Partial Memoization of Concurrency and Communication

Lukasz Ziarek KC Sivaramakrishnan Suresh Jagannathan

Department of Computer Science
Purdue University
{lziarek,chandras,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.

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 these communication actions. 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 whose threads may communicate 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.

To validate the effectiveness of our ideas, we consider the benefits of memoization for reducing the overhead of recomputation for streaming, server-based, and transactional applications executed on a multi-core machine. We show that on a variety of workloads, memoization can lead to substantial performance improvements without incurring high memory costs.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; D.1.3 [*Concurrent Programming*]: memoization; D.3.1 [*Formal Definitions and Theory*]: Semantics

Keywords Concurrent programming, partial memoization, software transactions, Concurrent ML, multicore systems.

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.

Copyright © ACM [to be supplied]...$5.00.

1. Introduction

Eliminating redundant computation is an important optimization supported by many language implementations. One important instance of this optimization class is memoization (Liu and Teitelbaum 1995; Pugh and Teitelbaum 1989; Acar et al. 2003), 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 multi-core platforms, and renewed interest in streaming (Gordon et al. 2006), speculative (Pickett and Verbrugge 2005) and transactional (Harris and Fraser 2003; Adl-Tabatabai et al. 2006) 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 and must be retried (Harris et al. 2005), 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 (Reppy 1999)). 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 constraints 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 (Reppy 1999).

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.

1.1 Contributions

Besides providing a formal characterization of these ideas, we also present performance evaluation of two parallel benchmarks. We consider the effect of memoization on improving performance of multi-threaded CML applications executing on a multicore architectures. Our results indicate that memoization can lead to substantial runtime performance improvement over a non-memoized version of the same program, 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 wall-clock performance for 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. Our formal treatment does not consider references, but there are no additional complications that ensue in order to handle them; our implementation supports all of Standard ML.

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) values consumed by f on channels in an earlier call are again ready to be sent to other threads; (d) channels created in an earlier call have the same actions performed on them, 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.

To avoid making a call, a send action performed within the applied function, for example, will need to be paired with a receive operation executed by some other thread. Unfortunately, there may be no thread currently scheduled that is waiting to receive on this channel. Consider an application that calls a memoized function f which (a) creates a thread T that receives a value on channel c, and (b) sends a value on c computed through values received on other channels that is then consumed by T. To safely use the memoized return value for f nonetheless still requires that T be instantiated, and that communication events executed in the first call can still be satisfied (e.g., the values f previously read on other channels are still available on those channels). Ensuring these actions can succeed may involve an exhaustive exploration of the execution state space to derive a schedule that allows us to consider the call in the context of a global state in which these conditions are satisfied. Because such an exploration may be infeasible in practice, our formulation also supports *partial* memoization. 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.

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, effectively encoding a server. Accessing and modifying a tree node's data is thus accomplished through a communication protocol with the node's input and output channels. Abstractly, a read corresponds to a receive operation (`recv`) from a node's output channel, and a write to a send (`send`) on a node's input channel.

When a node is initially created, we instantiate two new channels (`channel()`), and spawn a server. The function `server`, the concrete representation of a node, chooses between accepting a communication on the incoming channel `cIn` (corresponding to a write operation) and sending a communication on the outgoing channel `cOut` (corresponding to a read operation), synchronizes on the communication, and then updates the server through a recursive call.

```
datatype rbtree = N of {cOut: node chan, cIn: node chan}
         and node = Node of {color: color, value: int,
                             left:rbtree, right:rbtree}
                  | EmptyNode
fun node(c:color, v:int,
         l:rbtree, r:rbtree):rbtree =
 let val (cOut, cIn) = (channel(), channel())
     val node = Node{color=c,value=v,left=l,right=r}
     fun server(node) =
       sync(
        choose([wrap(sendEvt(cOut, node),
                             (fn x => server(node)) ),
                wrap(recvEvt(cIn),
                             (fn x => server(x)))]))
 in spawn(fn () => server(node)); N{cOut=cOut, cIn=cIn})
 end
```

