2005

Stabilizers: Safe Lightweight Check-pointing for Concurrent Programs

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

Philip Schatz

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

Report Number:
05-023

http://docs.lib.purdue.edu/cstech/1637

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
STABILIZERS: SAFE LIGHTWEIGHT CHECKPOINTING FOR CONCURRENT PROGRAMS

Lukasz Ziarek
Philip Schatz
Suresh Jagannathan

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #05-023
November 2005
Stabilizers: Safe Lightweight Checkpointing for Concurrent Programs

Lukasz Ziarek  Philip Schatz  Suresh Jagannathan
Department of Computer Science
Purdue University
{lziarek,schatzp,suresh}@cs.purdue.edu

Abstract
A checkpoint is a mechanism that allows program execution to be restarted from a previously saved state. Checkpoints can be used in conjunction with exception handling abstractions to recover from exceptional or erroneous events, to support debugging or replay mechanisms, or to facilitate algorithms that rely on speculative evaluation. While relatively straightforward in a sequential setting, for example through the capture and application of continuations, it is less clear how to ascribe a meaningful semantics for lightweight and safe checkpoints in the presence of concurrency. For a thread to correctly resume execution from a saved checkpoint, it must ensure that all other threads which have witnessed its unwanted effects after the checkpoint was established are also reverted to a meaningful earlier state. If this is not done, data inconsistencies and other undesirable behavior may result. However, automatically determining what constitutes a consistent global state is not straightforward since thread interactions are a dynamic property of the program; requiring applications to specify such states explicitly is not pragmatic if interactions are complex.

In this paper, we present a safe and efficient on-the-fly checkpointing mechanism for concurrent programs. We introduce a new abstraction called stabilizers that permits the specification and restoration of globally consistent checkpoints. This state is computed through lightweight monitoring of communication events among threads (e.g., message-passing operations or updates to shared variables). Our implementation results show that the memory and computation overheads for using stabilizers on highly-concurrent server applications is small, averaging roughly 4 to 6%, leading us to conclude that stabilizers are a viable abstraction for defining restorable checkpoint state in complex concurrent programs.

Keywords: Concurrent programming, checkpointing, consistency, continuations, exception handling, message-passing, shared memory.

1. Introduction

Checkpointing mechanisms allow applications to preserve state, and to resume execution from saved states when necessary. Checkpoints have obvious utility for error recovery [28], program replay and debugging [?]; they can be used to support applications that engage in transactional behavior [?], speculative execution [27] or persistence [13, 32]; and, they can be used to build exception handlers that restore memory to a previous state [?]. In functional languages, continuations provide a simple checkpointing facility: defining a checkpoint corresponds to capturing a continuation [35], and restoring a checkpoint corresponds to invoking this continuation. In such a scheme, resuming a checkpointed computation with possibly different results, requires simply supplying different values to the saved continuation that represents the checkpoint in question. When references are involved, a sensible checkpoint state would need to also store their values along with the continuation when the checkpoint is taken.

Unfortunately, defining and manipulating checkpoints becomes significantly more complex in the presence of concurrency. A thread that wishes to establish a checkpoint can simply save its local state, but of course there is no guarantee that the global state of the program is consistent if control ever reverts back to this point. For example, suppose a communication event via message-passing occurs between two threads and the sender subsequently rolls back control to a local checkpointed state established prior to the communication. A spurious unhandled execution of the (re)sent message may result because the receiver has no knowledge that a rollback of the sender has occurred, and thus has no need to expect retransmission of a previously executed message. A simple remedy to this problem would require the state of all active threads to be simultaneously recorded whenever any thread establishes a checkpoint. While this solution is sound, it can lead to substantial inefficiencies and complexity. A thread that establishes a checkpoint and performs actions prior to a rollback may induce effects on other threads by engaging in communication actions (e.g., sending and receiving messages) with them; these threads are necessarily required to revert to an earlier consistent state as a result of the rollback. On the other hand, there may be other threads unaffected by the checkpointed thread’s actions. A scheme that fails to take recognize these distinctions would be unnecessarily conservative in its treatment of rollback, and would be inefficient in practice, especially if checkpoints are reverted often.

In general, the problem of computing a sensible checkpoint requires computing the transitive closure of dependencies that manifest among threads from the time the checkpoint is established to the time it is invoked. If a thread T1 establishes a checkpoint at program point p, and attempts to revert control back to p at some later point p’, any thread T2 that witnesses T1’s effects between p and p’
is subject to rollback as well. Indeed, the rollback operation applies transitively to threads that indirectly witness \( T_1 \)'s effects via any communication with \( T_2 \) that occurs within this interval.

Existing checkpoint approaches can be classified into four broad categories: (a) schemes that require applications to provide their own specialized checkpoint and recovery mechanisms [5, 6]; (b) schemes in which the compiler determines where checkpoints can be safely inserted [4]; (c) checkpoint strategies that require operating system or hardware monitoring of thread state [9, 23, 26]; and (d) library implementations that capture and restore state [14].

Checkpointing functionality provided by an application or a library relies on the programmer to define meaningful checkpoints. 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. If all application threads run within the same process, saving and restoring checkpoints may be expensive since only a small number of threads may be affected as a result of a rollback. If application threads run in separate processes, each process may get checkpointed at different intervals, violating the need for serialization.

To alleviate the burden of defining and restoring safe checkpoints in a concurrent program, we propose a new language abstraction for dynamic, composable on-the-fly checkpointing called stabilizers. Stabilizers encapsulate two operations, one to initiate monitoring of code for communication and thread creation events, and to establish a thread-local checkpoint when the code is evaluated; and the other to revert control and state to a safe global checkpoint. The checkpoints defined by stabilizers are composable: a monitored procedure can freely create and return other monitored procedures. Our checkpointing mechanism is a middle ground between the transparency afforded by operating systems or compilers, and the precision afforded by user-injected checkpoints. In our approach, applications are required to identify meaningful per-thread program points where a checkpoint may be performed, when a rollback operation occurs, control reverts to one of these saved checkpoints for each thread.

