Combining Shared State with Speculative Parallelism in a Functional Language

Matthew Le
Rochester Institute of Technology
ml9951@cs.rit.edu

Matthew Fluet
Rochester Institute of Technology
mtf@cs.rit.edu

Abstract

Purely functional programming languages have proven to be an attractive option for implementing parallel applications. The lack of mutable state eliminates the possibility for race conditions, which relieves programmers of reasoning about the exponential interleavings of threads and nondeterministic behavior. Unfortunately, there are applications that by making use of shared state can achieve significant constant factor speedups compared to their purely functional counterparts.

IVars have been proposed as a possible solution, allowing threads to share information via write-once references, while preserving a deterministic semantics. However, in the presence of speculative parallelism (cancelation), this determinism guarantee is lost. In this work we show how to go about combining these two concepts by proposing a dynamic rollback mechanism for enforcing determinism. We have formalized the semantics of a parallel functional language extended with IVars, speculative parallelism, and our proposed rollback mechanism using the Coq proof assistant, and have proven that it preserves determinism. Additionally, we describe a preliminary implementation in the context of the Manticore project, and give some initial performance results.

1. Introduction

Writing parallel applications is a notoriously difficult task. Programmers are forced to reason about the nondeterministic behavior arising from an exponential interleaving of threads. One way of avoiding this difficulty is to use a functional language when developing parallel applications, since functional languages prohibit the alteration of shared state. Race conditions and nondeterminism arise when multiple threads attempt to read from and write to the same location in memory. Since functional languages do not allow writes, we avoid race conditions altogether, making determinism an easy property to enforce. Unfortunately, there are applications that can be more efficiently or more naturally implemented when shared state is used. One attempt at addressing this problem is the use of IVars [ANP89], which are shared references that may only be written to once. IVars have been proven to preserve determinism in an otherwise purely functional parallel language [MNP11, BBC$^+$10], while allowing threads to communicate intermediate results to one another via shared memory.

In this work we show that the determinism guarantee for IVars does not hold in the presence of speculation – a method for parallelizing programs, where unneeded tasks may be canceled. Additionally this paper makes the following contributions:

- We propose a rollback mechanism that can be used to restore deterministic execution in the presence of speculation and IVars.
- We provide a formal semantics of a parallel language with IVars, speculative parallelism, and the proposed rollback mechanism.
- We give a mechanized proof, using the Coq Proof Assistant, that the rollback mechanism preserves determinism of the language that combines IVars and speculative parallelism.
- We describe an implementation that is under development in the Manticore project and give some preliminary results.

Source code for the Coq formalization can be found at: `http://people.rit.edu/ml9951/research.html`.

2. Background

2.1 IVars

IVars are shared memory references that may only be written to once, originally proposed as part of the parallel functional language Id [ANP89]. The interesting property about IVars is that they do not compromise the determinism guarantees that one can make about an otherwise purely functional parallel language. Meanwhile, they strictly increase the expressiveness of the language. As an example, consider an application implementing producer-consumer parallelism where two threads are running in parallel, one of which writes data into a shared buffer and the other processes this data as soon as it becomes available. This sort of pattern cannot be efficiently implemented in a purely functional language. In such a language, the producer would be required to produce all of its elements before the consumer could start processing them. On the other hand, if we are able to make use of IVars, we could implement the shared buffer as a linked list giving us the desired behavior.

Informally, the semantics of IVars are as follows. When an IVar is created it is empty. When it is written to, it becomes full and if a thread attempts to write to it again, a runtime error is produced and the program terminates. If a thread tries to read from an IVar that is empty, it blocks until the contents are filled, after which it can read from the IVar an arbitrary number of times without synchronization.

2.2 Speculative Parallelism

Speculation is a method for parallelizing applications, where some number of parallel tasks are created, and if it turns out that any of these tasks are unneeded, they are canceled. This sort of pattern arises frequently in search problems where we want to search multiple paths in parallel, and then when a solution is found, we would like to cancel the rest of the search threads so as to free up resources for future computations. The research literature has given rise to many examples of speculatively parallel algorithms [PRV10, Bur85, JLM$^+$09].

```
1   exception E
2   val i = IVar.new()
3   val _ = (|raise E, IVar.put(i, 10)|)
4             handle E => ((), ())
5   val x = IVar.get i
```

Figure 1. Nondeterministic Example

2.3 Manticore

Manticore is a compiler for a purely functional subset of Standard ML which has been extended with parallel features. These parallel features are given a sequential semantics, allowing programmers to reason about parallel computations in the same way they would their sequential counterparts. The most basic parallel construct in Manticore is the parallel tuple, denoted

$$(|e_1,\ \ldots,\ e_n|)$$

Parallel tuples express fork-join parallelism, where each expression e_i is evaluated in parallel. The result of the entire expression is a data structure containing the results of each e_i.

Additionally, we provide a construct for asynchronously spawning threads via **fork**. The **fork** construct takes an expression that gets evaluated in a separate thread allowing the main thread to continue with the execution of the remainder of the program.

In addition to parallelism, Manticore also supports exception handling. The semantics for a regular sequential tuple is to evaluate each e_i in left to right order, so if an exception gets raised, it will always be the leftmost exception in the tuple that gets propagated. Enforcing the sequential semantics for parallel tuples in the presence of exceptions works as follows. If expression e_i raises an exception, then the threads evaluating expressions e_{i+1} through e_n are canceled and we wait for the previous $i-1$ elements to terminate to check if they raise an exception.

2.4 Determinism

In Manticore we can encode a notion of speculative parallelism using the parallel constructs and exception handling features while maintaining the sequential semantics [FRRS08]. Unfortunately, if we were to incorporate IVars into the language, we would lose this guarantee due to the cancelation associated with raised exceptions. As an example, consider the code in Figure 1. In line 2 we create an empty IVar, and then in parallel raise an exception and write to this IVar. There are two ways in which this can play out. The cancelation can go through before the write, leaving the IVar empty, or the write can go through before the cancelation leaving the IVar full with the value 10. These two scenarios lead to two different observable behaviors of our program, either it could block indefinitely due to a read from an empty IVar, or it could terminate with x bound to the value 10. In order to enjoy the benefits of a deterministic parallel language, we must extend the Manticore runtime system to avoid these race conditions.

3. Preserving Determinism

In order to preserve a deterministic semantics for parallel tuples in the presence of exception handling we would like to make it seem to the programmer as if canceled threads "never happened" in the first place. Implementing this for a purely functional language is not too difficult, however, in the presence of shared references such as IVars, it becomes substantially more complex.

