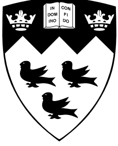


McGill University
School of Computer Science
Sable Research Group

Understanding Method Level Speculation

Sable Technical Report No. 2010-2

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

April 21st, 2010

Understanding Method Level Speculation

Christopher J.F. Pickett[1], Clark Verbrugge[1], and Allan Kielstra[2]

[1] School of Computer Science, McGill University
Montréal, Québec, Canada
{cpicke,clump}@sable.mcgill.ca

[2] 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. In this work we seek to establish a basic framework for understanding MLS designs. We first present a stack-based abstraction of MLS that encompasses major design choices such as in-order and out-of-order nesting and is also suitable for implementations. We then use this abstraction to develop the structural operational semantics for a series of progressively more flexible MLS models. Assuming the most flexible such model, we provide transition-based visualizations that reveal the speculative execution behaviour for a number of simple imperative programs. These visualizations show how specific parallelization patterns can be induced by combining common programming idioms with precise decisions about where to speculate. We find that the runtime parallelization structures are complex and non-intuitive, and that both in-order and out-of-order nesting are important for exposing parallelism. The overwhelming conclusion is that either programmer or compiler knowledge of how implicit parallelism will develop at runtime is necessary to maximize performance.

1 Introduction

Method level speculation (MLS) is a technique for runtime parallelization of sequential programs. Upon reaching a method (alternatively function, procedure) invocation, a “parent” thread will fork (spawn) a “child” thread that begins executing the method continuation in parallel while the parent executes the body of the call. Memory dependences in the child are buffered, such that any changes can be rolled back or discarded if necessary. When the parent thread returns from the call, it stops the child thread, validates its dependences, and if no violations occurred commits the results to main memory and resumes where the child left off. Although the distinction between which thread is the child and which is the parent varies, MLS can be seen as the most optimistic and most automatic of a number of continuation-based parallelization schemes: futures, safe futures, parallel call, and implicit parallelization for functional languages.

The initial hope with MLS, as with most work on automatic parallelization, was that irregular programs would run faster, exploiting parallelism with no additional programmer intervention. However, due to variability in the balances between parent and child thread lengths, value predictability, and the likelihood of dependence violations, some fork points end up being much better than others, and the overhead of bad forking decisions can easily dominate execution time. Naturally, one's first thought is to change the set of fork points to accomodate. Although this does have an effect on parallelism, it does so not only by eliminating the overhead from unprofitable speculation, but also by changing the dynamic thread structure and hence enabling parallelism where it was previously precluded. The complication is that changing the dynamic thread structure in turn changes the suitability of fork points. For an online or offline adaptive system that creates threads based on the suitability of fork points, this creates a feedback loop. The end result is that sometimes parallelism is obtained, sometimes not, but ultimately it is difficult to explain *why* things play out the way they do. A lack of insight into an MLS system we previously created [1, 2] was the initial motivation for this work.

There has been significant work on selecting fork points and the final effect on performance. There has been much less focus, at least in the case of MLS, on studying the relationship between program structure, choice of fork point, and the resultant parallel behaviour. For this we need an abstract way to describe the program structure and choice of fork point, and we need a way to "see" the behaviour, which in this case is a visualization of how parallel structures evolve over time.

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

Fig. 1. *Choice of fork points under MLS.*

By way of example, consider the code in Figure 1, in which we assume threads are created as soon as possible. If the speculation model prohibits nesting and only allows one child per parent thread at a time, then the parent executes `b();X;` while child 1 executes `c();Y;`. If the model allows out-of-order nesting, under which a parent can have more than one child at once, then the parent executes `b();`, child 2 executes `X;`, and child 1 executes `c();Y;`. If instead in-order nesting is allowed, under which children can create children of their own, then the parent executes `b();X;`, child 1 executes `c();`, and child 3 executes `Y;`. If both in-order and out-of-order nesting are permitted, then `b()`, `X`, `c()`, and `Y` can all execute in parallel. The precise nature of the resulting parallelism is not intuitively obvious, and depends on the interaction of code, MLS design, and of course scheduling.

In this work, we provide an abstract description of fork point choice via a unified stack-based model of MLS that encompasses all of these scenarios. We also provide a series of sub-models referring to our larger unified one that are each described by their structural operational semantics, where the structures involved are the precise relationships between call stacks and threads. We then take an abstract view of program structure that isolates useful work from higher-level organizational concerns, enabling

a focus on the effects of code layout. Finally, we provide and employ a method for visualizing runtime parallel behaviour that relies on showing the state evolution that arises from repeated application of our structural rules.

Given such a framework, we can compare the parallelization models used by MLS-like systems directly, and begin to understand at a non-superficial level why the results differ between them. Our vision is that this understanding can be used to inform programmers and compiler writers trying to structure or restructure programs for efficient implicit parallelism, and to guide the design of runtime systems. Finally, our approach provides a basis for inventing novel extensions: it allows for rapid specification and visualization without the burden of implementation.

We make the following specific contributions:

1. We propose a stack-based operational semantics as a unified model of MLS. This model is derived from a working implementation [1, 2]. Our model provides support for lazy stack buffering at the frame level, a unique optimization designed to improve the speed of local variable access.
2. We provide several MLS sub-models, each of which is described by its structural operational semantics, and relate them to our unified stack model. These sub-models are suitable for direct visualization of programs executing under MLS.
3. We examine the behaviour of a number of common coding idioms in relation to our stack formalism. We show how these idioms map to specific parallel runtime structures depending on code layout and fork point choice. We derive several guidelines as to how parallelism can be exposed, and end with a complex example that demonstrates implicit workload scheduling using recursion.

In Section 2, we present our unified stack model, which we use to develop a series of MLS sub-models in Section 3. We explore coding idioms and behaviour in Section 4, present related work in Section 5, and finally conclude and discuss future work.

2 Stack Abstraction

We now present a core stack abstraction that directly encodes the call stack and thread manipulations central to all 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.

The standard sequential call stack model found in most languages has two simple operations that manipulate stack frames: PUSH for entering methods, and POP for exiting methods. Frames stores local variables and other context required for correct method execution, and for well-behaved languages the operations must be matched. For languages that support multithreading, START and STOP operations for creating and destroying non-speculative threads are also necessary. Our parallel call stack model for MLS is simply a parallel extension of this standard. It introduces three new operations, FORK, COMMIT, and ABORT. These new operations manipulate stack frames, but they also have the power to create and destroy speculative threads. FORK can now 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.

We make several assumptions: 1) well-ordered PUSH and POP nesting is provided by the underlying language, even in the case of exceptional control flow; 2) stack operations complete atomically; 3) non-stack operations, while not explicitly modelled, may be freely interleaved with stack operations on running threads; 4) speculative accesses to global state variables, if they exist, are handled externally, for example via some transactional memory or dependence buffering system; 5) register values are spillable to a frame on demand; and 6) stacks grow upwards.

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, it retrieves and copies them 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: our experience with a software implementation indicates a steep performance penalty for copying entire thread stacks.