For example, the procedure `contains` 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 `recvT` operation reads the value from each node, and based on this value `check` navigates the tree:

```
fun recvT(N{cOut, cIn}:rbtree) = recv(cOut)
fun contains (n:int, tree:rbtree):bool =
  let fun check(n,  tree) =
    (case recvT(tree)
       of Node {color,value,left,right} =>
          (case Int.compare (value, n)
             of EQUAL => true
              | GREATER => check (n,left)
              | LESSER => check (n,right))
        | EmptyNode => false)
  in check(n, tree)
  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 unbolded, 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 node values: (red, 48), (black, 76), (red, 85), and (black, 79) in that order during a traversal. In Fig. 1 thread T1 can take advantage of memoization, while thread T2 subsequently recolors the node containing 85.

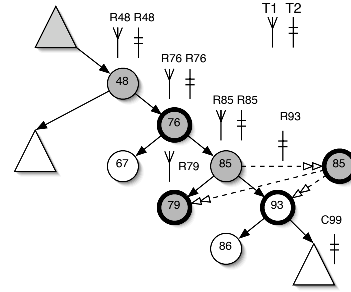

Figure 1. Memoization can be used to avoid redundant tree traversals. In this example, two threads traverse a red-black tree. Each node in the tree is represented as a process that outputs the current value of the node, and inputs new values. The shaded path illustrates memoization potential.

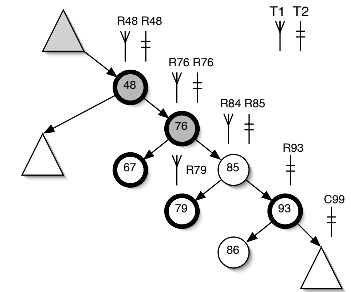

Figure 2. Even if the constraints stored in a memoized call cannot be fully satisfied at a subsequent call, it may be possible to discharge some fraction of these constraints, retaining some of the optimization benefits memoization affords.

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 85 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.

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 recomputing 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. Communication between threads is achieved using synchronous channels.

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. Although the core language has no support for selective communication, extending it to support choice is straight forward. Memoization would simply record the result of the choice and replay would only be possible if the recorded choice was satisfiable.

In the following, we write $\overline{\alpha}$ to denote a sequence of zero or more elements, $\overline{\beta}.\overline{\alpha}$ to denote sequence concatenation, and $\emptyset$ to denote an empty sequence. Metavariables $\mathtt{x}$ and $\mathtt{y}$ range over variables, $\mathtt{t}$ ranges over thread identifiers, $\mathtt{l}$ ranges over channels, $\mathtt{v}$ ranges over values, and α, β denote tags that label individual actions in a program's execution. We use P to range over program states, E for evaluation contexts, and $\mathtt{e}$ 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 send. 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. 3. Expressions are either variables, locations that represent channels, λ-abstractions, function applications, thread creation operations, or communication actions that send and receive messages on channels. We do not consider references in this core language as they can be modeled in terms of operations on channels (Reppy 1999).

A thread context $\mathtt{t}[E[\mathtt{e}]]$ denotes an expression $\mathtt{e}$ available for execution by a thread with identifier $\mathtt{t}$ within context E. Evaluation is specified via a relation ($\longmapsto$) that maps a program state (P) to another program state. Evaluation rules are applied up to commutativity of parallel composition ($\|$). An evaluation step is marked with a tag that indicates the action performed by that step. As shorthand, we write $P \overset{\overline{\alpha}}{\longmapsto} P'$ to represent the sequence of actions $\overline{\alpha}$ that transforms P to P'.

Application (rule App) substitutes the argument value for free occurrences of the parameter in the body of the abstraction, and channel creation (rule Channel) results in the creation of a new location that acts as a container for message transmission and reception. A spawn action (rule Spawn), given an expression $\mathtt{e}$ 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 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 ($\overline{C}$) 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 (σ), 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 memo store contains only one function/value pair for simplicity of the presentation. We can envision extending the memo store to contain multiple memoized versions of a function based on its arguments or constraints. We utilize two thread contexts $\mathtt{t}[\mathtt{e}]$ and $\mathtt{t}_{\overline{C}}[\mathtt{e}]$, the former to indicate that evaluation of terms should be captured within the memo store, and the latter to indicate that previously recorded constraints should be discharged. We elaborate on their use below.

The definition of the language augmented with memoization support is given in Fig. 4. We now define evaluation using a new relation ($\Longrightarrow$) defined over two global configurations. In one case, a configuration consists of a program state (P) and a memo store (σ). This configuration is used when evaluation does not leverage memoized information. The second configuration is defined in terms of a thread id and constraint sequence pair ($(\mathtt{t}, \overline{C})$), a program state ($P$), and a memo store ($\sigma$); transitions use this configuation when discharging constraints recorded from a previous memoized application.

In addition, a thread state is augmented to hold an additional structure. The memo state (θ) records the function identifier (δ), the argument ($\mathtt{v}$) supplied to the call, the context (E) in which the call is performed, and the sequence of constraints ($\overline{C}$) that are built during the evaluation of the application being memoized.

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 ($\mathtt{R}$ for receive and $\mathtt{S}$ 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 ($\mathtt{Sp}$), the expression being spawned, and the continuation immediately prior to the spawn. For a channel creation operation ($\mathtt{Ch}$), the constraint records the location of the channel and the continuation immediately prior to the creation operation. Returns are also modeled as constraints ($\mathtt{Rt}, \mathtt{v}$) where $\mathtt{v}$ is the return value of the application being memoized.

