

Partial Memoization of Concurrency and Communication

Lukasz Ziarek Suresh Jagannathan

Department of Computer Science
Purdue University
{lziarek,suresh}@cs.purdue.edu

Abstract

Memoization is a well-known optimization technique used to eliminate redundant calls for pure functions. If a call to a function f with argument v yields result r , a subsequent call to f with v can be immediately reduced to r without the need to re-evaluate f 's body if the association between f , v , and r was previously recorded.

Understanding memoization in the presence of concurrency and communication is significantly more challenging. For example, if f communicates with other threads, it is not sufficient to simply record its input/output behavior; we must also track inter-thread dependencies induced by communication actions performed by these threads. Subsequent calls to f can be elided only if we can identify an interleaving of actions from these call-sites that lead to states in which these dependencies are satisfied. Similar issues arise if f spawns additional threads.

In this paper, we consider the memoization problem for a higher-order concurrent language which threads may communicate with one another through synchronous message-based communication. To avoid the need to perform unbounded state space search that may be necessary to determine if *all* communication dependencies manifest in an earlier call can be satisfied in a later one, we introduce a weaker notion of memoization called *partial memoization* that gives implementations the freedom to avoid performing some part, if not all, of a previously memoized call if the application evaluates in a state in which some, but not all, dependencies are satisfied.

To validate the effectiveness of our ideas, we consider the benefits of memoization for reducing the overhead of recomputation for speculative and transactional applications. For example, we show that on some workloads, memoization can reduce the overhead of re-executing an aborted transaction or failed speculation by over 40% without incurring high memory costs.

1. Introduction

Eliminating redundant computation is an important optimization supported by many language implementations. One important instance of this optimization class is memoization [11, 14, 2], a well-known dynamic technique that can be utilized to avoid performing

a function application by recording the arguments and results of previous calls. If a call is supplied an argument that has been previously cached, the execution of the function body can be elided, with the corresponding result immediately returned instead.

When functions perform effectful computations, leveraging memoization becomes significantly more challenging. Two calls to a function f that performs some stateful computation need not generate the same result if the contents of the state f uses to produce its result are different at the two call-sites.

Concurrency and communication introduce similar complications. If a thread calls a function f that communicates with functions invoked in other threads, then memo information recorded with f must include the outcome of these actions. If f is subsequently applied with a previously seen argument, and its communication actions at this call-site are the same as its effects at the original application, re-evaluation of the pure computation in f 's body can be avoided. Because of thread interleavings, synchronization, and non-determinism introduced by scheduling choices, making such decisions is non-trivial.

Nonetheless, we believe memoization can be an important component in a concurrent programming language runtime. Our belief is enforced by the widespread emergence of multicore and manycore platforms, and renewed interest in speculative and transactional abstractions to program these architectures. For instance, optimistic concurrency abstractions rely on efficient control and state restoration mechanisms. When a speculation fails because a previously available computation resource becomes unavailable, or when a transaction aborts due to a serializability violation [6] and must be retried [9], their effects must be undone. Failure represents wasted work, both in terms of the operations performed whose effects must now be erased, and in terms of overheads incurred to implement state restoration; these overheads include logging costs, read and write barriers, contention management, etc. One way to reduce this overhead is to avoid subsequent re-execution of those function calls previously executed by the failed computation whose results are unchanged. The key issue is understanding when utilizing memoized information is safe, given the possibility of concurrency, communication, and synchronization among threads.

In this paper, we consider the memoization problem for a higher-order concurrent language in which threads communicate through synchronous message-passing primitives (e.g. Concurrent ML [17]). A synchronization event acknowledges the existence of an external action performed by another thread willing to send or receive data. If such events occur within a function f whose applications are memoized, then avoiding re-execution at a call-site c is only possible if these actions are guaranteed to succeed at c . In other words, using memo information requires discovery of interleavings that satisfy the communication constraints imposed by a previous call. If we can identify a global state in which these con-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

straints are satisfied, the call to c can be avoided; if there exists no such state, then the call must be performed. Because finding such a state can be expensive (it may require an unbounded state space search), we consider a weaker notion of memoization: by recording the context in which a memoization constraint was generated, implementations can always choose to simply resume execution of the function at the program point associated with the constraint using the saved context. In other words, rather than requiring global execution to reach a state in which all constraints in a memoized application are satisfied, *partial memoization* gives implementations the freedom to discharge some fraction of these constraints, performing the rest of the application as normal. Although our description and formalization is developed in the context of message-based communication, the applicability of our solution naturally extends to shared-memory communication as well given the simple encoding of the latter in terms of the former [17].

Whenever a constraint built during memoization is discharged on a subsequent application, there is a side-effect on the global state that occurs. For example, consider a communication constraint associated with a memoized version of a function f that expects a thread T to receive data d on channel c . To use this information at a subsequent call, we must identify the existence of T , and having done so, must propagate d along c for T to consume. Thus, whenever a constraint is satisfied, an effect that reflects the action represented by that constraint is performed. We consider the set of constraints built during memoization as forming an ordered *log*, with each entry in the log representing a condition that must be satisfied to utilize the memoized version, and an effect that must be performed if the condition holds. The point of memoization for our purposes is thus to avoid performing the pure computations that execute between these effectful operations.

Besides providing a formal characterization of these ideas, we also present performance evaluation of our implementation. We consider the effect of memoization on a speculative message-based implementation of a concurrent data structure: when a speculative traversal of the structure fails because of intervening updates, memoization can help avoid complete re-traversal of the structure on subsequent queries. We also consider a benchmark based on a transaction-aware extension of Concurrent ML [17]. Our application is STMBench7 [7], a highly tunable benchmark for measuring transaction overheads, re-written to leverage CML synchronous communication. Our results indicate that memoization can lead to substantial reduction in transaction abort overheads, in some cases in excess of 40% improvement in execution time compared with an implementation that performs no memoization, with only modest increases in memory overhead (15% on average). To the best of our knowledge, this is the first attempt to formalize a memoization strategy for effectful higher-order concurrent languages, and to provide an empirical evaluation of its impact on improving performance for intensive multi-threaded workloads.

The paper is organized as follows. The programming model is defined in Section 2. Motivation for the problem is given in Section 3. The formalization of our approach is given in Sections 4 and Section 5. A detailed description of our implementation and results are given in Sections 6 and 7. We discuss previous work and provide conclusions in Section 8.

2. Programming Model

Our programming model is a simple synchronous message-passing dialect of ML similar to CML. Threads communicate using dynamically created channels through which they produce and consume values. Since communication is synchronous, a thread wishing to communicate on a channel that has no ready recipient must block

until one exists, and all communication on channels is ordered. As mentioned above, our formulation does not consider references, but there are no additional complications that ensue in order to handle them.

In this context, deciding whether a function application can be avoided based on previously recorded memo information depends upon the value of its arguments, its communication actions, channels it creates, threads it spawns, and the return value it produces. Thus, the memoized result of a call to a function f can be used at a subsequent call if (a) the argument given matches the argument previously supplied; (b) recipients for values sent by f on channels in an earlier memoized call are still available on those channels; (c) a value that was consumed by f on some channel in an earlier call is again ready to be sent by another thread; (d) a channel it created in an earlier call (at a given program point) has the same actions performed on it, and (e) threads created by f can be spawned with the same arguments supplied in the memoized version. Ordering constraints on all sends and receives performed by the procedure must also be enforced. A successful application of a memoized call yields a new state in which the effects captured within the constraint log have been performed; thus, the values sent by f are received by waiting recipients, senders on channels from which f expects to receive values propagate these values on those channels, and channels and threads that f is expected to create are created.

3. Motivation

As a motivating example, we consider how memoization can be profitably utilized in a concurrent message-passing red-black tree implementation. The data structure supports concurrent insertion, deletion, and traversal operations.

A node in the tree is a tuple containing the node’s color, an integer value, and links to its left and right children. Associated with every node is a thread that reads from an input channel, and outputs the node’s data on an output channel. Accessing and modifying a tree node’s data is thus accomplished through a communication protocol with the node’s corresponding channels. Abstractly, a read corresponds to a receive operation from a node’s output channel, and a write to a send on a node’s input channel.

```
datatype rbtree = Empty
  | N of cOut: node chan, cIn: node chan
  and node = Node of color: color, value: int,
    left:rbtree, right:rbtree
  | EmptyNode
fun node(color:color, value:int,
  left:rbtree, right:rbtree):rbtree =
  let val (cOut, cIn) = (chan(), chan())
      val n = Node{color,value,left,right}
      fun server(n) =
        let val _ = CML.send(cOut, n)
            val n = CML.recv(cIn)
        in server(n)
        end
  in spawn(fn () => server(n)); N{cOut, cIn}
  end
```

For example, the following procedure queries the tree to determine if a node containing its integer argument is present. It takes as input the number being searched, and the root of the tree from which to begin the traversal. The `recv` operation reads from the output channel of each node, and navigates the tree based on the result:

```

fun contains (n:int, t:rbtree):bool =
  let fun check(n, N{cOut, cIn}) =
      (case recv(cOut)
       of Empty => false
        | Node {color,value,left,right} =>
          (case Int.compare (value, n)
           of EQUAL => true
            | GREATER => check (n,left)
            | LESSER => check (n,right)))
      in check(n, t)
  end

```

Memoization can be leveraged to avoid redundant traversals of the tree. Consider the red/black tree shown in Fig. 1. Triangles represent abstracted portions of the tree, red nodes are unshaded, and black nodes are marked as bold circles. Suppose we memoize the call to `contains` which finds the node containing the value 79. Whenever a subsequent call to `contains` attempts to find the node containing 79, the traversal can directly use the result of the previous memoized call if both the structure of the tree along the path to the node, and the values of the nodes on the path remain the same. Both of these properties are reflected in the values transmitted by node processes.

