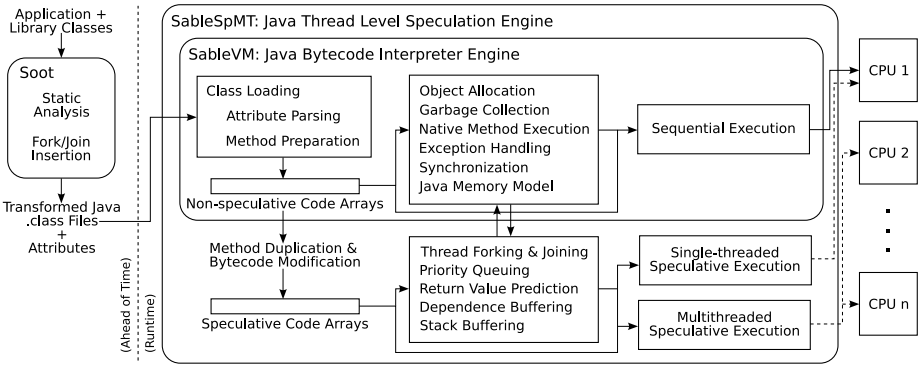

Fig. 2. *The SableSpMT thread level speculation execution environment.* SableSpMT is an extension of SableVM. Soot is used to transform, analyse, and attach attributes to `.class` files in an ahead-of-time step. SableVM reads in these classes during class loading, parsing attributes and preparing method bodies. Sequential execution depends only the non-speculative code arrays, and interacts with normal JVM support components. Speculative execution requires preparation of special speculative code arrays, and depends on additional TLS support components. SableSpMT's single-threaded execution mode shares processors with non-speculative execution, whereas the multithreaded mode splits single non-speculative threads across multiple processors.

depend on the speculative code arrays for safe out-of-order execution. Various TLS runtime support facilities are needed, including priority queueing, return value prediction, dependence buffering, and stack buffering. SableSpMT also interacts with SableVM's runtime support components, including a semi-space copying garbage collector, native method execution, exception handling, synchronization, and the Java memory model. Outside of thread forking and joining, speculation has negligible impact on and is largely invisible to normal multithreaded VM execution, with $s = \max(n - p, 0)$ speculative threads running on free processors, where n is the number of processors and p is the number of non-sleeping non-speculative parent Java threads.

4 Java TLS Design

We now describe the main Java TLS structures in our design for SMLP at the virtual machine level. These can be broadly classified into speculative method preparation components, speculative runtime support components, and speculative execution modes.

4.1 Speculative Method Preparation

The preparation of method bodies for TLS can be broken into several steps. Static analysis takes place and classfile attributes are parsed, fork and join points are inserted, bytecode instructions are modified, and parallel speculative code arrays are generated. Some of these steps take place ahead of time as a matter of technical convenience, and may overlap with each other.

The final stages of preparation occur when a method is invoked for the first time. Once primed for speculation, a child thread can be forked at any callsite within the method body. Furthermore, speculation can continue across method boundaries as long as the methods being invoked or returned to have been similarly prepared.

Static Analysis and Attribute Parsing An advantage to language level TLS is the ability to use high level program information. In our case we incorporate information from the Soot compiler analysis framework [26], and include two analyses for improved return value prediction [27]. The first is a *parameter dependence* analysis that determines which method parameters will affect the return value; this is used to reduce the memory requirements and improve the accuracy of a memoization predictor. The second is a

return value use analysis that detects return values that are unconsumed or appear only inside boolean and branch expressions; this is used to relax constraints on predictor accuracy. The results are encoded using Soot's attribute generation framework [28], and parsed by SableVM during class loading. During method preparation, the analysis data are associated with callsites for use by the return value prediction component.

Fork and Join Insertion The SableSpMT TLS engine needs the ability to fork and join child threads. We introduce new `SPMT_FORK` and `SPMT_JOIN` instructions that provide this functionality. Under SMLP threads are forked and joined immediately before and after method invocations, and so these instructions are inserted around callsites, represented by the `INVOKE<X>` instructions.

Soot is used in another AOT pass to perform the insertion. We place calls to dummy static void `Spmt.fork()` and `Spmt.join()` around every callsite, and during runtime method preparation these are replaced with the appropriate `SPMT_FORK` and `SPMT_JOIN` instructions. This approach has several advantages: first, transformed classfiles will run in the absence of TLS support, the dummy methods being trivially inlined; second, integration with a static analysis to determine good fork points is facilitated; and third, bytecode offsets are automatically adjusted.

