Speculative N-Way Barriers

Lukasz Ziarek Suresh Jagannathan
Department of Computer Science; Purdue University
{lziarek,suresh}@cs.purdue.edu

Matthew Fluet Umut A. Acar
Toyota Technological Institute at Chicago
{fluet,acar}@tti-c.org

Abstract

Speculative execution is an important technique that has historically been used to extract concurrency from sequential programs. While techniques to support speculation work well when computations perform relatively simple actions (e.g., reads and writes to known locations), understanding speculation for multi-threaded programs in which threads may communicate and synchronize through multiple shared references is significantly more challenging, and is the focus of this paper.

We use as our reference point a simple higher-order concurrent language extended with an n-way barrier and a fork/join execution model. Our technique permits the expression guarded by the barrier to speculatively proceed before the barrier has been satisfied (i.e. before all threads that synchronize on that barrier have done so) and to have participating threads that would normally block on the barrier to speculatively proceed as well. Our solution formulates safety properties under which speculation is correct in a fork/join model, and uses traces to validate these properties modularly on a per-thread and per-synchronization basis.

1. Introduction

Speculative execution is an important technique for extracting concurrency from sequential programs. The correctness of a speculation is defined in terms of visibility of values and preservation of dependencies. Validating the correctness of a speculative computation involves *assuming* a state against which the speculation is performed, and *checking* whether this assumed state is consistent with the actual global state when the speculation completes. To ensure that accesses to shared state made by a speculative thread do not incorrectly witness actions that logically follow its execution, these threads typically execute in isolation. A computation following a speculation that does not witness an action performed by the speculation must be re-executed.

Existing speculative execution techniques (10; 8; 16) create concurrent threads from sequential programs. The effects of speculative threads are revoked when conflicts that violate dependencies found in the sequential program are detected. Detection can happen either in hardware or software. Software transactions implemented using optimistic concurrency control (1; 12) also can be viewed as a form of speculation.

However, these mechanisms typically impose strong restrictions on the actions that can be performed within a speculative context, typically limiting them to operations on shared data whose effects can be easily revoked or isolated.

The formalization of speculative execution in multi-threaded programs in which speculative threads are allowed to *safely* communicate with other speculative and non-speculative computations is the focus of this paper. We use as our reference point a simple higher-order concurrent language equipped with an n-way barrier, intended to be used in conjunction with a fork/join execution model. In a fork/join model, child threads are forked from a parent thread. Both child and parent execute in isolation, with their own private copies of the heap. Updates can be propagated among threads at join points (in our case, n-way barriers).

Our solution dynamically records dependence information among all threads participating in an n-way barrier. When a thread synchronizes on a barrier, two pieces of information become available: (1) the *effects* propagated by the threads as a consequence of the synchronization, and (2) the *history* of the thread's actions on shared state up to this point. These histories are maintained on a per-thread and per-barrier basis. A speculative computation also maintains a history of *assumptions* that serve as an execution context for the speculation. Our formulation is agnostic to how these assumptions are chosen. These assumptions are validated against the thread histories of those threads that participate in the corresponding barrier. Validation occurs when a barrier is fully synchronized; it guarantees that the values read by the speculation are consistent with the values written by participating threads, and conversely that the values written by the speculation were not prematurely seen by these threads. When validation fails, execution reverts to the latest *commit* point, a point representing a state guaranteed not to contain the effects of any potentially incorrect speculative actions.

The contributions of this paper are as follows: (1) we present a formal characterization of safe speculation in a higher-order language equipped with an n-way barrier, and in which concurrency is expressed using a fork-join execution model. The validity of the speculative actions performed by threads that join an n-way barrier is determined modularly on a per thread and per speculation basis by using traces; (2) our formulation allows for inter-thread communication. All threads which participate in a speculative barrier explicitly bind values through the pattern and can also implicitly communicate values through updates via shared memory; (3) our treatment does not restrict speculation depth (i.e., a thread may speculatively execute over many barriers). Consequently, there are no limitations on the actions that can be performed within a speculative context: a speculation can initiate other speculative computations, fork new threads, and communicate with other computations (both speculative and non-speculative).

To our knowledge, this is the first attempt to (a) formalize the safety properties of speculative computation for multi-threaded programs in which speculative computations are not required to execute in isolation, and (b) enrich the fork-join execution mode with

speculative support. Besides these theoretical contributions, we believe our results have practical importance, since existing fork-join algorithms can leverage the safety conditions enforced by our semantics to profitably exploit speculation on multi-core platforms.

2. Motivation

Consider how we might write the well-know Floyd-Warshall algorithm to compute all-pairs shortest-path within a fork/join execution model. The algorithm takes as input the edge-weight matrix of a weighted directed graph and returns the matrix of shortest-length paths between all pairs of vertices in the graph. Consider the pseudo-code presented in ML-like syntax given below.

```
fun innerLoop(Matrix, i, j, k) =
  if (j < size(Matrix))
  then let val newpath = get(Matrix, i, k) +
                         get(Matrix, k, j)
       in if (newpath < get(Matrix, i, j)
          then update(Matrix, i, j, newpath);
               innerLoop(Matrix, i, j+1, k)
          else innerLoop(Matrix, i, j+1, k)
       end
  else ()
fun loop(Matrix, i, j, k) =
  if( i < (tid+1) * size(Matrix)/n)
  then innerLoop(Matrix, i, j, k);
       loop(Matrix, i+1, j, k)
  else join n-way barrier;
       loop(Matrix, tid * size(Matrix)/n, 0, k+1)
```

The algorithm forks n threads (code not shown), each with a copy of the matrix. Each thread performs `size(Matrix)/n` calls to `loop`. Before a thread can exit the loop, it must synchronize with all other threads at the barrier point. Unfortunately, the join point creates a bottle neck as it does not consider which threads are data dependent upon one another. Although threads blocked on the barrier may be able safely to proceed once all threads which manipulate entries in the matrix that they require on the subsequent iteration have completed, the synchronization requires instead that they wait until *all* threads signal their completion. Allowing threads to proceed *speculatively* past the barrier can extract additional concurrency without complicating the synchronization within the algorithm.

3. Programming Model

Our programming model is based on a fork-join execution model in which threads synchronize through n-way barriers. We support a dynamic number of forked threads. Threads execute with local stores, initially inherited from their parent. Besides creating and manipulating references, threads can create and synchronize (i.e., join) on barriers. We express barrier synchronization using pattern matching. The barrier expression $\texttt{match}(\overline{\texttt{p}}) \rightarrow \texttt{e}$ waits for the patterns in $\overline{\texttt{p}}$ to be satisfied and then executes `e` in a new thread of control. If there are no patterns, then `e` represents a computation that executes asynchronously with respect to the computation that evaluated the match expression. (Thus, a match with an empty pattern can be used to unconditionally fork a thread.) The result of a match is of type `unit`, since the body evaluates in a separate thread of control. Barriers are used as the only communication medium as explained below.

A pattern `p` is either a conjunction or disjunction of *base patterns*[1]. The base pattern `rd(l)` causes the enclosing match expression to block waiting for another thread to read from location `l`. The base pattern `wr(l)` causes the enclosing match expression to block waiting for a thread to write to location `l`. The base pattern `wr(l, v)` behaves similarly, except that it causes its enclosing match expression to block until another thread writes value `v` to location `l`. Finally, the base pattern `wr?(l, v)` reads the contents of the location `l` – if the location contains `v`, the pattern has no effect; otherwise, it causes the enclosing match expression to block until another thread writes `v` to `l`. A thread can only join on one pattern in a given barrier, thus a barrier composed of n base patterns will wait for n threads to join.

A base pattern becomes *active* when a thread matches against it (i.e. creates a barrier). Other threads participate in the pattern by reading or writing to locations specified in the base patterns which comprise the overall pattern. Thus, a thread that attempts to read from location `l` (`!l`) causes it to participate with the base pattern `rd(l)` in a currently blocked match expression. Similarly, a thread that attempts to write to location `l` (`l := v`) causes it participate with the base pattern `wr(l)`. A thread performing a read or write action on which several patterns are blocked is obligated to join with one, by participating in the pattern's barrier.

Since barriers are synchronous, the evaluation of the match body as well as all threads participating in the barrier's pattern are delayed until the pattern is satisfied. Consider the expression

$$\texttt{match (rd(l)} \wedge \texttt{wr(l))} \Rightarrow \texttt{e}$$

executed by thread T. If another thread T_1 attempts to perform a read of location `l` (by evaluating `!l`), then T_1 can join with `rd(l)`. At this point, T_1 participates in the barrier and is blocked until another thread T_2 performs a write to location `l` and joins the pattern `wr(l)`. When this occurs, T_1 and T_2 can resume execution, and `e` is evaluated in a new thread of control. When T_1 resumes, the contents of `l` it dereferences is the value stored by T_2. This example illustrates a simple handoff of a value through location `l` that ensures that every write to `l` performed by some thread is *always* witnessed by some other thread, a property difficult to ensure using ordinary shared-memory primitives. A program in which no such read occurs will not terminate since the barrier will not complete.