The path is depicted by shading relevant nodes in gray in Fig. 1. More concretely, we can avoid the recomputation of the traversal if communication with node processes remains unchanged. Informally, memo information associated with a function f can be leveraged to avoid subsequent applications of f if communication actions performed by the memoized call can be satisfied in these later applications. Thus, to successfully leverage memo information for a call to `contains` with input 79, we would need to ensure a subsequent call of the function with the same argument would be able to receive the sequence of values: $[\dots, \text{Node}\{\text{value}=48, \text{color}=\text{red}\}, \text{Node}\{\text{value}=76, \text{color}=\text{black}\}, \text{Node}\{\text{value}=85, \text{color}=\text{red}\}, \text{Node}\{\text{value}=79, \text{color}=\text{black}\}, \dots]$ in that order during the traversal of tree by `check`. In Fig. 3 thread T1 can take advantage of memoization, while thread T2 subsequently recolors the node containing 85.

Because of the actions of T2, subsequent calls to `contains` with argument 79 cannot avail of the information recorded during evaluation of the initial memoized call. As a consequence of T2's update, a subsequent traversal by T1 would observe a change to the tree. Note however, that even though the immediate parent of 79 has changed in color, the path leading up to node 84 has not (see Fig. 2). By leveraging partial memoization on the earlier memoized call to `contains`, a traversal attempting to locate node 79 can avoid traversing this prefix, if not the entire path. Notice that if the node with value 85 was later recolored, and assuming structural equality of nodes is used to determine memoization feasibility, full memoization would once again be possible.

Now, consider what happens if T2 inserts a node that contains the value 87 into the original tree. Through a series of rotations, a new tree is produced (see Fig. 3). The original traversal by `contains` to locate 79 is depicted in gray. Partial memoization allows avoiding the traversal up to the node containing the value 85. Because of the alteration of the tree, however, no further savings are possible. To capture the new structure of the tree, we can adapt an incremental memoization strategy that resumes memoization of the traversal at this point. In other words, re-memoization of the call to `contains` with argument 79 can start from the point the previous attempt to utilize memo information failed. Subsequent calls to `contains` which search the tree for 79 can utilize this newly recorded information to elide their traversals.

The series of received values recorded for the new memoized version of `contains` is: $[\dots, \text{Node}\{\text{value}=48, \dots\},$

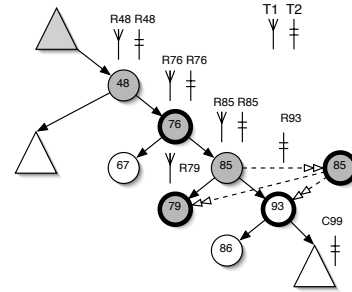


Figure 1. Memoization can be used to avoid redundant tree traversals. In this example, two threads traverse a red-black tree. Each node in the tree is represented as a process that outputs the current value of the node, and inputs new values. The shaded path illustrates memoization potential.

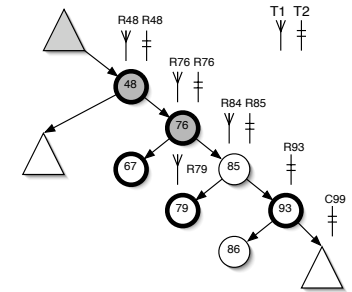


Figure 2. Even if the constraints stored in a memoized call cannot be fully satisfied at a subsequent call, it may be possible to discharge some fraction of these constraints, retaining some of the optimization benefits memoization affords.

$\text{Node}\{\text{value}=85, \dots, \{\text{value}=76, \dots\}, \text{Node}\{\text{value}=79, \dots\}, \dots]$ and is depicted graphically in Fig. 3.

As this example suggests, a key requirement for effectively utilizing memoized function applications is the ability to track communication (and other effectful) actions performed by previous instantiations. Provided that the global state would permit these same actions (or some subset thereof) to succeed if a function is re-executed with the same inputs, memoization can be employed to avoid re-computing applications, or to reduce the amount of the application that is executed. We note that although the example presented dealt with memoization of a function that operates over base integer values, our solution detailed in the following sections considers memoization in the context of *any* value that can appear on a channel, including closures, vectors, and datatype instances.

4. Semantics

Our semantics is defined in terms of a core call-by-value functional language with threading and communication primitives. For perspicuity, we first present a simple multi-threaded language with synchronous channel based communication. We then extend this core language with memoization primitives, and subsequently consider refinements of this semantics to support partial memoization.

In the following, we write $\bar{\alpha}$ to denote a sequence of zero or more elements, $\beta.\bar{\alpha}$ to denote sequence concatenation, and \emptyset to denote an empty sequence. Metavariables x and y range over variables, t ranges over thread identifiers, l ranges over channels, v ranges

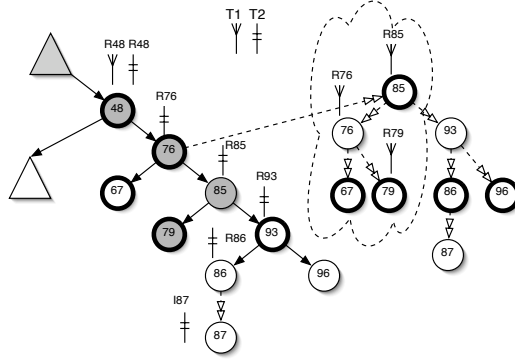


Figure 3. When a new node is inserted into the tree, previous memo information that records the path to node 79 can no longer be fully used. By rebuilding memo information from the point where the insertion takes place, subsequent calls can avoid the traversal to the node. In the figure, the tree transformation due to the insert is depicted utilizing dashed arrows. Dashed lines indicate structural changes to the tree that can be memoized to avoid (or reduce) subsequent traversals.

SYNTAX:

$$\begin{aligned}
P &::= P \parallel P \mid \tau[e] \\
e \in Exp &::= v \mid e(e) \mid \text{spawn}(e) \\
&\quad \mid \text{mkCh}() \mid \text{send}(e, e) \mid \text{recv}(e) \\
v \in Val &::= \text{unit} \mid \lambda x. e \mid 1
\end{aligned}$$

PROGRAM STATES:

$$\begin{aligned}
P &\in Process \\
\tau &\in Tid \\
x, y &\in Var \\
1 &\in Channel \\
\alpha, \beta &\in Tag = \{App, Ch, Spn, Com\}
\end{aligned}$$

EVALUATION CONTEXTS:

$$\begin{aligned}
E &::= [] \mid E(e) \mid v(E) \mid \text{spawn}(E) \mid \\
&\quad \text{send}(E, e) \mid \text{send}(1, E) \mid \text{recv}(E)
\end{aligned}$$

(APP)

$$\frac{}{P \parallel \tau[E[\lambda x. e(v)]] \xrightarrow{APP} P \parallel \tau[E[e[v/x]]]}$$

(SPAWN)

$$\frac{\tau' \text{ fresh}}{P \parallel \tau[E[\text{spawn}(\lambda x. e)]] \xrightarrow{Spn} P \parallel \tau[E[\text{unit}]] \parallel \tau'[e[\text{unit}/x]]}$$

(CHANNEL)

$$\frac{1 \text{ fresh}}{P \parallel \tau[E[\text{mkCh}()]] \xrightarrow{Ch} P \parallel \tau[E[1]]}$$

(COMM)

$$\frac{\tau' \text{ fresh}}{P = P' \parallel \tau[E[\text{send}(1, v)]] \parallel \tau'[E'[\text{recv}(1)]]} \xrightarrow{Com} P' \parallel \tau[E[\text{unit}]] \parallel \tau'[E'[v]]$$

Figure 4. A concurrent language with synchronous communication.

over values, and α, β denote tags that label individual actions in a program's execution. We use P to denote a program state comprised of a collection of threads, E for evaluation contexts, and e for expressions.

Our communication model is a message-passing system with synchronous send and receive operations. We do not impose a strict ordering of communications on channels; communication actions on the same channel by different threads are paired non-deterministically. To model asynchronous sends, we simply spawn a thread to perform the send¹. Spawning an expression (that evaluates to a thunk) creates a new thread in which the application of the thunk is performed.

4.1 Language

The syntax and semantics of the language are given in Fig. 4. Expressions are either variables, locations that represent channels, λ -abstractions, function applications, thread creation operations, or communication actions that send and receive messages on chan-

nels. We do not consider references in this core language as they can be modeled in terms of operations on channels [17].

A thread context $\tau[E[e]]$ denotes an expression e available for execution by a thread with identifier τ within context E . Local reductions within a thread are specified by an auxiliary relation, $e \rightarrow e'$, that evaluates expression e within some thread to a new expression e' .

Evaluation is specified via a relation ($\xrightarrow{\alpha}$) that maps a program state (P) to another program state. An evaluation step is marked with a tag that indicates the action performed by that step. As shorthand, we write $P \xrightarrow{\bar{\alpha}} P'$ to represent the sequence of actions $\bar{\alpha}$ that transforms P to P' .

Channel creation (rule CHANNEL) results in the creation of a new location that acts as a container for message transmission and receipt, and application (rule APP) substitutes the argument value for free occurrences of the parameter in the body of the abstraction. A spawn action (rule SPAWN), given an expression e that evaluates to a thunk changes the global state to include a new thread in which the thunk is applied. A communication event (rule COMM) synchronously pairs a sender attempting to transmit a value along a

¹ Asynchronous receives are not feasible without a mailbox abstraction [17].

specific channel in one thread with a receiver waiting on the same channel in another thread.

4.2 Partial Memoization

The core language presented above provides no facilities for memoization of the functions it executes. To support memoization, we must record, in addition to argument and return values, synchronous communication actions, thread spawns, channel creation etc. as part of the memoized state. These actions define a log of constraints that must be satisfied at subsequent applications of a memoized function, and whose associated effects must be discharged if the constraint is satisfied. To record constraints, we augment our semantics to include a *memo store*, a map that given a function identifier and an argument value, returns the set of constraints and result value that was previously recorded for a call to that function with that argument. If the set of constraints returned by the memo store is satisfied in the current state (and their effects performed), then the return value can be used and the application elided.