The exact checkpoint chosen is calculated dynamically based on monitoring communication patterns among executing threads. Suppose a checkpoint is established at program point \( p \) in thread \( t \) and a rollback is initiated at point \( p' \) following \( p \). Between \( p \) and \( p' \), \( t \) may be engaged in a number of communication events with other threads. To ensure that \( t \)'s re-execution from \( p \) is meaningful, the execution of those threads with which \( t \) has communicated must also be reverted to a stable point, a point that does not reflect any of \( t \)'s visible actions between \( p \) and \( p' \). Of course, reverting these threads to an earlier checkpoint state may in-turn require \( t \) to rollback its execution before \( p \). This may happen if these other threads revert to a checkpoint state that itself initiates communication with \( t \) prior to \( p \). Once a global stable state is discovered, all threads can resume execution from their respective program points in that state. Our approach guarantees that when a thread is rolled-back to a checkpointed state \( C \), other threads with which it has communicated prior to its last rollback are in states consistent with \( C \). No action is taken for threads that have not been influenced by \( t \)'s effects.

To calculate how to revert threads to safe checkpoints, the runtime system must keep track of thread states and trace communication events among threads. When a spawn or communication event occurs, information is recorded in a runtime data structure about the event as well as the thread's continuation prior to the event. Our checkpointing facility tracks thread interaction only through explicit communication events they generate (e.g., message-passing operations actions on channels that provide a communication medium between threads, or implicit communication through reads and writes of shared variables).

When a rollback action occurs, the runtime-maintained data structure is consulted to determine the proper checkpoint for all threads that maintains global consistency. A rollback is sensible only if re-execution results in a different execution path than the one that caused the rollback to occur initially. Thus, our solution critically relies on non-deterministic behavior: to ensure that rollbacks do not simply lead to infinite looping, subsequent re-execution of threads should lead to different thread interactions and behavior. For most multi-threaded programs, this requirement is not particularly onerous. However, to allow applications further control over the state in which a checkpoint resumes, stabilizers also come equipped with a simple compensation mechanism that may be executed before control is reverted to the checkpointed state (see Section 7?). Compensations also allow stabilizers to work in the presence of non-restorable actions such as I/O.

Like transactions, stabilizers provide a pleasant consistency guarantee. When control is reverted to the beginning of a monitored region, other threads which have witnessed (either directly or indirectly) effects performed within that region are unrolled as well, regardless of whether they themselves are executing within a stable section. Computation not dependent on the monitored region of code is unaffected. Thus, unlike transactions, the collection of threads that are affected by a restore action is dynamically determined. There are no issues related to livelock or deadlock in restoring a checkpoint: reverting to an earlier checkpoint is always guaranteed to succeed.

1.1 Stabilizers

To alleviate the burden of defining and restoring safe checkpoints in a concurrent program, we propose a new language abstraction for dynamic, composable on-the-fly checkpointing called stabilizers. Stabilizers encapsulate two operations, one to initiate monitoring of code for communication and thread creation events, and to establish a thread-local checkpoint when the code is evaluated; and the other to revert control and state to a safe global checkpoint. The checkpoints defined by stabilizers are composable: a monitored procedure can freely create and return other monitored procedures. Our checkpointing mechanism is a middle ground between the transparency afforded by operating systems or compilers, and the precision afforded by user-injected checkpoints. In our approach, applications are required to identify meaningful per-thread program points where a checkpoint may be performed, when a rollback operation occurs, control reverts to one of these saved checkpoints for each thread.

The exact checkpoint chosen is calculated dynamically based on monitoring communication patterns among executing threads. Suppose a checkpoint is established at program point \( p \) in thread \( t \) and a rollback is initiated at point \( p' \) following \( p \). Between \( p \) and \( p' \), \( t \) may have engaged in a number of communication events with other threads. To ensure that \( t \)'s re-execution from \( p \) is meaningful, the execution of those threads with which \( t \) has communicated must also be reverted to a stable point, a point that does not reflect any of \( t \)'s visible actions between \( p \) and \( p' \). Of course, reverting these threads to an earlier checkpoint state may in-turn require \( t \) to rollback its execution before \( p \). This may happen if these other threads revert to a checkpoint state that itself initiates communication with \( t \) prior to \( p \). Once a global stable state is discovered, all threads can resume execution from their respective program points in that state. Our approach guarantees that when a thread is rolled-back to a checkpointed state \( C \), other threads with which it has communicated prior to its last rollback are in states consistent with \( C \). No action is taken for threads that have not been influenced by \( t \)'s effects.

To calculate how to revert threads to safe checkpoints, the runtime system must keep track of thread states and trace communication events among threads. When a spawn or communication event occurs, information is recorded in a runtime data structure about the event as well as the thread's continuation prior to the event. Our checkpointing facility tracks thread interaction only through explicit communication events they generate (e.g., message-passing operations actions on channels that provide a communication medium between threads, or implicit communication through reads and writes of shared variables).

When a rollback action occurs, the runtime-maintained data structure is consulted to determine the proper checkpoint for all threads that maintains global consistency. A rollback is sensible only if re-execution results in a different execution path than the one that caused the rollback to occur initially. Thus, our solution critically relies on non-deterministic behavior: to ensure that rollbacks do not simply lead to infinite looping, subsequent re-execution of threads should lead to different thread interactions and behavior. For most multi-threaded programs, this requirement is not particularly onerous. However, to allow applications further control over the state in which a checkpoint resumes, stabilizers also come equipped with a simple compensation mechanism that may be executed before control is reverted to the checkpointed state (see Section 7?). Compensations also allow stabilizers to work in the presence of non-restorable actions such as I/O.