Conjunctive patterns cause the executing match expression to be blocked until all are joined, while a disjunctive pattern causes the executing match pattern to be blocked as long as none are joined. One way to encode a non-deterministic choice between conjunctive patterns is to simply match on multiple conjunctive patterns.

Threads execute with local stores, and propagate store bindings when they join a synchronization pattern. Threads can communicate values from their thread-local store through pattern matches. A new store is synthesized from the thread-local stores of all the participating threads in a pattern-match expression. This store becomes the thread-local store for all participating threads as well as the thread executing the match expression. (The body of a match with an empty pattern (i.e., a fork) is given an unmodified copy of its parent's thread-local store.) Therefore, a pattern match acts as both a barrier and a communication medium.

Every thread that joins a pattern contributes its store bindings to a synthesized store. When a thread matches on a write pattern (`wr(l)`), its binding for `l` is recorded in the synthesized store, along with all other bindings *except* for those on locations associated with earlier matched write patterns. When a thread matches on a read pattern, its store bindings are added to the synthesized store, with the same caveat on locations with previously established write bindings. A thread which joins a pattern, and contributes a set of bindings will have these bindings supersede the bindings for these locations provided earlier by other threads that have joined the pattern. When a pattern is satisfied, the synthesized store becomes

[1] To support a combination of conjunctive and disjunctive patterns, a mechanism similar to transactional events is required (5).

the thread-local store for all threads that joined the pattern and the match-expression body. Note that a thread that joins a read pattern blocks until the pattern is satisfied; when it resumes the value it reads is the value in the new store that reflects the contributions provided by the threads participating in the pattern. Thus, pattern matching serves as both a synchronization and a binding device.

3.1 Speculative Execution Model

A *speculative* barrier alters the synchronous semantics by allowing the body of a match expression to be executed before all base patterns have been joined. Moreover, the thread that participates in a barrier need not block until all other patterns in the match expression are satisfied. Thus, speculation occurs both in the expression protected by the barrier as well as in all threads which participate with it. The speculative actions executed by a thread T become non-speculative once all speculative barriers with which T is associated have all their patterns satisfied.

The thread-local store of the thread that initiates a speculation, (i.e., the thread that executes the body of a match expression) must be consistent with the store that would have existed had the speculation not happened, and the thread blocked. Clearly, this store should contain bindings for locations referenced by base patterns in barrier. Locations referenced by the speculative computation but not explicit in the pattern also require store bindings. The correctness of a joined set of speculative actions is determined when all participants of the barrier are known. When the last participant joins the barrier, the speculative actions of all participants are validated. The validation check determines if the chosen store was correct. To do this, the thread-local stores of all participants prior to joining the barrier are examined. If the store chosen for the speculation can be synthesized from these thread local stores without violating ordering dependencies, the check succeeds.

3.2 Examples

To illustrate these ideas, we depict possible executions of the programs shown in Fig. 1 in Fig. 2 and Fig. 3. In the examples, thread executions are depicted as solid black lines. Dashed lines represent blocked computation, and dashed boxes enclose speculative computation. Solid gray circles show validation points where a speculative computation is either committed or unrolled. A reversion action is defined by a solid double black arrow. The body of a match expression is shown as a square following the barrier. Data dependencies, shown as dotted white arrows, occur between read and write operations and patterns. All code fragments (Fig. 1) utilize three threads of control. Thread `T1` establishes barriers and threads `T2` and `T3` communicate with `T1` explicitly through shared locations `x` and `y` as well as implicitly through locations `z` and `w`.

Consider the execution of Fig. 1(a) using synchronous evaluation of barriers as shown in Fig. 2(a). Initially `T1` executes a match on conjunctive pattern consisting of `wr(x)` and `wr(y)`. The thread `T1` will block until writes to `x` and `y` occur. Next, thread `T3` updates the shared variable `z` with the value 3. Since `z` is not specified in the pattern, the write does not join with the barrier in `T1`. Thread `T2` subsequently writes the value 1 to `x`. Because the location `x` is specified in the pattern on which `T1` is blocked, the write can join with the pattern. Now both `T2` and `T1` are blocked waiting for the base pattern `wr(y)` to be satisfied. Eventually, thread `T3` updates the location `y` and joins the pattern `wr(y)`. Since `T3`'s join satisfies the entire barrier, all three threads become unblocked. A consistent store is synthesized for all three participants with $\{\,\texttt{x} \mapsto 1,\ \texttt{y} \mapsto 2,\ \texttt{z} \mapsto 3\}$. All three threads resume their executions with a local store reflecting these bindings. Thus, the body of `T1`'s barrier will read the values 1, 2, and 3.

Now, consider what happens when we allow threads `T1` and `T2` to proceed speculatively past the barrier boundary (see Fig. 2(b)). As before, `T1` establishes the conjunctive pattern on the locations `x` and `y`. However, it no longer blocks waiting for these base patterns to be satisfied. To allow speculative execution of the match body, a new thread-local store must be chosen that captures store bindings for the locations referenced in the pattern and match body. Suppose we correctly choose a thread-local store for `T1` such that $\{\,\texttt{x} \mapsto 1,\ \texttt{y} \mapsto 2,\ \texttt{z} \mapsto 3\}$. With this store, `T1` can safely execute past the match expression. When `T2`'s write joins the pattern, it too speculatively proceeds past its write, operating on the store chosen for the speculation. When thread `T3` writes to `y`, the pattern once again is satisfied. At this point, the validation check confirms that the chosen store bindings are consistent with the thread-local stores of `T1`, `T2`, and `T3` immediately prior to their matches on the pattern, and thus the behavior program under speculative evaluation is no different than its behavior under synchronous evaluation. The validation check is successful and the speculative executions of `T1` and `T2` become non-speculative.

In general, the choice of store bindings made when a speculation is initiated may not be correct. The execution depicted in Fig. 2(c) shows what happens when a speculation executes under an incorrect store. Consider what occurs when thread `T3` writes the value 2 to location `y` prior to the establishment of the barrier in `T1`. After this write `T1` executes the conjunctive pattern of `wr(x)` and `wr(y)`. Once again, it speculates beyond the barrier, and as before chooses a thread-local store with $\{\,\texttt{x} \mapsto 1,\ \texttt{y} \mapsto 2,\ \texttt{z} \mapsto 3\}$. Similarly, when thread `T2`'s subsequent write matches the pattern, it too speculatively proceeds past the barrier boundary with this chosen store. Unfortunately, thread `T3` writes the value 42 to `y`. Since this write matches the pattern, the pattern is satisfied and validation of all speculative actions triggered based on this pattern must be performed. Notice that the thread local store chosen for the speculation cannot be synthesized from the local stores of `T1`, `T2`, and `T3` extant at the point they participated in the barrier.

Now, we can redo `T1` and `T2`'s computations to consider this new binding for `y`. Since `T2`'s speculation did not depend on `y`'s value its execution can be marked as non-speculative. Thread `T1`'s computation, however, must be re-evaluated with `y`'s new value. Once `T1`'s computation is complete it also becomes non-speculative. Notice we can resume `T2` before `T1`'s re-execution is complete since there is no synchronization between the two threads after the barrier.

Speculations can be nested. Consider the example given in Fig. 1(b). Thread `T1` is extended to create a second barrier after the first. Consider this program's execution as shown in Fig. 3(a). Once again, we consider the scenario in which `T3` writes 2 to location `y` prior to the match expression in `T1`. Threads `T1` and `T2` will begin speculation past the barrier using a store with $\{\,\texttt{x} \mapsto 1,\ \texttt{y} \mapsto 2,\ \texttt{z} \mapsto 3\}$. During its speculation, `T1` writes to a shared location `w`. The value stored into this location is the value of `y` in `T1`'s local store, namely 2. Thread `T1` subsequently executes another barrier and again proceeds speculatively. Thread `T2` speculatively reads from location `z`. This satisfies the second barrier established by `T1`. Since `T2` is speculative, this barrier's ultimate validation depends on the validation of the first barrier. Thread `T2` then reads the value 2 from location `w`. This value was propagated to `T2` when it matched on the second barrier established by `T1`. Now, thread `T3` writes the value 42 to `y` and satisfies the first barrier. We must validate any speculations based on this barrier, namely those in `T1` and `T2`. As before, the thread-local store chosen for the speculation cannot be synthesized from the local stores of `T1`, `T2`, and `T3` immediately prior to their participating in the barrier since the binding for location `y` is incorrect. Thread `T1`'s computation must be adjusted to the new value of `y`. This requires a new binding for `w`, which in turn invalidates `T2`'s speculation.

```
T1 =  ...
      match(wr(x) ∧ wr(y)) ⇒
        !x; !y; !z;
T2 =  ...
      x := 1;
      ...
T3 =  ...
      z := 3;
      ...
      y := 2;
      ...
      y := 42;
      ...
```