The definition of the language augmented with memoization support is given in Fig. 5. We now define evaluation using a new relation (\Longrightarrow) that maps a program state (P) and a memo store (σ) to a new program state and a new memo store. As a useful convenience, we often write $P, \sigma \xrightarrow{\bar{\alpha}} P', \sigma'$ to denote the evaluation $P, \sigma \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} P', \sigma'$ where $\bar{\alpha} = \alpha_1 \dots \alpha_n$.

A thread state is augmented to hold two additional structures. The first ($\bar{\theta}$) records the sequence of constraints that are built during the evaluation of an application being memoized; the second (\bar{C}) holds the sequence of constraints that must be discharged at an application of a previously memoized function.

Constraints built during a memoized function application define actions that must be satisfied at subsequent call-sites in order to avoid complete re-evaluation of the function body. For a communication action, a constraint records the location being operated upon, the value sent or received, the action performed (R for receive and S for send), and the continuation immediately prior to the action being performed; the application resumes evaluation from this point if the corresponding constraint could not be discharged. For a spawn operation, the constraint records the action (Sp) and the expression being spawned. For a channel creation operation (Ch), the constraint records the location of the channel. Returns are also modeled as constraints ((Rt, v) where v is the return value of the application being memoized).

If function f calls function g , then actions performed by g must be satisfiable in any call that attempts to leverage the memoized version of f . Consider the following program fragment:

```
let fun f(...) = ...
  let fun g(...) = ... send(c,v) ...
  in ... end
in ... g(...) ... end
```

If we encounter an application of f after it has been memoized, then g 's communication action (i.e., the send of v on c) must be satisfiable at the point of the application to avoid performing the call. We therefore associate a *call stack* of constraints ($\bar{\theta}$) with each thread that defines the constraints seen thus far, requiring the constraints computed for an inner application to be satisfiable for any memoization of an outer one. The propagation of constraints to the memo states of all active calls is given by the operation \succ shown in Fig. 5.

Channels created within a memoized function must be recorded in the constraint sequence for that function (rule CHANNEL). Consider a function that creates a channel and subsequently initiates communication on that channel. If a call to this function was memoized, later applications that attempt to avail of memo information

must still ensure that the generative effect of creating the channel is not omitted. Function evaluation now associates a label with function evaluation that is used to index the memo store (rule FUN).

If a new thread is spawned within a memoized application, a spawn constraint is added to the memo state, and a new global state is created that starts memoization of the actions performed by the newly spawned thread (rule SPAWN). A communication action performed by two functions currently being memoized are also appropriately recorded in the corresponding memo state of the threads that are executing these functions. (rule COMM). When a memoized application completes, its constraints are recorded in the memo store (rule RET).

When a function f is applied to argument v , and there exists no previous invocation of f to v recorded in the memo store, the function's effects are tracked and recorded (rule APP). A syntactic wrapper B (for *build memo*) is used to identify such functions. Until an application of a function being memoized is complete, the constraints induced by its evaluation are not immediately added to the memo store. Instead, they are maintained as part of the constraint sequence ($\bar{\theta}$) associated with the thread in which the application occurs.

Consider an application of function f to value v that has been memoized. Since subsequent calls to f with v may not be able to discharge all constraints, we need to record the program points for all communication actions within f that represent potential resumption points from which normal evaluation of the function body proceeds; these continuations are recorded as part of any constraint that can fail (communication actions, and return constraints as described below). But, since the calling contexts at these other call-sites are different than the original, we must be careful to not include those outer contexts as part of the saved continuation. Thus, the contexts recorded as part of the saved constraint during memoization only define the continuation of the action up to the return point of the function; hence, the evaluation specified in the second antecedent in rule APP is given without the evaluation context representing the caller's continuation.

Note that all the consequents in all rules in this figure assume an empty constraint sequence (\emptyset). The constraints built as a result of evaluating these rules are discharged by the rules shown in Fig. 7. Thus, at any given point in its execution, a thread is either building up memo constraints within an application for subsequent calls to utilize, or attempting to discharge these constraints for applications indexed in the memo store.

The most interesting rule is the one that deals with determining how much of an application of a memoized function can be elided (rule MEMO APP). If an application of function f with argument v has been recorded in the memo store, then the application can be potentially avoided; if not, its evaluation is memoized by rule APP. If a memoized call is applied, we must examine the set of associated constraints that can be discharged. To do so, we employ an auxiliary relation \mathfrak{J} defined in Fig. 6. Abstractly, \mathfrak{J} checks the global state to see if all communication, channel creation, and spawn creation constraints, which represent the effectful actions in our language, can be satisfied, and returns a *schedule* (a sequence of action tags $\bar{\alpha}$ whose evaluation is dictated by the rules given in Fig. 7 (as seen by the structure of the last antecedent in rule MEMO APP).

The resulting global state P' reflects the effects of discharging these constraints. The thread performing the memoized call executes in a new state in which it no longer presumes to be discharging constraints (thus, the presence of \emptyset , the empty sequence of constraints, in the thread state on the right-hand side of the evaluation rule in the consequent), and in which the evaluation of the call is the expression associated with the first *failed* constraint in \mathfrak{J} . As we describe below, there is always at least one such constraint, namely

SYNTAX:

$$\begin{aligned}
P &::= P \parallel P \mid \langle \bar{\theta}, \bar{C}, \tau[e] \rangle \\
e \in Exp &::= \dots \mid B(v, e) \mid U(e) \\
v \in Val &::= \mathbf{unit} \mid \lambda_{\delta} x. e \mid 1
\end{aligned}$$

EVALUATION CONTEXTS:

$$E ::= \dots \mid B(v, E)$$

PROGRAM STATES:

$$\begin{aligned}
\delta &\in \mathit{MemoId} \\
c &\in \mathit{FailableConstraint} = (\{R, S\} \times \mathit{Loc} \times \mathit{Val}) + \mathit{Rt} \\
C &\in \mathit{Constraint} = (\mathit{FailableConstraint} \times \mathit{Exp}) + (\mathit{Sp} \times \mathit{Exp}) + (\mathit{Ch} \times \mathit{Location}) \\
\sigma &\in \mathit{MemoStore} = \mathit{MemoId} \times \mathit{Val} \rightarrow \mathit{Constraint}^* \\
\theta &\in \mathit{MemoState} = \mathit{MemoId} \times \mathit{Constraint}^* \\
\alpha, \beta &\in \mathit{Tag} = \{\mathit{Ch}, \mathit{Spn}, \mathit{Com}, \mathit{Fun}, \mathit{App}, \mathit{Ret}, \mathit{MCh}, \mathit{MSp}, \mathit{PMem}\}
\end{aligned}$$

CONSTRAINT ADDITION:

$$\frac{\bar{\theta}' = \{(\delta, \bar{C}.C) \mid (\delta, \bar{C}) \in \bar{\theta}\}}{\bar{\theta}, C \succ \bar{\theta}'}$$

(CHANNEL)

$$\frac{\bar{\theta}, (\mathit{Ch}, 1) \succ \bar{\theta}'}{P \parallel \langle \bar{\theta}, \theta, \tau[E[\mathit{mkCh}()]] \rangle, \sigma \xrightarrow{Ch} P \parallel \langle \bar{\theta}', \theta, \tau[E[1]] \rangle, \sigma}$$

(SPAWN)

$$\frac{\begin{array}{l} \tau' \text{ fresh} \quad \bar{\theta}, (\mathit{Sp}, \lambda_{\delta} x. e(\mathbf{unit})) \succ \bar{\theta}' \\ t_k = \langle \bar{\theta}', \theta, \tau[E[\mathbf{unit}]] \rangle \\ t_s = \langle \theta, \theta, \tau'[\lambda_{\delta} x. e(\mathbf{unit})] \rangle \end{array}}{P \parallel \langle \bar{\theta}, \theta, \tau[E[\mathit{spawn}(\lambda_{\delta} x. e)]] \rangle, \sigma \xrightarrow{Spn} P \parallel t_k \parallel t_s, \sigma}$$

(RET)

$$\frac{\theta = (\delta, \bar{C})}{P \parallel \langle \theta, \theta, \tau[E[B(v, v')]] \rangle, \sigma \xrightarrow{Ret} P \parallel \langle \bar{\theta}, \theta, \tau[E[v']] \rangle, \sigma' \mid (\delta, v) \mapsto \bar{C}.(\mathit{Rt}, v')}$$

(MEMO APP)

$$\frac{\begin{array}{l} (\delta, v) \in \mathit{Dom}(\sigma) \quad \sigma(\delta, v) = \bar{C} \\ \mathfrak{S}(\bar{C}, \theta, P) = C.\bar{C}', \bar{\alpha} \quad C = (c, e') \end{array}}{P \parallel \langle \bar{\theta}, \bar{C}, \tau[E[\lambda_{\delta} x. e(v)]] \rangle, \sigma \xrightarrow{\bar{\alpha}} P' \parallel \langle \bar{\theta}, C.\bar{C}', \tau[E[\lambda_{\delta} x. e(v)]] \rangle, \sigma' \\ P \parallel \langle \bar{\theta}, \theta, \tau[E[\lambda_{\delta} x. e(v)]] \rangle, \sigma \xrightarrow{\bar{\alpha}.PMem} P' \parallel \langle \bar{\theta}, \theta, \tau[E[e']] \rangle, \sigma'}$$

(FUN)

$$\frac{\delta \text{ fresh}}{P \parallel \langle \bar{\theta}, \theta, \tau[E[\lambda x. e]] \rangle, \sigma \xrightarrow{Fun} P \parallel \langle \bar{\theta}, \theta, \tau[E[\lambda_{\delta} x. e]] \rangle, \sigma}$$

(COMM)

$$\frac{\begin{array}{l} P = P' \parallel \langle \bar{\theta}, \theta, \tau[E[\mathit{send}(1, v)]] \rangle \parallel \langle \bar{\theta}', \theta, \tau'[E[\mathit{recv}(1)]] \rangle \\ \bar{\theta}, ((S, 1, v), E[\mathit{send}(1, v)]) \succ \bar{\theta}'' \quad \bar{\theta}', ((R, 1, v), E'[\mathit{recv}(1)]) \succ \bar{\theta}''' \\ t_s = \langle \bar{\theta}'', \theta, \tau[E[\mathbf{unit}]] \rangle \quad t_r = \langle \bar{\theta}''', \theta, \tau'[E[v']] \rangle \end{array}}{P, \sigma \xrightarrow{Com} P' \parallel t_s \parallel t_r, \sigma}$$

(APP)

$$\frac{(\delta, v) \notin \mathit{Dom}(\sigma)}{P \parallel \langle (\delta, \theta). \bar{\theta}, \theta, \tau[B(v, e[v/x])] \rangle, \sigma \xrightarrow{\bar{\alpha}'} P' \parallel \langle \bar{\theta}', \theta, \tau[B(v, v')] \rangle, \sigma' \\ P \parallel \langle \bar{\theta}, \theta, \tau[E[\lambda_{\delta} x. e(v)]] \rangle, \sigma \xrightarrow{App.\bar{\alpha}'} P' \parallel \langle \bar{\theta}', \theta, \tau[E[B(v, v')]] \rangle, \sigma'}$$

Figure 5. A concurrent language supporting memoization of synchronous communication and dynamic thread creation.