Table 1. *Java bytecode instructions modified to support speculation.* Each instruction is marked according to its behaviours that require special attention during speculative execution. These behaviours are marked "once", "maybe", or "yes" according to their probabilities of occurring within the instruction. "Forces stop" indicates whether the instruction may force termination of a speculative child thread, but does not necessarily imply abortion and failure. Not shown are branch instructions; these are trivially fixed to support jumping to the right `pc`.

instruction	reads global	writes global	locks object	unlocks object	allocates object	throws exception	enters native code	loads class(es)	orders memory	forces stop
`GETFIELD`	yes					maybe		once	maybe	maybe
`GETSTATIC`	yes							once	maybe	maybe
`<X>ALOAD`	yes					maybe				maybe
`PUTFIELD`		yes				maybe		once	maybe	maybe
`PUTSTATIC`		yes						once	maybe	maybe
`<X>ASTORE`		yes				maybe				maybe
`(I|L)(DIV|REM)`						maybe				maybe
`ARRAYLENGTH`						maybe				maybe
`CHECKCAST`						maybe		once		maybe
`ATHROW`						yes				yes
`INSTANCEOF`								once		maybe
`RET`										maybe
`MONITORENTER`	yes	yes	yes			maybe			yes	yes
`MONITOREXIT`	yes	yes		yes		maybe			yes	yes
`INVOKE<X>`	maybe	maybe	maybe			maybe	maybe	once	maybe	maybe
`<X>RETURN`	maybe	maybe		maybe		maybe	maybe	once	maybe	maybe
`NEW`		yes			yes	maybe		once		maybe
`NEWARRAY`		yes			yes	maybe				maybe
`ANEWARRAY`		yes			yes	maybe		once		maybe
`MULTIANEWARRAY`		yes			yes	maybe		once		maybe
`LDC_STRING`					once					once

Bytecode Instruction Modification The majority of Java's 201 bytecode instructions can be used verbatim for speculative execution; however, roughly 25% need modification to protect against potentially dangerous behaviours, as shown in Table 1. If these instructions were modified in place, the overhead of extra runtime conditionals would impact on the speed of non-speculative execution. Instead, modification takes place in a duplicate copy of the code array created especially for speculative execution. Indeed, the only significant change to non-speculative bytecode is the insertion of fork and join points. Problematic operations include:

- *Global memory access*. Reads from and writes to main memory require buffering, and so the `<X>A(LOAD|STORE)` and `(GET|PUT)(FIELD|STATIC)` instructions are modified to read and write their data using a dependence buffer, as described in Section 4.2. If final or volatile field access flags are set, these instructions may require a memory barrier, as described in Section 5, in which case speculation must also stop.
- *Exceptions*. In unsafe situations, many instructions must throw exceptions to ensure the safety of bytecode execution [25], including `(I|L)(DIV|REM)` that throw ArithmeticExceptions upon division by zero, and others that throw NullPointerExceptions, ArrayIndexOutOfBoundsExceptions, and ClassCastExceptions. Application or library code may also throw explicit exceptions using `ATHROW`. In both cases, speculation rolls back to the beginning of the instruction and stops immediately; however, the decision to abort or commit is deferred until the parent joins the child. Exceptions must also be handled safely if thrown by non-speculative parent threads with speculative children, as discussed in Section 5.
- *Detecting object references.* The `INSTANCEOF` instruction computes type assignability between a pre-specified class and an object reference on the stack. Normally, bytecode verification promises that the stack value is always a valid reference to the start of an object instance on the heap, but speculative execution cannot depend on this guarantee. Accordingly, speculation must stop if the reference does not lie within heap bounds, or if it does not point to an object header; currently we insert a magic word into all object headers, although a bitmap of heap words to object headers would be more accurate and space-efficient.
- *Subroutines.* `JSR` (jump to subroutine) is always safe to execute because the target address is hardcoded into the code array. However, the return address used by its partner `RET` is read from a local variable, and must point to a valid instruction. Furthermore, for a given subroutine, if the `JSR` occurs speculatively and the `RET` non-speculatively, or vice versa, the return address must be adjusted to use the right code array. Thus a modified *non-speculative* `RET` is also needed.
- *Synchronization.* The `INVOKE<X>` and `<X>RETURN` instructions may lock and unlock object monitors, and `MONITOR(ENTER|EXIT)` will always lock or unlock object monitors; they furthermore require memory barriers and are strongly ordering. These instructions are also marked as reading from and writing to global variables, as lockwords are stored in object headers. Speculative locking and unlocking is not currently supported, and always forces children to stop.
- *Method entry.* Speculatively, `INVOKE<X>` are prevented from entering unprepared methods and triggering class loading and method preparation. Furthermore, at non-static callsites, the receiver is checked to be a valid object instance, the target is checked to have the right stack effect, and the type of the target's class is checked for assignability to the receiver's type. Invokes are also prevented from entering native code or attempting to execute abstract methods.
- *Method exit.* After the synchronization check, the `<X>RETURN` instructions require three additional safety operations: 1) potential buffering of the non-speculative stack frame from the parent thread, as described in Section 4.2; 2) verifying that the caller is not executing a *preparation sequence*, a special group of instructions used in SableVM to replace slow instructions with faster versions [24]; and 3) ensuring that speculation does not leave bytecode execution entirely, which would mean Java thread death, VM death, or a return to native code.
- *Object allocation*. Barring an exception being thrown or GC being triggered, the `NEW` and `((MULTI|)A|)NEWARRAY` instructions are safe to execute. The `LDC_STRING` specialisation of `LDC` allocates a constant `String` object upon its first execution, the

address of which is patched into both non-speculative and speculative code arrays, and forces speculation to stop only once. Allocation and GC are discussed in greater detail in Section 5.