(a) Program for Fig. 2

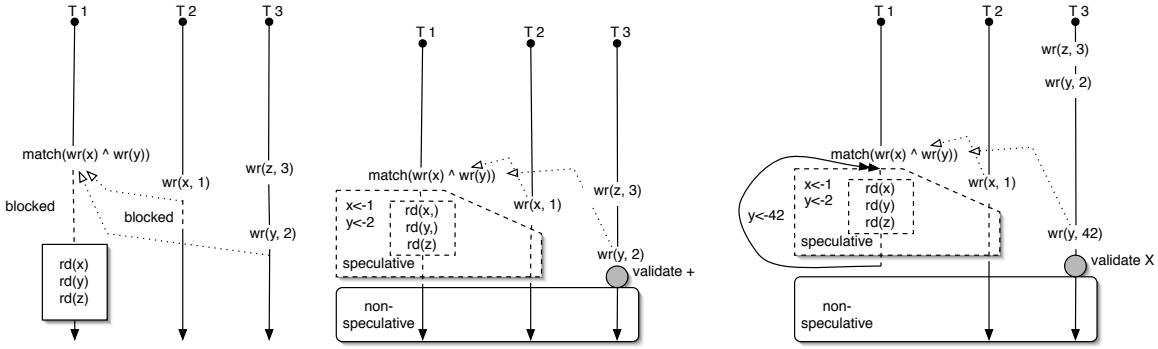
```
T1 =  ...
      match(wr(x)∧wr(y)) ⇒
        !x; !y; w := !y;
      ...
      match(rd(z)) ⇒ ...
T2 =  ...
      x := 1;
      ...
      ! z;
      ! w;
T3 =  ...
      y := 2;
      ...
      y := 42;
      ...
```

(b) Program for Fig. 3(a)

```
T1 =  ...
      match(wr(x)∧wr(y)) ⇒
        !x; !y;
      ...
      match(rd(z)) ⇒ ...
T2 =  ...
      x := 1;
      ...
      !z;
      !w;
T3 =  ...
      y := 2;
      w := 3;
      y := 42;
      ...
```

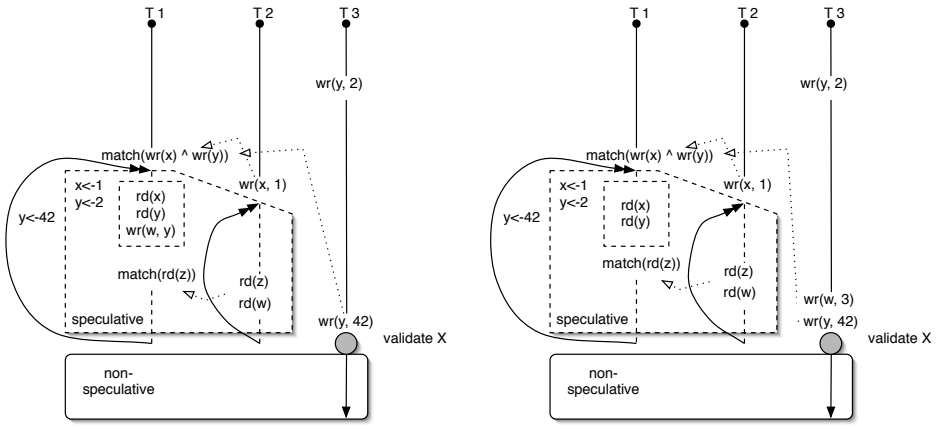
(c) Program for Fig. 3(b)

Figure 1.



(a) Synchronous Execution (b) Speculative Execution (c) Speculative Execution w/ Rollback

Figure 2.



(a) (b)

Figure 3. Two programs which require a rollback not only of the thread initiating the pattern match, but threads which have participated in the pattern and continued their execution speculatively.

A cascading rollback can also occur if a speculative thread is obligated to witness a specific value for a location when the barrier is satisfied. Consider the example program Fig. 1(c) and an execution in Fig. 3(b). Instead of T1 performing a write to location w, thread T3 writes to w *after* the speculative read occurs in T2. Unfortunately T2 is obligated to see T3's write to w since it would have executed prior to T2's read under synchronous evaluation. Writes to shared locations are naturally ordered in T3; similarly, T2's speculative actions *must* be ordered logically after the barrier. When T3 participates in the pattern, its writes become ordered *prior* to the barrier containing the pattern. Thus, the read from w in T2 logically executes after the write of w in T3.

4. Semantics

Our semantics is defined in terms of a core call-by-value higher-order language with first-class references, threads, and synchronization patterns. For perspicuity, we first present a semantics in which all synchronization patterns are synchronous. We then extend this language to support speculation past synchronization points. Updates to shared data by a thread become visible at synchronization points.

In the following, we write $\overline{\alpha}$ to denote a sequence of zero or more elements of kind α, $A.B$ to denote sequence concatenation, and ϕ to denote an empty sequence. Metavariables x and y range over variables, e ranges over expressions, v ranges over values, l ranges over locations, p ranges over base patterns, t ranges over thread identifiers, and γ ranges over pattern identifiers.

Expressions can be values, variables, applications, operations to create, read, or write locations, or a synchronization-pattern match. A value is either unit, an abstraction, a location, or a synchronization pattern. A base pattern p can either be a read pattern ($\texttt{rd}(\texttt{l})$) or a write pattern ($\texttt{wr}(\texttt{l})$). A read pattern causes the evaluating thread to block until another thread reads location l, and a write pattern causes the evaluating thread to block until another threads writes location l. We do not consider more expressive base patterns such as those described informally in Section 3, but their addition poses no complications to the semantics.

Base patterns can be aggregated to form a synchronization patterns. To simplify the presentation, we only consider synchronization patterns with conjunctive semantics; extending the semantics with disjunctive synchronization patterns is straightforward. An empty base-pattern denotes a synchronization pattern that is immediately satisfied.

4.1 Synchronous Evaluation

The syntax of the language is given in Fig. 4 and synchronous evaluation rules are given in Fig. 5. A program state π is a collection of threads. (Program states are equivalent modulo the associativity and commutativity of $\pi \| \pi$.) Each thread $\texttt{t}[\texttt{e}, \sigma]$ includes a thread identifier (t), an expression (e), and a thread-local store (σ). Our synchronous evaluation is formulated with respect to fork/join execution model: each thread maintains its own image of the (implicit) global store and propagates its updates when it joins with a synchronization pattern.

Global evaluation is specified via a relation ($\Rightarrow$) that maps a program state (π) and a pattern environment (ρ) to another program state and pattern environment. We write $\Rightarrow^*$ to denote the reflexive, transitive closure of this relation. Many of the rules decompose the process state into a distinguished thread t and remaining processes π as $\texttt{t}[E[e], \sigma] \| \pi$, where the thread's expression is further decomposed into an evaluation context E and active expression e. The rules APP and REF do not deal with synchronization patterns and are standard: function application (APP) substitutes the argument value for free occurrences of the parameter in the body of the abstraction, and evaluating a $\texttt{ref}(\texttt{v})$ expression (REF) creates a new location in the thread's local store. (Note that the new location l is globally fresh with respect to all thread-local stores.)

The pattern environment maps a *pattern identifier*, a unique identifier created whenever a synchronization-pattern match is evaluated, to a 5-tuple, $\langle \texttt{p}, \texttt{t}_b[\texttt{e}_b], \overline{\texttt{t}_r[\texttt{l}_r, E_r]}, \overline{\texttt{t}_w[\texttt{l}_w, \texttt{v}_w, E_w]}, \overline{\sigma'} \rangle$. The first component represents unsatisfied base patterns of the original synchronization pattern. The body of a synchronization-pattern match and any thread that joins with a synchronization pattern are blocked until the synchronization pattern becomes satisfied (i.e., until all base patterns are joined). The second, third, and fourth components represent blocked threads. The $\texttt{t}_b[\texttt{e}_b]$ component corresponds to the body of the synchronization-pattern match. The $\overline{\texttt{t}_r[\texttt{l}_r, E_r]}$ component corresponds to threads that have joined the synchronization pattern via a read pattern; these threads will be resumed with values read from the locations $\overline{\texttt{l}_r}$. The $\overline{\texttt{t}_w[\texttt{l}_w, \texttt{v}_w, E_w]}$ component corresponds to threads that have joined the synchronization pattern via a write pattern; these threads will be resumed with unit values and will contribute the bindings $\overline{\texttt{l}_w \mapsto \texttt{v}_w}$ to the new store synthesized by the synchronization-pattern match. The fifth component $\overline{\sigma'}$ corresponds to the thread-local stores of all the threads that have joined with the pattern (in the order in which they joined with the pattern).