The first step is to "undo" the effects of a thread when it is being canceled due to a raised exception. With IVars this simply amounts to resetting the contents of full IVars to empty. However, it is possible that before the cancelation occurred, other threads concurrently running were able to read the contents of this IVar.

```
1   exception E
2   val i = IVar.new()
3   val j = IVar.new()
4   val _ = fork((|raise E, IVar.put(i, 10)|)
5                 handle E => ((), ()))
6   val (_, x) = (|IVar.put(j, IVar.get i), IVar.get j|)
```

Figure 2. Transitive Rollback

	x	$\in$	Var
Values	V	$::=$	$x \mid i \mid \backslash x.M \mid$ **return** $M \mid M >>= N$
		$\mid$	**runPar** $M \mid$ **fork** $M \mid$ **new** $\mid$ **put** $i\ M$
		$\mid$	**get** $i \mid$ **done** $M \mid$ **spec** $M\ N$
		$\mid$	**specRun**$(M, N) \mid$ **specJoin**(M, N)
		$\mid$	**raise** $M \mid$ **handle** $M\ N$
Terms	M, N	$::=$	$V \mid M\ N \mid \cdots$
Heap	H	$::=$	$H, x \mapsto iv \mid \cdot$
Speculative State	s	$::=$	**S** $\mid$ **C**
IVar State	iv	$::=$	$\langle s \rangle \mid \langle s_1, ds, s_2, \Theta, M \rangle$
Thread ID	Θ	$::=$	$\cdot \mid \Theta : n \qquad n \in \mathbb{N}$
Thread Pool	T	$::=$	$\cdot \mid (T_1 \mid T_2) \mid \Theta[S_1, S_2, M]$
Action	A	$::=$	$(R, x, M) \mid (W, x, M) \mid (S, M)$
		$\mid$	$(A, x, M) \mid (F, \Theta, M) \mid CSpec$
Action Queue	S	$::=$	$\cdot \mid S : A$
Evaluation Context	E	$::=$	$[\cdot] \mid E >>= M \mid$ **specRun**(E, M)
		$\mid$	**handle** $E\ N \mid$ **specJoin**(N, E)
Configuration	σ	$::=$	$H; T \mid Error$

Figure 3. Speculative Par Monad Syntax

In this case, the runtime system must also rollback these threads to the point in which they read from the IVar. At this point, we also need to "undo" any effects that these threads may have done and rollback any threads that might have read from these IVars. This rollback continues until we reset all IVars and dependent readers that are transitively reachable from the effects done by the original thread that was canceled.

As an example of this transitive closure property, consider the code in Figure 2. In line 4 we fork a new thread to evaluate a parallel tuple that raises an exception and writes the value 10 to IVar i. The thread writing to the IVar is then canceled due to the raised exception, requiring the contents of i to be reset to empty. In line 6 we read the value written to i and write it into j. If this read occurs before the cancelation, then we must reset this thread back to before it performed the read. If we are going to reset this thread to before it did the read, then we must also "undo" the write to IVar j. Furthermore, if the second element of the parallel tuple is able to read from j, then this must also get rolled back to the point before it performed the read.

4. Formal Semantics

In this section we provide a formal semantics describing a method for performing the rollback that was alluded to in the previous section. The semantics presented in this paper are an extension of the Par Monad semantics as presented in [MNP11], which helps facilitate our proof of determinism described in the next section. Figure 3 gives the syntax of the language. Relative to [MNP11], We have added syntax for speculative computations, where **specRun** and **specJoin** are intermediate forms that arise throughout the execution of a program, and are not terms available in the surface language, as is **done**.

A heap is a finite map from IVar names to IVar states where an IVar state can be empty or full. If it is empty, we also indicate

$$\text{RunPar}\ \frac{\cdot; 1[\cdot, \cdot, M >>= \backslash x.\textbf{done}\ x] \rightarrow_s^* H'; T \mid \cdot : 1[\cdot, S_2, \textbf{done}\ N], \quad N \Downarrow_s V, \quad \text{Finished}(T)}{\textbf{runPar}\ M \Downarrow_s V}$$

$$\text{RunParError}\ \frac{\cdot; 1[\cdot, \cdot, M >>= \backslash x.\textbf{done}\ x] \rightarrow_s^* \textbf{Error}}{\textbf{runPar}\ M \Downarrow_s \textbf{Error}} \qquad \text{RunParDiverge}\ \frac{\cdot; 1[\cdot, \cdot, M >>= \backslash x.\textbf{done}\ x] \rightarrow_s^{\infty}}{\textbf{runPar}\ M \Downarrow_s \infty}$$

$\boxed{H; T \rightarrow_s \sigma}$

$$\text{Eval}\ \frac{M \neq V \quad M \Downarrow_s V}{H; \Theta[S_1, S_2, E[M]] \mid T \rightarrow_s H; \Theta[S_1, S_2, E[V]] \mid T} \qquad \text{Bind}\ \frac{}{H; \Theta[S_1, S_2, E[\textbf{return}\ N >>= M]] \mid T \rightarrow_s H; \Theta[S_1, S_2, E[M\ N]] \mid T}$$

$$\text{BindRaise}\ \frac{}{H; \Theta[S_1, S_2, E[\textbf{raise}\ M >>= N]] \mid T \rightarrow_s H; \Theta[S_1, S_2, E[\textbf{raise}\ M]] \mid T}$$

$$\text{Handle}\ \frac{}{H; \Theta[S_1, S_2, E[\textbf{handle}(\textbf{raise}\ M)N]] \mid T \rightarrow_s H; \Theta[S_1, S_2, E[N\ M]] \mid T}$$

$$\text{HandleReturn}\ \frac{}{H; \Theta[S_1, S_2, E[\textbf{handle}(\textbf{return}\ M)N]] \mid T \rightarrow_s H; \Theta[S_1, S_2, E[\textbf{return}\ M]] \mid T}$$

$$\text{Fork}\ \frac{n = \text{numSpawns}(s1, s2)}{H; \Theta[S_1, S_2, E[\textbf{fork}\ M]] \mid T \rightarrow_s H; \Theta[(F, \Theta : n, E[\textbf{fork}\ M]) : S_1, S_2, E[\textbf{return}()]] \mid \Theta : n[CSpec, \cdot, M] \mid T}$$

$$\text{New}\ \frac{H' = H[x \mapsto \langle \textbf{S} \rangle] \quad x \notin Domain(H)}{H; \Theta[S_1, S_2, E[\textbf{new}]] \mid T \rightarrow_s H'; \Theta[(A, x, E[\textbf{new}]) : S_1, S_2, E[\textbf{return}\ x]] \mid T}$$

$$\text{Get}\ \frac{H(x) = \langle s_1, ds, s_2, \Theta', M \rangle, \quad H' = H[x \mapsto \langle s_1, \Theta \uplus ds, s_2, \Theta', M \rangle]}{H; \Theta[S_1, S_2, E[\textbf{get}\ x]] \mid T \rightarrow_s H'; \Theta[(R, x, E[\textbf{get}\ x]) : S_1, S_2, E[\textbf{return}\ M]] \mid T}$$

$$\text{Put}\ \frac{H(x) = \langle s \rangle, \quad H' = H[x \mapsto \langle s, \emptyset, \textbf{S}, \Theta, M \rangle]}{H; \Theta[S_1, S_2, E[\textbf{put}\ x\ M]] \mid T \rightarrow_s H'; \Theta[(W, x, E[\textbf{put}\ x\ M]) : S_1, S_2, E[\textbf{return}()]] \mid T}$$

$$\text{Overwrite}\ \frac{\begin{array}{c} H(x) = \langle s_1, ds, \textbf{S}, \Theta', N \rangle, \quad \Theta'[S_1 : (W, x, N) : S_1', S_2', N'] \in T \\ \text{rollback}(\Theta', S_1', H, T) \rightsquigarrow (H', T'), \quad H'' = H'[x \mapsto \langle \emptyset, \cdot, \Theta, M \rangle] \end{array}}{H; \Theta[\cdot, S_2, E[\textbf{put}\ x\ M]] \mid T \rightarrow_s H''; \Theta[\cdot, S_2, E[\textbf{return}()]] \mid T'} \qquad \text{ErrorWrite}\ \frac{H(x) = \langle \textbf{C}, ds, \textbf{C}, \Theta', N \rangle}{H; \Theta[\cdot, S_2, E[\textbf{put}\ x\ M]] \mid T \rightarrow_s Error}$$