To the best of our knowledge, Table 1 is comprehensive. The outlined modifications are enough to support TLS for the SPECjvm98 benchmarks and are consistent with our understanding of the JVM Specification [25].

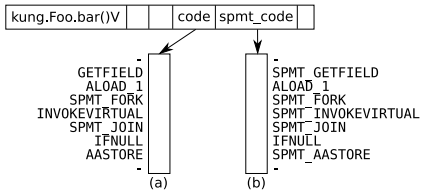

Fig. 3. *Parallel code arrays.* (a) non-speculative code array prepared for method `bar()`; (b) speculative version of the same code array with modified instructions.

Parallel Code Array Generation The goal of this extensive bytecode modification is to prepare parallel code arrays for speculative execution, as shown in Figure 3. The non-speculative array is duplicated, branch targets are adjusted, and modified instructions replace ordinary non-speculative versions where necessary. Additionally, `SPMT_FORK` and `SPMT_JOIN` surround every `INVOKE<X>` in *both* code arrays. Transitions between non-speculative and speculative execution are facilitated by identical instruction offsets in each array.

4.2 Speculative Runtime Support

In addition to preparing method bodies for speculative execution, the speculation engine provides various support components that interact with bytecode and allow for child thread startup, queueing, execution, and death to take place while ensuring correct execution through appropriate dependence buffering.

Thread Forking Speculative child threads are forked by non-speculative parents and also by speculative children at `SPMT_FORK` instructions. Speculating at every fork point is not necessarily optimal, and in the context of SMLP various heuristics for optimising fork decisions have been investigated [12]. SableSpMT permits relatively arbitrary fork heuristics; however, we limit ourselves to a simple "always fork" strategy in this paper as a more generally useful baseline measurement.

Having made the decision to fork a child, several steps are required. First, those variables of the parent thread environment (`JNIEnv`) that can be accessed speculatively are copied to a child `JNIEnv` struct; in this fashion, the child assumes the identity of its parent. Second, a child stack buffer is initialized and the parent stack frame is copied to the child, giving it an execution context. Third, a dependence buffer is initialized; this protects main memory from speculative execution, and allows for child validation upon joining. Fourth, the operand stack height of the child is adjusted to account for the stack effect of the invoke following the fork point, and the `pc` of the child is set to the first instruction past the invoke. Fifth, a return value is predicted for non-void methods; technically, any arbitrary value can be used as a "prediction", although the chance of speculation success is greatly reduced by doing so. In the above steps, memory reuse is critical in reducing the overhead of thread environment, dependence buffer, and stack buffer allocation.

Priority Queueing In the default multithreaded speculative execution mode, children are enqueued at fork points on a global $O(1)$ concurrent priority queue; higher priority

threads are those that are expected to do more useful work. The queue consists of an array of doubly-linked lists, one for each priority from 0–10, and supports `enqueue`, `dequeue`, and `delete` operations. Helper OS threads compete to dequeue and run children on separate processors. There is a single test-and-`test-and-set` (TATAS) lock protecting the queue, and queue synchronization is a small but non-negligible source of overhead. Priorities 0–10 are computed as $\min(l \times r/1000, 10)$, where l is the average bytecode sequence length and r is the success rate. We find that this function gives acceptable distributions, if somewhat biased towards lower priorities.

Shavit *et al.* considered scalable concurrent priority queues [29], and found that for a small number of priorities and processors that this design is optimal, except that synchronizing per-priority and using MCS [30] queue locks instead of TATAS spinlocks may afford some improvements. Closely related CLH locks [31] are available in SableSpMT; we find that although they distribute queue access much more evenly amongst competing threads, no speedup over TATAS locks is achieved.

Return Value Prediction Speculative children forked at non-void callsites need their operand stack height adjusted to account for the return value, and must be aborted if an incorrect value is used. Accurate return value prediction (RVP) can significantly improve the performance of Java SMLP [10], and we previously reported on our aggressive RVP implementation in SableSpMT [32], the use of two compiler analyses for extracting further accuracy [27], and the integration of RVP analysis into our framework [4].

Return value predictors are associated with individual callsites, and can use context, memoization, and hybrid strategies, amongst others. Additionally, attributes generated by the compiler analyses are parsed during method preparation, and can be used to relax predictor correctness requirements and reduce memory consumption. Accurate RVP can incur significant overheads [4], and it is likely that not synchronizing on dynamically expanding predictor hashtables and disabling sub-optimal predictors on a per-callsite basis can help to minimize the cost.

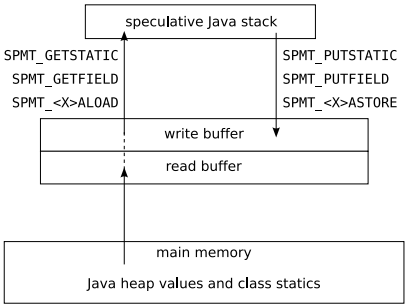

Fig. 4. *Dependence buffering.* When a speculative global load instruction is executed, first the write buffer is searched, and if it does not contain the address of the desired value then the read buffer is searched. If the value address is still not found, the value at that address is loaded from main memory. When a speculative global write instruction is executed, the write buffer is searched, and if no entry is found a new mapping is created.