Rt , the return constraint that holds the return value of the memoized call.

4.3 Constraint Matching

The constraint matching rules shown in Fig. 7 each check the global state to determine if the conditions expected by the constraint hold, and if so, perform a side-effect that expresses the expected effectful action.

Thus, a spawn constraint (rule MSPAWN) is always satisfied, and leads to the creation of a new thread of control. Observe that the application evaluated by the new thread is now a candidate for memoization if the thunk was previously applied and its result is recorded in the memo store.

A channel constraint of the form $(\mathit{Ch}, 1)$ (rule MCH) creates a new channel location $1'$, and replaces all occurrences of 1 found in the

remaining constraint sequence for this thread with $1'$; the channel location may be embedded within send and receive constraints, either as the target of the operation, or as the argument value being sent or received. Thus, discharging a channel constraint ensures that the effect of creating a *new* channel performed within an earlier memoized call is preserved on subsequent applications. The renaming operation ensures that later send and receive constraints refer to the new channel location. Both channel creation and thread creation actions never fail – they modify the global state with a new thread and channel, resp. but impose no pre-conditions on the state in order for these actions to succeed.

On the other hand, there are two communication constraint matching rules (MCom), both of which may indeed fail. If the current constraint expects to receive value v on channel 1 , and there exists a thread able to send v on 1 , evaluation proceeds to a state

```

let val (c1,c2) = (mkCh(),mkCh())
  fun f () = (send(c1,v1); ...; recv(c2))
  fun g () = (recv(c1); ...; recv(c2); g())
  fun h () = (...
    send(c2,v2);
    send(c2,v3);
    h());
  fun i () = (recv(c2); i())
in spawn(g); spawn(h); spawn(i);
  f(); ...;
  send(c2, v3); ...;
  f()
end

```

Figure 8. Determining if an application can fully leverage memo information may require examining an arbitrary number of possible thread interleavings.

in which the communication succeeds (the receiving thread now evaluates in a context in which the receipt of the value has occurred), and the constraint is removed from the set of constraints that need to be matched (rule MRECV). Note also that the sender records the fact that a communication with a matching receive took place in the thread’s memo state, and the receiver does likewise. Any memoization of the sender must consider the receive action that synchronized with the send, and the application in which the memoized call is being examined must record the successful discharge of the receive action. In this way, the semantics permits consideration of multiple nested memoization actions.

If the current constraint expects to send a value v on channel l , and there exists a thread waiting on l , the constraint is also satisfied (rule MSEND). A send operation can match with any waiting receive action on that channel. The semantics of synchronous communication allows us the freedom to consider pairings of sends with receives other than the one it communicated with in the original memoized execution. This is because a receive action places no restriction on either the value it reads, or the specific sender that provides the value. If these conditions do not hold, the constraint fails.

The function \mathfrak{S} leverages the evaluation rules defined in Fig. 7 by examining the global state and constructing a schedule (if one exists) that matches all constraints, except for the return constraint. \mathfrak{S} takes a constraint set (\bar{C}) , a schedule $(\bar{\alpha})$, and a program state and returns a set of unmatchable constraints (\bar{C}') and a new schedule $(\bar{\alpha}')$. Send and receive constraints are matched with threads blocked in the global program state on the opposite communication action. Once a thread has been matched with a constraint it is no longer a candidate for future communication since its communication action is *consumed* by the constraint. This guarantees that the candidate function will communicate at most once with each thread in the global state. Although a thread may in fact be able to communicate more than once with the candidate function, determining this requires arbitrary look ahead and is infeasible in practice. The sequence of actions yielded by \mathfrak{S} define the action tags that relate the sequence of steps undertaken from the current state to the new global state in which the call is fully elided, and the memoized return value directly supplied (see rule MEMO APP).

Observe that there is no evaluation rule for the R_t constraint that can consume it. This constraint contains the return value of the memoized function (see rule RET). If all other constraints have been satisfied, it is this return value that replaces the call in the current context (see the consequent of rule MEMO APP).

4.4 Example

The program fragment shown in Fig. 8 applies functions f , g , h and i . The calls to g , h , and i are evaluated within separate threads of control, while the applications of f takes place in the original thread. These different threads communicate with one other over shared channels $c1$ and $c2$.

To determine whether the second call to f can be elided we must examine the constraints that would be added to the thread state of the threads in which these functions are applied. First, spawn constraints would be added to the main thread for the threads executing g , h , and i . Second, a send constraint followed by a receive constraint, modeling the exchange of values $v1$ and either $v2$ or $v3$ on channels $c1$ and $c2$ would be included as well. For the sake of discussion, assume that the send of $v2$ by h was consumed by g and the send of $v3$ was paired with the receive in f when $f()$ was originally executed.

Consider the memoizability constraints built during the first call to $f()$. The send constraint on f ’s application can be satisfied by matching it with the corresponding receive constraint associated with the application of g ; observe $g()$ loops forever, consuming values on channels $c1$ and $c2$. The receive constraint associated with f can be discharged if g receives the first send by h , and f receives the second. A schedule that orders the execution of f and g in this way, and additionally pairs i with a send operation on $c2$ in the `let`-body would allow the second call to f to fully leverage the memo information recorded in the first. Doing so would enable eliding the pure computation in f (abstracted by \dots in its definition, performing only the effects defined by the communication actions (i.e., the send of $v1$ on $c1$, and the receipt of $v3$ on $c2$).

4.5 Issues

As this example illustrates, utilizing memo information completely may require forcing a schedule that pairs communication actions in a specific way, making a solution that requires *all* constraints to be satisfied infeasible in practice. Hence, rule MEMO APP allows evaluation to continue within an application that has already been memoized once a constraint cannot be matched. As a result, if during the second call f , i indeed received $v3$ from h , the constraint associated with the `recv` operation in f would not be satisfied, and the rules would obligate the call to block on the `recv`, waiting for h or the main body to send a value on $c2$.

Nonetheless, the semantics as currently defined does have limitations. For example, the function \mathfrak{S} does not examine future actions of threads and thus can only match a constraint with a thread if that thread is able to match in its current state. Hence, the rules do not allow leveraging memoization information for function calls involved in a producer/consumer relation. Consider the simple example given in Fig. 9. The second application of f can take advantage of memoized information only for the first send on channel c . This is because the global state in which constraints are checked only has the first `recv` made by g blocked on the channel. The second `recv` only occurs if the first is successfully paired with a corresponding send. Although in this simple example the second `recv` is guaranteed to occur, consider if g contained a branch which determined if g consumed a second value from the channel c . In general, constraints can only be matched against the current communication action of a thread.

Secondly, exploiting memoization may lead to starvation since subsequent applications of the memoized call will be matched based on the constraints supplied by the initial call. Consider the example shown in Fig. 10. If the initial application of f pairs with the send performed by g , subsequent calls to f that use this memo-

$$\begin{aligned}
\mathfrak{S}(((S, 1, v), e), \bar{C}, \bar{\alpha}, P \parallel \langle \bar{\theta}', \emptyset, \tau'[E'[\text{recv}(1)]] \rangle) &= \mathfrak{S}(\bar{C}, \bar{\alpha}, MCom, P) \\
\mathfrak{S}(((R, 1, v), e), \bar{C}, \bar{\alpha}, P \parallel \langle \bar{\theta}, \emptyset, \tau[E[\text{send}(1, v)]] \rangle) &= \mathfrak{S}(\bar{C}, \bar{\alpha}, MCom, P) \\
\mathfrak{S}((Rt, v), \bar{C}, \bar{\alpha}, P) &= (Rt, v), \bar{C}, \bar{\alpha} \\
\mathfrak{S}((Ch, 1), \bar{C}, \bar{\alpha}, P) &= \mathfrak{S}(\bar{C}, \bar{\alpha}, MCh, P) \\
\mathfrak{S}((Sp, e), \bar{C}, \bar{\alpha}, P) &= \mathfrak{S}(\bar{C}, \bar{\alpha}, MSp, P) \\
\mathfrak{S}(\bar{C}, \bar{\alpha}, P) &= \bar{C}, \bar{\alpha} \text{ otherwise}
\end{aligned}$$

Figure 6. The function \mathfrak{S} yields a schedule $(\bar{\alpha})$ in the form of a sequence of action tags for the maximum number of constraints matchable in the global state (P) .