$$\text{Spec}\ \frac{n = \text{numSpawns}(s1, s2)}{H; \Theta[S_1, S_2, E[\textbf{spec}\ M\ N]] \mid T \rightarrow_s H; \Theta[(F, \Theta : n, E[\textbf{spec}\ M\ N]) : S_1, S_2, E[\textbf{specRun}(M, N)]] \mid \Theta : n[(S, N) : CSpec, \cdot, N] \mid T'}$$

$$\text{SpecRB}\ \frac{\text{rollback}(\Theta : n, \cdot, H, \Theta : n[S_1' : (S, N_0), S_2', N] \mid T) \rightsquigarrow (H', \Theta : n[\cdot, S_2', N'] \mid T')}{H; \Theta[\cdot, S_2, E[\textbf{specRun}(\textbf{raise}\ M, N_0]] \mid \Theta : n[S_1' : (S, N_0), S_2', N] \mid T \rightarrow_s H'; \Theta[\cdot, S_2, E[\textbf{raise}\ M]] \mid T'}$$

$$\text{SpecJoin}\ \frac{\Theta : n[S_1' : (S, N_0), S_2', N] \in T}{H; \Theta[\cdot, S_2, E[\textbf{specRun}(\textbf{return}\ M_1, N_0)]] \mid T \rightarrow_s H; \Theta : n[adopt(S_1', E, \textbf{return}\ M_1), S_2', E[\textbf{specJoin}(\textbf{return}\ N_1, N)]] \mid T}$$

$$\text{SpecDone}\ \frac{}{H; \Theta[\cdot, S_2, E[\textbf{specJoin}(\textbf{return}\ N_1, \textbf{return}\ N_2)]] \mid T \rightarrow_s H; T \mid \Theta[\cdot, S_2, E[\textbf{return}(N_1, N_2)]]}$$

$$\text{SpecRaise}\ \frac{}{H; \Theta[\cdot, S_2, E[\textbf{specJoin}(\textbf{return}\ N_1, \textbf{raise}\ E)]] \mid T \rightarrow_s H; \Theta[\cdot, S_2, E[\textbf{raise}\ M]] \mid T}$$

$$\text{PopRead}\ \frac{H(x) = \langle \Theta, \textbf{C}, \uplus ds, \textbf{C}, \Theta', M \rangle}{H; \Theta[S_1 : (R, x, N'), S_2, N] \mid T \rightarrow_s H; \Theta[S_1, (R, x, N') : S_2, N] \mid T}$$

$$\text{PopWrite}\ \frac{H(x) = \langle \textbf{C}, ds, \textbf{S}, \Theta, M \rangle, \quad H' = H[x \mapsto \langle \textbf{C}, ds, \textbf{C}, \Theta, M \rangle]}{H; \Theta[S_1 : (W, x, N'), S_2, N] \mid T \rightarrow_s H'; \Theta[S_1, (W, x, N') : S_2, N] \mid T)}$$

$$\text{PopNewFull}\ \frac{H(x) = \langle \textbf{S}, ds, \textbf{S}, \Theta', M \rangle, \quad H' = H[x \mapsto \langle \textbf{C}, ds, \textbf{S}, \Theta', M \rangle]}{H; \Theta[S_1 : (A, x, M''), S_2, M'] \mid T \rightarrow_s H'; \Theta[S_1, (A, x, M'') : S_2, M'] \mid T}$$

$$\text{PopNewEmpty}\ \frac{H(x) = \langle \textbf{S} \rangle, \quad H' = H[x \mapsto \langle \textbf{C} \rangle]}{H; \Theta[S_1 : (A, x, M'), S_2, M] \mid T \rightarrow_s H'; \Theta[S_1, (A, x, M') : S_2, M] \mid T}$$

$$\text{PopFork}\ \frac{}{H; \Theta[S_1 : (F, \Theta', M'), S_2, M] \mid \Theta'[S_1' : CSpec, S_2', N] \mid T \rightarrow_s H; \Theta[S_1, (F, \Theta', M') : S_2, M] \mid \Theta : 1[S_1', CSpec : S_2', N] \mid T}$$

Figure 4. operational Semantics

$$\boxed{\text{rollback}(\Theta, S, H, T) \rightsquigarrow (H', T')}$$

$$\text{RBDone}\ \frac{}{\text{rollback}(\Theta, S, H, \Theta[S, S_2, M] \mid T) \rightsquigarrow (H, \Theta[S, S_2, M] \mid T)}$$

$$\text{RBRead}\ \frac{\begin{array}{c} H(x) = \langle s_1, \Theta' \uplus ds, \mathbf{S}, t, M\rangle, \\ H' = H[x \mapsto \langle s_1, ds, \mathbf{S}, t, M\rangle] \\ \text{rollback}(\Theta, S, H', \Theta'[S_1, S_2, M'] \mid T) \rightsquigarrow (H'', T') \end{array}}{\text{rollback}(\Theta, S, H, \Theta'[(R, x, M') : S_1, S_2, M] \mid T) \rightsquigarrow (H'', T')}$$

$$\text{RBFork}\ \frac{\begin{array}{c} T = \Theta''[CSpec, S_2', M''] \mid T' \\ \text{rollback}(\Theta, S, H, \Theta'[S_1, S_2, M'] \mid T') \rightsquigarrow (H', T'') \end{array}}{\text{rollback}(\Theta, S, H, \Theta'[(F, \Theta'', M') : S_1, S_2, M] \mid T) \rightsquigarrow (H', T'')}$$

$$\text{RBWrite}\ \frac{\begin{array}{c} H(x) = \langle s, \emptyset, \mathbf{S}, \Theta', M\rangle, \quad H' = H[x \mapsto \langle s\rangle] \\ \text{rollback}(\Theta, S, H', \Theta'[S_1, S_2, M'] \mid T) \rightsquigarrow (H'', T') \end{array}}{\text{rollback}(\Theta, S, H, \Theta'[(W, x, M') : S_1, S_2, M] \mid T) \rightsquigarrow (H'', T')}$$

$$\text{RBNew}\ \frac{\begin{array}{c} H(x) = \langle \mathbf{S}\rangle, \quad H' = H\backslash x \\ \text{rollback}(\Theta, S, H', \Theta'[S_1, S_2, M'] \mid T) \rightsquigarrow (H'', T') \end{array}}{\text{rollback}(\Theta, S, H, \Theta'[(A, x, M') : S_1, S_2, M] \mid T) \rightsquigarrow (H'', T')}$$

Figure 5. Rollback

whether or not is was allocated speculatively. If an IVar is full we record if it was allocated speculatively, the thread IDs of those who have read the IVar, whether or not it was written speculatively, the ID of the writer, and the term written to the IVar. A thread pool is a multiset of threads, where each thread has a thread ID, a list (queue) of speculative actions it has performed, a list of actions it has committed, and a term that it is evaluating. Action queues are a list of actions, where an action can be a read, write, spec, allocation, fork, or an action indicating it was created speculatively. Lastly, a configuration is a heap paired with a thread pool, or the error state.

The overall semantics of the language is described by a big step relation, which is used to represent the "usual" Haskell semantics. In this presentation and in [MNP11], we only give the big-step rule for **runPar** as the rest is entirely conventional. The RunPar rule then depends on a small step relation for the Speculative Par Monad presented in Figure 4. The small step semantics relates a heap H and a thread pool T to either a new Heap and new thread pool, or the error state if multiple writes occurred to a single IVar. Rules Bind, BindRaise, Handle, and HandleReturn are standard monadic bind and exception handling rules. The Eval rule dispatches back to the big step semantics for reducing non-monadic terms (such as beta reduction, creating tuples, projecting tuples, etc...).