The main abstraction is described via its operational semantics in Figure 2. 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$(|t)$: create a new non-speculative thread t with an empty stack.
STOP(t): destroy non-speculative thread t, provided its stack is empty.
PUSH(t, f): add a new frame with unique name f to the stack of thread t.
FORK$(t, f|u)$: execute PUSH(t, f), and then create a new child thread u that starts executing the method continuation using a buffered version of the previous frame from thread t. Cannot be issued on an empty stack.
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.
ABORT(t): execute POP(t) (internally JOIN) and 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) (internally JOIN) and 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.

We now turn to a detailed description of the operations in Figure 2. We model threads as unique integers, and maintain several thread sets: T is the set of all threads, T_l is the set of live threads, T_n and T_s are non-speculative and speculative threads

$$\mathrm{CREATE}(T_p, \sigma | t)\,\frac{T_p = T_n \oplus T_p = T_s}{t = |T|, T \uplus \{t\}, T_l \uplus \{t\}, T_p \uplus \{t\}, \mathit{stack}(t) = \sigma}$$

$$\mathrm{DESTROY}(t)\,\frac{T_p \subseteq T \;.\; t \in T_p \quad T_p = T_n \oplus T_p = T_s}{T_l \setminus= \{t\}, T_p \setminus= \{t\}} \qquad [*]\mathrm{START}(|t)\,\frac{}{\mathrm{CREATE}(T_n, \emptyset | t)}$$

$$[*]\mathrm{STOP}(t)\,\frac{t \in T_n \quad \mathit{stack}(t) = \emptyset}{\mathrm{DESTROY}(t)} \qquad [*]\mathrm{PUSH}(t, f)\,\frac{t \in T_l \quad f \notin F \quad \sigma = \mathit{stack}(t)}{\mathit{stack}(t) = \sigma : f, F \uplus \{f\}}$$

$$\mathrm{BUFFER}(t, e|e')\,\frac{t \in T_l \cup T_c \quad e \in \mathit{stack}(t) \quad e \in F \quad \mathit{child}(e) \notin T_s}{e' = e, F \uplus \{e'\}}$$

$$[*]\mathrm{FORK}(t, f|u)\,\frac{\dfrac{\mathrm{PUSH}(t, f)}{\sigma : e : f = \mathit{stack}(t), \mathrm{BUFFER}(t, e|e')} \quad \mathrm{CREATE}(T_s, e'|u)}{\mathit{parent}(u) = t, \mathit{child}(e) = u}$$

$$[*]\mathrm{POP}(t)\,\frac{\dfrac{t \in T_l \quad \sigma : e' : f' = \mathit{stack}(t) \quad f' \in F \quad \mathit{child}(e') \notin T_l}{\dfrac{t \in T_n}{e' \in F \oplus e' \notin F} \oplus \dfrac{t \in T_s}{e' \in F \oplus \nu p_{p \geq 0} \;.\; \mathrm{BUFFER}(p, e|e')}}}{\mathit{stack}(t) = \sigma : e'}$$

$$\mathrm{JOIN}(t|u)\,\frac{t \in T_l \quad \sigma : e : f = \mathit{stack}(t) \quad e, f \in F \quad \mathit{child}(e) \in T_l}{\mathit{stack}(t) = \sigma : e, u = \mathit{child}(e)}$$

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

$$\mathrm{MERGE_COMMITS}(t, u)\,\frac{\gamma = \mathit{commits}(t) \quad \delta = \mathit{commits}(u)}{\mathit{commits}(t) = \gamma : u : \delta, T_c \uplus \{u\}}$$

$$\mathrm{PURGE_COMMITS}(t)\,\frac{\begin{array}{c}\gamma : \delta = \mathit{commits}(t) \;.\; \delta = \nu\delta_{\delta \supseteq \emptyset} \;.\\ \forall c \in \delta\ \forall f \in \mathit{stack}(c), \mathit{child}(f) \notin T_l\end{array}}{\mathit{commits}(t) = \gamma, T_c \setminus= \{\delta\}}$$

$$\mathrm{CLEANUP}(t, u)\,\frac{\mathrm{DESTROY}(u)}{\mathrm{PURGE_COMMITS}(t)} \qquad [*]\mathrm{COMMIT}(t)\,\frac{\begin{array}{c}\mathrm{JOIN}(t|u) \quad \mathrm{MERGE_STACKS}(t, u)\\ \mathrm{MERGE_COMMITS}(t, u)\end{array}}{\mathrm{CLEANUP}(t, u)}$$

$$\mathrm{ABORT_ALL}(t)\,\frac{\dfrac{\forall f \in \mathit{stack}(t) \;.\; u = \mathit{child}(f) \in T_l}{\mathrm{ABORT_ALL}(u)}}{F \setminus= \{\mathit{stack}(t)\}, \mathrm{CLEANUP}(t, u)} \qquad [*]\mathrm{ABORT}(t)\,\frac{\begin{array}{c}\mathrm{JOIN}(t|u)\\ \mathrm{ABORT_ALL}(u)\end{array}}{\mathrm{CLEANUP}(t, u)}$$

Fig. 2. *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.

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: $T = T_n \cup T_s \cup T_c$, $T_n \cap T_s = \emptyset$, $T_s \subset T_l$, $T_l \subseteq T_n \cup T_s$, $T_n \cap T_l \neq \emptyset$, and $T_c \cap T_l = \emptyset$. 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, such that each newly pushed or buffered frame is not already in F. This invariant is maintained by removing frames from F when threads are aborted. Given a frame $f \in F$, buffering creates a new frame f' by appending $'$ to the name. Given a frame f', f is the less-speculative version of the same frame in some ancestor thread. Note that for notational convenience, e' and f' in POP(t) may belong to a non-speculative thread, in which case no such ancestor exists. Variables d, e, f and their primed derivatives represent concrete frames, whereas σ, ρ, π represent lists of frames. Similarly, variables p, t, u represent concrete threads, whereas γ, δ represent lists of threads.

In addition to these sets, there are several functions that maintain mappings between them. $stack(t \in T)$ maps t to a thread stack, which is a list of frames in F, $child(f \in F)$ 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. Rules act like functions, requiring a list of arguments. In some cases they also produce values, which are separated from arguments by $|$. The use of exclusive or ($\oplus$) indicates a choice between one rule and another or one set of premises and another. We use $S \uplus \{s\}$ and $S \setminus= \{s\}$ to indicate set additions and removals respectively. Finally, ν is the greatest fixed point operator from the μ-calculus that given some terminating condition maximizes its operand starting from $\top$. We use ν 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 t, 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 t to σ. DESTROY conversely removes t 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 t, 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.

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 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 both exist and have a speculative child.

MERGE_STACKS is called by COMMIT. It copies the live range of frames $d' : \rho$ from the child u to the parent t. It takes the stack of u and looks for a less-speculative ver-

sion 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 must have complete stacks. MERGE_COMMITS, as called by COMMIT, takes the commit list γ from the parent, 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. 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. It is called after the COMMIT and ABORT operations, and internally from ABORT_ALL. Whereas JOIN contains the common logic that precedes commits and aborts, CLEANUP contains the common logic that follows 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-world implementation that uses this stack abstraction, child threads must be stopped before they can be committed or aborted.