A read or write operation that takes place when there is no synchronization pattern waiting for a join on that operation, simply reads from or updates the thread-local store (READ and WRITE). We determine whether there is an active synchronization pattern by consulting the patten environment.

Evaluating a synchronization-pattern match augments the pattern environment with the synchronization pattern and the (blocked) body of the match (PMATCH). Since the body of the synchronization-pattern match will be evaluated in a new thread of control, a fresh thread identifier is generated. Every time a thread joins with an active synchronization pattern (i.e., a pattern found in the pattern environment), the joined base pattern is removed from the pattern in the pattern environment. When all base patterns for a synchronization pattern have been satisfied, the synchronization pattern becomes empty. At this point, the body of the synchronization-pattern match can be evaluated and all threads that have joined with this pattern can be resumed (PDONE). All threads resume execution with a local-store that is synthesized by merging the stores of all the participating threads, as described in Section 3. We omit a precise definition of $\mathsf{Merge}(\overline{\sigma'}, \overline{\texttt{l}_w \mapsto \texttt{v}_w})$; we simply require $\mathsf{Merge}(\overline{\sigma'}, \overline{\texttt{l}_w \mapsto \texttt{v}_w})$ to denote a proper function from a collection of stores and writes to a store that represents an appropriate propagation of store updates. Note that threads that have joined with the synchronization pattern via a read pattern are resumed with the value present at the appropriate location in the merged store, while threads that have joined with the synchronization pattern via a write pattern are resumed with the unit value.

Unconditionally spawning a thread to evaluate the expression e can be simulated by $\texttt{match}(\phi) \rightarrow \texttt{e}$. Note that the synchronization pattern is immediately satisfied, and the body e will begin executing with a fresh thread identifier and with a local store inherited from the parent thread (i.e., we reasonable assume that $\mathsf{Merge}(\phi.\sigma, \phi) = \sigma$).

Threads can join with synchronization patterns through rule PREAD or PWRITE. These rules remove the base pattern with which they match from the synchronization pattern with which they join (we write $\overline{\texttt{p}} \setminus i$ to denote pattern $\overline{\texttt{p}}$ with the i^{th} base pattern in $\overline{\texttt{p}}$ removed). When a thread joins with a read pattern, the location, context, and local store of the joining thread is recorded in the

SYNTAX:

$$\begin{array}{rcl} \mathtt{e} \in \mathit{Exp} & ::= & \mathtt{v} \mid \mathtt{x} \mid \mathtt{e(e)} \mid \mathtt{ref(e)} \mid \mathtt{!e} \mid \mathtt{e := e} \mid \mathtt{match(e)} \rightarrow \mathtt{e} \\ \mathtt{v} \in \mathit{Val} & ::= & \mathtt{unit} \mid \lambda\,\mathtt{x.e} \mid \mathtt{l} \mid \overline{\mathtt{p}} \\ \mathtt{p} \in \mathit{Pat} & ::= & \mathtt{rd(l)} \mid \mathtt{wr(l)} \end{array}$$

STORES:

$$\sigma \quad ::= \quad \{\mathtt{l} \mapsto \mathtt{v}, \ldots\}$$

PROCESSES:

$$\pi \quad ::= \quad \pi \| \pi \mid \mathtt{t}[\mathtt{e}, \sigma]$$

EVALUATION CONTEXTS:

$$E \quad ::= \quad [\,] \mid E(\mathtt{e}) \mid \mathtt{v}(E) \mid \mathtt{ref}(E) \mid \mathtt{!}E \mid E := \mathtt{e} \mid \mathtt{l} := E \mid \mathtt{match}(E) \rightarrow \mathtt{e}$$

Figure 4. Syntax, stores, processes, and evaluation contexts.

PATTERN ENVIRONMENTS:

$$\rho \quad ::= \quad \{\gamma \mapsto \langle \overline{\mathtt{p}}, \mathtt{t}_b[e_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_r]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]}, \overline{\sigma'} \rangle, \ldots\}$$

SYNCHRONOUS EVALUATION:

APP

$$\frac{}{\mathtt{t}[E[(\lambda\,\mathtt{x.e})(\mathtt{v})], \sigma] \| \pi, \rho \Rightarrow \mathtt{t}[E[\mathtt{e}[\mathtt{x}/\mathtt{v}]], \sigma] \| \pi, \rho}$$

REF

$$\frac{\mathtt{l}\ \mathit{fresh}}{\mathtt{t}[E[\mathtt{ref}(\mathtt{v})], \sigma] \| \pi, \rho \Rightarrow \mathtt{t}[E[\mathtt{l}], \sigma \uplus \{\mathtt{l} \mapsto \mathtt{v}\}] \| \pi, \rho}$$

READ

$$\frac{\begin{array}{c}\sigma(\mathtt{l}) = \mathtt{v} \\ \forall \gamma \in \mathit{Dom}(\rho) \qquad \rho(\gamma) = \langle \overline{\mathtt{p}}, \_, \_, \_, \_ \rangle \\ 1 \leq |\overline{\mathtt{p}}| = n \qquad \forall i \in \{1 \ldots n\} \qquad \mathtt{p}_i \neq \mathtt{rd(l)}\end{array}}{\mathtt{t}[E[!\mathtt{l}], \sigma] \| \pi, \rho \Rightarrow \mathtt{t}[E[\mathtt{v}], \sigma] \| \pi, \rho}$$

WRITE

$$\frac{\begin{array}{c}\forall \gamma \in \mathit{Dom}(\rho) \qquad \rho(\gamma) = \langle \overline{\mathtt{p}}, \_, \_, \_, \_ \rangle \\ 1 \leq |\overline{\mathtt{p}}| = n \qquad \forall i \in \{1 \ldots n\} \qquad \mathtt{p}_i \neq \mathtt{wr(l)}\end{array}}{\mathtt{t}[E[\mathtt{l} := \mathtt{v}], \sigma] \| \pi, \rho \Rightarrow \mathtt{t}[E[\mathtt{unit}], \sigma[\mathtt{l} \mapsto \mathtt{v}]] \| \pi, \rho}$$

PMATCH

$$\frac{\mathtt{t}_b, \gamma\ \mathit{fresh}}{\begin{array}{l}\mathtt{t}[E[\mathtt{match}(\overline{\mathtt{p}}) \rightarrow \mathtt{e}_b], \sigma] \| \pi, \rho \Rightarrow \\ \quad \mathtt{t}[E[\mathtt{unit}], \sigma] \| \pi, \rho \uplus \{\gamma \mapsto \langle \overline{\mathtt{p}}, \mathtt{t}_b[\mathtt{e}_b], \phi, \phi, \phi.\sigma \rangle\}\end{array}}$$

PDONE

$$\frac{\mathsf{Merge}(\overline{\sigma'}, \overline{\mathtt{l}_w \mapsto \mathtt{v}_w}) = \sigma''}{\begin{array}{l}\pi, \rho \uplus \{\gamma \mapsto \langle \phi, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}, E_r]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]}, \overline{\sigma'} \rangle\} \Rightarrow \\ \quad \mathtt{t}_b[\mathtt{e}_b, \sigma''] \| \mathtt{t}_r[E_r[\sigma''(\mathtt{l}_r)], \sigma''] \| \mathtt{t}_w[E_r[\mathtt{unit}], \sigma''] \| \pi, \rho\end{array}}$$

PREAD

$$\frac{1 \leq |\overline{\mathtt{p}}| = n \qquad i \in \{1 \ldots n\} \qquad \mathtt{p}_i = \mathtt{rd(l)}}{\begin{array}{l}\mathtt{t}[E[!\mathtt{l}], \sigma] \| \pi, \rho \uplus \{\gamma \mapsto \langle \overline{\mathtt{p}}, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_r]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]}, \overline{\sigma'} \rangle\} \Rightarrow \\ \quad \pi, \rho \uplus \{\gamma \mapsto \langle \overline{\mathtt{p}} \setminus i, \mathtt{t}_b[\mathtt{e}_b], \mathtt{t}_r[\mathtt{l}_r, E_r].\mathtt{t}[\mathtt{l}, E], \mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w], \sigma'.\sigma \rangle\}\end{array}}$$

PWRITE

$$\frac{1 \leq |\overline{\mathtt{p}}| = n \qquad i \in \{1 \ldots n\} \qquad \mathtt{p}_i = \mathtt{wr(l)}}{\begin{array}{l}\mathtt{t}[E[\mathtt{l} := \mathtt{v}], \sigma] \| \pi, \rho \uplus \{\gamma \mapsto \langle \overline{\mathtt{p}}, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_r]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]} \rangle\} \Rightarrow \\ \quad \pi, \rho \uplus \{\gamma \mapsto \langle \overline{\mathtt{p}} \setminus i, \mathtt{t}_b[\mathtt{e}_b], \mathtt{t}_r[\mathtt{l}_r, E_r], \mathtt{t}_w[E_w, \sigma_w].\mathtt{t}[\mathtt{l}, \mathtt{v}, E], \overline{\sigma'}.\sigma \rangle\}\end{array}}$$

Figure 5. Synchronous evaluation rules.