Consider an application of function $\mathtt{f}$ to value $\mathtt{v}$ that has been memoized. Since subsequent calls to $\mathtt{f}$ with $\mathtt{v}$ may not be able to discharge all constraints, we need to record the program points for all communication actions within $\mathtt{f}$ 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 [1] (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 them within saved continuations recorded within these constraints. 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; the memo state (θ) stores the evaluation context representing the caller's continuation. This context is restored once the application completes (see rule Ret).

[1] We also record continuations on non-failing constraints; while not strictly necessary, doing so simplifies our correctness proofs.

SYNTAX:

$$\begin{array}{rcl} P & ::= & P \| P \mid \mathtt{t}[\mathtt{e}] \\ \mathtt{e} \in Exp & ::= & \mathtt{x} \mid \mathtt{y} \mid \mathtt{v} \mid \mathtt{e}(\mathtt{e}) \mid \mathtt{spawn}(\mathtt{e}) \\ & \mid & \mathtt{mkCh}() \mid \mathtt{send}(\mathtt{e},\mathtt{e}) \mid \mathtt{recv}(\mathtt{e}) \\ \mathtt{v} \in Val & ::= & \mathtt{unit} \mid \lambda \mathtt{x}.\mathtt{e} \mid \mathtt{l} \end{array}$$

PROGRAM STATES:

$$\begin{array}{rcll} P & \in & Process & \\ \mathtt{t} & \in & Tid & \\ \mathtt{x},\mathtt{y} & \in & Var & \\ \mathtt{l} & \in & Channel & \\ \alpha,\beta & \in & Tag & = \{App, Ch, Spn, Com\} \end{array}$$

EVALUATION CONTEXTS:

$$\begin{array}{l} E ::= [\,] \mid E(\mathtt{e}) \mid \mathtt{v}(E) \mid \mathtt{spawn}(E) \mid \\ \quad \mathtt{send}(E,\mathtt{e}) \mid \mathtt{send}(\mathtt{l},E) \mid \mathtt{recv}(E) \end{array}$$

(APP)

$$\frac{}{P\|\mathtt{t}[E[\lambda \mathtt{x}.\mathtt{e}\,(\mathtt{v})]] \overset{App}{\longmapsto} P\|\mathtt{t}[E[\mathtt{e}[\mathtt{v}/\mathtt{x}]]]}$$

(CHANNEL)

$$\frac{\mathtt{l}\ fresh}{P\|\mathtt{t}[E[\mathtt{mkCh}()]] \overset{Ch}{\longmapsto} P\|\mathtt{t}[E[\mathtt{l}]]}$$

(SPAWN)

$$\frac{\mathtt{t}'\ fresh}{P\|\mathtt{t}[E[\mathtt{spawn}(\lambda \mathtt{x}.\mathtt{e})]] \overset{Spn}{\longmapsto} P\|\mathtt{t}[E[\mathtt{unit}]]\|\mathtt{t}'[\mathtt{e}[\mathtt{unit}/\mathtt{x}]]}$$

(COMM)

$$\frac{P = P'\|\mathtt{t}[E[\mathtt{send}(\mathtt{l},\mathtt{v})]]\|\mathtt{t}'[E'[\mathtt{recv}(\mathtt{l})]]}{P \overset{Com}{\longmapsto} P'\|\mathtt{t}[E[\mathtt{unit}]]\|\mathtt{t}'[E'[\mathtt{v}]]}$$

Figure 3. A concurrent language with synchronous communication.

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 ($\overline{\theta}$) 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. 4.

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). 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 memo state (θ) associated with the thread in which the application occurs.

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 $\mathfrak{I}$ defined in Fig. 5. Abstractly, $\mathfrak{I}$ checks the global state to determine which communication, channel creation, and spawn creation constraints (the possible effectful actions in our language) can be satisfied, and returns a set of failed constraints, representing those actions that could not be satisfied. The thread context ($\mathtt{t}_{\overline{C}}[\mathtt{e}]$) is used signal the utilization of memo information. The failed constraints are added to the original thread context.

Rule MEMO APP yields a new global configuration whose thread id and constraint sequence ($(\mathtt{t}, \overline{C})$) corresponds to the constraints satisfiable in the current global state (defined as $\overline{C''}$) for thread $\mathtt{t}$ as defined by $\mathfrak{I}$. These constraints, when discharged, will leave the thread performing the memoized call in a new state in which the evaluation of the call is the expression associated with the first *failed* constraint returned by $\mathfrak{I}$. As we describe below in Sec 4.3, there is always at least one such constraint, namely `Rt`, the return constraint that holds the return value of the memoized call. We also introduce a rule to allow the completion of memo information use (rule END MEMO). The rule installs the continuation of the first currently unsatisfied constraint; no further constraints are subsequently examined. In this formulation, the other failed constraints are simply discarded; we present an extension of this semantics in Section. 4.6 that make use of them.

4.3 Constraint Matching

The constraints built as a result of evaluating these rules are discharged by the rules shown in Fig. 6. Each rule in Fig. 6 is defined with respect to a thread id and constraint sequence. Thus, at any given point in its execution, a thread is either building up memo constraints (i.e., the thread is of the form $\mathtt{t}[\mathtt{e}]$) within an application for subsequent calls to utilize, or attempting to discharge these constraints (i.e., the thread is of the form $\mathtt{t}_{\overline{C}}[\mathtt{e}]$) for applications indexed in the memo store.

The function $\mathfrak{I}$ leverages the evaluation rules defined in Fig. 6 by examining the global state and determining which constraints can be discharged, except for the return constraint. $\mathfrak{I}$ takes a constraint set ($\overline{C}$) and a program state (P) and returns a sequence of unmatchable constraints ($\overline{C'}$). 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

SYNTAX:

$$P ::= P \| P \mid \langle \overline{\theta}, \mathtt{t}[\mathtt{e}] \rangle \mid \langle \overline{\theta}, \mathtt{t}_{\overline{C}}[\mathtt{e}] \rangle$$
$$\mathtt{v} \in Val ::= \mathtt{unit} \mid \lambda_\delta\, \mathtt{x.e} \mid \mathtt{l}$$
$$E \in Context$$

PROGRAM STATES:

$$\begin{array}{lcll}
\delta & \in & MemoId & \\
\mathtt{c} & \in & FailableConstraint = & (\{R,S\} \times Loc \times Val) + \mathtt{Rt} \\
C & \in & Constraint & = (FailableConstraint \times Exp) + \\