Like transactions, stabilizers provide a pleasant consistency guarantee. When control is reverted to the beginning of a monitored region, other threads which have witnessed (either directly or indirectly) effects performed within that region are unrolled as well, regardless of whether they themselves are executing within a stable section. Computation not dependent on the monitored region of code is unaffected. Thus, unlike transactions, the collection of threads that are affected by a restore action is dynamically determined. There are no issues related to livelock or deadlock in restoring a checkpoint: reverting to an earlier checkpoint is always guaranteed to succeed.

1.2 Contributions

This paper makes three contributions:

1. The design and semantics of stabilizers, a new language abstraction for defining and restoring meaningful checkpoints in concurrent programs in which threads communicate through both message-passing and shared memory. To the best of our knowledge, stabilizers are the first language-centric design of a checkpointing facility for concurrent programs that provides global consistency and safety guarantees when checkpointed state is restored.

2. A lightweight dynamic monitoring algorithm faithful to the semantics that constructs optimal checkpoints based on the context in which a restore action is performed. Optimality is defined with respect to the amount of rollback required to ensure that all threads resume execution after a checkpoint is restored in a consistent global state.

3. A detailed evaluation study in SML that quantifies the cost of using stabilizers on various server-side applications. Our results reveal that the cost of defining and monitoring thread state is small, typically adding roughly 6% overhead to overall execution time and about 10% memory overhead.

The remainder of the paper is structured as follows. In Section 2, we provide a motivating example that highlights the issues associated with safely checkpointing computation in concurrent programs. Section 3 describes the stabilizer abstraction, and runtime extensions used to support it. An operational semantics is given in Section 4. Implementation details are provided in Section 6. A detailed evaluation on the costs and overheads of using stabilizers is given in Section 7. Related work is presented in Section 8, and conclusions are given in Section 9.
2. Motivating Example

To motivate the use of stabilizers, consider the program fragment shown below. The program spawns a new asynchronous thread of control to compute the application of \( f \) to argument \( \text{arg} \). Function \( f \) in turn spawns a thread to compute the application of \( g \) to argument \( \text{arg}' \), and sends data on a channel \( c \) that may potentially be read by \( g \). In addition, \( g \) also reads data from channel \( c' \) that is not accessed by \( f \). Assume channels are synchronous, and thus sends and receives block if there is no matching recipient or sender (resp). In the example, both \( f \) and \( g \) can raise a Timeout exception. The desired behavior when a timeout occurs is to re-execute \( f \) presumably with a different argument, ensuring that none of \( f \)'s earlier effects remain visible when it is reapplied.

\[
\text{let val } c = \text{mkCh()}; \quad \text{end; let val } c' = \text{mkCh()}; \quad \text{end; fun } g \; y = \ldots \text{recv}(c) \ldots \text{recv}(c') \ldots \text{raise Timeout} \ldots \text{in handle Timeout =>} \ldots \text{fun } f \; x = \text{let val } _x = \text{spawn}(g(\text{arg}')) \text{val } _x = \text{send}(c,x) \ldots \text{in if ... then raise Timeout else ... end handle Timeout =>} \ldots \text{in spawn}(f(\text{arg})); \quad \text{end.}
\]

Ordinarily, an exception handler will not be able to restore the global program state such that \( f \) can be re-executed safely. Notice that \( f \) not only spawns a new thread, but also communicates data along channel \( c \). Simply reexecuting \( f \) without rewriting \( c \)'s receivers would be obviously incorrect. Furthermore, any thread such as \( g \) that reads a value communicated by \( f \) may store that value, propagate it to other threads, or perform arbitrary computation based on that value. If \( f \)'s reexecution propagates a new value for \( x \), its previous effects are no longer valid. Thus, to ensure that the execution of the handler results in a benign global state, all threads potentially affected by \( f \)'s communication on \( c \) must be identified. However, the handler's obligations do not end here. For example, consider thread \( g \) that also receives data from channel \( c' \). If \( g \) is reverted because it read data produced by \( f \), then the communication it established on channel \( c' \) is also suspect: reversion \( g \) without clearing that communication could lead to inconsistencies; the sender on \( c' \) assumes that the value it produced has been consumed, but \( g \)'s reexecution would effectively forget its receipt. Observe that both \( f \) and \( g \) provide their own local Timeout handlers; propagating the effect of a timeout exception raised locally involves restructuring the program to communicate such events among concurrently executing threads. Because the various scenarios that may arise depends upon runtime scheduling decisions, any scheme that purports to allow safe reversion of a previously executed computation must dynamically discovers safe states for all affected threads.

3. Programming Model

To dynamically calculate globally consistent states, we introduce a new abstraction called stabilizers. Stabilizers are expressed using two primitives, stable and stabilize, with the following signatures:

\[
\text{stable : ('a -> 'b) -> 'a -> 'b} \\
\text{stabilize : unit -> unit}
\]

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 \( \text{stable } f \) yields a new function \( f' \) identical to \( f \) except that interesting communication, shared memory access, locks, and spawn events are monitored and grouped.

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 noting that external non-revocable actions that occur within a stable section that must be reverted (e.g., I/O, foreign function calls, etc.) must be handled explicitly by the application through a compensation mechanism described in Section ??.

Unlike classical checkpointing schemes [34] 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, or the event prior to the stabilize call if it occurs outside a stable section.

An important design feature of stabilizers is that the are composable: there is no \textit{a priori} classification of the procedures that need to be monitored, nor is there any restriction against nesting stable sections. Moreover, 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; state saving and restoration decisions are determined without prejudice to the behavior of other monitored procedures.

3.1 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 unrolled. If a thread spawns another thread within a stable section that is being reverted, this new thread (and all its actions) must also be discarded. All threads which read from a shared variable must be unrolled if the thread that wrote the value is reverted to a state prior to the write. A program state is \textit{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).