TRACES:

$$\omega \quad ::= \quad \frac{\beta}{\mathtt{t}} \mid \frac{\mathtt{v} \Uparrow \mathtt{l}}{\mathtt{t}} \mid \frac{\mathtt{l} \rightarrow \mathtt{v}}{\mathtt{t}} \mid \frac{\mathtt{l} \leftarrow \mathtt{v}}{\mathtt{t}} \mid \frac{\overline{\mathtt{p}} \rightarrow_\gamma \mathtt{t}_b}{\mathtt{t}} \mid \frac{\gamma}{} \mid \frac{\mathtt{l} \rightarrow_{\gamma,i}}{\mathtt{t}} \mid \frac{\mathtt{l} \leftarrow_{\gamma,i} \mathtt{v}}{\mathtt{t}} \qquad\qquad \Omega \quad ::= \quad \overline{\omega}$$

Figure 6. Traces

pattern environment. (This allows the thread to be resumed with the value stored at `l` in the store that exists *after* the synchronization pattern is satisfied.) When a thread joins with a write pattern, it updates its local store and then the context and local store of the joining thread is recorded in the pattern environment. Joins occur eagerly – a read or write operation must join with a waiting read or write pattern if one exists in the pattern environment. If multiple such patterns exist, the pattern with which the thread joins is chosen non-deterministically.

4.2 Speculative Evaluation

We now wish to revise the synchronous-evaluation semantics given above to a speculative-evaluation semantics. We specify a new global evaluation relation ($\rightsquigarrow$) that maps a program state (π), a speculative pattern environment (ψ), a *committed trace* (Ω_C), and a *speculative trace* (Ω_S) to another program state, speculative pattern environment, committed trace, and speculative trace. We write $\rightsquigarrow^*$ to denote the reflexive, transitive closure of this relation.

A speculative pattern environment maps a pattern identifier to a 6-tuple, $\langle \mathtt{p}, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_r]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]}, \overline{\sigma'}, \sigma'' \rangle$. The first five components are similar to those in a synchronous pattern environment; they represent the unsatisfied base patterns of the original synchronization pattern the (notionally) blocked threads corresponding to the body of the synchronization-pattern match, readers, and writers, and the thread-local stores of all the threads that have joined with the pattern. The speculative semantics will allow a computation to proceed speculatively into the body of a speculative-pattern match and past a read or write that joins with a pattern; such a computation does so in the context of a local-store that *assumes* the values of the store that will exist after the synchronization pattern is satisfied (i.e., the store that is the merging of the local-stores of all the thread that ultimately join with this synchronization pat-

tern). The sixth component in the range of the speculative pattern environment is this assumed store.

A committed trace and a speculative trace are conservative approximations of the *history* of actions performed by threads (see Fig. 6). Theses traces are used to validate the behavior of speculative computations. Trace actions record the β-reduction of thread $\mathtt{t}$ ($\frac{\beta}{\mathtt{t}}$), the creation of a reference of location $\mathtt{l}$ with initial value $\mathtt{v}$ by thread $\mathtt{t}$ ($\frac{\mathtt{v}\Uparrow\mathtt{l}}{\mathtt{t}}$), the (non-joining) reading of a value $\mathtt{v}$ from location $\mathtt{l}$ by thread $\mathtt{t}$, ($\frac{\mathtt{l}\rightarrow\mathtt{v}}{\mathtt{t}}$), and the (non-joining) writing of a value $\mathtt{v}$ to location $\mathtt{l}$ by thread $\mathtt{t}$ ($\frac{\mathtt{l}\leftarrow\mathtt{v}}{\mathtt{t}}$). Operations that join a synchronization pattern are also recorded: the reading from location $\mathtt{l}$ by thread $\mathtt{t}$ that satisfies the i^{th} base pattern of the pattern identifier γ ($\frac{\mathtt{l}\rightarrow\gamma,i}{\mathtt{t}}$) and the writing of a value $\mathtt{v}$ to location $\mathtt{l}$ by thread $\mathtt{t}$ that satisfies the i^{th} base pattern of the pattern identifier γ ($\frac{\mathtt{l}\leftarrow\gamma,i\mathtt{v}}{\mathtt{t}}$). Finally, the start and end of synchronization-pattern matches are recorded: the start of the match on the synchronization pattern $\overline{\mathtt{p}}$ by thread $\mathtt{t}$, with pattern identifier γ and thread identifier $\mathtt{t}_b$ for the match body ($\frac{\overline{\mathtt{p}}\rightarrow\gamma\mathtt{t}_b}{\mathtt{t}}$) and the end of the synchronization-pattern match with pattern identifier γ ($\frac{\gamma}{\mathtt{t}}$).

The speculative-evaluation semantics makes use of two relations defined on traces to validate the behavior of speculative computations. The first, written $\Omega_1 \leftrightharpoons \omega_2$, asserts that the trace Ω_1 can be exchanged with the trace action ω_2.

$$\frac{}{\phi \leftrightharpoons \omega_2} \qquad \frac{\Omega_1 \leftrightharpoons \omega_2 \qquad \omega_1 \leftrightharpoons \omega_2}{\Omega_1.\omega_1 \leftrightharpoons \omega_2}$$

It is defined by induction on the trace Ω_1 making use of the auxiliary relation $\omega_1 \leftrightharpoons \omega_2$. This relation, which asserts that the trace action ω_1 can be exchanged with the trace action ω_2, holds when the thread identifiers and pattern identifiers in ω_1 are disjoint from the thread identifiers and pattern identifiers in ω_2. Intuitively, $\omega_1 \leftrightharpoons \omega_2$ holds when the action denoted by ω_2 could have been performed before the action denoted by ω_1. Requiring the thread identifiers to be disjoint ensures that the actions of a single thread cannot be exchanged and the actions of a thread cannot be exchanged with its creation (by a synchronization-pattern match). Similarly, requiring the pattern identifiers to be disjoint ensures that a thread's joining with a pattern cannot be exchanged with the start or end of the synchronization-pattern match.

The second relation defined on traces, written $\kappa(\Omega) = (\varpi, \varrho)$, asserts that the trace Ω can be *simulated* to yield a simulated program state (ϖ, denoting executing threads) and simulated pattern environment (ϱ, denoting active synchronization patterns and blocked threads). Trace simulation is similar to the synchronous-evaluation relation, except that a thread's expression and local-store are elided and all non-determinism is resolved by the trace. Space precludes giving the definition of the $\kappa(\Omega) = (\varpi, \varrho)$ relation.[2] However, we note that $\kappa(\phi) = (\mathtt{m}, \emptyset)$, where $\mathtt{m}$ is a (distinguished) thread identifier used for the *main* thread of the program. Because all non-determinism is dictated by the trace Ω, $\kappa(\Omega)$ denotes a partial function from a trace to a simulated program state and a simulated pattern environment.

The evaluation rules for our speculative extension are given in Fig. 7. Recall that the relation takes the form $\pi, \psi, \Omega_C, \Omega_S \rightsquigarrow \pi'', \psi'', \Omega''_C, \Omega''_S$. Intuitively, Ω_C and Ω''_C represent actions that have been committed; that is, actions that are known to correspond to a valid synchronous evaluation. Similarly, Ω_S and Ω''_S represent actions that have been speculatively executed; that is, actions that are not yet known to correspond to a valid synchronous execution. During speculative evaluation, actions are initially appended to the speculative trace. As actions can be validated, they are removed from the speculative trace and appended to the committed trace.

Since the rules APP and REF do not deal with synchronization patterns, they are similar to those from the synchronous evaluation. In addition, the speculative trace is appended with an appropriate trace action ($\frac{\beta}{\mathtt{t}}$ or $\frac{\mathtt{v}\Uparrow\mathtt{l}}{\mathtt{t}}$).

A read or write operation may simply read or update the thread-local store (READ and WRITE), while appending an appropriate trace action ($\frac{\mathtt{l}\rightarrow\mathtt{v}}{\mathtt{t}}$ or $\frac{\mathtt{l}\leftarrow\mathtt{v}}{\mathtt{t}}$) to the speculative trace. Unlike the corresponding synchronous-evaluation rules, there is no requirement that there is no synchronization pattern waiting for a join on the read or update. Intuitively, this is because the speculative pattern environment includes active synchronization patterns from the speculative execution of threads. Hence, an active synchronization pattern in the speculative pattern environment may correspond to a synchronization pattern from the "future", with which the present thread could not join.

