Speculative Multithreading in a Java Virtual Machine

Christopher J.F. Pickett and Clark Verbrugge
{cpicke,clump}@sable.mcgill.ca

March 25th, 2005

Abstract

Speculative multithreading (SpMT) is a dynamic program parallelisation technique that promises dramatic speedup of irregular, pointer-based programs as well as numerical, loop-based programs. We present the design and implementation of software-only SpMT for Java at the virtual machine level. We take the full Java language into account and we are able to run and analyse real world benchmarks in reasonable execution times on commodity multiprocessor hardware. We provide an experimental analysis of benchmark behaviour, uncovered parallelism, the impact of return value prediction, processor scalability, and a breakdown of overhead costs.

1   Introduction

Although languages such as Java provide (and continue to improve [33]) high-level language/API support for explicit parallelism, taking advantage of hardware concurrency remains a difficult task. This situation is being exacerbated by the increasing presence of multiprocessor and multicore machines as consumer level, commodity hardware [24]. In order to use that hardware effectively techniques need to be developed that place less burden on the application programmer. Purely automatic techniques for parallelism are thus highly desirable, but good, general performance has so far proved elusive [44, 46, 74].

Speculative Multithreading (SpMT) is a relatively new, automatic technique for medium to fine-grained automatic parallelisation that applies to a wide variety of programs, including irregular and non-numeric programs. It is typically defined at the hardware level [18, 69, 64], though some software approaches have been investigated [4, 30, 13]. SpMT has shown quite good potential speedups in simulation studies; it does not always achieve optimal speedup, but as a tradeoff between implementation complexity and use of available hardware resources it shows excellent promise as a general, automatic parallelisation strategy.

Here we describe the design and implementation of SableSpMT, an SpMT-based system for automatic parallelisation of Java programs. We have incorporated SpMT at the (Java) virtual machine level, as opposed to the hardware or pure source level. This allows us to take advantage of high level VM knowledge of Java programs, while still having low level control over execution behaviour. Our implementation is the first full, usable implementation of SpMT for a VM and the first to be able to accommodate the complete Java language semantics. This makes our system an ideal environment for detailed, realistic investigation of SpMT performance, both as a (virtual) hardware simulation, and as a strategy for Java optimisation. We provide a detailed analysis of benchmark performance as well as a breakdown of overhead costs under realistic workloads.

Most SpMT strategies focus on loop-level optimisations. For object-oriented languages, however, the extensive use of method calls and polymorphism make simple, easy to analyse loops less prevalent. Java itself is also not a generally favoured language for intensive numeric computing applications, and so irregular programs with complex control structure are quite common. Our approach to SpMT is optimized for this environment. We base our main design on *Speculative Method-Level Parallelism* (SMLP), a form of SpMT that uses method invocations as potential spawning points for parallel execution. To further exploit frequent method invocations we incorporate a sophisticated, hybrid *return value prediction* (RVP) system [51]; this helps to reduce inter-thread dependences, and has been shown to be a valuable optimisation for SMLP [25]. Other optimisations specific to Java program behaviour within our implementation context are also applied.

Our experiences with real world Java benchmarks show that SpMT is feasible in the VM environment. Our experimental system is able to run standard industry and academic Java benchmarks at reasonable speeds, and can be used on existing multiprocessor hardware. Overhead cost and processor utilization are

1

the major concerns in SpMT design, particularly for software based approaches. Our design includes a number of internal optimisations to help in both respects. Still, there is significant room for further overhead optimisation, and exploration of variant designs; at this time, we cannot claim to have achieved actual speedup. The data we provide, as well as our software environment itself, are intended to facilitate research experimentation.

1.1 Contributions

Specific contributions of our work include:

1. We present the first actual implementation of SpMT within a non-trivial (Java) virtual machine, SableSpMT. Our design provides a complete, automatic method-level speculation environment for Java programs, and incorporates an optimized return value prediction system as well as a variety of more specific implementation optimisations. Our design gives full consideration to Java language features, including synchronization, exceptions, native methods, and GC.

2. Using our system, we have provided a detailed analysis of speculative performance on the SPEC JVM98 benchmark suite [63] with realistic input settings (ie the recommended size 100). Other analyses of Java and SpMT have used simulated hardware systems with either greatly reduced, and therefore not especially realistic inputs (e.g., [25] uses SPEC size 1, which mostly consists of startup and test harness activity) or have simulated only a limited subset of VM behaviour [71].

3. Our framework is designed to simplify research investigation of this problem. We have implemented our system in a Java interpreter for simplicity of investigation and ease of modification. We use SableVM, a non-trivial, standards-compliant, highly portable and open source research virtual machine [20]. As well as true speculative multithreaded execution we also include a "single-threaded" mode that safely mimics the behaviour of the speculative system without the complexity of actually producing concurrent threads. This deterministic mode allows for easy debugging and experimentation on SpMT strategies, as well as analyses of program properties and reaction to speculative operations.

4. We give a complete, experimental breakdown of overhead costs involved in our speculative system. This data provides a good indication of how to further improve speculative designs for JVMs, as well as where to direct future optimisation strategies for achieving an industrial strength SpMT implementation at the VM level.

1.2 Roadmap

In the next section we give basic background on both speculative multithreading and the virtual machine environment we used in our study. Section 3 describes our complete implementation design, and Section 4 provides details on basic design optimisations that help make our approach feasible. In Section 5 we give experimental results on our implementation, including a breakdown and analysis of the various forms of overhead, and how it relates to our design choices. Section 6 describes related work in the area, and finally we give directions for future work and conclude in Sections 7 and 8 respectively.

2 Background

In speculative multithreading, a sequential region of code is split into two or more threads so that it may execute on multiple processors. Only the first thread, which executes the sequentially earliest code, is guaranteed to be correct, and is termed the *non-speculative* or parent thread. All other threads execute with the potential of computing incorrect values, and are termed *speculative* or child threads. Speculative threads are required to execute in a completely safe state that cannot affect overall program correctness, and may be aborted as soon as a violation is detected. Speculative threads may themselves create child threads, which will in turn be speculative.

To ensure correctness, all writes by a speculative thread get written to a thread-specific cache called a *dependence buffer*, and so cannot affect main memory. Similarly, all reads by a speculative thread are loaded into the dependence buffer. When execution in the non-speculative parent thread reaches the point where the child thread was forked, the child receives a signal to stop speculation and begin the join process. The buffered reads of the speculative thread are verified to not have changed with respect main memory, and if so the buffered writes are `committed` to main memory. If there was a *dependence violation* such that a value read by the speculative thread was later written by the non-speculative thread, the speculative thread aborts and the code is re-executed. Figure 1 depicts the general speculative multithreading execution model.
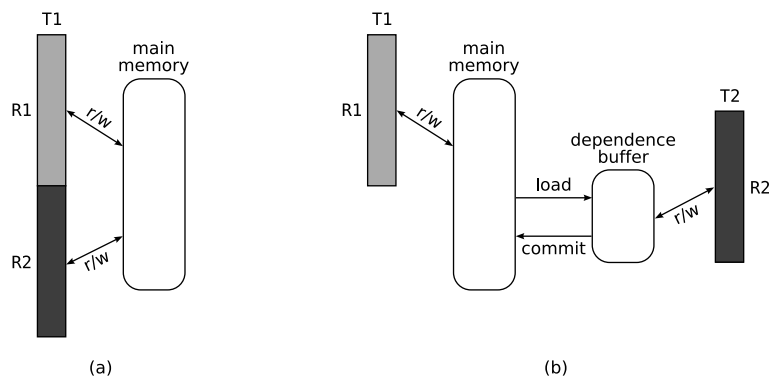
Figure 1: *General speculative multithreading execution model.* (a) Under a non-speculative execution model, thread `T1` executes sequentially, with region `R1` preceding `R2`, and all reads and writes access main memory directly. (b) Thread `T1` has been partitioned into two threads, with `R1` executing in `T1` and `R2` executing in `T2`. `T1` is the non-speculative parent thread whereas `T2` is the speculative child thread. All of `T2`'s reads and writes pass through a dependence buffer. When `T2` attempts to read a value for the first time, it fetches the value from main memory (or from a value predictor) before storing it into the dependence buffer. When `T1` is finished, `T2`'s results, stored in the dependence buffer, are checked against `T1`'s results, stored in main memory, and if there exist no violations, then `T2` commits its results to main memory. At this point, execution resumes non-speculatively at the end of `R2`. If there exist violations, then `T2`'s results are discarded and execution resumes non-speculatively at the beginning of `R2`.

The basic speculative execution model requires a dependence buffer, violation testing, thread abortion, and thread committal, but makes no demands as to how programs are partitioned into threads. Specifically, two issues that must be addressed when preparing sequential code for speculative execution are where to fork new speculative threads, and where to verify the speculative execution and commit the results to memory if correct. There are various strategies that have evolved with respect to thread creation and termination, and these can be categorized into four broad categories: loop-level, method-level, lock-level, and arbitrary.

In loop-level speculation (Figure 2), speculative execution of future loop iterations are started at the beginning of the current loop iteration in the parent thread. Granularity of the speculation can be controlled by forking speculative threads more than one iteration ahead of the parent thread, effectively unrolling the loop.
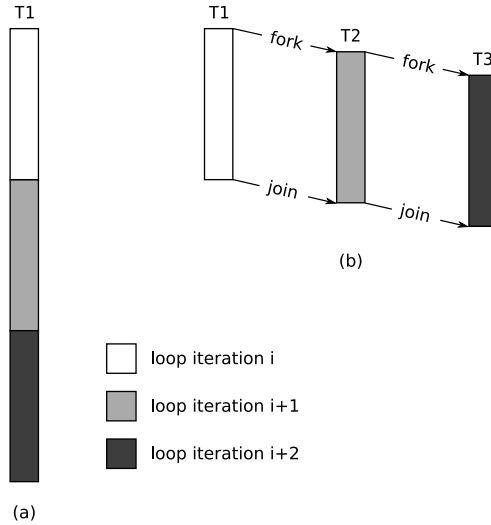
Figure 2: *Loop-level Speculation execution model.* (a) Under a sequential, non-speculative execution model, thread `T1` executes loop iterations i, $i + 1$, and $i + 2$ in order. (b) In loop-level speculation `T1`, at the beginning of iteration i, forks a new speculative thread `T2` to begin iteration $i + 1$ on a separate processor, which in turn forks `T3`, and so on until there are no more processors available. When thread `T1`, which is non-speculative, completes, all speculations in `T2` are validated and committed to memory if there are no violations. Otherwise, `T2` and `T3` are both invalidated, and `T2` restarts execution non-speculatively at the beginning of $i + 1$, immediately forking $i + 2$ in a new thread `T4`. This process continues until all loop iterations have completed, at which point speculation terminates in all child threads (who are executing loop iterations outside loop bounds) and the program enters a non-speculative, non-loop region.

Many programs have complicated control structures that may not always respond well to loop-based parallelization. Speculative Method-Level Parallelism (SMLP) focuses on method invocations as speculation points, with the non-speculative thread entering the method and the speculative thread starting from the method return point. Speculative threads are committed when the non-speculative thread returns from the call at which point, if there are no violations, execution continues from wherever the speculative thread has reached. Under this model only the youngest thread may fork a new speculative thread. Figure 3 depicts the general SMLP execution model.

Lock-level speculation, or speculative locking, allows for threads to enter and exit critical sections speculatively, and can be used to extract further parallelism from explicitly multithreaded programs and permit coarse-grained locking strategies. Arbitrary speculation subsumes loop-, method-, and lock-level strategies: speculation may occur at any point. This is of course maximally flexible, and can offer good rewards with sufficient analysis information [3]. It is however also complex to implement, and requires maximal preprocessing or runtime analysis overhead.

We have elected to use SMLP as the main design paradigm in SableSpMT. SMLP is expected to better accommodate Java's invocation dense program structure, as well as our intended domain of irregular, non-numeric applications, while still being a potentially feasible dynamic implementation.

