Stabilizers: A Modular Checkpointing Abstraction for Concurrent Functional Programs

Lukasz Ziarek Philip Schatz Suresh Jagannathan

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

Abstract

Transient faults that arise in large-scale software systems can often be repaired by re-executing the code in which they occur. Ascribing a meaningful semantics for safe re-execution in multi-threaded code is not obvious, however. For a thread to correctly re-execute a region of code, it must ensure that all other threads which have witnessed its unwanted effects within that region are also reverted to a meaningful earlier state. If not done properly, data inconsistencies and other undesirable behavior may result. However, automatically determining what constitutes a consistent global checkpoint is not straightforward since thread interactions are a dynamic property of the program.

In this paper, we present a safe and efficient checkpointing mechanism for Concurrent ML (CML) that can be used to recover from transient faults. We introduce a new linguistic abstraction called *stabilizers* that permits the specification of per-thread monitors and the restoration of globally consistent checkpoints. Safe global states are computed through lightweight monitoring of communication events among threads (e.g. message-passing operations or updates to shared variables).

Our experimental results on several realistic, multithreaded, server-style CML applications, including a web server and a windowing toolkit, show that the overheads to use stabilizers are small, and lead us to conclude that they are a viable mechanism for defining safe checkpoints in concurrent functional programs.

Keywords: Concurrent programming, error recovery, checkpointing, transactions, Concurrent ML, exception handling.

1. Introduction

A transient fault is an exceptional condition that can be often remedied through re-execution of the code in which it is raised. Typically, these faults are caused by the temporary unavailability of a resource. For example, a program that attempts to communicate through a network may encounter timeout exceptions because of high network load at the time the request was issued. Transient faults may also arise because a resource is inherently unreliable; consider a network protocol that does not guarantee packet delivery. In large-scale systems comprised of many independently executing components, failure of one component may lead to transient faults in others even after the failure is detected [7]. For example, a client-server application that enters an unrecoverable error state may need to be rebooted; here, the server behaves as a temporarily unavailable resource to its clients who must re-issue requests sent during the period the server was being rebooted. Transient faults may also occur because program invariants are violated. Serializability violations that occur in software transaction systems [17, 19, 32] are typically rectified by aborting the offending transaction and having it re-execute.

A simple solution to transient fault recovery would be to capture the global state of the program before an action executes that could trigger such a fault. If the fault occurs and raises an exception, the handler only needs to restore the previously saved program state. Unfortunately, transient faults often occur in long-running server applications that are inherently multi-threaded but which must nonetheless exhibit good fault tolerance characteristics; capturing global program state is costly in these environments. On the other hand, simply re-executing a computation without taking prior thread interactions into account can result in an inconsistent program state and lead to further errors, such as serializability violations.

Suppose a communication event via message-passing occurs between two threads and the sender subsequently re-executes this code to recover from a transient fault. A spurious unhandled execution of the (re)sent message may result because the receiver would have no knowledge that a re-execution of the sender has occurred. Thus, it has no need to expect re-transmission of a previously executed message. In general, the problem of computing a sensible checkpoint for a transient fault requires calculating the transitive closure of dependencies manifest among threads and the section of code which must be re-executed.

To alleviate the burden of defining and restoring safe and efficient checkpoints in concurrent functional programs, we propose a new language abstraction called *stabilizers*. Stabilizers encapsulate three operations. The first initiates monitoring of code for communication and thread creation events, and establishes thread-local checkpoints when monitored code is evaluated. This thread-local checkpoint can be viewed as a restoration point for any transient fault encountered during the execution of the monitored region. The second operation reverts control and state to a safe global checkpoint when a transient fault is detected. The third operation allows previously established checkpoints to be reclaimed.

The checkpoints defined by stabilizers are first-class and composable: a monitored procedure can freely create and return other monitored procedures. Stabilizers can be arbitrarily nested, and

2006 ACM International Conference on Functional Programming

work in the presence of a dynamically-varying number of threads and non-deterministic selective communication. We demonstrate the use of stabilizers for several large server applications written in Concurrent ML.

Stabilizers provide a middle ground between the transparency afforded by operating systems or compiler-injected checkpoints, and the precision afforded by user-injected checkpoints. In our approach, thread-local state immediately preceding a non-local action (e.g., thread communication, thread creation, etc.) is regarded as a possible checkpoint for that thread. In addition, applications may explicitly identify program points where local checkpoints should be taken, and can associate program regions with these specified points. When a rollback operation occurs, control reverts to one of these saved checkpoints for each thread. Rollbacks are initiated to recover from transient faults. The exact set of checkpoints chosen is determined by safety conditions that ensure that a globally consistent state is preserved. Our approach guarantees that when a thread is rolled-back to a thread-local checkpoint state C, other threads with which it has communicated will be placed in states consistent with C.

This paper makes three contributions:

1. The design and semantics of *stabilizers*, a new modular language abstraction for transient fault recovery in concurrent functional programs. To the best of our knowledge, stabilizers are the first *language-centric* design of a checkpointing facility that provides global consistency and safety guarantees for transient fault recovery in programs with dynamic thread creation, and selective communication [25].
2. A lightweight dynamic monitoring algorithm faithful to the semantics that constructs efficient global checkpoints based on the context in which a restoration action is performed. Efficiency is defined with respect to the amount of rollback required to ensure that all threads resume execution after a checkpoint is restored to a consistent global state.
3. A detailed evaluation study for Concurrent ML that quantifies the cost of using stabilizers on various open-source server-style applications. Our results reveal that the cost of defining and monitoring thread state is small, typically adding roughly no more than four to six percent overhead to overall execution time. Memory overheads are equally modest.

The remainder of the paper is structured as follows. Section 2 describes the stabilizer abstraction. Section 3 provides a motivating example that highlights the issues associated with transient fault recovery in a highly multi-threaded webserver, and how stabilizers can be used to alleviate complexity and improve robustness. An operational semantics is given in Section 4. A strategy for incremental construction of checkpoint information is given in Section 5. Implementation details are provided in Section 6. A detailed evaluation on the costs and overheads of using stabilizers for transient fault recovery is given in Section 7, related work is presented in Section 8, and conclusions are given in Section 9.

2. Programming Model

Stabilizers are expressed using three primitives with the following signatures:

```
stable     : ('a -> 'b)  -> 'a -> 'b
stabilize  : unit -> 'a
cut        : unit -> unit
```

A *stable section* is a monitored section of code whose effects are guaranteed to be reverted as a single unit. The primitive `stable` is used to define stable sections. Given function *f* the evaluation of `stable` *f* yields a new function *f'* identical to *f* except that interesting communication, shared memory access, locks, and spawn events are monitored and grouped. Thus, all actions within a stable section are associated with the same checkpoint. This semantics is in contrast to classical checkpointing schemes where there is no manifest grouping between a checkpoint and a collection of actions.

The second primitive, `stabilize` reverts execution to a dynamically calculated global state; this state will always correspond to a program state that existed immediately prior to execution of a stable section, communication event, or thread spawn point for each thread. We qualify this claim by observing that external non-revocable operations that occur within a stable section that needs to be reverted (e.g., I/O, foreign function calls, etc.) must be handled explicitly by the application prior to an invocation of a `stabilize` action. Note that similar to operations like `raise` or `exit` that also do not return, the result type of `stabilize` is synthesized from the context in which it occurs.

Informally, a stabilize action reverts all effects performed within a stable section much like an abort action reverts all effects within a transaction. However, whereas a transaction enforces atomicity and isolation until a commit occurs, stabilizers enforce these properties only when a stabilize action occurs. Thus, the actions performed within a stable section are immediately visible to the outside; when a stabilize action occurs these effects along with their witnesses are reverted.

The third primitive, `cut`, establishes a point beyond which stabilization cannot occur. An error is raised if a `stabilize` action attempts to revert state beyond the point at which a `cut` action occurs. Cut points can be used to prevent the unrolling of irrevocable actions within a program (e.g., I/O) or to bound the amount of rollback that a stabilization action may trigger. A cut is a per-thread delimiting action; a cut established in a particular thread has no effect on other threads unless they perform stabilization actions that require restoring this thread past the cut point because of previously established dependencies.

Unlike classical checkpointing schemes [28] or exception handling mechanisms, the result of invoking `stabilize` does not guarantee that control reverts to the state corresponding to the dynamically-closest stable section. The choice of where control reverts depends upon the actions undertaken by the thread within the stable section in which the `stabilize` call was triggered.