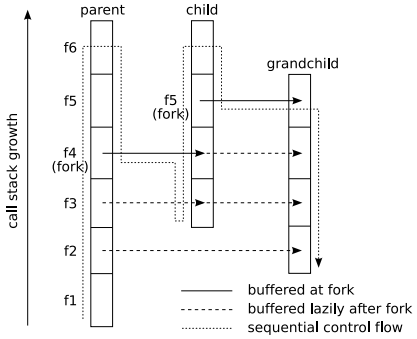

Fig. 5. *Stack buffering. f1* through *f6* are stack frames corresponding to Java methods. A speculative child is forked at *f4* in the parent, and in turn a second-generation grandchild thread is forked at *f5* in the child. Stack frames are buffered on forking, and additionally when children return from methods; *f2* in the grandchild is buffered from the non-speculative parent, as its immediate ancestor never descended below *f3*.

Dependence Buffering Most TLS designs propose a mechanism for buffering reads from and writes to main memory by speculative threads in order to prevent against potential dependence violations. In Java, main memory consists of object instances and arrays on the garbage-collected heap, and static fields in class loader memory.

In hardware, dependence buffers can be built as table based structures similar to caches [2], and we propose a similar design for software TLS, as shown in Figure 4. Buffer objects are attached to speculative threads on startup, and internally consist of nine read and nine write sub-buffers, specialised for the eight primitive types and also for object references. Use of type-specific sub-buffers is an optimisation possible in high level TLS environments, and helps reduce buffer space requirements. The sub-buffers are implemented as open addressing hashtables; values are stored using the value address as a key, and fast lookup is provided by double hashing. A backing linked list allows for fast iteration during validation and committal.

Stack Buffering As well as heap and static data, speculative threads may also access local variables and data stored on the Java operand stack. It follows that stack accesses must be buffered to protect the parent stack in the event of failure, as shown in Figure 5. The simplest mechanism for doing so is to copy stack frames from parent threads to separate child stacks both on forking children and on exiting methods speculatively. Additionally, children must create new stack frames for any methods they enter.

Pointers to child threads are stored one per stack frame, and this allows for convenient *out-of-order* thread spawning [33] where each parent can have multiple immediate children, exposing additional parallelism. Although not supported by our SableSpMT implementation at this time, child threads can also fork their own children, which when combined with out-of-order spawning leads to a tree of children for a single fork point.

Thread Joining Upon reaching some termination condition, a speculative child will stop execution and leave its entire state ready for joining by its parent. The child may stop of its own accord if it attempts some illegal behaviour as summarized in Table 1, if it reaches an *elder sibling*, that is, a speculative child forked earlier on by the same parent at a lower stack frame, or if it reaches a pre-defined speculative sequence length limit. The parent may also signal the child to stop if it reaches the join point associated with the child's fork point, or if it reaches the child's forking frame at the top of the VM exception handler loop. SableSpMT uses a per-instruction asynchronous polling strategy within speculative threads to detect parent signals. Optimising poll points to occur only at backward branches may offer improvements in the speed of speculative bytecode interpretation, at the expense of longer wait times for the parent.

The join process involves verifying the safety of child execution and committing results. First, a full memory barrier is issued, and the child is then validated according to four tests: 1) the predicted return value is checked against the actual return value for non-void methods, according to the safety constraints of static analyses [27]; 2) the parent is checked for not having had its root set garbage-collected since forking the child; 3) the dependence buffers are checked for overflow or corruption; and 4) values in the read dependence buffer are checked against main memory for violations.

If the child passes all four tests, then the speculation is safe; all values in the write buffer are flushed to main memory, buffered stack frames entered by the child are copied to the parent, and non-speculative execution resumes with the `pc` and operand stack size set as the child left them. Otherwise, execution continues non-speculatively at the first instruction past the `SPMT_JOIN`. Regardless of success or failure, the child's memory is recycled for use at future fork points. Note that buffer commits may result in a reordering of the speculative thread's write operations, which must in turn respect the requirements imposed by the new Java memory model, as discussed in Section 5.

4.3 Speculative Execution

SableSpMT supports two speculative execution modes, a single-threaded mode where bytecode interpretation alternates between non-speculative and speculative execution in a single thread, and a truly multithreaded mode that depends on multiple processors for parallelisation. Both modes allow for non-speculative Java threads to coexist with the speculative sytem. The former mode has previously been described as appropriate for debugging, testing, porting, and limit analyses [4], and so here we focus on multithreaded execution.

In the multithreaded mode, children are assigned priorities at fork points based on speculation histories, and enqueued on the $O(1)$ priority queue. A minimal amount of initialization is done to limit the impact of fork overhead on non-speculative threads. There is a pool of helper OS threads running, one per free processor, and these dequeue and execute children according to priority.

If the parent thread joins a child that it previously enqueued, and that child did not get removed by a helper OS thread, the child is deleted by simply unlinking it from the list for that priority, and its memory is recycled. Otherwise, if the child has started, the parent signals it to stop, and then begins the usual validation procedure. If successful, the parent jumps ahead, otherwise the parent simply continues.

