Stabilizers: A Checkpointing Abstraction for Concurrent (Functional) Programs

Suresh Jagannathan

joint work with Lukaz Ziarek and Philip Schatz

PURDUE UNIVERSITY

Observations

- Classical approaches to coordinating activities of multiple threads:
  - Impose a heavy burden on programmer to balance safety and performance
  - Pose well-known issues with deadlocks, data races, priority inversion, interaction with external actions, etc.
  - Scalability impacted by the use of mutual-exclusion
- Advent of multi-core processors exacerbate these concerns
- Opportunity for principled language design
  - Abstractions that
    - simplify concurrent program structure
    - without sacrificing efficiency or scalability
- Examples:
  - Software transactions
  - Safe software-based speculation

Issues

- Expressivity:
  - Software transactions
    - Modularity concerns raise important issues:
      - Multi-threaded transactions
      - Open nesting semantics
- Robustness:
  - Errors and exceptional conditions may arise in long-lived computations
- These are closely-related issues

Robustness

- How can an exception handler ensure that global state is consistent after it executes?
    - ★ Consider thread communication within a handler scope
    - ★ How does a handler revert thread state to one which is consistent with views of other threads?
    - ★ Failure to ensure consistency can lead to deadlock, or erroneous results
- Difficult for applications to enforce consistency statically because of non-determinism and implicit, dynamically-defined thread dependencies
    - ★ If a thread broadcasts some data, how can an application efficiently determine the set of threads that read this data?

Checkpoints

- Checkpoints provide a means to globally revert a computation to an earlier state.
- Transparent approaches: compiler or operating system
  - May not be efficient or semantically meaningful
- Non-transparent: Library or application-directed
  - Precise but non-trivial to construct
- Our idea:
  - Applications define thread-local program points where checkpoint is feasible.
    - When a thread attempts to restore execution to a previous checkpoint, control reverts to one of these points for each thread.
      - The exact checkpoint chosen is calculated dynamically based on lightweight monitoring of thread communication events:
        - message-passing through channels
        - shared memory

Stabilizers

- Signatures
  - ★ stable: `('a -> 'b) -> ('a -> 'b)`
  - ★ `stabilize: unit -> 'a`
- Declare monitored section of code
  - ★ Track inter-thread actions including communication and shared memory access
  - ★ Defines a thread-local checkpoint
- Maintain a global dependency structure
  - ★ Construct a global checkpoint from a collection of thread-local ones based on (transitive) thread dependencies
- Serve as building blocks for
  - ★ multi-threaded open-nested transactions
  - ★ safe software-based speculative execution

Example

```
let val c = channel()
    val c' = channel()
    fun g y = ... recv(c) ... recv(c')
              ...
              raise Timeout
              ...
              in handle Timeout => ...
    fun f x = let val _ = spawn(g(...))
                  val _ = send(c,x)
                  ...
               in if ...
                  then raise Timeout
                  else ...
               end handle Timeout => ...
in spawn(f(arg))
end
```

What happens if f raises a timeout exception?

Must re-execute it, erasing effects from the earlier evaluation

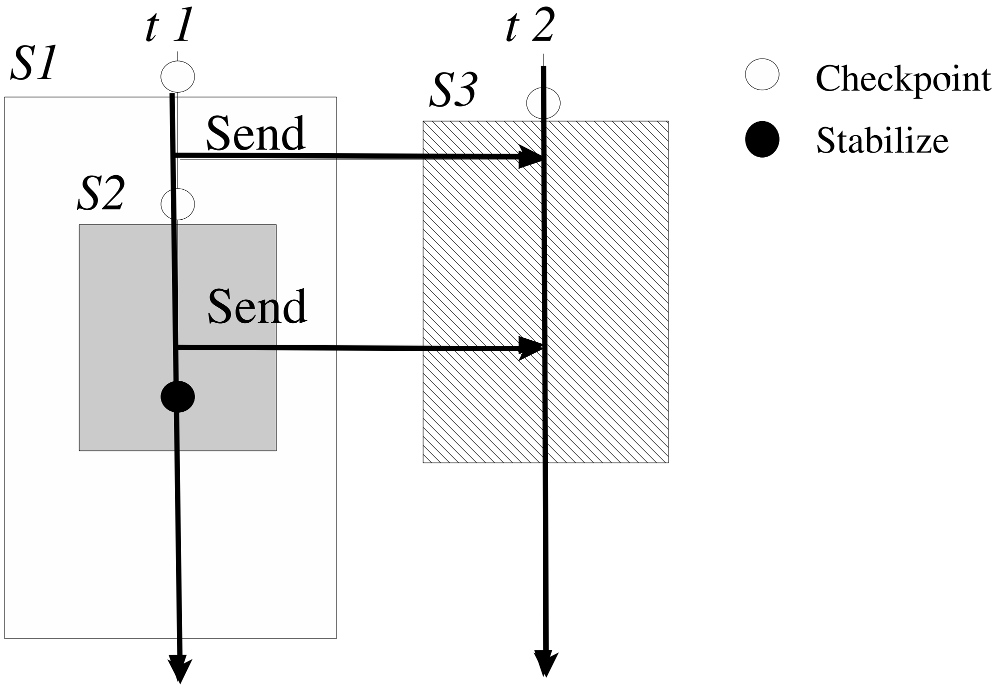
Determining the set of events that must be restored depends on dynamic scheduler events.