2.1 VM Environment

Our implementation is integrated into the free / open source Java Virtual Machine, SableVM [20]. This is a complete implementation of the Java specifications that is available for several GNU/Linux distributions, and is to designed to be useful for experimentation and investigation of virtual machine characteristics,

4
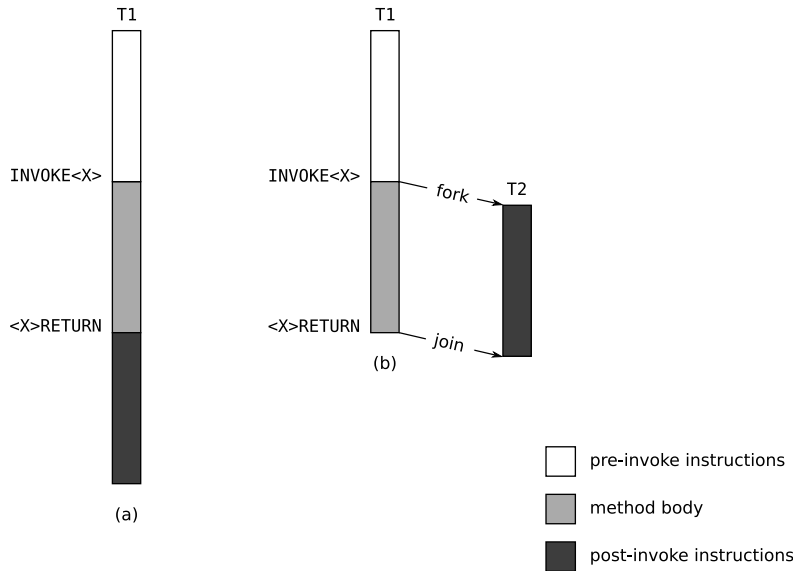
Figure 3: *Speculative Method-Level Parallelism execution model.* (a) Under a sequential, non-speculative execution model, thread `T1` encounters a call to method `foo()`, and executes the method body before returning to the next statement after the call. (b) Under SMLP, the same thread `T1` now forks a new speculative thread `T2` to continue past the return point when it encounters the call to `foo()`. When `foo()` returns, all flow dependences between `T1` and `T2` are checked, and if there are no violations, `T2` commits its results to memory.

highly portable to new architectures, and efficient, small, and robust enough for use by end users. SableVM has several optimized modes of interpretation, but at the time of our efforts did not contain a Just-in-Time (JIT) compiler. This simplifies the implementation complexity for experimental research in SpMT designs, although as we discuss in Section 7, research in a JIT context would also be useful.

Execution in SableVM is through a basic (or optimized) interpreter loop. Input Java bytecode is first *prepared*, or converted to an internal code array and accompanying internal structures. Executing code interacts with the usual runtime services, including a semi-space copying garbage collector, native thread support, class loading, exception handling, etc. The complete language and Java API is supported and SableVM is capable or running Eclipse [26] and other large, complex Java programs.

3 Java SpMT Design

In the following subsections we describe the main VM structures that are affected by SpMT and how our modifications address speculative requirements and ensure safety. This basic design also includes consideration of distinguishing features of the Java language in Section 3.7, and a description of our single-threaded analysis mode in Section 3.8.

The core design includes a variety of modifications to SableVM's internal structures. Most of these changes would be required in any VM environment, if at a somewhat more abstract level; our intention is to provide a description generic but precise enough to enable direct translation of these ideas into other Java runtimes.
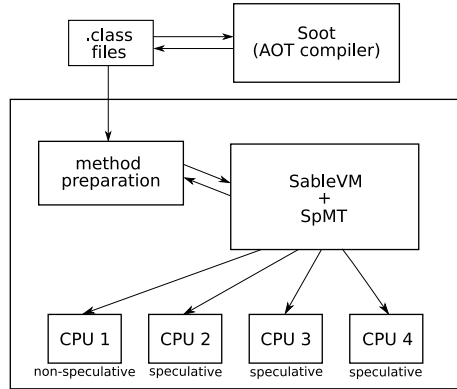
Figure 4: *The SableSpMT speculative multithreading execution environment.* SableVM prepares methods at runtime from dynamically loaded classes, which are read in from Java .class files. Soot is used to transform and attach attributes to these classes in an ahead-of-time step, although this could also occur at runtime. Method preparation is enhanced to support speculative multithreading, and the SpMT engine uses specially prepared method bodies to split single-threaded tasks across multiple processors on an SMP machine.

3.1 Execution Environment

An overview of the SableSpMT execution environment is given in Figure 4. The SableVM switch interpreter has been modified to use specially prepared speculative versions of Java methods, which are dynamic code arrays prepared alongside normal non-speculative versions. Methods are defined in Java .class files, and part of the extra information required for speculation may be encoded in these files, although there is no fundamental need for an ahead-of-time analysis in our design. The speculation engine forks threads at runtime, and these execute Java in an out-of-order fashion using the speculative versions of methods. Speculative threads run on separate processors, and there are a maximum of $s = p - n$ speculative threads running at once, where p is the number of processors and n is the number of non-sleeping non-speculative Java threads.

Any common SMP machine with a POSIX environment available is sufficient for speculative multithreading with SableSpMT. SableVM ports exist for 13 different architectures, and ports of the SpMT engine to these architectures should be relatively straightforward; SableSpMT currently runs on SMP `x86` and `x86_64` architectures. Since the implementation is completely POSIX-compliant and written in ANSI C, most of the porting complexity derives from defining the right atomic operations in assembly language.

3.2 Method Preparation

There are numerous steps involved in preparing a method for speculative execution. We implement an additional pass over the code array after normal Java methods have been prepared in SableVM, and augment several of the other passes. SableVM uses word-sized (32-bit or 64-bit) versions of the 8-bit Java instructions for the sake of efficiency [19], and so we can add as many additional instructions as necessary. Additional instructions are not actually required, but they do reduce execution overhead and simplify the design.

It is worth noting that in preparation of the speculative method code array the majority of Java's 200 odd instructions can be used verbatim, and only those that have the potential to violate program correctness need special attention.

6

3.2.1 Fork and Join Instructions

The first critical step is the insertion of new `SPMT_FORK` and `SPMT_JOIN` instructions into the code array. In speculative method-level parallelism we want to fork threads at method callsites and join them upon return from the method, and we need some way to instruct the VM to do this. This functionality could also be triggered by the `INVOKE<X>` and `<X>RETURN` instructions, but the use of specific speculative instructions provides a clean conceptual break, and can be more easily extended to support other speculation strategies.

Code is processed ahead of time for simplicity in this effort. Following a technique also employed by JikesRVM [27] to obtain new functionality from Java instructions, we insert calls to dummy static void `Spmt.fork()` and `Spmt.join()` methods around every single normal Java `INVOKE<X>` instruction using the Soot [70] compiler analysis framework as a bytecode instrumentation tool. These empty methods have zero side effects, add minimal space overhead, and can be trivially inlined in the case of non-speculative execution. SableSpMT, however, is engineered to recognize and replace them with the appropriate `SPMT_FORK` and `SPMT_JOIN` instructions during method preparation.

Note that this approach, although simplistic, is relatively inexpensive and provides the flexibility to use ahead-of-time analyses to determine good fork points: fork and join instructions can simply be omitted around undesirable fork points. Further use of compiler analysis information is part of our future work.

3.2.2 Modified Bytecode Instructions

The `SPMT_FORK` and `SPMT_JOIN` methods surrounding every `INVOKE<X>` are present in both the non-speculative and speculative versions of method bodies. Many other Java instructions require special speculative versions, and these are listed in Table 1. Through insertion of these replacement instructions into a separate code array, speculative execution becomes almost completely invisible to normal Java threads, with the necessary exception of fork and join instructions. Branch instructions with trivial fixups are not shown.

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 calls to a variable dependence buffer, as described in Section 3.5. Note that the buffer only cares about the addresses and widths of the data it holds in memory and not their location or functionality, and so we can use the same code to buffer reading from and writing to arrays on the heap, object instances on the heap, and class statics in class loader memory.

Numerous instructions may throw exceptions, and if this occurs, we stop speculation immediately. These include the `(I|L)(DIV|REM)` instructions that throw `ArithmeticException`'s upon attempting integer division by zero, and many others that throw `NullPointerException`'s, `ArrayIndexOut-OfBoundsException`'s, and `ClassCastException`'s. Full details on the exceptions that each Java bytecode instructions may throw are given in the JVM Specification [37]. In addition to these implicit instruction-specific exceptions, user or class library code may throw an explicit exception through use of the `ATHROW` instruction, and again here we stop speculation immediately. It is important to note that *stopping* speculation does not necessarily imply `aborting` a speculative child and thereby failing the join process. There is actually somewhat more subtlety to Java exceptions and SpMT than simple termination of speculative thread execution, and they are discussed further in Section 3.7.3.

The `INSTANCEOF` instruction requires computing type assignability between a pre-specified class and an object reference on the stack, and is built on the assumption that the reference in question is actually an object instance. This can normally be asserted before execution by a Java bytecode verifier, but we are

7

instruction	reads global	writes global	locks object	unlocks object	allocates object	throws exception	enters native code	loads classes	forces stop
GETFIELD	always					sometimes		first time	sometimes
GETSTATIC	always							first time	first time
<X>ALOAD	always					sometimes			sometimes
PUTFIELD		always				sometimes		first time	sometimes
PUTSTATIC		always						first time	first time
<X>ASTORE		always				sometimes			sometimes
(I|L)(DIV|REM)						sometimes			sometimes
ARRAYLENGTH						sometimes			sometimes
CHECKCAST						sometimes		first time	sometimes
ATHROW						always			always
INSTANCEOF								first time	sometimes
RET									sometimes
MONITORENTER	always	always	always			sometimes			always
MONITOREXIT	always	always		always		sometimes			always
INVOKE<X>	sometimes	sometimes	sometimes			sometimes	sometimes	first time	sometimes
<X>RETURN	sometimes	sometimes		sometimes		sometimes	sometimes	first time	sometimes
NEW		always			always	sometimes		first time	sometimes
NEWARRAY		always			always	sometimes			sometimes
ANEWARRAY		always			always	sometimes		first time	sometimes
MULTIANEWARRAY		always			always	sometimes		first time	sometimes
LDC_STRING					first time				first time

Table 1: *Java instructions modified to support speculation.* Each instruction is marked according to its behaviours that require special attention during speculative execution. These behaviours are marked "always", "sometimes", or "first time" according to whether or not their execution is conditional within the instruction. "Forces stop" indicates if the instruction may force termination of a speculative thread, but does not have a strict correlation with abortion and failure of the speculative sequence. Not shown are branch instructions; these are trivially fixed to support jumping to the right pc.

executing this code unsafely and out of order, and so there is nothing dictating that a valid reference will always be on the stack; this means that we may need to stop execution if the reference is invalid. Detecting whether an object reference is valid requires either a magic word in the object header or a bitmap defining the positions of objects on the heap; we currently use a magic word but will move to a bitmap-based solution in the future, as this kind of information may be useful for other VM research as well, memory management in particular.

The JSR (jump to subroutine) instruction is always safe to execute because the target address is hardcoded into the code array, but the return address used by its partner RET is read from a local variable. In speculative execution we always assume the worst, and therefore must check the validity of the target address, because we cannot make assertions about bytecode verifiability. Furthermore, even if the address does turn out to be valid, it may be the case that the JSR was reached non-speculatively but speculation started before the RET was encountered, and thus the return address in the local might point to the wrong code array; in this case we simply look at the starting positions of the two arrays to determine where the destination address lies and then fix it up if necessary. We actually need a modified RET instruction in the *non-speculative* code array as well, in the event that speculation stops inside a subroutine and subsequently gets committed, which would lead to non-speculative code having a speculative return address in one of its local variables.

The INVOKE<X> and <X>RETURN instructions may lock and unlock object monitors, and MONITOR-

(ENTER | EXIT) will always lock or unlock object monitors. We never allow this to occur speculatively, and so the speculative execution engine needs to be protected from any code that would normally permit synchronization operations. In addition, we mark these instructions in Table 1 as reading from and writing to global variables, as lockwords are stored in the header of object instances on the Java heap. Although this information about heap access is not particularly useful in the absence of speculative locking, we present it for the sake of completeness. Further details on synchronization are given in Section 3.7.4.

The INVOKE<X> instructions are also modified to ensure that speculative threads will never enter an unprepared method and start SableVM's lazy method preparation process. Furthermore, for polymorphic callsites, i.e. the INVOKE(VIRTUAL | INTERFACE) instructions, we check that the receiver is a valid object instance and that the speculative target found through its virtual table pointer has the right net effect on the stack and that the type of its containing class is assignable to the receiver's type. Although we could use strcmp() or some hashing tricks to verify target signatures, entering the wrong method occurs so rarely that it does not justify the extra overhead. It is worth noting that our experience shows speculative code *will* invariably enter the wrong method and on occasion get trapped inside infinite loops, even for the simplest of benchmarks such as check, and so a limit on the number of speculative instructions executed per thread is needed to prevent this, as described in Section 3.8.

<X>RETURN instructions, in addition to the synchronization check, require two things: 1) buffering of the non-speculative stack frame from the parent thread (more on stacks and our lazy stack buffering strategy in Section 3.4), and 2) verifying that the caller method is not in the middle of a *preparation sequence*, a special type of instruction sequence used in SableVM to prepare and replace instructions that have slow and fast versions [19].