3 Individual MLS Models

Using the stack abstraction from Figure 2, we now develop a series of concentric and progressively more flexible MLS models each described by their structural operational semantics, shown individually in Figures 3–9. They provide an exhaustive visual reference to the MLS design considerations in our unified abstraction by exposing the core state evolution patterns that define execution. We use these models 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 ($\not\succ$). Above the inference line is the corresponding $[*]$ command from Figure 2, followed by model-specific restrictions on behaviour and local variable mappings. For a mapping $x = y$, x is a value found in the transitive expansion of the $[*]$ command from Figure 2 and y is the local value. 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 7, 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 2, 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 3 contains a simple structured non-speculative stack model common to many languages. Non-speculative 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 2, the latter being the penultimate operation on a thread, followed by STOP.

Figure 4 contains the simplest MLS stack model, one that extends Figure 3 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$\not\succ$ 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$\not\succ$ must match with N$\prec$, N$\uparrow$ must match N$\downarrow$, and N$\uparrow\bot$ is precluded for speculative threads because BUFFER$(\tau, e|e')$ will not complete.

The model in Figure 5 extends Figure 4 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$\uparrow\bot$ takes the right-hand case and so buffers e' from its parent. Finally, this model updates N$\succ$ and N$\not\succ$ to handle situations where the child may have left e' via S$\uparrow\bot$ or S$\downarrow$, now representing the child thread stack by φ' instead of e'.

The next model in Figure 6 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 4 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 express in software, but offers significantly limited parallelism [1].

After out-of-order nesting comes in-order nesting in Figure 7. I$\prec$ allows speculative thread α to create β independently of its parent. N$\not\succ$ 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$\uparrow\bot$ 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 $\rightsquigarrow$ indicates that there is a path of buffered frames from π' backwards to π. This rule is an extended version of S$\uparrow\bot$, which only handles buffering from the immediate parent. S$\uparrow$ works nicely as is with in-order speculation, and S$\uparrow\bot$ works not only in the simple case above but also when the buffered frame is in some committed thread $c \in T_c$.

In Figure 8, 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, but is 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 9, the last restrictions are removed so that all of the features in the main abstraction in Figure 2 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$\not\succ$] shows a recursive abort on a

$$\text{START}\ \frac{\text{START}(|\tau)}{\begin{array}{c}\Rightarrow\\ \tau\end{array}} \qquad \text{STOP}\ \frac{\text{STOP}(\tau)}{\begin{array}{c}\Rightarrow\\ \tau\end{array}} \qquad \text{N}\!\downarrow\ \frac{\text{PUSH}(\tau, f) \quad \tau \in T_n}{\begin{array}{ccc} & & f\\ \sigma & \Rightarrow & \sigma\\ \tau & & \tau\end{array}}$$

$$\text{N}\!\uparrow\ \frac{\begin{array}{c}\text{POP}(\tau) \quad e \in stack(\tau)\\ e' = e \qquad f' = f\end{array}}{\begin{array}{ccc} f & & \\ e & & e\\ \sigma & \Rightarrow & \sigma\\ \tau & & \tau\end{array}} \qquad \text{N}\!\uparrow\!\bot\ \frac{\begin{array}{c}\text{POP}(\tau) \quad e \notin stack(\tau)\\ e' = e \qquad f' = f\end{array}}{\begin{array}{ccc} f & \Rightarrow & \\ \tau & & \tau\end{array}}$$

Fig. 3. *Adults-only model.* No speculation.

$$\text{N}\!\prec\ \frac{\begin{array}{c}\text{FORK}(\tau, f|\alpha) \quad \tau \in T_n\\ \forall d \in \sigma, child(d) \notin T_l\end{array}}{\begin{array}{cccc} & & f & \\ e & & e - e' & \\ \sigma & \Rightarrow & \sigma & \\ \tau & & \tau & \alpha\end{array}} \qquad \text{N}\!\succ\ \frac{\begin{array}{c}\text{COMMIT}(\tau)\\ \rho = \emptyset\\ d = e \quad d' = e'\end{array}}{\begin{array}{cccc} f & & & \\ e - e' & & & e'\\ \sigma & & \Rightarrow & \sigma\\ \tau & \alpha & & \tau\end{array}} \qquad \text{N}\!\not\succ\ \frac{\text{ABORT}(\tau)}{\begin{array}{cccc} f & & & \\ e - e' & & & e\\ \sigma & & \Rightarrow & \sigma\\ \tau & \alpha & & \tau\end{array}}$$

Fig. 4. *Totalitarian model.* One speculative child allowed, but only non-speculative threads can perform stack operations.

$$\text{S}\!\downarrow\ \frac{\begin{array}{c}\text{PUSH}(\alpha, f)\\ \sigma = \pi' \qquad \omega \neq \emptyset\\ e' = car(\pi')\ .\ e \in \pi\end{array}}{\begin{array}{ccc} \omega & & \omega \quad f\\ \pi - \pi' & \Rightarrow & \pi - \pi'\\ \tau \quad \alpha & & \tau \quad \alpha\end{array}} \qquad \text{S}\!\uparrow\ \frac{\begin{array}{c}\text{POP}(\alpha) \qquad f' = f\\ \sigma : e' = \pi' \quad \omega \neq \emptyset\\ d' = car(\pi')\ .\ d \in \pi\end{array}}{\begin{array}{ccc} \omega \quad f & & \omega\\ \pi - \pi' & \Rightarrow & \pi - \pi'\\ \tau \quad \alpha & & \tau \quad \alpha\end{array}}$$

$$\text{S}\!\uparrow\!\bot\ \frac{\begin{array}{c}\text{POP}(\alpha) \quad f' = \pi'\\ \sigma : e' = \emptyset \quad \omega \neq \emptyset\\ f = car(\pi)\end{array}}{\begin{array}{ccc} \omega & & \omega\\ \pi - \pi' & & \pi -\\ e & & e \quad e'\\ \varphi & \Rightarrow & \varphi\\ \tau \quad \alpha & & \tau \quad \alpha\end{array}} \qquad \text{N}\!\succ\ \frac{\begin{array}{c}\text{COMMIT}(\tau)\\ d' : \rho = \varphi'\\ d : \pi : e = \varphi\end{array}}{\begin{array}{cccc} f & & & \\ \varphi - \varphi' & & & \varphi'\\ \sigma & & \Rightarrow & \sigma\\ \tau & \alpha & & \tau\end{array}} \qquad \text{N}\!\not\succ\ \frac{\begin{array}{c}\text{ABORT}(\tau)\\ \sigma : e = \sigma : \varphi\end{array}}{\begin{array}{cccc} f & & & \\ \varphi - \varphi' & & & \varphi\\ \sigma & & \Rightarrow & \sigma\\ \tau & \alpha & & \tau\end{array}}$$

Fig. 5. *Kid-friendly model.* Allows PUSH and POP actions on speculative threads, overriding N$\succ$ and N$\not\succ$ to accomodate.

$$\text{O}\!\prec\ \frac{\begin{array}{c}\text{FORK}(\tau, f|\beta) \qquad \tau \in T_n\\ d' = car(\pi')\ .\ d = car(\pi)\end{array}}{\begin{array}{ccc} & & f\\ e & & e - e'\\ \pi - \pi' & & \pi \text{———} \pi'\\ \sigma & \Rightarrow & \sigma\\ \tau \quad \alpha & & \tau \quad \beta \quad \alpha\end{array}}$$