Composability is an important design feature of stabilizers: there is no *a priori* classification of the procedures that need to be monitored, nor is there any restriction against nesting stable sections. Stabilizers separate the construction of monitored code regions from the capture of state. It is only when a monitored procedure is applied that a potential thread-local restoration point is established. The application of such a procedure may in turn result in the establishment of other independently constructed monitored procedures. In addition, these procedures may themselves be applied and have program state saved appropriately; thus, state saving and restoration decisions are determined without prejudice to the behavior of other monitored procedures.

2.1 Interaction of Stable Sections

When a stabilize action occurs, matching inter-thread events are unrolled as pairs. If a send is unrolled, the matching receive must also be reverted. If a thread spawns another thread within a stable section that is being unrolled, this new thread (and all its actions) must also be discarded. All threads which read from a shared variable must be reverted if the thread that wrote the value is unrolled to a state prior to the write. A program state is *stable* with respect to a statement if there is no thread executing in this state affected by the statement (i.e., all threads are in a point within their execution prior to the execution of the statement and its transitive effects).

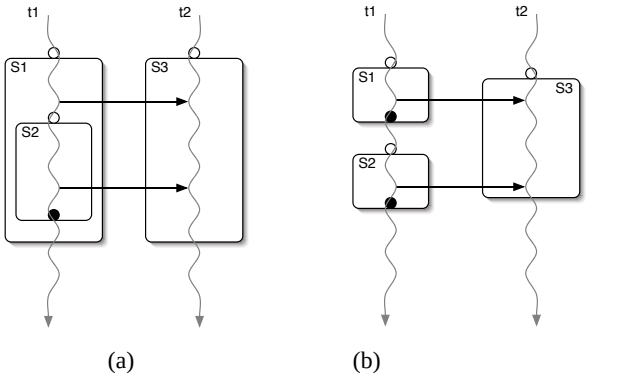


Figure 1. Interaction between stable sections.

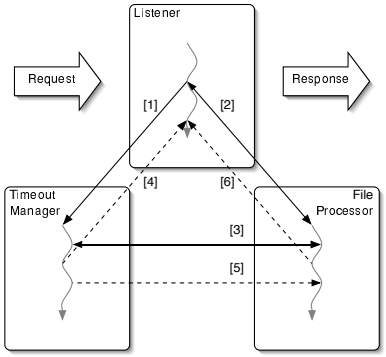


Figure 2. `Swerve` module interactions for processing a request (solid lines) and error handling control and data flow (dashed lines) for timeouts. The number above the lines indicates the order in which communication actions occur.

For example, consider thread t_1 that enters a stable section S_1 and initiates a communication event with thread t_2 (see Fig. 1(a)). Suppose t_1 subsequently enters another stable section S_2, and again establishes a communication with thread t_2. Suppose further that t_2 receives these events within its own stable section S_3. The program states immediately prior to S_1 and S_2 represent feasible checkpoints as determined by the programmer, depicted as white circles in the example. If a rollback is initiated within S_2, then a consistent global state would require that t_2 revert back to the state associated with the start of S_3 since it has received a communication from t_1 initiated within S_2. However, discarding the actions within S_3 now obligates t_1 to resume execution at the start of S_1 since it initiated a communication event within S_1 to t_2 (executing within S_3). Such situations can also arise without the presence of nested stable sections. Consider the example in Fig. 1(b). Once again, the program is obligated to revert t_1, since the stable section S_3 spans communication events from both S_1 and S_2.

3. Motivating Example

`Swerve` [20] (see Fig 2) is an open-source third-party Web server wholly written in Concurrent ML. The server is composed of five separate interacting modules. Communication between modules and threads makes extensive use of CML message passing semantics. Threads communicate over explicitly defined channels on which they can either send or receive values. To motivate the use of stabilizers, we consider the interactions of three of `Swerve`'s modules: the `Listener`, the `File Processor`, and the `Timeout Manager`. The `Listener` module receives incoming `HTTP` requests and delegates file serving requirements to concurrently executing processing threads. For each new connection, a new listener is spawned; thus, each connection has one main governing entity. The `File Processor` module handles access to the underlying file system. Each file that will be hosted is read by a file processor thread that chunks the file and sends it via message-passing to the thread delegated by the listener to host the file. Timeouts are processed by the `Timeout Manager` through the use of timed events on channels. Threads can poll these channels to check if there has been a timeout. In the case of a timeout, the channel will hold a flag signaling time has expired, and is empty otherwise.

Timeouts are the most frequent transient fault present in the server, and difficult to deal with naively. Indeed, the system's author notes that handling timeouts in a modular way is "tricky" and devotes an entire section of the user manual explaining the pervasive cross-module error handling in the implementation. Consider the typical execution flow given in Fig 2. When a new request is received, the listener spawns a new thread for this connection that is responsible for hosting the requested page. This hosting thread first establishes a timeout quantum with the timeout manager (1) and then notifies the file processor (2). If a file processing thread is available to process the request, the hosting thread is notified that the file can be chunked (2). The hosting thread passes to the file processing thread the channel on which it will receive its timeout notification (2). The file processing thread is now responsible to check for explicit timeout notification (3).

Since a timeout can occur before a particular request starts processing a file (4) (i.e. within the hosting thread defined by the `Listener` module) or during the processing of a file (5) (i.e. within the `File Processor`), the resulting error handling code is cumbersome. Moreover, the detection of the timeout itself is handled by a third module, the `Timeout Manager`. The result is a complicated message passing procedure that spans multiple modules, each of which must figure out how to deal with timeouts appropriately. The unfortunate side effect of such code organization is that modularity is compromised. The code now contains implicit interactions that cannot be abstracted (6) (i.e. the `File Processor` must explicitly notify the `Listner` of the timeout). The `Swerve` design illustrates the general problem of dealing with transient faults in a complex concurrent system: how can we correctly handle faults that span multiple modules without introducing explicit cross-module dependencies to handle each such fault?

Fig. 3 shows the definition of `fileReader`, a `Swerve` function in the file processing module that sends a requested file to the hosting thread by chunking the file contents into a series of smaller packets. The file is opened by `BinIOReader`, a utility function in the `File Processing` module. The `fileReader` function must check in every iteration of the file processing loop whether a timeout has occurred by calling the `Timeout.expired` function due to the restriction that CML threads cannot be explicitly interrupted. If a timeout has occurred, the procedure is obligated to notify the receiver (the hosting thread) through an explicit send on channel `consumer`.

Stabilizers allow us to abstract this explicit notification process by wrapping the file processing logic of `sendFile` in a stable section. Suppose a call to `stabilize` replaced the call to `CML.send(consumer, Timeout)`. This action would result in unrolling both the actions of `sendFile` as well as the receiver, since the receiver is in the midst of receiving file chunks.

However, a cleaner solution presents itself. Suppose that we modify the definition of the `Timeout` module to invoke `stabilize`, and wrap its operations within a stable section as shown in Fig. 4. Now, there is no need for any thread to poll for the timeout event. Since the hosting thread establishes a timeout quantum by commu-

```
fun fileReader name abort consumer =
let fun sendFile() =
  let fun loop strm =
    if Timeout.expired abort
    then CML.send(consumer, Timeout)
    else let val chunk =
               BinIO.inputN(strm, fileChunk)
         in ... read a chunk of the file
            ... and send to receiver
            loop strm)
         end
  in (case BinIOReader.openIt abort name
        of NONE => ()
         | SOME h => (loop (BinIOReader.get h);
                       BinIOReader.closeIt h)
  end
```

```
fun fileReader name abort consumer =
let fun sendFile() =
  let fun loop strm =
    let val chunk =
          BinIO.inputN(strm, fileChunk)
    in ... read a chunk of the file
       ... and send to receiver
       loop strm)
    end
  in stable fn() =>
       (case BinIOReader.openIt abort name
         of NONE   =>()
         | SOME h =>(loop (BinIOReader.get h);
                     BinIOReader.closeIt h)) ()
  end
```

Figure 3. An excerpt of the The `File Processing` module in `Swerve`. The right-hand side shows the code modified to use stabilizers. Italics mark areas in the original where the code is changed.

```
let fun expired (chan) = isSome (CML.poll chan)
    fun trigger (chan) = CML.send(chan, timeout)
    ...
in ...; trigger(chan)
end
```

```
let fun trigger (chan) = stabilize()
    ..
in stable (fn() => ... ; trigger(chan)) ()
end
```

