2007

Memoizing Communication

Lukasz Ziarek
Purdue University, lziarek@cs.purdue.edu

Jeremy Orlow

Suresh Jagannathan
Purdue University, suresh@cs.purdue.edu

Report Number:
07-019

MEMOIZING COMMUNICATION

Lukasz Ziarek
Jeremy Orlow
Suresh Jagannathan

Department of Computer Science
Purdue University
West Lafayette, IN 47907

CSD TR #07-019
July 2007

Memoizing Communication

Lukasz Ziarek Jeremy Orlow Suresh Jagannathan

Department of Computer Science
Purdue University
{lziarek,jorlow,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 these communication events. Subsequent calls to f can be avoided only if we can identify an interleaving of actions from these call-sites that lead to states in which these dependencies are satisfied. Formulating the issues necessary to discover these interleavings is the focus of this paper.

Specifically, we consider the memoization problem for Concurrent ML (20), in which threads may communicate with one another through synchronous message-based communication. Besides formalizing the ideas underlying memoization in this context, we also consider a realistic case study that uses memoization to reduce re-execution overheads for aborted transactions in a transaction-aware CML extension. Our benchmark results indicate that memoization-based optimizations can lead to substantial reduction in re-execution costs for long-lived transactions (up to 43% on some benchmarks), without incurring high memory overheads.

1. Introduction

Eliminating redundant computation is an important optimization supported by many language implementations. One important instance of this optimization class is memoization (15; 17; 3), a well-known dynamic technique that can be used 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 avoided, 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 used to produce its result are different at the two call-sites.

Concurrency adds further 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 and non-determinism introduced by scheduling decisions, however, making such conclusions is non-trivial.

Nonetheless, we believe memoization can be an important component in a concurrent programming language runtime. For instance, memoization can allow the computation performed by threads in stream or pipeline-based concurrent programs (8) to be optimized to avoid re-computing outputs for previously seen inputs. As another example, concurrency abstractions built using transactions or speculation typically 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 (9), their effects are typically 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. (13). 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 memoization is safe, given the possibility of internal concurrency, communication, and synchronization among threads created by the transaction.

In this paper, we consider the memoization problem for pure CML (20), a concurrent message-passing dialect of ML that supports first-class synchronous events. 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 for CML 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.

Besides providing a formal characterization of memoization in this context, we also present a detailed performance evaluation of our implementation. We use as a case study, a transaction-aware extension of CML that supports multi-threaded transactions[1]. Our benchmark is STMBench7 (10), a highly tunable benchmark for measuring transaction overheads, re-written to leverage CML synchronous communication. Our results indicate that memoization can lead to substantial performance gains, in some cases in excess of 43% improvement in execution time compared with an implementation that performs no memoization, with only modest increases in memory overhead (15% on average). To our knowledge, this is the first attempt to formalize the memoization problem for CML, and to provide an empirical evaluation of its impact on improving performance for intensive multi-threaded workloads.

The paper is organized as follows. Motivation for the problem is given in Section 2. The formalization of our approach, semantics, and definition of partial memoization are presented in Section 3 and Section 4. A detailed description of our implementation, benchmarks, and results are given in Sections 5, 6, and 6.4. We discuss previous work and provide conclusions in Section 7.

2. Programming Model and Motivation

Our programming model is pure CML (20), a message-passing dialect of ML with support for first-class synchronous events, and dynamic thread creation. 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 formulation does not consider references, although they can be effectively encoded using CML message-passing primitives. We also do not consider selective memoization techniques (3) to record precise dependencies within memoized functions to reduce memoization overheads; incorporating these mechanisms into our framework pose no additional complications.

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, threads it spawns, and the return value it produces. Thus, the memoized return value of a call to a function f can be used 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; and (d) 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.

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 involves a systematic exploration of the execution state space to induce 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.

[1] A multi-thread transaction is a transaction composed of multiple threads, all of whose lifetimes are bounded by the transaction length. The transaction is responsible for managing its component threads.

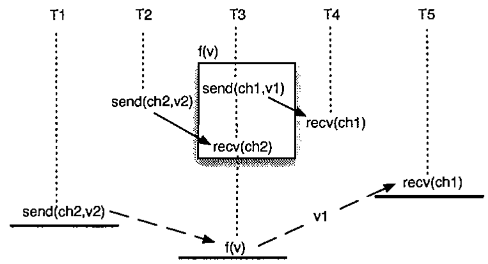


Figure 1. A CML program consists of a collection of threads that may communicate with one another via synchronous message passing. In the figure, the first call to `f` by thread `T3` results in a communication action between `T2` and `T3` via channel `c2`, and `T3` and `T4` via channel `c1`. By memoizing this information, we can avoid performing non-effectful computation in the second application. Note that in the second call, threads `T1` and `T5` are available to satisfy `f`'s communication actions. Rather than performing the second call in its entirety, we can immediately return the value yielded by the first, producing a new global state in which value `v2` is removed from channel `ch2` and value `v1` is deposited on channel `ch1` and consumed by `T5`, thus allowing threads `T1`, `T3`, and `T5` to proceed. The dashed lines indicate the communication actions that must be satisfied to avoid the second call. Note that the second call to `f` entails communication actions with threads different from the first.

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

Figure 2. By memoizing the first call to `f`, we can avoid evaluating the pure computation abstracted by "..." in `f`'s body in the second since there is only a single receiver on channel `c1`.

2.1 Tracking Communication Actions

A key requirement for effective memoization of CML function applications is the ability to track communication actions performed by memoized functions. Provided that the global state would permit these same actions to succeed if a function is re-executed with the same inputs, memoization can be employed to avoid

Consider the example code fragment, presented in Fig. 2, that spawns a thread to execute function `g` and performs two calls to `f`. The first call to `f` sends `v1` on channel `c1`; the only receiver for this message is `g`, which consumes `v1`, and sends `v2` on channel `c2`. If the global state at the point when the second call is performed has `g` waiting to receive on `c1`, the pure computation performed by the function (operations other than the `send`) can be avoided. Instead of performing the call, we can simply return `f`'s result, and deposit `v1` on channel `c1`, knowing that there is a waiting receiver. Note, however, that at the point the call is performed, the recursive invocation of `g` may not have taken place, and thus there may be no waiting receiver on `c1`. To safely avoid re-evaluating `f()`, we must delay the application at this call point until the thread computing `g` can proceed to the receive action on `c1`. Indeed, if

```
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())
in (spawn(g); spawn(h); f(); ...; f())
end
```

Figure 3. Because there may be multiple possible interleavings that pair synchronous communication actions among concurrently executing threads, leveraging memoization requires dynamically tracking these events.

the recursive call never takes place, it would be incorrect to use the memoized return value for `f()` since the second call would normally have blocked on the send operation in the absence of an accepting receiver.