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 $S_2$. 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 $S_1$, then a consistent global state would require that $t_2$ revert back to the state associated with the start of $S_2$ since it has received a communication from $t_1$ initiated within $S_2$. However, discarding the actions within $S_2$ 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 $S_2$). 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 $t_1$ to, since the stable section $S_3$ spans communication events from both $S_1$ and $S_2$.

3.2 Example

Consider a real world example of stabilizers take from the Sverwe webservice given in Fig 2 and a modification of this code to use stabilizers. The function `sendFile()` send a requested file to a client. The original code checks in each iteration of the function `loop` if the request has timeout. If the timeout has occurred, the `sendFile()` function is obligated to notify the consumer through an explicit send on channel consumer. The consumer must then notify all modules he communicates with so that they may handle their timeout explicitly. Stabilizers allow us to abstract this explicit notification procedure. We can wrap the `loop()` function in a stable section and replace the explicit send on channel consumer with a call to `stabilize()`. If a timeout occurs, we will simply re-execute the loop and the consumer will receive the file. However, we can do better than this. By removing the if `Abort.abort` abort and the entire then branch, we can avoid checking in each iteration of the loop if a timeout has occurred. Since the `Abort` module relies on CML’s time events to discover if it has timedout, we can wrap the appropriate timeout event with a call to `stabilize()`. CML wrap (timeEvt, stabilize()). When the event triggers a timeout, it will call stabilize causing the `sendFile()` to revert to the start of its stable section and causing the consumer to revert prior to seeing the file as well as any threads which transitively depend on the consumer. In this case, we were able to abstract the entire timeout handling functionality by a simple use of stabilizers. Notice, only a few lines of code needed to be changed. Additional actions that need to be taken for a timeout can be modeled as compensations to stable sections. We could easily wrap the consumer in a stable section and provide the appropriate compensation code.

4. Semantics

Our semantics is defined in terms of a core call-by-value functional language with threading primitives (see Fig. 2). 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. To this core language we add two new primitives: `stable` and `stabilize()`. The expression `stable($x.e$)` creates a stable function, $\lambda x.e$ whose effects are monitored. When a stable function is applied, a global checkpoint is established, and its body, denoted as `stable(e)`, is evaluated in the context of this checkpoint. The second primitive, `stabilize()`, is used to initiate a rollback.

In the following, we use metavariables $v$ to range over values, and $\delta$ to range over stable section identifiers. We also use $P$ for thread terms, and $e$ for expressions. We use over-bar to represent a finite ordered sequence, for instance, $\overline{f_1 f_2 \ldots f_n}$. The term $\alpha.\overline{\alpha}$ denotes the prefix extension of the sequence $\overline{\alpha}$ with a single element $\alpha$, $\overline{\alpha}$. The suffix extension, $\overline{\alpha}$ denotes sequence concatenation, $\phi$ denotes an empty sequence, and $\overline{\alpha} \leq \overline{\beta}$ holds if $\overline{\alpha}$ is a prefix of $\overline{\beta}$. We write $| \overline{\alpha} |$ to denote the length of sequence $\overline{\alpha}$.

The syntax and semantics of the language are given in Figure 3. 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 $\delta$) 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. Thus, we write $\tau | e | \tau$ if a thread with identifier $\tau$ is executing expression $e$ in the context of stable section $\delta$; since stable sections can be nested, the notation generalizes to sequences of stable section identifiers with sequence order reflecting nesting relationships. Our semantics is defined up to congruence of threads ($P | P' \equiv P \parallel P'$). We write $P \parallel \{e\}$ to denote the set of threads that do not include a thread with identifier $\tau$, and $P \parallel \{\tau|e\}$ to denote the set of threads that contain a thread executing expression $e$ with identifier $\tau$.

Expressions are variables, locations that represent channels, lambda-abstractions, function applications, thread creations, communication actions to send and receive messages on channels, or operations define a stable section, and to stabilize global state to a consistent checkpoint. We do not consider references in this core language as they can be modelled in terms of operations on channels.

Program evaluation is specified by a global reduction relation, $P, \Delta \rightarrow_{\text{reg}} P', \Delta'$, that maps a program state to a new program state. A program state consists of a collection of evaluating threads ($P$) and a stable map ($\Delta$) that defines a finite function associating stable section identifiers to states. We tag each evaluation step with an action that defines the effects induced by evaluating the expression. We write $\rightarrow^{\ast}$ to denote the reflexive transitive closure of this relation. Local reductions within a thread are specified by an auxiliary relation, $e \rightarrow e'$ that evaluates expression $e$ within some thread to a new expression $e'$. The actions of interest are those that involve communication events, or manipulate stable sections.

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 $E_{\delta}^{\ast}\{\delta\}$ to denote an expression $e$ available for execution by thread $\tau \in P$ in a program state; the sequence $\delta$ indicates that the ordered sequence of nested stable sections within which the expression evaluates.

The local evaluation rules are standard: holes in evaluation contexts can be replaced by the value of the expression substituted for the hole, function application substitutes the value of the actual parameter for the formal in the function body, channel creation results in the creation of a new location that acts as a receptacle for message transmission and receipt, and a function supplied as an argument to a stable expression yields a stable function.

There are five 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. 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.

The 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
fun sendFile() =
    let fun loop strm =
        if Abort.aborted abort
        then CML.send(consumer, XferAbort)
        else let val chunk =
                Bin10.inputN(strm, fileChunk)
                in if Word8Vector.length chunk = 0
                then CML.send(consumer, XferDone)
                else (CML.send(consumer, XferBytes chunk); loop strm)
        end
        in case Bin10Reader.openIt abort name of NONE => ()
   /
    SOME h => (loop (Bin10Reader.get h);
               Bin10Reader.closeIt h)
    end
end

Figure 2. File Reading Code From the Swerve Web Server Augmented with Stabilizers.