Returns must also be monitored to ensure the speculative thread does not leave bytecode execution entirely. A return in SableVM can also lead to thread death, VM death, or a return to native code, all of which are unsafe for speculative execution. These situations are trapped and force speculation to end. Similarly, invokes are monitored to ensure native code is not entered.
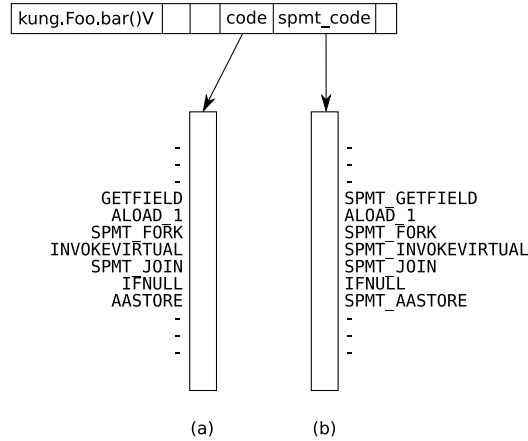
Finally, we consider the object allocation instructions, (MULTI)(A)NEW(ARRAY). Barring conditions that would lead to an exception being thrown or triggering garbage collection, there is nothing particular to speculative multithreading that prohibits object allocation on the global heap. We discuss allocation and garbage collection in greater detail in Section 3.7.2. Note that the LDC_STRING specialisation of the LDC instruction allocates a constant object upon being reached for the first time, later having the address of this object patched directly into the code array, and so we always stop and allow the first execution to occur non-speculatively.

To the best of our knowledge Table 1 is comprehensive, and documents all pitfalls that are inherent in attempting to execute Java bytecode speculatively. We welcome any clarifications on corner cases or our attention being drawn to omissions that we have made; so far, these modifications have been enough to support speculative execution of all benchmarks we have tried and are consistent with our understanding of the JVM Specification [37].

3.2.3   Parallel Instruction Code Arrays

As discussed, the goal of this extensive instruction substitution is to prepare a parallel code array for each method that speculative threads can use without breaking sequential execution semantics.

This code array structure is shown in Figure 5. The only instructions that get special SpMT versions are those discussed in Section 3.2.2. SPMT_FORK and SPMT_JOIN instructions are present in *both* arrays, and

9

```
kung.Foo.bar()V      code spmt_code
```

```
     -                    -
     -                    -
     -                    -
  GETFIELD            SPMT_GETFIELD
  ALOAD_1             ALOAD_1
  SPMT_FORK           SPMT_FORK
  INVOKEVIRTUAL       SPMT_INVOKEVIRTUAL
  SPMT_JOIN           SPMT_JOIN
  IFNULL              IFNULL
  AASTORE             SPMT_AASTORE
     -                    -
     -                    -
     -                    -

     (a)                  (b)
```

Figure 5: *Parallel code arrays.* (a) non-speculative code array prepared for method bar(); (b) speculative version of the same code array with modified instructions.

enable both non-speculative and speculative threads to create and join children. In the following sections, we examine the use of these two code arrays in speculative method level parallelism.

3.3 Forking Speculative Threads

When a non-speculative parent thread reaches an SPMT_FORK instruction, it will attempt to fork a new speculative child thread. The first step is to decide whether or not the current fork point is suitable for starting speculative children. Rather than simply make an on/off binary decision, we also attempted to build a good heuristic that assigns fork priorities depending on several pieces of dynamic and static data, including:

1. A static upper bound on the size of code reached through the method, which can be obtained from a callgraph if there are no intraprocedural or interprocedural backward branches over the transitive closure of the method call. We implemented such an analysis in Soot [70] using the Jimple intermediate representation of bytecode and the callgraph derived from the points-to analysis provided by Spark [34].

2. Dynamic upper and lower bounds on transitive method size, and more importantly an average size. This information is kept on either a per-target or per-callsite basis.

3. A history of speculation successes and failures per-callsite, kept over either the entire execution of the application to date or over the last N speculative forks.

4. A history of the lengths of speculative sequences per-callsite, again kept since application startup or over the last N forks. We might also be interested in a static analysis to avoid sequences that are guaranteed to have a short maximum length due to an unavoidable instruction that forces speculation to stop, such as a MONITORENTER or an ATHROW within 10 instructions of the fork point.

5. The number of times a speculative child was forked but joined again by the parent before actually executing any instructions; this represents 100% wasted fork/join overhead.

6. The number of times a speculative child was forced to stop prematurely because it encountered another speculative child that had been forked at this callsite; this represents extra fork/join overhead inherent in a scheme with fork points that are too frequent.

When the decision to fork a child has been made, several steps must be taken. Primarily these involve memory allocation, and an SpMT-specific memory manager is most helpful in reducing overhead here. An enumeration of all steps required follows:

1. The thread environment of the parent is copied over to the child, although only those variables in this structure which can be touched speculatively need to be safe.

2. The current parent stack frame is copied to the child; details of our stack buffering strategy are given in Section 3.4.

3. A dependence buffer is initialized, this will act to protect main memory from speculative execution and allow for child verification at join time. Dependence buffering is described in Section 3.5.

4. The height of the Java operand stack upon return from the non-speculative invoke is computed; this is a constant for any given callsite and it is not necessary to parse method signatures on each invoke.

5. (optional) A return value may be predicted for non-void methods. This is not strictly required for speculation, as any random value can be used. Thus we describe our return value prediction as an optimisation in Section 4.1.

Then, depending on the execution model, we either skip over the invoke and switch immediately to the speculative code array (single-threaded simulation, Section 3.8.1) or enqueue the child on a global priority queue and proceed with non-speculative execution (true speculative multithreading, Section 3.8.2. It is better if the enqueuing takes place as soon as possible, to reduce the time lost to speculative forking in the non-speculative parent thread, and in our current system only step 1 above is completed by the parent.

3.4   Speculation and the Java Stack

During execution of a Java method, the VM will interpret bytecode instructions and use them to modify the Java *operand* stack; Java bytecode provides the basis for a stack machine at runtime. At the same time, each method gets its own frame on the Java *call* stack. In SableVM, the operand stacks and call stacks are interleaved; the arguments to a callee method at the callsite in the caller frame become the callee's parameter local variables when its frame is pushed.

Speculatively, we need to buffer all stack accesses, to protect the parent non-speculative thread in the event of failure. The simplest way to do this is to copy the entire parent stack over to the child. A lazy strategy is more efficient, however, and we instead copy stack frames from the parent only as the child exits them. We now present a moderately complex example of a parent thread with two speculative children to illustrate the details and safety mechanisms of our stack buffering strategy.

In Figure 6 a non-speculative parent thread and its two speculative children are shown. Each thread moves up and down the Java call stack as it enters and exits methods. Speculative execution proceeds as follows. The parent thread pushes frames *f1* through *f4* on the stack. Just before the callsite that will push *f5*, it decides to fork a child. After forking the parent executes methods at levels *f5* and *f6*. The child gets the caller frame copied over and continues execution past the invoke. It exits *f4* and so needs to copy over *f3*
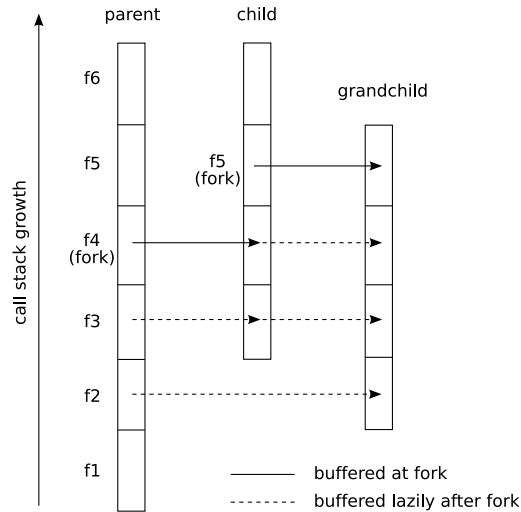
Figure 6: *Call stack buffering. f1* through *f6* are stack frames corresponding to Java methods. A speculative child is forked at *f4* in the parent, and in turn a second-generation grandchild thread is forked at *f5* in the child.

from the parent. This is always safe, because the parent will join the child at *f4* before either returning to *f3* in the event of a failure, or jumping ahead to the child's current position in the event of success.
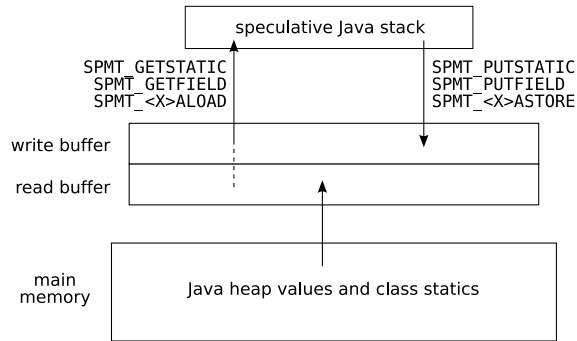
Sometime in *f3*, the child encounters another invoke and moves back up to the *f4* level. This may be a completely different method from that in which the child was forked. No buffering is needed, as this is new speculative execution without immediate dependence on prior computation. Similarly, when the child enters *f5*, it does not copy the frame over from the parent. In *f5*, before entering *f6*, the child decides to fork its *own* child; this is the grandchild of the original non-speculative parent. Before the child return from *f6* to *f5*, the grandchild exits several methods and ends up down at *f2*. Again it is safe to copy *f4* and *f3* from the child as the child will stop when it gets to *f5*, where it sees the grandchild has been forked.

Furthermore, it is safe for the grandchild to buffer *f2* from the *parent*: the parent can only return to *f2* if it aborts the initial child forked in *f4*, and this will lead recursively to the abortion of the grandchild (see Section 3.6 on joining); otherwise, it will commit the child, copy over all stack frames entered speculative and jump ahead, not returning below *f5* before joining the grandchild. By ensuring that a given thread never leaves a stack frame without joining its child, we can promise that speculative frames copied from ancestors will never be corrupted.

A novel concept in this design is that it allows for multiple *immediate* children in a parent non-speculative thread: at each method entered we allow a child to be forked, and as we associate children with stack frames, this means that there can be one child per frame on the stack. This exposes significantly more parallelism than a model in which each thread is restricted to a single immediate descendant. Multiple immediate children per thread are *not* shown in Figure 6, but the result of the extension is a tree of children rather than a chain.

3.5   Dependence Buffering

Common to all speculative multithreading proposals is a mechanism for buffering potentially unsafe memory reads and writes in speculative threads. We already buffer stack frames and modification of local variables as described in the previous section, and so now we need only worry about accesses to main memory. In

12

```
                        ┌─────────────────────────┐
                        │  speculative Java stack  │
                        └─────────────────────────┘
           SPMT_GETSTATIC    ▲             │    SPMT_PUTSTATIC
           SPMT_GETFIELD     │             │    SPMT_PUTFIELD
           SPMT_<X>ALOAD     │             ▼    SPMT_<X>ASTORE
                        ┌─────────────────────────┐
    write buffer        │             ▼            │
                        ├─────────────────────────┤
    read buffer         │        ▲                 │
                        └─────────────────────────┘
                                    │
                                    │
    main                ┌─────────────────────────┐
    memory              │ Java heap values and class statics │
                        └─────────────────────────┘
```

Figure 7: *Dependence buffering.* Read and write buffers are allocated in a layered structure and allow for speculative threads to communicate with main memory. The dashed line indicates that the read buffer is searched during speculative heap/static loads only if the value has not already been written into the write buffer by a speculative heap/static store.

Java, main memory consists of data on the garbage-collected heap and static variables.

Each speculative thread gets its own dependence buffer. This is initialized at fork time, so that when a thread starts up, all of the buffer entries are empty. Furthermore, by setting an "uninitialized" flag, we can defer the real initialization until the buffer is actually used, thus eliminating overhead in the event that it remains unused. We can still further break down overhead costs by having per-type buffers internally, one for each of the 8 Java primitive types and a 9th for object references.

Each per-type buffer in turn consists of two sub-buffer layers, a read buffer and a write buffer. These are implemented as hashtables with a backing array acting as a linked list; the nearest equivalent in the Java Collections Framework is `java.util.LinkedHashMap`. The hashtable uses direct addressing and provides fast entry lookup, and the backing list allows for fast iteration through non-NULL elements. Value addresses are the keys in these buffers, and values themselves are stored as mappings to these keys. Dependence buffers are typically implemented as fixed-size table structures in hardware SpMT designs, and we have followed the same conventions here.