Figure 4. An excerpt of the `Timeout Manager` module in `Swerve`. The right-hand side shows the code modified to use stabilizers. The expired function can be removed and a trigger now calls `stabilize`. Italics mark areas in the original where the code is changed.

```
fn () =>
  let fun receiver() =
    case CML.recv consumer
      of info     => (sendInfo info; ...)
       | chunk    => (sendBytes bytes; ...)
       | timeout  =>  error handling code
       | done     => ...
    ...
    in ... ; loop receiver
    end
```

```
stable fn () =>
  let fun receiver() =
    case CML.recv consumer
      of info     => (sendInfo info; ...)
       | chunk    => (sendBytes bytes; ...)
       | done     => ...
  ...
  in ... ; loop receiver
  end
```

Figure 5. An excerpt of the `Listener` module in `Swerve`. The main processing of the hosting thread is wrapped in a stable section and the timeout handling code can be removed. The right-hand side shows the code modified to use stabilizers. Italics mark areas in the original where the code is changed.

nicating with `Timeout` and passes this information to the file processor thread, any `stabilize` action performed by the `Timeout` manager will unroll all actions related to processing this file. This transformation therefore allows us to specify a timeout mechanism without having to embed non-local timeout handling logic within each thread that potentially could be affected. The hosting thread itself is also simplified (as seen in Fig. 5); by wrapping its logic within a stable section, we can remove all of its timeout error handling code as well. A timeout is now handled completely through the use of stabilizers localized within the `Timeout` module. This improved modularization of concerns does not lead to reduced functionality or robustness. Indeed, a stabilize action causes the timed-out request to be transparently re-processed, or allows the webserver to process a new request, depending on the desired behavoir. Thus, each module only has to manage its own components and does not have to explicitly communicate with other modules in the case of a timeout error.

4. Semantics

Our semantics is defined in terms of a core call-by-value functional language with threading primitives (see Fig. 6). For perspicuity, we first present an interpretation of stabilizers in which evaluation of stable sections immediately results in the capture of a consistent global checkpoint. Furthermore, we restrict the language to capture checkpoints only upon entry to stable sections, rather than at any communication or thread creation action. This semantics reflects a simpler characterization of checkpointing than the informal description presented in Section 2. In Section 5, we refine this approach to construct checkpoints incrementally.

In the following, we use metavariables v to range over values, and δ to range over stable section or checkpoint identifiers. We also use P for thread terms, and `e` for expressions. We use over-bar to represent a finite ordered sequence, for instance, $\overline{f}$ represents $f_1\, f_2\, \ldots\, f_n$. The term $\alpha.\overline{\alpha}$ denotes the prefix extension of the sequence $\overline{\alpha}$ with a single element α, $\overline{\alpha}.\alpha$ the suffix extension, $\overline{\alpha\alpha'}$ denotes sequence concatenation, ϕ denotes empty sequences and

SYNTAX:

$$
\begin{array}{rcl}
P & ::= & P \| P \mid \mathtt{t}[\mathtt{e}]_{\overline{\delta}} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{l} \mid \lambda\,\mathtt{x}.\mathtt{e} \\
& \mid & \mathtt{mkCh}() \mid \mathtt{send}(\mathtt{e},\mathtt{e}) \mid \mathtt{recv}(\mathtt{e}) \mid \mathtt{spawn}(\mathtt{e}) \\
& \mid & \mathtt{stable}(\mathtt{e}) \mid \overline{\mathtt{stable}}(\mathtt{e}) \mid \mathtt{stabilize}() \mid \mathtt{cut}()
\end{array}
$$

EVALUATION CONTEXTS:

$$
\begin{array}{rcl}
E & ::= & \bullet \mid E(\mathtt{e}) \mid \mathtt{v}(E) \mid \\
& & \mathtt{send}(E,\mathtt{e}) \mid \mathtt{send}(\mathtt{l},E) \mid \\
& & \mathtt{recv}(E) \mid \mathtt{stable}(E) \mid \overline{\mathtt{stable}}(E) \\
E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{e}] & ::= & P \| \mathtt{t}[E[\mathtt{e}]]_{\overline{\delta}}
\end{array}
$$

PROGRAM STATES:

$$
\begin{array}{rcll}
P & \in & \mathit{Process} & \\
\mathtt{t} & \in & \mathit{Tid} & \\
\mathtt{x} & \in & \mathit{Var} & \\
\mathtt{l} & \in & \mathit{Channel} & \\
\delta & \in & \mathit{StableId} & \\
\mathtt{v} & \in & \mathit{Val} & = \ \mathtt{unit} \mid \lambda\,\mathtt{x}.\mathtt{e} \mid \mathtt{stable}(\lambda\,\mathtt{x}.\mathtt{e}) \mid \mathtt{l} \\
\alpha,\beta & \in & \mathit{Op} & = \ \{\text{LR,SP,COMM,SS,ST,ES,CUT}\} \\
\Lambda & \in & \mathit{StableState} & = \ \mathit{Process} \times \mathit{StableMap} \\
\Delta & \in & \mathit{StableMap} & = \ \mathit{StableId} \xrightarrow{\text{fin}} \mathit{StableState}
\end{array}
$$

LOCAL EVALUATION RULES:

$$
\begin{array}{rcl}
\lambda\,\mathtt{x}.\mathtt{e}(\mathtt{v}) & \rightarrow & \mathtt{e}[\mathtt{v}/\mathtt{x}] \\
\mathtt{mkCh}() & \rightarrow & \mathtt{l}, \quad \mathtt{l} \ \text{fresh}
\end{array}
$$

$$
\frac{\mathtt{e} \rightarrow \mathtt{e}'}{E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{e}],\Delta \overset{\text{LR}}{\Longrightarrow} E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{e}'],\Delta}
$$

GLOBAL EVALUATION RULES:

$$
\frac{\mathtt{t}' \text{fresh}}{E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{spawn}(\mathtt{e})],\Delta \overset{\text{SP}}{\Longrightarrow} P \| \mathtt{t}[E[\mathtt{unit}]]_{\overline{\delta}} \| \mathtt{t}'[\mathtt{e}]_{\phi},\Delta}
$$

$$
\frac{\mathtt{t}' \text{fresh} \qquad P = P' \| \mathtt{t}[E[\mathtt{send}(\mathtt{l},\mathtt{v})]]_{\overline{\delta}} \| \mathtt{t}'[E'[\mathtt{recv}(\mathtt{l})]]_{\overline{\delta'}}}{P,\Delta \overset{\text{COMM}}{\Longrightarrow} P' \| \mathtt{t}[E[\mathtt{unit}]]_{\overline{\delta}} \| \mathtt{t}'[E'[\mathtt{v}]]_{\overline{\delta'}},\Delta}
$$

$$
\frac{}{E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{cut}()],\Delta \overset{\text{CUT}}{\Longrightarrow} E^{\mathtt{t},P}_{\phi}[\mathtt{unit}],\phi}
$$

$$
\frac{\begin{array}{c}\delta' \text{ fresh} \qquad \forall \delta \in \mathit{Dom}(\Delta), \quad \delta' \geq \delta \\ \Delta' = \Delta[\delta' \mapsto (E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{stable}(\lambda\,\mathtt{x}.\mathtt{e})(\mathtt{v})],\Delta)] \\ \Lambda = \Delta'(\delta_{min}), \quad \delta_{min} \leq \delta \quad \forall \delta \in \mathit{Dom}(\Delta')\end{array}}{E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{stable}(\lambda\,\mathtt{x}.\mathtt{e})(\mathtt{v})],\Delta \overset{\text{SS}}{\Longrightarrow} E^{\mathtt{t},P}_{\delta'.\overline{\delta}}[\overline{\mathtt{stable}}(\mathtt{e}[\mathtt{v}/\mathtt{x}])],\Delta[\delta' \mapsto \Lambda]}
$$

$$
E^{\mathtt{t},P}_{\delta.\overline{\delta}}[\overline{\mathtt{stable}}(\mathtt{v})],\Delta \overset{\text{ES}}{\Longrightarrow} E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{v}],\Delta - \{\delta\}
$$

$$
\frac{\Delta(\delta) = (P',\Delta')}{E^{\mathtt{t},P}_{\delta.\overline{\delta}}[\mathtt{stabilize}()],\Delta \overset{\text{ST}}{\Longrightarrow} P',\Delta'}
$$

Figure 6. A core call-by-value language for stabilizers.

sets, and $\overline{\alpha} \leq \overline{\alpha'}$ holds if $\overline{\alpha}$ is a prefix of $\overline{\alpha'}$. We write $\mid \overline{\alpha} \mid$ to denote the length of sequence $\overline{\alpha}$.