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. Note that, while easily defined, the semantics is conservative: there may be checkpoints that involve less unrolling that the semantics does not identify. We discuss how to calculate optimal checkpoints in Section ??.

The soundness of the semantics is defined by an erasure property on stabilize actions. Consider the sequence of actions α that comprise a possible execution of a program. Suppose that there is a stabilize operation that occurs in α. 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

$$E^\alpha_P[y][s], \Delta \Rightarrow^* P', \Delta' \Rightarrow^* P'' \parallel [z[v]], \Delta_f$$

Then, there exists an equivalent evaluation

$$E^{\alpha'.\beta}_P[y][s], \Delta \Rightarrow^* P'' \parallel [z[v]], \Delta_f$$

where $\alpha' \leq \alpha$

The proof of this theorem which defines a bisimulation on evaluation sequences is given in a companion technical report [2].

5. Incremental Construction

Although correct, our semantics is overly conservative because a global checkpoint state is computed upon entry to every stable section. 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 unraveled. 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. 4 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 graph. A graph consists of a set of nodes representing interesting program points and hold thread state at a point, edges that connect nodes that have shared dependencies, and maps to associate each thread with its current node in the graph, and its set of active stable sections. Nodes are indexed by ordered node identifiers.

Informally, the actions of each thread in the graph is represented by a chain of nodes that define temporal ordering on thread-local actions. Back-edges are established to nodes representing stable sections; these nodes define possible checkpoints. Sources of backedges are communication actions that occur within a stable section, or entry to 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 (transitionally) 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 $P, G \leadsto P', G'$ evaluates a process P executing action α with respect to a communication graph G to yield a new process $P'$ and new graph $G'$. The auxiliary relation
SYNTAX:
\[ P ::= P|P \mid t[e] \mid e \mid \text{mkCh()} \mid \text{send}(e,e) \mid \text{recv}(e) \mid \text{spawn}(e) \mid \text{stabile}(e) \mid \text{stabilize} \]

EVALUATION CONTEXTS:
\[ E ::= e \mid E(e) \mid v(E) \mid \text{send}(E,e) \mid \text{send}(1,E) \mid \text{recv}(E) \mid \text{stable}(E) \mid \text{stabile}(E) \]

\[ E^e_{\delta,P}[e] ::= P||E[E[e]]_{\delta} \]

\[ e \rightarrow e' \]

\[ E^e_{\delta,P}[e], \Delta \xrightarrow{\alpha} E^e_{\delta,P}[e'], \Delta \]

GLOBAL EVALUATION RULES:
\[ \forall \delta \in \text{Dom}(\Delta), \; \delta' \geq \delta \]
\[ \Delta' = \Delta[\delta' \mapsto (E^e_{\delta,P}[\lambda x.e(v)], \Delta)] \]
\[ \Lambda = \Delta'[\delta_{\text{min}}], \; \delta_{\text{min}} \leq \delta \; \forall \delta \in \text{Dom}(\Delta') \]

\[ E^e_{\delta,P}[\text{stable}(v)], \Delta \xrightarrow{\alpha} E^e_{\delta,P}[\text{stable}(v)], \Delta - \delta \]

\[ \Delta(\delta) = (P', \Delta') \]

\[ E^e_{\delta,P}[\text{stabilize}], \Delta \xrightarrow{\alpha} P', \Delta' \]

PROGRAM STATES:
\[ P \in \text{Process} \]
\[ t \in \text{Tid} \]
\[ x \in \text{Var} \]
\[ l \in \text{Channel} \]
\[ \delta \in \text{StableId} \]
\[ v \in \text{Val} \]
\[ \alpha, \beta \in \text{Op} = \{\text{sp,ss,comm,ss,ss,comm} \} \]
\[ \Lambda \in \text{StableState} = \text{Process} \times \text{StableMap} \]
\[ \Delta \in \text{StableMap} = \text{StableId} \xrightarrow{\alpha} \text{StableState} \]

LOCAL EVALUATION RULES:
\[ \lambda x.e(v) \rightarrow e[v/x] \]
\[ \text{mkCh()} \rightarrow 1, \; 1 \text{ fresh} \]
\[ \text{stable}(\lambda x.e) \rightarrow \lambda x.e \]

Figure 3. A core call-by-value language for stabilizers.

\[ t[e], G \Downarrow G' \text{ models intra-thread actions within the graph. It creates a new node to capture thread-local state, and sets the current node marker for the thread to this node. In addition, if the action occurs within a stable section, a back-edge is established from that node to this section. This backedge is used to identify a potential rollback point.} \]

New nodes are created by the procedure \textit{addNode} that creates a new node whose node identifier is greater than any existing node in the graph.

We define a correspondence theorem between the two semantics that formalizes the intuition that incremental checkpoint construction results in less rollback than a global point-in-time checkpoint semantics:

Theorem[Correspondence] Let
\[ E^e_{\delta,P}[e], \Delta_0 \xrightarrow{\alpha ST} P', \Delta' \xrightarrow{\alpha ST} P'' || e[v], \Delta_f \]

Then whenever
\[ E^e_{\delta,P}[e], G_0 \xrightarrow{\alpha ST} P', G' \xrightarrow{\alpha ST} G_f, E^e_{\delta,P}[v], G_f, \]

\[ |\beta'| \leq |\beta| \]

The proof of this theorem is provided in [?].

5.1 Example

To illustrate the semantics, consider the sequence of actions shown in Fig. ??.