A model of a dependence buffer is pictured in Figure 7. When the speculative `SPMT_(GET(STATIC|-FIELD)|<X>ALOAD)` load instructions are 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 the speculative `SPMT_(PUT(STATIC|-FIELD)|<X>ASTORE)` instructions are executed, the write buffer is searched, and if no entry is found a new mapping is created. The default buffer size is 128 entries per primitive type, and we find this is more than sufficient, with overflow occurring rarely. It is likely that some small memory savings can be achieved here; however, this is not a pressing concern, and we can afford to be much more liberal with memory in a software implementation of SpMT.

3.6 Joining Speculative Threads

Upon reaching some termination condition, a speculative thread will stop execution and leave its entire state ready for joining by a parent thread. This termination condition may be: A) some pre-defined speculative sequence length limit being reached; B) a parent thread signalling the child to stop when it reaches the `SPMT_JOIN` instruction; C) the parent signalling the child to stop when it reaches the top of the VM exception handler loop in that stack frame (refer to Section 3.7.3); or D) the child reaching an instruction that

presents speculatively impermissable behaviour as discussed in Section 3.2.2.

The join process is straightforward. The parent thread returns from a method call and finds a speculative child on the stack. Either way the parent will ensure that the child has stopped execution before proceeding. If in the VM exception handler loop, the child is simply aborted and its memory is freed to a free list; execution continues non-speculatively until the exception is either caught or leads to VM death. Otherwise, the child may be successfully joined, and the following validation process takes place:

1. If the method returned a value, that value is checked for safety against the value used by the speculative thread. This value may have been predicted, or it may have been random. If unsafe, the child is aborted.

2. The child is checked for having seen the same number of garbage collections as the parent (refer to Section 3.7.2 for more about GC). If fewer, the child is aborted.

3. The status of dependence buffers is checked, and if overflown or otherwise corrupted, the child is aborted.

4. All values in the read dependence buffer are compared with the value at the same address in main memory; if violations occur, the child is once again aborted.

At this time, if the child has not been aborted, all values in the write buffer are flushed to main memory, the buffered stack frames entered by the child are copied to the parent, and non-speculative execution resumes at the `pc` and operand `stack_size` where the child left off. If aborted, the child's memory is freed and execution continues non-speculatively at the first instruction past the `SPMT_JOIN`.

3.7   Intricacies of the Java Language

Several traps await the unsuspecting implementor when trying to enhance a JVM to support speculative multithreading. We assert that it is necessary for an SpMT implementation in Java to consider these features in order for it to be considered fully general.

The features to be covered in the next four sections are native methods, garbage collection, exceptions, and synchronization. The common patterns that will emerge from them are that:

1. They are unsafe to execute speculatively in our current model. However, this does not entirely preclude future work to support them.

2. They *are* able to execute in a parent non-speculative thread that has speculative children running concurrently.

3. They form *speculation barriers* across which speculation cannot occur.

4. They do not necessarily force abortion of *all* speculative children when they occur, and their impact can be minimized.

3.7.1   Native Methods

The Java specification has provisions for native methods, which are methods not implemented in Java and executed as bytecode but implemented in a platform-specific language like C that gets compiled to machine code, and interacts with Java through a native interface such as the Java Native Interface (JNI) [36] or directly as part of the VM internals. Native code can thus be found as part of the class libraries, the JVM itself, or user code. Native methods are fundamental to Java execution, but in general are nicely hidden away from your typical programmer. For example, all thread management, timing, and I/O operations require native methods, as the functions they require are simply outside the scope of Java bytecode, but programmers are provided with pure Java wrappers around them.

In speculative execution, we need explicit control over the code being executed, and since this is not available for native methods they form a hard speculation barrier. However, we must still be able to execute native code non-speculatively, and it is safe to do so while speculative threads are alive. In terms of our model, we can fork and join children at callsites that have native targets, and more broadly we can do so at any call over which the transitive closure contains a native method.

As an example of where speculation is still useful for timing-dependent execution, consider mp3 playback. It should be possible to structure the application such that all I/O occurs in order and at the right time, but that decoding and processing of the mp3 can occur speculatively, such that the playback buffer suffers fewer underruns for a given computation load. Depending on the application, this might be achievable without any transformations.

3.7.2   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 proposal to extend it should consider this feature; indeed, many Java programs will simply run out of memory without GC.

SableVM uses a simple copying collector [19], and so object references get invalidated upon every collection. This means that a speculative thread started before GC will be invalidated after GC. However, since the threads are speculative, they must be invisible to the rest of the VM, and in particular cannot be traced during collection. All reads from and writes to the heap are buffered, and so having a speculative thread attempt to access the heap during GC is completely safe. Our solution to the problem of invalidating threads is to keep a count of the number of collections that any given non-speculative thread has seen. When a child is forked, this count is copied over. Upon returning to the fork point and joining the child, if the number of collections seen in the parent is greater than the count in the child, we force its failure.

The default copying collector in SableVM is invoked relatively infrequently, and we find that GC is responsible for a negligible amount of speculative invalidations. Other garbage collection strategies are assuredly more difficult to negotiate with, and it is likely the case that pinning of speculatively accessed objects is required to handle them properly. Needless to say, exploration of these ideas is outside the scope of this paper, but it would be entirely possible to pursue them within our framework.

As for object allocation, we *are* able to allocate objects speculatively, as discussed in Section 3.2.2. Speculative threads compete with non-speculative threads to acquire the global mutex protecting the heap, and any allocated objects are reachable starting only from the stack of the speculative thread. If allocation would trigger either GC or an `OutOfMemoryError`, we stop speculation immediately (and thus slightly modified routines for speculative object allocation are required). When a speculative thread gets joined and commit-

ted, child stack frames are copied to the parent and allocated objects become reachable non-speculatively. If on the other hand a child with speculatively allocated objects is invalidated, then the objects become completely unreachable and will be collected in the next collection.

The disadvantage of allocating objects speculatively is that more collector activity than normal will be necessary, as aborted children will pollute the heap with unreachable objects. However, we did not observe a large increase in GC counts when speculation was enabled, and furthermore providing this facility greatly extends the maximum length of speculative sequences, as objects are allocated often in many Java programs. Another advantage is that technically we do not need to buffer reads from and writes to speculative objects, but this is an optimisation not implemented at this time.

3.7.3 Exceptions

Java allows for bytecode instructions to throw exceptions, either implicitly as shown in Table 1 or explicitly through use of the `ATHROW` instruction.

Speculatively, we stop immediately upon encountering an exception. The rationale for this design decision is three-fold. First, exceptions are rarely encountered in normal execution – exception-heavy applications like *jack* throw an exception for less than 1% of all method calls [51]. Second, although it would be possible, writing a speculative exception handler is a somewhat tricky process and not the best use of our resources at this time. Third, and perhaps most importantly, exceptions that would occur speculatively are likely to be the result of incorrect computation and have a high correlation with sequence failure, meaning that attempting to process the exception speculatively would result in even more wasted cycles.

Non-speculatively, if an exception is thrown, caught, and handled within the same method without being rethrown, then there is no need to abort speculative children, and this process is completely invisible to them. However, exceptions may be thrown out of methods, and this leads to stack unwinding in the search for an appropriate handler, eventually ending in VM death if none are found. If in the process of popping stack frames we encounter a speculative child, we must signal that child to stop and force its immediate failure, to prevent memory leaks and unwanted computation. However, we do *not* need to invalidate all children on the stack in the event of an exception, only those that are encountered during stack unwinding.

Java compilers like `jikes` and `javac` compile idioms such as `try {} catch () {}` and `try {} catch () {} finally {}` in the Java language to use exception handlers with `JSR` and `RET` instructions [37]. The safety of these instructions with respect to speculative execution is discussed in Section 3.2.2. It is worth noting that other bytecode compilers are permitted to exploit them for other purposes, and our considerations about them with respect to speculative safety are not limited to exception handling.

There is a hidden complication with exception handlers and forking speculative children. One of the properties we would like to guarantee is that no stack frame ever has more than one speculative child, and this is checked with an assertion in our implementation of the `SPMT_FORK` instruction. If the `INVOKE<X>` following a fork were to throw an exception non-speculatively, we might end up in a bytecode exception handler with a speculative child on the stack. This would not be a problem if there were no callsites within the handler, but in fact frequently exception handlers do call methods, to report errors or otherwise act on the thrown exception object, and thus we also encounter fork and join instructions within the handlers themselves!

To avoid reaching a fork with a child on the stack, it is imperative that the frame have its child aborted at the top of the *VM* exception handler loop, i.e. before looking up and jumping to any particular bytecode exception handler pc. This technique also solves a problem with `finally` blocks: if an exception is

thrown within a `try` block and uncaught in the current frame, control will pass to the `finally` block before proceeding to the `catch` block [37]. Fortunately, the VM exception handler loop must still be entered before jumping to the `finally` block, and thus any children can easily be aborted.

3.7.4 Synchronization

The last feature of the Java language that requires treatment is the ability to synchronize threads and protect shared data using per-object monitors. In bytecode this can be accomplished in two ways, either explicitly through the `MONITORENTER` and `MONITOREXIT` instructions, or implicitly through a synchronized method entry or exit. As noted in Section 3.2.2, these operations cannot be executed speculatively.

Speculative locking, in which child threads *are* allowed to acquire and release monitors, has been previously explored by several groups [42, 43, 58, 55], and is discussed in greater detail in Section 6. It would be most interesting to evaluate these methods within our current system, and is a line of research we intend to pursue in the near future. In particular, the structured locking rules of the JVM specification ensure that the state of an object lock will be identical before and after synchronized method invocation on that object, which should simplify analysis and afford certain optimisations.

As for non-speculative locking and unlocking while speculative children are alive, it is safe, just as garbage collection, exception handling, and native method calls are safe. Perhaps the most interesting thing to note is that we can safely start speculative threads inside a critical section and join them before exiting the critical section. Thus code which is traditionally thought of as a parallelism bottleneck can be parallelised, in turn encouraging coarse-grained locking, which is desirable from a software engineering perspective for its easier programmability.
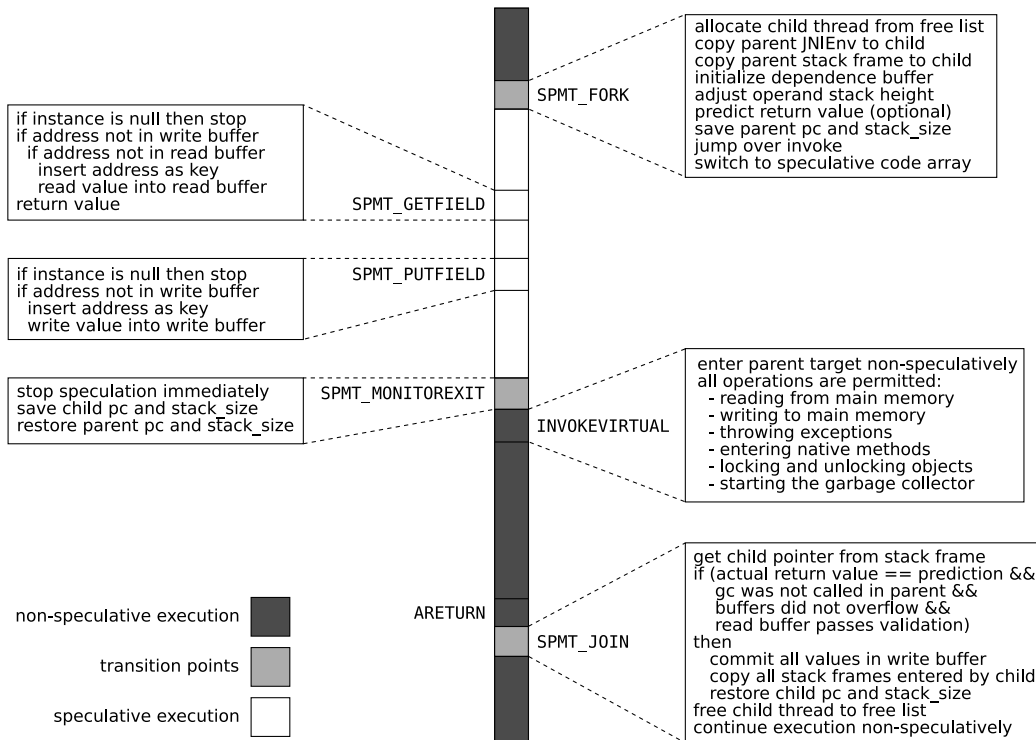
3.8 Execution Modes

Now that we have described in detail all the steps and data structures required for speculation in Java bytecode, we present two execution models. We developed our design hand-in-hand with our implementation; many of the previously described techniques were non-obvious, and we alternated between finding flaws in implementation that we had not previously considered and refining our design.