The Fork rule spawns a new thread, and records a fork action on the thread performing the fork along with the thread ID of the forked thread. we uniquely name threads by adding a number onto the forking thread's ID that is equal to the number of threads that have already been created by this thread. The forked thread is then created with an action on its stack indicating it was created speculatively, and not allowing it to commit any actions. When the forking thread has a fork action at the head of its action queue, it can commit this action, moving the fork action over to its commit list, and moving the $CSpec$ action over to the forked thread's commit list. The New rule allocates a new IVar, marking it as having been allocated speculatively. When the allocation action makes its way to the head of the action queue, it can then change the state of the IVar from speculative to commit using the PopNewFull or PopNewEmpty rule. The Get rule is used to read from an IVar, if the IVar is full, then we add a read action to the threads speculative action queue, and record the thread's ID in the IVar indicating that if this IVar is rolled back, this thread is a dependent reader. The PopRead rule can then be used to commit this read action assuming the IVar is now in commit mode. The Put rule is used to write to an IVar, assuming it is empty, we fill the contents of the IVar and add a write action to the thread's action list. This action can then be committed using the PopWrite rule, which sets the status of the IVar to commit written.

The Overwrite rule applies when a thread has no speculative actions (i.e. it is in commit mode) and is attempting to write to an IVar that is speculatively full. When looking up the IVar in the heap we see that it previously was written by thread Θ', which we then lookup in the pool and split its speculative action queue into those actions that happened after the write, and those that happened before the write to this IVar. We then perform a rollback with respect to thread Θ', which is described later. For now it suffices to know that it undoes all actions performed by Θ', up to S_1', which correspond to the actions performed before the write to x. We then update IVar x to contain the value being written by thread Θ. The ErrorWrite rule is similar to Overwrite, except the IVar being written is commit full, which corresponds to an error.

The Spec rule begins a speculative computation, which behaves similarly to the Fork rule with a few differences. First, notice that we add two actions to the created thread's speculative action list, The first is an action indicating it was created speculatively as is done in the Fork rule, but we also include the (S, N) action indicating that it is the right branch of a speculative computation with initial term N. When the fork action makes its way to the front of Θ's action list, we remove the $CSpec$ action, but the (S, N) action remains on the speculative list, disallowing this thread from committing anything until it joins with its corresponding commit thread in the SpecJoin rule.

The SpecRB rule corresponds to canceling a speculative thread, where we rollback the canceled thread's actions similarly to what is done in the Overwrite rule. The SpecJoin rule is used for joining a speculative computation. When the thread executing the left branch of a speculative computation is finished, we adopt the term being evaluated by the speculative thread, and all of its speculative actions and transition to the **specJoin** intermediate form. The SpecDone and SpecRaise rules are used to finish a speculative computation when the right branch evaluates to a returned value or raised exception respectively.

Figure 5 provides the semantics for performing a rollback. The rollback function takes a thread ID, Θ, to rollback with respect to, a list of actions, S, such that the rollback stops when thread Θ has this list of actions S, a heap, and a thread pool. The result of a rollback is then a new heap and a new thread pool.

The RBDone rule indicates that the rollback is complete when thread Θ has as its action list S. The RBRead rule is used to undo a read action. It must be the case that the thread's ID is present in the set of dependent readers on the IVar when looked up in the heap, so we remove the ID from the set, and continue with the rollback, resetting the thread to the term associated with the read action. RBFork is applicable when the thread associated with a fork action has nothing but the created speculative action in its speculative list, we then proceed with the rollback by throwing away the forked thread, and reset the forking thread to the term associated with the action. RBWrite undoes a write action when the IVar written to has no recorded dependent readers, we then proceed by resetting the IVar to empty and resetting the writing thread to the term associative with the write action. RBNew undoes an allocation action when the IVar being rolled back was speculatively created, we remove it from the heap and continue after resetting the thread back to the term associated with the allocation action.

$$
\begin{array}{rcl}
\mathcal{E}[\![H;T]\!] & = & \mathcal{E}[\![H]\!];\mathcal{E}[\![T]\!] \\
\mathcal{E}[\![T_1 \mid T_2]\!] & = & \mathcal{E}[\![T_1]\!] \mid \mathcal{E}[\![T_2]\!] \\
\mathcal{E}[\![\Theta[S_1 : (R, x, M'), S_2, M]]\!] & = & M' \\
\mathcal{E}[\![\Theta[S_1 : (W, x, M'), S_2, M]]\!] & = & M' \\
\mathcal{E}[\![\Theta[S_1 : (A, x, M'), S_2, M]]\!] & = & M' \\
\mathcal{E}[\![\Theta[S_1 : (F, \Theta', M'), S_2, M]]\!] & = & M' \\
\mathcal{E}[\![\Theta[S_1 : (S, M'), S_2, M]]\!] & = & \cdot \\
\mathcal{E}[\![\Theta[S_1 : CSpec, S_2, M]]\!] & = & \cdot \\
\mathcal{E}[\![\Theta[\cdot, S_2, M]]\!] & = & M \\
\mathcal{E}[\![H, x \mapsto \langle \mathbf{S} \rangle]\!] & = & \mathcal{E}[\![H]\!] \\
\mathcal{E}[\![H, x \mapsto \langle \mathbf{C} \rangle]\!] & = & \mathcal{E}[\![H]\!], x \mapsto \langle\rangle \\
\mathcal{E}[\![H, x \mapsto \langle \mathbf{S}, ds, s, \Theta, M \rangle]\!] & = & \mathcal{E}[\![H]\!] \\
\mathcal{E}[\![H, x \mapsto \langle \mathbf{C}, ds, \mathbf{S}, \Theta, M \rangle]\!] & = & \mathcal{E}[\![H]\!], x \mapsto \langle\rangle \\
\mathcal{E}[\![H, x \mapsto \langle \mathbf{C}, ds, \mathbf{C}, \Theta, M \rangle]\!] & = & \mathcal{E}[\![H]\!], x \mapsto \langle M \rangle
\end{array}
$$