5 Java Language Considerations

Several traps await the unsuspecting implementor that tries to enhance a JVM to support thread level speculation. These traps are actually core features of the Java language — object allocation, garbage collection, native method execution, exception handling, synchronization, and the Java memory model — and a Java TLS implementation must handle them all safely in order to be considered fully general. The impact of these features is measured in Section 6.

Object Allocation Object allocation occurs frequently in many Java programs, and permitting speculative allocation significantly increases maximum child thread lengths. Additionally, it is unnecessary to buffer accesses to objects allocated speculatively. Speculative threads can either allocate without synchronization from a thread-local heap, or compete with non-speculative threads to acquire a global heap mutex. Normally, speculation must stop if the object to be allocated has a non-trivial finalizer, i.e. not `Object.finalize()`, for it would be incorrect to finalize objects allocated by aborted children; however, in SableVM, finalization is disabled altogether, as permitted by the JVM Specification [25]. Allocation also forces speculation to stop if either GC or an OutOfMemoryError would be triggered as a result. Object references only become visible to non-speculative Java threads upon successful thread validation and committal; aborted children will have their allocated objects reclaimed in the next collection. Although this does increase collector pressure, we did not observe any difference in GC counts at the default heap size when speculative allocation was enabled.

Garbage Collection All objects in Java are allocated on the garbage-collected Java heap. This is one of the main attractions of the language, and as such, any serious proposal to extend it must consider this feature; indeed, many Java programs will simply run out of memory without GC. SableVM uses a stop-the-world semi-space copying collector by default [24], and every object reference changes upon every collection; thus, any speculative thread started before GC must be invalidated after GC. Speculative threads are invisible to the rest of the VM, and are not stopped or traced during collection; however, heap accesses are buffered, and so speculation can safely continue during GC. Threads are invalidated if the collection count of the parent thread increases between the fork and join points. The default collector in SableVM is invoked relatively infrequently, and we

find that GC is responsible for a negligible amount of speculative invalidations. Other GC algorithms are trickier to negotiate with, and may require either pinning of speculatively accessed objects or updating of dependence buffer entries.

Native Methods Java provides access to native code through the Java Native Interface (JNI), and native methods are used in class libraries, application code, and the VM itself for low-level operations such as thread management, timing, and I/O. Speculation must stop upon encountering native methods, as these cannot be executed in a buffered environment without significant further analysis. However, non-speculative threads can safely execute native code while their speculative children execute pure bytecode continuations.

Exceptions Implicit or explicit exceptions simply force speculation to stop. Speculative exception handling is not supported in SableSpMT for three reasons: 1) exceptions are rarely encountered, even for "exception-heavy" applications like `jack` [32]; 2) writing a speculative exception handler is somewhat complicated; and 3) exceptions in speculative threads are often the result of incorrect computation, and thus further progress is likely to be wasted effort. In Java source code, `try` {} `catch()` {} and `try` {} `catch()` {} `finally` {} may be compiled to use exception handlers with `JSR` and `RET` instructions [25]. The speculative safety of these instructions is discussion in Section 4.1, and does not depend on their usage for exception handling.

Non-speculatively, if exceptions are thrown out of a method in search of an appropriate exception handler, any speculative children encountered as stack frames are popped must be aborted. In order to guarantee a maximum of one child per stack frame, children *must* be aborted at the *top* of the VM exception handler loop, before jumping to the handler `pc`. This prevents speculative children from being forked inside either `catch` or `finally` blocks while another speculative child is executing in the same stack frame.

Synchronization Object access is synchronized either explicitly by the `MONITORENTER` and `MONITOREXIT` instructions, or implicitly via synchronized method entry and exit. Different groups have explored speculative locking [34, 35], in which reads and writes to global object locks are buffered, and this will be interesting to consider in future work. In the absence of such strategies, speculative synchronization is prohibited and must force children to stop; somewhat surprisingly, synchronization has been unsafely ignored by Java TLS studies in the past [1, 10]. Non-speculatively, synchronization always remains safe, and it is even possible to fork and join speculative threads inside critical sections. Thus code which is traditionally considered a parallelism bottleneck can be parallelised, and this encourages coarse-grained locking, desirable from a software engineering perspective for its easier programmability.

The Java Memory Model Existing proofs on the safety of load and store reordering under TLS are correct for single-threaded programs [36], but the new Java memory model (JMM) [3] imposes additional constraints on multithreaded execution; in turn, the JSR-133 Cookbook specifies the insertion of memory barriers at various places in order to meet these constraints [37]. Speculative execution can only continue past a memory barrier if the dependence buffer records an exact interleaving of memory accesses and the relevant barrier operations; that we reuse entries for value addresses already in the buffer and do not record memory barriers precludes doing so in our current design.

The orderings required for various API calls, including non-speculative thread creation and joining, are provided by our design due to their implementations as native methods, which already force speculation to stop. For object synchronization several rules apply; most critically, a memory barrier is required before unlock operations to guarantee that writes in the critical section are visible to future threads entering the same monitor. By disabling speculative locking entirely we provide a much stronger guarantee than required; future work on speculative locking will need a finer grained approach.