 & & & ((\mathtt{Sp} \times Exp) \times Exp) + ((\mathtt{Ch} \times Location) \times Exp) \\
\sigma & \in & MemoStore & = MemoId \times Val \to Constraint^* \\
\theta & \in & MemoState & = MemoId \times Val \times Context \times Constraint^* \\
\alpha, \beta & \in & Tag & = \{Ch, Spn, Com, Fun, App, Ret, MCom, \\
 & & & MCh, MSp, MemS, MemE, MemR, MemP\}
\end{array}$$

CONSTRAINT ADDITION:

$$\frac{\overline{\theta'} = \{(\delta, \mathtt{v}, E, \overline{C}.C) \mid (\delta, \mathtt{v}, E, \overline{C}) \in \overline{\theta}\}}{\overline{\theta}, C \succ \overline{\theta'}}$$

(CHANNEL)

$$\frac{\overline{\theta}, ((\mathtt{Ch}, \mathtt{l}), E[\mathtt{mkCh}()]) \succ \overline{\theta'} \quad \mathtt{l}\ fresh}{P \| \langle \overline{\theta}, \mathtt{t}[E[\mathtt{mkCh}()]] \rangle, \sigma \stackrel{Ch}{\Longrightarrow} P \| \langle \overline{\theta'}, \mathtt{t}[E[\mathtt{l}]] \rangle, \sigma}$$

(FUN)

$$\frac{\delta\ fresh}{P \| \langle \overline{\theta}, \mathtt{t}[E[\lambda\, \mathtt{x.e}]] \rangle, \sigma \stackrel{Fun}{\Longrightarrow} P \| \langle \overline{\theta}, \mathtt{t}[E[\lambda_\delta\, \mathtt{x.e}]] \rangle, \sigma}$$

(SPAWN)

$$\frac{\begin{array}{c}\mathtt{t}'\ fresh \quad \overline{\theta}, ((\mathtt{Sp}, \lambda_\delta\, \mathtt{x.e}(\mathtt{unit})), E[\mathtt{spawn}(\lambda_\delta\, \mathtt{x.e})]) \succ \overline{\theta'} \\ t_k = \langle \overline{\theta'}, \mathtt{t}[E[\mathtt{unit}]] \rangle \\ t_s = \langle \emptyset, \mathtt{t}'[\lambda_\delta\, \mathtt{x.e}(\mathtt{unit})] \rangle\end{array}}{P \| \langle \overline{\theta}, \mathtt{t}[E[\mathtt{spawn}(\lambda_\delta\, \mathtt{x.e})]] \rangle, \sigma \stackrel{Spn}{\Longrightarrow} P \| t_k \| t_s, \sigma}$$

(COMM)

$$\frac{\begin{array}{c}P = P' \| \langle \overline{\theta}, \mathtt{t}[E[\mathtt{send}(\mathtt{l}, \mathtt{v})]] \rangle \| \langle \overline{\theta'}, \mathtt{t}'[E'[\mathtt{recv}(\mathtt{l})]] \rangle \\ \overline{\theta}, ((\mathtt{S}, \mathtt{l}, \mathtt{v}), E[\mathtt{send}(\mathtt{l}, \mathtt{v})]) \succ \overline{\theta''} \quad \overline{\theta'}, ((\mathtt{R}, \mathtt{l}, \mathtt{v}), E'[\mathtt{recv}(\mathtt{l})]) \succ \overline{\theta'''} \\ t_s = \langle \overline{\theta''}, \mathtt{t}[E[\mathtt{unit}]] \rangle \quad t_r = \langle \overline{\theta'''}, \mathtt{t}'[E'[\mathtt{v}]] \rangle\end{array}}{P, \sigma \stackrel{Com}{\Longrightarrow} P' \| t_s \| t_r, \sigma}$$

(RET)

$$\frac{\theta = (\delta, \mathtt{v}, E, \overline{C})}{\begin{array}{c}P \| \langle \theta.\overline{\theta}, \mathtt{t}[\mathtt{v}'] \rangle, \sigma \stackrel{Ret}{\Longrightarrow} \\ P \| \langle \overline{\theta}, \mathtt{t}[E[\mathtt{v}']] \rangle, \sigma[(\delta, \mathtt{v}) \mapsto \overline{C}.(\mathtt{Rt}, \mathtt{v}')]\end{array}}$$

(APP)

$$\frac{(\delta, \mathtt{v}) \notin Dom(\sigma) \quad \theta = (\delta, \mathtt{v}, E, \emptyset)}{P \| \langle \overline{\theta}, \mathtt{t}[E[\lambda_\delta\, \mathtt{x.e}\,(\mathtt{v})]] \rangle, \sigma \stackrel{App}{\Longrightarrow} P \| \langle \theta.\overline{\theta}, \mathtt{t}[\mathtt{e}[\mathtt{v}/\mathtt{x}]] \rangle, \sigma}$$

(MEMO APP)

$$\frac{\begin{array}{c}(\delta, \mathtt{v}) \in Dom(\sigma) \quad \sigma(\delta, \mathtt{v}) = \overline{C} \\ \Im(\overline{C}, P) = \overline{C'} \quad \overline{C} = \overline{C''}.\overline{C'}\end{array}}{P \| \langle \overline{\theta}, \mathtt{t}[E[\lambda_\delta\, \mathtt{x.e}\,(\mathtt{v})]] \rangle, \sigma \stackrel{MemS}{\Longrightarrow} (\mathtt{t}, \overline{C''}), P \| \langle \overline{\theta}, \mathtt{t}_{\overline{C'}}[E[\lambda_\delta\, \mathtt{x.e}\,(\mathtt{v})]] \rangle, \sigma}$$

(END MEMO)

$$\frac{C = (\mathtt{c}, \mathtt{e}')}{(\mathtt{t}, \emptyset), P \| \langle \overline{\theta}, \mathtt{t}_{C.\overline{C}}[E[\lambda_\delta\, \mathtt{x.e}\,(\mathtt{v})]] \rangle, \sigma \stackrel{MemE}{\Longrightarrow} P \| \langle \overline{\theta}, \mathtt{t}[E[\mathtt{e}']] \rangle, \sigma}$$

Figure 4. A concurrent language supporting memoization of synchronous communication and dynamic thread creation.

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. We discuss the implications of this restriction in Section 4.5.

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 $((\mathtt{Ch}, \mathtt{l}), E[\mathtt{e}])$ (rule MCH) creates a new channel location $\mathtt{l}'$, and replaces all occurrences of $\mathtt{l}$ found in the remaining constraint sequence for this thread with $\mathtt{l}'$; 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, respectfully, but impose no preconditions on the state in order for these actions to succeed.

There are two communication constraint matching rules ($\stackrel{MCom}{\Longrightarrow}$), both of which may indeed fail. If the current constraint expects to receive value $\mathtt{v}$ on channel $\mathtt{l}$, and there exists a thread able to send $\mathtt{v}$ on $\mathtt{l}$, evaluation proceeds to a state 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 $\mathtt{v}$ on channel $\mathtt{l}$, and there exists a thread waiting on $\mathtt{l}$, 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 there is no matching receiver, the constraint fails.

Observe that there is no evaluation rule for the $\mathtt{Rt}$ constraint that can consume it. This constraint contains the return value of the memoized function (see rule RET). If all other constraints have

$$
\begin{array}{lcl}
\Im(((\mathtt{S},\mathtt{l},\mathtt{v}),\mathtt{e}).\overline{C}, P \| \langle \theta, \mathtt{t}[E[\mathtt{recv}(\mathtt{l})]]\rangle) & = & \Im(\overline{C}, P) \\
\Im(((\mathtt{R},\mathtt{l},\mathtt{v}),\mathtt{e}).\overline{C}, P \| \langle \theta, \mathtt{t}[E[\mathtt{send}(\mathtt{l},\mathtt{v})]]\rangle) & = & \Im(\overline{C}, P) \\
\Im((\mathtt{Rt},\mathtt{v}), P) & = & (\mathtt{Rt},\mathtt{v}) \\
\Im(((\mathtt{Ch},\mathtt{l}),\mathtt{e}).\overline{C}, P) & = & \Im(\overline{C}, P) \\
\Im(((\mathtt{Sp},\mathtt{e}'),\mathtt{e}).\overline{C}, P) & = & \Im(\overline{C}, P) \\
\Im(\overline{C}, P) & = & \overline{C}, \quad \textit{otherwise}
\end{array}
$$

Figure 5. The function $\Im$ yields the set of constraints $\overline{C}$ which are not satisfiable in program state P.