There were a couple of key factors that allowed us to progress this far. First was the clean and compliant implementation of the JVM Specification that is SableVM. At the expense of some speed optimisations, it is written in a way that is very easy to extend, and we did not suffer major research bottlenecks trying to understand the structure of the VM. Second was the use of a single-threaded *simulation* of speculative multithreading to work out the problems in our design (Section 3.8.1. This mode saved us from having to debug concurrency issues at the same time as general speculative safety issues, and also allowed for relatively easy experimentation on uniprocessors.

3.8.1 Single-threaded Simulation

In the single-threaded simulation mode, we interleave non-speculative and speculative execution of Java in a single thread. This is accomplished through state saving and restoration, and transitions between the non-speculative and speculative parallel code arrays.

The complete execution of a speculative child is shown in Figure 8. All of the important steps have been previously examined, and so for the most part this is review. The steps themselves are listed in point form

Figure 8: *Single-threaded simulation mode.* Java execution transitions to speculative execution at fork points, continues past the invoke until a termination condition is reached, jumps back to non-speculative execution to complete the body of the invoke, and attempts to the join the child when the join point is reached.

in the figure, and so here we simply draw attention to the features of the simulation.

When an SPMT_FORK instruction is reached, we prepare the speculative starting state and switch to the execution of that state. So that we can return to callsite to execute the INVOKE<X> non-speculatively, we save the non-speculative pc and operand stack_size, set the child state as the current environment, switch code arrays, and begin speculative execution immediately past the callsite.

After some speculative instructions, which may or may not involve the dependence buffer (in the figure, both a speculative heap read and a speculative heap write are shown), we reach some termination condition, be it a pre-defined sequence length limit or an instruction we cannot execute speculatively (in the figure, we reach an SPMT_MONITOREXIT and stop because exiting a critical section speculatively is currently unsupported).

At this termination condition, we now save the *speculative* pc and stack_size, restore the parent non-speculative pc and stack_size in the ordinary code array, and go back to execute the body of the INVOKE<X> non-speculatively. Upon return, we encounter the forked child on the stack, and begin the validation and committal process. If speculation succeeded, we jump ahead to where the child left off, at the equivalent position in the non-speculative code array, and otherwise abort the child and re-execute its body non-speculatively. In the event of a successful join in the figure, the next instruction to be executed would be a non-speculative MONITOREXIT.

Moving development from simulation mode to the true multithreading mode described in the next section was relatively straightforward. We hope this simpler mode will improve the development speed of and effort required for future implementation variations as well. Speculative coverage data obtained using this

18
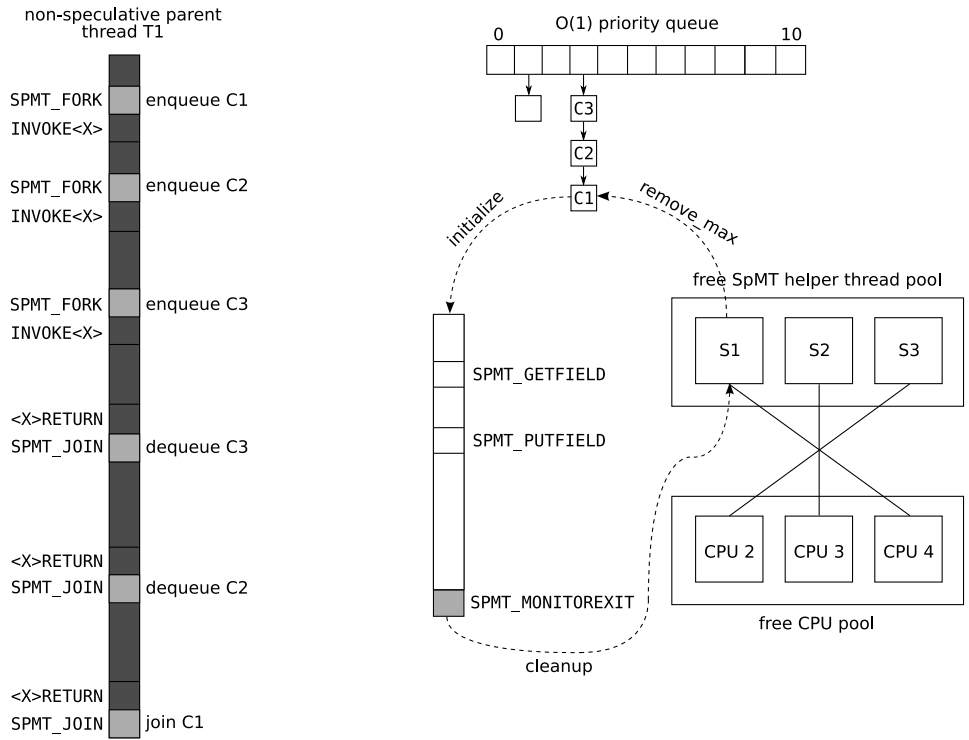
Figure 9: *Multithreaded mode.* Non-speculative parent threads enqueue children on an $O(1)$ priority queue. Speculative helper threads remove children from the queue, and then initialize and execute them, leaving them on the parent stack to be joined. If a parent thread reaches a child that it enqueued before that child has been started, it dequeues it and continues non-speculatively without attempting the speculative child join procedure.

simulation mode is presented in Section 5.3.

3.8.2 Multithreaded Mode

Multithreaded execution occurs much the same as in the single-threaded simulation, and we focus on the specifics of our threading implementation.

At a fork point, as mentioned in Section 3.3, a minimal amount of work is done non-speculatively to reduce the overhead of thread startup. As soon as the parent environment is copied to a child, we enqueue the child on an $O(1)$ priority queue, using a scoring heuristic with the components described in Section 3.3. This is a global priority queue that must be locked using an atomic operation before modification, however as we shall see in Section 5.1 it is likely not the most efficient implementation or design.

As shown in Figure 9, there is a pool of SpMT helper threads. These are POSIX `pthreads`, and currently there is one executing for every free CPU in the system. Other thread–CPU mappings are possible, indicated by the crossed lines in the figure, and determining the best mapping for a particular situation is a good direction to take in future work. These threads are waiting for speculative children to be enqueued, and as soon as one becomes available it will get picked up by a free helper and initialized, and then bytecode interpretation will begin. In the figure, child `C1` is picked up by helper thread `S1` and the same speculative as shown for the single-threaded simulation is executed, albeit in a separate thread on a separate processor.

Meanwhile, the parent thread reaches addttional fork points higher up on the call stack, and enqueues child

`C2` and `C3`. For whatever reason, these do not get picked up by helper threads `S2` and `S3` (assume for the sake of argument that `S2` and `S3` are busy wtth other children not pictured), and so the parent thread reaches them again before they have been started. This leads to `C3` and `C2` being dequeued, in that order, rather than stopped, validated, and etther commttted or aborted by the parent. Upon returning to the fork/join point of `C1`, the parent finds `S1` has left it on the stack already stopped at an `SPMT_MONITOREXIT`, and so the usual validation procedure takes place. If successful, the parent jumps ahead, and if not, the parent simply continues.

4   Optimisations

Given a basic SMLP implementation as described in which speculative threads are forked at every method call, there is ample room for optimisation. In this section we discuss what we consider to be two of the most important techniques for improving SMLP performance: return value prediction and efficient enqueuing algorithms.
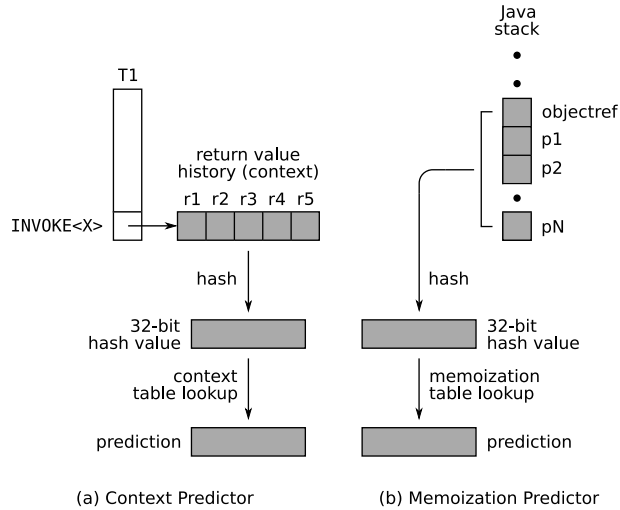
4.1   Return Value Prediction

One of the key optimisations that exists for SMLP is return value prediction (RVP). All non-void Java methods return values, and in a naïve speculation strategy, random values are pushed onto the operand stack at fork points to adjust the stack height correctly, and speculation is aborted if these values turn out to be incorrect.

Hu *et al.* previously showed that the SPEC JVM98 benchmarks benefit significantly from RVP in a simulation of 8-processor SMLP hardware [25]. Without any RVP, they were able to achieve an average speedup of 1.52 over the benchmark suite, with their best hybrid predictor a speedup of 1.92, and with perfect RVP a speedup of 2.76. On this basis we concluded that accurate return value prediction was highly desirable, and set out to achieve the highest prediction rates possible and to explore the relationship between table-based predictor memory requirements and predictor accuracy [51].

We implemented several well-known predictors from the literature in SableVM, including fixed-space last value and stride predictors, and a table-based context predictor. A last value predictor simply predicts the last value seen at a given callsite, and a stride predictor applies the difference (stride) between the last two returned values to the last return value to predict the next value. An order-*k* context predictor keeps a history of return values over the last *k* calls, and using hashtables to associate predictions with value histories; we implemented a standard order-5 context predictor. Performance of table-based prediction strategies naturally depends on table-size, up to a certain point, and at the performance limit of existing prediction strategies using per-callsite hashtable, we were able to achieve an average accuracy of 72% over SPEC JVM98.

We also introduced a new memoization-based predictor that uses method arguments as inputs to a hash function to predict values. This complemented existing prediction strategies nicely, and when included in a hybrid increased the average prediction rate over SPEC JVM98 from 72% to 81%. Models of context and memoization predictors are shown in Figure 10; they are functionally quite similar, the difference being that the context predictor gets its inputs from a history of return values, whereas the memoization predictor gets its inputs from method arguments.

After implementation of these return value predictors in SableVM, we looked to a couple of ahead-of-time compiler analyses for improved return value prediction in Soot [50], with the aim of increasing accuracy and reducing memory costs. The first is a slicing-based *parameter dependence* analysis that finds method

Figure 10: *Context and memoization predictor models. r1* through *r5* are the most through least recent return values, and *p1* through *pN* are method parameters.

parameters upon which the return value does not depend, and allows for reduction of inputs to the memoization predictor hash function. At runtime, this analysis finds 10% of all non-void calls return a return value that is not fully dependent on method arguments. The second is a *return value use* analysis that tracks the use of return values after a method call and determines if they are unconsumed, in which case the value does not need to be at all accurate, or if they are consumed but used only inside a boolean or branch expression, in which case accuracy constraints are relaxed. This analysis allows for an average 17% of return values at runtime to be substituted with partially or completely inaccurate predictions.

We now employ the best hybrid prediction strategy from our RVP implementation [51], and furthermore use classfile attributes to exploit the results of our compiler analyses [50] in SableSpMT. The impact of our return value prediction optimisations is examined in Section 5.3.

4.2 Efficient Speculation Decisions

As our experimental data on speculation overhead show (refer to Section 5.1), a good choice of fork points and efficient enqueing algorithms are critical to the success of SMLP and SpMT in general.

In Section 3.3, several relevant pieces of static and dynamic data associated with methods, callsites, and speculative sequences are described. The current per-callsite heuristic we use to assign priorities between 0 and 10 to threads for placement on the shared priority queue described in Section 3.8.2 is:

```
score = average_sequence_length * success_rate

        total_spmt_instructions   total_spmt_commits
      = ----------------------- * --------------------
          callsite_fork_count       callsite_fork_count

priority = (score > 1000) ? 10 : (score / 1000)
```

We find that this algorithm gives decent priority distributions, if somewhat biased towards lower priorities.

21

We also experimented with disabling speculation at callsites if 1) failure rates are too high; 2) average speculative sequence lengths are too short; or 3) if the callsite is reached often by another speculative child forked higher up on the stack. In all cases, we found that disabling speculation for short sequences had a strong correlation with increased speed, but we also found that the number of instructions executed speculatively dropped significantly. Thus we are hesitant to report data on the effect of disabling speculation, as it is unclear if the increase in performance is due primarily to reduced fork/join overhead, or to eliminating short and failing sequences. A full study that aims to reduce the overhead of speculative execution in our system without sacrificing parallelism is needed; here we aim to provide a complete design and initial baseline implemention.