Unfortunately, reasoning about whether an application can leverage memoized information is usually more difficult. Fig. 3 presents a slightly modified version of the program shown in Fig. 2 that introduces an auxiliary function `h`. Procedure `f` communicates with `g` via channel `c1`. It also either receives value `v2` or `v3` from `h` depending upon its interleaving with `g`. Suppose that in the first call to `f()`, `f` receives `v3` on `c2` because `g` consumed `v2`. We can avoid performing the pure computation in the body of `f` in the second call if the interleaving among these threads is such that `v2` is consumed by a subsequent recursive call of `g`, allowing the send of `v3` by `h` on channel `c2` to be paired with the receive by `f`. In this case, the value `v3` can be (implicitly) consumed, allowing `h` to proceed, and the memoized return value of `f` can be used as the result of the call. Thus, deciding whether memoized information can be used to avoid performing the second call to `f` requires reasoning about the interactions between `h` and `g`, and may involve identifying a specific schedule to ensure synchronous operations in `f` can be satisfied at the second call, and mirror their behavior under the memoized execution.

Notice that if `v2` and `v3` are equal, the receive in `f` can be paired with either send in `h`. Thus, we can exploit memoization under a different interleaving of threads, and need not require that all communication actions within the function be paired identically as in the original evaluation.

3. Semantics

Our semantics is defined in terms of a core call-by-value functional language with threading and communication primitives (see Fig. 4). 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 language.

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 ϕ to denote an empty sequence. Metavariables x and y range over variables, t ranges over threads, l ranges over channels, v ranges 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 send[2]. Spawning an expression (that evaluates to a thunk) creates a new thread in which the application of the thunk is performed.

3.1 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 channels. We do not consider references in this core language as they can be modeled in terms of operations on channels (20).

A thread context $\langle t_P, E[\mathtt{e}]\rangle$ denotes an expression e available for execution by thread $\mathtt{t} \in P$ within context E. Local reductions within a thread are specified by an auxiliary relation, $\mathtt{e} \rightarrow \mathtt{e}'$, that evaluates expression e within some thread to a new expression $\mathtt{e}'$. The local evaluation rules are standard: channel creation results in the creation of a new location that acts as a container for message transmission and receipt, and application substitutes the argument value for free occurrences of the parameter in the body of the abstraction.

Global evaluation is specified via a relation ($\longmapsto$) that maps a program state (P) to another program state. We write $\longmapsto^*$ to denote the reflexive, transitive closure of this relation. An evaluation step is marked with a tag (or sequence of tags) that indicates the action (or sequence of actions) performed by that step.

The global actions of interest are those that involve spawn and communication events. A spawn action, given by the SPAWN rule, 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 (given by 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.

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

If function f calls function g, then actions performed by g must be satisfiable in any memoization of f. For example, if g performs a synchronous communication action s, and we encounter an application of f after it has been memoized, then s must be satisfiable at that state to avoid performing the call. We therefore associate a call stack of constraints ($\overline{\theta}$) with each thread that defines

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

SYNTAX:

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

EVALUATION CONTEXTS:

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

$$\langle \mathtt{t}_P, E[\mathtt{e}]\rangle \quad ::= \quad P \| \mathtt{t}[E[\mathtt{e}]]$$

PROGRAM STATES:

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

LOCAL EVALUATION:

$$\frac{\mathtt{e} \rightarrow \mathtt{e}'}{\langle \mathtt{t}_P, E[\mathtt{e}]\rangle \overset{\mathit{Local}}{\longmapsto} \langle \mathtt{t}_P, E[\mathtt{e}']\rangle}$$

$$\frac{\mathtt{l}\ \ \text{fresh}}{\mathtt{mkCh}() \rightarrow \mathtt{l}} \qquad\qquad (\lambda\,\mathtt{x}.\mathtt{e})\ \mathtt{v} \rightarrow \mathtt{e}[\mathtt{v}/\mathtt{x}]$$

GLOBAL EVALUATION:

(SPAWN)

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

(COMM)

$$\frac{\begin{array}{c}\mathtt{t}'\text{fresh} \\ P = P' \| \mathtt{t}[E[\mathtt{send}(\mathtt{l},\mathtt{v})]] \| \mathtt{t}'[E'[\mathtt{recv}(\mathtt{l})]]\end{array}}{P \overset{\mathit{Com}}{\longmapsto} P' \| \mathtt{t}[E[\mathtt{unit}]] \| \mathtt{t}'[E'[\mathtt{v}]]}$$

Figure 4. A concurrent language with synchronous communication.

SYNTAX:

$$
\begin{array}{rcl}
P & ::= & P \| P \mid \langle \overline{\theta}, \overline{C}, \mathtt{t}[\mathtt{e}]\rangle \\
\mathtt{e} \in \mathit{Exp} & ::= & \ldots \mid \mathtt{B}(\mathtt{v},\mathtt{e}) \mid \mathtt{U}(\mathtt{e}) \\
\mathtt{v} \in \mathit{Val} & ::= & \mathtt{unit} \mid \lambda_\delta\,\mathtt{x}.\mathtt{e} \mid \mathtt{l}
\end{array}
$$

EVALUATION CONTEXTS:

$$E ::= \ldots \mid \mathtt{B}(v, E)$$

$$\langle \mathtt{t}_P, \overline{\theta}, \overline{C}, E[\mathtt{e}]\rangle ::= P \| \langle \overline{\theta}, \overline{C}, \mathtt{t}[E[\mathtt{e}]]\rangle$$

PROGRAM STATES:

$$
\begin{array}{rcll}
\delta & \in & \mathit{MemoId} & \\
C & \in & \mathit{Constraint} = & (\mathit{Loc} \times \mathit{Val} \times \{\mathtt{R},\mathtt{S}\} \times \mathit{Exp}) + (\mathtt{Sp} \times \mathit{Exp}) \\
& & & (\mathtt{Ch} \times \mathit{Channel}) \\
\sigma & \in & \mathit{MemoStore} = & \mathit{MemoId} \times \mathit{Val} \rightarrow \mathit{Constraint}^* \times \mathit{Val} \\
\theta & \in & \mathit{MemoState} = & \mathit{MemoId} \times \mathit{Constraint}^* \\
\alpha,\beta & \in & \mathit{Tag} = & \{\mathit{Ch}, \mathit{Spn}, \mathit{Com}, \mathit{MCom}, \mathit{Fun}, \mathit{App}, \mathit{Ret}, \\
& & & \mathit{MCh}, \mathit{MSp}, \mathit{MRet}, \mathit{Mem}, \mathit{Fail}, \mathit{PMem}\}
\end{array}
$$

CONSTRAINT ADDITION:

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

(CHANNEL)

$$\frac{\begin{array}{c}\delta\ \text{fresh} \quad \overline{\theta}, (\mathtt{Ch}, \mathtt{l}) \succ \overline{\theta'} \\ p_a = \langle \mathtt{t}_P, \overline{\theta}, \phi, E[\mathtt{mkCh}()]\rangle \\ p_b = \langle \mathtt{t}_P, \overline{\theta'}, \phi, E[\mathtt{l}]\rangle\end{array}}{p_a, \sigma \overset{\mathit{Ch}}{\Longrightarrow} p_b, \sigma}$$