When thread \( t_1 \) spawns a new thread \( t_2 \), a new node \( n_2 \) is created to represent \( t_2 \)'s actions, and an edge between the current node referenced by \( t_1 \) in the graph (\( n_1 \)) to \( n_2 \) is established (see

Figure 5. Incremental checkpoint construction.
backward edges are established from the current node to the closest
(a)). When an edge between
roll-back to it local checkpoint represented by
for both threads are created, and a bi-directional edge between them
performed (see (e)). Since both threads are now in stable sections,
and
is not executing within any stable section currently, and thus no
ment stabilizers, we needed to only modify the core CML library.
The number of saved continuations is thus directly proportional to
the size of the communication graph. Since our graph building al-
gorithm maintains checkpoints at the entry to a stable section or
prior to a communication event, the overheads for saving continu­
ations is a function of the number of stable sections and communi­
cation; thus, the number of dynamically spawned threads affects the graph
2 Notice that we consider a thread spawn as a one way implicit communi­
cation; thus, the number of dynamically spawned threads affects the graph
6.1 Handling References
Until now, we have elided the presentation of how to efficiently track shared memory access. Naively tracking each read and write separately would be inefficient and would limit Stabilizers to a functional setting. There exist two problems for Stabilizers: avoiding logging un-necessary writes, and avoiding tracking duplicate dependencies based on reads. We first present an algorithm to solve the first problem, and then show a modification to the graph building algorithm as a solution to the second.
Notice that for a given stable section, it is enough to monitor the first write for a given memory location since each stable section is unrolled as a single unit. For a given write to a memory location, we need only monitor the first read for a given thread (if this write is unrolled, the reading thread will always be unrolled atleast before the first read). For each stable section, 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. When the program enters a stable section, we create an empty version list for this section. When a write is encounter within a stable section for a

\[
E^n\cdot P[\text{spawn}()] \xrightarrow{\text{SP}} P'' \oplus (P \ominus P')
\]

\[
E^n\cdot P[\text{stabilize}], G \xrightarrow{\text{SS}} P'^*\ominus P', G'\\n\]

\[
\tau = \text{REACH}(n(t)) \quad P' = \{ t[e] \mid \langle i,t[s] \rangle \in \tau \text{ s.t. } i \leq j \forall (j,t[e']) \in \tau \}
\]

\[
\tau = \text{REACH}(n(t)) \quad P' = \{ t[e] \mid \langle i,t[s] \rangle \in \tau \text{ s.t. } i \leq j \forall (j,t[e']) \in \tau \}
\]

\[
\tau = \text{REACH}(n(t)) \quad P' = \{ t[e] \mid \langle i,t[s] \rangle \in \tau \text{ s.t. } i \leq j \forall (j,t[e']) \in \tau \}
\]

\[
\tau = \text{REACH}(n(t)) \quad P' = \{ t[e] \mid \langle i,t[s] \rangle \in \tau \text{ s.t. } i \leq j \forall (j,t[e']) \in \tau \}
\]

\[
\tau = \text{REACH}(n(t)) \quad P' = \{ t[e] \mid \langle i,t[s] \rangle \in \tau \text{ s.t. } i \leq j \forall (j,t[e']) \in \tau \}
\]

\[
\tau = \text{REACH}(n(t)) \quad P' = \{ t[e] \mid \langle i,t[s] \rangle \in \tau \text{ s.t. } i \leq j \forall (j,t[e']) \in \tau \}
\]
given reference, we first check if the reference is in our version list.
If there is no entry for the reference in our version list, this must be
the first write to this reference within this section. We therefore
log the current value of the reference and add it to our version list
before allowing the write to occur. If, however, the reference has an
entry within the version list for this stable section, we have already
logged the reference and nothing further needs to be recorded; the
write will proceed as normal.

Until a nested stable section exits, it is possible for a call to
stabilize to unroll to the start of this section. Therefore, any nested
stable section requires its own version list until it exits. Since the
first write for a given memory location may occur within a nested
stable section, the data stored in this stable sections version list
must be propagated to the outer section. This requires a merging
of the two version lists when any inner section exits. However, if
the outer stable section has monitored a particular memory location
and the inner section has also monitored the same memory location
we only need to store the outer section’s version. When we merge
the two version lists, we add versions for memory locations not
already present in the outer stable sections version list. Consider
the following two code snippets in Fig 6 which illustrate how the
algorithm works. In program A we must propagate the information
stored about y from the inner section’s version list to the outer
section’s version list. However, in program B, the inner section
creates a new version of x based on the outer section’s write to
x. When we merge the two version lists, we only propagate the
version of y stored by the inner section.

Efficiently monitoring read dependencies requires us to adopt
a different methodology. For a program to be correctly synchron­
zied and race free, all reads and writes from a location x must be
protected by lock 1, or more generally a set of locks L. Therefore,
it is enough to monitor lock acquires/releases to adequately infer
shared memory dependencies in correctly synchronized, race free
programs. Each successive code segment protected by a lock ac­
quire is dependent on the previous code segments protected by
that lock. To model this behaviour we augment our graph data struc­
ture with the domain L, a mapping between a lock l and a node n.
We define the lock acquire action as $AQ{l}^{x}$ and add the follow­
ning rule to our graph building equations. When a thread t acquires
a lock l, we must add a backwards edge between the node n and
the current locked stable section $L(l)$. After this, we set the current
node of the acquiring thread to the last locked node for l.

$$t, G \downarrow S (n_s, e_s, \eta, \sigma, L)$$

$$AQ_l^{x} S, G \downarrow S, (n_s, e_s', \eta, \sigma, L[l \rightarrow n])$$

6.2 Calculating the Checkpoint

The scope of a rollback for a given stabilize call is defined as all
nodes reachable from the current node of the invoking thread. This
set of nodes can be calculated by performing a depth first search
(DFS) starting from the current node of the thread which called
stabilize. Notice, that a rollback will only ever unroll multiple
threads if they are connected through the use of locks or commu­
nication events. Nested stable sections will revert to the outer most
stable section only if there exists a series of communication events
which joins the two stable sections (see example 1).