Evaluating a synchronization-pattern match augments the pattern environment with the synchronization pattern and the body of the match (SMATCH), while appending a trace action ($\frac{\overline{\mathtt{p}}\rightarrow\gamma\mathtt{t}_b}{\mathtt{t}}$) to the speculative trace. However, unlike the corresponding synchronous-evaluation rule, the speculative-evaluation rule immediately includes a thread to evaluate the body of the synchronization-pattern match in the program state. This thread is executed with a new store (σ'') that is the assumed store that will exist after the synchronization pattern is satisfied. This assumed store is also recorded in the speculative pattern environment, along with the synchronization pattern and the body of the match, to be validated in the future. A portion of this validation occurs when the synchronization pattern in the speculative pattern environment becomes empty (i.e., when all base patterns for a synchronization pattern have been satisfied) (SDONE). At this point, all threads that have joined with this pattern have been recorded in the speculative pattern environment. Thus, the rule asserts that the assumed store is, in fact, the one derived by merging the stores of all the participating threads ($\mathsf{Merge}(\overline{\sigma'}, \overline{\mathtt{l}_w \mapsto \mathtt{v}_w}) = \sigma''$). In addition, the rule appends a trace action ($\frac{\gamma}{\mathtt{t}}$) to the speculative trace. Note that this rule only *speculatively* completes the synchronization-pattern match; that is, it validates the merging of the (*speculative*) thread-local stores of the participating threads, but does not validate the global consistency of the completed synchronization-pattern match with respect to the actions of other threads. This consistency is validated by the SCOMMIT rule (described below).

Threads can join with synchronization patterns through the rules SREAD and SWRITE. Like the corresponding synchronous-evaluation rules, these rules remove the base pattern with which they match from the synchronization pattern with which they join. However, unlike the corresponding synchronous-evaluation rules, the speculative-evaluation rules immediately resume the joining threads in the program state. Note that in the SREAD rule, the value is read from the assumed store recorded in the speculative pattern environment, which represents the value assumed to be present in the store at location $\mathtt{l}$ after all base patterns for the synchronization pattern have been satisfied. As expected, each of the rules appends an appropriate trace action ($\frac{\mathtt{l}\rightarrow\gamma,i}{\mathtt{t}}$ or $\frac{\mathtt{l}\leftarrow\gamma,i\mathtt{v}}{\mathtt{t}}$) to the speculative trace.

As noted above, the SDONE rule only validates the merging of the (*speculative*) thread-local stores of threads joining with a synchronization pattern. The SCOMMIT rule validates the global consistency of individual thread actions and completed synchronization-pattern matches. Thus, the validation of a synchronization-pattern match is established piece-meal, without requiring an atomic validation of all participating threads and their actions. Intuitively, the SCOMMIT rule moves trace actions from the speculative trace to the

[2] See `http://ttic.uchicago.edu/~fluet/research/speculation/proofs.pdf` for the complete definition.

PROCESSES:

$$\pi \quad ::= \quad \pi \| \pi \;\mid\; \mathtt{t}[\mathtt{e}, \sigma]$$

PATTERN ENVIRONMENTS:

$$\psi \quad ::= \quad \{\gamma \mapsto \langle \overline{\mathtt{p}}, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_r]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]}, \overline{\sigma'}, \sigma'' \rangle, \ldots\}$$

SPECULATIVE EVALUATION:

APP

$$\frac{}{\mathtt{t}[E[(\lambda\,\mathtt{x}.\mathtt{e})(\mathtt{v})], \sigma] \| \pi, \psi, \Omega_C, \Omega_S \rightsquigarrow \mathtt{t}[E[\mathtt{e}[\mathtt{x}/\mathtt{v}]], \sigma] \| \pi, \psi, \Omega_C, \Omega_S . \frac{\beta}{\mathtt{t}}}$$

REF

$$\frac{\mathtt{l}\ \textit{fresh}}{\mathtt{t}[E[\mathtt{ref}(\mathtt{v})], \sigma] \| \pi, \psi, \Omega_C, \Omega_S \rightsquigarrow \mathtt{t}[E[\mathtt{l}], \sigma \uplus \{\mathtt{l} \mapsto \mathtt{v}\}] \| \pi, \psi, \Omega_C, \Omega_S . \frac{\mathtt{v} \Uparrow \mathtt{l}}{\mathtt{t}}}$$

READ

$$\frac{\sigma(\mathtt{l}) = \mathtt{v}}{\mathtt{t}[E[!\mathtt{l}], \sigma] \| \pi, \psi, \Omega_C, \Omega_S \rightsquigarrow \mathtt{t}[E[\mathtt{v}], \sigma] \| \pi, \psi, \Omega_C, \Omega_S . \frac{\mathtt{l} \rightarrow \mathtt{v}}{\mathtt{t}}}$$

WRITE

$$\frac{}{\mathtt{t}[E[\mathtt{l} := \mathtt{v}], \sigma] \| \pi, \psi, \Omega_C, \Omega_S \rightsquigarrow \mathtt{t}[E[\mathtt{unit}], \sigma[\mathtt{l} \mapsto \mathtt{v}]] \| \pi, \psi, \Omega_C, \Omega_S . \frac{\mathtt{l} \leftarrow \mathtt{v}}{\mathtt{t}}}$$

SMATCH

$$\frac{\mathtt{t}_b, \gamma\ \textit{fresh}}{\mathtt{t}[E[\mathtt{match}(\overline{\mathtt{p}}) \rightarrow \mathtt{e}_b], \sigma] \| \pi, \psi, \Omega_C, \Omega_S \rightsquigarrow \mathtt{t}[E[\mathtt{unit}], \sigma] \| \pi \| \mathtt{t}_b[\mathtt{e}_b, \sigma''], \psi \uplus \{\gamma \mapsto \langle \overline{\mathtt{p}}, \mathtt{t}_b[\mathtt{e}_b], \phi, \phi, \phi.\sigma, \sigma'' \rangle\}, \Omega_C, \Omega_S . \frac{\overline{\mathtt{p}} \rightarrow_\gamma \mathtt{t}_b}{\mathtt{t}}}$$

SDONE

$$\frac{\mathsf{Merge}(\overline{\sigma'}, \overline{\mathtt{l}_w \mapsto \mathtt{v}_w}) = \sigma''}{\pi, \psi \uplus \{\gamma \mapsto \langle \phi, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_r]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]}, \overline{\sigma'}, \sigma'' \rangle\}, \Omega_C, \Omega_S \rightsquigarrow \pi, \psi, \Omega_C, \Omega_S . \frac{\gamma}{}}$$

SREAD

$$\frac{1 \leq |\overline{\mathtt{p}}| = n \qquad i \in \{1 \ldots n\} \qquad \mathtt{p}_i = \mathtt{rd}(\mathtt{l})}{\mathtt{t}[E[!\mathtt{l}], \sigma] \| \pi, \psi \uplus \{\gamma \mapsto \langle \overline{\mathtt{p}}, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_r]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]}, \overline{\sigma'}, \sigma'' \rangle\}, \Omega_C, \Omega_S \rightsquigarrow \mathtt{t}[E[\sigma''(\mathtt{l})], \sigma''] \| \pi, \psi \uplus \{\gamma \mapsto \langle \overline{\mathtt{p}} \setminus i, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_r]}.\mathtt{t}[\mathtt{l}, E], \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]}, \overline{\sigma'}, \sigma'' \rangle\}, \Omega_C, \Omega_S . \frac{\mathtt{l} \rightarrow_{\gamma,i}}{\mathtt{t}}}$$

SWRITE

$$\frac{1 \leq |\overline{\mathtt{p}}| = n \qquad i \in \{1 \ldots n\} \qquad \mathtt{p}_i = \mathtt{wr}(\mathtt{l})}{\mathtt{t}[E[\mathtt{l} := \mathtt{v}], \sigma] \| \pi, \psi \uplus \{\gamma \mapsto \langle \overline{\mathtt{p}}, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_r]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]}, \overline{\sigma'}, \sigma'' \rangle\}, \Omega_C, \Omega_S \rightsquigarrow \mathtt{t}[E[\mathtt{unit}], \sigma''] \| \pi, \psi \uplus \{\gamma \mapsto \langle \overline{\mathtt{p}} \setminus i, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_r]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]}.\mathtt{t}[\mathtt{l}, \mathtt{v}, E], \overline{\sigma'}, \sigma'' \rangle\}, \Omega_C, \Omega_S . \frac{\mathtt{l} \rightarrow_{\gamma,i}}{\mathtt{t}}}$$

SCOMMIT

$$\frac{\Omega_X \leftrightharpoons \omega \qquad \kappa(\Omega_C.\omega) = (\_, \_)}{\pi, \psi, \Omega_C, \Omega_X.\omega.\Omega_Y \rightsquigarrow \pi, \psi, \Omega_C.\omega, \Omega_X.\Omega_Y}$$

Figure 7. Speculative evaluation rules.

committed trace. The requirement that $\Omega_X \leftrightharpoons \omega$ ensures that the action ω is not dependent upon any action in Ω_X; this implies that the speculative-evaluation step that introduced ω could have taken place before all of the speculative-evaluation steps that introduced Ω_X. The requirement that $\kappa(\Omega_C.\omega) = (\_, \_)$ ensures that the trace $\Omega_C.\omega$ can be simulated to yield some simulated program state and simulated pattern environment. Since trace simulation is similar to the synchronous-evaluation relation, the definedness of $\kappa(\Omega_C.\omega)$ implies that the speculative-evaluation step that introduced ω corresponds to a synchronous-evaluation step that can take place after a sequence of synchronous-evaluation steps that correspond to Ω_C.