Another component of time-efficient speculation is the actual enqueuing algorithm and associated data structures. Although our priority queue provides $O(1)$ `enqueue`, `dequeue`, and `remove_max` operations, it is globally synchronized and a sure source of speculation overhead. Greiner and Blelloch provide an in-depth analysis of queuing algorithms and an implementation strategy for speculative futures in an abstract functional language running on an abstract machine [21, 22]. Their results are probably not directly transferrable, as their model evaluates function arguments speculatively as opposed to ours which executes speculative continuations past method invocations. Nevertheless, the ideas and proofs given for time-efficient queuing algorithms and data structures in speculative execution are relevant to this work, and deserve consideration in future optimisation research.

On the other hand, Shavit *et al.* consider scalable priority queues in [61], finding that for queues with a bounded number of priorities running on systems with a small number of processors that our design is efficient, except that we should consider both synchronizing per-priority and using CLH [39] or MCS [45] queue locks instead of test-and-`test-and-set` spinlocks.
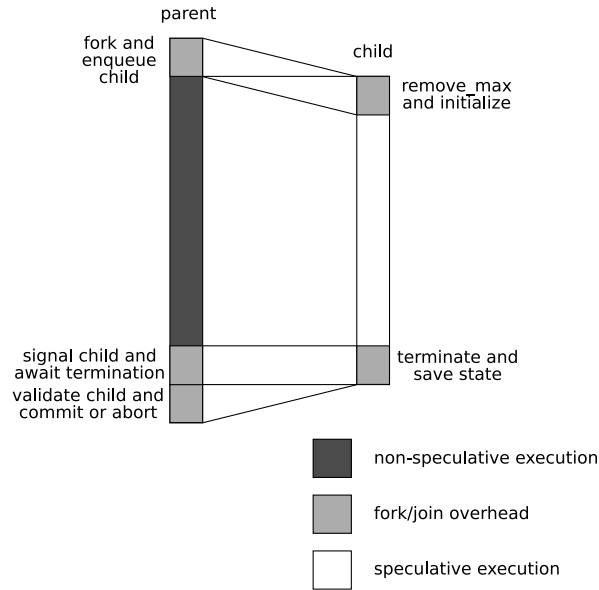
5 Experimental Analysis

In the following subsections we present data gathered over SPEC JVM98 [63] at size 100, using true speculative multithreading on a 4-way SMP machine, with the exception of some results on speculative thread sizes obtained using our single-threaded simulation for purposes of comparison.

We do not obtain speedup over sequential execution in our present implementation. Current running speeds over the SPEC benchmarks are within one order of magnitude of the unmodified SableVM switch interpreter engine (up to 10 times slower). As discussed in Section 4.2, optimisations both with respect to the choice of fork points and the queuing algorithms employed are expected to improve performance, but this is a significant research effort in its own right and outside the scope of the current work.

Nevertheless, as we examine speculation overhead, speculative thread sizes, and the percentage of bytecode instructions that can be executed in parallel, we shall see that there is room for optimism regarding the ultimate performance and moreover the current value of our design and implementation.

5.1 Speculation Overhead

In Figure 11, the parent thread is depicted as suffering overhead when forking, enqueuing, joining, and validating a child thread. Similarly, the child suffers overhead when it starts up and when it reaches some stopping condition, as outlined in Section 3.6. We instrumented our SableSpMT engine using the `rdtsc` instruction in order to profile execution times, and present overhead breakdowns for both non-speculative parent and speculative helper threads in Tables 2 and 3.

parent

fork and enqueue child

child

remove_max and initialize

signal child and await termination

validate child and commit or abort

terminate and save state

non-speculative execution

fork/join overhead

speculative execution

Figure 11: *Speculation overhead.* Both non-speculative parent and speculative child threads suffer wasted cycles due to overhead at fork at join points.

execution	`comp`	`db`	`jack`	`javac`	`jess`	`mpeg`	`mtrt`	`rt`
bytecode	39%	24%	29%	30%	21%	59%	49%	58%
fork	6%	15%	13%	13%	11%	5%	3%	4%
enqueue	4%	10%	10%	9%	7%	3%	2%	2%
other	2%	5%	3%	4%	4%	2%	1%	2%
join	53%	59%	57%	56%	67%	34%	47%	36%
pred_update	7%	13%	12%	11%	12%	6%	7%	7%
dequeue	5%	5%	5%	4%	5%	2%	2%	2%
wait	15%	14%	11%	11%	19%	8%	26%	11%
pred_check	4%	4%	4%	5%	7%	3%	2%	3%
buffer_check	4%	6%	6%	5%	5%	3%	1%	2%
child_pass	5%	5%	7%	6%	6%	3%	2%	3%
child_fail	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%
cleanup	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%
other	11%	10%	10%	12%	11%	7%	5%	6%

Table 2: *Non-speculative thread overhead breakdown.* The three main categories of execution times are normal bytecode and native method execution, time spent forking speculative children, and time spent joining speculative children. `enqueue` is incorporated into fork time, and the categories below `join` make up total join time. The `other` categories account for execution cycles that we did not instrument directly, and partially include the cost of instrumentation overhead itself.

The striking result in Table 2 is that the parent spends so much of its time forking and joining speculative threads that its opportunities for making progress through normal Java bytecode execution are reduced by up to 5-fold. We see that joining threads is significantly more expensive than forking threads, and that within the join process, predictor updates and waiting for the speculative child to halt execution are the most

execution	comp	db	jack	javac	jess	mpeg	mtrt	rt
child_wait	86%	82%	78%	78%	78%	55%	53%	71%
child_init	3%	4%	4%	4%	4%	2%	5%	4%
child_run	9%	12%	16%	16%	17%	41%	40%	24%
child_cleanup	<1%	<1%	<1%	<1%	<1%	<1%	<1%	<1%
bytecode	58%	50%	65%	64%	57%	83%	51%	56%
fork	35%	40%	28%	29%	36%	13%	41%	36%
pred_query	33%	38%	25%	26%	33%	11%	38%	33%
other	2%	2%	3%	3%	3%	2%	3%	3%
join	2%	2%	2%	2%	2%	1%	2%	2%

Table 3: *Speculative thread overhead breakdown.* Helper SpMT threads execute in a loop, waiting to remove children from the priority queue, initializing them, running them, and cleaning them up after thread termination. The running process itself (child_run) is divided into bytecode execution, the end of the fork process which involves making a return value prediction, and coming to a halt such that joining the parent non-speculative thread is possible. The other sub-category under fork accounts for the parts of the fork process, once the child is actually executing bytecode, that were not instrumented directly.

costly sub-categories. Other overhead sub-categories are not insignificant, and in general, optimisations to any of them will improve performance. We did not measure the speed of normal bytecode execution vs. speculative bytecode execution, although it will be instructive to look at the additional overhead on the SPMT_<X> instructions in Table 1.

The easiest way to eliminate overhead and bring non-speculative bytecode execution times much closer to 100% and thus overall running speed much closer to normal, is to disable speculation at fork points with histories of short and/or unsuccessful speculation attempts. However, whilst we acknowledge that this technique is probably necessary to achieve speedup in any finalized system, it presents two immediate problems. First, it reduces the amount of parallelism in the system as fewer speculative threads are forked and joined; thus, the limit on potential speedup and processor scalability obtainable through the implementation is reduced. Second, it sidesteps the problem of addressing overhead issues and designing efficient speculation algorithms to eliminate them.

In future studies, we will first attempt to reduce overhead as much as possible using aggressive optimisation techniques, whilst still enqueuing children at every fork point, and when we are confident that a performance limit has been reached, we will begin to disable speculation. At this time it is likely that compiler analyses to suggest good fork points and also reorganize code such that its structure is more amenable to SMLP can be exploited.

In Table 3, we observe several notable points about the execution of speculative children. First, the SpMT helper threads spend the majority of their time waiting to dequeue children from the priority queue and run them. The implication is that the priority queue is empty most of the time. In the experiments done in this section, we do not allow for speculative children to fork speculative children, and it is fairly intuitive that this is a contributor to the queue being empty. We do allow for multiple *immediate* children on the stack, as described at the end of Section 3.4, and in future work will analyse the effect that forking several generations of children has on both non-speculative and speculative overhead.

We also note that when the helper threads *are* running speculative children, they spend a majority of their time executing Java bytecode; in fact, proportionally more time is spent in Java bytecode speculatively
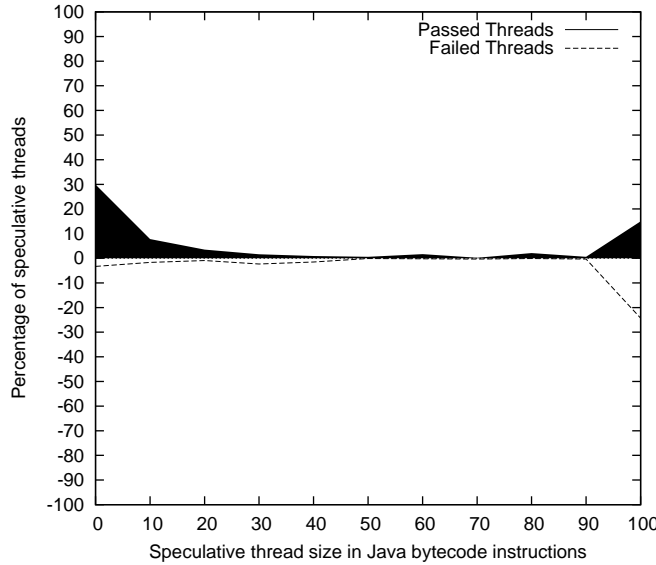
Figure 12: *Speculative sequence lengths in single-threaded simulation mode.*
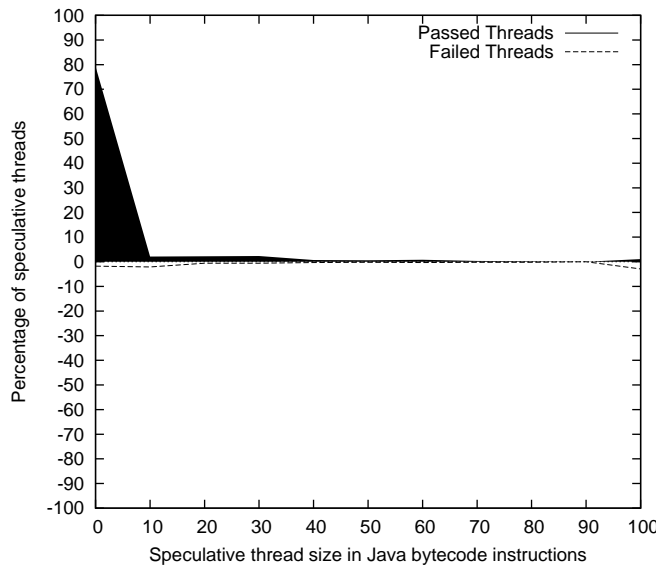
Figure 13: *Speculative sequence lengths in multithreaded mode on a 4-way SMP machine.*

than non-speculatively. Outside of bytecode execution, we see that predictor lookup is expensive, most likely because of synchronization on dynamically-expanding hashtables. We will address this category of overhead when we revisit our return value prediction work in the context of a true speculative multithreading environment; the implementation employed here was developed primarily to achieve high accuracies in the single-threaded simulation mode [51] (although in Section 5.3 it is shown that the existing return value prediction framework still helps to expose additional parallelism in the multithreaded execution mode).

5.2 Speculative Thread Sizes

In Figures 12 and 13, we see speculative thread sizes taken as an average over all of SPEC JVM98, for both single-threaded simulation and true speculative multithreading modes. Failed threads can be considered as wasted cycles, and are thus shown as negative percentages in the graph. As we move from the simulation to a multiprocessor, the speculative thread sizes decrease rather dramatically, jumping from 30% of all threads having a length of 0–10 bytecode instructions to nearly 80%.

The explanation for this shift is two-fold. First, in the simulation, speculative threads execute until they reach either 1000 instructions or an unpassable speculative instruction such as `MONITOREXIT`. In the multithreaded mode, however, as soon as a parent returns from a call and finds a child on the stack, it will signal it and await its termination. Thus the question is one of load balancing, and these data indicate that the parent is likely to enter a short leaf method after forking a child. Compiler transformations such as inlining that increase method granularity should in general allow children to execute more instructions before being joined; indeed, one of the contributors to speedup that Hu *et al.* identified in their SMLP experiments was the presence of an inlining JIT compiler [25]. Alternatively, we could opt not to fork children if the parent is known either statically or dynamically to be entering a short method.

Second, speculative execution overhead is non-existant in the single-threaded simulation, whilst during true multithreaded execution children suffer delays that impede the progress they make before being joined by their parent. As we eliminate the overhead identified in Section 5.1, we expect the average speculative child thread size to grow.