Our communication model is a message-passing system with synchronous send and receive operations. We do not impose a strict ordering of communication actions on channels; communication actions on the same channel are paired non-deterministically. To model asynchronous sends, we simply spawn a thread to perform the send[1]. To this core language we add three new primitives: `stable` and `stabilize`. When a stable function is applied, a global checkpoint is established, and its body, denoted as $\overline{\mathtt{stable}}(e)$, is evaluated in the context of this checkpoint. The second primitive, `stabilize`, is used to initiate a rollback and the third, `cut`, clears all current checkpoints. An attempt to restore to a checkpoint which is no longer present results in a stuck state that is tantamount to an error.

The syntax and semantics of the language are given in Fig. 6. Expressions are variables, locations to represent channels, λ-abstractions, function applications, thread creations, communication actions to send and receive messages on channels, or operations to define stable sections, and to stabilize global state to a consistent checkpoint. We do not consider references in this core language as they can be modeled in terms of operations on channels. We describe how to handle references efficiently in an implementation in Section 6.2.

A program is defined as a collection of threads. Each thread is uniquely identified, and is also associated with a *stable section identifier* (denoted by δ) that indicates the stable section the thread is currently executing within. Stable section identifiers are ordered under a relation that allows us to compare them (e.g., they could be thought of as integers incremented by a global counter). Thus, we write $\mathtt{t}[\mathtt{e}]_{\delta}$ if a thread with identifier `t` is executing expression `e` in the context of stable section δ; since stable sections can be nested, the notation generalizes to sequences of stable section identifiers with sequence order reflecting nesting relationships. We omit decorating a term with stable section identifiers when appropriate. Our semantics is defined up to congruence of threads ($\mathtt{P} \| \mathtt{P}' \equiv \mathtt{P}' \| \mathtt{P}$). We write $\mathtt{P} \ominus \{\mathtt{t}[\mathtt{e}]\}$ to denote the set of threads that do not include a thread with identifier `t`, and $\mathtt{P} \oplus \{\mathtt{t}[\mathtt{e}]\}$ to denote the set of threads that contain a thread executing expression `e` with identifier `t`.

We use evaluation contexts to specify order of evaluation within a thread, and to prevent premature evaluation of the expression encapsulated within a `spawn` expression. We define a thread context $E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{e}]$ to denote an expression `e` available for execution by thread $\mathtt{t} \in P$ within context E; the sequence $\overline{\delta}$ indicates the ordered sequence of nested stable sections within which the expression evaluates.

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: function application substitutes the value of the actual parameter for the formal in the function body, and channel creation results in the creation of a new location that acts as a container for message transmission and receipt.

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

Program evaluation is specified by a global reduction relation, $P, \Delta, \stackrel{\alpha}{\Longrightarrow} P', \Delta'$, that maps a program state to a new program state. We tag each evaluation step with an action, α, that defines the effects induced by evaluating the expression. We write $\stackrel{\overline{\alpha}}{\Longrightarrow}{}^*$ to denote the reflexive, transitive closure of this relation. The actions of interest are those that involve communication events, or manipulate stable sections. We use labels LR to denote local reduction actions, SP to denote thread creation, COMM to denote thread communication, SS to indicate the start of a stable section, ST to indicate a stabilize operation, ES to denote the exit from a stable section, and CUT to indicate a cut action. A program state consists of a collection of evaluating threads (P) and a stable map (Δ) that defines a finite function associating stable section identifiers to states. A program begins evaluation with an empty stable map.

There are six global evaluation rules. The first describes changes to the global state when a thread to evaluate expression `e` is created; the new thread evaluates `e` in a context without any stable identifier. The second describes how a communication event 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. Evaluating a `cut` simply discards all previous stored checkpoints.

The remaining three, and most interesting, global evaluation rules are ones involving stable sections. When a stable section is newly entered, a new stable section identifier is generated; these identifiers are related under a total order that allows the semantics to express properties about lifetimes and scopes of such sections. The newly created identifier is mapped to the current global state and this mapping is recorded in the stable map. This state represents a possible checkpoint. The actual checkpoint for this identifier is computed as the state in the stable map that is mapped by the least stable identifier. This identifier represents the *oldest* active checkpointed state. This state is either the state just checkpointed, in the case when the stable map is empty, or represents some earlier checkpoint state known to not have any dependencies with actions in other stable sections. In other words, if we consider stable sections as forming a tree with branching occurring at thread creation points, the checkpoint associated with any stable section represents the root of the tree at the point where control enters that section.

When a stable section exits, the thread context is appropriately updated to reflect that the state captured when this section was entered no longer represents an interesting checkpoint; the stable section identifier is removed from the resulting stable map. A stabilize action simply reverts to the state captured by the outermost stable section of this thread. While easily defined, the semantics is highly conservative because there may be checkpoints that involve less unrolling that the semantics does not identify. Consider the example given in Fig. 7 where two threads execute calls to monitored functions `f, h, g` in that order. Because `f` is monitored, a global checkpoint is taken prior to its call. Now, suppose that the call to `h` by *Thread 2* occurs before the call to `f` completes. Observe that `h` communicates with function `g` via a synchronous communication action on channel `c`. Assuming no other threads in the program, `h` cannot complete until `g` accepts the communication. Thus, when `g` is invoked, the earliest global checkpoint calculated by the stable section associated with the call is the checkpoint established by the `stable` section associated with `f`, which happens to be the checkpoint referenced by the stable section that monitors `h`. In other words, stabilize actions performed within either `h` or `g` would result in the global state reverting back to the start of `f`'s execution, even though `f` completed successfully. This strategy, while correct, is unnecessarily conservative as we describe in the next section.

Thread 1

```
let fun f() = ...
    fun g() = ...
              recv(c)
              ...
in stable g
     (stable f ())
end
```

Thread 2

```
let fun h() =
          ...
          send(c, v)
          ...
in stable h ()
end
```

Figure 7. The interaction of thread communication and stable sections.

The soundness of the semantics is defined by an *erasure* property on stabilize actions. Consider the sequence of actions $\overline{\alpha}$ that comprise a possible execution of a program. Suppose that there is a `stabilize` operation that occurs in $\overline{\alpha}$. The effect of this operation is to revert the current global program state to an earlier checkpoint. However, given that program execution successfully continued after the `stabilize` call, it follows that there exists a sequence of actions from the checkpoint state that yields the same state as the original, but which does *not* involve execution of the `stabilize` operation. In other words, `stabilize` actions can never manufacture new states, and thus have no effect on the final state of program evaluation. We formalize this property in the following safety theorem.

Theorem[Safety.] Let

$$E_\phi^{\mathtt{t},P}[\mathtt{e}], \Delta \stackrel{\overline{\alpha}}{\Longrightarrow}{}^* P', \Delta' \stackrel{\text{ST}.\overline{\beta}}{\Longrightarrow}{}^* P''\|\mathtt{t}[\mathtt{v}], \Delta_f$$

If $\overline{\alpha}$ is non-empty, there exists an equivalent evaluation

$$E_\phi^{\mathtt{t},P}[\mathtt{e}], \Delta \stackrel{\overline{\alpha'}.\overline{\beta}}{\Longrightarrow}{}^* P''\|\mathtt{t}[\mathtt{v}], \Delta_f$$

such that $\overline{\alpha'} \leq \overline{\alpha}$.

5. Incremental Construction

Although correct, our semantics is overly conservative because a global checkpoint state is computed upon entry to every stable section. Furthermore, communication events that establish inter-thread dependencies are not considered in the checkpoint calculation. Thus, all threads, even those unaffected by effects that occur in the interval between when the checkpoint is established and when it is restored, are unrolled. A better alternative would restore thread state based on the actions witnessed by threads within checkpoint intervals. If a thread T observes action α performed by thread T' and T is restored to a state that precedes the execution of α, T' can be restored to its *latest* local checkpoint state that precedes its observance of α. If T witnesses no actions of other threads, it is unaffected by any `stabilize` calls those threads might make. This strategy leads to an improved checkpoint algorithm by reducing the severity of restoring a checkpoint, limiting the impact to only those threads that witness global effects, and establishing their rollback point to be as temporally close as possible to their current state.