(FUN)

$$\frac{\begin{array}{c}\delta\mathit{fresh} \\ p_a = \langle \mathtt{t}_P, \overline{\theta}, \phi, E[\lambda\,\mathtt{x}.\mathtt{e}]\rangle \\ p_b = \langle \mathtt{t}_P, \overline{\theta}, \phi, E[\lambda_\delta\,\mathtt{x}.\mathtt{e}]\rangle\end{array}}{p_a, \sigma \overset{\mathit{Fun}}{\Longrightarrow} p_b, \sigma}$$

(APP)

$$\frac{\begin{array}{c}(\delta, \mathtt{v}) \notin \mathit{Dom}(\sigma) \\ p_a = \langle \mathtt{t}_P, \overline{\theta}, \phi, E[(\lambda_\delta\,\mathtt{x}.\mathtt{e})\,\mathtt{v}]\rangle \\ p_b = \langle \mathtt{t}_P, (\delta, \phi).\overline{\theta}, \phi, E[\mathtt{B}(\mathtt{v}, \mathtt{e}[\mathtt{v}/\mathtt{x}])]\rangle\end{array}}{p_a, \sigma \overset{\mathit{App}}{\Longrightarrow} p_m, \sigma}$$

(SPAWN)

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

(COMM)

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

(RET)

$$\frac{\theta = (\delta, \overline{C})}{\langle \mathtt{t}_P, \theta.\overline{\theta}, \phi, E[\mathtt{B}(\mathtt{v}, \mathtt{v}')]\rangle, \sigma \overset{\mathit{Ret}}{\Longrightarrow} \langle \mathtt{t}_P, \overline{\theta}, \phi, E[\mathtt{v}']\rangle, \sigma[(\delta, \mathtt{v}) \mapsto (\overline{C}, \mathtt{v}')]}$$

(MEMO APP)

$$\frac{\begin{array}{c}(\delta, \mathtt{v}) \in \mathit{Dom}(\sigma) \\ \langle \mathtt{t}_P, \overline{\theta}, \phi, E[(\lambda_\delta\,\mathtt{x}.\mathtt{e})\,\mathtt{v}]\rangle, \sigma \overset{\overline{\alpha}.\mathit{Mem}}{\rightsquigarrow} P', \sigma'\end{array}}{\langle \mathtt{t}_P, \overline{\theta}, \phi, E[(\lambda_\delta\,\mathtt{x}.\mathtt{e})\,\mathtt{v}]\rangle, \sigma \overset{\overline{\alpha}.\mathit{Mem}}{\Longrightarrow} P', \sigma'}$$

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

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). In addition, when a function f is applied to argument v, and there exists no previous invocation of f to v, 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 state ($\overline{\theta}$) associated with the thread in which the application occurs. Note that all the rules in this figure assume an empty constraint sequence (ϕ); these rules deal with ordinary expression evaluation, and are not responsible for discharging memoization constraints on applications of a previously memoized call. 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.

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 reason for this last component is explained in Section 3.4.) For a spawn operation, the constraint records the action (`Sp`) and the expression being spawned. For a channel creation operation, the constraint records the location of the channel.

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, along with its return value, are recorded in the memo store (rule RET).

The most interesting rule is the one that deals with determining whether 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.

To determine whether the global state permits the discharge of all constraints associated with the call, we employ an auxiliary evaluation relation ($\rightsquigarrow$) shown in Fig. 6. Our formulation attempts to memoize any application whose evaluation with the supplied argument has already been recorded in the memo store. The $\rightsquigarrow$ relation is well-defined only if all necessary memoization constraints are satisfiable. It acts as an *oracle* that examines all possible transitions from the current global state, attempting to find an execution path in which all constraints necessary to ensure the call can be elided are discharged.

The states examined may contain function expressions ($\overset{Fun}{\rightsquigarrow}$), spawn expressions ready to create new threads ($\overset{Spn}{\rightsquigarrow}$), channel expressions that create new channels ($\overset{Ch}{\rightsquigarrow}$), synchronous communication actions ready to be paired ($\overset{Com}{\rightsquigarrow}$), applications that can be tracked for memoization ($\overset{App}{\rightsquigarrow}$), applications of memoized functions that can be elided ($\overset{Mem}{\rightsquigarrow}$), and return values of applications ($\overset{Ret}{\rightsquigarrow}$). These rules are identical to the definitions defined in Fig. 5.

To utilize memo evaluation, the constraints associated with a memoized function applied to the same argument found in the memo store are added to the thread context (rule MEMO). Evaluation is complete when there are no more constraints left to examine. The application is tagged with a `U` wrapper (for *use* memo) to identify it as a potential beneficiary of previously recorded memo information. Since it leverages the definition of $\Longrightarrow$, memo evaluation is also defined by non-deterministic interleaving of the actions performed by different threads. Evaluation is well-defined provided that there is at least one such interleaving in which all constraints of the memoization candidate can be satisfied. Evaluation enters a stuck state if no such interleaving exists.

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 `(Ch,l)` (rule MCH) creates a new channel location `l'`, and replaces all occurrences of `l` found in the remaining constraint sequence for this thread with `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.

There are three communication constraint matching rules ($\overset{MCom}{\rightsquigarrow}$). If the current constraint expects to receive value `v` on channel `l`, and there exists a thread able to send `v` on `l`, evaluation proceeds to a state in which the communication succeeds, 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 that the value.

The global state may also 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 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.

In the course of determining whether an application can leverage a previous memo, expressions may be evaluated that lead to

MEMO CORE EVALUATION

$$\frac{\begin{array}{c}\alpha \in \{Ch, Spn, Ret, Com, Fun, Mem, App\} \\ P, \sigma \overset{\alpha}{\Longrightarrow} P', \sigma'\end{array}}{P, \sigma \overset{\alpha}{\leadsto} P', \sigma'}$$

MEMO:

$$\frac{\begin{array}{c}\sigma(\delta, \mathtt{v}) = (\overline{C}, \mathtt{v}') \\ \langle \mathtt{t}_P, \overline{\theta}, \overline{C}, E[\mathtt{U}((\lambda_\delta\, \mathtt{x.e})\, \mathtt{v})]\rangle, \sigma \overset{\overline{\alpha}}{\leadsto}{}^{*} P' \| \langle \overline{\theta'}, \phi, \mathtt{t}[E[\mathtt{v}']]\rangle, \sigma'\end{array}}{\langle \mathtt{t}_P, \overline{\theta}, \phi, E[(\lambda_\delta\, \mathtt{x.e})\, \mathtt{v}]\rangle, \sigma \overset{\overline{\alpha}.Mem}{\leadsto} P' \| \langle \overline{\theta'}, \phi, \mathtt{t}[E[\mathtt{v}']]\rangle, \sigma'}$$

(MCH)

$$\frac{\begin{array}{c}C = (\mathtt{Ch}, \mathtt{l}) \quad \mathtt{l}'\text{fresh} \\ \overline{C'} = \overline{C}[\mathtt{l}'/\mathtt{l}]\end{array}}{\langle \overline{\theta}, C.\overline{C}, \mathtt{t}[\mathtt{e}]\rangle, \sigma \overset{MCh}{\leadsto} \langle \overline{\theta}, \overline{C'}, \mathtt{t}[\mathtt{e}]\rangle, \sigma}$$