Soundness. As alluded to above, the various validation checks made by the SDONE and SCOMMIT rules are meant to ensure that the speculative-evaluation relation is sound with respect to the synchronous-evaluation relation. We demonstrate this correspondence in Theorem 1.[3] The soundness theorem is formulated with respect to a fixed (but arbitrary) initial main thread $\mathtt{m}[e_0, \emptyset]$.

[3] See `http://ttic.uchicago.edu/~fluet/research/speculation/proofs.pdf` for details.

To formally state the soundness theorem, we extend the synchronous-evaluation relation with committed traces $(\pi, \rho, \Omega_C \Rightarrow \pi', \rho', \Omega'_C)$ in the obvious manner. We also introduce an *erasure* function to convert a speculative-evaluation program state and pattern environment to a synchronous-evaluation program state and pattern environment ($[\![\pi, \psi]\!] = (\pi^\star, \rho^\star)$; Fig. 8). This (partial) function removes from the program state any processes that correspond to speculative evaluation.

The soundness theorem asserts that a sequence of speculative-evaluation steps can be reconstructed as a sequence of speculative-evaluation steps that perform only the committed trace actions (and none of the speculative trace actions) followed by a sequence of speculative-evaluation steps that perform only the speculative trace actions (and no additional committed trace actions). Furthermore, the speculative-evaluation program state and pattern environment corresponding to the committed trace actions only can be erased to a synchronous-evaluation program state and pattern environment that is a synchronous evaluation of the initial program state.

THEOREM 1. **(Soundness)**

SPECULATIVE PROCESS AND PATTERN ENVIRONMENT ERASURE:

$$
\begin{array}{rcl}
\llbracket \pi, \psi \rrbracket & = & \llbracket \pi, \psi; \emptyset \rrbracket \\
\\
\llbracket \pi, \emptyset; \rho \rrbracket & = & (\pi, \rho) \\
\llbracket \mathtt{t}_b[\mathtt{e}_b, \sigma''] \| \overline{\mathtt{t}_r[E_r[\sigma''(\mathtt{l}_r)], \sigma'']} \| \overline{\mathtt{t}_w[E_w[\mathtt{unit}], \sigma'']} \| \pi, & = & \mathsf{let}\ (\pi^\star, \rho^\star) = \llbracket \pi, \psi; \rho \rrbracket \\
\quad \psi \uplus \{\gamma \mapsto \langle \overline{\mathtt{p}}, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_r]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w, \sigma_w]}, \overline{\sigma'}, \sigma'' \rangle\}; \rho \rrbracket & & \quad \mathsf{in}\ (\pi^\star, \rho^\star \uplus \{\gamma \mapsto \langle \overline{\mathtt{p}}, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_l]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]}, \overline{\sigma'} \rangle\})
\end{array}
$$

Figure 8. Speculative process and pattern environment erasure.

If $\pi, \psi, \Omega_C, \Omega_S \rightsquigarrow^* \pi'', \psi'', \Omega''_C, \Omega''_S$
and $\exists \pi^\dagger, \psi^\dagger, \pi^\star, \rho^\star$
such that $\mathtt{m}[e_0, \emptyset], \emptyset, \phi, \phi \rightsquigarrow^* \pi^\dagger, \psi^\dagger, \Omega_C, \phi$
and $\pi^\dagger, \psi^\dagger, \Omega_C, \phi \rightsquigarrow^* \pi, \psi, \Omega_C, \Omega_S$
and $\llbracket \pi^\dagger, \psi^\dagger \rrbracket = (\pi^\star, \rho^\star)$
and $\mathtt{m}[e_0, \emptyset], \emptyset, \phi \Rightarrow^* \pi^\star, \psi^\star, \Omega_C,$
then $\exists \pi^\ddagger, \psi^\ddagger, \pi^*, \rho^*$
such that $\mathtt{m}[e_0, \emptyset], \emptyset, \phi, \phi \rightsquigarrow^* \pi^\ddagger, \psi^\ddagger, \Omega''_C, \phi$
and $\pi^\ddagger, \psi^\ddagger, \Omega''_C, \phi \rightsquigarrow^* \pi'', \psi'', \Omega''_C, \Omega''_S$
and $\llbracket \pi^\ddagger, \psi^\ddagger \rrbracket = (\pi^*, \rho^*)$
and $\mathtt{m}[e_0, \emptyset], \emptyset, \phi \Rightarrow^* \pi^*, \psi^*, \Omega''_C.$

The proof is a straightforward induction and case analysis on the derivation $\pi, \psi, \Omega_C, \Omega_S \rightsquigarrow^* \pi'', \psi'', \Omega''_C, \Omega''_S$; the most interesting case is that of the SCOMMIT rule, which requires a number of supporting lemmas that formalize the behavior of traces.

A simple corollary of Theorem 1 states that the committed trace of a speculative evaluation of the initial program state corresponds to a synchronous evaluation of the initial program state:

THEOREM 2. **(Soundness)**
If $\mathtt{m}[\mathtt{e}_0, \emptyset], \emptyset, \phi, \phi \rightsquigarrow^* \pi'', \psi'', \Omega''_C, \Omega''_S,$
then $\exists \pi^\ddagger, \psi^\ddagger, \pi^*, \rho^*$
such that $\mathtt{m}[\mathtt{e}_0, \emptyset], \emptyset, \phi, \phi \rightsquigarrow^* \pi^\ddagger, \psi^\ddagger, \Omega''_C, \phi$
and $\pi^\ddagger, \psi^\ddagger, \Omega''_C, \phi \rightsquigarrow^* \pi'', \psi'', \Omega''_C, \Omega''_S$
and $\llbracket \pi^\ddagger, \psi^\ddagger \rrbracket = (\pi^*, \rho^*)$
and $\mathtt{m}[\mathtt{e}_0, \emptyset], \emptyset, \phi \Rightarrow^* \pi^*, \rho^*, \Omega''_C.$

The converse of Theorem 2 is also true, since every synchronous-evaluation step can be simulated by a corresponding speculative-evaluation step followed by an SCOMMIT step:

THEOREM 3. **(Converse Soundness)**
If $\mathtt{m}[\mathtt{e}_0, \emptyset], \emptyset, \phi \Rightarrow^* \pi^*, \rho^*, \Omega''_C,$
then $\exists \pi^\ddagger, \psi^\ddagger$
such that $\mathtt{m}[\mathtt{e}_0, \emptyset], \emptyset, \phi, \phi \rightsquigarrow^* \pi^\ddagger, \psi^\ddagger, \Omega''_C, \phi$
and $\llbracket \pi^\ddagger, \psi^\ddagger \rrbracket = (\pi^*, \rho^*).$

4.3 Discussion

As specified above, the speculative-evaluation semantics is sound with respect to the synchronous-evaluation semantics. However, the choice of an assumed store in the rule for speculative synchronization-pattern match (SMATCH) effectively implements an oracle. The oracle utilizes arbitrary power in predicting the program's future actions. Implementations may wish to bound the amount of predictive power or use none at all. For example, one simple choice for the assumed store at a synchronization-pattern match would be the current thread-local store of the thread executing the match. Speculation admits the possibility that the assumed store may be incorrect, as it depends upon future actions. Nonetheless, we can augment the speculative-evaluation semantics with additional rules to detect failure and to recover.

In Fig. 9, we provide an extension to the speculation-evaluation semantics that allows the completion of a synchronization-pattern match to fail. When a such a failure is detected (SFAIL), the semantics yields a failed state ($\diamond$). Failure is detected when the assumed store does not equal the store derived by merging the stores of all the participating threads. Evaluation is halted at a fail state.

This form of failure corresponds to a true discrepancy between the speculative evaluation and the synchronous evaluation implied by the committed trace. That is, while the speculative-evaluation relation may continue to make some progress with other threads (and even commit additional trace actions), the failing synchronization pattern will continue to fail in the future.