Figure 6. Erasure

$$
\begin{array}{rcl}
\mathcal{US}[\![H;T]\!] & = & \mathcal{US}[\![H]\!];\mathcal{US}[\![T]\!] \\
\mathcal{US}[\![T_1 \mid T_2]\!] & = & \mathcal{US}[\![T_1]\!] \mid \mathcal{US}[\![T_2]\!] \\
\mathcal{US}[\![\Theta[S_1 : (R, x, M'), S_2, M]]\!] & = & \Theta[\cdot, S_2, M'] \\
\mathcal{US}[\![\Theta[S_1 : (W, x, M'), S_2, M]]\!] & = & \Theta[\cdot, S_2, M'] \\
\mathcal{US}[\![\Theta[S_1 : (A, x, M'), S_2, M]]\!] & = & \Theta[\cdot, S_2, M'] \\
\mathcal{US}[\![\Theta[S_1 : (F, \Theta', M'), S_2, M]]\!] & = & \Theta[\cdot, S_2, M'] \\
\mathcal{US}[\![\Theta[S_1 : (S, M'), S_2, M]]\!] & = & \Theta[\cdot : (S, M'), S_2, M'] \\
\mathcal{US}[\![\Theta[S_1 : CSpec, S_2, M]]\!] & = & \cdot \\
\mathcal{US}[\![\Theta[\cdot, S_2, M]]\!] & = & \Theta[\cdot, S_2, M] \\
\mathcal{US}[\![H, x \mapsto \langle \mathbf{S} \rangle]\!] & = & \mathcal{US}[\![H]\!] \\
\mathcal{US}[\![H, x \mapsto \langle \mathbf{C} \rangle]\!] & = & \mathcal{US}[\![H]\!], x \mapsto \langle \mathbf{C} \rangle \\
\mathcal{US}[\![H, x \mapsto \langle \mathbf{S}, ds, s, \Theta, M \rangle]\!] & = & \mathcal{US}[\![H]\!] \\
\mathcal{US}[\![H, x \mapsto \langle \mathbf{C}, ds, \mathbf{S}, \Theta, M \rangle]\!] & = & \mathcal{US}[\![H]\!], x \mapsto \langle \mathbf{C} \rangle \\
\mathcal{US}[\![H, x \mapsto \langle \mathbf{C}, ds, \mathbf{C}, \Theta, M \rangle]\!] & = & \mathcal{US}[\![H]\!], x \mapsto \langle \mathbf{C}, \emptyset, \mathbf{C}, \Theta, M \rangle
\end{array}
$$

Figure 7. Unspeculate

5. Proof of Determinism

The overall proof strategy is to first prove an equivalence to the original Par Monad, which is known to be deterministic [MNP11, BBC+10], and then deducing determinism for our speculative extension from this equivalence. For the reader's convenience, we have restated the semantics of the original Par Monad in the Appendix. Those familiar with [MNP11] will notice some slight differences between the two presentations. First, we have used an explicit heap for IVars, where as the original semantics mixes threads with IVars in the style of the π-calculus. Second, we have added syntax for speculative computations in Par, however, it is evaluated sequentially and essentially equivalent to a special case of the bind construct. More concretely, **spec** $M\ N$ can be de-sugared to $M\ >>=\ \backslash i.(N\ >>=\ \backslash j.\mathbf{return}(i, j))$ where i does not occur free in N. Lastly, in the original semantics, threads were allowed to terminate in the middle of a computation when they complete, where as in our presentation, we keep them around to the end of a **runPar**.

Before stating our equivalence theorem, we first introduce an erasure in Figure 6 that relates speculative program states to non speculative (Par Monad) program states. Intuitively, the erasure recursively goes through the program state, and "throws away" speculative work. If a thread has speculative actions, we reset them to the term associated with the oldest action in their list for read, write, allocation, and fork actions. If the oldest action indicates that it was created speculatively, or it is a thread executing the right branch of a **spec**, then we simply throw away the thread as these threads would not yet have been created in the non speculative semantics. When erasing the heap, we throw out any IVars that were speculatively created. If an IVar was commit created, but was speculatively written, then the erasure simply resets it to empty.

We can now relate the behaviors in one language to the behaviors in the other, where behaviors are defined as:

$$
\begin{array}{rcl}
\beta_s[M] & = & \{V \mid \mathbf{runPar}\ M \Downarrow_s b\} \\
\beta_p[M] & = & \{V \mid \mathbf{runPar}\ M \Downarrow_p b\}
\end{array}
$$

Here the s subscript is used to denote a large step in the speculative semantics and a p subscript is used to denote a large step in the non speculative (Par) semantics. Also, in this case b represents all possible outcomes of **runPar** (i.e. b could be some term M, **Error**, or ∞).

There is an interesting point to be made about proving an equivalence between diverging programs. In the speculative language it is possible to have divergent programs that can converge in the non-specualtive language if care is not taken. As an example consider the program:

$$\mathbf{runPar}\ (\mathbf{spec}\ (\mathbf{raise}\ M)\ N)$$

Where N is a divergent computation. In the speculative language, there is nothing that forces us to make progress on the commit portion of a speculative computation, therefore this program could infinitely take steps on N, despite the fact that if the left branch of the **spec** ever got a chance to run it would cancel the divergent computation. In the non speculative language this is not an issue as progress cannot be made on the right branch of a **spec** until the left branch has been evaluated to a raised exception or returned value. Typically one would define divergence as:

$$\frac{H; T \rightarrow_s H'; T' \quad H'; T' \rightarrow_s^\infty}{H; T \rightarrow_s^\infty}$$

However for our purposes we must state a more restrictive version of divergence:

$$\frac{H; T \rightarrow_{\text{spec}}^* H'; T' \quad H'; T' \rightarrow_{\text{commit}} H''; T'' \quad H''; T'' \rightarrow_s^\infty}{H; T \rightarrow_s^\infty}$$

Where the $\rightarrow_{\text{commit}}$ relation is the same as the step relation presented in Figure 4 except that we restrict that the thread taking the step does not have any uncommitted actions and the $\rightarrow_{\text{spec}}$ relation is the complement of $\rightarrow_{\text{commit}}$. Essentially we are enforcing a fairness policy requiring that progress must be made on a commit thread in order for a program state to be divergent. Note that this leaves the class of speculatively divergent programs undefined in our formalism, however, we do not believe this is an issue as those programs will have a defined behavior in an actual implementation assuming a fair scheduling policy.

At this point we are able to state our equivalence theorem

Theorem 1 (Equivalence)**.** $\forall M, \beta_s[M] = \beta_p[M]$

Proof Sketch. We show $\forall b \in \beta_s[M] \Rightarrow b \in \beta_p[M]$ and $\forall b \in \beta_p[M] \Rightarrow b \in \beta_s[M]$. The most interesting case is showing $V \in \beta_s[M] \Rightarrow V \in \beta_p[M]$ where V is the result of a successfully converging program in the speculative language (i.e. not an error or divergent program), which follows from Lemma 1 ☐

Lemma 1 (Speculative Implies Nonspeculative)
If $\cdot; 1[\cdot, \cdot, M\ >>=\ \backslash x.\mathbf{done}\ x] \rightarrow_s^* H_s; T_s \mid 1[\cdot, S_2, \mathbf{done}\ N]$ *and* *Finished*(T_s) *then*
$\exists H_p\ \ T_p, \cdot; M\ \ >>=\ \ \backslash x.\mathbf{done}\ \ x\ \ \rightarrow_p^*\ \ H; T_p\ \mid\ \mathbf{done}\ \ N$ *and* $\mathcal{E}[\![H_s; T_s]\!] = H_p; T_p$ *and Finished*(T_p)

This is proven with a good amount of infrastructure behind it. First we define a metafunction in Figure 7 similar to erasure that relates a program state to its "commit frontier" which essential abandons any speculative work that has been done. This unspeculate function is then used to state a well-formedness property on speculative program states:

$$\frac{\mathcal{US}[\![H;T]\!] \rightarrow_s^* H;T}{\mathrm{WF}(H;T)}$$

Intuitively, this says that a program state is well formed if we can abandon all speculative work that has been done and get back to the exact point we were at before unspeculating. Lemma 1 then follows from a more general restatement.