Fig. 9 presents a refinement to the semantics that incrementally constructs a dependency graph as part of program execution. This new definition does not require stable section identifiers or stable maps to define checkpoints. Instead, it captures the communication actions performed by threads within a data structure. This structure consists of a set of nodes representing interesting program points, and edges that connect nodes that have shared dependencies. Nodes are indexed by ordered node identifiers, and hold thread state. We also define maps to associate threads with nodes, and their set of active stable sections.

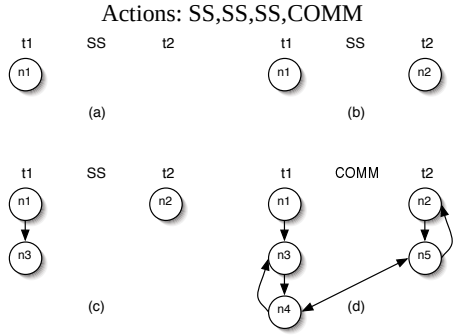


Figure 8. An example of incremental checkpoint construction.

Informally, the actions of each thread in the graph are represented by a chain of nodes that define temporal ordering on thread-local actions. Backedges are established to nodes representing stable sections; these nodes define possible *per-thread* checkpoints. Sources of backedges are communication actions that occur within a stable section, or the exit of a nested stable section. Edges also connect nodes belonging to different threads to capture inter-thread communication events.

Graph reachability is used to ascertain a global checkpoint when a `stabilize` action is performed: when thread T performs a `stabilize` call, all nodes reachable from T's current node in the graph are examined, and the context associated with the *least* such reachable node for each thread is used as the thread-local checkpoint for that thread. If a thread is not affected (transitively) by the actions of the thread performing the rollback, it is not reverted to any earlier state. The collective set of such checkpoints constitutes a global state.

The evaluation relation $P, G \overset{\alpha}{\rightsquigarrow} P', G'$ evaluates a process P executing action α with respect to a communication graph G to yield a new process P' and new graph G'. As usual, $\overset{\overline{\alpha}}{\rightsquigarrow}{}^*$ denotes the reflexive, transitive closure of this relation. Programs initially begin evaluation with respect to an empty graph. The auxiliary relation $\mathtt{t}[\mathtt{e}], G \Downarrow G'$ models intra-thread actions within the graph. It creates a new node to capture thread-local state, and sets the current node marker for the thread to this node. In addition, if the action occurs within a stable section, a back-edge is established from that node to this section. This backedge is used to identify a potential rollback point. If a node has a backedge the restoration point will be determined by traversing these backedge, thus it is safe to not store thread contexts with such nodes ($\perp$ is stored in the node in that case). New nodes added to the graph are created with a node identifier guaranteed to be greater than any existing node.

When a stabilization action occurs, the set of nodes reachable from the node representing the enclosing stable section is calculated. Significantly, this set should not include a node corresponding to a `cut` operation. The presence of such a node in the reach set indicates an attempt to stabilize a computation beyond a cut point and is erroneous. The new graph reflects the restoration; G/N is the graph G with the subgraph rooted at nodes $n \in N$ removed.

We define the following theorem that formalizes the intuition that incremental checkpoint construction results in less rollback than a global point-in-time checkpoint strategy:

Theorem[Efficiency] If

$$E_\phi^{\mathtt{t},P}[\mathtt{e}], \Delta_0 \overset{\overline{\alpha}.\mathrm{ST}}{\Longrightarrow}{}^* P', \Delta'$$

and

$$E_\phi^{\mathtt{t},P}[\mathtt{e}], G_0 \overset{\overline{\alpha}.\mathrm{ST}}{\rightsquigarrow}{}^* P'', G'$$

then $P', \Delta' \overset{\overline{\beta}}{\Longrightarrow}{}^* P'', \Delta''$.

5.1 Example

To illustrate the semantics, consider the sequence of actions shown in Fig. 8 that is based on the example given in Fig. 7. The node n_1 represents the start of the stable section monitoring function `f` (a). Next, a monitored instantiation of `h` is created, and a new node associated with this context is allocated in the graph (b). Monitoring of function `g` results in a new node to the first thread with an edge from the previous node joining the two (see c). Lastly, consider the exchange of a value on channel `c` by the two threads. Nodes corresponding to the communication are created, along with backedges to their respective stable section (d).

Recall the global checkpointing scheme would restore to a global checkpoint created at the point the monitored version of `f` was produced, regardless of where a stabilization action took place. In contrast, a stabilize call occurring within the execution of either `g` or `h` using this incremental scheme would restore the first thread to the continuation stored in node n_3 (corresponding to the context immediately preceding the call to `g`), and would restore the second thread to the continuation stored in node n_2 (corresponding to the context immediately preceding the call to `h`).

6. Implementation

Our implementation is incorporated within MLton [20], a whole-program optimizing compiler for Standard ML. The main changes to the underlying infrastructure were the insertion of read and write barriers to track shared memory updates, and hooks to the Concurrent ML [25] library to update the communication graph. State restoration is thus a combination of restoring continuations as well as reverting references. The implementation is roughly 3K lines of code to support our data structures, checkpointing, and restoration code, as well as roughly 200 lines of changes to CML.

6.1 Supporting First-Class Events

Because our implementation is an extension of the core CML library, it supports first-class events [25] as well as channel-based communication. The handling of events is no different than our treatment of messages. If a thread is blocked on an event with an associated channel, we insert an edge from that thread's current node to the channel. We support CML's selective communication with no change to the basic algorithm. Since CML imposes a strict ordering of communication events, each channel must be purged of spurious or dead data after a stabilize action. CML utilizes transaction identifiers for each communication action, or in the case of selective communication, a series of communication actions. CML already implements clearing channels of spurious data when a `sync` operation occurs on a selective communication. This is done lazily by tagging the transaction identifier as *consumed*. Communication actions check and remove any data so tagged. We utilize this same process for clearing channels during a stabilize action.

6.2 Handling References

We have thus far elided details on how to track shared memory access to properly support state restoration actions in the presence of references. Naively tracking each read and write separately would be inefficient. There are two problems that must be addressed: (1) unnecessary writes should not be logged; and (2) spurious dependencies induced by reads should be avoided.

Notice that for a give stable section, it is enough to monitor the first write to a given memory location since each stable section is unrolled as a single unit. For every write to location l, we need to only monitor the first read performend by another thread to l; if this

SYNTAX AND EVALUATION CONTEXTS

$$\begin{array}{rcl} P & ::= & P \| P \mid \mathtt{t}[\mathtt{e}]_{\overline{\delta}} \\ E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{e}] & ::= & P \| \mathtt{t}[E[\mathtt{e}]]_{\overline{\delta}} \end{array}$$

$$\frac{\mathtt{e} \rightarrow \mathtt{e}'}{E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{e}], G \overset{\text{LR}}{\rightsquigarrow} E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{e}'], G}$$

PROGRAM STATES

$$\begin{array}{rcccl} n & \in & \mathit{Node} & = & \mathit{NodeId} \times (\mathit{Process} + \bot) \\ n \mapsto n' & \in & \mathit{Edge} & = & \mathit{Node} \times \mathit{Node} \\ \delta & \in & \mathit{StableID} & & \\ \eta & \in & \mathit{CurNode} & = & \mathit{Thread} \overset{\text{fin}}{\rightarrow} \mathit{Node} \\ \sigma & \in & \mathit{StableSections} & = & \mathit{StableID} \overset{\text{fin}}{\rightarrow} \mathit{Node} \\ G & \in & \mathit{Graph} & = & \mathcal{P}(\mathit{Node}) \times \mathcal{P}(\mathit{Edge}) \times \mathit{CurNode} \times \mathit{StableSections} \end{array}$$