Example

```
let val c = channel()
    val c’ = channel()
    fun g y = ... recv(c) ... recv(c’)
              ...
              raise Timeout
              ...
              in handle Timeout => ...
    fun f x = stable fn () =>
                      let val _ = spawn(g(...))
                          val _ = send(c,x)
                            ...
                      in if ...
                         then raise Timeout
                         else ...
                      end handle Timeout => stabilize()  ()
in spawn(f(arg))
end
```

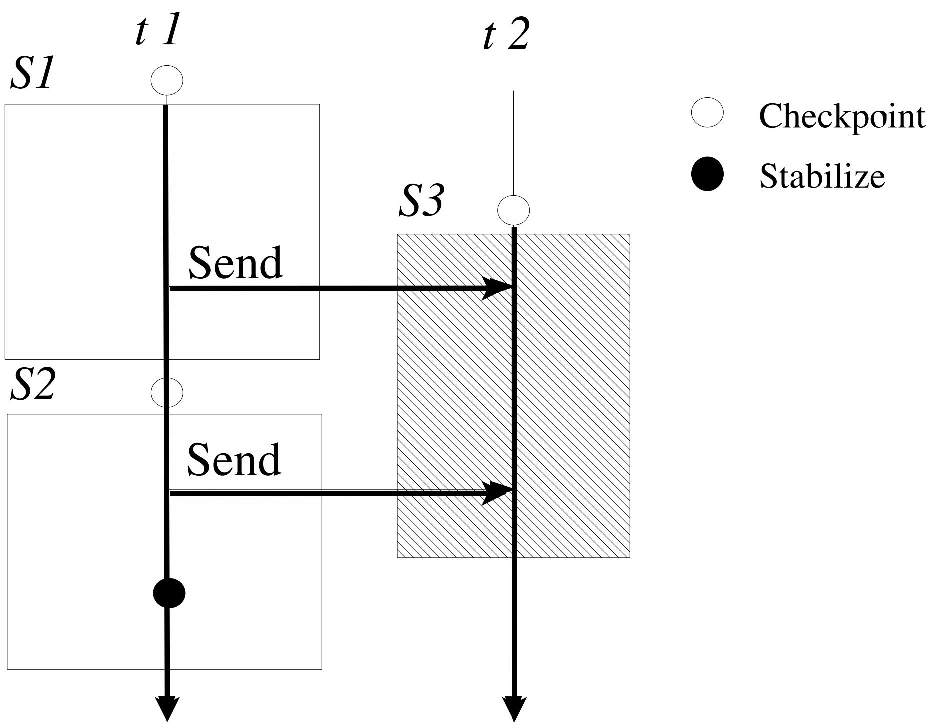
A timeout exception reverts the computation to a state in which the spawn of g, and its receipt on channel c have been discarded.

