McGill University
School of Computer Science
Sable Research Group

Understanding Method Level Speculation

Sable Technical Report No. 2009-2

Christopher J.F. Pickett and Clark Verbrugge and Allan Kielstra
{cpicke,clump}@sable.mcgill.ca, kielstra@ca.ibm.com

September 19th, 2009

Understanding Method Level Speculation

Christopher J. F. Pickett Clark Verbrugge
School of Computer Science, McGill University
Montréal, Québec, Canada
{cpicke,clump}@sable.mcgill.ca

Allan Kielstra
IBM Toronto Lab
Markham, Ontario, Canada
kielstra@ca.ibm.com

Abstract

Method level speculation (MLS) is an optimistic technique for parallelizing imperative programs, for which a variety of MLS systems and optimizations have been proposed. However, runtime performance strongly depends on the interaction between program structure and MLS system design choices, making it difficult to compare approaches or understand in a general way how programs behave under MLS. Here we develop an abstract list-based model of speculative execution that encompasses several MLS designs, and a concrete stack-based model that is suitable for implementations. Using our abstract model, we show equivalence and correctness for a variety of MLS designs, unifying in-order and out-of-order execution models. Using our concrete model, we directly explore the execution behaviour of simple imperative programs, and show how specific parallelization patterns are induced by combining common programming idioms with precise speculation decisions. This basic groundwork establishes a common basis for understanding MLS designs, and suggests more formal directions for optimizing MLS behaviour and application.

1. Introduction

Method level speculation (MLS) is an optimistic execution technique for parallelizing sequential programs. Under MLS, when a non-speculative "parent" thread reaches a method invocation, it can fork a speculative "child" thread that begins executing at the method continuation, as if the method has already returned. Children execute in an isolated fashion, and upon returning from the method call, if there were no memory dependence violations between the child and the parent, the child state is committed to memory, and the parent resumes execution where the child left off. Given low enough overheads, the resultant parallelism is then a source of speedup on multiprocessor machines.

Performance of MLS strongly depends on the choice of fork points, as well as the basic model of MLS execution. MLS designs may permit parents to have multiple children (*out-of-order* nesting), or for children to fork further children (*in-order* nesting), with significant variance in subsequent behaviour. In Figure 1, for instance, the initial creation of a speculative child at the call to `a()` followed by out-of-order nesting creates parallelism between between `b()` and `X` but not between `c()` and `Y`, whereas with in-order nesting the situation is reversed. Lack of understanding of such fundamental differences makes both the choice of appropriate fork heuristics and comparison of different MLS models difficult.

```
a() {  // parent creates child 1 here
  b(); // can parent create child 2 to execute X?
  X;
}      // child 1 begins execution here
c();   // can child 1 create child 3 to execute Y?
Y;
```

Figure 1. *MLS choices.* Assuming execution of `a()` does not complete before `c()` is started, with out-of-order nesting `b()`, `X`, and `c()Y` are parallelized. With in-order nesting `b()X`, `c()`, and `Y` execute concurrently.

To better understand the behaviour of programs under MLS, we develop two models of MLS. We describe an abstract, list-based model that represents a broad variety of MLS designs, and use it to demonstrate correctness in terms of sequential equivalence. Our model is general and flexible enough to include in-order, out-of-order, mixed, and even non-MLS designs. We relate this model to a more concrete, stack-based representation that provides detail suitable for implementation, while exposing the program stack manipulations that underly different MLS strategies. From this formalism we show how simple differences in coding idioms produce widely varying parallel execution patterns, which implies that runtime efficiency strongly depends on the exact MLS strategy. The technical complexity inherent in MLS implementation tends to require that individual studies commit to a specific speculation model; our approach can be used to further explain the performance characteristics of any such study, and to facilitate deeper exploration of the impact of fundamental MLS design decisions.

1.1 Contributions

We make the following specific contributions:

- Using a novel, list-based abstraction of general speculation we show correctness in both sequential and multithreaded contexts for in-order and out-of-order MLS, as well as non-method based techniques such as loop-level speculation or even arbitrary speculation.
- We propose a stack operation semantics as a concrete model, suitable for both implementation and direct visualization of how and when speculative code may execute under MLS. Beyond basic fork/commit matching, previous work has not considered a descriptive semantics of stack buffering, instead relying on general dependence buffering or transactional memory support.
- By examining a number of common coding idioms in relation to our stack formalism, we are able to show how speculative execution can map to various specific forms of parallelism depending on code layout and precise speculation choices.

The following section describes our core list-based model, including a proof of correctness for several MLS designs. This is followed in Section 3 by our stack formalism, which we then use to show relevant code behaviours in Section 4. Related work is presented in Section 5, followed by conclusions and future work.

2. List Abstraction

Although designs that support in-order nesting clearly differ from out-of-order, and hybrid models of increasing complexity are possible, basic componentry is similar. The list-based model we present here takes advantage of these basic similarities to produce a common abstraction, encompassing not only arbitrary nesting choices, but also speculative designs beyond pure method level parallelism. Here we first define and apply our abstraction to sequential programs, and then extend the result to multithreaded contexts.

Our proof of sequentiality assumes a program $\mathcal{P}$ is composed of one or more program threads t each executing individual code P^t (where the thread is clear we use P). At each point in execution the thread operating on code P computes the result of $P_a|_n$, where a is the code index of the start of computation and n is the number of operations performed.

Speculative execution may be initiated at any method invocation (although the model easily extends to arbitrary speculation points), with speculative code executing until signaled to terminate by the return of its parent thread, or until it encounters an *unsafe* operation—I/O, synchronization or other operations that may not be safely executed speculatively. Once terminated and joined with its parent, a speculative thread is validated to ensure correct execution was performed and either committed or (eventually) aborted. We now argue that the computation of $\mathcal{P}$ in both sequential and parallel contexts is equivalent with or without MLS. Throughout this proof we make the basic assumption that speculative execution is strongly isolated, having no direct impact on actual program output until committed by a non-speculative thread.

2.1 Sequential Programs

Proving speculative computation matches sequential requires showing that the committed results of speculation always produce an execution trace equivalent to some sequential execution. This reduces to demonstrating the following properties:

1. Committed speculative computation always begins with correct input.
2. Commits are performed in an order that respects sequential execution.
3. Committed and non-speculative execution represents a complete trace without gaps.

Prior to any frame push a speculative thread may be forked, and at the corresponding pop a speculative thread may be committed; threads may be aborted at any point. At any point during execution the speculative child hierarchy forms a tree with a well defined order of computation between threads. We use the precedence inherent in this structure in order to inductively show the correspondence between speculative execution and sequential execution. The basic representation will then be as a list, generated as a threading of the speculative heirarchy:

Definition 2.1. *A* speculation list *is a tuple $(V, R, \alpha, \delta, \omega)$, where V is a linked list of threads, $R : V \times V$ is the transitive reduction of a total ordering relation, and functions $\alpha : V \to (\text{Code} \times \text{Env})$, $\delta : V \to (\mathcal{N} \times \text{Code} \times \text{Env})$ and $\omega : V \to (\text{Code} \times \text{Env})$ all map threads to partial or full execution states—initial assumptions, full current state (including count of operations executed), and ending state respectively.*

If r is the first node in the list then r is a non-speculative thread, while all other $v \in V, v \neq r$ are speculative.

Speculation lists are constructed dynamically during a program execution. The list begins consisting of a singleton node r and mappings $\alpha(r) = (0, I)$, $\delta(r) = (0, 0, \emptyset)$, $\omega(r) = (\text{EOF}, \emptyset)$, where 0 is the program counter at entry, I is the initial environment, and EOF indicates program termination. Since only one thread exists initially ordering is also trivial: $R = \emptyset$.

As operations are performed a new speculation list L is constructed based on the previous L' and current instruction i executed by a given thread t. To simplify notation where not otherwise indicated elements of L are the same as L'. Figure 2 shows the rules applied to each possible operation and below we further describe each step. The operation JoinPoint : Code $\times$ Env $\to$ Env defines the subset of the given environment that identifies reaching the specified code location in the same stack frame.

OP If i is a basic computation, i : Code $\times$ Env $\to$ Code $\times$ Env, then the current state must be appropriately advanced. This includes an increment to the count of operations executed. Operation i is isolated except for reading heap values, which if they are not found in the heap of $\delta'(t)$ or in existing assumptions must be predicted, speculatively retrieved from the parent environment, or otherwise derived. Any such assumptions are recorded by adding the variable and value read to the input state for later comparison during commit validation.

FORK In method-level speculation forks are performed only at method calls; let y be the code point immediately corresponding to the method return associated with fork operation i.

A new speculative thread s is created so as to begin execution after the method returns, at code point y, starting in the current state of t and inheriting t's stack. No initial assumptions about input state have been made ($\alpha(s) = (y, \emptyset)$), although this could be populated with predicted values. The termination of thread t is also adjusted to ensure it stops when s starts.

Not shown in the FORK rule of Figure 2 is that s is added to the speculative list as a new immediate successor to t: $V = V' \cup \{s\}$, $R = R' \backslash \{(t, v)\} \cup \{(t, s), (s, v)\}$.

Note that this effectively partitions the designated execution of p ($[\alpha'(p), \omega'(p))$) into $[\alpha'(p), \alpha(s))$ followed by $[\alpha(s), \omega'(p))$.

ABORT**,** COMMIT Let i be an abort or commit of thread s. Terminating a thread requires it be removed from the speculative list, undoing the effect of its insertion. We do not abort or join the root, non-speculative thread, so assume a predecessor p exists: $(p, s) \in R'$. List operations not shown in Figure 2 for ABORT and COMMIT consist of $V = V' \backslash \{s\}$, $R = R' \backslash \{(p, s), (s, v)\} \cup \{(p, v)\}$.