Loads and stores of volatile fields also require memory barriers, to ensure interprocessor visibility between operations. Similarly, the loads and stores of final fields require barriers, except that on `x86` and `x86_64` these are no-ops [37]. However, speculatively, we must stop on final field stores, which appear only in constructors, to ensure that a final field is not used before the object reference has been made visible, a situation that is made possible by reordering writes during commit operations. Our conservative solution is to stop speculation on all volatile loads and stores and also all final stores.

In Java, finalizers are executed after object collection, typically by a separate finalizer thread. However, aggressive code optimisations can drastically shorten object lifetimes, such that an object finalizer might even run before initialization has completed [38], and accordingly, the new JMM specifies that finalization can only occur after the constructor has exited. This can be problematic for Java TLS if successful speculation past the constructor ends up deleting the object reference, and unordered commits allow the finalizer to run before all of the constructor's writes are flushed. We could conservatively disallow speculative threads to be joined if the parent encounters a non-trivial finalizer after forking; again, a further simplification is afforded by SableVM in that finalizers are not run at all. Avoiding finalizers is in general part of good Java programming practice.

6 Experimental Analysis

In this section we employ the SableSpMT framework to analyse the impact of both speculation support components and Java language features on TLS execution. All experiments were performed on a 1.8 GHz 4-way SMP AMD Opteron machine running Linux 2.6.7, with all free processors running speculative threads. We use the SPECjvm98 benchmark suite at size 100 (S100), and a speculative child thread is forked at every callsite. Nested speculation is disabled, but out-of-order spawning does take place. Although `raytrace` is technically not part of SPECjvm98 and therefore excluded from geometric means, we include results for purposes of comparison; it is the single-threaded equivalent of `mtrt`.

Table 2. *Child thread termination.*

termination reason	`comp`	`db`	`jack`	`javac`	`jess`	`mpeg`	`mtrt`	`rt`
class resolution and loading	2.14K	1.76K	94.8K	487K	3.80K	14.7K	4.79K	5.64K
failed object allocation	1	3	23	17	39	0	28	40
invalid object reference	563	553K	342K	280K	431K	485	407K	278K
finals and volatiles	842	1.45M	2.17M	1.11M	1.95M	888	115K	68.8K
synchronization	4.30K	26.8M	6.95M	17.0M	4.89M	10.4K	658K	351K
unsafe method entry or exit	2.66K	1.55K	16.0K	622K	2.62K	1.65K	3.60K	3.00K
implicit non-`ATHROW` exception	989K	828K	9.57K	572K	78.6K	2.00K	31.2K	20.8K
explicit `ATHROW` exception	0	0	187K	82	0	0	0	0
native code entry	332	28.2K	1.02M	1.02M	2.63M	527K	259K	260K
elder sibling reached	1.24M	3.81M	5.06M	16.1M	5.62M	14.1M	4.03M	4.23M
deleted from queue	348K	686	559K	3.13M	2.55M	4.48M	34.2M	1.57M
signalled by parent	202M	92.6M	20.1M	42.1M	56.3M	80.8M	122M	124M
TOTAL CHILD COUNT	204M	127M	36.5M	82.4M	74.5M	99.9M	162M	131M

In Table 2, total counts are given for all child thread termination reasons. In all cases, the majority of children are signalled by their parent thread to stop speculation. Significant numbers of child threads are deleted from the queue, and elder siblings are frequently reached. We looked at the average thread lengths for speculative children, and found them to be quite short, typically in the 0–10 instruction range. These data all indicate that threads are being forked too frequently, and are consistent with the general understanding of Java application behaviour: there are many short leaf method calls and the call graph is very dense [23]. Inlining methods will change the call graph structure, and it has previously been argued that inlined Java SMLP execution benefits from coarser granularity [10]. Introducing inlining into our system and exploring fork heuristics are therefore part of future work. Outside of these categories, it is clear that synchroniza-

tion and the memory barrier requirements for finals and volatiles are important; enabling speculative locking and recording barrier operations would allow threads to progress further. Native methods can also be important, but are much harder to treat. The other safety considerations of the Java language do not impact significantly on speculative execution; even speculative exceptions are responsible for a minority of thread terminations.

Table 3. *Child thread success and failure.*

join status	comp	db	jack	javac	jess	mpeg	mtrt	rt
exception in parent	0	0	386K	23.4K	0	0	0	0
incorrect prediction	18.0M	22.7M	2.80M	11.3M	5.80M	7.73M	4.85M	3.72M
garbage collection	4	20	119	206	470	0	90	68
buffer overflow	0	0	0	10	0	0	0	0
dependence violation	1.60M	1.44K	160K	1.53M	342K	14.7M	4.14M	4.00M
TOTAL FAILED	19.6M	22.7M	3.34M	12.9M	6.14M	22.4M	9.00M	7.72M
TOTAL PASSED	184M	103M	32.6M	66.4M	65.8M	73.0M	119M	122M

Data on the number of speculative thread successes and failures, as well as a breakdown of failure reasons, are given in Table 3. Failures due to GC, buffer overflows and exceptions are quite rare, and the majority of failures typically come from incorrect return value prediction. This again emphasizes the importance of accurate RVP in Java SMLP, and the weak impact of exceptions and GC. Dependence violation counts are not insignificant, and reusing predictors from the RVP framework for generalised load value prediction should help to lower them. In general, failures are much less common than successes, the geometric mean failure rate being 12% of all speculations. While this is encouraging, many threads are quite short due to an abundance of method calls and therefore forked children, and the high overheads imposed by thread startup, so it is likely the case that had they progressed a lot further, more violations would have occurred.