```
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, v4); ...;
   f()
end
```

Figure 7. Determining if an application can fully leverage memo information may require examining an arbitrary number of possible thread interleavings.

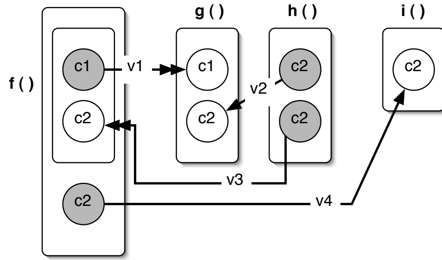

Figure 8. The communication pattern of the code in Fig 7. Circles represent operations on channels. Gray circles are sends and white circles are receives. Double arrows represent communications that are captured as constraints during memoization.

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. 7 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`. The communication pattern is depicted in Fig. 8. Separate threads of control are shown as rectangles. Communication actions are represented as circles; gray for send actions and white for receives. The channel on which the communication action takes place is annotated on the circle and the value which is sent on the arrow. Double arrows represent communication actions for which constraints are generated.

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

```
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.

`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 ...) 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 to `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 $\Im$ does not examine future actions of threads and thus can only match a constraint with a thread if that thread is able to match the constraint in the 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 memoized 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`.

(MCH)

$$\frac{C = ((\mathtt{Ch},\mathtt{l}),_) \quad \mathtt{l}'\ \textit{fresh} \qquad \overline{C''} = \overline{C}[\mathtt{l}'/\mathtt{l}] \quad \overline{\theta},C \succ \overline{\theta'}}{(\mathtt{t},C.\overline{C}),P\|\langle\overline{\theta},\mathtt{t}_{\overline{C'}}[E[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]]\rangle,\sigma \stackrel{MCh}{\Longrightarrow} (\mathtt{t},\overline{C''}),P\|\langle\overline{\theta'},\mathtt{t}_{\overline{C'}}[E[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]]\rangle,\sigma}$$

(MSPAWN)

$$\frac{C = ((\mathtt{Sp},\mathtt{e}'),_) \quad \mathtt{t}'\ \textit{fresh} \quad \overline{\theta},C \succ \overline{\theta'}}{(\mathtt{t},C.\overline{C}),P\|\langle\overline{\theta},\mathtt{t}_{\overline{C'}}[E[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]]\rangle,\sigma \stackrel{MSp}{\Longrightarrow} (\mathtt{t},\overline{C}),P\|\langle\overline{\theta'},\mathtt{t}_{\overline{C'}}[E[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]]\rangle\|\langle\emptyset,\mathtt{t}'[\mathtt{e}']\rangle,\sigma}$$

(MRECV)

$$\frac{\begin{array}{c} C = ((\mathtt{R},\mathtt{l},\mathtt{v}),_) \\ t_s = \langle\overline{\theta},\mathtt{t}[E[\mathtt{send}(\mathtt{l},\mathtt{v})]]\rangle \quad t_r = \langle\overline{\theta'},\mathtt{t}'_{\overline{C'}}[E'[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]]\rangle \\ \overline{\theta'},C \succ \overline{\theta'''} \quad \overline{\theta},((\mathtt{S},\mathtt{l},\mathtt{v}),E[\mathtt{send}(\mathtt{l},\mathtt{v})]) \succ \overline{\theta''} \\ t_{s'} = \langle\overline{\theta''},\mathtt{t}[E[\mathtt{unit}]]\rangle \quad t_{r'} = \langle\overline{\theta'''},\mathtt{t}'_{\overline{C'}}[E'[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]]\rangle \end{array}}{(\mathtt{t}',C.\overline{C}),P\|t_s\|t_r,\sigma \stackrel{MCom}{\Longrightarrow} (\mathtt{t}',\overline{C}),P\|t_{s'}\|t_{r'},\sigma}$$

(MSEND)

$$\frac{\begin{array}{c} C = ((\mathtt{S},\mathtt{l},\mathtt{v}),_) \\ t_s = \langle\overline{\theta'},\mathtt{t}'_{\overline{C'}}[E[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]]\rangle \quad t_r = \langle\overline{\theta},\mathtt{t}[E'[\mathtt{recv}(\mathtt{l})]]\rangle \\ \overline{\theta'},C \succ \overline{\theta'''} \quad \overline{\theta},((\mathtt{R},\mathtt{l},\mathtt{v}),E'[\mathtt{recv}(\mathtt{l})]) \succ \overline{\theta''} \\ t_{s'} = \langle\overline{\theta'''},\mathtt{t}'_{\overline{C'}}[E[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]]\rangle \quad t_{r'} = \langle\overline{\theta''},\mathtt{t}[E'[\mathtt{v}]]\rangle \end{array}}{(\mathtt{t}',C.\overline{C}),P\|t_s\|t_r,\sigma \stackrel{MCom}{\Longrightarrow} (\mathtt{t}',\overline{C}),P\|t_{s'}\|t_{r'},\sigma}$$