6.3 Pruning the Graph

The size of the communication graph grows with the number of
communication events, thread creation, 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 through a DFS from the outermost stable
section for each thread. These sections of the graph can be safely
trimmed and are no longer needed. To calculate the part of the graph
which is no longer reachable at a given execution point, we perform
a reachability test on all threads. The set of nodes that comprise the
result precisely characterize all nodes which could potentially be
rolled back in the future, all other nodes and edges can safely be
trimmed from the graph.

6.4 Implementation

Our implementation is incorporated within MLton [22], a whole
program optimizing compiler for Standard ML. The only changes
to the underlying infrastructure were light weight write barriers and
hooks in the CML library. We capture thread state by simply saving
the thread’s continuation at points defined by the graph construction
algorithm. Continuations are stored within nodes of the graph. The
number of saved continuations is thus directly proportional to the
size of the communication graph.

Because our implementation is an extension of the core CML
library, it supports first-class events as well as channel-based com­
munication. The handling of events is no different than our treat­
ment of messages. If a thread is blocked on an event with an associ­
ated channel, we insert an edge from that thread’s current node
to the channel. Similarly, the graph building algorithm does not
change for full communication based on events. Our implementa­
tion supports the basic send and recv events, from which more
complex events can be generated via combinators. By instrumenting
base events, our implementation is able to handle arbitrary first
class events transparently. Thus, we are also able to 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 stabi­
lize action. Notice that each thread can be blocked on at most one
communication event and one channel. Therefore, there can be at
most one value to clear from a channel per thread when stabiliz­
ing a program. Since CML stores both the blocking thread and the
value on the channel, it is straightforward to determine the values
that must be cleared from a channel.

7. Performance Results

To measure the cost of stabilizers with respect to various concurrent
programming paradigms, we present two synthetic benchmarks to
quantify pure memory and time overheads and a number of real
world benchmarks to illustrate average overheads in real programs.
To measure the costs of our stabilize abstraction, our benchmarks
were executed in two different ways: CML - the benchmark run
under core CML, Graph - the benchmark run with on-the-fly
checkpointing, but in which no stabilize actions are performed
and graph pruning is disabled. We compare CML to Graph to illus­
trate the costs of maintaining the information needed to compute
a safe checkpoint state. This comparison captures the most com­
mon case, since we expect stabilize calls to occur infrequently
in real programs. By disabling graph pruning, our measurements
illustrate worst case overheads or an upper bound on graph size.
Our benchmarks results are presented as overheads normalized to
CML. The benchmarks were run on an Intel P4 2.4 GHz machine
with one GByte of memory running Gentoo Linux.

Our first synthetic benchmark, Asynchronous Communication,
measures the costs of building and maintaining our graph structure
as well as the cost of stabilize actions in the presence of asyn­
crchronous communication. The 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 communications. The second synthetic benchmark,
Communication Pipeline, measures similar effects as the first, but
captures the behavior of computations that generate threads which
communicate in a synchronous pipeline fashion. The benchmark
spawns a series of threads, each of which defines a channel used to communicate with its predecessor. Each thread blocks until it receives a value on its input channel and then sends an acknowledgment to the thread spawned before it. The first and last threads in the chain are connected to form a circular pipeline.

Our real-world benchmarks include a pair of eXene benchmarks (Triangles and Nbody), Pretty: a pretty printing library written on top of eXene, and Swerve: a web server. We consider eXene, the underlying benchmark and utilize programs built with eXene as a test bed. The eXene toolkit, by Reppy and Gansner, is an X Windows toolkit written in CML. The eXene toolkit, roughly 30,000 lines of code (15,801 for the core of eXene and 14,650 in widget code), implements the functionality of xlib; all communication with the X server is written in SML. Events from the X server and control messages between widgets are distributed in streams (coded as CML event values) through the window hierarchy. Each window is encoded as a number of CML threads, each of which handles various events. Drawing is done by calling imperative drawing procedures, but high-level events are communicated through messages on channels. The toolkit eXene manages the X calls through a series of servers, dynamically spawned for each connection and screen.

Swerve is a webserver written in CML able to process numerous connections and consisting of 9,915 lines of code. The server architecture is de-coupled into five modules: Listener, Htp, Store, File Handler, and Logging, each of which communicates with the others through channels. The listener listens on the main port for requests, spawning a new concurrent object that implements the Htp protocol. The Htp module parses the request and uses the URL and other information to query the store to send the response back to the client over the socket. The Store module consists of a tree of nodes representing different URL's. Each node contains two threads for dispatching requests to avoid any deadlocks. Once a request is matched it is sent to a file or CGI handler. The file handler reads any request files from the file system. Communication is done exclusively over channels. The connection protocol is made up of messages passed between the http object, store, and handle.

The runtime result for checkpointing the asynchronous communication benchmark is presented in Fig. 7(a), and the total allocation overhead is presented in Fig. 7(b). As we would expect, the runtime cost to simply maintain the graph grows linearly with the number of asynchronous communications. Memory overheads also grow linearly. There is a significant initial memory overhead due to pre-allocating hash tables used to store the current node of each thread and to maintain an association between channels and their nodes. The cost to stabilize also grows linearly with the number of asynchronous communications that need to be unrolled in one stabilize call. For a rollback of two thousand asynchronous communication events, reverting to a stable state takes roughly 74 milliseconds; 44 to run the benchmark and construct the graph, and 30 to calculate affected threads, prune the graph, and restore state. This is not surprising given that the application only performs actions that result in modifications to the communication graph. The runtime and memory overheads for the pipeline benchmark are shown in Fig. 7(c) and Fig. 7(d); these overheads are comparable to those measured for the earlier benchmark. In fact, the number of threads and communication events is exactly the same since an asynchronous communication requires the spawning of a new thread, which in turn performs a communication event. The time costs associated with stabilizing all threads were similar to those seen for asynchronous communication.