Fig. 6. *Catholic model.* Provides out-of-order nesting via O$\prec$ to allow an arbitrary number of speculative children for non-speculative threads.

$$\text{I}\!\prec\ \frac{\begin{array}{c}\text{FORK}(\alpha, f|\beta)\\ \sigma = \pi' \qquad \omega \neq \emptyset\\ d' = car(\pi')\ .\ d \in \pi\end{array}}{\begin{array}{ccc} & & f\\ \omega \quad e & & \omega \quad e - e'\\ \pi - \pi' & \Rightarrow & \pi - \pi'\\ \tau \quad \alpha & & \tau \quad \alpha \quad \beta\end{array}} \qquad \text{I}\!\succ\ \frac{\begin{array}{c}\text{COMMIT}(\tau) \quad \tau \in T_n\\ \omega \neq \emptyset \quad d' : \rho = \varphi' : \omega\\ d : \pi : e = \varphi\end{array}}{\begin{array}{ccc} f \quad \omega & & \omega\\ \varphi - \varphi' - \varphi'' & & \varphi \text{———} \varphi''\\ \sigma & \Rightarrow & \sigma\\ \tau \quad \alpha \quad \beta & & \tau - \alpha \quad \beta\end{array}}$$

$$\text{I}\!\uparrow\!\bot\ \frac{\begin{array}{c}\text{POP}(\beta) \quad f' = \pi'' \quad \sigma : e' = \emptyset\\ \omega, \upsilon \neq \emptyset \qquad f = car(\pi')\\ car(\pi') \rightsquigarrow car(\pi)\\ \tau = \nu p_{p \geq 0}\ .\ \text{BUFFER}(p, e|e')\end{array}}{\begin{array}{ccc} \omega \quad \upsilon & & \omega \quad \upsilon\\ \pi \cdots \pi' - \pi'' & & \pi \cdots \pi' -\\ e & & e \qquad e'\\ \varphi & \Rightarrow & \varphi\\ \tau \quad \alpha \quad \beta & & \tau \quad \alpha \quad \beta\end{array}}$$

Fig. 7. *One big happy model.* Provides in-order nesting via I$\prec$ to allow speculative children of speculative threads. Note that in I$\uparrow\bot$, τ may be speculative.

$$\text{S}\succ\ \frac{\begin{array}{cc}\text{COMMIT}(\alpha) & \omega \neq \emptyset \\ d' : \rho = \varphi'' & d : \pi : e = \varphi'\end{array}}{\begin{array}{l}\overset{\omega}{\varphi}\!-\!\overset{f}{\varphi'}\!-\!\varphi'' \\ \sigma \\ \tau \quad \alpha \quad \beta\end{array} \Rightarrow \begin{array}{l}\overset{\omega}{\varphi}\!-\!\varphi'' \\ \sigma \\ \tau \quad \alpha\end{array}}
\qquad
\text{SI}\succ\ \frac{\begin{array}{cc}\text{COMMIT}(\alpha) & \omega, \upsilon \neq \emptyset \\ d' : \rho = \varphi'' : \upsilon & d : \pi : e = \varphi'\end{array}}{\begin{array}{l}\overset{\omega}{\varphi}\!-\!\overset{f}{\varphi'}\!-\!\overset{\upsilon}{\varphi''}\!-\!\varphi''' \\ \sigma \\ \tau \quad \alpha \quad \beta \quad \gamma\end{array} \Rightarrow \begin{array}{l}\overset{\omega}{\varphi}\!-\!\overset{\upsilon}{\varphi''}\!\text{———}\!\varphi''' \\ \sigma \\ \tau \quad \alpha\!-\!\beta \quad \gamma\end{array}}$$

$$[\text{I}\succ\text{MERGE}]\ \frac{\begin{array}{cc}\text{COMMIT}(\tau) & \omega \neq \emptyset \\ d' : \rho = \varphi''' : \omega & d : \pi : e = \varphi'\end{array}}{\begin{array}{l}\overset{f}{\varphi'}\!\text{——}\!\overset{\omega}{\varphi'''}\!\text{——}\!\varphi'''' \\ \sigma \\ \tau\!-\!\alpha \quad \beta\!-\!\gamma \quad \delta\end{array} \Rightarrow \begin{array}{l}\overset{\omega}{\varphi'''}\!\text{————}\!\varphi'''' \\ \sigma \\ \tau\!-\!\alpha\!-\!\beta\!-\!\gamma \quad \delta\end{array}}$$

Fig. 8. *Nuclear model.* Allows speculative threads to commit their own children. [I≻MERGE]'s behaviour is provided by I≻.

$$\text{IO}\prec\ \frac{\begin{array}{cc}\text{FORK}(\alpha, f|\gamma) & \omega \neq \emptyset \\ \multicolumn{2}{c}{d' = car(\pi')\ .\ d = car(\pi)}\end{array}}{\begin{array}{l}\overset{\omega}{\pi}\!-\!\overset{e}{\pi'}\!-\!\pi'' \\ \sigma \\ \tau \quad \alpha \quad \beta\end{array} \Rightarrow \begin{array}{l}\overset{\omega}{\pi}\!-\!\overset{e-\overset{f}{e'}}{\pi'}\!\text{——}\!\pi'' \\ \sigma \\ \tau \quad \alpha \quad \gamma \quad \beta\end{array}}
\quad
[\text{IO}\not\succ]\ \frac{\begin{array}{cc}\text{ABORT}(\tau) & \omega \neq \emptyset \\ \multicolumn{2}{c}{\sigma : e = \sigma : \varphi}\end{array}}{\begin{array}{l}\overset{f}{\varphi}\!-\!\overset{\overset{\omega}{\pi_{\tau}}-\pi'}{\varphi'}\!\text{——}\!\varphi'' \\ \sigma \\ \tau \quad \alpha \quad \gamma \quad \beta\end{array} \Rightarrow \begin{array}{l}\varphi \\ \sigma \\ \tau\end{array}}
\quad
[\text{OI}\prec]\ \frac{\begin{array}{cc}\multicolumn{2}{c}{\text{FORK}(\beta, f|\gamma)} \\ \sigma = \pi' & \omega \neq \emptyset \\ \multicolumn{2}{c}{d' = car(\pi')\ .\ d \in \pi}\end{array}}{\begin{array}{l}\overset{\overset{\omega}{\pi}-\overset{e}{\pi'}}{\varphi}\!\text{——}\!\varphi' \\ \tau \quad \beta \quad \alpha\end{array} \Rightarrow \begin{array}{l}\overset{\overset{\omega}{\pi}-\overset{e-\overset{f}{e'}}{\pi'}}{\varphi}\!\text{————}\!\varphi' \\ \tau \quad \beta \quad \gamma \quad \alpha\end{array}}$$

Fig. 9. *Libertarian model.* Allows both in-order and out-of-order nesting. [IO⊁] and [OI≺] are provided by N⊁ and I≺.

thread with both in- and out-of-order nesting, and [OI≺] shows in-order nesting after out-of-order nesting has taken place, as already allowed by O≺ followed by I≺.