(MSPAWN)

$$\frac{C = (\mathtt{Sp}, \mathtt{e}') \quad \mathtt{t}'\mathit{fresh} \quad \overline{\theta}, C \succ \overline{\theta''}}{\langle \mathtt{t}_P, \overline{\theta}, C.\overline{C}, E[\mathtt{e}]\rangle, \sigma \overset{MSp}{\leadsto} P \| \langle \overline{\theta}, \overline{C'}, \mathtt{t}[E[\mathtt{e}]]\rangle \| \langle \phi, \phi, \mathtt{t}'[\mathtt{U}(\mathtt{e}')]\rangle, \sigma}$$

(MRECV)

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

(MSEND)

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

(MCOM)

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

(MRET)

$$\frac{\sigma(\delta, \mathtt{v}) = (\overline{C}, \mathtt{v}')}{\langle \mathtt{t}_P, \overline{\theta}, \phi, E[\mathtt{U}((\lambda_\delta\, \mathtt{x.e})\, \mathtt{v})]\rangle, \sigma \overset{MRet}{\leadsto} \langle \mathtt{t}_P, \overline{\theta}, \phi, E[\mathtt{v}']\rangle, \sigma}$$

Figure 6. Memoization can be expressed via a set of constraints associated with different calls, and an exploration of possible interleavings whose execution would allow these constraints to be satisfied.

new states in which existing constraints can be satisfied. If all constraints are satisfied, evaluation yields a new global state that safely permits the result value previously recorded in the memo store to be returned (rule MRET).

3.3 Example

To illustrate how memo evaluation works, consider the program shown in Fig. 7. The program consists of two recursive functions, f′ and g′, which exchange data over a shared channel l. Although calls to f′ and g′ cannot be memoized since their execution does not terminate, calls to f and g can be memoized when both sends and receives are suitably paired. When invoked, f may receive any of four possible combinations of values on channel ch: (a) 1 followed by 1, (b) 1 followed by 2, (c) 2 followed by 1, or (d) 2 followed by 2. These possibilities reflect the different thread interleavings possible for the different thread instantiations of g by g'.

Thus, there are four possible memoized versions of f, one for each pair of values that the function may receive. Notice that for every call to a memoized version of g(), there exist a sequence of evaluation steps that leads to a state in which its constraints can be satisfied. This is due to the fact that there will always be a matching receive (provided by the recursive calls of f') for every send g performs. Thus, because memo evaluation performs an exhaustive state space search, it is guaranteed to find an interleaving among the various threads evaluating g() that satisfies the constraints for the original memoized version of f() for all its subsequent calls. For example, suppose f initially received values 1 and 2 (in that order) on ch. Subsequent calls to g() can be memoized by ensuring the global state has an application of f waiting to receive 1 and 2; subsequent calls to f can be memoized by ensuring the global state has an application of g willing to send 1 and 2. These conditions can be satisfied through repeated use of the MCOM rule to discharge the paired communication constraints on recursive invocations of f() and g() based on their initial memoized execution.

We depict this characterization in the evaluation tree shown in Fig. 8. We omit unnecessary thread creation actions, and reason only about the order of sends produced by various incarnations of g. Although there could be many concurrent calls to f and g, the evaluation tree represents the abstract interleavings of communications that could satisfy f's constraints. The evaluation tree itself is defined recursively, due to the recursive definition of the program. For any given call to f there exist precisely four evaluation paths based on the combination of values it can receive.

3.4 Partial Memoization

The semantics defined thus far yields a global state in which memoization constraints are satisfied, if possible, and is not well-defined otherwise. An implementation of the semantics is also not scalable

(CORE)

$$\frac{\alpha \in \{Ch, Fun, Spn, Ret, Com, MCh, MCom, MRet, MSp, Local\} \qquad P,\sigma \overset{\alpha}{\rightsquigarrow} P',\sigma'}{P,\sigma \overset{\alpha}{\hookrightarrow} P',\sigma'}$$

(APP)

$$\frac{(\delta,\mathtt{v}) \notin Dom(\sigma) \qquad \langle \mathtt{t}_P, (\delta,\phi).\overline{\theta}, \phi, \mathtt{B}(\mathtt{v}, \mathtt{e}[\mathtt{v}/\mathtt{x}])\rangle \overset{\overline{\alpha}}{\Longrightarrow}{}^{*} \langle \mathtt{t}_P, \overline{\theta'}, \phi, \mathtt{B}(\mathtt{v},\mathtt{v}')\rangle, \sigma'}{\langle \mathtt{t}_P, \overline{\theta}, \phi, E[(\lambda_\delta\, \mathtt{x}.\mathtt{e})\, \mathtt{v}]\rangle, \sigma \overset{App.\overline{\alpha}}{\Longrightarrow}{}^{*} \langle \mathtt{t}_P, \overline{\theta'}, \phi, E[\mathtt{B}(\mathtt{v},\mathtt{v}')]\rangle, \sigma'}$$

(PARTIAL MEMO APP)

$$\frac{(\delta,\mathtt{v}) \in Dom(\sigma) \qquad \langle \mathtt{t}_P, \overline{\theta}, \phi, E[(\lambda_\delta\, \mathtt{x}.\mathtt{e})\, \mathtt{v}]\rangle, \sigma \overset{\overline{\alpha}.PMem}{\hookrightarrow} P',\sigma'}{\langle \mathtt{t}_P, \overline{\theta}, \phi, E[(\lambda_\delta\, \mathtt{x}.\mathtt{e})\, \mathtt{v}]\rangle, \sigma \overset{\overline{\alpha}.PMem}{\Longrightarrow} P',\sigma'}$$

(PARTIAL MEMO)

$$\frac{\beta \in \{Fail, MRet\} \qquad \sigma(\delta,\mathtt{v}) = (\overline{C},\mathtt{v}') \qquad \langle \mathtt{t}_P, \overline{\theta}, \overline{C}, E[\mathtt{U}((\lambda_\delta\, \mathtt{x}.\mathtt{e})\, \mathtt{v})]\rangle \overset{\overline{\alpha}.\beta}{\hookrightarrow}{}^{*} P' \| \langle \overline{\theta'}, \phi, \mathtt{t}[E[\mathtt{e}']]\rangle, \sigma'}{\langle \mathtt{t}_P, \overline{\theta}, \phi, E[(\lambda_\delta\, \mathtt{x}.\mathtt{e})\, \mathtt{v}]\rangle, \sigma \overset{\overline{\alpha}.\beta.PMem}{\hookrightarrow} P' \| \langle \overline{\theta'}, \phi, \mathtt{t}[E[\mathtt{e}']]\rangle, \sigma'}$$

(FAIL)

$$\frac{\gamma \in \{\mathtt{S},\mathtt{R}\} \qquad C = \langle \_, \_, \gamma, \mathtt{e}'\rangle}{\langle \mathtt{t}_P, \overline{\theta}, C.\overline{C}, E[\mathtt{U}((\lambda_\delta\, \mathtt{x}.\mathtt{e})\, \mathtt{v})]\rangle, \sigma \overset{Fail}{\hookrightarrow} \langle \mathtt{t}_P, \overline{\theta}, \phi, E[\mathtt{e}']\rangle, \sigma}$$

FIGURE 9. PARTIAL MEMOIZATION.

```
let val l = mkCh()
    fun f() = let val _ = recv(ch)
                  val _ = recv(ch)
              in ()
              end
    fun g() = let val _ = send(ch, 1)
                  val _ = send(ch, 2)
              in ()
              end
    fun f'() = spawn(f); f'()
    fun g'() = spawn(g); g'()
in spawn(f'); spawn(g')
end
```

Figure 7. Example evaluation for recursive functions $\mathtt{f}'$ and $\mathtt{g}'$.