Example



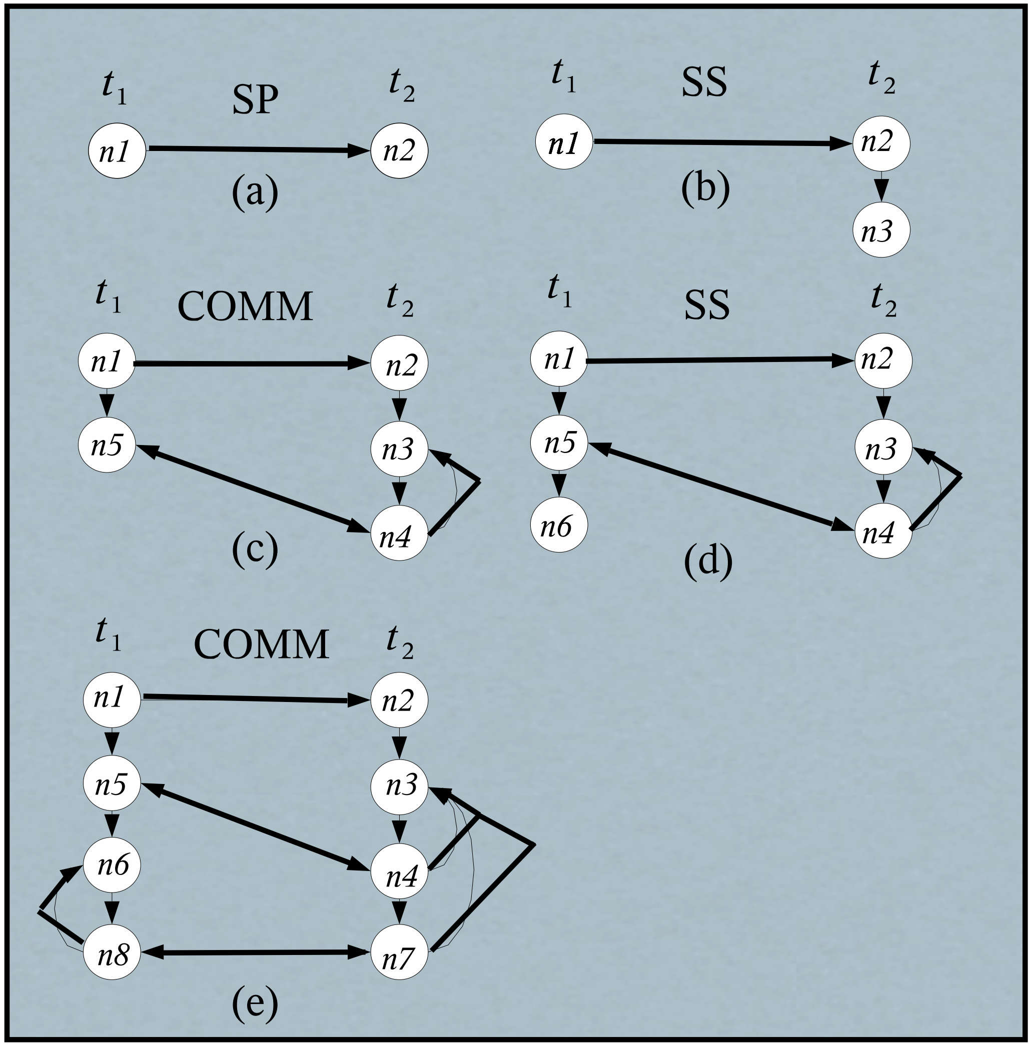
Sections chosen for rollback depends upon communication actions performed

Example



Sections chosen for rollback depends upon communication actions performed

Dependency Graph



Nodes record context information (continuations) and edges reflect dependencies

Establish dependencies among threads and their actions:

(a) thread spawn
(b) stable section entry
(c) inter-thread communication event
(d) stable section entry
(e) further communication

Characteristics

- Properties:
  - ★Safety: A stabilize action never yields an infeasible state.
  - ★Correspondence: Stabilization is never worse than global checkpointing
- A rich abort semantics:
  - ★ More expressive than classical transactional undo semantics
  - ★Set of participating threads is determined by (transitive) cross-thread dataflow dependencies that occur within monitored sections.
  - ★ Basis for an open nested transactions
  - ★ Fine-grained speculative computation
  - ★ Avoids the need for non-local exception handling logic in every potentially affected thread

Overheads

- Implemented in MLton
  - ★ Insertion of write barriers and eliminating read barriers
  - ★ Compensations
  - ★ hooks in the CML library to update the dependency graph
- Overheads to maintain checkpoints small, roughly 6%
  - ★ eXene: a windowing toolkit
  - ★ Swerve: a web server

	Threads	Channels	Events	Shared Writes	Shared Reads	Graph Size	Runtime Overheads (%)
Triangle	205	79	187	88	88	.19	.59
N-Body	240	99	224	224	273	.29	.81
Pretty	801	340	950	602	840	.74	6.23
Swerve	10532	231	902	9339	80293	5.43	6.60

Conclusions

- Stabilizers are an on-the-fly checkpointing abstraction.
- Improve robustness and expressivity of concurrency and synchronization abstractions
  - ★ Valuable for long-lived applications
  - ★ Useful to help coordinate activities of dynamically-related threads
- Provides useful safety guarantees
- Can be implemented with relatively small overhead