The above models illustrate the core behaviour patterns of common speculation strategies. In the next section, we explore a series of stack evolutions that assume support for the final combined stack model in Figure 9, although in some cases one of the less flexible models will suffice.

4 Speculation Idioms

Simple changes in the structure of input programs and choice of fork points can dramatically affect the dynamic structure of the speculative call stack. In this section we explore several common code idioms and their behaviour under MLS using the full stack abstraction. We examine straight-line code, if-then conditionals, iteration, head recursion, and tail recursion, with a view towards discovering idiomatic code structures and speculation decisions that yield interesting parallel execution behaviours. We present a series of linear state evolutions to visualize the results, each of which was generated from a list of operations by an Awk script implementation of our model.

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 guides the stack evolutions in cases where more than one operation is possible. These simplistic execution timing assumptions yield surprisingly complex behaviour, and so provide a good basis for understanding even more complex situations.

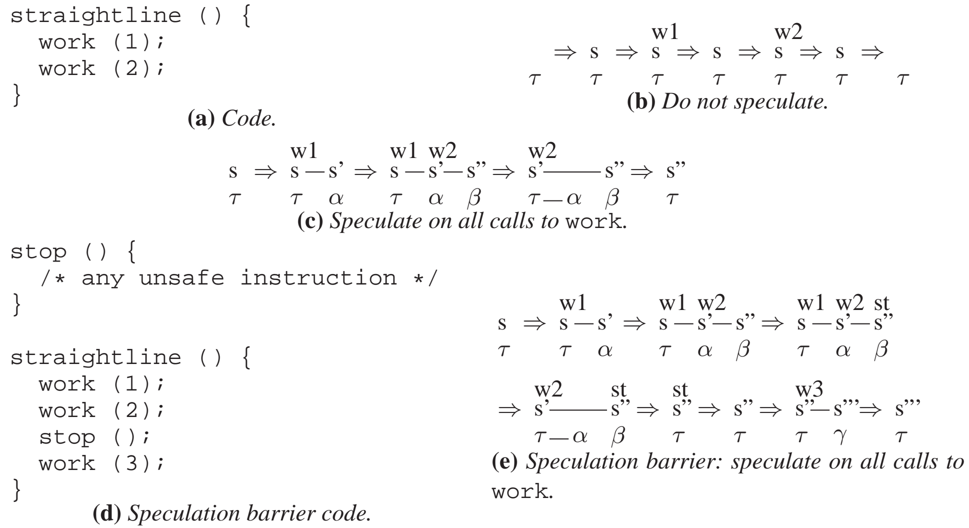


Fig. 10. *Straight-line.*

Straight-line The simplest code idiom in imperative programs is straight-line code, where one statement executes after the next without branching, as in Figure 10. In 10a, two sequential calls to `work` are shown, with the non-speculative stack evolution in 10b. In future evolutions we omit the initial N$\downarrow$ and final N$\uparrow\perp$. In 10c, 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, the choices between PUSH and FORK clearly affect which threads execute which regions of code, and whether they have useful work to do. In 10d, a function `stop` is introduced containing an unsafe operation that acts as a speculation barrier. The result in 10e 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.

If-then Another simple code idiom is if-then conditional branching. If the value of the conditional is speculative, then particular code paths followed depending on the value themselves become speculative. In Figure 11, speculating on the call to `work(1)`, it is necessary to predict a boolean return value. If the speculation is correct, as in 11d and 11e, then the speculative work w2 or w3 respectively is committed. Otherwise, that work is aborted, as in 11f and 11g.

For this speculation idiom to be useful, the function producing the return value should take a long time to execute. Further, extensions to our speculation model could allow for multiple predicted return values, associating one speculative thread with 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 will be 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);
}
```

(a) *Code.*

```
       w1       w2       w3
i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ i ⇒ i
τ    τ    τ    τ    τ    τ    τ
```

(b) *Do not speculate,* `work(1)` *returns true.*

```
       w1       w3
i ⇒ i ⇒ i ⇒ i ⇒ i
τ    τ    τ    τ    τ
```

(c) *Do not speculate,* `work(1)` *returns false.*

```
       w1          w1 w2      w2            w3
i ⇒ i — i' ⇒ i — i' ⇒ i' ⇒ i' ⇒ i' ⇒ i'
τ    τ    α     τ    α     τ     τ     τ     τ
```

(d) *Speculate on* `work(1)`*, predict true correctly.*

```
       w1          w1 w3      w3
i ⇒ i — i' ⇒ i — i' ⇒ i' ⇒ i'
τ    τ    α     τ    α     τ     τ
```

(e) *Speculate on* `work(1)`*, predict false correctly.*

```
       w1          w1 w2            w3
i ⇒ i — i' ⇒ i — i' ⇒ i ⇒ i ⇒ i
τ    τ    α     τ    α     τ    τ    τ
```

(f) *Speculate on* `work(1)`*, predict true incorrectly.*