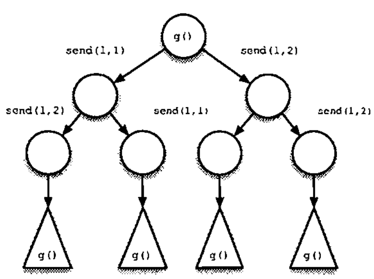


Figure 8. The memo evaluation tree for the example code presented in Fig. 7

as it must perform arbitrary state space exploration to determine memoization feasibility.

We therefore wish to determine whether a call can be memoized without performing a potentially unbounded number of evaluation steps. Even if it is not readily possible to determine if all constraints necessary to memoize an application can be satisfied, it may be possible to determine that some prefix of the constraint sequence can be discharged. Partial memoization allows us to avoid re-executing those actions associated with this prefix.

```
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 10. Determining if an application can be memoized may require examining an arbitrary number of possible thread interleavings.

Consider the example presented in Fig 10. The program fragment 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`. Determining whether the receive constraint associated with `f` can be matched requires more work. To match constraints properly, we need to force a schedule that causes `g` to receive the first send by `h` and `f` to receive the second, causing execution of the thread executing `i()` to block until the second call to `f` completes.

Fixing such a schedule is tantamount to examining an unbounded set of interleavings. Instead, we could leverage memo information for `f()` to avoid performing the send, and all computation upto the receive operation, and we could leverage memo information for `g()` to avoid performing the matching receive and all computation upto the receive on channel `c2`; these constraints are guaranteed to be satisfied when the second call to `f` is performed. Because the receive constraint for `f()` and `g()` on channel `c2` may not be immediately satisfiable at `f`'s second call, we can resume execution of `f()` and `g()` at their respective receive operations on `c2`.

Our partial memoization extension to the memo evaluation rules is presented in Fig. 9. These evaluation rules share much in common with the memo evaluation rules (see rule CORE). Channel and thread creation, function return, and synchronous communication operations behave as before; in particular, the constraints added to the memo store are unchanged. Function and channel evaluation are also unchanged.

The main difference arises in the way function application is treated. If an application of a function `f` to argument `v` has not been recorded in the memo store, it can be memoized (see Rule APP). Since subsequent calls to `f` with `v` may not be able to discharge all constraints, however, we need to record the program points for all communication actions within `f` that represent potential resumption points; these continuations are recorded as part of the stored constraint. 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 upto the return point of the function.

Rule PARTIAL MEMO determines whether an application of a function `f` to an argument `v` that has already been recorded in the memo store can utilize previously recorded memo information. Its structure is similar to the structure of memo evaluation shown in Fig. 6 except that it allows a non-deterministic *failure* action to be taken. As communication constraints are being discharged, the rules permit the installation of the partial continuation saved in the constraint tuple for that communication (expression `e'` in rule FAIL); no further constraints are examined. Thus, the thread performing this call will resume execution from the saved program point.

4. Safety, Efficiency, and Correspondence

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

$$\mathcal{T}((P_1 \| P_2), \sigma) = \mathcal{T}(P_1, \sigma) \| \mathcal{T}(P_2, \sigma)$$
$$\mathcal{T}(\langle \overline{\theta}, \overline{C}, \mathtt{e}\rangle, \sigma) = \mathcal{T}(\mathtt{e}, \sigma)$$

$$\begin{array}{rcl} \mathcal{T}(\lambda_\delta\, \mathtt{x.e}) & = & \lambda\, \mathtt{x.e} \\ \mathcal{T}((\mathtt{e}_1)\, \mathtt{e}_2) & = & \mathcal{T}(\mathtt{e}_1)(\mathcal{T}(\mathtt{e}_2)) \\ \mathcal{T}(\mathtt{spawn(e)}) & = & \mathtt{spawn}(\mathcal{T}(\mathtt{e})) \\ \mathcal{T}(\mathtt{send}(\mathtt{e}_1, \mathtt{e}_2)) & = & \mathtt{send}(\mathcal{T}(\mathtt{e}_1), \mathcal{T}(\mathtt{e}_2)) \\ \mathcal{T}(\mathtt{recv(e)}) & = & \mathtt{recv}(\mathcal{T}(\mathtt{e})) \\ \mathcal{T}(\mathtt{B(v,e)}) & = & \mathcal{T}(\mathtt{e}) \\ \mathcal{T}((\mathtt{U}(\lambda_\delta\, \mathtt{x.e}))\, \mathtt{v}) & = & \mathcal{F}(\mathtt{v}', \overline{\mathtt{C}}) \text{ if } \sigma(\delta, \mathtt{v}) = \overline{C} \\ & & \mathtt{e} \quad \text{otherwise} \end{array}$$

where

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

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

$\mathcal{T}$ 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

$$\langle \mathtt{t}_P, \overline{\theta}, \phi, E[(\lambda_\delta\, \mathtt{x.e})\, \mathtt{v}]\rangle, \sigma \overset{\overline{\alpha}.Mem}{\Longrightarrow} \langle \mathtt{t}_{P'}, \overline{\theta'}, \phi, E[\mathtt{v}']\rangle, \sigma'$$

then

$$\langle \mathtt{t}_{\mathcal{T}(P,\sigma)}, \mathcal{T}(E[(\lambda\, \mathtt{x.e})\, \mathtt{v}])\rangle \longmapsto^{*} \langle \mathtt{t}_{\mathcal{T}(P',\sigma')}, \mathcal{T}(E[\mathtt{v}'])$$

□

Proof. The proof is by induction on the length of $\overline{\alpha}$. Each of the elements comprising $\overline{\alpha}$ correspond to an action necesssary 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).

If $|\overline{\alpha}|$ is one, then α must be MRET, which is the only rule that strips the U tag. The MRET rule simply installs the memoized return value of the function being memoized. The value yielded by MRET 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.

For the inductive step, we examine each memoizable action in turn. 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. Similarly, MCOM also corresponds directly to the COMM rule. From the definition of $\mathcal{T}$, we can split any COMM rule into an MRECV or MSEND by transforming one half of the communicaiton.

The rules for RET and MRET do not correspond to any core evaluation rules. However, when paired with APP and MEMO APP, the pairs correspond to a core evaluation application. Both RET and MRET remove B's and U's respectively inserted by APP and MEMO APP, and thus such pairing is always feasible. By the definition of $\mathcal{T}$ and the induction hypothesis, the value yielded by RET or MRET corresponds to the value yielded by application under core evaluation. □

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 $\rightsquigarrow$ in Fig. 6. 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 $\overline{\alpha}$ be the smallest sequence such that

$$\langle \mathtt{t}_P, \overline{\theta}, \phi, E[(\lambda_\delta\, \mathtt{x.e})\, \mathtt{v}]\rangle, \sigma \overset{\overline{\alpha}.Mem}{\Longrightarrow} \langle \mathtt{t}_{P'}, \overline{\theta'}, \phi, E[\mathtt{v'}]\rangle, \sigma'$$

holds, and let

$$\langle \mathtt{t}_{\mathcal{T}(P,\sigma)}, \mathcal{T}(E[(\lambda\, \mathtt{x.e})\, \mathtt{v}])\rangle \overset{\overline{\beta}}{\longmapsto}{}^{*} \langle \mathtt{t}_{\mathcal{T}(P',\sigma')}, \mathcal{T}(E[\mathtt{v'}])$$

If there are m occurrences of *Ret* tags and n occurrences of *Mem* tags in $\overline{\alpha}$, then $|\overline{\alpha}| \leq |\overline{\beta}| + m + n.\Box$

Proof. As before, the proof follows from the definition of $\mathcal{T}$ and $\mathcal{F}$, and proceeds by induction on the length of $\overline{\alpha}$.

Without loss of generality, let $\overline{\alpha}$ be the *smallest* sequence for which the relation holds. As before, we proceed with the proof by induction on the length of $\overline{\alpha}$.

If $|\overline{\alpha}|$ is one, then α must be MRET, which is the only rule that strips the U tag. Observe that MRET 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).