GLOBAL EVALUATION RULES

$$\frac{n = \text{ADDNODE}(\mathtt{t}[E[\mathtt{e}]]_{\phi}, \mathbf{N}) \quad G' = \langle\, \mathbf{N} \cup \{n\}, \mathbf{E} \cup \{\eta(\mathtt{t}) \mapsto n\}, \eta[\mathtt{t} \mapsto n], \sigma \,\rangle}{\mathtt{t}[E[\mathtt{e}]]_{\phi}, \langle\, \mathbf{N}, \mathbf{E}, \eta, \sigma \,\rangle \Downarrow G'}$$

$$\frac{n = \sigma(\delta) \quad n' = \text{ADDNODE}(\bot, \mathbf{N}) \quad G' = \langle\, \mathbf{N} \cup \{n'\}, \mathbf{E} \cup \{\eta(\mathtt{t}) \mapsto n', n' \mapsto n\}, \eta[\mathtt{t} \mapsto n'], \sigma \,\rangle}{\mathtt{t}[E[\mathtt{e}]]_{\delta.\overline{\delta}}, \langle\, \mathbf{N}, \mathbf{E}, \eta, \sigma \,\rangle \Downarrow G'}$$

$$\frac{\mathtt{t}[E[\mathtt{spawn(e)}]]_{\overline{\delta}}, G \Downarrow \langle\, \mathbf{N}, \mathbf{E}, \eta, \sigma \,\rangle \quad \mathtt{t}' \; \mathit{fresh} \quad n = \text{ADDNODE}(\mathtt{t}'[\mathtt{e}]_{\phi}, \mathbf{N}) \quad G' = \langle\, \mathbf{N} \cup \{n\}, \mathbf{E} \cup \{\eta(\mathtt{t}) \mapsto n\}, \eta[\mathtt{t}' \mapsto n], \sigma \,\rangle}{E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{spawn(e)}], G \overset{\text{SP}}{\rightsquigarrow} P \| \mathtt{t}[E[\mathtt{unit}]]_{\overline{\delta}} \| \mathtt{t}'[\mathtt{e}]_{\phi}, G'}$$

$$\frac{G = \langle\, \mathbf{N}, \mathbf{E}, \eta, \sigma \,\rangle \quad \delta \;\; \mathit{fresh} \quad n = \text{ADDNODE}(\mathtt{t}[\mathtt{E[stable}(\lambda\,\mathtt{x.e})(\mathtt{v})]]_{\overline{\delta}}, \mathbf{N}) \quad G' = \langle\, \mathbf{N}, \mathbf{E} \cup \{\eta(\mathtt{t}) \mapsto n\}, \eta[\mathtt{t} \mapsto n], \sigma[\delta \mapsto n] \,\rangle}{E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{stable}(\lambda\,\mathtt{x.e})(\mathtt{v})], G \overset{\text{SS}}{\rightsquigarrow} E^{\mathtt{t},P}_{\delta.\overline{\delta}}[\overline{\mathtt{stable}}(\mathtt{e[v/x]})], G'}$$

$$\frac{P = P' \| \mathtt{t}[E[\mathtt{send(l,v)}]]_{\overline{\delta}} \| \mathtt{t}'[E'[\mathtt{recv(l)}]]_{\overline{\delta}} \quad \mathtt{t}[E[\mathtt{send(l,v)}]]_{\overline{\delta}}, G \Downarrow G' \quad \mathtt{t}'[E[\mathtt{recv(l)}]]_{\overline{\delta}}, G' \Downarrow G'' \quad G'' = \langle\, \mathbf{N}, \mathbf{E}, \eta, \sigma \,\rangle \quad G''' = \langle\, \mathbf{N}, \mathbf{E} \cup \{\eta(\mathtt{t}) \mapsto \eta(\mathtt{t}'), \eta(\mathtt{t}') \mapsto \eta(\mathtt{t})\}, \eta, \sigma \,\rangle}{P, G \overset{\text{COMM}}{\rightsquigarrow} P' \| \mathtt{t}[E[\mathtt{unit}]]_{\overline{\delta}} \| \mathtt{t}'[E'[\mathtt{v}]]_{\overline{\delta}}, G'''}$$

$$\frac{G = \langle\, \mathbf{N}, \mathbf{E}, \eta, \sigma \,\rangle \quad G' = \langle\, \mathbf{N}, \mathbf{E}, \eta, \sigma - \{\delta\} \,\rangle}{E^{\mathtt{t},P}_{\delta.\overline{\delta}}[\overline{\mathtt{stable}}(\mathtt{v})], G \overset{\text{ES}}{\rightsquigarrow} E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{v}], G'}$$

$$\frac{\mathtt{t}[\mathtt{E[cut()]}]_{\overline{\delta}}, G \Downarrow G'}{E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{cut()}], G \overset{\text{CUT}}{\rightsquigarrow} E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{unit}], G'}$$

$$\frac{G = \langle\, \mathbf{N}, \mathbf{E}, \eta, \sigma \,\rangle \quad \sigma(\delta) = n \quad \tau = \text{REACH}(n, \mathbf{E}) \quad \langle\, k, \mathtt{t}[\mathtt{E[cut()]}] \,\rangle \notin \tau \quad P' = \{\mathtt{t}[\mathtt{e}] \mid \langle\, i, \mathtt{t}[\mathtt{e}] \,\rangle \in \tau \text{ s.t. } i \leq j \; \forall \langle\, j, \mathtt{t}[\mathtt{e}'] \,\rangle \in \tau\} \quad P'' = P' \oplus (P \ominus P') \quad G' = G/\tau}{E^{\mathtt{t},P}_{\delta.\overline{\delta}}[\mathtt{stabilize()}], G \overset{\text{ST}}{\rightsquigarrow} P'', G'}$$

$$\text{REACH}(n, \mathbf{E}) = \{n\} \cup \text{REACH}(n', \mathbf{E} - \{n \mapsto n'\}) \quad \forall n' \text{ s.t. } n \mapsto n' \in \mathbf{E}$$

Figure 9. Incremental Checkpoint Construction.

write is unrolled, the reading thread thread must be unrolled to at least before this read. To monitor writes, we create a version list in which we store reference/value pairs. For each reference in the list, its matching value corresponds to the value held in the reference prior to the execution of the stable section; our current implementation does not track writes occuring outside a stable section. When the program enters a stable section, we create an empty version list for this section. When a write is encountereted within a monitored procedure, a write barrier is executed that checks if the reference being written is in the verion list maintained by the section. If there is no entry for the reference, one is created, and the current value of the reference is recorded. Otherwise, no action is required.

Until a nested stable section exits, it is possible for a call to stabilize to unroll to the start of this section. A nested section is created when a monitored procedure is defined within the dynamic context of another monitored procedure. Nested sections require maintaining their own version lists. Version list information in these sections must be propagated to the outer section upon exit. However, the propagation of information form nested sections to outer ones is not trivial; if the outer section has monitored a particular memory location that has also been updated by the inner one, we only need to store the outer section's version, and the value preserved by the inner one can be discarded.

Efficiently monitoring read dependencies requires us to adopt a different methodology. We assume read operations occur much more frequently that writes, and thus it would be impractical to have barriers on all read operations to record dependency information in the communication graph. However, we observe that for a program to be correctly synchronized, all read and writes on a location l must be protected by a lock. Therefore, it is sufficient to monitor lock acquires/releases to infer shared memory dependencies. By incorporating happens-before dependency edges on lock operations, stabilize actions initiated by a writer to a shared location can be effectively propagated to readers that mediate access to that location via a common lock. A lock acquire is dependent on the previous acquisition of the lock.

6.3 Graph Representation

The main challenge in the implementation was developing a compact representation of the communication graph. We have implemented a number of node/edge compaction algorithms allowing for fast culling of redundant information. For instance, any two nodes that share a backedge can be collapsed into a single node. We also ensure that there is at most one edge between any pair of nodes. Any addition to the graph affects at most two threads. We use thread-local meta-data to find the most recent node for each

Benchmark	LOC incl. eXene	Threads	Channels	Comm. Events	Shared Writes	Shared Reads	Graph Size(MB)	Overheads (%) Runtime	Overheads (%) Memory
`Triangles`	16501	205	79	187	88	88	.19	0.59	8.62
`N-Body`	16326	240	99	224	224	273	.29	0.81	12.19
`Pretty`	18400	801	340	950	602	840	.74	6.23	20.00
`Swerve`	9915	10532	231	902	9339	80293	5.43	2.85	4.08

Table 1. Benchmark characteristics, dynamic counts, and normalized overheads.

thread. The graph is thus never traversed in its entirety. The size of the communication graph grows with the number of communication events, thread creation actions, lock acquires, and stable sections entered. However, we do not need to store the entire graph for the duration of program execution. As the program executes, parts of the graph will become unreachable. The graph is implemented using weak references to allow unreachable portions to be safely reclaimed by the garbage collector. As we describe below, memory overheads are thus minimal.

A `stabilize` action has complexity linear in the number of nodes and edges in the graph. Our implementation utilizes a combination of depth-first search and bucket sorting to calculate the resulting graph after a stabilize call in linear time. DFS identifies the part of the graph which will be removed after the stabilize call and a modified bucket sort actually performs the removal. Only sections of the graph reachable from the stabilize call are traversed, resulting in a fast restoration procedure.

