Exploring Failure Transparency and the Limits of Generic Recovery

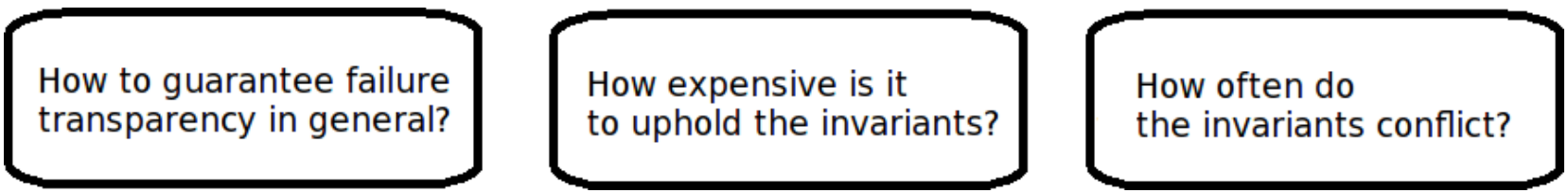
David E. Lowell, Subhachandra Chandra, Peter M. Chen

Presented by Miroslav Cupak

10/29/2012

Introduction

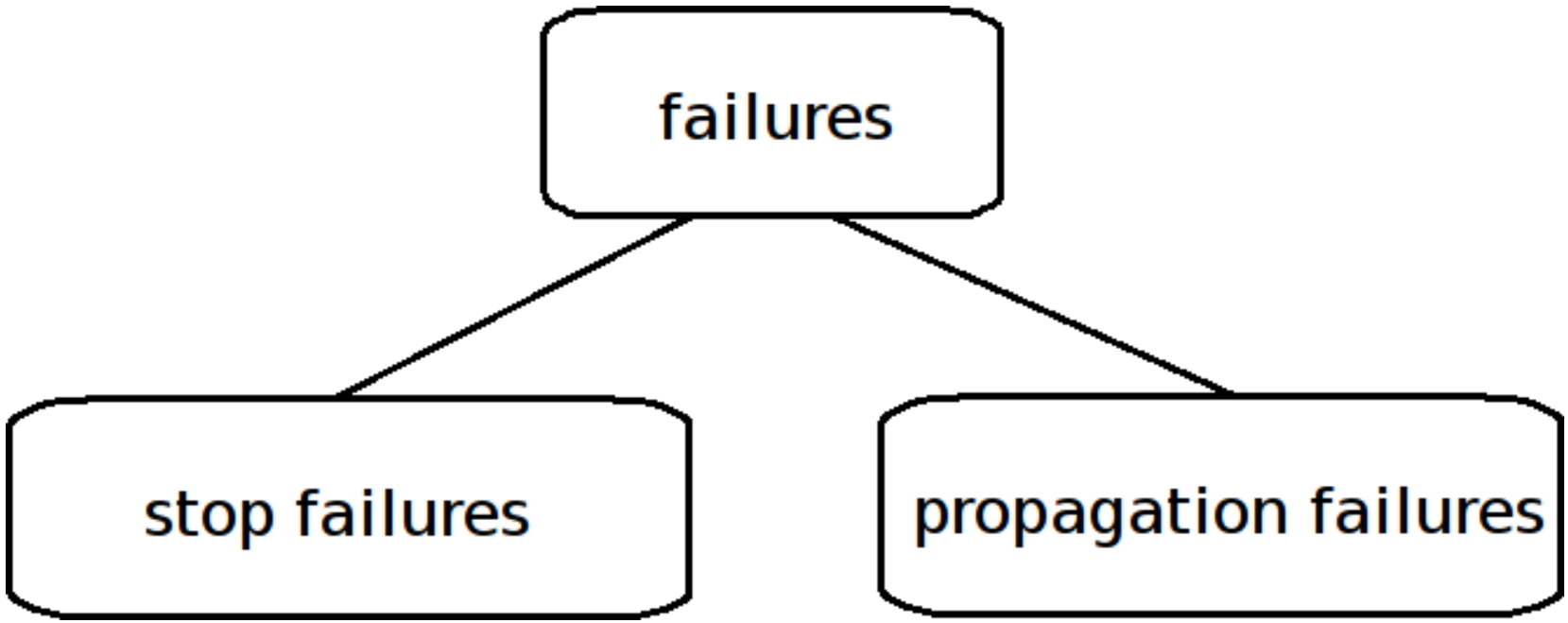
- failure recovery as an illusion of failure-free operation
- goal of the paper:
  - 2 invariants for failure transparency - save/lose work
  - evaluation of real-life applications