For the inductive step, we consider each rule under memoized evaluation in turn. 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 MRET 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 MRET remove U's and B's respectively inserted by APP and MEMO APP. Thefore each sequence will contain one additional rule for each APP and MEMO APP step.

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 $\overline{\alpha}$. $\Box$

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

Theorem[Correspondence] If

$$\langle \mathtt{t}_P, \overline{\theta}, \phi, E[(\lambda_\delta\, \mathtt{x.e})\, \mathtt{v}]\rangle, \sigma \overset{\overline{\alpha}.Fail}{\hookrightarrow} \langle \mathtt{t}_{P'}, \overline{\theta}, \phi, E[\mathtt{e'}]\rangle, \sigma$$

and

$$\langle \mathtt{t}_P, \overline{\theta}, \phi, E[(\lambda_\delta\, \mathtt{x.e})\, \mathtt{v}]\rangle, \sigma \overset{\overline{\beta}.Mem}{\rightsquigarrow} \langle \mathtt{t}_{P''}, \overline{\theta}, \phi, E[\mathtt{v'}]\rangle, \sigma$$

then

$$\langle \mathtt{t}_{\mathcal{T}(P',\sigma)}, \mathcal{T}(E[\mathtt{e'}])\rangle \longmapsto^{*} \langle \mathtt{t}_{\mathcal{T}(P'',\sigma)}, \mathcal{T}(E[\mathtt{v'}])\rangle$$

$\Box$

Proof. The proof follows the same structure as the proof of safety, and is shown via induction on the length of $\overline{\beta} - \overline{\alpha}$.

By the definition of Safety and $\mathcal{T}$, all program states created by subsequences of $\overline{\beta}$ can be transformed into equivalent program states yielded by core evaluation. Notice that a FAIL transition can only occur when a memoization candidate has a non-empty set of constraints.

The base case for the induction is when $|\overline{\beta} - \overline{\alpha}|$ is two. The sequence $\overline{\beta}$ must contain an additional constraint (call it β'') as well as an MRET transition for the completion of the application, neither of which are found in $\overline{\alpha}$. Therefore, the following must hold:

$$\langle \mathtt{t}_P, \overline{\theta}, \phi, E[(\lambda_\delta\, \mathtt{x.e})\, \mathtt{v}]\rangle, \sigma \overset{\overline{\beta'}.\overline{\beta''}.Mem}{\rightsquigarrow} \langle \mathtt{t}_{P''}, \overline{\theta}, \phi, E[\mathtt{v'}]\rangle, \sigma$$

where $\overline{\beta''} = \beta'''$.MRET. By the safety theorem, there exists a transition under core evaluation which yields the effect of β''.

For the inductive step, we examine each of the rules under $\hookrightarrow$. Notice PARTIAL MEMO APP corresponds to MEMO APP and only adds a different tag. Therefore, by induction all $\overline{\alpha}$ sequences ending in *PMem* satisfy the theorem. A similar argument holds for APP. Partial memoization behaves identically to memoization in the case when the sequence is not terminated by FAIL.

Thus, consider the FAIL rule. By the structure of the rules a given thread can only take one FAIL transition for any given MEMO APP. Therefore, by our induction hypothesis all sequences prior to a FAIL transition result in a state which corresponds to a core evaluation state. The FAIL rule installs a delimited continuation which can be evaluated under core evaluation since it is a valid term in the core language (the continuation has not yet been memoized). Therefore, the state produced by the FAIL rule must have a valid corresponding core evaluation state.$\Box$

5. Implementation

Our implementation is incorporated within MLton (16), a whole-program optimizing compiler for Standard ML. The main changes to the underlying compiler and library infrastructure are the insertion of barriers to monitor function arguments and return values, hooks to the Concurrent ML (20) library to monitor channel based communication, and changes to the Concurrent ML scheduler to determine memoization feasibility. The entire implementation is roughly 2.5K lines of SML.

5.1 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.

5.2 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. Successful communications occurring within memoized functions were added to the log maintained in the memo table in the form of a constraints, as described previously. Selective communication and complex composed events were also logged upon completion. A complex composed event, on the other hand, simply reduces to a sequence of communications that are logged separately.

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. Benchmarks

6.1 STMBench7

STMBench7 (10) is a comprehensive, tunable multi-threaded benchmark designed to compare different software transactional memory (STM) implementations and designs. Based on the well-known 007 database benchmark (5), 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 are composed of 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 parallel port to Concurrent ML (roughly 1.5K lines of CML). In our implementation, nodes in the input graph are represented as message-passing servers with one receiving channel and output channels to all other adjacent nodes. 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 its adjacent nodes. A transaction can thus be implemented as a series of channel based communications with various server nodes.

6.2 STM Implementation

Our STM implements an eager versioning, lazy conflict detection protocol (4; 21). Shared references in the original Java program are implemented in terms of channel-based communication in the CML port as described above. Since channels are simply heap-allocated data structures, they require no special runtime treatment to guarantee isolation and atomicity. However, all basic synchronous operations in CML involve some side-effect on a channel, through the deposition and removal of values. Since these side-effects would confound the serializability check performed by the STM, we also provide a non-side-effecting version of `recv` that does not remove the contents of the channel on which it is synchronized. Our encoding of shared-memory writes clears the contents of the appropriate channel, and provides a new synchronous value available to subsequent readers. We thereby allow the STM to track accesses to channels in the same way it would track accesses to locations in a shared-memory system.

The STM supports nested, multi-threaded transactions. A multi-threaded transaction is defined as a transaction whose processing is split among a number of threads created by the parent starting the transaction. The threads which comprise a multi-threaded transaction must synchronize at the transaction's commit point.

6.3 Example

Fig. 11 shows a code snippet that is responsible for modifying the height parameters of a building's structural component. A change made by the function `Traversal` affects two components of a design, but the specific changes to each component are disjoint and amenable for concurrent execution. Thus, the modification can easily be expressed as disjoint traversals, expressed by the function `findAtomicPart`. The `sclHgt` function shown in Fig. 11 changes the height parameter of distinct structural parts. Observe that although the height parameter of $\texttt{pid}_2$ depends on the new height of $\texttt{pid}_1$, the traversal to find the part can be executed in parallel. Once $\texttt{pid}_1$ is updated, the traversal for $\texttt{pid}_2$ can complete.

The `atomic` keyword brackets an expression that is to be executed atomically, and also serves to identify memoization candidates. In this example, the transaction created by `Traversal` may fail to commit if parts of the underlying graph referenced by `sclHgt` and `findAtomicPart` changes. Such changes are reflected as modifications to shared channels that hold values of different nodes in the graph.

Observe that much of the computation performed within the transaction is expressed as simple (read-only) graph traversals. Given that most changes are likely to take place on atomic parts and not on higher-level graph components such as complex or base assemblies, the traversal performed by the re-execution is likely to overlap substantially with the original traversal. Of course, when