Once removed the thread's predecessor becomes responsible for completing any remainder of the the child's execution: $\omega(p) = \omega'(s)$. In the case of an abort this completes the operation, giving the simple ABORT rule in Figure 2. Note that aborts can be issued at any time,

A successful commit is performed only by an immediate predecessor, which has reached its termination point. Validation requires that the termination state match the input assumptions of the speculative thread. If so the child speculative state is merged into the parent state, overwriting the parent's stack, and causing the parent to inherit any new assumptions. Heap state in child threads is only partial, and so a commit requires merging parent and speculative heap states, giving child bindings preference. These commits are captured by a non-associative binary merge operator, $\sqcup$, defined as follows:

$$H_1 \sqcup H_2 = H_1 \cup \{(x, v) \in H_2 | \ (x, w) \notin H_1 \text{ for any } w\}$$

Our proof of the equivalence of MLS to (some) sequential execution first demonstrates the equivalence for single-threaded programs under MLS, and then argues for the case of multithreaded programs. For this we rely on the following two simple lemmas, presented without proofs:

Lemma 2.2. *Let H be an environment in $\delta(s)$ for some s. There does not exist a variable v and distinct values $m_1 \neq m_2$ such that $(v, m_1) \in H$ and $(v, m_2) \in H$.*

Lemma 2.3. *Let s be such that $(s, t) \notin R$ for any t. Then $\omega(s) = (EOF, \emptyset)$.*

The above lemmas establish that environments do not have multiple mappings for the same variable/memory-location (Lemma 2.2), and that the final thread in the speculation list is responsible for completing execution (Lemma 2.3). Both properties are invariants of every transformation.

$$\textsc{abort}\ \dfrac{(p,s)\in R'}{\omega(p)=\omega'(s)}$$

$$\textsc{fork}\ \dfrac{\begin{array}{cc} s=\text{new} & y=\text{c}+1 \\ \delta'(t)=(n,\text{c},H) & \text{JoinPoint}(H,y)=S \end{array}}{\alpha(s)=(y,\emptyset),\ \delta(s)=(0,y,\emptyset),\ \omega(s)=\omega'(t),\ \omega(t)=(y,S)}$$

$$\textsc{op}\ \dfrac{\begin{array}{c} \text{safe}(i)\vee\ (*,t)\notin R' \\ \delta'(t)=(n,c,H_\delta) \qquad \alpha'(t)=(j,H_\alpha) \\ \omega'(t)\neq\text{JoinPoint}(\text{c},H_\delta) \quad i(c,H_\delta)=(d,H'_\delta) \\ \text{read\_heap}(i,c,H'_\delta)-H_\delta=r_H \end{array}}{\delta(t)=(n+1,d,H'_\delta),\ \alpha(t)=(j,H_\alpha\cup r_H)}$$

$$\textsc{commit}\ \dfrac{\begin{array}{ccc} (p,s)\in R' & & \delta'(s)=(n,x,H_s) \\ \omega'(p)=(y,S_p) & \delta'(p)=(m,y,H_p) & \text{JoinPoint}(H_p,y)=S_p \\ \alpha'(p)=(r,H_{p\alpha}) & \alpha'(s)=(y,H_{s\alpha}) & H_{s\alpha}\subseteq H_p \end{array}}{\delta(p)=(n+m,x,H_s\sqcup H_p),\ \omega(p)=\omega'(s),\ \alpha(p)=(r,H_{p\alpha}\sqcup H_{s\alpha})}$$

Figure 2. Rules used in constructing the speculation list. List operations and identity relations are excluded for simplicity.

The proof proceeds by showing that a given thread computation is a correct execution of the program, at least given its presumed inputs. The following lemma shows this property is preserved under all transformations,

Lemma 2.4. *For all $s\in V$, if $\alpha(s)=(a,H_\alpha)$ and $\delta(s)=(n,b,H_\delta)$, then s has performed the computation and derived correct result: $(b,H_\delta)=P_a|_n(H_\alpha)$.*

Proof. This property is true initially. FORK initializes this property for speculative threads, and both FORK and ABORT operations trivially preserve it.

- OP A regular operation composes a function i onto the existing computation; assuming the computation prior to application was $P_a|_{n-1}$ this constructs $i\circ(P_a|_{n-1})=P_a|_n$. The property thus holds provided the input to i is correct. The existing output of $P_a|_{n-1}$ is correct by assumption, so a contrary argument implies that following an OP there exists a pair (w,v) such that $P_a|_n(\alpha(s)\sqcup\{(w,v)\})\neq P_a|_n\alpha(s)$. Such a (w,v) could only have an impact on the input of i if it was not already found in either (the state of) $\delta'(s)$ or in the existing input state $\alpha'(s)$. By construction, however, $\alpha(s)=\alpha'(s)\cup r_H$ where r_H are exactly such reads, forming a contradiction.
- COMMIT Inductively, p computes $P_r|_m(H_{p\alpha})$ and s computes $P_y|_n(H_{s\alpha})$. Assume that after the COMMIT $\delta(p)\neq P_r|_{m+n}(H_{p\alpha})$. In order for this to be true the input to s, $H_{s\alpha}$, must represent a different environment from that found in $\delta'(p)$. Our contrary assumption then becomes that $H_{s\alpha}\sqcup H_p\neq H_p\sqcup H_{s\alpha}$. Let (v,m_s) and $(v,m_p), m_s\neq m_p$ be pairs in $H_{s\alpha}$ and H_p respectively. By construction in rule COMMIT $H_{s\alpha}\subseteq H_p$, and so both $(v,m_s),(v,m_p)\in H_p$. This contradicts Lemma 2.2.

The property is thus an invariant under all rule operations. □

From Lemma 2.4 it is relatively easy to show that the computation of a single thread corresponds to sequential computation of the same code, given the same inputs.

Lemma 2.5. *Let P be the code executed under speculative execution, started by non-speculative user thread r at code position 0 and begun in (fully specified) state I. Once r cannot apply any of the rules in Figure 2 r has computed $P_0|_x(I)$, for x maximal.*

Proof. Since I is fully specified r_H is always empty for r, and $\alpha(r)=(0,I)$ for the root non-speculative thread r is trivially preserved by every transformation. By Lemma 2.4 r always computes $P_0|_n(I)$ for some n. Since the ABORT rule can be applied to any speculative thread at any time, the inability of r to apply any rules means no other threads exist. Thus by Lemma 2.3 $\omega(r)=(\text{EOF},\emptyset)$, Since r cannot FORK or OP r has completed as many operations as it can, and m is maximal. □

From this it is straightfoward to establish an equivalence between sequential and MLS execution.

Theorem 2.6. *Let P be a sequential program. Execution of P under MLS is equivalent to sequential execution.*

Proof. This follows immediately from Lemma 2.5. Since ABORT may be invoked immediately after FORK, effectively generating a sequential execution, all MLS executions of P are equivalent to sequential. □

2.2 Multithreaded Programs

When executing a program that is already multithreaded each non-speculative thread may have its own speculative thread hierarchy; MLS execution of a multithreaded programs is thus easily modeled in our formalism by a set of independent speculation lists. The use of shared data and thread communication primitives, however, adds additional complexity. For each such unsafe primitive the OP rule of Figure 2 must be extended to explicitly describe how shared data is propagated to other non-speculative threads, and naturally strongly depends on the underlying thread communication and shared memory consistency models.

Following recent memory models specifications for Java [16] and C++ [4], here we show correctness for the most useful model of *correctly synchronized* programs, ones guaranteed free of race conditions and which may thus demonstrate only sequentially consistent behaviours. For this we also assume that all operations that imply direct communication between non-speculative threads are unsafe operations—this includes explicit thread control and synchronization, as well as access to shared variables outside of synchronization (*volatile* data). Note that we assume OP has been suitably extended to correctly define any such communication primitives, and if support for speculative commits is desired, suitable mutual exclusion between concurrent COMMIT and/or ABORT operations on the same thread. Speculative list operations are assumed atomic in all cases.

In such a context the equivalence of MLS to non-MLS execution is conceptually straightforward, building on the observation that speculation within one thread is always confined to segments of code that have no visibility to other non-speculative threads.

Lemma 2.7. *Let $\mathcal{P}$ be a correctly synchronized program executing under method-level speculation, and let s be a speculative thread in the speculation list of non-speculative thread r. Let w:*`x=v` *be a variable write performed by s. If t is a non-speculative thread $t\neq r$ then t cannot observe w until after s is fully committed by r.*

Proof. Assume to the contrary; writes of s are fully buffered until commit, and thus the observation can only occur during the actual commit of s by r. By Lemma 2.5 the computation of s properly composes with r as a continuation of the code and input to r. Since s executes no unsafe operations, including thread communication of any form, if s performs w then w is reachable without encountering synchronization in an execution of r without speculation. If this is possible then a scheduling of r,t exists that would produce a race condition between t and r, contradicting the presumed property of $\mathcal{P}$ being correctly synchronized. □

Lemma 2.7 establishes that speculation does not induce race conditions where none exist. Combined with Theorem 2.6 this allows us to conclude the equivalence also exists for multithreaded programs.