(MCH)

$$\frac{C = (Ch, 1) \quad 1' \text{ fresh} \quad \bar{C}' = \bar{C}[1'/1] \quad \bar{\theta}, C \succ \bar{\theta}'}{P \parallel \langle \bar{\theta}, C, \bar{C}, \tau[e] \rangle, \sigma \xrightarrow{MCh} P \parallel \langle \bar{\theta}', \bar{C}', \tau[e] \rangle, \sigma}$$

(MSPAWN)

$$\frac{C = (Sp, e') \quad \tau' \text{ fresh} \quad \bar{\theta}, C \succ \bar{\theta}'}{P \parallel \langle \bar{\theta}, C, \bar{C}, \tau[E[e]] \rangle, \sigma \xrightarrow{MSp} P \parallel \langle \bar{\theta}', \bar{C}, \tau[E[e]] \rangle \parallel \langle \emptyset, \emptyset, \tau'[e'] \rangle, \sigma}$$

(MRECV)

$$\frac{C = ((R, 1, v), -) \quad t_s = \langle \bar{\theta}, \emptyset, \tau[E[\text{send}(1, v)]] \rangle \quad t_r = \langle \bar{\theta}', C, \bar{C}, \tau'[e'] \rangle \quad \bar{\theta}', C \succ \bar{\theta}''' \quad \bar{\theta}, ((S, 1, v), E[\text{send}(1, v)]) \succ \bar{\theta}'' \quad t_{s'} = \langle \bar{\theta}'', \emptyset, \tau[E[\text{unit}]] \rangle \quad t_{r'} = \langle \bar{\theta}''', C, \tau'[e'] \rangle}{P \parallel t_s \parallel t_r, \sigma \xrightarrow{MCom} P \parallel t_{s'} \parallel t_{r'}, \sigma}$$

(MSEND)

$$\frac{C = ((S, 1, v), -) \quad t_s = \langle \bar{\theta}', C, \bar{C}, \tau'[e'] \rangle \quad t_r = \langle \bar{\theta}, \emptyset, \tau[E[\text{recv}(1)]] \rangle \quad \bar{\theta}', C \succ \bar{\theta}''' \quad \bar{\theta}, ((R, 1, v), E[\text{recv}(1)]) \succ \bar{\theta}'' \quad t_{s'} = \langle \bar{\theta}''', C, \tau'[e'] \rangle \quad t_{r'} = \langle \bar{\theta}'', \emptyset, \tau[E[v]] \rangle}{P \parallel t_s \parallel t_r, \sigma \xrightarrow{MCom} P \parallel t_{s'} \parallel t_{r'}, \sigma}$$

Figure 7. Constraint matching is defined by four rules. Communication constraints are matched with threads performing the opposite communication action of the constraint.

```

let fun f() = (send(c,1); send(c,2))
      fun g() = (recv(c); recv(c))
in spawn(g); f(); ...; spawn(g); f()
end

```

Figure 9. The second application of f can only be partially memoized up to the second send since only the first receive made by g is blocked in the global state.

```

let fun f() = recv(c)
      fun g() = send(c,1); g()
      fun h() = send(c,2); h()
in spawn(g); spawn(h); f(); ...; f()
end

```

Figure 10. Memoization of the function f can lead to the starvation of either of g or h depending on which value the original application of f consumed from channel c .

ized version will also pair with g since h produces different values. This leads to starvation of h . Although this behavior is certainly legal, one might reasonably expect a scheduler to interleave the sends of g and h .

4.6 Schedule Aware Partial Memoization

To address the limitations highlighted above, we consider a new definition of partial memoization (see Fig. 11) that permits constraints to be discharged *lazily*, based on subsequent actions of threads. The rule behaves roughly as before, but instead of installing the continuation stored at the first failed constraint, the rule

saves the remaining unsatisfied constraints for later use. To prevent further evaluation of the candidate, the application is tagged with a U wrapper (for *use memo*) to identify it as a potential beneficiary of previously recorded memo information. We define a new rule to resume memoization (rule RESUME MEMO). If a function is tagged with a U wrapper and has a set of constraints waiting to be matched, we can once again apply our \mathfrak{S} function.

We also introduce a rule to allow the completion of memo information use (rule END MEMO). The rule can be applied non-deterministically and installs the continuation of the first currently unsatisfied constraint; no further constraints are subsequently examined. Thus, the thread performing this call will resume execution from the saved program point as before, but the point at which this resumption happens may occur well after the point at which the call was made. Finally, we extend our evaluation rules to allow constraints to be matched against other constraints. The global state may contain threads that have matching send and receive constraints (rule MCOM). Thus, we may encounter multiple applications whose arguments have been memoized in the course of attempting to discharge memoization constraints. Specifically, there may exist two threads each performing an application of a previously memoized function whose memo states define matching send and receive constraints. In this case, the constraints on both sender and receiver can be safely discharged. This allows calls which attempt to use previously memoized constraints to match against constraints extant in other calls that also attempt to exploit memoized state.

(PARTIAL MEMO)

$$\frac{\begin{array}{l} (\delta, \mathbf{v}) \in \text{Dom}(\sigma) \quad \sigma(\delta, \mathbf{v}) = \bar{C} \\ \mathfrak{S}(\bar{C}, \emptyset, P) = C.\bar{C}', \bar{\alpha} \quad C = (c, e') \\ P \parallel \langle \bar{\theta}, \bar{C}, \tau[E[\lambda_{\delta} x.e(\mathbf{v})]] \rangle, \sigma \xrightarrow{\bar{\alpha}} P' \parallel \langle \bar{\theta}, C.\bar{C}', \tau[E[\lambda_{\delta} x.e(\mathbf{v})]] \rangle, \sigma' \\ P \parallel \langle \bar{\theta}, \emptyset, \tau[E[\lambda_{\delta} x.e(\mathbf{v})]] \rangle, \sigma \xrightarrow{\bar{\alpha}} P' \parallel \langle \bar{\theta}, C.\bar{C}', \tau[E[U(\lambda_{\delta} x.e(\mathbf{v}))]] \rangle, \sigma' \end{array}}{}$$

(MCOM)

$$\frac{\begin{array}{l} C = ((S, \mathbf{1}, \mathbf{v}), -) \quad C' = ((R, \mathbf{1}, \mathbf{v}), -) \\ t_s = \langle \bar{\theta}, C.\bar{C}, \tau[e] \rangle \quad t_r = \langle \bar{\theta}', C'.\bar{C}', \tau'[e'] \rangle \\ \bar{\theta}, C \succ \bar{\theta}' \quad \bar{\theta}', C' \succ \bar{\theta}'' \\ t_{s'} = \langle \bar{\theta}'', \bar{C}, \tau[e] \rangle \quad t_{r'} = \langle \bar{\theta}''', \bar{C}', \tau'[e'] \rangle \end{array}}{P \parallel t_s \parallel t_r, \sigma \xrightarrow{MCom} P \parallel t_{s'} \parallel t_{r'}, \sigma}$$

(RESUME MEMO)

$$\frac{\begin{array}{l} (\delta, \mathbf{v}) \in \text{Dom}(\sigma) \quad \sigma(\delta, \mathbf{v}) = \bar{C} \\ \mathfrak{S}(\bar{C}, \emptyset, P) = C.\bar{C}', \bar{\alpha} \quad C = (c, e') \\ P \parallel \langle \bar{\theta}, \bar{C}, \tau[E[\lambda_{\delta} x.e(\mathbf{v})]] \rangle, \sigma \xrightarrow{\bar{\alpha}} P' \parallel \langle \bar{\theta}, C.\bar{C}', \tau[E[\lambda_{\delta} x.e(\mathbf{v})]] \rangle, \sigma' \\ P \parallel \langle \bar{\theta}, \bar{C}, \tau[E[U(\lambda_{\delta} x.e(\mathbf{v}))]] \rangle, \sigma \xrightarrow{\bar{\alpha}} P' \parallel \langle \bar{\theta}, C.\bar{C}', \tau[E[U(\lambda_{\delta} x.e(\mathbf{v}))]] \rangle, \sigma' \end{array}}{}$$

(END MEMO)

$$\frac{C = (c, e)}{P \parallel \langle \bar{\theta}, C.\bar{C}, \tau[E[U(\lambda_{\delta} x.e(\mathbf{v}))]] \rangle, \sigma \xrightarrow{PMem} P \parallel \langle \bar{\theta}, \emptyset, \tau[E[e]] \rangle, \sigma}$$

Figure 11. Schedule Aware Partial Memoization.

4.7 Incremental Partial Memoization

Consider what occurs when not all constraints in a previously memoized call can be discharged. In this case we begin re-evaluation of the function from the stored (partial) continuation of the first failed constraint. Constraint matching could have failed for a number of reasons. In the simple case, evaluation was simply unlucky and we choose non-deterministically to fail. However, constraint failure could have occurred because the constraint can no longer be satisfied under any evaluation; consider the example described in Fig. 3. A rotation in the red-black tree changes the underlying structure of the tree and thus the communication pattern. Any memo information stored for a call to `contains` whose path traversed the rotation will fail its constraints at the first node affected by the rotation. Our definition of partial memoization for a call to function `f` is inextricably tied to the first set of constraints generated by the first application of the function `f` with a specific input. Observe, we can begin rebuilding constraints whenever a constraint fails during a call which is leveraging memo information. In our red black tree example, memoization would begin a new at the first node affected by a rotation.

To provide further expressivity, we introduce two new rules in Fig. 12. The first (rule REMEMO) converts a call for which we are discharging constraints to one in which we are *building* constraints. A new memo state (θ) is created that contains the constraints that have been successfully matched thus far. The (partial) continuation from the first non-matching (presumably failed) constraint is then installed within a syntactic build memo wrapper (`B`). The new thread state has θ' on the top of the memo state stack since new constraints can be now subsequently added. The REBUILD rule evaluates the installed continuation to yield a value \mathbf{v}' ; it allows memoization constraints to be built from *any* arbitrary expression.

5. Soundness

We define schedule filtering as the extraction of all actions performed by a given thread τ from a schedule $\bar{\alpha}$. The extracted actions of the given thread are a subsequence of the original schedule.