```
let fun findAtomicPart(object, pid) =
      let val assembly =
               travCAssembly(object, pid)
          val bag = travAssembly(assembly, pid)
          val component = travBag(bag, pid)
          val part = traveComp(component, pid)
      in part
      end
    fun sclHgt(object, pid, c) =
      let val part = findAtomicPart(object, pid)
          val newHeight = height(part)*recv(c)
          val _ = changeHeight(part, newHeight)
       in send(c,newHeight)
       end
    fun Traversal(object, pid1, pid2, height) =
      atomic(fn () =>
               let val c1 = channel()
                   val c2 = channel()
                   val _ = spawn(sclHgt(object,
                                        pid1,
                                        c1))
                   val _ = spawn(sclHgt(object,
                                        pid2,
                                        c2))
               in send(c1, height);
                  send(c2, recv(c1));
                  recv(c2)
               end)
in Traversal()
end
```

Figure 11. Example program illustrating a multi-threaded transaction that traverses a CAD/CAM object.

the transaction executes, it may be that some portion of the graph has changed. Without knowing exactly which part of the graph has been modified by other transactions, the only obvious safe point for re-execution is the beginning of the traversal.

Memoization helps avoid unnecessary re-traversal of the graph when the `Traversal` procedure is re-executed. If (a) the arguments to the first call to `sclHgt` remain the same, (b) the same value is read by the function `height`, (c) there is a value waiting to be received on channel c_1 that is the same as in the original execution, and (d) there is a recipient waiting to consume the value sent along c_1, then the call can be memoized. Notice a receipt of the send of the newly calculated height is guaranteed to complete since the parent thread will always receive on the channel c_1.

The second execution of `sclHgt` is more complex. Although it requires the same constraints as the previous execution, the value it receives on channel c_2 is dependent on the parent's execution of `Traversal` (which sends a value on c_2), which in turn depends on the first execution of `sclHgt`. As we have discussed earlier, memoization decisions that depend on synchronous actions, therefore, cannot necessarily be made at a call site without inspecting the state of other threads. Moreover, these decisions may depend on communication actions other threads may perform in the future; deciding whether the second invocation of `sclHgt` can be memoized depends upon whether the first invocation can be, which in turn depends upon the global state changes that may have occurred between the original (aborted) execution of `Traversal`, and its re-execution.

6.4 Results

To measure the effectiveness of our memoization technique, we executed two configurations of the benchmark, and measured overheads and performance by averaging results over ten executions. The *transactional* configuration uses our STM implementation without any memoization. The *memoized transactional* configuration implements partial memoization of aborted transactions. When a transaction aborts and is re-executed, the applications it originally performed may have initiated new threads of control, and have had these threads communicate with one another using CML primitives. Our memoization techniques can be used to ameliorate the overhead of re-execution.

The benchmark was run on an Intel P4 2.4 GHz machine with one GByte of memory running Gentoo Linux, compiled and executed using MLton release 20051202. Our experiments are not executed on a multiprocessor because the utility of memoization for this benchmark is determined by performance improvement as a function of transaction aborts, and not on raw wallclock speedups.

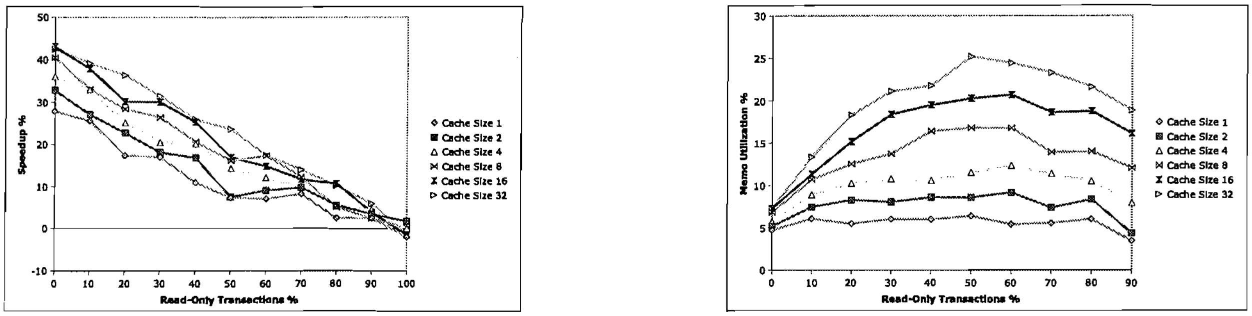
All tests were measured against a graph of about one million nodes. In this graph, there were approximately 280k complex assemblies and 140K assemblies whose bags referenced one of 100 components; by default, each component contained a parts graph of 100 nodes. Each transaction was represented as a separate thread of control. Each node in the graph was represented as a server, constructed from a lightweight CML thread that communicated on two channels. Therefore, our benchmark utilized roughly 500K threads and 1M channels. Transactions, themselves, were composed of at least 7 channel operations to traverse the depth of the tree. On average about 20 nodes of the parts graph were traversed by each transaction.

Our tests varied two independent variables: the read-only/read-write transaction ratio (see Fig. 12) and part graph size (see Fig. 13). The former is significant because only transactions that modify values can cause aborts. Thus, an execution where all transactions are read-only or which never abort cannot be accelerated, but one in which transactions can frequently abort offers potential opportunities for memoization. The latter test is significant because the size of the graph directly correlates to the transaction length. By varying the size of the graph, we alter the number of nodes that each transaction accesses, and thus lengthen or shorten transaction times.

For each test, we also varied the maximum number of memo entries (labeled cache size in the graphs) stored for each function. Tests with a small number experienced less memo utilization than those with a large one. Naturally, the larger the size of the cache used to hold memo information, the greater the overhead. In the case of read-only non-aborting transactions (shown in Fig. 12), performance slowdown is correlated to the maximum memo cache size.

Our experiments consider four different performance facets: (a) runtime improvements for transactions with different read-write ratios across different memo cache sizes (Fig. 12(a)); (b) the amount of memoization exhibited by transactions, again across different memo cache sizes (Fig. 12(b)); (c) runtime improvements as a function of transaction length and memo cache size (Fig. 13(a)); and, (d) the degree of memoization utilization as a function of transaction length and memo cache size (Fig. 13(b). Memory overheads are proportional to cache sizes and averaged roughly 15% for caches with 16 entries. Runs with 32 entry caches had overheads of approximately 18%.