```
       w1          w1 w3           w2       w3
i ⇒ i — i' ⇒ i — i' ⇒ i ⇒ i ⇒ i ⇒ i ⇒ i
τ    τ    α     τ    α     τ    τ    τ    τ    τ
```

(g) *Speculate on* `work(1)`*, predict false incorrectly.*

Fig. 11. *If-then.*

Iteration The most common code idiom considered for speculation is loop iteration. Chen & Olukotun demonstrated that if a loop body is extracted into a method call, then method level speculation can subsume loop level speculation [3]. We explore an example loop under different speculation assumptions in Figure 12 to better understand the behaviour. Speculating on all calls to `work` in 12c, the loop is quickly divided up into one iteration per thread for as many threads as there are iterations.

To limit this aggressive parallelization, we explored speculating on every m in n calls. In 12d, a child is forked every 1 in 2 calls. The stack evolves to a point where both w1 and w2 are executing concurrently and no other stack operations are possible. Once w1 and w2 complete, a number of intermediate evolutions open up, but they all lead to the same state with w3 and w4 executing concurrently. Effectively, the loop is parallelized across 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 our model does not explicitly support; or 2) unroll the loop and push multiple iterations into the loop body, per 12g.

Tail Recursion Tail recursion is explored in Figure 13. 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` usefully populates the stack with successive calls to `work`. However, this also creates just as many useless threads that only ever fall out of the recursion, although they stop almost immediately as they encounter elder siblings. Speculating on just `work` in 13d is good, and yields a stack

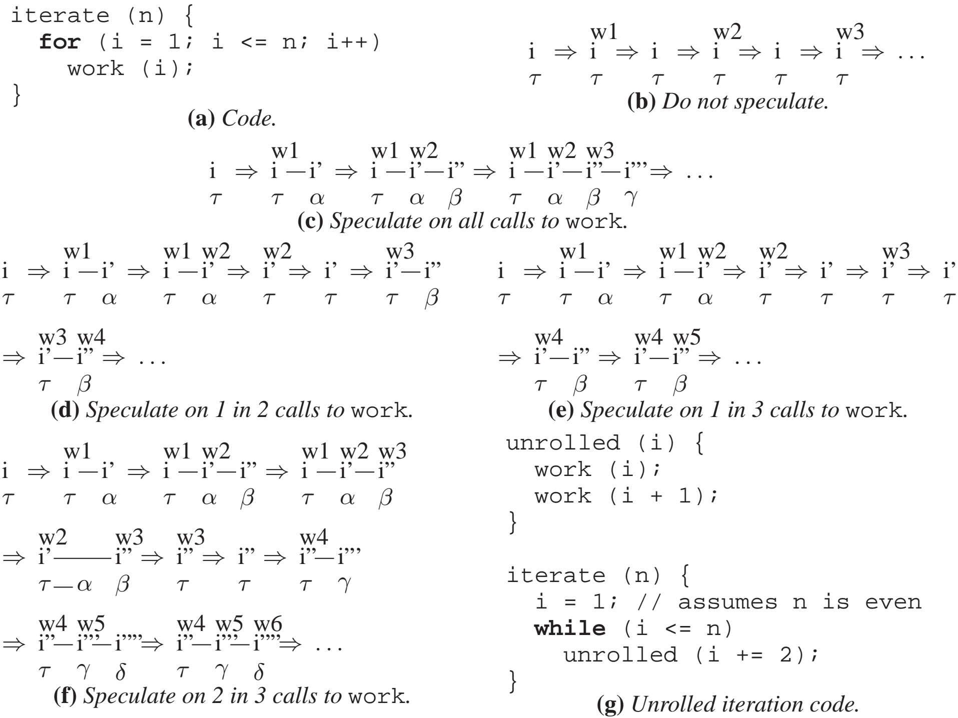


Fig. 12. *Iteration.*

structure identical to that produced by speculating on all calls in iteration, as in Figure 12c, modulo the interleaving `recurse` frames. On the contrary, speculating on just `recurse` is bad, because calls to `work` are never parallelized.

Speculating on 1 in 2 calls to `work` yields again a structure directly comparable to iteration, where w1 and w2 will execute in parallel before the stack evolves to w3 and 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 interesting behaviour which sees speculative children unwind the stack by one frame before stopping.

Head Recursion Head recursion is considered in 14. 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 execute in parallel, and then the stack evolves to a state where wl and wk execute in parallel.

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

(a) *Code.*

```
                  w2
        w1     r2 r2
r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒ ...
τ    τ    τ    τ    τ
```

(b) *Do not speculate.*

```
                                   w2
      w1           w1 r2           w1 r2—r2'
r1 ⇒ r1—r1'⇒      r1—r1—r1''⇒     r1—r1'———r1''⇒ ...
τ     τ   α        τ  α  β         τ  α  γ  β
```

(c) *Speculate on all calls (inefficient).*

```
                         w2
      w1        w1 r2    w1 r2—r2'
r1 ⇒ r1—r1'⇒   r1—r1'⇒  r1—r1'    ⇒ ...
τ     τ  α      τ  α     τ  α  β
```

(d) *Speculate on all calls to* `work` *(good).*

```
                               w2
      w1             r2        r2
r1 ⇒ r1 ⇒ r1 ⇒ r1—r1'⇒ r1—r1'⇒ ...
τ    τ    τ    τ  α    τ  α
```

(e) *Speculate on all calls to* `recurse` *(bad).*

```
                                                                    w3            w3 r4
                         w2      w2             r3        r3        r3—r3'        r3—r3'
      w1       w1 r2     w1 r2   r2      r2     r2—r2'    r2—       r2———         r2———
r1 ⇒ r1—r1'⇒  r1—r1'⇒   r1—r1'⇒ r1'⇒   r1'⇒   r1'     ⇒ r1' r1'⇒  r1'    r1'⇒  r1'    r1''
τ     τ  α     τ  α      τ  α    τ       τ      τ  β      τ  β      τ  γ  β       τ  γ  β

      w4
   w3 r4
   r3—r3'
   r2———
⇒ r1'     r1'⇒ ...
   τ  γ  β
```

(f) *Speculate on 1 in 2 calls to* `work` *and* `recurse` *(inefficient).*

```
                                                                                  w4
                                                       w3         w3 r4      w3 r4
                         w2      w2            r3      r3—r3'     r3—r3'     r3—r3'
      w1       w1 r2     w1 r2   r2      r2    r2      r2         r2         r2
r1 ⇒ r1—r1'⇒  r1—r1'⇒   r1—r1'⇒ r1'⇒   r1'⇒ r1'⇒   r1'    ⇒   r1'    ⇒   r1'     ⇒ ...
τ     τ  α     τ  α      τ  α    τ       τ     τ       τ  β       τ  β       τ  β
```

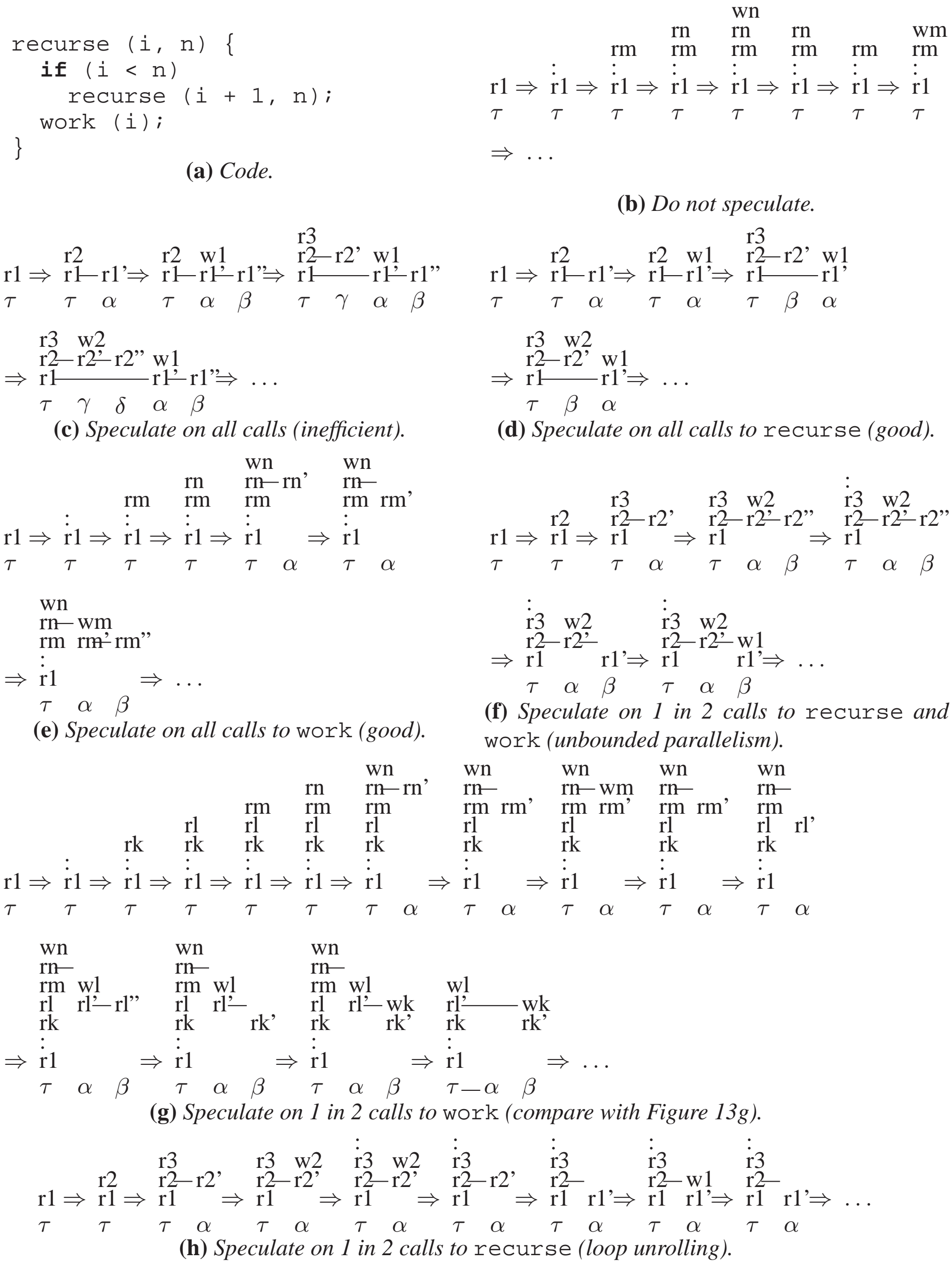
(g) *Speculate on 1 in 2 calls to* `work` *(good).*