Figure 6. Constraint matching is defined by four rules. Communication constraints are matched with threads performing the opposite communication action of the constraint.

```
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`.

4.6 Schedule Aware Partial Memoization

To address the limitations in the previous section, we define two new symmetric rules to pause and resume memoization (see Fig. 11). Pausing memoization (rule PAUSE MEMO) is similar to the rule END MEMO in Fig. 4 except the failed constraints are not discarded and the thread context is not given an expression to evaluate. Instead the thread retains its log of currently unsatisfied constraints which prevents its further evaluation. This effectively pauses the evaluation of this thread but allows regular threads to continue normal evaluation. Notice we only pause a thread utilizing memo information once it has correctly discharged its constraints. We could envision an alternative definition which pauses non-deterministically on any constraint and moves the non-discharged constraints back to the thread context which holds unsatisfied constraints. For the sake of simplicity we opted for *greedy* semantics which favors the utilization of memoization.

We can resume the paused thread, enabling it discharge other constraints using the rule RESUME MEMO, which begins constraint discharge anew for a paused thread. Thus, if a thread context has a set of constraints that were not previously satisfied and evaluation is not utilizing memo information, we can once again apply our $\Im$ function. Note that the use of memo information can be ended at any time (rule END MEMO can be applied instead of PAUSE MEMO). We can, therefore, change a thread in a paused state into a bona fide thread by first applying RESUME MEMO. If $\Im$ does not indicate we can discharge any additional constraints, we simple apply the rule END MEMO.

We also extend our evaluation rules to allow constraints to be matched against other constraints (rule MCOM). This is accomplished by matching constraints between two paused threads. Of course, it is possible that two threads, both of which were paused on a constraint that was not satisfiable may nonetheless satisfy one another. This happens when one thread is paused on a send constraint and another on a receive constraint both of which match on the channel and value. 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.

5. Soundness

We can relate the states produced by memoized evaluation to the states constructed by the non-memoizing evaluator. To do so, we first define a transformation function $\mathcal{T}$ that transforms process states (and terms) defined under memo evaluation to process states (and terms) defined under non-memoized evaluation (see Fig. 12). Since memo evaluation stores evaluation contexts in $\overline{\theta}$ they must be extracted and restored. This is done in the opposite order that they were pushed onto the stack $\overline{\theta}$ since the top represents the most recent call. Functions currently in the process of utilizing memo information must be resumed from the expression captured in the first non-discharged constraint. Similarly threads which are currently paused must also be resumed.

Our safety theorem ensures memoization does not yield states which could not be realized under non-memoized evaluation:

Theorem[Safety] If

$$P\|\langle\overline{\theta},\mathtt{t}[E[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]]\rangle,\sigma \stackrel{MemS.\overline{\beta_1}.MemE.\overline{\beta_2}}{\Longrightarrow} P'\|\langle\overline{\theta'},\mathtt{t}[E[\mathtt{v}']]\rangle,\sigma'$$

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

$$\mathcal{T}(P\|\langle\overline{\theta},\mathtt{t}[E[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]]\rangle) \stackrel{\alpha_1}{\longmapsto} \ldots \stackrel{\alpha_n}{\longmapsto} \mathcal{T}(P'\|\langle\overline{\theta'},\mathtt{t}[E[\mathtt{v}']]\rangle)$$

□

The proof [2] is by induction on the length of $\overline{\beta_1}.MemE.\overline{\beta_2}$. Each of the elements comprising $\overline{\beta_1}.MemE.\overline{\beta_2}$ corresponds to an action necessary to discharge previously recorded memoization constraints. We can show that every β step taken under memoization corresponds to some number of pure steps, and zero or one side-effecting steps under non-memoized evaluation: zero steps for returns and memo actions (e.g. MEMS, MEME, MEMP, and MEMR), and one step for core evaluation, and effectful actions (e.g., MCH, MSPAWN, MRECV, MSEND, and MCOM). Since a function which

[2] see `http://www.cs.purdue.edu/homes/lziarek/memoproof.pdf` for full details.

(PAUSE MEMO)

$$\frac{}{(\mathtt{t},\emptyset),P,\sigma \stackrel{MemP}{\Longrightarrow} P,\sigma}$$

(RESUME MEMO)

$$\frac{\Im(\overline{C},P)=\overline{C'} \quad \overline{C}=\overline{C''}.\overline{C'}}{P\|\langle\overline{\theta},\mathtt{t}_{\overline{C}}[E[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]]\rangle,\sigma \stackrel{MemR}{\Longrightarrow} (\mathtt{t},\overline{C''}),P\|\langle\overline{\theta},\mathtt{t}_{\overline{C'}}[E[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]]\rangle,\sigma}$$

(MCOM)

$$\frac{\begin{array}{c} C=((\mathtt{S},\mathtt{l},\mathtt{v}),_) \quad C'=((\mathtt{R},\mathtt{l},\mathtt{v}),_) \\ t_s=\langle\overline{\theta},\mathtt{t}_{\mathtt{C}.\overline{\mathtt{C}}}[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]\rangle \quad t_r=\langle\overline{\theta'},\mathtt{t}'_{\mathtt{C'}.\overline{\mathtt{C'}}}[\lambda_{\delta_1}\,\mathtt{x.e'}\,(\mathtt{v'})]\rangle \\ \overline{\theta},C\succ\overline{\theta''} \quad \overline{\theta'},C'\succ\overline{\theta'''} \\ t_{s'}=\langle\overline{\theta''},\mathtt{t}_{\overline{\mathtt{C}}}[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]\rangle \quad t_{r'}=\langle\overline{\theta'''},\mathtt{t}'_{\overline{\mathtt{C'}}}[\lambda_{\delta_1}\,\mathtt{x.e'}\,(\mathtt{v'})]\rangle \end{array}}{P\|t_s\|t_r,\sigma \stackrel{MCom}{\Longrightarrow} P\|t_{s'}\|t_{r'},\sigma}$$

Figure 11. Schedule Aware Partial Memoization.

$$\begin{array}{rcl}
\mathcal{T}((\mathtt{t},\overline{C}),P\|\langle\overline{\theta},\mathtt{t}_{\overline{C'}}[E[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]]\rangle) & = & \mathcal{T}(P\|\langle\overline{\theta},\mathtt{t}_{\overline{C}.\overline{C'}}[E[\lambda_\delta\,\mathtt{x.e}\,(\mathtt{v})]]\rangle) \\
\mathcal{T}((P_1\|P_2)) & = & \mathcal{T}(P_1)\|\mathcal{T}(P_2) \\
\mathcal{T}(\langle\overline{\theta},\mathtt{t}[\mathtt{e}]\rangle) & = & \mathtt{t}[\mathcal{T}(E_n[\ldots E_1[\mathtt{e}]])] \quad \theta_i=(\delta_i,v_i,E_i,\overline{C_i}) \in \overline{\theta} \\
\mathcal{T}(\langle\overline{\theta},\mathtt{t}_{(_,\mathtt{e'}).\overline{C}}[\mathtt{e}]\rangle) & = & \mathtt{t}[\mathcal{T}(E_n[\ldots E_1[\mathtt{e'}]])] \quad \theta_i=(\delta_i,v_i,E_i,\overline{C_i}) \in \overline{\theta} \\
\mathcal{T}(\lambda_\delta\,\mathtt{x.e}) & = & \lambda\,\mathtt{x.e} \\
\mathcal{T}(\mathtt{e}_1\,(\mathtt{e}_2)) & = & \mathcal{T}(\mathtt{e}_1)(\mathcal{T}(\mathtt{e}_2)) \\
\mathcal{T}(\mathtt{spawn}(\mathtt{e})) & = & \mathtt{spawn}(\mathcal{T}(\mathtt{e})) \\
\mathcal{T}(\mathtt{send}(\mathtt{e}_1,\mathtt{e}_2)) & = & \mathtt{send}(\mathcal{T}(\mathtt{e}_1),\mathcal{T}(\mathtt{e}_2)) \\
\mathcal{T}(\mathtt{recv}(\mathtt{e})) & = & \mathtt{recv}(\mathcal{T}(\mathtt{e})) \\
\mathtt{e} & & \text{otherwise}
\end{array}$$

Figure 12. $\mathcal{T}$ defines an erasure property on program states. The first four rules remove memo information and restore evaluation contexts.

is utilizing memoized information does not execute pure code (rule APP under $\longmapsto$), it may correspond to a number of APP transitions under $\longmapsto$.

6. Implementation

Our implementation is incorporated within Multi-MLton, an extension of MLton (MLton), a whole-program optimizing compiler for Standard ML, that provides support for parallel thread execution. The main extensions to Multi-MLton to support partial memoization involve insertion of read and write barriers to track accesses and updates of references, barriers to monitor function arguments and return values, and hooks to the Concurrent ML library to monitor channel based communication.

6.1 Multi-MLton