Theorem 2.8. *Let $\mathcal{P}$ be a correctly synchronized program, executed under MLS. An observable execution state of $\mathcal{P}$ is a set of non-speculative thread states prior to any partially completed* COMMIT *operations. If H is an observable state of $\mathcal{P}$ started with input I, then a non-MLS execution exists for the same input I reaching the same state H.*

Proof. Assume to the contrary and let H be first observable state generated by MLS execution that is not reachable as some non-MLS state. Observable state consists of threads, their code-positions and heaps/stacks, and assuming sequentially consistent execution, H is generated by applying some rule to a state H'. The ABORT and FORK rules do not affect non-speculative state, and OP actions change state deterministically. By Lemma 2.7 COMMITs are invisible to other threads, and by Theorem 2.6 speculative execution correctly performs single-threaded execution. A COMMIT by a non-speculative thread is visible to itself, creating intermediate execution states different from those that would be generated by the expected program order, but in the absence of committing unsafe operations these operations affect only the committing thread, and partial commit states are excluded by assumption. Thus H could only not be reachable from a non-MLS execution if H' is an unreachable state. This contradicts the assumption that H is the first such unreachable state. □

2.3 Discussion

The speculation list model supports a variety of MLS model parameters, unifying several potential implementation strategies for both method-level and more broadly-defined thread speculation. By allowing FORK operations by arbitrary threads the basic model describes fundamental approaches to both in-order and out-of-order MLS: speculative threads can start further speculation, and all threads can have multiple speculative children, respectively.

Implementation of either strategy is typically simplified by allowing only non-speculative threads to COMMIT or ABORT. This is particularly simple in out-of-order designs, where FORK is already specialized to only non-speculative threads. In this case no extra synchronization is required by MLS other than suspension of the speculative thread while it is actually committed (and protection of internal MLS data structures) [19]. In-order strategies bring a minor additional complexity in the potential for overlapping COMMIT and FORK operations. If speculative COMMITs (and ABORTs) are permitted, however, synchronization is further required between the multiple COMMIT, FORK and ABORT operations that may be performed on the same thread. The greater synchronization overhead of speculative commit/abort should in practice be balanced with the benefit provided by offloading actual commit operations to otherwise idle speculative threads, as well as any improvement brought by different speculative execution patterns (see Section 4). Note that, as shown in Lemma 2.7, in all cases the actual writes issued by a COMMIT need not be atomic.

MLS designs also vary in how input data required by speculative threads is retrieved. Speculative input may be retrieved optimistically from a parent or global environment, more pessimistically by arranging synchronization between speculative child and parent to correctly propagate data [24], or through (return) value prediction [14]. Our design supports a flexible validation approach, and so is independent of how speculative input data are gathered. Data acquired optimistically are verified as part of the COMMIT rule in Figure 2. Data acquired pessimistically are certainly correct and thus may be modeled by omitting or assuming the clause $H_{s\alpha} \subseteq H_p$ in rule COMMIT is always true.

In our formalism we use the JoinPoint operator to specify matching execution points (states) between parent termination and speculative child start. For MLS, and assuming method invocations may access only local and global scopes, matching execution points are easily identified by comparing stack pointer values: once a parent thread returns to the frame in which the child was forked the matching join point has been reached. The JoinPoint operator is in fact easily extended to allow loop-based [23] and arbitrary [3] thread speculation models. For loop-based speculation JoinPoints are defined as loop index and stack pointer values, and the code offset of the loop header. Arbitrary speculation may require more complex state depending on the model, although basic intra-method speculation requires only code and stack pointer values.

The list structure used in our formalism implies each thread has only one speculative successor at any one time. If CPUs are available and thread validation uncertain, it may be advantageous to instead fork multiple speculative threads at the same time, each evaluating different scenarios, and thus increasing the likelihood of a valid commit. This can be modeled by providing a FORK$_n$ rule that creates multiple speculative threads from the same fork point, along with extending our speculation list structure to a DAG. Other operations, COMMIT, ABORT and OP, continue to apply unaltered.

3. Stack Abstraction

In the previous section both speculative heap and stack states were abstracted by the same state variable. Here we present a stack abstraction that directly exposes the call stack manipulations central to MLS designs. This abstraction is flexible and supports in-order nesting, out-of-order nesting, in-order speculative commits, and any combination thereof. Specific models that implement these features using our abstraction are developed in Section 3.2.

In addition to FORK, COMMIT, and ABORT operations from the previous section, the stack also supports PUSH and POP operations to allow for method entry and exit. As in non-speculative programs, these new operations manipulate frames on the stack, such that a given frame contains the local variables of a method. Register values are assumed to be spillable to a frame on demand. Under this abstraction, FORK can be called instead of PUSH, pushing a frame and creating a new child thread, and upon return COMMIT or ABORT will be called to match the FORK instead of POP. Note that we assume stack operations complete atomically. Non-stack operations are not explicitly modelled and are assumed to be freely interleaved with stack operations on running threads. Speculative accesses to global variables are also handled externally.

The model has two unique features that separate it from naïve speculation where all reads and writes go through a dependence buffer or transactional memory subsystem. First, child threads buffer stack frames from their less-speculative parents, such that all local variable accesses go directly through a local frame. This is intended to reduce the load on the dependence tracking system. Second, stack frames are buffered as lazily as possible: on forking, only the frame of the current method is copied to the child. If the child ever needs lower down frames from some parent thread, these will be retrieved and copied on demand. This lazy copying introduces significant complexity: the POP operation may need to buffer a frame, and the COMMIT operation needs to copy back only the range of live frames from the child thread stack. We include it as a practical measure intended to make our abstraction useful, based on our experience with implementing MLS that indicates a steep performance penalty for copying entire thread stacks.

The main abstraction is described via its operational semantics in Figure 3. It has seven publicly available operations, each marked with $[*]$. These in turn use a number of internal operations, both for purposes of clarity and for logic reuse. A summary of the public operations and their observable behaviour follows:

START$()$: create a new non-speculative thread with an empty stack.

STOP(t): destroy non-speculative thread t with an empty stack.

PUSH(t, f): add a new frame with unique name f to the stack of thread t.

POP(t): remove the top frame from the stack of thread t. The matching operation must be a PUSH, and for speculative threads there must be a frame to pop to.

FORK(t, f): fork a new child thread that starts executing the method continuation in the current frame of thread t and then execute PUSH(t, f). Cannot be issued on an empty stack.

ABORT(t): execute POP(t) and then abort the child thread attached to the frame underneath, recursively aborting all of its children. The matching operation must be a FORK.

COMMIT(t): execute POP(t) and then commit the child thread attached to the frame underneath, copying all of its live stack frames and any associated child pointers. Committed children with children of their own are kept on a list attached to t until no references to them exist, lest another speculative thread attempt to copy a stack frame from freed memory. The matching operation must be a FORK.

3.1 Detailed description

We now provide sufficient detail to understand the operations in Figure 3. We model threads as unique integers, and maintain several thread sets: T is the set of all threads, T_l represents live threads, T_n and T_s are non-speculative and speculative threads respectively, and T_c is the set of committed threads that may still be referenced by some $t \in T_s$. Some invariants apply to these sets: $T_n \cup T_s \supseteq T_l$, $T_n \cap T_s = \emptyset$, $T_n \neq \emptyset$, $T_c \cap T_l = \emptyset$, and $T_c \cup T_n \cup T_s = T$. Elements are never removed from T, such that each new thread gets a unique ID based on the current size of T, namely $|T|$. Stack frames are modeled by a set of unique frames F.

In addition to these sets, there are several functions that maintain mappings between them. $stack(t \in T)$ maps t to a thread stack, $child(f \in F)$ maps f to a speculative child thread u, $parent(u \in T_s)$ maps u to the $t \in T$ that forked it, and $commits(t \in T)$ maps t to a list of threads in T_c. Initially all mappings and sets are empty.

Our rules make use of a few specific operators and conventions. The use of exclusive or ($\oplus$) indicates a choice between one rule and another or one set of premises and another, and we use $S \uplus \{s\}$ (or $S \ \backslash\!= \ \{s\}$) to indicate set additions (or removals). Given a frame f', f is the less-speculative version of the same frame in a parent thread. Finally, ν is the greatest fixed point operator from the μ-calculus that maximizes its operand starting from $\top$ given some terminating condition. ν is used to find the greatest, or "most speculative," thread p in POP, and to find the longest sub-list δ without child dependences in PURGE_COMMITS.

Lastly, we give a brief description of each rule. CREATE initializes a new thread u, adds it to T_l, T, and T_n or T_s depending on whether the thread is speculative or not, as given by T_p, and initializes the stack of u to σ. DESTROY conversely removes u from T_l and either T_n or T_s. Note that after destroy, committed threads will move to T_c, from which they are later removed only by PURGE_COMMITS. START simply calls CREATE to make a new non-speculative thread u, and STOP calls DESTROY to remove it.