7. Performance Results

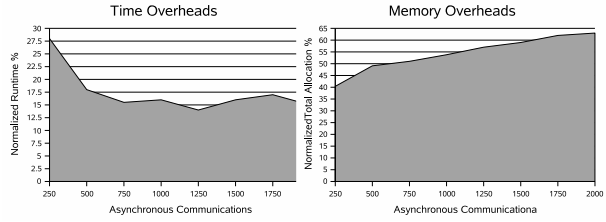
To measure the cost of stabilizers with respect to various concurrent programming paradigms, we present a synthetic benchmark to quantify pure memory and time overheads, and examine several server-based open-source CML benchmarks to illustrate average overheads in real programs. The benchmarks were run on a Intel P4 2.4 GHz machine with one GByte of memory running Gentoo Linux, compiled and executed using MLton release 20041109.

To measure the costs of our abstraction, our benchmarks are executed in three different ways: one in which the benchmark is executed with no actions monitored, and no checkpoints constructed; one in which the entire program is monitored, effectively wrapped within a `stable` call, but in which no checkpoints are actually restored; and one in which relevant sections of code are wrapped within stable sections, exception handlers dealing with transient faults are augmented to invoke `stabilize`, and faults are dynamically injected to trigger restoration.

7.1 Synthetic Benchmarks

Our synthetic benchmark spawns two threads, a source and a sink, that communicate asynchronously. We measure the cost of our abstraction with regard to an ever increasing load of asynchronous communication events. This benchmark measures the overhead of logging program state and communication dependencies with *no* opportunity for amortizing these costs among other non-stabilizer related operations. These numbers represent worst case overheads for monitoring thread interactions.

The runtime overhead is presented in Fig. 10(a), and the total allocation overhead is presented in Fig. 10(b). As expected, the cost to simply maintain the graph grows linearly with the number of asynchronous communications performed and runtime overheads remain constant. There is a significant initial memory and runtime cost because we pre-allocate hash tables used to index nodes in the graph.



(A) (B)

Figure 10. Synthetic benchmark overheads.

7.2 Open-Source Benchmarks

Our other benchmarks include several `eXene` [16] benchmarks, `Triangles` and `Nbody`, mostly display programs that create threads to draw objects; and `Pretty`, a pretty printing library written on top of `eXene`. The `eXene` toolkit is a library for X Windows, implementing the functionality of `xlib`, written in CML and comprising roughly 16K lines of Standard ML. Events from the X server and control messages between widgets are distributed in streams (coded as CML event values) through the window hierarchy. `eXene` manages the X calls through a series of servers, dynamically spawned for each connection and screen. The last benchmark we consider is `Swerve`, a webserver written in CML whose major modules communicate with one other using message-passing channel communication; it makes no use of `eXene`. All the benchmarks create various CML threads to handle various events; communication occurs mainly through a combination of message-passing on channels, with occasional updates to shared data.

For these benchmarks, stabilizers exhibit a runtime slow down upto approximately 6% over a CML program in which monitoring is not performed (see Table 1). For a highly-concurrent application like `Swerve`, the overheads are even smaller, on the order of 3%. The cost of using stabilizers is only dependent on the number of inter-thread actions and shared data dependencies that are logged. These overheads are well amortized over program execution.

Memory overheads to maintain the communication graph are larger, although in absolute terms, they are quite small. Because we capture continuations prior to executing communication events and entering stable sections, part of the memory cost is influenced by representation choices made by the underlying compiler. Nonetheless, benchmarks such as `Swerve` that create over 10K threads, and employ non-trivial communication patterns, require only 5MB to store the communication graph, a roughly 4% overhead over the memory consumption of the original program.

To measure the cost of calculating and restoring a globally consistent checkpoint, we consider three experiments. The first is a simple unrolling of `Swerve` (see Table. 2), in which a call to `stabilize` is inserted during the processing of a varying number of concurrent web requests. This measurement illustrates the cost of restoring to a consistent global state that can potentially affect a large number of threads. Although we expect large checkpoints to be rare, we note that restoration of such checkpoints is nonetheless

Reqs	Graph Size	Channels Num	Channels Cleared	Threads Affected	Runtime (seconds)
20	1130	85	42	470	0.005
40	2193	147	64	928	0.019
60	3231	207	84	1376	0.053
80	4251	256	93	1792	0.094
100	5027	296	95	2194	0.132

Table 2. Restoration of the entire webserver.

Benchmark	Channels Num	Channels Cleared	Threads Total	Threads Affected	Runtime (seconds)
`Swerve`	38	4	896	8	.003
`eXene`	158	27	1023	236	.019

Table 3. Instrumented recovery.

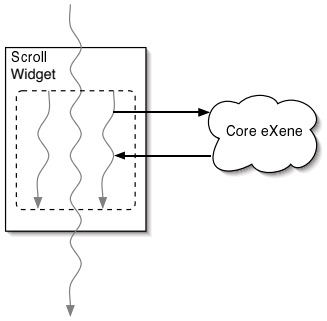


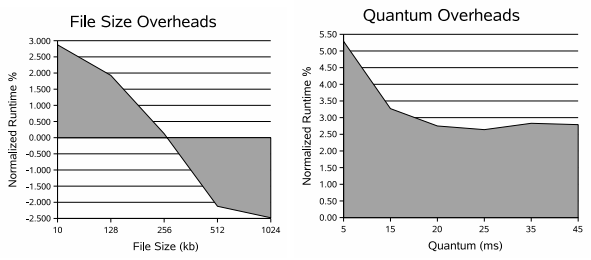
Figure 11. An `eXene` scroll bar widget spawns several independent threads, including a control thread that communicates with other `eXene` components.

quite fast. The graph size is presented as the total number of nodes. Channels can be affected by an unrolling in two different ways: a channel may contain a value sent on it by a communicating thread but which has not been consumed by a receiver, or a channel may connect two threads which have successfully exchanged a value. In the first case we must clear the channel of the value if the thread which placed the value on the channel is unrolled; in the later case no direct processing on the channel is required. The table also shows the total number of affected channels and those which must cleared.

7.3 Injecting Stabilizers

To quantify the cost of using stabilizers in practice, we extended `Swerve` and `eXene` and replaced some of their error handling mechanisms with stabilizers (see Table 3). For `Swerve`, the implementation details were given in Section 3. Our benchmark manually injects a timeout every ten requests, stabilizes the program, and re-requests the page.

For `eXene`, we augment a scrollbar widget used by the pretty printer. In `eXene` the state of a widget is defined by the state of its communicating threads, and no state is stored in shared data. The scroll bar widget is composed of three threads which communicate over a set of channels. The widget's processing is split between two helper threads and one main controller thread. Any error handled by the controller thread must be communicated to the helper threads and vice versa. The interactions of the scroll bar widget and the rest of `eXene` is depicted in Fig. 11. The dotted box represents a stable section encompassing the processing of the widget. We manually



(A) (B)

Figure 12. `Swerve` benchmark overheads.

inject the loss of packets to the `X` server, stabilize the widget, and wait for new interaction events. The loss of packets is injected by simply dropping every tenth packet which is received from the `X` server. Ordinarily, if `eXene` ever loses an X server packet, its default behavior is to terminate execution since there is no easy mechanism available to restore the state of the widget to a globally consistent point. Using stabilizers, however, packet loss exceptions can be safely handled by the widget. By stabilizing the widget, we return it to a state prior to the failed request. Subsequent requests will redraw the widget as we would expect; thus, stabilizers permit the scroll bar widget to recover from a lost packet without pervasive modification to the underlying `eXene` implementation.

Finally, to measure the sensitivity of stabilization to application-specific parameters, we compare our stabilizer-enabled version of `Swerve` to the stock configuration by varying two program attributes: file size and quantum. Since stabilizers eliminate the need for polling during file processing, we suspect our runtime costs would improve as file sizes increased. Our tests were run on both versions of `Swerve`; for a given file size, 20 requests are processed. The results (see Fig. 12(a)) indicate that for large file sizes (upward of 256KB) our implementation is slightly more efficient than the original implementation. Our slow down for small file sizes (on the order of 10KB) is proportional to our earlier benchmark results.