To support parallel execution, we modified the MLton compiler to support parallel threads. A POSIX pthread executes on each processor. Each pthread manages a lightweight Multi-MLton thread queue. Each pthread switches between lightweight MLton threads on its queue when it is preempted. Pthreads are spawned and managed by Multi-MLton's runtime. Currently, our implementation does not support migration of Multi-MLton threads to different thread queues.

The underlying garbage collector also supports parallel allocation. Associated with every processor is a private memory region used by threads it executes; allocation within this region requires no global synchronization. These regions are dynamic and growable. All pthreads must synchronize when garbage collection is triggered. Data shared by threads found on different processors are copied to a shared memory region that requires synchronization to access.

6.2 Parallel CML and hooks

Our parallel implementation of CML is based on Reppy's parallel model of CML (Reppy and Xiao 2008). We utilize low level locks implemented with compare and swap to provide guarded access to channels. Whenever a thread wishes to perform an action on a channel, it must first acquire the lock associated with the channel. Since a given thread may only utilize one channel at a time, there is no possibility of deadlock.

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

The constraint matching engine 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 Supporting Memoization

Any SML function can be annotated as a candidate for memoization. For such annotated functions, its arguments and return values at different call-sites, the communication it performs, and information about the threads it spawns are recorded in a memo table. Memoization information is logged through hooks to the CML runtime and stored by the underlying client code. In addition, to support partial memoization, the continuations of logged communication events are also saved.

Our memoization implementation extended CML channels to be aware of memoization constraints. Each channel structure contained a queue of constraints waiting to be solved on the channel. Because it will not 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. Our memoization technique relies on efficient equality tests, and approximate equality on reals and functions; the latter is modeled as closure equality.

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.

6.4 Benchmarks

We examined three benchmarks to measure the effectiveness of partial memoization in a parallel setting. The first benchmark is a streaming algorithm for approximating a k-clustering of points on a geometric plane. The second is a port of the STMBench7 benchmark (Guerraoui et al. 2007). STMBench7 utilizes channel based communication instead of shared memory and bears resemblance to the red-black tree program presented in Section 3. The third is Swerve (Ziarek et al. 2006), a highly-tuned webserver written in CML.

Similar to most streaming algorithms (Matthew Mccutchen and Khuller 2008), a k-clustering application defines a number of worker threads connected in a pipeline fashion. Each worker maintains a cluster of points that sit on a geometric plane. A stream generator creates a randomized data stream of points. A point is passed to the first worker in the pipeline. The worker computes a convex hull of its cluster to determine if a *smaller* cluster could be constructed from the newly received point. If the new point results in a smaller cluster, the outlier point from the original cluster is passed to the next worker thread. On the other hand, if the received point does not alter the configuration of the cluster, it is passed on to the next worker thread. The result defines an approximation of n clusters (where n is the number of workers) of size k (points that compose the cluster).

STMBench7 (Guerraoui et al. 2007), is a comprehensive, tunable multi-threaded benchmark designed to compare different STM implementations and designs. Based on the well-known 007 database benchmark (Carey et al. 1993), 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 leaves 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. Our port, besides translating the assembly tree to use a CML-based server abstraction (as discussed in Section 3), also involved building an STM implementation to support atomic sections, loosely based on the techniques described in (Saha et al. 2006; Adl-Tabatabai et al. 2006). 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 communications with various server nodes.

Swerve is a web server written entirely in CML. It consists of a collection of modules that communicate using CML's message-passing operations. There are three critical modules of interest: (a) a *listener* that processes incoming requests; (b) a *file processor* that handles access to the underlying file system; and, (c) a *timeout* manager that regulates the amount of time allocated to serve a request. There is a dedicated listener for every distinct client, and each request received by a listener is delegated to a server thread responsible for managing that request. Requested files are broken into chunks and packaged as messages sent back to the client.

7. Results

Our benchmarks were executed on a 16-way AMD Opteron 865 server with 8 processors, each containing two symmetric cores, and 32 GB of total memory, with each CPU having its own local memory of 4 GB. Access to non-local memory is mediated by a hyper-transport layer that coordinates memory requests between processors. To measure the effectiveness of our memoization technique, we executed two configurations (one memoized, and the other non-memoized) of our k-clustering algorithm, STMBench7, and Swerve, and measured overheads and performance by averaging results over ten executions. Non-memoized executions utilized a clean version of Multi-MLton without memoization hooks and barriers.

The k-clustering algorithm utilizes memoization to avoid redundant computations based on previously witnessed points as well as redundant computations of clusters. For STMBench7 the *non-memoized* configuration uses our STM implementation without any memoization where as the memoized configuration implements partial memoization of aborted transactions. Swerve uses memoization to avoid reading and processing previously requested files from disk.

For k-clustering, we computed 16 clusters of size 60 out of a stream of 10K randomly generated points. This resulted in the creation of 16 workers threads, one stream generating thread, and a

sink thread which aggregates the computation results. 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. STMBench7 creates a number of threads proportional to the number of nodes in the underlying data structure; this is roughly 400K threads for our configuration. Our experiments on Swerve were executed using the default server configuration and were exercised using HTTPerf, a well known tool for measuring webserver performance.

The benchmarks represent three very different programming models – pipeline stream-based parallelism (*k*-clustering), dynamically established communication links (Swerve), and software transactions (STM-Bench7), and leverage different executions models – *k*-clustering makes use of long-lived worker-threads while STMBench7 utilizes many lightweight server threads. Swerve utilizes both lightweight server threads in conjunction with long-lived worker threads. Each run of the benchmarks have execution times that range between 1 and 3 minutes.

For *k*-clustering we varied the number of repeated points generated by the stream. Configurations in which there is a high degree of repeated points offer the best performance gain (see Fig. 13(b)). For example, an input in which 50% of the input points are repeated yields roughly 50% performance gain. However, we also observe roughly 17% performance improvement even when *all* points are randomized. This is because the cluster's convex hull shrinks as the points which comprise the cluster become geometrically compact. Thus, as the convex hull of the cluster shrinks, the likelihood of a random point being contained within the convex hull of the cluster is reduced. Memoization can take advantage of this phenomenon by avoiding recomputation of the convex hull as soon as it can be determined that the input point resides outside the current cluster. Although we do not envision workloads that have high degrees of repeatability, memoization nonetheless leads to a 30% performance gain on a workload in which only 10% of the inputs are repeated.