PUSH takes a fresh f and appends it to $stack(t)$, where t is live, also adding f to F. BUFFER takes either a live or committed thread, the name of a frame e in its stack, and provided there is no child attached to e creates e' for use by its caller, which is either FORK or POP. FORK first calls PUSH, buffers e' from e, creates u, and sets u as e's child and t as u's parent.

$$\textsc{create}(T_p, \sigma)\ \frac{T_p = T_n \oplus T_p = T_s}{u = |T|, T \uplus \{u\}, T_l \uplus \{u\}, T_p \uplus \{u\}, stack(u) = \sigma}$$

$$\textsc{destroy}(u)\ \frac{T_p \subseteq T \ .\ u \in T_p \quad T_p = T_n \oplus T_p = T_s}{T_l \ \backslash\!= \{u\}, T_p \ \backslash\!= \{u\}}$$

$$[*]\textsc{start}()\ \frac{}{\textsc{create}(T_n, \emptyset)}$$

$$[*]\textsc{stop}(u)\ \frac{u \in T_n \quad stack(u) = \emptyset}{\textsc{destroy}(u)}$$

$$[*]\textsc{push}(t, f)\ \frac{t \in T_l \quad f \notin F \quad \sigma = stack(t)}{stack(t) = \sigma : f, F \uplus \{f\}}$$

$$\textsc{buffer}(t, e)\ \frac{t \in T_l \cup T_c \quad e \in stack(t) \quad e \in F \quad child(e) \notin T_s}{e' = e, F \uplus \{e'\}}$$

$$[*]\textsc{fork}(t, f)\ \frac{\dfrac{\textsc{push}(t, f)}{\sigma : e : f = stack(t), \textsc{buffer}(t, e)} \quad \textsc{create}(T_s, e')}{parent(u) = t, child(e) = u}$$

$$[*]\textsc{pop}(t)\ \frac{\dfrac{t \in T_l \quad \sigma : e' : f' = stack(t) \quad f' \in F \quad child(e') \notin T_l}{\dfrac{t \in T_n}{e' \in F \oplus e' \notin F} \oplus \dfrac{t \in T_s}{e' \in F \oplus \nu p_{p \geq 0}\ .\ \textsc{buffer}(p, e)}}}{stack(t) = \sigma : e'}$$

$$\textsc{join}(t)\ \frac{t \in T_l \quad \sigma : e : f = stack(t) \quad e, f \in F \quad child(e) \in T_l}{stack(t) = \sigma : e, u = child(e)}$$

$$\textsc{merge\_stacks}(t, u)\ \frac{\dfrac{d' : \rho = stack(u)}{\dfrac{d \in stack(t)}{\sigma : d : \pi : e = stack(t)} \oplus \dfrac{d \notin stack(t)}{\sigma = \emptyset}}}{stack(t) = \sigma : d' : \rho}$$

$$\textsc{merge\_commits}(t, u)\ \frac{\gamma = commits(t) \quad \delta = commits(u)}{commits(t) = \gamma : u : \delta, T_c \uplus \{u\}}$$

$$\textsc{purge\_commits}(t)\ \frac{\begin{array}{c}\gamma : \delta = commits(t)\ .\ \delta = \nu\delta_{\delta \supseteq \emptyset}\ .\\ \forall c \in \delta\ \forall f \in stack(c), child(f) \notin T_l\end{array}}{commits(t) = \gamma, T_c \ \backslash\!= \{\delta\}}$$

$$\textsc{cleanup}(t, u)\ \frac{\textsc{destroy}(u)}{\textsc{purge\_commits}(t)}$$

$$[*]\textsc{commit}(t)\ \frac{\begin{array}{c}\textsc{join}(t) \quad \textsc{merge\_stacks}(t, u)\\ \textsc{merge\_commits}(t, u)\end{array}}{\textsc{cleanup}(t, u)}$$

$$\textsc{abort\_all}(t)\ \frac{\dfrac{\forall f \in stack(t)\ .\ u = child(f) \in T_l}{\textsc{abort\_all}(u)}}{\textsc{cleanup}(t, u)}$$

$$[*]\textsc{abort}(t)\ \frac{\textsc{join}(t) \quad \textsc{abort\_all}(u)}{\textsc{cleanup}(t, u)}$$

Figure 3. *Stack operations.* Externally available operations are marked with $[*]$. START and STOP create and destroy non-speculative threads, PUSH, POP, FORK, COMMIT, and ABORT operate on existing threads, and all other operations are internal.

POP takes the stack of t, and checks that the top frame f' is valid and there is no child attached to the frame e' underneath. If t is non-speculative, it does not matter if e' exists, we can always pop f. If the t is speculative, either e' exists and is found in $stack(t)$, or due to our lazy stack copying approach it needs to be retrieved and buffered from the most speculative parent thread p that contains e. JOIN has similar conditions to POP, except that here e must exist and also have a speculative child.

MERGE_STACKS is called by COMMIT, and is used to unify the (partial set of) frames copied into a child with those found in a parent thread. It takes the stack of u and looks for a less-speculative version d of its bottom frame d' in its parent. If found, then frames d through e in t are replaced with the child stack, otherwise the entire parent stack is replaced. Note that d will always be found if $t \in T_n$, since non-speculative threads have full stacks. MERGE_COMMITS, as called by COMMIT, takes the commit list γ from the parent and appends the child u and the child commit list δ, and adds u to T_c. PURGE_COMMITS is called every time CLEANUP is called, and it removes threads without child dependences from the most speculative end of a commit list until either all committed threads have been purged or it encounters a dependency.

CLEANUP simply destroys u and then purges t, and is called after any COMMIT or ABORT operation, and internally from ABORT_ALL. It contains the common logic that follows commits and aborts, whereas JOIN contains the common logic that precedes them. COMMIT is just a composite operation that joins t, merges stacks and commit lists using u from JOIN, and then cleans up. ABORT has a similar structure, calling ABORT_ALL internally, which performs a depth-first search looking for live children, and destroying them post-order. In any real implementation that uses this stack abstraction, child threads must be stopped before they can be committed or aborted.

3.2 Structural operational semantics

Using the stack abstraction from Figure 3, we now develop a series of progressively more flexible behaviour models, shown individually in Figures 4–10. We then use these rules in Section 4 to explore and understand the behaviour of various code idioms under speculation.

In these models, each rule is named, possibly with symbols for PUSH ($\downarrow$), POP ($\uparrow$), FORK ($\prec$), COMMIT ($\succ$), and ABORT ($\nsucc$). Above the inference line is the corresponding $[*]$ command from Figure 3, followed by restrictions on behaviour and local variable mappings. Below the line is a visual depiction of the transition from one stack state to the next. Threads are named in increasing order by τ, α, β, γ, δ, such that $\tau \in T_n$ and $\{\alpha,\ldots,\delta\} \subseteq T_s$, except in rule I$\uparrow\perp$ from Figure 8, where τ may be in T_s. Shown for each thread t is the value of $stack(t)$, which grows upwards, the value of $commits(t)$, which grows left-to-right starting at t, and for each $f \in stack(t)$ a horizontal line to the child thread if $child(f) \in T_l$. As in Figure 3, variables d, e, f and derivatives are given to concrete frames, whereas $\sigma, \rho, \pi, \omega, \varphi, \upsilon$ range over frames. A valid speculation for a given program is described and can be visualized by a sequence of rule applications, each of which acts atomically.

Figure 4 contains a simple structured non-speculative stack model common to many languages. Non-speculatives threads can START and STOP, delimiting the computation. In N$\downarrow$, a new frame can be pushed, where $\sigma \subseteq F$ and so may be $\emptyset$. N$\uparrow$ and N$\uparrow\perp$, match the two cases $e \in stack(\tau)$ and $e \notin stack(\tau)$ of POP(t) in Figure 3, the latter being the penultimate operation on a thread, followed by STOP.

Figure 5 contains the simplest MLS stack model, one that extends Figure 4 to allow non-speculative threads to fork and join a single child at a time. In this model, speculative threads cannot perform any operations, including simple method entry and exit. For N$\prec$, there is a restriction on children being attached to prior stack frames, which prevents out-of-order speculation. N$\succ$ is the simplest COMMIT(t) possible, with the child stack containing only one frame, and N$\nsucc$ is similarly simple with no recursion required in ABORT(τ). Finally, the restriction $\tau \in T_n$ in N$\downarrow$ and N$\prec$ is sufficient to prevent speculative child threads from doing anything other than local computation in the buffered frame e': N$\succ$ and N$\nsucc$ must match with N$\prec$, N$\uparrow$ must match N$\downarrow$, and N$\uparrow\perp$ is precluded for speculative threads because BUFFER(τ, e) will not complete.

The model in Figure 6 extends Figure 5 to allow speculative children to enter and exit methods. A speculative push S$\downarrow$ simply creates a new frame for α, specifying that π' is linked to π via some frame e' at the bottom of π' to the corresponding $e \in \pi$. S$\uparrow$ takes the left-hand case in POP(t) where $e' \in F$, whereas S$\uparrow\perp$ takes the right-hand case and so buffer e' from its parent. Finally, this model updates N$\succ$ and N$\nsucc$ to handle situations where the child may have left e' via S$\uparrow\perp$, represents the child thread stack by φ'.

The next model in Figure 7 simply adds one operation to allow out-of-order nesting in non-speculative threads, O$\prec$. This rule specifies that if there is some lower stack frame d in π with a child attached, a new thread can be forked from e, complementing N$\prec$ in Figure 5 which prohibits this. All other existing operations continue to work as expected in this model. As an implementation note, this model is relatively straightforward to implement in software, but offers significantly limited parallelism [19].

After out-of-order nesting comes in-order nesting in Figure 8. I$\prec$ allows speculative thread α to create β independently of its parent. N$\nsucc$ will recursively abort these threads without modification, but I$\succ$ is required to allow a parent thread to commit child thread α with a grandchild β, maintaining the link to β and merging α onto the commit list of the parent. After β gets committed via N$\succ$, α will be freed, assuming there are no more children. I$\uparrow\perp$ is yet more complex, specifying that in order to buffer frame e', parent threads will be searched backwards starting from the grandparent until e is found. Here $\leadsto$ indicates that there is a path of buffered frames from π' backwards to π. This rule is an extended version of S$\uparrow\perp$, which only handles buffering from the immediate parent. S$\uparrow$ works nicely as is with in-order speculation, and S$\uparrow\perp$ works not only in the simple case above but also in the case where the buffered frame is in some committed thread $c \in T_c$.

In Figure 9, speculative commits are now permitted. There are two simple rules, S$\succ$ and SI$\succ$, which complement N$\succ$ and I$\succ$ respectively. In the former β is purged from $commits(\alpha)$, whereas in the latter it is kept because of dependency γ. [I$\succ$MERGE] is implied by I$\succ$, and so adds nothing, and only shown to illustrate the full process of merging committed thread lists, where α and γ were already committed and β gets added between them.