Lemma 2 (Speculative Implies Nonspeculative WF)
If WF$(H_s;T_s)$ *and* $H_s;T_s \rightarrow_s^* H_s';T_s'$ *then*
$\exists H_p'\ T_p', \mathcal{E}[\![H_s;T_s]\!] \rightarrow_p^* H_p';T_p'$ *and* $\mathcal{E}[\![H_s';T_s']\!] = H_p';T_p'$

Proof Sketch. By induction on the derivation of $H_s;T_s \rightarrow_s^* H_s';T_s'$ and case analysis on the first step taken in the derivation. If the first step is a speculative step (i.e. the thread taking the step has uncommitted actions), then take zero steps in the non speculative semantics as $\mathcal{E}[\![H_s;T_s]\!] = \mathcal{E}[\![H_s';T_s']\!]$. If the first step corresponds to Eval, Bind, BindRaise, Handle, HandleReturn, Fork, New. Get, Put, Overwrite, ErrorWrite, Spec, SpecRB, SpecJoin, SpecDone, or SpecRaise, and the thread taking the step does not have any uncommitted actions, then we take the one corresponding step in the non speculative semantics. If the first step corresponds to PopRead, PopWrite, PopNewFull, PopNewEmpty, or PopFork, then the speculative program must "catch up" by performing the action being committed and all of the pure steps between the action being committed and the next uncommitted action if any. Fortunately, the sequence of steps necessary to catch up is given to us by the well-formedness derivation. ☐

Once we have established the equivalence, we can deduce determinism easily assuming that the non speculative language is deterministic

Theorem 1 (Par Monad Deterministic)
If **runPar** $M \Downarrow_p V_1$ *and* **runPar** $M \Downarrow_p V_2$*, then* $V_1 = V_2$

Proof Sketch. This is assumed based on previous work ☐

Theorem 2 (Speculative Par Monad Deterministic)
If **runPar** $M \Downarrow_s V_1$ *and* **runPar** $M \Downarrow_s V_2$*, then* $V_1 = V_2$

Proof Sketch. By case analysis on both **runPar** $M \Downarrow_s V_1$ and **runPar** $M \Downarrow_s V_2$. If V_1 and V_2 are the results of successfully converging programs, then by Lemma 1 we have **runPar** $M \Downarrow_p V_1$ and **runPar** $M \Downarrow_p V_2$. From Theorem 1 we have $V_1 = V_2$. The other cases are proven similarly.

☐

Note that many cases and supporting lemmas are left out for brevity and that the proof sketches provided are only meant to give the reader a high level intuition as to how the details of the proof fit together. Full details about the proof can be found in the Coq formalization at `http://people.rit.edu/ml9951/research.html`

6. Implementation

In addition to the formal semantics and determinism proof we have also begun a preliminary implementation as a part of the Manticore project. We have implemented the rollback mechanism and an IVar library using the BOM intermediate language that is used for much of the rest of the runtime system and thread scheduling infrastructure [FRR08]. One key feature that the BOM intermediate language has is first class continuations, which allow us to "reset" threads to previous points in their evaluation.

6.1 Threads in Manticore

In Manticore, threads are simply represented as a unit continuation and a pointer to thread local storage. We store the action list described in the formal semantics inside of thread local storage. When a thread is created, we can provide a cancelable object such that each time the thread is scheduled, it first checks to see if a flag in the cancelable object has been set and if so, it terminates. More details about about cancelation and thread scheduling can be found in [FRRS11].

6.2 IVars

An IVar is represented as a record almost identically as it is in the formal semantics. The main difference is in the list of dependent readers of an IVar. In the formal semantics, this is simply a multi set of thread IDs, however, in our implementation it is actually a tuple containing the cancelable object associated with the reader, a continuation corresponding to the current continuation of the reader at the point in which it read the IVar, and a pointer to the list of actions it has performed. When a thread reads from an IVar, it captures its current continuation, and stores it in the IVar along with its cancelable object and action pointer. In the event that a rollback is invoked, we recursively go through the list of actions to be rolled back doing the following for each action:

- If the action is a fork action, cancel the forked thread (cancelable object is stored in the action object) and append all of the forked thread's actions to the list of actions to be rolled back
- If the action is a read action, we simply continue with the rollback
- If the action is a write action, reset the IVar to empty, and process each of the dependent readers associated with this IVar.
    - When processing dependent readers, we recurse down the list of actions they have been performed and look for the oldest read action to the IVar being rolled back. Note that it must be the oldest action because if the thread read from the IVar multiple times, we need to reset it back to the point at which it read from the IVar for the first time.
    - We then reset this thread to the continuation associated with this read action and append the actions occurring after the read to the list of actions to be rolled back.

Note that we do not record an action for allocating an IVar. This is done in the formal semantics for the purposes of maintaining the well-formedness property and is not necessary to rollback the creation of IVars as they will simply be garbage collected. The reason this is important for preserving the well-formedness property is that after unspeculating a program state, it must be able to run forward to exactly the state it was in prior to unspeculating. This means that the names chosen for IVars in the heap must be the same as they were previously, which would not be possible if speculatively allocated IVars where not removed from the heap.

As a final technical detail, when "resetting" threads to previous points in their evaluation we actually simply cancel the thread to be reset. We then create a new thread with the same identity, except that it begins its evaluation at the continuation corresponding to the point in which it is to be "reset".

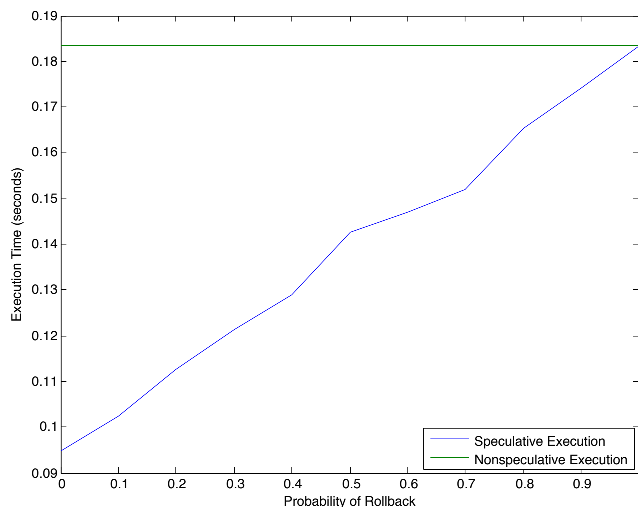

Figure 8. Rollback Overhead

7. Preliminary Results

Our implementation is still in its early stages, however, we have been able to perform some preliminary evaluation in order to give the reader a sense as to what sort of overhead is introduced by our logging and rollback mechanism in Manticore.

7.1 Producer Consumer

The first benchmark is a simple program that spawns two threads, one that repeatedly writes some arbitrary data to a linked list of IVars and another that reads each element of the linked list as it becomes available. This gives us some idea as to what sort of price we pay even if we have no interest in doing any sort of speculative computation. For a program that writes 5,000 IVars we see only a 5% slowdown relative to an implementation that performs no logging.

7.2 Measuring Rollback

In an effort to measure the overhead introduced by the rollback mechanism we have constructed a synthetic benchmark that forks a thread that speculatively writes to an IVar, and with a given probability raises an exception to rollback the write after a predetermined amount of time. After forking this thread, the main thread then reads from the speculatively written IVar in order to record a dependent reader and then enters a spin loop for the same predetermined amount of time as the forked thread. When a rollback occurs, the runtime system will then reset the written IVar to empty and reset the main thread to before the point that it read from the IVar. If a rollback does not occur, then the two spin loops are executed in parallel and should, in theory, achieve 2X speedup. Figure 8 shows the results of the experiment varying the probability of performing a rollback from 0 to 1 in 0.1 increments. The execution times for each probability interval are the average of 500 iterations. For the non speculative results, we simply run the two spin loops sequentially in order to get a baseline execution that does not involve the runtime system. The results indicate that for this particular scenario, the overhead of a rollback is essentially free. The average runtime of the non speculative case is 0.1833 seconds vs. 0.1834 for the average speculative runtimes with a 100% chance of a rollback occurring.