```
                                                                                      w4
                                           w3        w3                 r4            r4
                          w2          r3   r3        r3        r3       r3            r3
      w1          r2      r2     r2   r2—r2'   r2—r2'    r2—       r2—      r2—           r2—
r1 ⇒ r1 ⇒ r1 ⇒ r1 ⇒   r1 ⇒  r1 ⇒  r1       ⇒ r1      ⇒ r1  r1'⇒ r1  r1'⇒ r1  r1'⇒    r1  r1'
τ    τ    τ    τ      τ     τ     τ  α      τ  α      τ  α     τ  α     τ  α         τ  α

                       w5            w5
           r5          r5            r5
   r4      r4—r4'      r4—r4'        r4—
   r3      r3          r3            r3  r3'
   r2—     r2———       r2———         r2———
⇒ r1  r1'⇒ r1     r1'⇒ r1      r1'⇒ r1      r1'⇒ ...
   τ  α     τ  β  α    τ  β  α       τ  β  α
```

(h) *Speculate on 1 in 2 calls to* `recurse` *(bad).*

Fig. 13. *Tail recursion.*

We were again surprised by speculating on 1 in 2 calls to `recurse`: α executes w2, and after returning the stack evolves until it executes 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` start all together and are then 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



(a) *Code.*

(b) *Do not speculate.*

(c) *Speculate on all calls (inefficient).*

(d) *Speculate on all calls to* `recurse` *(good).*

(e) *Speculate on all calls to* `work` *(good).*

(f) *Speculate on 1 in 2 calls to* `recurse` *and* `work` *(unbounded parallelism).*

(g) *Speculate on 1 in 2 calls to* `work` *(compare with Figure 13g).*

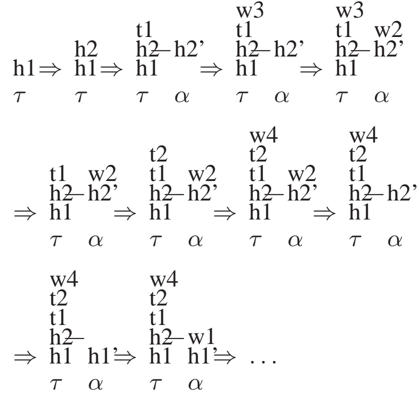
(h) *Speculate on 1 in 2 calls to* `recurse` *(loop unrolling).*

Fig. 14. *Head recursion.*

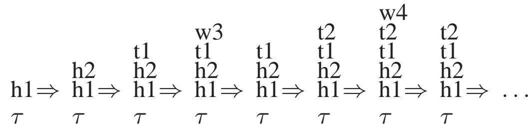
`recurse`. The unrolling within a given batch is in-order, but the creation of batches themselves is out-of-order.

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

head2 (i, n) {
  tail1 (i, n);
  work (i);
}

tail1 (i, n) {
  work (i);
  tail2 (i, n);
}

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

(a) *Two head then two tail code.*



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



(c) *Two head then two tail: do not speculate.*