Table 4. *Impact of TLS support components on application speedup.* The priority queue was disabled by only enqueueing threads if a processor was free, return value prediction was disabled by always predicting zero, and the remaining components were disabled by forcing premature thread termination upon attempting to use them.

experiment	comp	db	jack	javac	jess	mpeg	mtrt	rt	mean
forced failure baseline	1297s	931s	293s	641s	665s	669s	1017s	1530s	722s
no priority queueing	0.94x	1.22x	1.35x	1.32x	1.58x	0.97x	1.68x	2.05x	1.27x
no return value prediction	1.03x	1.17x	1.28x	1.24x	1.44x	1.03x	1.72x	1.70x	1.25x
no dependence buffering	1.04x	1.22x	1.12x	1.05x	1.16x	1.02x	0.95x	0.97x	1.08x
no object allocation	0.95x	1.30x	1.39x	1.26x	1.55x	0.98x	1.13x	1.23x	1.21x
no method entry and exit	0.94x	1.02x	0.97x	0.98x	1.02x	0.95x	0.79x	0.91x	0.95x
full runtime TLS support	1.06x	1.27x	1.39x	1.37x	1.64x	1.01x	1.82x	2.08x	1.34x

Currently, thread overheads preclude actual speedup, and runtimes are within one order of magnitude [4]. This is competitive with hardware simulations providing full architectural and program execution detail [39], but we are also optimistic about techniques for achieving real speedup. In order to factor out the effects of fork and join overhead, we use a baseline execution time where speculation occurs as normal, but failure is automatically induced at every join point, calculating a mean relative speedup of 1.34x.

Table 4 shows the impact of disabling individual support components on Java TLS execution times. We note first of all that `compress` and `mpegaudio` are resilient to parallelisation, likely due to our current, naïve thread forking strategies. In some cases, disabling components can even lead to slight speedup. This phenomenon occurs if overhead costs outweigh component benefits; for example, disabling return value prediction can mitigate the cost of committing many short threads. In general, we can provide a partial ordering of support components by importance: the priority queue is least important; method entry and exit, or stack buffering, and dependence buffering are most important; return value prediction and speculative object allocation lie somewhere in-between.

7 Conclusions and Future Work

Language and software based thread level speculation requires non-trivial consideration of the language semantics, and Java in particular imposes some strong TLS design constraints. Here we have defined a complete system for Java TLS, taking into account various aspects of high level language and virtual machine behavioural requirements. Our implementation work and experimental analysis of Java-specific behaviour show that while most of these concerns do not result in a significant impact on TLS performance, conservatively correct treatment of certain aspects can reduce potential speedup, most notably synchronization. Part of our future work is thus to investigate different forms of speculative locking [34, 35] within a Java-specific context.

Our design focuses on defining correct Java semantics in the presence of TLS, and demonstrating the associated cost. However, as with any speculative system, performance and TLS overhead are also major concerns, and efforts to improve speedup in many fashions are worthwhile, as suggested by previous profiling results [4]. We are confident that overhead can be greatly reduced in our prototype implementation, through optimisation of individual components, greater use of high level program information, and employment of general and Java-specific heuristics for making forking decisions and assigning thread priorities. Further speedup is also expected by allowing speculative children to spawn speculative children, and by supporting load value prediction, both increasing the potential parallelism. Longer term future work includes an implementation of TLS within the IBM Testarossa JIT and J9 VM, where we hope to incorporate and measure these and other improvements, and research JIT-specific TLS problems and opportunities.

Acknowledgements

We would like to thank Etienne Gagnon for his help in SableVM development. We would also like to thank our colleagues Allan Kielstra and Mark Stoodley for constructive criticism of initial drafts of this paper. This research was funded by an IBM CAS fellowship, NSERC, FQRNT, and McGill University.