Definition[Schedule Filtering]

$$\bar{\alpha} \vdash_{\tau} \bar{\beta}$$

then

$$\bar{\alpha} = \alpha_1 \dots \alpha_n \quad \bar{\beta} \succeq \bar{\alpha}$$

and

$$P_1, \sigma_1 \xrightarrow{\alpha_0} \dots \xrightarrow{\alpha_n} P_n, \sigma_n \\ \forall \alpha \in \bar{\beta}$$

$$\exists P \parallel \langle \bar{\theta}, \bar{C}, \tau[E[e]] \rangle, \sigma_i \xrightarrow{\alpha_i} P' \parallel \langle \bar{\theta}', \bar{C}', \tau[E[e']] \rangle, \sigma_{i+1}$$

such that

$$P_i = P \parallel \langle \bar{\theta}, \bar{C}, \tau[E[e]] \rangle$$

and

$$P_{i+1} = P' \parallel \langle \bar{\theta}', \bar{C}', \tau[E[e']] \rangle$$

□

We define the concept of constraint capture in a subset of the language which does not support the use of memoized functions. For a given application its constraint set includes all actions which effect the global state in the order the function executed those actions.

Lemma[Constraint Capture (no Memo)] If

$$P \parallel \langle \phi, \emptyset, \tau[E[\lambda_{\delta} x.e(\mathbf{v})]] \rangle, \sigma \xrightarrow{App, \bar{\alpha}} P' \parallel \langle (\delta, \bar{C}), \emptyset, \tau[E[B(\mathbf{v}, \mathbf{v}')]] \rangle, \sigma'$$

and

$$\{PMem, App\} \notin \bar{\alpha}' \quad \bar{\alpha}' \vdash_{\tau} \bar{\alpha}''$$

then

$$\forall \{Ch, Sp, Com\} \in \bar{\alpha}''$$

$$\exists \{(Ch, \mathbf{1}), (Sp, e), ((S, \mathbf{1}, \mathbf{v}), e), ((R, \mathbf{1}, \mathbf{v}), e)\} \in \bar{C}$$

□

The proof is by induction on the length of $\bar{\alpha}''$. The base case is a sequence of length one. We examine the three potential cases $\{Com, Ch, Sp\}$.

`Com` - By definition of rule (Comm): adds a $(S, \mathbf{1}, \mathbf{v}), e$ constraint in the case of a send and a $(R, \mathbf{1}, \mathbf{v}), e$ receive.

`Ch` - By definition of rule (Channel): adds a $(Ch, \mathbf{1})$ constraint.

`Sp` - By definition of rule (Spawn): adds a (Sp, e) constraint.

We assume the theorem holds for sequences of length n and show that it holds for sequences of length $n + 1$. If the $n + 1$ step is $\{Com, Ch, Sp\}$ then we apply the same argument outlined in the base case. The rest holds by induction.

Lemma[\mathfrak{S} Safety] If

$$\mathfrak{S}(\bar{C}, \phi, P) = \bar{C}', \bar{\alpha}$$

(REMEMO)

(REBUILD)

$$\begin{array}{c}
\sigma(\delta, v) = \overline{C'}.C.\overline{C} \quad C = (c, e) \\
\theta = (\delta, \overline{C'}) \\
\hline
P \parallel \langle \overline{\theta}, C.\overline{C}, \tau[E[U(\lambda_\delta x.e(v))]] \rangle, \sigma \xrightarrow{PMem} \\
P \parallel \langle \overline{\theta}, \theta, \tau[E[B(v, e)]] \rangle, \sigma
\end{array}
\qquad
\begin{array}{c}
P \parallel \langle \overline{\theta}, \theta, \tau[B(v, e)] \rangle, \sigma \xrightarrow{\overline{\alpha}} P \parallel \langle \overline{\theta}, \theta, \tau[B(v, v')] \rangle, \sigma' \\
\hline
P \parallel \langle \overline{\theta}, \theta, \tau[E[B(v, e)]] \rangle, \sigma \xrightarrow{\overline{\alpha}} P \parallel \langle \overline{\theta}, \theta, \tau[E[B(v, v')]] \rangle, \sigma'
\end{array}$$

Figure 12. Incremental Partial Memoization.

then

$$P, \sigma \xrightarrow{\overline{\alpha}} P', \sigma$$

□

The proof is by induction on the length of $\overline{\alpha}$. The base case is a sequence of length one. We examine the three potential cases $\{\text{MCom}, \text{MCh}, \text{MSp}\}$.

MCom - By definition of \Im we know there exists a thread willing to either send or receive the value stored in the constraint. If the constraint is a send constraint, there exists a thread willing to receive in P . If the constraint is a receive constraint, there exists a thread willing to send the specific value in P . By definition we can apply the (MSend) or (MRecv) rules. **MCh** - This evaluation step can always be taken since it does not depend on the current program state. By definition we can apply the (MCh) rule. **MSp** - This evaluation step can always be taken since it does not depend on the current program state. By definition we can apply the (MSpawn) rule.

We assume the theorem holds for sequences of length n and show that it holds for sequences of length $n + 1$. By definition of \Im each constraint can be matched with only one thread in P . The rest holds by the argument outlined in the base case and induction.

We now define constraint capture in the presence of functions whose applications are elided through the use of memo information.

Theorem[Constraint Capture] If

$$P \parallel \langle \overline{\theta}, \theta, \tau[E[\lambda_\delta x.e(v)]] \rangle, \sigma \xrightarrow{\overline{\alpha}.Pmem.\overline{B'}} P' \parallel \langle (\delta, \overline{C}), \overline{\theta}, \theta, \tau[E[v']] \rangle, \sigma'$$

and

$$\{\text{PMem}, \text{App}\} \notin \overline{\alpha'} \quad \overline{\alpha'} \vdash_\tau \overline{\alpha''}$$

then

$$\forall \{Ch, Sp, Com, MCh, MSp, MCom, MRecv, MSend\} \in \overline{\alpha''}$$

$$\exists \{(Ch, 1), (Sp, e), ((S, 1, v), e), ((R, 1, v), e)\} \in \overline{C}$$

□

The proof is by induction on the length of $\overline{\alpha''}$. The base case is a sequence of length one. We examine the eight potential cases $\{\text{Com}, \text{Ch}, \text{Sp}, \text{MCom}, \text{MCh}, \text{MSp}, \text{MRecv}, \text{MSend}\}$.

Com - By definition of rule (Comm): adds a $(S, 1, v), e$ constraint in the case of a send and a $(S, 1, v), e$ receive.

Ch - By definition of rule (Channel): adds a $(Ch, 1)$ constraint.

Sp - By definition of rule (Spawn): adds a (Sp, e) constraint.

By lemmas *Constraint Capture (no memo)* and \Im *Safety* we know that constraints are generated for all functions and discharged safely.

MRecv - By definition of rule (MSend): adds a $(S, 1, v), e$ constraint.

MSend - By definition of rule (MRecv): adds a $(R, 1, v), e$ constraint.

MCh - By definition of rule (MCh): adds a $(Ch, 1)$ constraint.

MSp - By definition of rule (MSpawn): adds a (Sp, e) constraint.

The final case **MCom** (Rule MCom) matches constraints from a sending and a receiving thread based on the channel and value and it adds a $(S, 1, v), e$ constraint in the case of a send and a $(S, 1, v), e$ in the case of a receive.

We assume the theorem holds for sequences of length n and show that it holds for sequences of length $n + 1$. If the $n + 1$ step is $\{\text{Com}, \text{Ch}, \text{Sp}\}$ then we apply the same argument outlined in the base case. The rest holds by induction.

We can relate the states produced by memoized evaluation to the states constructed by the non-memoizing evaluator using the following transformation operators.

$$\begin{aligned}
\mathcal{T}((P_1 \parallel P_2), \sigma) &= \mathcal{T}(P_1, \sigma) \parallel \mathcal{T}(P_2, \sigma) \\
\mathcal{T}((\overline{\theta}, \overline{C}, e), \sigma) &= \mathcal{T}(e, \sigma)
\end{aligned}$$

$$\mathcal{T}(\lambda_\delta x.e, \sigma) = \lambda x.e$$

$$\mathcal{T}(e_1(e_2), \sigma) = \mathcal{T}(e_1)(\mathcal{T}(e_2))$$

$$\mathcal{T}(\text{spawn}(e), \sigma) = \text{spawn}(\mathcal{T}(e))$$

$$\mathcal{T}(\text{send}(e_1, e_2), \sigma) = \text{send}(\mathcal{T}(e_1), \mathcal{T}(e_2))$$

$$\mathcal{T}(\text{recv}(e), \sigma) = \text{recv}(\mathcal{T}(e))$$

$$\mathcal{T}(B(v, e), \sigma) = \mathcal{T}(e)$$

$$\mathcal{T}(U(\lambda_\delta x.e)(v), \sigma) = \mathcal{F}(v', \overline{C}) \text{ if } \sigma(\delta, v) = \overline{C}$$

$$e \quad \text{otherwise}$$

where

$$\mathcal{F}(e, []) = e$$

$$\mathcal{F}(e, C.\overline{C}) = \begin{cases} \mathcal{F}(\lambda_{-}e(\text{send}(1, v)), \overline{C}) & \text{if } C = ((S, 1, v), -) \\ \mathcal{F}(\lambda_{-}e(\text{recv}(1)), \overline{C}) & \text{if } C = ((R, 1, -), -) \\ \mathcal{F}(\lambda_{-}e(\text{spawn}(e')), \overline{C}) & \text{if } C = (Sp, e') \\ \mathcal{F}(\lambda_{x.e}(\text{mkCh}()), \overline{C}[x/1]) & \text{if } C = (Ch, 1) \\ & \text{and } x \notin FV(e) \end{cases}$$

\mathcal{T} transforms process states (and terms) defined under memo evaluation to process states (and terms) defined under non-memoized evaluation. It uses an auxiliary transform \mathcal{F} to translate constraints found in the memo store to core language terms. Each constraint defines an effectful action (e.g., sends, receives, channel creation, and spawns).

These operators provide a translation from the memo state defining constraints maintained by the memo evaluator to non-memoized terms. Defining the expression corresponding to a constraint is straightforward; the complexity in \mathcal{F} 's definition is because we must maintain the order in which these effects occur. We enforce ordering through nested function application, in which the most deeply nested function in the synthesized expression yields the memoized return value.