Finally, in Figure 10, the last restrictions are removed so that all of the features in the main abstraction in Figure 3 are available. In this case, it suffices to provide IO$\prec$, which allows speculative threads to create child threads out-of-order. This was formerly prohibited by O$\prec$, which only applied to non-speculative threads. The other two rules are again shown only for purposes of illustration: [IO$\nsucc$] shows a recursive abort on a thread with both in- and out-of-order nesting, and [OI$\prec$] shows in-order nesting after out-of-order nesting has taken place, as already allowed by O$\prec$ followed by I$\prec$.

The above models illustrate the behaviour of common speculation strategies. In the next section, we explore a series of stack evolutions that rely upon this final combined stack model for their behaviour.

4. Speculation Idioms

Simple changes to the structure of input programs and decisions regarding when to speculate can dramatically affect the dynamic

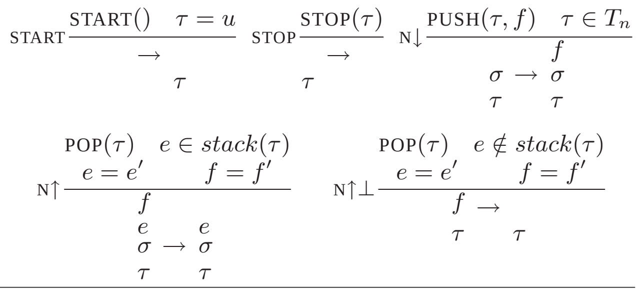
Figure 4. *Adults-only model.* No speculation.

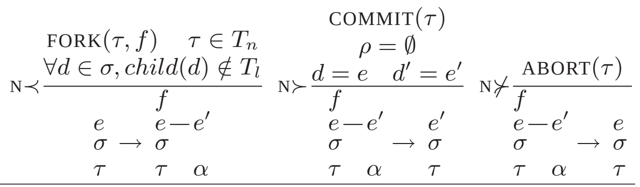
Figure 5. *Totalitarian model.* One speculative child allowed, but only non-speculative threads can perform stack operations.

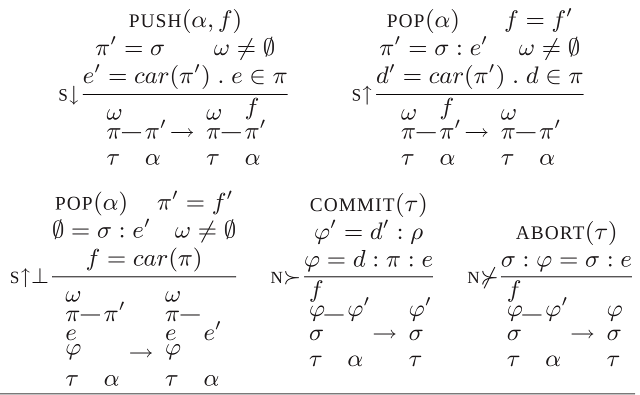
Figure 6. *Kid-friendly model.* Allows PUSH and POP actions on speculative threads, overriding N$\succ$ and N$\not\succ$ to accomodate.

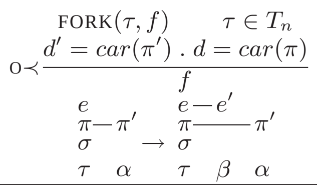
Figure 7. *Catholic model.* Provides out-of-order nesting via O$\prec$ to allow an arbitrary number of speculative children for non-speculative threads

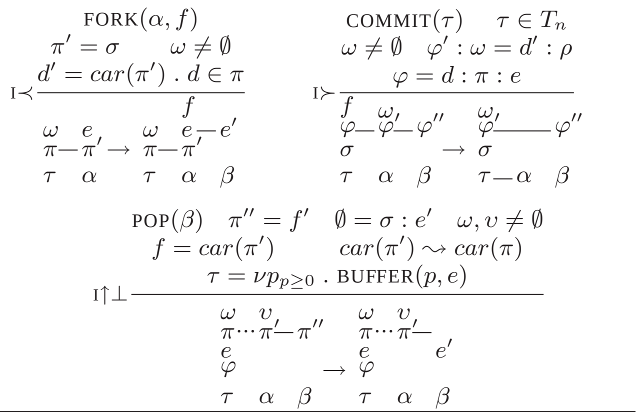
Figure 8. *One big happy model.* Provides in-order nesting via I$\prec$ to allow speculative children of speculative threads.

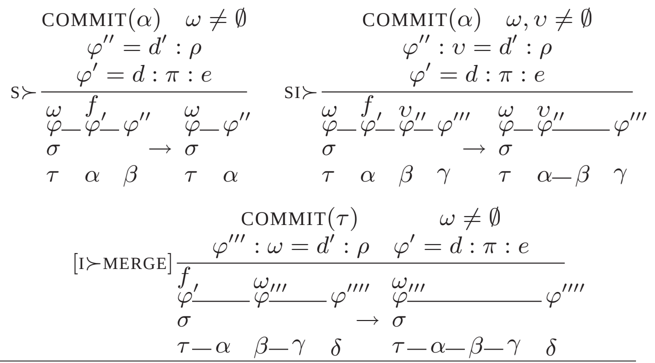
Figure 9. *Nuclear model.* Allows speculative threads to commit their own children. [I$\succ$MERGE]'s behaviour is provided by I$\succ$.

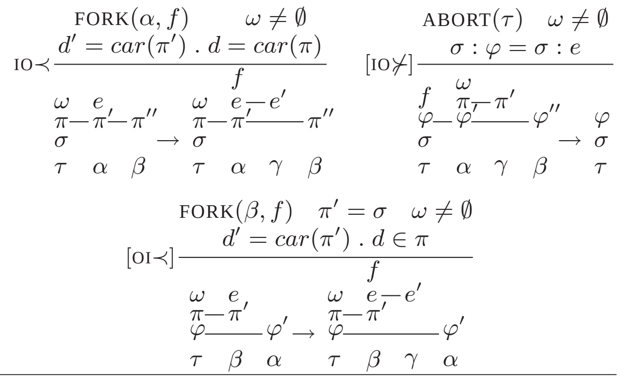
Figure 10. *Libertarian model.* Allows both in-order and out-of-order nesting. [IO$\not\succ$] and [OI$\prec$] are provided by N$\not\succ$ and I$\prec$.

structure of the speculative call stack. In this section we explore several common code idioms and their behaviour under MLS using our stack abstraction from Section 3. We examine straight-line code, simple if-then conditionals, and finally explore basic iteration and recursion in-depth, with a view towards discovering idiomatic code structures and speculation decisions that yield interesting parallel execution behaviours.

In the examples that follow, we assume that useful computation can be represented by calls to a `work` function whose running time is both constant and far in excess of the running time of all non-work computation. Thus we can reason that if a thread is executing a work function, it will not return from that function until all other computations possible before its return have completed. This reasoning will guide the stack evolutions in cases where more than one operation is possible. Although our execution timing assumptions are simplistic, the behaviour is still complex and interesting, and it provides a basis for understanding more complex situations.

4.1 Straight-line Code

The simplest code idiom in imperative programs is straight-line code, where one statement executes after the next without branching. In Figure 11, two sequential calls to `work` are shown, and the

```
straightline () {
  work (1);
  work (2);
}
```

Figure 11. *Straight-line code.*

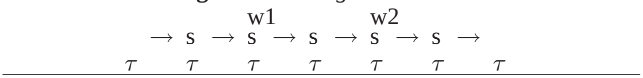

Figure 12. *Straight-line: do not speculate.*

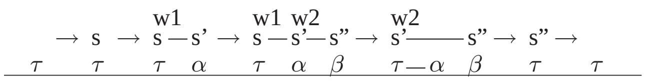

Figure 13. *Straight-line: speculate on all calls to* `work`.

```
straightline () {        stop () {
  work (1);                /* unsafe */
  work (2);              }
  stop ();
  work (3);
}
```

Figure 14. *Speculation barrier code.*

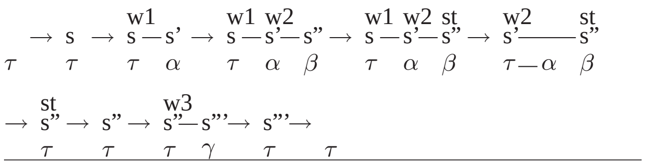
w1 w1 w2 w1 w2 st w2 st
→ s → s—s' → s—s'—s" → s—s'—s" → s'———s"
τ τ τ α τ α β τ α β τ—α β
st w3
→ s" → s" → s"—s"' → s"' →
τ τ τ γ τ τ

Figure 15. *Speculation barrier: speculate on all calls to* `work`.

non-speculative stack evolution is shown Figure 12. In Figure 13, speculation occurs on all calls to `work`: the parent thread τ executes `work(1)`, α executes `work(2)`, and β executes a continuation which does nothing useful. τ returns from w1 and commits α, then returns from w2 and commits β, and finally pops s" to exit the program.

Even in this simple example, choices about whether to use PUSH or FORK clearly affect which threads execute which regions of code, and whether all speculative threads have useful work to do. In Figure 14, a function `stop` is introduced that contains an unsafe operation that will act as a speculation barrier. The result in Figure 15 is that w3 is not executed speculatively. Again, although simple, the impact of unsafe instructions on speculative parallelism is important to consider; in some cases, artificial speculation barriers may even be helpful.

4.2 Conditional branching

Another simple code idiom is conditional branching. If the value of the conditional is speculative, then particular code paths followed depending on the value themselves become speculative. In Figures 16–22, speculating on the call to `work(1)`, it is necessary to predict a boolean return value. If the speculation is correct, as in Figures 19 and 20, then the speculative work w2 or w3 respectively ends up being committed, otherwise aborted.