Certainly these results will vary based on the "size" of the rollback, meaning if we had more threads dependently reading from the IVar, or we were speculatively writing to more IVars, the execution time would definitely be different as the runtime system would need to do more work. Future work will include a more in-depth analysis of these parameters.

8. Related Work

This work builds on two broad categories of related projects, those that deal with deterministic parallelism in the presence of shared state, and those that deal with speculative parallelism.

8.1 Shared State

IVars were first proposed in the language Id [ANP89], which is also a parallel functional language, however, they sacrifice determinism by also adding MVars, which are shared references that can be written an arbitrary number of times with implicit synchronization. More recently, IVars have been adopted by parallel languages such as the Par Monad of Haskell [MNP11] and some of the Concurrent Collections work[BBC$^+$10], however, neither of these works support speculative parallelism.

LVars are a new abstraction that were recently proposed by Kuper et al. [KN13, KTKN14] that generalize IVars to allow multiple writes, but restrict that they must be monotonically increasing in some fashion. LVars suffer from the same problem as IVars in that they also lose their determinism guarantee in the presence of cancelation. More recently, they have proposed an elegant solution for combining LVars with speculative parallelism [KTTN14], where threads can perform speculative work (i.e. can potentially be canceled) if they are read only. They do however, allow speculative threads to write to memoization tables such that they can "help out" other threads, however, one shortcoming to this solution is that performance becomes difficult to reason about as a programmer. Note that parallel speedup is only achieved if the speculative thread is able to write to the memoization table before another thread needs this result. If it does not make it there in time, then not only is there no benefit, but the speculative thread corresponds to wasted work. On the other hand in this work, if the commit portion of a parallel tuple finishes before the speculative threads, it simply waits for them to complete and then joins with them.

Welc et al. proposed a solution for enforcing a sequential semantics for Java futures [WJH05, NZJ08], a concurrency abstraction taken from Multilisp [Hal85]. They too extend their runtime system to enforce deterministic execution, but in a very different way relative to our approach. First, for each thread that is spawned, they create a new copy for each object that it writes to. This does not allow for the type of fine grained sharing that we are able to support in our producer-consumer benchmark. Additionally, if their runtime system detects that a thread has violated the sequential semantics, they restart the thread from the beginning, where as our approach is able to simply rollback a thread to the exact point in which the violation occurred, avoiding redundant work.

Bocchino et al. give a region based type and effect system for guaranteeing determinism at compile time for parallel Java programs [BAD$^+$09]. Their approach requires annotations on Java programs specifying what "regions" objects are allocated in. They then extend their Java compiler to statically verify that concurrently executing threads do not manipulate objects that are allocated in the same regions.

8.2 Speculative Parallelism

There is a large body of work that has been done on transparent speculative parallelism, where the compiler and runtime system automatically perform value prediction and control the amount of parallelism in the program, however, more relevant to this work is the notion of programmable speculative parallelism. Programmable

speculative parallelism was first introduced in [Bur85] in the context of the Mirranda language. Their approach uses a purely functional language, so they do not deal with any of the rollback issues that we present in this work.

More recently, Prabhu et al. propose language constructs for specifying speculatively parallel algorithms and formalize their semantics using the lambda calculus extended with shared references [PRV10]. Rather than providing a runtime system that performs rollbacks in the event of a miss-speculated value, they describe an analysis that is performed at compile-time that guarantees that they will never need to perform any rollbacks. Their analysis is necessarily conservative, making certain types of sharing patterns not expressible in their language.

Software Transactional Memory (STM) can be seen as a form of speculative parallelism. Transactional memory allows programmers to wrap code in "atomic" blocks that the runtime system guarantees to be executed in isolation [ST95]. STM uses a form of "optimistic" concurrency where threads execute code inside of transactions and upon completion check to see if any of the memory locations they read or wrote were compromised by other concurrently running threads. If so, they abort the transaction and restart from the beginning. Transactional memory is different from our work in the sense that they provide no guarantees about deterministic execution, and is concerned only with atomicity.

9. Conclusions and Future Work

Giving parallel constructs a deterministic semantics makes reasoning about parallel programs substantially easier. In this work we have shown how we can extend the expressiveness of Manticore by adding IVars and still be able to guarantee deterministic execution. We have formalized the semantics of this extended language and provided a proof of its correctness using the Coq proof assistant.

For our preliminary implementation we have tried to remain faithful to the formal semantics as much as possible to ensure correctness without worrying too much about performance. In the immediate future we plan on fine tuning our implementation of the runtime system in Manticore to improve efficiency and perform a more thorough evaluation. This idea of combining speculative parallelism with IVars is a new programming model that has not been explored elsewhere so coming up with interesting benchmark programs is also a bit of a challenge and something we look to explore further in the future. Lastly, we believe it would be interesting in generalizing our approach to the LVars programming model. As mentioned in the previous section, this is an extension of the IVars model that permits multiple writes to shared references, so extending both our implementation and our formal semantics presents some interesting challenges.

References

[ANP89] Arvind, R. S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM TOPLAS*, **11**(4), October 1989, pp. 598–632.

[BAD+09] Bocchino, Jr., R. L., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *OOPSLA'09*, Orlando, Florida, USA, 2009. ACM, pp. 97–116.

[BBC+10] Budimlić, Z., M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tacsirlar. Concurrent collections. *Sci. Program.*, **18**(3-4), August 2010, pp. 203–217.

[Bur85] Burton, F. W. Speculative computation, parallelism, and functional programming. *IEEE Trans. Computers*, **34**(12), 1985, pp. 1190–1193.

[FRR08] Fluet, M., M. Rainey, and J. Reppy. A scheduling framework for general-purpose parallel languages. In *ICFP '08*, Victoria, BC, Candada, September 2008. ACM, pp. 241–252.

[FRRS08] Fluet, M., M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. In *ICFP '08*, Victoria, BC, Candada, September 2008. ACM, pp. 119–130.

[FRRS11] Fluet, M., M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. *JFP*, **20**(5–6), 2011, pp. 537–576.

[Hal85] Halstead Jr., R. H. Multilisp: A language for concurrent and symbolic computation. *ACM TOPLAS*, **7**, 1985, pp. 501–538.

[JLM+09] Jones, C. G., R. Liu, L. Meyerovich, K. Asanović, and R. Bodík. Parallelizing the web browser. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, Berkeley, California, 2009.

[KN13] Kuper, L. and R. R. Newton. Lvars: lattice-based data structures for deterministic parallelism. In *FHPC'13*, Boston, Massachusetts, USA, 2013. ACM, pp. 71–84.

[KTKN14] Kuper, L., A. Turon, N. R. Krishnaswami, and R. R. Newton. Freeze after writing: Quasi-deterministic parallel programming with lvars. In *POPL'14*, San Diego, California, USA, 2014. ACM, pp. 257–270.

[KTTN14] Kuper, L., A. Todd, S. Tobin-Hochstadt, and R. Newton. Taming the parallel effect zoo. In *PLDI'14*, Edinburgh, UK, 2014. ACM.

[MNP11] Marlow, S., R. Newton, and S. Peton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*. ACM, 2011, pp. 71–82.

[NZJ08] Navabi, A., X. Zhang, and S. Jagannathan. Quasi-static scheduling for safe futures. In *PPOPP'08*, Salt Lake City, UT, USA, 2008. ACM, pp. 23–32.