Given the ability to transform memoized states to non-memoized ones, we can define a safety theorem that ensures memoization does

not yield states which could not be realized under non-memoized evaluation:

Theorem[Safety] If

$$P \parallel \langle \bar{\theta}, \emptyset, \tau[E[\lambda_\delta x.e(v)]] \rangle, \sigma \xrightarrow{\bar{\alpha}.PMem.\bar{\beta}} P' \parallel \langle \bar{\theta}', \emptyset, \tau[E[v']] \rangle, \sigma'$$

then there exists $\alpha_1, \dots, \alpha_n \in \{App, Ch, Spn, Com\}$ s.t.

$$\langle \tau_{\mathcal{T}(P,\sigma)}, \mathcal{T}(E[\lambda_\delta x.e(v)]) \rangle \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} \langle \tau_{\mathcal{T}(P',\sigma')}, \mathcal{T}(E[v']) \rangle$$

□

The proof is by induction on the length of $\bar{\alpha}$. Each of the elements comprising $\bar{\alpha}$ corresponds to an action necessary to discharge previously recorded memoization constraints. We can show that every α step taken under memoization corresponds to zero or one step under non-memoized evaluation; zero steps for returns and memo actions that strip or build context tags U and B , and one step for core evaluation, and effectful actions (e.g., MCH, MSPAWN, MRECV, MSEND, and MCOM).

Partial Memoization and Scheduler aware – If $|\bar{\alpha}|$ is one, then α must be PMEM, which is the only rule that strips the U tag. The PMEM rule simply installs the memoized return value of the function being memoized. The value yielded by PMEM is the value previously recorded in the memo store. By the definition of RET this value must be the same as the value yielded by the application under core evaluation. In this case the function has no constraints to be discharged.

For the inductive step, we examine each memoizable action in turn. By lemmas *Constraint Capture* and \mathfrak{S} *Safety* we know that constraints are generated for all functions and discharged safely. A channel or thread creation action (i.e., MCH or MSPAWN) correspond directly to their core evaluation counterparts modulo renaming. The rules for MRECV and MSEND correspond to the COMM rule, sending or receiving the memoized value on a specific channel. Since the RESUME MEMO rule allows for constraints to be matched with other constraints we much examine the MCOM rule. MCOM also corresponds directly to the COMM rule. From the definition of \mathcal{T} , we can split any MCOM rule into an MRECV or MSEND by transforming one half of the communication.

The rules for RET and END MEMO do not correspond to any core evaluation rules. However, when paired with APP and PARTIAL MEMO, the pairs correspond to a core evaluation application. Both RET and END MEMO remove B 's and U 's respectively inserted by APP and PARTIAL MEMO, and thus such pairing is always feasible. By the definition of \mathcal{T} and the induction hypothesis, the value yielded by RET or END MEMO corresponds to the value yielded by application under core evaluation. Notice that the rule RESUME MEMO only matches constraints based on \mathfrak{S} . □

Incremental – The proof for incremental partial memoization follows directly from the proof of scheduler aware partial memoization since the rule REMEMO corresponds to END MEMO but inserts a B . For each REMEMO there will exist a REBUILD which removes the B inserted by REMEMO. These two rules behave similarly to the pairing of APP and RET and the same argument holds. □

Determining whether a function call can use previously constructed memo information is not free since every constraint match is defined as an evaluation step under \Rightarrow . An application can be *profitably* memoized only if the work to determine if it is memoizable is less than the work to evaluate it without employing memoization. Steps taken by the memo evaluator that match constraints, or initiate other memoization actions define work that would not be performed otherwise; conversely, memoization can avoid performing local steps taken to fully evaluate an application, although it may induce local actions in other threads to reach a global state

in which memoization constraints can be discharged. We formalize this intuition thus:

Theorem[Efficiency] Let $\bar{\alpha}$ be the smallest sequence such that

$$P \parallel \langle \bar{\theta}, \emptyset, \tau[E[\lambda_\delta x.e(v)]] \rangle, \sigma \xrightarrow{\bar{\alpha}.PMem.\bar{\alpha}'} P' \parallel \langle \bar{\theta}', \emptyset, \tau[E[v']] \rangle, \sigma'$$

holds, and let

$$\langle \tau_{\mathcal{T}(P,\sigma)}, \mathcal{T}(E[\lambda_\delta x.e(v)]) \rangle \xrightarrow{\bar{\beta}.\bar{\alpha}'} \langle \tau_{\mathcal{T}(P',\sigma')}, \mathcal{T}(E[v']) \rangle$$

If there are m occurrences of *Ret* tags and n occurrences of *PMem* tags in $\bar{\alpha}$, then $|\bar{\alpha}'| \leq |\bar{\beta}| + m + n$. □

As before, the proof follows from the definition of \mathcal{T} and proceeds by induction on the length of $\bar{\alpha}$. We examine all three definitions of partial memoization.

Partial Memoization and Scheduler aware – Without loss of generality, let $\bar{\alpha}$ be the *smallest* sequence for which the relation holds. As before, we proceed with the proof by induction on the length of $\bar{\alpha}$.

If $|\bar{\alpha}|$ is one, then α must be PMEM, which is the only rule that strips the U tag. Observe that END MEMO discharges no constraints, and yields the value recorded in the memo store. The minimal number of evaluation steps for an application under core evaluation is one (for an application of an abstraction that immediately yields a value). In the base case no constraints are discharged.

For the inductive step, we consider each rule under memoized evaluation in turn. By lemmas *Constraint Capture (no memo)* and \mathfrak{S} *Safety* we know that constraints are generated for all functions and discharged safely. By the structure of the rules and the safety theorem, evaluation steps taken by MCH and MSPAWN correspond directly to their core evaluation rule counterparts. The rules for MSEND and MRECV correspond to a single COMM step under core evaluation. The MCOM rule discharges memoization constraints in two threads. It consumes a single step under memo evaluation.

The rules for RET and END MEMO do not correspond to any core evaluation rules. However, when paired with APP and MEMO APP, the pairs correspond to an application. Both RET and END MEMO remove U 's and B 's respectively inserted by APP and MEMO APP. Therefore each sequence will contain one additional rule for each APP and MEMO APP step. Notice that the rule RESUME MEMO only augments the sequence based on constraints matched.

The rest of the rules have direct correspondence to rules in core evaluation. In a regular application each of the rules adds to the length of the sequence; in a memo application these steps are either skipped (in the case of an ordinary application), or contribute to the length of $\bar{\alpha}$. □

Incremental – The proof for incremental partial memoization follows directly from the proof of scheduler aware partial memoization since the rule REMEMO corresponds to END MEMO but inserts a B . For each REMEMO there will exist a REBUILD which removes the B inserted by REMEMO. These two rules behave similarly to the pairing of APP and RET and the same argument holds. □

A memoization candidate that induces a *PMem* transition under partial memoization may nonetheless be fully memoizable under memo evaluation. Moreover, the global state yielded by the *PMem* transition can be used by the non-memoizing evaluator to reach the same global state reached by successful memoization.

6. Implementation

Our implementation is incorporated within MLton [12], a whole-program optimizing compiler for Standard ML. The main changes

to the underlying compiler and library infrastructure are the insertion of write barriers to track shared memory updates, barriers to monitor function arguments and return values, hooks to the Concurrent ML [17] library to monitor channel based communication, and changes to the Concurrent ML scheduler. The entire implementation is roughly 5K lines of SML: 3K for the STM, and 300 lines of changes to CML.

6.1 Supporting Memoization

Because it will not in general be readily apparent if a memoized version of a CML function can be utilized at a call site, we delay a function application to see if its constraints can be matched; these constraints must be satisfied in the order in which they were generated.

Constraint matching can certainly fail on a receive constraint. A receive constraint obligates a receive operation to read a *specific* value from a channel. Since channel communication is blocking, a receive constraint that is being matched can choose from all values whose senders are currently blocked on the channel. This does not violate the semantics of CML since the values blocked on a channel cannot be dependent on one another; in other words, a schedule must exist where the matched communication occurs prior to the first value blocked on the channel.

Unlike a receive constraint, a send constraint can only fail if there are (a) no matching receive constraints on the sending channel that expect the value being sent, or (b) no receive operations on that same channel. A CML receive operation (not receive constraint) is ambivalent to the value it removes from a channel; thus, any receive on a matching channel will satisfy a send constraint.

If no receives or sends are enqueued on a constraint's target channel, a memoized execution of the function will block. Therefore, failure to fully discharge constraints by stalling memoization on a presumed unsatisfiable constraint does not compromise global progress. This observation is critical to keeping memoization overheads low.

Thus, in the case that a constraint is blocked on a channel that contains no other pending communication events or constraints, memoization induces no overheads, since the thread would have blocked regardless. However, if there exist communications or constraints that simply do not match the value the constraints expects, we can fail, and allow the thread to resume execution from the continuation stored within the constraint. To trigger such situations, we implement a simple heuristic. Our implementation records the number of context switches to a thread attempting to discharge a communication constraint. If this number exceeds a small constant (three in the benchmarks presented in the next section), memoization stops, and the thread continues execution within the function body immediately prior to that communication point.

Our memoization technique relies on efficient equality tests. We extend MLton's poly-equal function to support equality on reals and closures. Although equality on values of type real is not algebraic, built-in compiler equality functions were sufficient for our needs. To support efficient equality on functions, we approximate function equality as closure equality. Unique identifiers are associated with every closure and recorded within their environment; runtime equality tests on these identifiers are performed during memoization.

Memoization data is discarded during garbage collection. This prevents unnecessary build up of memoization meta-data during execution. As a heuristic, we also enforce an upper bound for the amount of memo data stored for each function, and the space that each memo entry can take. A function that generates a set of constraints whose size exceeds the memo entry space bound is not

memoized. For each memoized function, we store a list of memo meta-data. When the length of the list reaches the upper limit but new memo data is acquired upon an application of the function to previously unseen arguments, one entry from the list is removed at random.