For this speculation idiom to be useful, the function producing the return value should take a long time to execute. Further, extensions to the basic list and stack speculation models could allow for multiple predicted return values, with one speculative thread each. This would provide a kind of speculative hedging, and may be worthwhile given excess resources. Nested ifs have similar behaviour to this example, although the prediction for the outer test is more important than the inner test in terms of limiting wasted computation, since the inner speculation is under its control.

```
if_then () {
  if (work (1)) {
    work (2);
  }
  work (3);
}
```

Figure 16. *If-then code.*

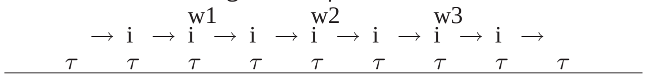

Figure 17. *If-then: do not speculate,* `work(1)` *returns true.*

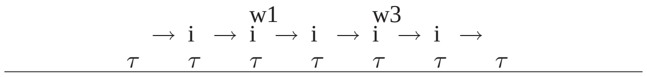

Figure 18. *If-then: do not speculate,* `work(1)` *returns false.*

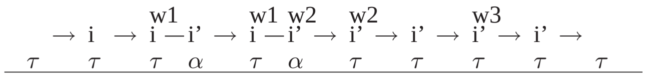

Figure 19. *If-then: speculate on* `work(1)`*, predict true correctly.*

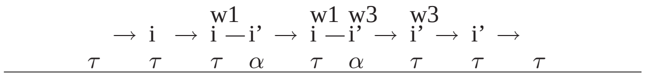

Figure 20. *If-then: speculate on* `work(1)`*, predict false correctly.*

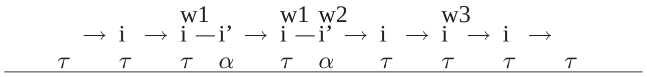

Figure 21. *If-then: speculate on* `work(1)`*, predict true incorrectly.*

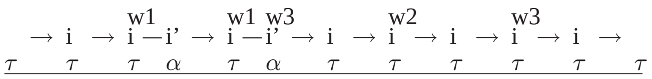

Figure 22. *If-then: speculate on* `work(1)`*, predict false incorrectly.*

4.3 Iteration

The most common code idiom considered for speculation is loop iteration. Chen & Olukotun [5] demonstrated that if a loop body is extracted into a method call, then method level speculation can subsume loop level speculation. We explore an example loop under different speculation assumptions in Figures 23–29 to better understand the behaviour. Speculating on all calls to `work`, the loop is quickly divided up into one iteration per thread, for as many threads as there are iterations.

In many cases loop bodies may be small, and speculating on every m in n calls/iterations may be more appropriate. In Figure 26 speculation is performed on every 1 in 2 calls. In this case the stack evolves to a point where both w1 and w2 are executing concurrently and no other stack operations are possible. From then on, although there are a number of possible intermediate evolutions they all lead to the same w3/w4 state. Effectively, the loop is parallelized into two threads, each executing one iteration at a time. Speculating on every 1 in 3 calls, a similar pattern emerges, except that a non-parallel execution of w3 is interjected. Speculating on every 2 in 3 calls, w1, w2, and w3 execute in parallel, and once they complete the stack evolves until w4, w5, and w6 execute in parallel.

A general rule for iteration under MLS then is that speculating on every $n-1$ in n calls to `work` will parallelize the loop across n threads, each executing one iteration. To support multiple subsequent iterations executing in the same thread, there are two options: 1) pass `i + 2` when speculating, which is not directly possible with our stack model; 2) unroll the loop and push multiple iterations into the loop body, as in Figure 29.

```
iterate (n) {
  for (i = 1; i <= n; i++) {
    work (i);
  }
}
```

Figure 23. *Iteration code.*

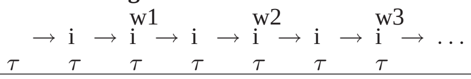

Figure 24. *Iteration: do not speculate.*

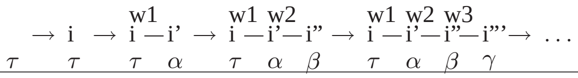

Figure 25. *Iteration: speculate on all calls to* `work`.

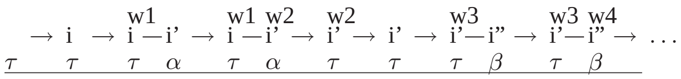

Figure 26. *Iteration: speculate on 1 in 2 calls to* `work`.

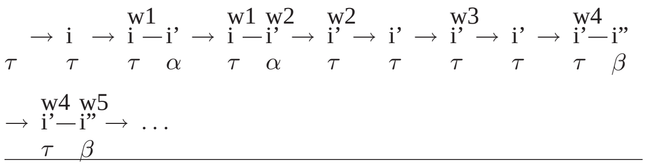

Figure 27. *Iteration: speculate on 1 in 3 calls to* `work`.

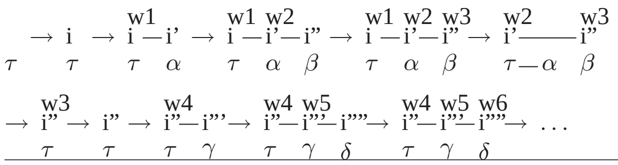

Figure 28. *Iteration: speculate on 2 in 3 calls to* `work`.

```
iterate (n) {            unrolled (i) {
  i = 1;                   work (i);
  while (i <= n) {         i++;
    unrolled (i);          work (i);
  }                        i++;
}                        }
```

Figure 29. *Unrolled iteration code.*

4.4 Tail Recursion

Tail recursion is explored in Figure 30–37. It is well known that tail recursion can be efficiently converted to iteration, and we see the same behaviour in these examples. Speculating on both `recurse` and `work` does populate the stack with successive calls to `work`, but it also creates just as many useless threads that will only ever fall out of the recursion, although they will stop almost immediately as they encounter elder siblings. Speculating on just `work` is good, and yields a stack structure identical to that produced by speculating on all calls in iteration, as in Figure 25, once the interleaving `recurse` frames are removed. On the contrary, speculating on just `recurse` is bad, such that calls to `work` are never parallelized.

Speculating on 1 in 2 calls to `work` yields again a structure directly comparable to iteration, where w1/w2 will execute in parallel before the stack evolves to w3/w4. Speculating on 1 in 2 calls to `work` and `recurse` is similar but more wasteful. Speculating on 1 in 2 calls to `recurse` is bad, but yields an interesting behaviour where the speculative children unwind the stack by one frame before stopping.

```
recurse (i, n) {
  work (i);
  if (i < n) {
    recurse (i + 1, n);
  }
}
```

Figure 30. *Tail recursion code.*

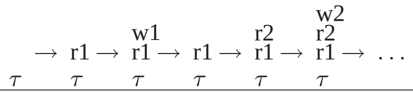

Figure 31. *Tail recursion: do not speculate.*

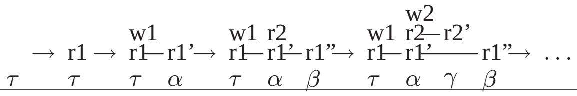

Figure 32. *Tail recursion: speculate on all calls (inefficient).*

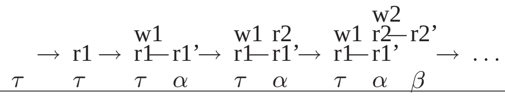

Figure 33. *Tail recursion: speculate on all calls to* `work` *(good).*

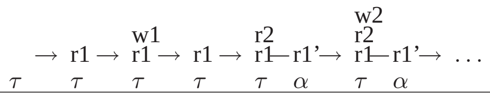

Figure 34. *Tail recursion: speculate on all calls to* `recurse` *(bad).*

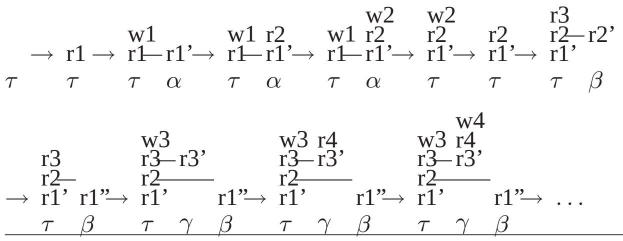

Figure 35. *Tail recursion: speculate on 1 in 2 calls to* `work` *and* `recurse` *(inefficient).*

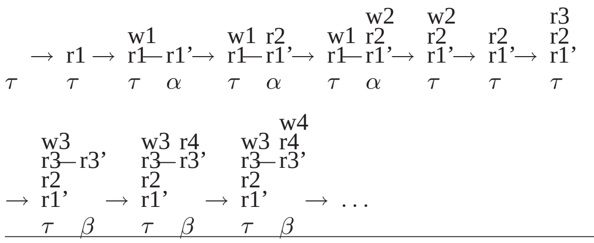

Figure 36. *Tail recursion: speculate on 1 in 2 calls to* `work` *(good).*

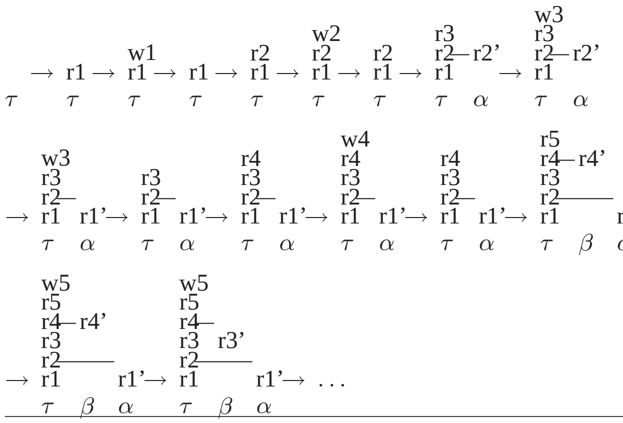

Figure 37. *Tail recursion: speculate on 1 in 2 calls to* `recurse` *(bad).*

```
recurse (i, n) {
  if (i < n) {
    recurse (i + 1, n);
  }
  work (i);
}
```