THEOREM 4. **(Failure Safety ($\rightsquigarrow$ from Fig. 7 and Fig. 9))**
If $\mathtt{m}[e_0, \emptyset], \emptyset, \phi, \phi \rightsquigarrow^* \pi', \psi', \Omega'_C, \Omega'_S \rightsquigarrow \diamond$
and $\pi', \psi', \Omega'_C, \Omega'_S \rightsquigarrow^* \pi'', \psi'', \Omega''_C, \Omega''_S,$
then $\pi'', \psi'', \Omega''_C, \Omega''_S \rightsquigarrow \diamond.$

We may utilize the fact that the speculative-evaluation semantics is sound with respect to the synchronous-evaluation semantics to provide a simple but effective recovery mechanism. In particular, we may use the committed trace to synthesize a known good state (namely, the state corresponding to the synchronous evaluation that yields the committed trace) and resume the computation with an empty speculative trace. In Fig. 10, we provide an extension to the speculative-evaluation semantics that implements this intuition. Since the SDONE rule does not resume blocked threads from the speculative pattern environment (rather, it simply allows the speculatively executing threads to continue executing), we make use of a *synthesis* function to convert a synchronous-evaluation program state and pattern environment to a speculative-evaluation program state and pattern environment ($\llbracket\!\llbracket \pi, \psi \rrbracket\!\rrbracket = (\pi^\dagger, \rho^\dagger)$). This (total) function adds to the program state any processes blocked in the pattern environment; like the SMATCH rule, it introduces a new store (σ'') that is the assumed store that will exist after the synchronization pattern is satisfied. While the choice of the assumed store remains oracular, the choice can be made with the knowledge of the local stores of some of the necessarily participating threads.

The SRECOVER rule provides a simple, somewhat heavyweight, recovery mechanism. However, it does demonstrate that the speculative-evaluation semantics has sufficient information to facilitate recovering from an incorrectly-chosen assumed store, and to allow program computation to make progress. Indeed, a realistic implementation would likely alternate between (bounded) speculation and synchronous evaluation, to ensure progress with respect to synchronous-evaluation alone. We believe that it can serve as a starting point for more sophisticated recovery strategies.

For example, one attractive special case is the situation where a synchronization pattern is satisfied and committable, but has an incorrectly-chosen assumed store:

$$
\frac{\begin{array}{c}\mathsf{Merge}(\overline{\sigma'}, \overline{\mathtt{l}_w \mapsto \mathtt{v}_w}) = \sigma^\dagger \neq \sigma'' \\ \Omega_S \leftrightharpoons^{\gamma} \qquad \kappa(\Omega_C.^{\gamma}) = (\_, \_) \\ (\pi, \psi, \Omega_S) \ominus \mathtt{t}_b.\overline{\mathtt{t}_r}.\overline{\mathtt{t}_w} = (\pi^\dagger, \psi^\dagger, \Omega^\dagger_S)\end{array}}{\begin{array}{c}\pi, \psi \uplus \{\gamma \mapsto \langle \phi, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_r]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]}, \overline{\sigma'}, \sigma'' \rangle\}, \Omega_C, \Omega_S \rightsquigarrow \\ \mathtt{t}_b[\mathtt{e}_b, \sigma^\dagger] \| \overline{\mathtt{t}_r[E_r[\sigma^\dagger(\mathtt{l}_r)], \sigma^\dagger]} \| \overline{\mathtt{t}_w[E_w[\mathtt{unit}], \sigma^\dagger]} \| \pi^\dagger, \psi^\dagger, \Omega_C.^{\gamma}, \Omega^\dagger_S\end{array}}
$$

SPECULATIVE EVALUATION WITH FAILURE:

$$\textsc{SFail}\quad \frac{\mathsf{Merge}(\overline{\sigma'}, \overline{\mathtt{l}_w \mapsto \mathtt{v}_w}) \neq \sigma''}{\pi, \psi \uplus \{\gamma \mapsto \langle \phi, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_r]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]}, \overline{\sigma'}, \sigma''\rangle\}, \Omega_C, \Omega_S \rightsquigarrow \diamond}$$

Figure 9. Speculative evaluation with failure.

SPECULATIVE PROCESS AND PATTERN ENVIRONMENT SYNTHESIS:

$$\begin{array}{rcl} [\![\pi, \rho]\!] & = & [\![\pi, \rho; \emptyset]\!] \\ [\![\pi, \emptyset; \psi]\!] & = & (\pi, \psi) \\ [\![\pi, \rho \uplus \{\gamma \mapsto \langle \overline{\mathtt{p}}, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_r]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]}, \overline{\sigma'}\rangle\}; \psi]\!] & = & \mathsf{let}\ (\pi^\dagger, \rho^\dagger) = [\![\pi, \rho; \psi]\!] \\ & & \mathsf{in}\ (\mathtt{t}_b[\mathtt{e}_b, \sigma''] \| \overline{\mathtt{t}_r[E_r[\sigma''(\mathtt{l}_r)], \sigma'']} \| \overline{\mathtt{t}_w[E_w[\mathtt{unit}], \sigma'']} \| \pi^\dagger, \\ & & \quad \rho^\dagger \uplus \{\gamma \mapsto \langle \overline{\mathtt{p}}, \mathtt{t}_b[\mathtt{e}_b], \overline{\mathtt{t}_r[\mathtt{l}_r, E_l]}, \overline{\mathtt{t}_w[\mathtt{l}_w, \mathtt{v}_w, E_w]}, \overline{\sigma'}, \sigma''\rangle\}) \end{array}$$

SPECULATIVE EVALUATION WITH RECOVERY:

$$\textsc{SRecover}\quad \frac{\mathtt{m}[e_0, \emptyset], \emptyset, \phi \Rightarrow^* \pi^*, \rho^*, \Omega_C \qquad [\![\pi^*, \rho^*]\!] = (\pi^\ddagger, \psi^\ddagger)}{\pi, \psi, \Omega_C, \Omega_S \rightsquigarrow \pi^\ddagger, \psi^\ddagger, \Omega_C, \phi}$$

Figure 10. Speculative evaluation with recovery.

In this situation, it suffices to remove any components from the program state, speculative pattern environment, and speculative trace that depend upon the (incorrectly) speculated actions of the participating threads, but retaining the computation and actions of independent threads. We envision using self-adjusting computation (2) as one way to efficiently propagate the difference between the assumed store (σ'') and the actual merged store ($\sigma^\dagger$), possibly reusing portions of the speculative computation that don't depend on incorrect assumptions.

5. Related Work and Conclusions

Our work is influenced by previous research on both implementation techniques for speculative execution, and the formalization of high-level concurrency abstractions. Speculation has long been used by processor architects and compiler writers (6; 14; 15; 11; 10; 13) to extract parallelism from sequential programs. These techniques speculatively execute concurrent threads and revoke (or stall) execution in the presence of conflicts. Kulkarni *et al.* (9) present their experience in parallelizing large-scale irregular applications using speculative parallelization.

Transactional memory designs based on optimistic concurrency (3; 7) allow transactions to execute in isolation, logging updates and validating these updates are consistent (e.g., serializable) with other concurrently executing transactions. Transactions execute optimistically under the assumption that changes they perform are unlikely to be invalidated by other transactions. When this assumption fails, the transaction must be aborted and restarted.

Cilk (4) is a multithreaded extension of C with support fork-join parallelism. Speculative threads can be aborted, but no safety guarantees are provided. The basic synchronization mechanism (`sync`) acts as a barrier that waits for completion of all children spawned by the thread. Cilk provides no mechanism similar to our `match` abstraction, nor does it allow safe speculative execution beyond `sync` points.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of Transactional Memory and Automatic Mutual Exclusion. In *POPL*, pages 63–74, 2008.

[2] U. A. Acar, A. Ahmed, and M. Blume. Imperative Self-Adjusting Computation. In *POPL*, pages 309–322, 2008.

[3] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *PLDI*, pages 26–37, 2006.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Journal of Parallel and Distributed Computing*, pages 207–216, 1995.

[5] K. Donnelly and M. Fluet. Transactional events. In *ICFP*, pages 124–135, 2006.

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

[7] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402, 2003.

[8] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *PLDI*, pages 59–70, 2004.

[9] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic Parallelism Requires Abstractions. In *PLDI*, pages 211–222, 2007.

[10] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: A TLS Compiler that Exploits Program Structure. In *PPoPP*, pages 158–167, 2006.

[11] J. F. Martínez and J. Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *ASPLOS*, pages 18–29, 2002.

[12] K. F. Moore and D. Grossman. High-level Small-Step Operational Semantics for Transactions. In *POPL*, pages 51–62, 2008.

[13] A. Navabi, X. Zhang, and S. Jagannathan. Quasi-static Scheduling for Safe Futures. In *PPoPP*, pages 23–32, 2008.

[14] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA*, pages 414–425, 1995.

[15] J. Steffan and T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization, booktitle = HPCA. page 2, 1998.

[16] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *ISCA*, pages 1–12, 2000.