6.2 CML hooks

The underlying CML library was also modified to make memoization efficient. The bulk of the changes were hooks to monitor channel communication and spawns, and to support constraint matching on synchronous operations, and to log successful communication (including selective communication and complex composed events).

The constraint matching engine also required a modification to the channel structure. Each channel is augmented with two additional queues to hold send and receive constraints. When a constraint is being tested for satisfiability, the opposite queue is first checked (e.g. a send constraint would check the receive constraint queue). If no match is found, the regular queues are checked for satisfiability. If the constraint cannot be satisfied immediately it is added to the appropriate queue.

6.3 Benchmarks

We examined two benchmarks to measure the effectiveness of partial memoization in a highly concurrent setting. The first benchmark is a speculative red-black tree implementation closely mirroring the example presented in Section 3. The second is a port of the STMBench7 benchmark. The port utilizes stream based channel communication instead of shared memory.

STMBench7 [7] is a comprehensive, tunable multi-threaded benchmark designed to compare different STM implementations and designs. Based on the well-known 007 database benchmark [4], STMBench7 simulates data storage and access patterns of CAD/CAM applications that operate over complex geometric structures. At its core, STMBench7 builds a tree of assemblies whose leafs contain bags of components; these components have a highly connected graph of atomic parts and design documents. Indices allow components, parts, and documents to be accessed via their properties and IDs. Traversals of this graph can begin from the assembly root or any index and sometimes manipulate multiple pieces of data.

STMBench7 was originally written in Java. We have implemented a port to Standard ML (roughly 1.5K lines of SML); our implementation also involved building an STM implementation to support atomic sections that is loosely based on the techniques described in [18, 3]. In our port, all nodes in the complex assembly structure and atomic parts graph are represented as servers with one receiving channel and handles to all other adjacent nodes. Handles to other nodes are simply the channels themselves. Each server thread waits for a message to be received, performs the requested computation, and then asynchronously sends the subsequent part of the traversal to the next node. A transaction can thus be implemented as a series of channel based communications with various server nodes.

7. Results

To measure the effectiveness of our memoization technique, we executed two configurations of STMBench7 and the red-black tree, and measured overheads and performance by averaging results over ten executions. For STMBench7 the *non-memoized* configuration uses our STM implementation without any memoization where as the memoized configuration implements partial memoization of aborted transactions. The original configuration of red-black tree

uses speculation without leveraging memoization and the memoized configuration utilizes partial memoization to avoid traversals previously seen. The benchmarks were run on an Intel P4 2.4 GHz machine with one GByte of memory running Gentoo Linux, compiled and executed using MLton release 20051202.

STMBench7 was executed on a graph in which there were approximately 280k complex assemblies and 140k assemblies whose bags referenced one of 100 components; by default, each component contained a parts graph of 100 nodes. The red-black tree consisted of 150k nodes. Both benchmarks create a number of threads proportional to the number of nodes in the underlying data structure being manipulated.

Our tests varied the read-only/read-write ratio (see Fig. 13) within transactions in STMBench7, and the the percent of repeated traversals for the red-black tree. Only transactions that modify values can cause aborts. Thus, an execution where all transactions are read-only cannot be accelerated, but one in which transactions can frequently abort offers potential opportunities for memoization. Similarly, for the red-black tree, configurations in which there is a high degree of repeated traversals offer the best performance gain. We varied the percentage of repeated traversals which (see Fig. 13) were chosen out of a set of 100 previously executed traversals. Each line in the graph shows performance under a workload that reflected a certain percentage of change to the tree induced by insertions, deletions, etc. Writes were not repeated.

For STMBench7 we varied the maximum number cached memoized transaction runs (this was held constant at 16 for the red-black tree) stored for each function. Tests with a small number experienced less memoized data utilization than those with a large one. Naturally, the larger the size of the cache used to hold memoized information, the greater the overhead.

Memoization leads to substantial performance improvements when aborts are likely to be more frequent, or when updates to the tree are likely to lead to a non-memoizable speculation. percentage of read-only transactions is 60%, we see a 20% improvement in runtime performance compared to a non-memoizing implementation for STMBench7. As another data point, we observe roughly 17% performance improvement for the red-black tree implementation when 80% of all queries are repeated, even when 20% of all actions are updates. Memory overheads were proportional to cache sizes and averaged roughly 15% for caches of size 16. The cost to support memoization is seen when there are 100% read-only transactions (for STMBench7) or 0% repeated traversals (for the red-black tree); for both benchmarks, the overhead is roughly 7%. In such cases, there are never any aborts or repeated traversals, and thus no benefits accrue from using memoization.

8. Related Work

Memoization, or function caching [11, 15, 10, 19], is a well understood method to reduce the overheads of redundant function execution. Memoization of functions in a concurrent setting is significantly more difficult and usually highly constrained [13]. We are unaware of any existing techniques or implementations that apply memoization to the problem of reducing transaction overheads in languages that support selective communication and dynamic thread creation. Our approach also bears resemblance to the procedure summary construction for concurrent programs [16]. However, these approaches tend to be based on a static analysis (e.g., the cited reference leverages procedure summaries to improve the efficiency of model checking) and thus are obligated to make use of memoization greedily. Because our motivation is quite different, our approach can consider lazy alternatives, ones that leverage

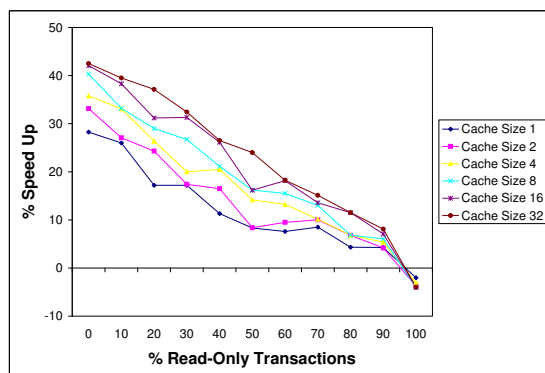
synchronization points to stall memo information use, resulting in potentially improved runtime benefit.

Self adjusting mechanisms [2] leverage memoization along with change propagation to automatically alter a program's execution to a change of inputs given an existing execution run. Memoization is used to identify parts of the program which have not changed from the previous execution while change propagation is harnessed to install changed values where memoization cannot be applied. New proposals [1] have been presented for self-adjusting techniques to be applied in context in which references are updated more than once. While it may be possible using these techniques to propagate changes among communicating threads, we know of no work that directly studies this problem. All existing designs consider the problem of adapting complete programs to changes in their data, which leads to a model quite different from the ideas discussed here. There has also been recent work on using change propagation in a parallel setting [8]. The programming model assumes fork/join parallelism, and is therefore not suitable for the kinds of contexts we consider.

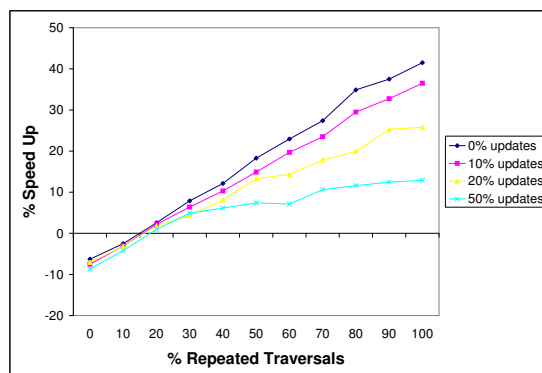
Our technique also shares some similarity with transactional events [5]. Transactional events require arbitrary lookahead in evaluation to determine if a complex composed event can commit. Partial memoization avoids the need for arbitrary lookahead, failure to discharging memoization constraints simply causes execution to proceed as normal.

References

- [1] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative Self-Adjusting Computation. In *POPL*, pages 309–322, 2008.
- [2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective Memoization. In *POPL*, pages 14–25, 2003.
- [3] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *PLDI*, pages 26–37, 2006.
- [4] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The 007 benchmark. *SIGMOD Record*, 22(2), 1993.
- [5] Kevin Donnelly and Matthew Fluet. Transactional Events. In *ICFP*, pages 124–135, 2006.
- [6] Jim Gray and Andreas Reuter. *Transaction Processing*. Morgan-Kaufmann, 1993.
- [7] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *EuroSys*, pages 315–324, 2007.
- [8] Matthew Hammer, Umut A. Acar, Mohan Rajagopalan, and Anwar Ghuloum. A Proposal for Parallel Self-Adjusting Computation. In *Workshop on Declarative Aspects of Multicore Programming*, 2007.
- [9] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *Proceedings of the ACM Conference on Principles and Practice of Parallel Programming*, pages 48–60, 2005.
- [10] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *PLDI*, pages 311–320, 2000.
- [11] Yanhong A. Liu and Tim Teitelbaum. Caching Intermediate Results for Program Improvement. In *PEPM*, pages 190–201, 1995.
- [12] MLton. <http://www.mlton.org>.
- [13] Christopher J. F. Pickett and Clark Verbrugge. Software Thread Level Speculation for the Java Language and Virtual Machine Environment. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, 2005.
- [14] W. Pugh and T. Teitelbaum. Incremental Computation via Function Caching. In *POPL*, pages 315–328, 1989.



(a)



(b)

Figure 13. Figure (a) presents normalized runtime speedup with a varying read to write ratio for STMBench7. Figure (b) shows normalized runtime speedup with a varying traversal to update ratio for a speculative red-black tree implementation with a cache size of 16.

- [15] William Pugh. An Improved Replacement Strategy for Function Caching. In *LFP*, pages 269–276, 1988.
- [16] Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof. Summarizing procedures in concurrent programs. In *POPL*, pages 245–255, 2004.
- [17] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
- [18] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a High-Performance Software Transactional Memory system for a Multi-Core Runtime. In *PPoPP*, pages 187–197, 2006.
- [19] Kedar Swadi, Walid Taha, Oleg Kiselyov, and Emir Pasalic. A Monadic Approach for Avoiding Code Duplication When Staging Memoized Functions. In *PEPM*, pages 160–169, 2006.