Since our graph algorithm requires monitoring various communication events, lowering the time quantum allocated to each thread may adversely affect performance, since the overhead for monitoring the graph consumes a greater fraction of a thread's computation per quantum. Our tests compared the two versions of `Swerve`, keeping file size constant at 10KB, but varying the allocated quantum. (see Fig 12(B)). Surprisingly, the results indicate that stablizer overheads become significant only when the quantum is less than 5 ms. As a point of comparison, CML's default quanta is 20ms.

8. Related Work

Being able to checkpoint and rollback parts or the entirety of an execution has been the focus of notable research in the database [10] as well as parallel and distributed computing communities [13, 22, 24]. Checkpoints have been used to provide fault tolerance for long-lived applications, for example in scientific computing [28, 2] but have been typically regarded as heavyweight entities to construct and maintain.

Existing checkpoint approaches can be classified into four broad categories: (a) schemes that require applications to provide their own specialized checkpoint and recovery mechanisms [4, 5]; (b) schemes in which the compiler determines where checkpoints can be safely inserted [3]; (c) techniques that require operating system or hardware monitoring of thread state [8, 21, 24]; and (d) library implementations that capture and restore state [11]. Checkpointing functionality provided by an application or a library relies on the programmer to define meaningful checkpoints. For many multi-

threaded applications, determining these points is non-trivial because it requires reasoning about global, rather than thread-local, invariants. Compiler and operating-system injected checkpoints are transparent to the programmer. However, transparency comes at a notable cost: checkpoints may not be semantically meaningful or efficient to construct.

Recent work in the programming languages community has explored abstractions and mechanisms closely related to stabilizers and their implementation for maintaining consistent state in distributed environments [14], detecting deadlocks [9], and gracefully dealing with unexpected termination of communicating tasks in a concurrent environment [15]. For example, kill-safe thread abstractions [15] provide a mechanism to allow cooperating threads to operate even in the presence of abnormal termination. Stabilizers can be used for a similar goal, although the means by which this goal is achieved is quite different. Stabilizers rely on unrolling thread dependencies of affected threads to ensure consistency instead of employing specific runtime mechanisms to reclaim resources.

In addition to stabilizers, functional language implementations have utilized continuations for similar tasks. For example, Tolmach and Appel [29] describe a debugging mechanism for SML/NJ that utilized captured continuations to checkpoint the target program at given time intervals. This work was later extended [30] to support multithreading, and was used to log non-deterministic thread events to provide replay abilities.

Another possibility for fault recovery is micro-reboot [7], a fine-grained technique for surgically recovering faulty application components which relies critically on the separation of data recovery and application recovery. Micro-reboot allows for a system to be restarted without ever being shut down by rebooting separate components. Unlike checkpointing schemes, which attempt to restore a program to a consistent state within the running application, micro-reboot quickly restarts an application component, but the technique itself is oblivious to program semantics.

The ability to revert to a prior point within a concurrent execution is essential to transaction systems [1, 16, 23]; outside of their role for database concurrency control, such approaches can improve parallel program performance by profitably exploiting speculative execution [26, 31]. Harris *et al.* proposes a transactional memory system [18] for Haskell that introduces a `retry` primitive to allow a transactional execution to safely abort and be re-executed if desired resources are unavailable. However, this work does not propose to track or revert effectful thread interactions within a transaction. In fact, such interactions are explicitly rejected by the Haskell type-system. There has also been recent interest in providing transactional infrastructure for ML [27], and in exploring the interaction between transactional semantics and first-class synchronous operations [12]. Our work shares obvious similarities with all these efforts.

9. Conclusions and Future Work

Stabilizers are a novel on-the-fly checkpointing abstraction for concurrent functional programs. Unlike other checkpointing schemes, stabilizers are not only able to identify the smallest subset of threads which must be unrolled, but also provide useful safety guarantees. As a language abstraction, stabilizers can be used to simplify program structure especially with respect to error handling, debugging, and consistency management. Our results indicate that stabilizers can be implemented with small overhead and thus serve as an effective and promising checkpointing abstraction for high-level concurrent programs.

There are several important directions we expect to pursue in the future. While the use of `cut` can delimit the extent to which control is reverted as a result of a `stabilize` call, a more general and robust approach would be to integrate a rational compensation semantics [6] for stabilizers in the presence of stateful operations. We also plan to explore richer ways to describe the interaction between stable sections and their restoration, for example by providing a facility to have threads restore program state in other executing threads, and to investigate the interaction of stabilizers with other transaction-based concurrency control mechanisms.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 24(2):23–34, June 1995.

[2] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive Incremental Checkpointing for Massively Parallel Systems. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 277–286, New York, NY, USA, 2004. ACM Press.

[3] Micah Beck, James S. Plank, and Gerry Kingsley. Compiler-Assisted Checkpointing. Technical report, University of Tennessee, Knoxville, TN, USA, 1994.

[4] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated Application-Level Checkpointing of MPI Programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 84–94, New York, NY, USA, 2003. ACM Press.

[5] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-Level Checkpointing for Shared Memory Programs. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 235–247, New York, NY, USA, 2004. ACM Press.

[6] R. Bruni, H. Melgratti, and U. Montanari. Theoretical Foundations for Compensations in Flow Composition Languages. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*, pages 209–220, New York, NY, USA, 2005. ACM Press.

[7] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - A Technique for Cheap Recovery. In *6th Symposium on Operating Systems Design and Implementation*, San Francisco, California, 2004.

[8] Yuqun Chen, James S. Plank, and Kai Li. CLIP: A Checkpointing Tool for Message-Passing Parallel Programs. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, pages 1–11, New York, NY, USA, 1997. ACM Press.

[9] Jan Christiansen and Frank Huch. Searching for Deadlocks while Debugging Concurrent Haskell Programs. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 28–39, New York, NY, USA, 2004. ACM Press.

[10] Panos K. Chrysanthis and Krithi Ramamritham. ACTA: the SAGA continues. In *Database Transaction Models for Advanced Applications*, pages 349–397. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[11] William R. Dieter and James E. Lumpp Jr. A User-level Checkpointing Library for POSIX Threads Programs. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 224, Washington, DC, USA, 1999. IEEE Computer Society.

[12] Kevin Donnelly and Matthew Fluet. Transactional events. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, New York, NY, USA, 2006. ACM Press.

[13] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[14] John Field and Carlos A. Varela. Transactors: a Programming Model for Maintaining Globally Consistent Distributed State in Unreliable Environments. In *POPL '05: Proceedings of the 32nd*

ACM SIGPLAN-SIGACT sysposium on Principles of programming languages, pages 195–208, New York, NY, USA, 2005. ACM Press.

[15] Matthew Flatt and Robert Bruce Findler. Kill-safe Synchronization Abstractions. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 47–58, New York, NY, USA, 2004. ACM Press.

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

[17] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 388–402. ACM Press, 2003.

[18] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. In *ACM Conference on Principles and Practice of Parallel Programming*, 2005.

[19] 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.

[20] http://www.mlton.org.

[21] D. Hulse. On Page-Based Optimistic Process Checkpointing. In *IWOOOS '95: Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems*, page 24, Washington, DC, USA, 1995. IEEE Computer Society.

[22] Mangesh Kasbekar and Chita Das. Selective Checkpointing and Rollback in Multithreaded Distributed Systems. In *21^{st} International Conference on Distributed Computing Systems*, 2001.

[23] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *TODS*, 6(2):213–226, 1981.

[24] Kai Li, Jeffrey Naughton, and James Plank. Real-time Concurrent Checkpoint for Parallel Programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, 1990.

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

[26] Martin Rinard. Effective Fine-Grained Synchronization for Automatically Parallelized Programs Using Optimistic Synchronization Primitives. *ACM Transactions on Computer Systems*, 17(4):337–371, November 1999.

[27] Michael F. Ringenburg and Dan Grossman. Atomcaml: first-class atomicity via rollback. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 92–104, New York, NY, USA, 2005. ACM Press.

[28] Asser N. Tantawi and Manfred Ruschitzka. Performance Analysis of Checkpointing Strategies. *ACM Trans. Comput. Syst.*, 2(2):123–144, 1984.

[29] Andrew P. Tolmach and Andrew W. Appel. Debugging Standard ML Without Reverse Engineering. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 1–12, New York, NY, USA, 1990. ACM Press.

[30] Andrew P. Tolmach and Andrew W. Appel. Debuggable Concurrency Extensions for Standard ML. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 120–131, New York, NY, USA, 1991. ACM Press.

[31] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 439–453. ACM Press, 2005.

[32] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional Monitors for Concurrent Objects. In *European Conference on Object-Oriented Programming*, pages 519–542, 2004.