References

1. Chen, M.K., Olukotun, K.: The Jrpm system for dynamically parallelizing Java programs. In: ISCA. (2003) 434–446
2. Steffan, J.G., Colohan, C.B., Zhai, A., Mowry, T.C.: The STAMPede approach to thread-level speculation. TOCS (2005) To appear.
3. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL. (2005) 378–391
4. Pickett, C.J.F., Verbrugge, C.: SableSpMT: A software framework for analysing speculative multithreading in Java. In: PASTE. (2005)
5. Franklin, M.: The Multiscalar Architecture. PhD thesis, University of Wisconsin–Madison (1993)
6. Figueiredo, R., Fortes, J.: Hardware support for extracting coarse-grain speculative parallelism in distributed shared-memory multiprocessors. In: ICPP. (2001) 214–226
7. Ooi, C.L., Kim, S.W., Park, I., Eigenmann, R., Falsafi, B., Vijaykumar, T.N.: Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In: ICS. (2001) 368–380
8. Tsai, J.Y., Huang, J., Amlo, C., Lilja, D.J., Yew, P.C.: The superthreaded processor architecture. TC **48** (1999) 881–902
9. Chen, M.K., Olukotun, K.: Exploiting method-level parallelism in single-threaded Java programs. In: PACT. (1998) 176–184
10. Hu, S., Bhargava, R., John, L.K.: The role of return value prediction in exploiting speculative method-level parallelism. JILP **5** (2003)
11. Bhowmik, A., Franklin, M.: A general compiler framework for speculative multithreading. In: SPAA. (2002) 99–108
12. Whaley, J., Kozyrakis, C.: Heuristics for profile-driven method-level speculative parallelization. In: ICPP. (2005) 147–156

13. Quiñones, C.G., Madriles, C., Sánchez, J., Marcuello, P., González, A., Tullsen, D.M.: Mitosis compiler: An infrastructure for speculative threading based on pre-computation slices. In: PLDI. (2005) 269–279
14. Bruening, D., Devabhaktuni, S., Amarasinghe, S.: Softspec: Software-based speculative parallelism. In: FDDO-3. (2000)
15. Kazi, I.H., Lilja, D.J.: Coarse-grained thread pipelining: A speculative parallel execution model for shared-memory multiprocessors. TPDS **12** (2001) 952–966
16. Rundberg, P., Stenström, P.: An all-software thread-level data dependence speculation system for multiprocessors. JILP **3** (2001)
17. Cintra, M., Llanos, D.R.: Toward efficient and robust software speculative parallelization on multiprocessors. In: PPoPP. (2003) 13–24
18. Cook, J.J.: Reverse execution of Java bytecode. The Computer Journal **45** (2002) 608–619
19. Yoshizoe, K., Matsumoto, T., Hiraki, K.: Speculative parallel execution on JVM. In: 1st UK Workshop on Java for High Performance Network Computing. (1998)
20. Kazi, I.H., Lilja, D.J.: JavaSpMT: A speculative thread pipelining parallelization model for Java programs. In: IPDPS. (2000) 559–564
21. Arnold, M., Ryder, B.G.: Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In: ECOOP. Volume 2374 of LNCS. (2002) 498–524
22. Dieckmann, S., Hölzle, U.: A study of the allocation behavior of the SPECjvm98 Java benchmarks. In: ECOOP. Volume 1628 of LNCS. (1999) 92–115
23. Dufour, B., Driesen, K., Hendren, L., Verbrugge, C.: Dynamic metrics for Java. In: OOPSLA. (2003) 149–168
24. Gagnon, E.M.: A Portable Research Framework for the Execution of Java Bytecode. PhD thesis, McGill University (2002) `http://www.sablevm.org`.
25. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. 2nd edn. Sun Microsystems (1999)
26. Vallée-Rai, R.: Soot: A Java bytecode optimization framework. Master's thesis, McGill University (2000) `http://www.sable.mcgill.ca/soot/`.
27. Pickett, C.J.F., Verbrugge, C.: Compiler analyses for improved return value prediction. Technical Report SABLE-TR-2004-6, Sable Research Group, McGill University (2004)
28. Pominville, P., Qian, F., Vallée-Rai, R., Hendren, L., Verbrugge, C.: A framework for optimizing Java using attributes. In: CC. Volume 2027 of LNCS. (2001) 334–354
29. Shavit, N., Zemach, A.: Scalable concurrent priority queue algorithms. In: PODC. (1999) 113–122
30. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. TOCS **9** (1991) 21–65
31. Magnusson, P.S., Landin, A., Hagersten, E.: Queue locks on cache coherent multiprocessors. In: ISPP. (1994) 165–171
32. Pickett, C.J.F., Verbrugge, C.: Return value prediction in a Java virtual machine. In: VPW2. (2004) 40–47
33. Renau, J., Tuck, J., Liu, W., Ceze, L., Strauss, K., Torrellas, J.: Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation. In: ICS. (2005)
34. Martínez, J.F., Torrellas, J.: Speculative locks for concurrent execution of critical sections in shared-memory multiprocessors. In: WMPI. (2001)
35. Rajwar, R., Goodman, J.R.: Speculative lock elision: Enabling highly concurrent multithreaded execution. In: MICRO. (2001) 294–305
36. Kim, S.W., Ooi, C.L., Eigenmann, R., Falsafi, B., Vijaykumar, T.N.: Reference idempotency analysis: A framework for optimizing speculative execution. In: PPoPP. (2001) 2–11
37. Lea, D.: The JSR-133 cookbook for compiler writers. `http://gee.cs.oswego.edu/dl/jmm/cookbook.html` (2005)
38. Pugh, B.: A problematical case for finalizers. `http://www.cs.umd.edu/~pugh/java/memoryModel/archive/1276.html` (2003)
39. Krishnan, V., Torrellas, J.: A direct-execution framework for fast and accurate simulation of superscalar processors. In: PACT. (1998) 286–293