In real applications, stabilizers exhibit a runtime slow down of approximately 6% over a CML program in which monitoring is not performed (see Table 1). The cost of using stabilizers is only dependent on the number of inter-thread actions and dependencies performed (note: the number of references tracked is also dependent on stabilizers and lock dependencies). As predicted, the overheads for tracking program dependencies is easily amortized across program execution. Memory overheads to maintain the communication graph are larger. 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. Our mechanism would benefit from a lightweight low-overhead representation of continuations [33, 3]. Because we were interested in understanding the worst-case bounds of our approach, graph pruning was not employed to reduce graph size. Despite this constraint, our benchmarks show that even in programs which utilize over 10k threads with non trivial communication patterns only requires only a 4was a much larger percent of the total memory utilized by the program, but still took up less than 1mb of memory. It is important to note that we do not expect stabilizers to have to monitor and store every event, graph pruning can significantly reduce the size of the graph during execution.

8. Previous Work

Being able to checkpoint and rollback parts or the entirety of an execution has been the focus of notable research in the database [11] as well as parallel and distributed computing communities [15, 24, 26]. Reverting to previous state provides a measure of fault tolerance for long-running applications [34]. Classically, checkpoints have been used to provide fault tolerance for long-running, critical executions, for example in scientific computing [2] but have been typically regarded as heavyweight entities to construct and maintain.
explored abstractions and mechanisms closely related to stabilizers and their implementation for maintaining consistent state in a concurrent environment [18]. The work presented here is distinguished from these efforts in its focus on defining a coordinated safe checkpointing scheme for concurrent message-passing functional programs.

In addition to stabilizers, functional language implementations have utilized continuations for similar tasks. For example, Tolmach and Appel [35] describe a debugging mechanism for SML/NJ that utilized captured continuations to checkpoint the target program at a specific point. This work was later extended [36] to support multithreading, and was used to log non-deterministic thread events to provide replay abilities.

Another possibility for fault recovery is micro-reboot [8], a fine-grain 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. Such a recovery mechanism can greatly reduce downtime for applications whose availability is critical. Unlike checkpointing schemes, which attempt to restore a program to a consistent state within the running application, micro-reboot quickly restarts an application component. However, micro-reboot suffers the same problems that plague most transparent fault recovery mechanisms; namely, such constructs are ignorant of program semantics, resulting in their use only when an error becomes a system fault.

The ability to revert to a prior point within a concurrent execution is essential to transaction systems based on optimistic (or speculative) concurrency [1, 19, 25]; outside of their role for database concurrency control, such approaches can improve parallel program performance by profitably exploiting speculative execution [31, 37]. Optimistic concurrency allows multiple threads to access a guarded object concurrently. As long as these data accesses are disjoint, no error occurs. If a thread commits is changes to a shared object, which was access by another thread, the second thread must be reverted to a state prior to its data access to ensure a serializable schedule. Harris proposes a transactional memory system [21] for Haskell that introduces a retry primitive to allow a transactional execution to safely abort and be reexecuted if desired resources are unavailable. When the retry primitive is invoked, the transaction is unrolled and re-executed. 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.

9. Future Work and Conclusion

Although stabilizers are a useful checkpointing abstraction, they lack the ability to provide restarted computations with alternative inputs. One possible extension to the stabilizer abstraction is to integrate them with a compensation model [7]. Using exception-style syntax, we envision the programmer writing e1 \begin{math} \text{stable} \end{math} e2 to denote the execution of e2 within a stable section. If e1 is unrolled through a \begin{math} \text{stabilize} \end{math} call, its compensation e2 would be executed. Compensations are one way to provide the programmer an ability to execute alternate code or to modify the heap. Currently, we provide the programmer the ability to write \begin{math} \text{first-class stable} \end{math} functions, but a \begin{math} \text{stabilize} \end{math} call is static. We could envision a system where there exists an explicit pairing between \begin{math} \text{stable} \end{math} and \begin{math} \text{stabilize} \end{math}, where a call to \begin{math} \text{stabilize} \end{math} stabilizes a specific stable section. The \begin{math} \text{stabilize} \end{math} function itself would be \begin{math} \text{first-class} \end{math}, allowing threads to rollback stable sections within other threads.

Stabilizers are a useful on-the-fly checkpointing abstraction for functional languages. Unlike other transparent checkpointing schemes, stabilizers are not only able to identify the smallest subset of threads which must be unrolled, but also provide useful safety guarantees. Using our abstraction, programmers do not need to reason about inserting global static checkpoints. Instead stabilizers provide the primitive \begin{math} \text{stable} \end{math} to group events local to threads and to specify interesting points within a thread. Our runtime system automatically tracks the interactions of communication events and stable sections, and calculates the closest checkpoint that still maintains data-consistency across all threads. Our results indicate that stabilizers can be implemented with modest overhead by leveraging control abstractions like continuations already available in functional languages, and thus serve as an effective checkpointing abstraction for these languages.

References


Table 1. Benchmark Overheads

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC (eXene)</th>
<th>Spawns Chans Comms</th>
<th>Writes Reads</th>
<th>Raw Mem (MB)</th>
<th>Norm. Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangles</td>
<td>500</td>
<td>16301</td>
<td>205 79 187</td>
<td>88 88</td>
<td>0.185</td>
</tr>
<tr>
<td>N-Body</td>
<td>356</td>
<td>16137</td>
<td>240 99 224</td>
<td>224 273</td>
<td>0.282</td>
</tr>
<tr>
<td>Pretty</td>
<td>2400</td>
<td>18201</td>
<td>801 340 950</td>
<td>602 840</td>
<td>0.737</td>
</tr>
<tr>
<td>Swerve</td>
<td>9915</td>
<td>-</td>
<td>10532 231 902</td>
<td>9339 80293</td>
<td>5.421</td>
</tr>
</tbody>
</table>

Figure 7. Synthetic Benchmark overheads.