Thread lengths are usually small in hardware SpMT designs and simulations, particularly for non-numeric programs; in [28] SPECINT2000 benchmarks result in thread sizes up to 43 machine instructions, a significant improvement over previous work. Our Java bytecode thread lengths can be much larger, not uncommonly 100s and sometimes 1000s of bytecodes; of course this does not directly map to improved performance, and the relation between Java bytecodes and machine instructions is not trivial, but it can be seen as an indication that improvements in parallelism are possible through higher level approaches. One of the reasons that speculative threads actually succeed at such long lengths is that we have information about language features that is not available at the hardware level, and which can be used to avoid entering into unsafe speculative situations (refer to Section 3.2.2 for a detailed analysis of Java bytecode and its effect on speculative safety).
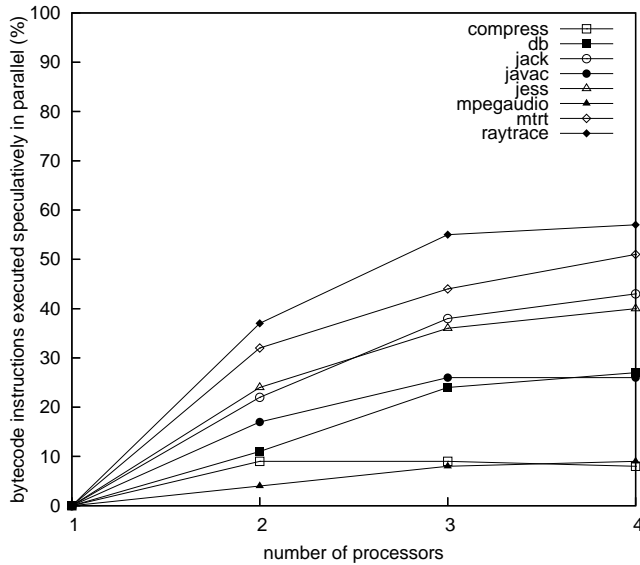
5.3 Speculative Coverage

Finally we look at speculative *coverage*, or the amount of parallelism in terms of bytecode instructions that our implementation is able to uncover in multiprocessor systems. In Figures 14 and 15, we examine the effect that introducing additional CPUs into our system has on uncovering parallelism in the SPEC JVM98 benchmarks, and the benefits of return value prediction. In these experiments, there is one helper SpMT thread mapped to each additional CPU after the first, which executes non-speculative Java code.

We find that introducing more CPUs increases parallelism, and that in our current system, in which speculative threads are *only* forked by the parent non-speculative thread, the processor scaling is best as we move from zero to one to two speculative CPUs; adding a third CPU, whilst still beneficial, does not have as great an impact. We also find that return value prediction plays an important role in speculative method-level parallelism, corroborating the result previously obtained by Hu `et al.` in their analysis of the SPEC benchmarks in trace-based Java SMLP. In the absence of RVP, we achieve an average bytecode instruction parallelism of 19%, and with the introduction of RVP into the system this increases to 33%.

26
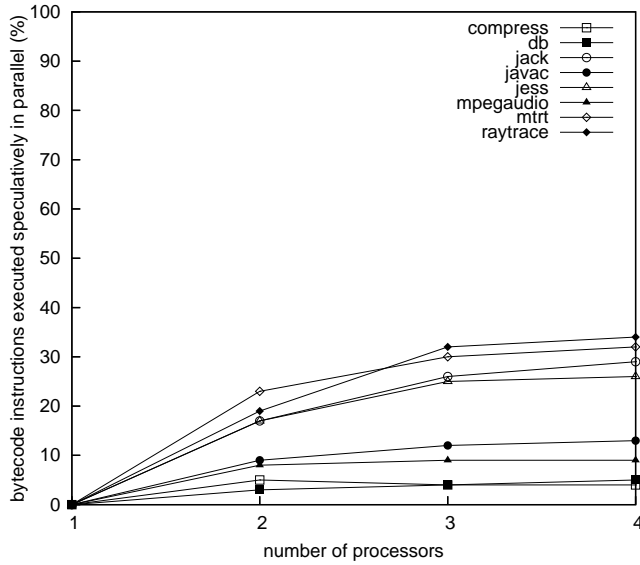
Figure 14: Speculative coverage without RVP

Figure 15: Speculative coverage with RVP

Finally, it would appear from the graphs that the SPEC JVM98 benchmarks can be broken into two classes. Somewhat predictably, `raytrace` and `mtrt` exhibit the highest degrees of parallelism, as these programs are known to be embarrassingly parallel; nevertheless, it is interesting that we can obtain this extra parallelism out of `raytrace` without actually using any concurrent transformations of the code. In the future, we plan to study a set of concurrent algorithms and determine just how close SMLP is able to bring a sequential program to its hand-parallelized equivalent. `jack` and `jess` are in the same class as `mtrt` and `raytrace`, exhibiting relatively high degrees of parallelism both with and without RVP. Interestingly, with respect to the second class of benchmarks which exhibit lower degrees of parallelism, adding RVP to the system allows for separation of `db` and `javac` from `mpegaudio` and `compress`. Again this supports that accurate return value prediction can significantly impact performance in an SMLP system, and can often change the observed behaviour of a program.

6 Related Work

In a general senses, automatic approaches to parallelism have been sought for many years [5]. These have been most successful when analysing loop-based, highly-structured scientific applications, typically C or Fortran based [12, 23], though Java experiments have been done as well [1]. Various efforts have looked at analysis methods to improve applicability and performance of such approaches and more abstract measurements have been done in this context [44, 54, 62], but designs and results for arbitrary, irregular, OO programs remain less common.

Speculative multithreading approaches have been developed primarily in the context of novel hardware environments. A number of general purpose speculative architectures such as the Multiscalar architecture [18], the SuperThreaded architecture [69], Trace processors [56], MAJC [68], and several other designs [64, 17, 32] have been proposed, and simulation studies have generally shown quite good potential speedups. Steffan et al. give a recent implementation and good overview of the state of the art in [65]. An abstract, general formula for speculative efficiency is defined by Greiner and Blelloch [21, 22].

From the speculative hardware level, an executing Java Virtual Machine does not exhibit distinguished performance in comparison with other applications [71]. As an interpreted language, however, Java can provide higher level abstractions and information than generic machine code, and the performance of specialized hardware and software for speculative Java execution has thus been of interest. Chen and Olukotun pioneered work on method level speculation for Java, showing reasonable speedups on a simulated speculative architecture [9]. This work has been extended to incorporate hardware analysis modules that cooperate with the VM to improve runtime performance [11], and has culminated in an overall design that achieves excellent speedups in simulated executions of a variety of benchmarks [10]. Hu et al. used Java traces applied to simulated hardware as part of a study of the impact of return value prediction [25].

Whether applied to Java or not, hardware designs necessarily require the expensive step of hardware construction. Still, there have been fewer studies purely at the software level. "SoftSpec" is a software speculation environment: loops with identifiable strides are speculatively parallelized at runtime [4]. Kazi and Lilja analysed a software form of the SuperThreaded architecture [30], and more general software designs have been evaluated on C [57] and Fortran benchmarks [13] with respect to loop-level speculation. Prahbu and Olukotun have advocated manual, C source transformations and analyses to help thread level speculation map to SpMT hardware [53].

Only very limited studies on language-level speculation for Java have been done previously. Yoshizoe *et al.* give results from a partially hand-done loop-level speculation strategy implemented in a rudimentary (e.g., no GC) prototype VM [72]. They show good speedup for simple situations, but lack of heap analysis limits their results. Kazi and Lilja provided the first convincing evidence that language level speculation can be very effective in Java using manual source transformations [29]. Although source level (versus VM-level) transformations mitigate the features of SpMT that can be supported, these studies show that special hardware is not a requirement of SpMT, and also that there are no inherent problems with the Java language, platform, or programming model. Using data from Java programs translated to C and executed on simulated hardware, Warg and Stenström argue that Java-based SpMT has inherently high overhead costs which can only be addressed through hardware support [71]. Data from our experiments, however, indicates that there is sufficient potential parallelism to offset quite large overheads.

6.1 Value Prediction

We have made extensive use of return value prediction to improve our speculative system. Value prediction itself is a well-known technique for allowing speculative execution of various forms to proceed beyond normal execution limits, and a wide variety of prediction strategies have been defined and analyzed. These extend from relatively simple last value predictors [38] to more complex differential context (DFCM) value predictors [59, 6], hybrid predictors [8, 25, 51], and even machine-learning based perceptron predictors [67, 60]. Most value predictors are designed with hardware implementation constraints in mind, though again software techniques have been examined; Li *et al.*, for example, develop a software prediction scheme based on profiling and inserting recovery code following a cost-driven model [35]. Value prediction in SpMT has been explored by several groups [25, 14, 9, 48, 49, 71]. The utility of return value prediction for method level speculation in Java was shown by Hu et al. [25], with further prediction accuracy investigated by the authors [51], and many optimizations to value prediction accuracy and cost have been considered [7, 16, 50]. The return value prediction approach here is based on the design in [51] and [50].

6.2 SpMT Optimizations

A naive implementation of software or VM-based SpMT can result in relatively high overhead costs, and so optimization of the different operations involved is critical, both in terms of research exploration, and with respect to practical usage as a research tool. As well as return value prediction we have employed a number of optimization techniques specific to our implementation, discussed in Section 4. Other optimizations exist, however, that have shown good promise in more general situations.

Selecting appropriate speculation sites is of course a crucial aspect of speculative performance, and several static techniques based on ahead-of-time compiler transformations for SpMT architectures have been suggested. Bhowmik and Franklin describe a general compiler support framework including both loop-based and non-loop-based thread partitioning [3]. Johnson *et al.* transform and solve the partitioning problem using a weighted, min-cut cost model [28]. Dynamic thread partitioning strategies reduce preprocessing needs and have been described for several simulated architectures [15, 41]. Unfortunately, being aimed at SpMT hardware the threads extracted from these approaches are too low-level and fine grained (e.g. 11-43 machine instructions in SPEC-INT 2000 benchmarks in [28]) to apply directly in our environment.

Specific compiler optimizations have also been developed. Zhai *et al.* define and evaluate instruction scheduling optimizations based on the length of the *critical forwarding path*, or time between a definition and a dependent use of the same variable [73]. This can be quite effective at reducing stalls when variable dependencies between threads are enforced through synchronization. A flow analysis described in [31] depends on a compiler to label memory references as *idempotent* if they need not be tracked in speculative storage and can instead access main memory directly. This reduces the overhead in terms of space and time of buffering the reads and writes of a speculative thread. Ooi *et al.* develop further optimizations to reducing buffer costs given hardware buffering constraints [47]. In SMLP no locals are visible from other threads, and so can easily be designated idempotent without implementing an analysis. In Java this means that values on a thread's stack need not be tracked in the speculative storage. The VM environment, however, is well suited to incorporating compiler optimizations and analyses, and use of such information is part of our future work.

Lock synchronization poses another potential problem for speculative execution; lock changes affect the program globally, and conservatively may require the speculative thread stop if not carefully tracked. However, locking itself can be quite amenable to speculation, and optimizations are indeed possible. Martínez

and Torrellas show how Speculative Locking can reduce the impact of contention on coarse-grained locking structures [42, 43]; this has been extended to allow *post-facto* lock acquisition ordering through the Speculative Lock Reordering strategy of Rundberg and Stenström [58]. Rajwar and Goodman define a microarchitecture-based speculative lock acquisition system [55]. In our current implementation synchronization causes speculative threads to stop speculating, and so inclusion of a speculative approach to locking would be quite useful.

7 Future Work

Our SableSpMT design is an initial, prototype implementation. We have achieved quite reasonable performance levels in this context, and are able to handle a fairly complete range of input programs. Nevertheless, there are a number of restrictions that could be lifted, as well as enhancements and optimizations that could be applied to improve performance and applicability.

Although we handle the complete Java language, to ensure safety our speculative threads must stop speculating at synchronization points. The impact of these reductions in speculation is not large in our (mostly sequential) benchmark suite, but to better accommodate lock intensive programs some form of speculative locking [42, 58] would be quite useful. An implementation within our framework would allow for detailed analysis of the interaction with other optimizations to speculative behaviour.

Experimentation in our system is simplified through use of a highly-portable, research virtual machine, SableVM. We would also like to port our work to a JIT compiler environment, such as Jikes [27], or very recently a branch of SableVM itself [2]. Although the presence of a JIT complicates a research SpMT implementation, JITs have a significant impact on performance, and the improved performance of natively executing code in a VM may expose more relative overhead in the SpMT implementation. Unrelated JIT optimizations, such as inlining, may also change the character of speculative execution, and costs of RVP. Of course a JIT also implies new optimization opportunities, including specialized, native speculative code, and techniques based on the presence of dynamic program analysis information.