In STMbench7, the utility of memoization is closely correlated to the number and frequency of aborted transactions. Our tests varied the read-only/read-write ratio (see Fig. 13(a)) within transactions. 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 (because of conflicts due to concurrent writes) offer potential opportunities for memoization. Thus, the cost to support memoization is seen when there are 100% read-only transactions; in this case, the overhead is roughly 11%. These overheads arise because of the cost to capture memo information (storing arguments, continuations, etc) and the cost associated with trying to utilize the memo information (discharging constraints).

Notice that as the number of transactions which perform modifications to the underlying data-structure increases so do memoization gains. For example, when the percentage of read-only transactions is 60%, we see an 18% improvement in runtime performance compared to a non-memoizing implementation for STMBench7.

We expected to see roughly a linear trend correlated to the increase in transactions which perform an update. However, we noticed that performance degrades about 10% from a read/write ratio of 20 to a read/write ratio of zero. This phenomenon occurs because memoized transactions are more likely to complete on their first try when there are fewer modifications to the structure. Since a non-memoized transaction requires longer to complete, it has a greater chance of aborting when there are frequent updates. This difference becomes muted as the number of changes to the data structure increase.

In Swerve, we observe an increase in performance correlated to the size of the file being requested by HTTPerf (see Fig. 13(c)); performance gains are capped at roughly 80% for file sizes greater than 8 MB. For each requested file, we build constraints corresponding to the file chunks read from disk. As long as no errors are encountered, the Swerve file processor sends the file chunks to be processed into HTTP packets by another module. After each chunk has been read from disk the file processor polls other modules for timeouts and other error conditions. If an error is encountered, subsequent file processing is stopped and the request is terminated. Partial memoization is particularly well suited for Swerve's file processing semantics because control is reverted to the error handling mechanism precisely at the point an error is detected. This corresponds to a failed constraint.

For all benchmarks, memory overheads are proportional to cache sizes and averaged roughly 15% for caches of size eight. The cache size defines the number of different memoized calls for a function maintined; thus a cache size of eight means that every memoized function records effects for eight different arguments.

8. Related Work

Memoization, or function caching (Liu and Teitelbaum 1995; Pugh 1988; Heydon et al. 2000; Swadi et al. 2006), 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 (Pickett and Verbrugge 2005). 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 (Qadeer et al. 2004). 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.

Recently software transactions (Harris and Fraser 2003; Saha et al. 2006) have emerged as a new method to safely control concurrent execution. There has also been work on applying these techniques to a functional programming setting (Harris et al. 2005; Ringenburg and Grossman 2005). These proposals usually rely on an *optimistic* concurrency control model that checks for serializability violations prior to committing the transaction, aborting when a violation is detected. Our benchmark results suggest that partial memoization can help reduce the overheads of aborting optimistic transactions.

Self adjusting mechanisms (Acar et al. 2008; Ley-Wild et al. 2008) 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. There has also been recent work on using change propagation in a parallel setting (Hammer et al. 2007). The programming model assumes fork/join parallelism, and is therefore not suitable for the kinds of contexts we consider. We believe our memoization technique is synergistic with current self-adjusting techniques and can be leveraged along with self-adjusting computation to create self-adjusting programs which utilize message passing.

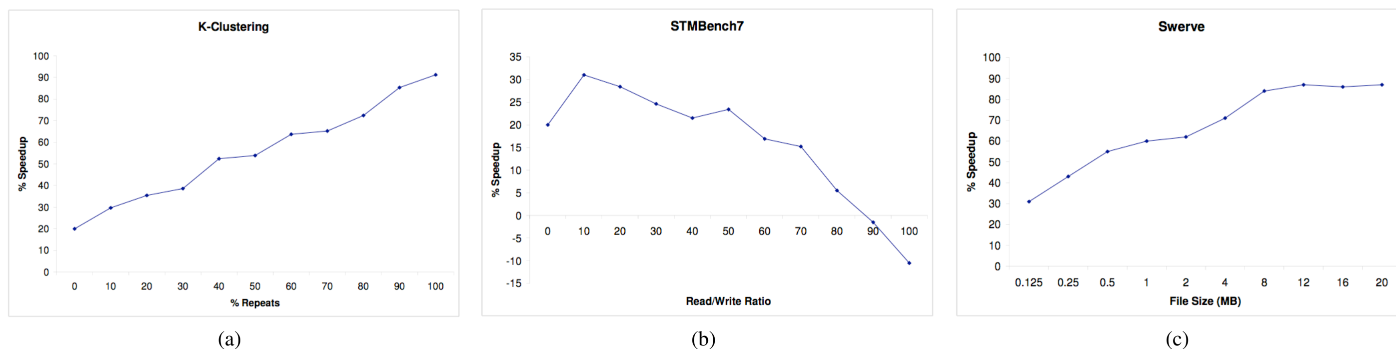

Figure 13. Normalized runtime % speedup for k-clustering, STMBench7, and Swerve benchmarks of memoized evaluation compared to non-memoized execution.

Our technique also shares some similarity with transactional events (Donnelly and Fluet 2006; Effinger-Dean et al. 2008). Transactional events explore a state space of possible communications finding matching communications to ensure atomicity of a collection of actions. To do so transactional events require arbitrary lookahead in evaluation to determine if a complex composed event can commit. Partial memoization also explores potential matching communication actions to satisfy memoization constraints. However, partial memoization avoids the need for arbitrary lookahead – failure to discharge memoization constraints simply causes execution to proceed as normal.

Acknowledgements

Thanks to Matthew Fluet and Dan Spoonhower for their help in the design and development of Multi-MLton. This work is supported by the National Science Foundation under grants CCF-0701832 and CCF-0811631, and an Intel graduate fellowship.

References

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective Memoization. In *POPL*, pages 14–25, 2003.

Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative Self-Adjusting Computation. In *POPL*, pages 309–322, 2008.

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.

Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The 007 benchmark. *SIGMOD Record*, 22(2):12–21, 1993.

Kevin Donnelly and Matthew Fluet. Transactional Events. In *ICFP*, pages 124–135, 2006.

Laura Effinger-Dean, Matthew Kehrt, and Dan Grossman. Transactional events for ml. In *ICFP '08*, pages 103–114, 2008. ISBN 978-1-59593-919-7.

Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *ASPLOS-XII*, pages 151–162, 2006.

Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *EuroSys*, pages 315–324, 2007.

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.

Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402, 2003.

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.

Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *PLDI*, pages 311–320, 2000.

Ruy Ley-Wild, Matthew Fluet, and Umut A. Acar. Compiling self-adjusting programs with continuations. In *ICFP*, pages 321–334, 2008.

Yanhong A. Liu and Tim Teitelbaum. Caching Intermediate Results for Program Improvement. In *PEPM*, pages 190–201, 1995.

Richard Matthew Mccutchen and Samir Khuller. Streaming algorithms for k-center clustering with outliers and with anonymity. In *APPROX '08 / RANDOM '08*, pages 165–178, 2008.

MLton. http://www.mlton.org.

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.

W. Pugh and T. Teitelbaum. Incremental Computation via Function Caching. In *POPL*, pages 315–328, 1989.

William Pugh. An Improved Replacement Strategy for Function Caching. In *LFP*, pages 269–276, 1988.

Shaz Qadeer, Sriram K. Rajamani, and Jakob Rehof. Summarizing procedures in concurrent programs. In *POPL*, pages 245–255, 2004.

John Reppy and Yingqi Xiao. Towards a Parallel Implementation of Concurrent ML. In *DAMP 2008*, January 2008.

John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

Michael F. Ringenburg and Dan Grossman. AtomCaml: First-Class Atomicity via Rollback. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 92–104, 2005.

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.

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.

Lukasz Ziarek, Philip Schatz, and Suresh Jagannathan. Stabilizers: A Modular Checkpointing Abstraction for Concurrent Functional Programs. In *ACM International Conference on Functional Programming*, pages 136–147, 2006.