[PRV10] Prabhu, P., G. Ramalingam, and K. Vaswani. Safe programmable speculative parallelism. In *PLDI'10*, Toronto, Ontario, Canada, 2010. ACM, pp. 50–61.

[ST95] Shavit, N. and D. Touitou. Software transactional memory. In *PODC '95*, Ottowa, Ontario, Canada, 1995. ACM, pp. 204–213.

[WJH05] Welc, A., S. Jagannathan, and A. Hosking. Safe futures for java. In *OOPSLA'05*, San Diego, CA, USA, 2005. ACM, pp. 439–453.

Acknowledgments

Thanks to Vitor Rodriquez and Zack Fitzimmons for their many useful comments that helped to improve this work. This research is supported by the National Science Foundation under Grants CCF-0811389 and CCF-101056

10. Appendix

$\boxed{\text{Finished}(H;T)}$

$$\frac{\text{Finished}(H;T_1) \mid \text{Finished}(H;T_2)}{\text{Finished}(H;T_1 \mid T_2)} \qquad \frac{H(x) = \langle \mathbf{C} \rangle}{\text{Finished}(H;\Theta[S_1,S_2,E[\mathbf{get}\ x]])}$$

$$\frac{}{\text{Finished}(H;\Theta[\cdot,S_2,\mathbf{return}\ M])} \qquad \frac{}{\text{Finished}(H;\Theta[S:A,S_2,M])}$$

Figure 9. Finished Thread Pool

$$
\begin{array}{rcl}
adopt(S:(R,x,M),E,M') & = & adopt(S,E,N):(R,x,E[\textbf{specJoin}(N,M)]) \\
adopt(S:(W,x,M),E,M') & = & adopt(S,E,N):(W,x,E[\textbf{specJoin}(N,M)]) \\
adopt(S:(A,x,M),E,M') & = & adopt(S,E,N):(A,x,E[\textbf{specJoin}(N,M)]) \\
adopt(S:(F,\Theta,M),E,M') & = & adopt(S,E,N):(F,\Theta,E[\textbf{specJoin}(N,M)]) \\
adopt(S:(S,M),E,M') & = & adopt(S,E,N):(S,E[\textbf{specJoin}(N,M)]) \\
adopt(S:CSpec,E,M') & = & adopt(S,E,N):CSpec
\end{array}
$$

Figure 10. Action Adoption

$$
\begin{array}{rrcl}
 & x & \in & Var \\
\text{Values} & V & ::= & x \mid i \mid \backslash x.M \mid \textbf{return}\ M \mid M >>= N \mid \textbf{runPar}\ M \mid \textbf{fork}\ M \mid \textbf{new} \mid \textbf{put}\ i\ M \\
 & & \mid & \textbf{get}\ i \mid \textbf{done}\ M \mid \textbf{spec}\ M\ N \mid \textbf{specRun}(M,N) \mid \textbf{specJoin}(M,N) \mid \textbf{raise}\ M \\
 & & \mid & \textbf{handle}\ M\ N \\
\text{Terms} & M,N & ::= & V \mid M\ N \mid \cdots \\
\text{Heap} & H & ::= & H, x \mapsto iv \mid \cdot \\
\text{IVar State} & iv & ::= & \langle\rangle \mid \langle M \rangle \\
\text{Evaluation Context} & E & ::= & [\cdot] \mid E >>= M \mid \textbf{specRun}(E,M) \mid \textbf{handle}\ E\ N \mid \textbf{specJoin}(N,E) \\
\text{Thread Pool} & T & ::= & \cdot \mid (T_1 \mid T_2) \mid M \\
\text{Configuration} & \sigma & ::= & H;T \mid Error
\end{array}
$$

Figure 11. Original Par Monad Syntax

$$
\text{FPar}\ \frac{\text{Finished}_\text{p}(H;T_1) \quad \text{Finished}_\text{p}(H;T_2)}{\text{Finished}_\text{p}(H;T_1 \mid T_2)} \qquad
\text{FBlocked}\ \frac{H(x) = \langle\rangle}{\text{Finished}_\text{p}(H;E[\textbf{get}\ x])} \qquad
\text{FDone}\ \frac{}{\text{Finished}_\text{p}(H;\textbf{return}\ M)}
$$

Figure 12. Original Par Monad Finished

$$
\text{RunPar}\ \frac{(M >>= \backslash x.\textbf{done}\ x) \rightarrow_p^* \textbf{done}\ N \mid T \quad N \Downarrow V \quad \text{Finished}_\text{p}(T)}{\textbf{runPar}\ M \Downarrow V}
$$

$$
\text{Eval}\ \frac{M \Downarrow V}{H;T \mid E[M] \rightarrow_p H;T \mid E[V]} \qquad
\text{Bind}\ \frac{}{H;T \mid E[\textbf{return}\ N >>= M] \rightarrow_p H;T \mid E[M\ N]}
$$

$$
\text{BindRaise}\ \frac{}{H;T \mid E[\textbf{raise}\ N >>= M] \rightarrow_p H;T \mid E[\textbf{raise} N]} \qquad
\text{Handle}\ \frac{}{H;T \mid E[\textbf{handle}(\textbf{raise}\ M)N] \rightarrow_p H;T \mid E[M\ N]}
$$

$$
\text{HandleRet}\ \frac{}{H;T \mid E[\textbf{handle}(\textbf{return}\ M)N] \rightarrow_p H;T \mid E[\textbf{return}\ M]} \qquad
\text{Fork}\ \frac{}{H;T \mid E[\textbf{fork}\ M] \rightarrow_p H;E[\textbf{return}()] \mid M \mid T}
$$

$$
\text{New}\ \frac{x \notin Domain(H) \quad H' = H[x \mapsto \langle\rangle]}{H;E[\textbf{new}] \mid T \rightarrow_p H';T \mid E[\textbf{return}\ x]} \qquad
\text{Get}\ \frac{H(x) = \langle M \rangle}{H;E[\textbf{get}\ x] \mid T \rightarrow_p H;E[\textbf{return}\ M] \mid T}
$$

$$
\text{Put}\ \frac{H(x) = \langle\rangle \quad H' = H[x \mapsto \langle M \rangle]}{H;E[\textbf{put}\ x\ M] \mid T \rightarrow_p H';E[\textbf{return}()] \mid T} \qquad
\text{Spec}\ \frac{}{H;E[\textbf{spec}\ M\ N] \mid T \rightarrow_p H;E[\textbf{specRun}(M,N)] \mid T}
$$

$$
\text{SpecRun}\ \frac{}{H;E[\textbf{specRun}(\textbf{return}\ M,N)] \mid T \rightarrow_p H;E[\textbf{specJoin}(\textbf{return}\ M,N)] \mid T}
$$

$$
\text{SpecRaise}\ \frac{}{H;E[\textbf{specRun}(\textbf{raise}\ M,N)] \mid T \rightarrow_p H;E[\textbf{raise}\ M] \mid T}
$$

$$
\text{specJoin}\ \frac{}{H;E[\textbf{specJoin}(\textbf{return}\ M,\textbf{return}\ N)] \mid T \rightarrow_p H;E[\textbf{return}(M,N)] \mid T}
$$

$$
\text{specJoinRaise}\ \frac{}{H;E[\textbf{specJoin}(\textbf{return}\ M,\textbf{raise}\ N)] \mid T \rightarrow_p H;E[\textbf{raise}\ N] \mid T}
$$

Figure 13. Original Par Monad Operational Semantics