Figure 38. *Head recursion code.*

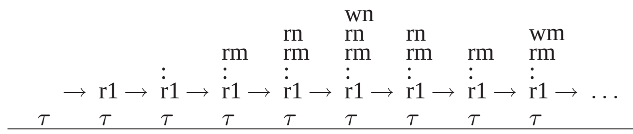

Figure 39. *Head recursion: do not speculate.*

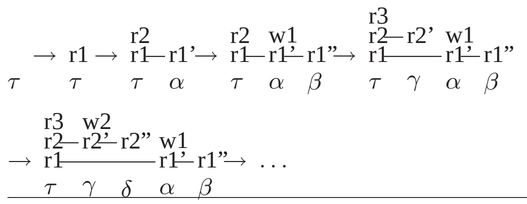

Figure 40. *Head recursion: speculate on all calls (inefficient).*

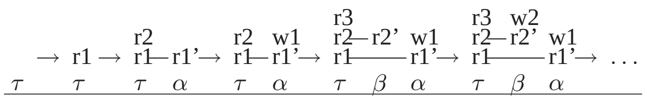

Figure 41. *Head recursion: speculate on all calls to* `recurse` *(good).*

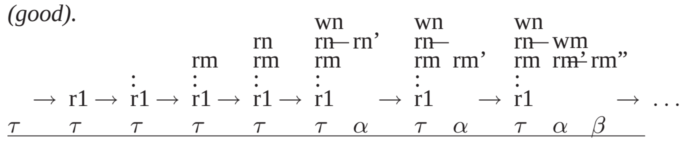

Figure 42. *Head recursion: speculate on all calls to* `work` *(good).*

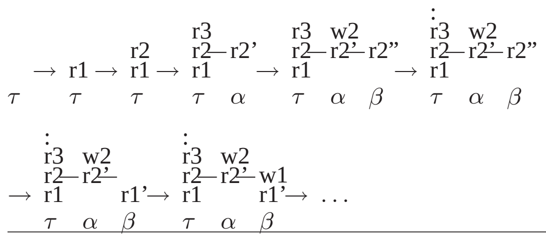

Figure 43. *Head recursion: speculate on 1 in 2 calls to* `recurse` *and* `work` *(unbounded parallelism).*

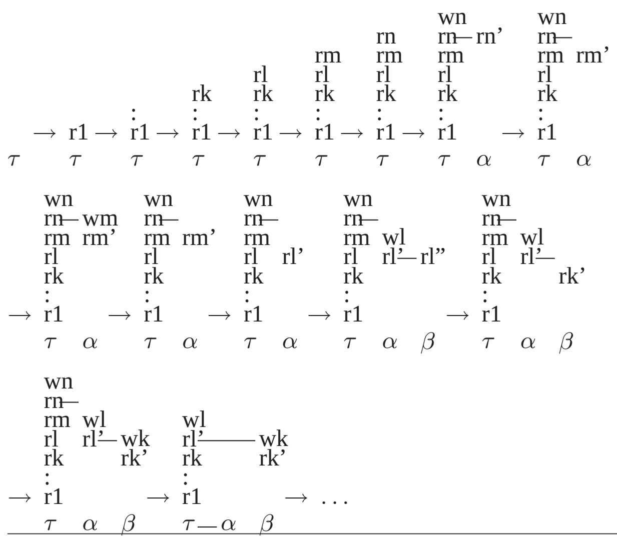

Figure 44. *Head recursion: speculate on 1 in 2 calls to* `work` *(compare with Figure 36).*

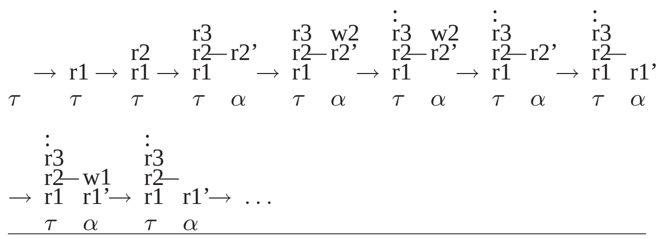

Figure 45. *Head recursion: speculate on 1 in 2 calls to* `recurse` *(loop unrolling).*

4.5 Head Recursion

Head recursion is considered in Figures 38–45. Here the call to `work` comes after the call to `recurse` instead of before. Speculating on all calls is inefficient, just as for tail recursion, whereas speculating on just `recurse` is good, allowing for calls to `work` to be executed out-of-order. This is expected given that head recursion is seen as dual to tail recursion. However, surprisingly, speculating on just `work` is also good: the stack gets unwound in-order. For head recursion, the support for in-order nesting and out-of-order nesting support in our stack model helps ensure that parallelism is obtained.

Speculating on 1 in 2 calls to `recurse` and `work` yields unbounded parallelism, where pairs of two calls are unwound in-order within a pair, and out-of-order between pairs. Speculating on 1 in 2 calls to `work` yields a bounded parallelism structure comparable to iteration, where first wn and wm will execute in parallel, and then the stack will evolve so that wl and wk execute in parallel.

We were again surprised by speculating on 1 in 2 calls to `recurse`: α executes w2, and after returning the stack evolves until it is executing w1. This pattern is strikingly similar to loop unrolling, where two successive calls execute in the same thread. This particular example is unbounded, however, because nothing prevents the growth of τ up the stack, such that every two calls to `work` will start all together, and then be unrolled all together. In general, calls to work can be divided into batches of size $b = n/t$ and distributed evenly across threads by choosing to speculate on every 1 in b calls to `recurse`. The unrolling within a given batch is in-order, but the creation of batches themselves is out-of-order.

4.6 Mixed Head and Tail Recursion

Finally, we experimented with a mix of head and tail recursion, as in Figures 46–49. Given the interesting behaviours seen in isolation in Sections 4.4 and 4.5, it seemed reasonable that a combination might yield even more interesting results. Tail recursion has two distinguishing properties under speculation: it provides in-order distribution across threads, and it prevents the calling thread from proceeding immediately to the top of the stack because useful work must be done first. On the other hand, head recursion is able to provide behaviour comparable to loop unrolling in a single thread. However, head recursion is uncapped and will always proceed immediately to the top of the stack.

Figures 46 and 48 constitute a minimal example that uses head recursion to provide batch processing and tail recursion to limit stack growth. In Figure 48, the repeating pattern is two head recursive calls followed by two tail recursive calls, such that speculation only occurs on the first of the two tail recursive calls. This creates a thread α that executes the first two calls to `work` out-of-order, while the parent thread τ executes the second two calls to `work` in-order. Except during brief periods of stack state evolution, there will only ever two threads actively executing code.

We can use this pattern to schedule batches of size b across t threads when the depth of the recursion is unknown or when only $b \times t$ calls should be scheduled at once. We need a pattern

```
head1 (i, n) {          tail1 (i, n) {
  head2 (i, n);           work (i);
  work (i);               tail2 (i, n);
}                       }

head2 (i, n) {          tail2 (i, n) {
  tail1 (i, n);           work (i);
  work (i);               head1 (i, n);
}                       }
```

Figure 46. *Two head then two tail code.*

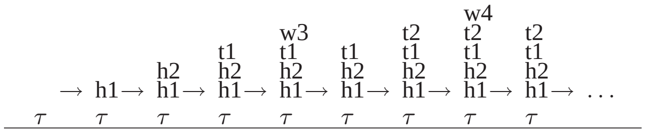

Figure 47. *Two head then two tail: do not speculate.*

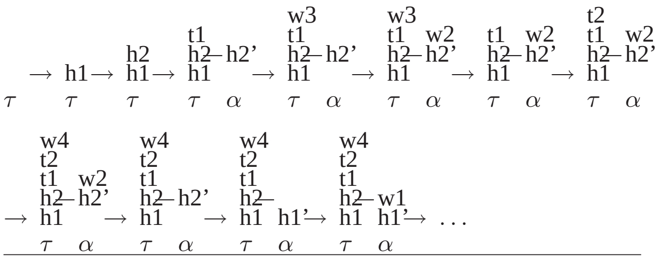

Figure 48. *Two head then two tail: call* `head1(1,n)` *and speculate on* `tail1` *in* `head2` *(creates two batches of two calls each).*

```
recurse (i, n, b, t) {
  if (i < n && (i - 1) % (b * t) < b * (t - 1)) {
    if (i % b == 1 && i % (b * t) > b) {
      spec recurse (i + 1, n, b, t);
    } else {
      recurse (i + 1, n, b, t);
    }
  }
  work (i);
  if (i < n && (i - 1) % (b * t) >= b * (t - 1)) {
    if (i % b == 1 && i % (b * t) > b) {
      spec recurse (i + 1, n, b, t);
    } else {
      recurse (i + 1, n, b, t);
    }
  }
}
```

Figure 49. *Mixed head and tail recursion code.* To split work into multiple threads, call `recurse (1,n,b,t)`, where `n` is the number of calls to `work`, `b` is the batch size, and `t` is the number of threads. Speculation points are indicated by the `spec` keyword.

of $b \times (t-1)$ head recursive calls followed by b tail recursive calls, speculating on the first tail recursive call in the pattern and on every $(cb+1)^{\text{th}}$ head recursive call for $c \in \mathbb{N}_1$. For example, to distribute work in batches of size 3 across 4 threads, use a pattern of 9 head recursive calls followed by 3 tail recursive calls, and speculate on the 4^{th} and 7^{th} head recursive calls and the first tail recursive call. A general function that provides this behaviour is shown in Figure 49.