Many SpMT hardware studies make use of (static) compiler analysis information [3, 10]. Part of our return value prediction design also allows for the optional use of static analysis information through the Java class-file attribute mechanism [52]. There are, of course, many other compiler analyses that may help different aspects of speculative operation. Information on expected relative length of speculative threads, critical forwarding path information [73], return value predictability, method purity [66], and so on can be computed offline, and applied at runtime. Dynamic information tends to be more precise, but the use of static analyses has the advantage of minimal extra runtime overhead. A larger-scale use of ahead-of-time information would be to optimize for a specific VM startup sequence. Startup code in a JVM can account for a significant fraction of execution time for smaller Java programs, and the use of a built-in, optimized sequence of SpMT operations could improve the common startup path.

Our existing optimizations could also be further tuned, and increased in scope. General load value predictors [8], for example, could be used to predict heap values, and thus improve speculation depth. Software overhead for load prediction is potentially quite high, and so this would have to be combined with profiling or static information to select prediction sites judiciously. Our current environment provides an excellent base for such experimentation.

We have designed and used a single-threaded mode for both debugging and research purposes. This mode has been instrumental in experimentally validating the functionality and correctness of the implementation, and it has also been useful in determining the potential limits of speculative coverage. We would like to

extend this mode to other speculative approaches, and use it as a tool to experimentally determine upper bounds on speculative behaviour. This information would be useful in quickly evaluating properties of different speculative strategies for Java, or even specific Java workloads.

Finally, we have not yet given full consideration to the impact of the new Java memory model [40] on our optimizations. The use of strongly ordering instructions during queuing and thread joins act as a memory barrier before and after each speculative execution. This limits thread inconsistency, but write buffering and a hierarchy of speculative threads implies that a detailed consideration is required. We hope to investigate and enforce JMM compliance in future versions.

8 Conclusions

Efficient automatic parallelization is a difficult goal. There have been successes for scientific applications, but in the context of object-oriented languages and programs based on irregular data structures traditional static, loop-based techniques are less effective. Thread level speculation has shown considerable promise for being able to extract some parallelism from such applications. Significant further research is required, however, in order to optimally exploit SpMT for use in higher level environments and on commodity multi-processors.

We have examined the use of thread level speculation within a Java virtual machine by implementing a complete, robust design for speculative method level parallelism. Our approach includes optimized designs for thread communication, memory management, and a high quality return value prediction system. We are able to handle the entire Java language, including exceptions, garbage collection, native methods, finalization, and already multithreaded inputs. This implementation runs at reasonable speeds, and is thus both a practical tool for larger scale investigations of various forms, and a good indication that SpMT is feasible at the VM level.

We have gathered and analysed data on speculative performance for a number of relatively large, real-world benchmarks. We are thus able to provide non-trivial results on SpMT performance and overhead costs in the VM context. This data shows where overhead costs are concentrated, and provides important indicators for the direction of future optimization and implementation research.

This work is research level, and experimental; we do not claim to have solved all performance issues related to SpMT in a VM environment. There is certainly room for improvement in our final performance figures. However, we have demonstrated that an implementation is feasible, and provided important data and analysis information. We hope our work here, both in future optimization directions and in providing a practical vehicle for experimentation, inspires further research in this area.

9 Acknowledgements

References

[1] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. Automatic loop transformations and parallelization for Java. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 1–10. ACM Press, 2000.

[2] D. Bélanger. SableJIT: a retargetable just-in-time compiler. Master's thesis, McGill University, Montréal, Canada, February 2005.

[3] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA '02)*, Aug. 2002.

[4] D. Bruening, S. Devabhaktuni, and S. Amarasinghe. Softspec: Software-based speculative parallelism. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.

[5] M. Burke, R. Cytron, J. Ferrante, W. Hsieh, V. Sarkar, and D. Shields. Automatic discovery of parallelism: a tool and an experiment (extended abstract). In *PPEALS '88: Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, pages 77–84. ACM Press, 1988.

[6] M. Burtscher. An improved index function for (D)FCM predictors. *Computer Architecture News*, 30(3):19–24, June 2002.

[7] M. Burtscher, A. Diwan, and M. Hauswirth. Static load classification for improving the value predictability of data-cache misses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 222–233. ACM Press, 2002.

[8] M. Burtscher and B. G. Zorn. Hybrid load-value predictors. *IEEE Transactions on Computers*, 51(7):759–774, July 2002.

[9] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 1998.

[10] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *Proceedings of the 30th annual International Symposium on Computer Architecture (ISCA)*, pages 434–446. ACM Press, June 2003.

[11] M. K. Chen and K. Olukotun. TEST: A tracer for extracting speculative threads. In *Symposium on Code Generation and Optimization (CGO '03)*, Mar. 2003.

[12] J.-H. Chow, L. E. Lyon, and V. Sarkar. Automatic parallelization for symmetric shared-memory multiprocessors. In *CASCON '96: Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, page 5. IBM Press, 1996.

[13] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 13–24. ACM Press, June 2003.

[14] M. Cintra and J. Torrellas. Eliminating squashes through learning cross-thread violations in specu-lative parallelization for multiprocessors. In *Eighth International Symposium on High-Performance Computer Architecture (HPCA '02)*, Feb. 2002.

[15] L. Codrescu and D. S. Wills. On dynamic speculative thread partitioning and the MEM-slicing algo-rithm. *Journal of Universal Computer Science*, 6(10):908–927, 2000.

[16] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 71–81. ACM Press, 2004.

[17] R. Figueiredo and J. Fortes. Hardware support for extracting coarse-grain speculative parallelism in distributed shared-memory multiprocessors, 2001.

[18] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin–Madison, 1993.

[19] E. M. Gagnon. *A Portable Research Framework for the Execution of Java Bytecode*. PhD thesis, McGill University, Montréal, Québec, Dec. 2002.

[20] E. M. Gagnon. SableVM. `http://www.sablevm.org/`, 2004.

[21] J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 309–321, Jan. 1996.

[22] J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Transactions on Programming Languages and Systems*, 21(2):240–285, 1999.

[23] E. Gutiérrez, O. Plata, and E. L. Zapata. A compiler method for the parallel execution of irregular re-ductions in scalable shared memory multiprocessors. In *ICS '00: Proceedings of the 14th international conference on Supercomputing*, pages 78–87. ACM Press, 2000.

[24] M. Hachman. Intel to redefine performance at IDF. `http://www.extremetech.com/ article2/0,1558,1641930,00.asp`, September 2004.

[25] S. Hu, R. Bhargava, and L. K. John. The role of return value prediction in exploiting speculative method-level parallelism. *Journal of Instruction-Level Parallelism*, 5:1–21, Nov. 2003.

[26] O. T. International. Eclipse platform technical overview. `http://www.eclipse.org/ whitepapers/eclipse-overview.pdf`, February 2003.

[27] Jikes Research Virtual Machine. `http://www-124.ibm.com/developerworks/oss/ jikesrvm/index.shtml`.

[28] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 59–70. ACM Press, 2004.

[29] I. H. Kazi and D. J. Lilja. JavaSpMT: A speculative thread pipelining parallelization model for Java programs. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 559–564. IEEE, May 2000.

[30] I. H. Kazi and D. J. Lilja. Coarse-grained thread pipelining: A speculative parallel execution model for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):952–966, 2001.

[31] S. W. Kim, C. Ooi, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Reference idempotency analysis: a framework for optimizing speculative execution. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '01)*, volume 36, pages 2–11, June 2001.

[32] S. W. Kim, C.-L. Ooi, I. Park, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor, 2001.

[33] D. Lea. The `java.util.concurrent` synchronizer framework. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, pages 1–9, St. John's, Newfoundland, Canada, July 2004.

[34] O. Lhotak. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, Montréal, Québec, Dec. 2002.

[35] X.-F. Li, Z.-H. Du, Q. Zhao, , and T.-F. Ngai. Software value prediction for speculative parallel threaded computations. In *The First Value-Prediction Workshop*, pages 18–25, San Diego, CA, jun 2003.

[36] S. Liang. *The Java Native Interface. Programmer's Guide and Specification*. Addison-Wesley, Reading, Massachusetts, 1st edition, June 1999.

[37] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, 2nd edition, 1999.

[38] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 138–147. ACM Press, 1996.

[39] P. S. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171. IEEE Computer Society, Jan. 1994.

[40] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*, pages 378–391. ACM Press, 2005.

[41] P. Marcuello and A. Gonzalez. Thread-spawning schemes for speculative multithreading. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 55–64, Feb. 2002.

[42] J. F. Martínez and J. Torrellas. Speculative locks for concurrent execution of critical sections in shared-memory multiprocessors. In *Workshop on Memory Performance Issues (WMPI), at the International Symposium on Computer Architecture (ISCA '01)*, Gothenburg, Sweden, June 2001.

[43] J. F. Martínez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '02)*, pages 18–29, San Jose, CA, Oct. 2002.

[44] K. S. McKinley. Evaluating automatic parallelization for efficient execution on shared-memory multiprocessors. In *ICS '94: Proceedings of the 8th international conference on Supercomputing*, pages 54–63. ACM Press, 1994.

[45] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, Feb. 1991.

[46] N. Mukherjee and J. R. Gurd. A comparative analysis of four parallelisation schemes. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 278–285. ACM Press, 1999.

[47] C. Ooi, S. W. Kim, I. Park, R. Eigenmann, B. Falsafi, and T. N. Vijaykumar. Multiplex: unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *International Conference on Supercomputing*, pages 368–380, 2001.

[48] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715.

[49] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, Oct. 1999.

[50] C. J. F. Pickett and C. Verbrugge. Compiler analyses for improved return value prediction. Technical Report SABLE-TR-2004-6, Sable Research Group, McGill University, Oct. 2004.

[51] C. J. F. Pickett and C. Verbrugge. Return value prediction in a Java virtual machine. In *Second Value-Prediction and Value-Based Optimization Workshop*, pages 40–47, Boston, MA, October 2004.

[52] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing java using attributes. In *Compiler Construction, 10th International Conference (CC 2001)*, pages 334–554, 2001.

[53] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12. ACM Press, 2003.

[54] W. Pugh and D. Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages Systems*, 16(4):1248–1278, 1994.

[55] R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution, 2001.

[56] E. Rotenberg. *Trace Processors: Exploiting Hierarchy and Speculation*. PhD thesis, University of Wisconsin–Madison, 1999.

[57] P. Rundberg and P. Stenström. An all-software thread-level data dependence speculation system for multiprocessors. *Journal of Instruction-Level Parallelism*, 3:1–28, Oct. 2001.

[58] P. Rundberg and P. Stenström. Reordered speculative execution of critical sections. In *Proceedings of the International Conference on Parallel Processing (ICPP '02) (submitted)*, Feb. 2002.

[59] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO)*, pages 248–258, Dec. 1997.

[60] J. Seng and G. Hamerly. Exploring perceptron-based register value prediction. In *Second Value-Prediction and Value-Based Optimization Workshop*, pages 10–16, Boston, MA, October 2004.

[61] N. Shavit and A. Zemach. Scalable concurrent priority queue algorithms. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing (PODC '99)*, pages 113–122. ACM Press, 1999.

[62] B. So, S. Moon, and M. W. Hall. Measuring the effectiveness of automatic parallelization in SUIF. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 212–219. ACM Press, 1998.

[63] The SPEC JVM Client98 benchmark suite. `http://www.spec.org/jvm98/jvm98/`.

[64] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *ISCA*, pages 1–24, 2000.

[65] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. The stampede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 2005. to appear.

[66] A. Sălcianu and M. Rinard. A combined pointer and purity analysis for java programs. Technical Report MIT-CSAIL-TR-949, Massachusetts Institute of Technology, May 2004.

[67] A. Thomas and D. Kaeli. Value prediction with perceptrons. In *Second Value-Prediction and Value-Based Optimization Workshop*, pages 3–9, Boston, MA, October 2004.

[68] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse. The MAJC architecture: A synthesis of parallelism and scalability. *IEEE Micro*, 20(6):12–25, 2000.

[69] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, 1999.

[70] R. Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, McGill University, Montréal, Québec, July 2000.

[71] F. Warg and P. Stenström. Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 221–230. IEEE, Sept. 2001.

[72] K. Yoshizoe, T. Matsumoto, and K. Hiraki. Speculative parallel execution on JVM. In *First UK Workshop on Java for High Performance Network Computing*, 1998.

[73] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication. In *ASPLOS X*, San Jose, CA, USA, Oct. 2002.

[74] G. Zhang, P. Unnikrishnan, and J. Ren. Experiments with auto-parallelizing SPEC2000FP benchmarks. In *LCPC '04: The 17th International Workshop on Languages and Compilers for Parallel Computing*, 2004.