Memoization leads to substantial performance improvements when aborts are likely to be more frequent. For example, even when the percentage of read-only transactions is 60%, we see a 20% improvement in runtime performance compared to a non-memoizing implementation. The percentage of transactions that utilize memo information is related to the size of the memo cache and the likelihood of the transaction aborting; recall only functions within transactions are candidates for memoization. In cases where abort rates are low, for example when there is a sizable fraction of read-only transactions, memo utilization decreases. This is because a func-



(a) (b)

Figure 12. Figure (a) presents normalized runtime speedup with a varying read to write ratio. Figure (b) shows the average percent of transactions which are memoizable as read/write ratios change.

tion can be applied potentially many times, with the majority of applications not leveraging memoization because they were not in aborted transactions. Therefore, memo utilization for these functions will be much lower than a memoized function applied within an aborted transaction.

To measure the impact of transaction size on performance and utilization, we varied the length of the random traversals in the atomic parts graph. As Fig. 13(a) illustrates, smaller transactions offer a smaller chance for memoization (they are more likely to complete), and thus provide less opportunities for performance gains; longer-lived transactions have a greater chance of taking advantage of memo information. This is precisely the motivation for considering memoization in this benchmark. Indeed, we see a roughly 30% performance improvement once the part size contains more than 80 nodes and when the memo cache size is 16 or 32.

7. Related Work and Conclusions

Memoization, or function caching (15; 18; 14), 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 (6). We are unaware of any existing techniques or implementations that apply memoization to the problem of optimizing execution for languages that support first-class channels and dynamic thread creation.

Self adjusting mechanisms (2; 3; 1) leverage memoization along with change propagation to automatically alter a program's execution to a change of inputs given an existing execution run. Selective 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. The combination of these techniques has provided an efficient execution model for programs which are executed a number of times in succession with only small variations in their inputs. However, such techniques require an initial and complete run of the program to gather needed memoization and dependency information before they can adjust to input changes.

New proposals (11) have been presented for self adjusting techniques to be applied in a multi-threaded context. However, these proposals impose significant constraints on the programs considered. References and shared data can only be written to once, forcing self adjusting concurrent programs to be meticulously hand crafted. Additionally such techniques provide no support for synchronization between threads nor do they provide the ability to restore to any control point other than the start of the program.

Reppy and Xiao (19) present a program analysis for CML that analyzes communication patterns to optimize message-passing operations. A type-sensitive interprocedural control-flow analysis is used to specialize communication actions to improve performance. While we also use CML as the underlying subject of interest, our memoization formulation is orthogonal to their techniques.

Our memoization technique shares some similarity with transactional events (7). Transactional events require arbitrary lookahead in evaluation to determine if a complex composed event can commit. We utilize a similar approach to formalize memo evaluation. Unlike transactional events, which are atomic and must either complete entirely or abort, we are not obligated to discover if an application is completely memoizable. If a memoization constraint cannot be discharged, we can continue normal execution of the function body from the failure point.

References

[1] Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An Experimental Analysis of Self-Adjusting Computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 96–107, 2006.

[2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–259, 2002.

[3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective Memoization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–25, 2003.

[4] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37, 2006.

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

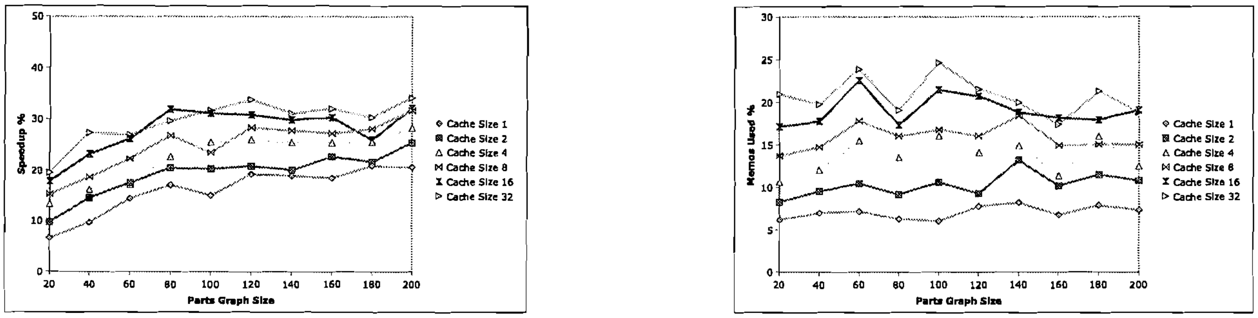
[6] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A Concurrent Logical Framework II: Examples and Applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

[7] Kevin Donnelly and Matthew Fluet. Transactional Events. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 124–135, 2006.

[8] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *ACM Conference on Architectural Support for Programming Languages and Systems*, pages 151–162, 2006.

[9] Jim Gray and Andreas Reuter. *Transaction Processing*. Morgan-Kaufmann, 1993.

[10] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: a Benchmark For Software Transactional Memory. In *Proceedings of*



(a) (b)

Figure 13. Figure (a) shows normalized runtime speedup compared to varying transactional length. Figure (b) shows the percentage of aborted transactions which are memoizable as transaction duration changes.

the European Conference on Operating Systems, 2007.

[11] 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.

[12] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable Memory Transactions. In *ACM Conference on Principles and Practice of Parallel Programming*, pages 48–60, 2005.

[13] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software Transactional Memory for Dynamic-Sized Data Structures. In *ACM Conference on Principles of Distributed Computing*, pages 92–101, 2003.

[14] Allan Heydon, Roy Levin, and Yuan Yu. Caching Function Calls Using Precise Dependencies. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 311–320, 2000.

[15] Yanhong A. Liu and Tim Teitelbaum. Caching Intermediate Results for Program Improvement. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 190–201, 1995.

[16] MLton. http://www.mlton.org.

[17] W. Pugh and T. Teitelbaum. Incremental Computation via Function Caching. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 315–328, 1989.

[18] William Pugh. An Improved Replacement Strategy for Function Caching. In *Proceedings of the ACM conference on LISP and Functional Programming*, pages 269–276, 1988.

[19] John Reppy and Yingqi Xiao. Specialization of CML Message-Passing Primitives. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 315–326, 2007.

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

[21] 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 *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, 2006.