4.7 Discussion

We can see from these examples that the dynamic parallelization behaviour induced by method level speculation is not obvious, and there are surely more interesting patterns to be found. We cannot take ordinary programs with call and return semantics, provide a set of parallelization operations that significantly perturbs the normal execution order, and expect to obtain dramatic performance results, especially if we do not understand the underlying behaviour. We can however use investigations of sequential program behaviour under our stack model to derive generalizations about program structure and the correlation with performance or lack thereof.

Method level speculation is a powerful system for automatic parallelization, particularly when relatively arbitrary speculation choices are permitted. The challenge is to restructure sequential code so that any inherent parallelism can be fully exploited. In general, parallel programming is an optimization, and thus cannot be divorced from knowledge of what different code structures imply for the runtime system if performance is to be maximed. Just as tail-recursion is favoured in sequential programs for its efficient conversion to iteration, so should other idioms in sequential programms be for their efficient conversion to parallel code. Of course, the end goal is for a compiler to remove this optimization burden from the programmer wherever possible via automation.

5. Related Work

We have proposed multiple abstractions of MLS in order to show correctness, equivalence, and precise implementation behaviour. These properties are naturally addressed in various approaches to speculative parallelism, with roots in models of parallel functional languages, and more recently in the context of proving correctness and performance of transactional memory.

The focus on method calls as a means to achieve parallelism in MLS suggests an affinity for functional language contexts, where the lack (or at least greater control) of method side-effects reduces implementation complexity. Functional MLS designs have been shown, e.g. in Haskell [13], although abstractions have more typically focused on other techniques, such as lazy evaluation, futures [26], and other optimistic methods [6]. Griener and Blelloch [9], for example, define a *parallel speculative λ-calculus* to help model performance and prove time efficiency, and equivalence of parallel lazy evaluation to sequential is shown by Baker-Finch *et al.* [2].

The strong isolation properties assumed by speculative threads also suggest a *software transactional memory* (STM) approach [22]. As with MLS, transactional designs differ in key choices as to when concurrent operations may be performed, how they may be nested, and visibility of intermediate calculations. *Transactional Featherweight Java*, for instance, is used to show serializability of both versioning and two-phase locking approaches to transaction control [15]. Other major differences exist in terms of transaction nesting and hence available parallelism. Harris *et al.*, provide a composable abstraction for Haskell, including support for one form of nested transactions, although with limited parallelism [12]. Moore & Grossman also use a small-step semantic approach, to investigate different nesting forms, showing equivalence between weaker models that enable greater parallelism, and using a type system to verify correctness in terms of progress of transactional substeps [17]. Abadi *et al.*, have a similar goal, also building a type-based approach to prove correctness. They develop a specialized calculus of automatic mutual exclusion, and use it to examine the impact of weak atomicity models. Guerraoui and Kapalka argue that *opacity* is a fundamental serialization criterion, and use that to show correctness, as well as complexity bounds [10]. MLS of course has fundamental differences from STM—speculative execution is not user-specified, and is potentially unbounded. Nesting models, however, have some similarity, and the correspondence between different MLS nesting strategies and weak transactional nesting would be interesting to explore.

5.1 MLS

MLS itself is a form of speculative multithreading (SpMT), which has been relatively well-studied from a hardware perspective and has been a subject of research for well over a decade. Garzaran *et al.* reviewed and classified most of the core approaches [7]. A primary problem in SpMT is deciding where to fork speculative child tasks, with systems proposed that operate at the loop level [23], basic-block level [3], and of course at the method level [5].

According to Chen & Olukotun [5], Oplinger *et al.* were the first to propose the concept of MLS in a limit study for C programs that sought to identify the maximum amounts of loop and method level parallelism available [18]. Hammond, Willey, and Olukotun later designed the Hydra chip multiprocessor for thread level speculation that included support for method level speculation [11]. Chen & Olukotun concurrently described a more realistic method level speculation system for Java, which combined a modified version of the Kaffe JVM and JIT compiler running on the Hydra architecture [5]. They found encouraging amounts of speculative parallelism in JIT-compiled code, and noticed that method level speculation can subsume loop-level speculation if loop bodies are extracted into method calls. Thread level speculation in general must balance overhead costs with potential parallelization, and can benefit from a variety of optimizations [25], and in particular the design of effective forking heuristics [27]. MLS brings the additional need for *return value prediction*, which reduces misspeculation costs due to common dependencies between method return values and continuations [14].

Our effort to unify several MLS models is largely motivated by the difficulty in comparing disparate proposals and in understanding the performance impact of design choices. While out-of-order execution has been argued critical to achieving good speculative performance [21], in-order, and out-order designs also have potential to complement each other, and some authors allow full speculative hierarchies [28]. The limited parallelism exposed by a purely out-of-order design in our own experimental work has further indicated that the choice of technique can greatly impact performance and parallelism characteristics [19, 20]. Higher level abstractions, as we offer in this work, provide an essential basis for developing optimized designs.

6. Conclusions and Future Work

MLS implementations face a number of design choices, and fundamental among them is whether and how to support speculative nesting. We presented a list abstraction showing that various MLS nesting designs, and even non-method-based speculative designs can be uniformly modeled and shown correct. Future work here includes developing an associated cost model, which can then be used to compare the performance impact of MLS nesting approaches.

MLS behaviour, as evidenced by our stack abstraction, can strongly depend on the specific selection of fork points. Although this is a fairly trivial observation, a detailed, high-level understanding of how and why parallelism is achieved is important to an efficient implementation. Our stack abstraction and analysis demonstrates that high-level parallelization strategies can be induced by specific speculation decisions. This gives insight on the source of performance and indicates useful directions for further optimization, either by using code transformations to rearrange code so as to produce specific prallelization patterns, or as in other fork heuristic designs by generating runtime hints to better control fork decisions. Our stack abstraction also suggests application to understanding other stack-based parallelism models, such as found in *parallel call* [8] and languages such as Pillar [1]. Parallel call was not designed to be speculative, but a translation of the speculation rules of our system would be straightforward.

Acknowledgments

This research was supported by the IBM Toronto Centre for Advanced Studies and the Natural Sciences and Engineering Research Council of Canada.

References

[1] T. Anderson, N. Glew, P. Guo, B. T. Lewis, W. Liu, Z. Liu, L. Petersen, M. Rajagopalan, J. M. Stichnoth, G. Wu, and D. Zhang. Pillar: A parallel implementation language. In *LCPC'07*, volume 5234 of *LNCS*, pages 141–155, Oct. 2007.

[2] C. Baker-Finch, D. J. King, and P. Trinder. An operational semantics for parallel lazy evaluation. In *ICFP '00*, pages 162–173, New York, NY, USA, 2000. ACM.

[3] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *SPAA*, pages 99–108, New York, NY, USA, Aug. 2002. ACM Press.

[4] H. J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI '08*, pages 68–78. ACM, 2008.

[5] M. K. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *PACT'98*, pages 176–184, Oct. 1998.

[6] R. Ennals and S. P. Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In *ICFP'03*, pages 287–298, Aug. 2003.

[7] M. J. Garzarán, M. Prvulovic, J. M. Llabería, V. Viñals, L. Rauchwerger, and J. Torrellas. Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. *TACO*, 2(3):247–279, Sept. 2005.

[8] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: Implementing a fast parallel call. *JPDC*, 37(1):5–20, Aug. 1996.

[9] J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. *TOPLAS*, 21(2):240–285, 1999.

[10] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP'08*, pages 175–184, Feb. 2008.

[11] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII*, pages 58–69, Oct. 1998.

[12] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP*, pages 48–60, 2005.

[13] T. Harris and S. Singh. Feedback directed implicit parallelism. In *ICFP'07*, pages 251–264, Oct. 2007.

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

[15] S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Sci. Comput. Program.*, 57(2):164–186, 2005.

[16] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL'05*, pages 378–391, Jan. 2005.

[17] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL'08*, pages 51–62, Jan. 2008.

[18] J. T. Oplinger, D. L. Heine, and M. S. Lam. In search of speculative thread-level parallelism. In *PACT'99*, pages 303–313, Oct. 1999.

[19] C. J. F. Pickett and C. Verbrugge. SableSpMT: A software framework for analysing speculative multithreading in Java. In *PASTE'05*, pages 59–66, Sept. 2005.

[20] C. J. F. Pickett and C. Verbrugge. Software thread level speculation for the Java language and virtual machine environment. In *LCPC'05*, volume 4339 of *LNCS*, pages 304–318, Oct. 2005.

[21] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation. In *ICS'05*, pages 179–188, 2005.

[22] N. Shavit and D. Touitou. Software transactional memory. In *PODC'95*, pages 204–213, Aug. 1995.

[23] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *TOCS*, 23(3):253–300, Aug. 2005.

[24] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *HPCA '02*, pages 65–75, Washington, DC, USA, 2002. IEEE.

[25] F. Warg. *Techniques to Reduce Thread-Level Speculation Overhead*. PhD thesis, Dept. of CSE, Chalmers U. of Tech., Göteborg, Sweden, May 2006.

[26] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *OOPSLA'05*, pages 439–453, Oct. 2005.

[27] J. Whaley and C. Kozyrakis. Heuristics for profile-driven method-level speculative parallelization. In *ICPP'05*, pages 147–156, 2005.

[28] M. Zahran and M. Franklin. Dynamic thread resizing for speculative multithreaded processors. In *ICCD'03*, pages 313–318, Oct. 2003.