```
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);
}
```

(d) *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.*

Fig. 15. *Mixed head and tail recursion.*

Mixed Head and Tail Recursion Finally, we experimented with a mix of head and tail recursion, as in Figure 15. Given the interesting behaviours seen for these kinds of recursion in isolation, 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 15a and 15b constitute a minimal example that uses head recursion to provide batch processing and tail recursion to limit stack growth. In 15b, 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 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 4th and 7th head recursive calls and the first tail recursive call. A generalized function that provides this behaviour is given in 15d.

Discussion We can see from these examples that the dynamic parallelization behaviour induced by MLS is not obvious, and that there are surely more interesting patterns to be found. The key lesson here is that 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 favoured 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.

5 Related Work

MLS is a form of thread level speculation (TLS), or speculative multithreading (SpMT), which has been relatively well-studied from a hardware perspective and has been a subject of research for over a decade. Garzaran *et al.* reviewed and classified most of the core TLS approaches [4]. A primary problem in TLS is deciding where to fork speculative child tasks, with systems proposed that operate at the loop level [5], basic-block level [6], and of course at the method level [3].

According to Chen & Olukotun [3], 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 [7]. Hammond, Willey, and

Olukotun later designed the Hydra chip multiprocessor for TLS that included MLS support [8]. Chen & Olukotun concurrently described a more realistic MLS system for Java, which combined a modified version of the Kaffe JVM and JIT compiler running on the Hydra architecture [3]. They found encouraging amounts of speculative parallelism in JIT-compiled code, and noticed that MLS can subsume loop level speculation if loop bodies are extracted into method calls. In previous work we developed an interpreter-based MLS system for Java [1,2], the design of which Schätti later followed to implement a partial MLS system for the HotSpot JVM bytecode interpreter [9].

MLS in general must balance overhead costs with potential parallelism, and can benefit from profile information to guide forking heuristics [10–12]. Return value prediction can further assist MLS by allowing the method continuation to proceed past consumption of the return value [13]; c.f. safe futures which are similar in many respects but do not allow this [14]. A key consideration in all of the work on TLS that our approach here elides is the impact of speculative data dependences. We consider the data dependence problem not unimportant but orthogonal to the one of how to create efficient stack and thread interleavings. In general, the strong isolation properties assumed by speculative threads require some kind of transactional memory (TM) subsystem [15]. MLS and purely transactional approaches differ, however, in that under MLS the speculative execution is not user-specified and is potentially unbounded.

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. In-order speculation is usually supported in hardware, while adding out-of-order execution to a hardware system is complicated but has been argued critical to achieving good speculative performance [16]. On the other hand, out-of-order speculation is much simpler to support in software, and while in-order speculation can significantly increase the choice of fork points, it poses a significant memory allocation problem. For most MLS studies, the lack of precise semantics means that even though a direct mapping to our model almost certainly exists, it can be difficult to nail down. As an example, in their study of MLS fork heuristics for Java, Whaley and Kozyrakis claim to allow speculative threads to create speculative threads, which meets the definition of in-order nesting [10]. However, all of their examples actually demonstrate out-of-order nesting. Similarly, Zhai describes stack management for speculation [17], but does not provide details on the complexities of entering and exiting stack frames speculatively.

Our study also relates to continuation-based parallelization in other contexts. Non-speculatively, Goldstein *et al.* provide an efficient implementation of parallel call that uses a stack abstraction dual to the one presented here: the child thread executes the method and the parent thread executes the continuation [18]; Pillar is a new language that supports this abstraction [19]. Although parallel call was not designed to be speculative, the speculation rules of our system could nevertheless be translated straightforwardly. In a functional context, Mattson, Jr. found that speculative evaluation in Haskell can be supported with low overhead [20]. Ennals and Peyton Jones present a similar optimistic execution system that works together with the lazy evaluation model in Haskell [21]. They provide an operational semantics, but do not model the stack explicitly nor use their semantics to visualize behaviour. Harris & Singh later extended this model to work with the Haskell thunk structure allocated in optimized programs [22].

6 Conclusions and Future Work

Empirical studies of language implementation strategies can only provide so much understanding. For a strategy such as MLS, there is obviously significant performance potential, but the results can be confusing and moreover mired in system-specific performance details. At some point, formalizing the model and exploring behaviour in abstract terms can provide a fresh perspective.

In this work we developed a unified model of MLS that accounts for the major design variations involved in building such a system. The individual sub-models run from completely sequential to highly speculative, and exhaustively cover the core patterns of behaviour encoded in the unified model. This model is valuable purely from a specification standpoint, facilitating system comprehension, comparison, testing, and implementation. Initial work suggests our model is also suitable for a proof of MLS correctness. Once obtained, showing equivalence with other continuation-based parallelization systems can then be used to transfer proof results. Our model finally lends itself to rapid design prototyping of future extensions, which might normally require significant implementation effort to explore.

The other half of this work entails an exploration of MLS behaviour using our model as a tool for insight. We identified some key relationships between program structure, choice of fork point, and resultant speculation behaviour. In some cases we labelled them as good, bad, or inefficient in terms of exposing parallelism, and in others we used them to synthesize desirable higher level behaviours. These experiments demonstrated that all features of the speculation model are useful for creating parallelism, including both in-order and out-of-order nesting, and that robustness and flexibility in an MLS system are important. Our experience here is that accurately predicting how the parallel state will evolve without actually trying scenarios out is overwhelmingly impractical. In the future, automated explorations of this nature may yield further insights. In general, we found that MLS behaviour is fragile, but that if understood it can be beneficially controlled. We believe that maximizing parallelism will require a combination of programmer awareness, compiler transformation, profile information, and judicious static and/or dynamic forking decisions.

Finally, our visualization methods could be equally well applied to actual MLS implementations, including our own Java-based SableSpMT [1, 2]. Our stack diagrams are unique for their compactness, linear scalability, uniform symbol density, lack of overlapping lines, and relation to actual data structures. The state evolutions require only a simple event trace along with unique thread and frame identifiers as inputs. This avenue would be useful for empirical studies of real-world programs.

Acknowledgments

This research was supported by the IBM Toronto Centre for Advanced Studies and the Natural Sciences and Engineering Research Council of Canada. We would like to thank Sam Sanjabi for his insightful comments on an earlier draft of this paper.

References

1. Pickett, C.J.F., Verbrugge, C.: SableSpMT: A software framework for analysing speculative multithreading in Java. In: PASTE'05. (September 2005) 59–66
2. Pickett, C.J.F., Verbrugge, C.: Software thread level speculation for the Java language and virtual machine environment. In: LCPC'05. Volume 4339 of LNCS. (October 2005) 304–318
3. Chen, M.K., Olukotun, K.: Exploiting method-level parallelism in single-threaded Java programs. In: PACT'98. (October 1998) 176–184
4. Garzarán, M.J., Prvulovic, M., Llabería, J.M., Viñals, V., Rauchwerger, L., Torrellas, J.: Tradeoffs in buffering speculative memory state for thread-level speculation in multiprocessors. TACO **2**(3) (September 2005) 247–279
5. Steffan, J.G., Colohan, C., Zhai, A., Mowry, T.C.: The STAMPede approach to thread-level speculation. TOCS **23**(3) (August 2005) 253–300
6. Bhowmik, A., Franklin, M.: A general compiler framework for speculative multithreading. In: SPAA, New York, NY, USA, ACM Press (August 2002) 99–108
7. Oplinger, J.T., Heine, D.L., Lam, M.S.: In search of speculative thread-level parallelism. In: PACT'99. (October 1999) 303–313
8. Hammond, L., Willey, M., Olukotun, K.: Data speculation support for a chip multiprocessor. In: ASPLOS-VIII. (October 1998) 58–69
9. Schätti, G.: Hotspec - a speculative JVM. Master's thesis, ETH, Zürich, Switzerland (January 2008)
10. Whaley, J., Kozyrakis, C.: Heuristics for profile-driven method-level speculative parallelization. In: ICPP'05. (June 2005) 147–156
11. Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., Torrellas, J.: POSH: A TLS compiler that exploits program structure. In: PPoPP'06. (March 2006) 158–167
12. Warg, F.: Techniques to Reduce Thread-Level Speculation Overhead. PhD thesis, Dept. of CSE, Chalmers U. of Tech., Göteborg, Sweden (May 2006)
13. Hu, S., Bhargava, R., John, L.K.: The role of return value prediction in exploiting speculative method-level parallelism. JILP **5** (November 2003) 1–21
14. Welc, A., Jagannathan, S., Hosking, A.: Safe futures for Java. In: OOPSLA'05. (October 2005) 439–453
15. Larus, J.R., Rajwar, R.: Transactional Memory. Morgan & Claypool (December 2006)
16. Renau, J., Tuck, J., Liu, W., Ceze, L., Strauss, K., Torrellas, J.: Tasking with out-of-order spawn in TLS chip multiprocessors: Microarchitecture and compilation. In: ICS'05. (June 2005) 179–188
17. Zhai, A.: Compiler optimization of value communication for thread-level speculation. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA (January 2005)
18. Goldstein, S.C., Schauser, K.E., Culler, D.E.: Lazy threads: Implementing a fast parallel call. JPDC **37**(1) (August 1996) 5–20
19. Anderson, T., Glew, N., Guo, P., Lewis, B.T., Liu, W., Liu, Z., Petersen, L., Rajagopalan, M., Stichnoth, J.M., Wu, G., Zhang, D.: Pillar: A parallel implementation language. In: LCPC'07. Volume 5234 of LNCS. (October 2007) 141–155
20. Mattson, Jr., J.S.: An effective speculative evaluation technique for parallel supercombinator graph reduction. PhD thesis, University of California at San Diego, La Jolla, California, USA (1993)
21. Ennals, R., Jones, S.P.: Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In: ICFP'03. (August 2003) 287–298
22. Harris, T., Singh, S.: Feedback directed implicit parallelism. In: ICFP'07. (October 2007) 251–264