Guaranteeing Failure Transparency

- requirements:
  - automatized
  - fast
  - generic
- tools:
  - commit events
  - rollback of a process
  - reexecution
- problems:
  - undoable and redoable operations
  - some things hard to undo
  - some things hard to redo

Guaranteeing Failure Transparency: Failures



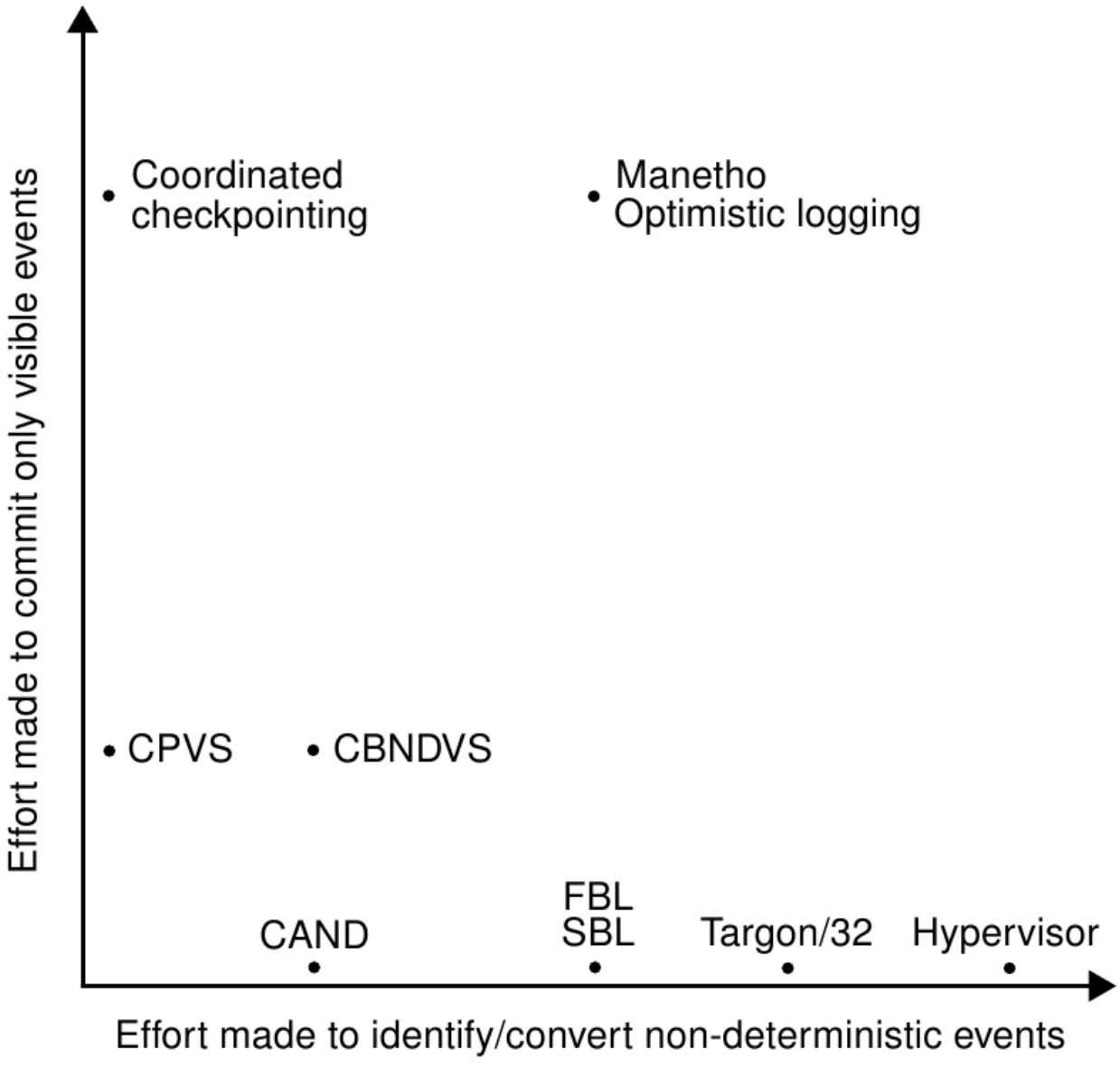
Guaranteeing Failure Transparency: Stop Failures

- consistent recovery
  - Recovery is consistent if and only if there exists a complete, failure-free execution of the computation that would result in a sequence of visible events equivalent to the sequence of visible events actually output in the failed and recovered run.
  - ignore ordering and duplicates
  - non-deterministic events

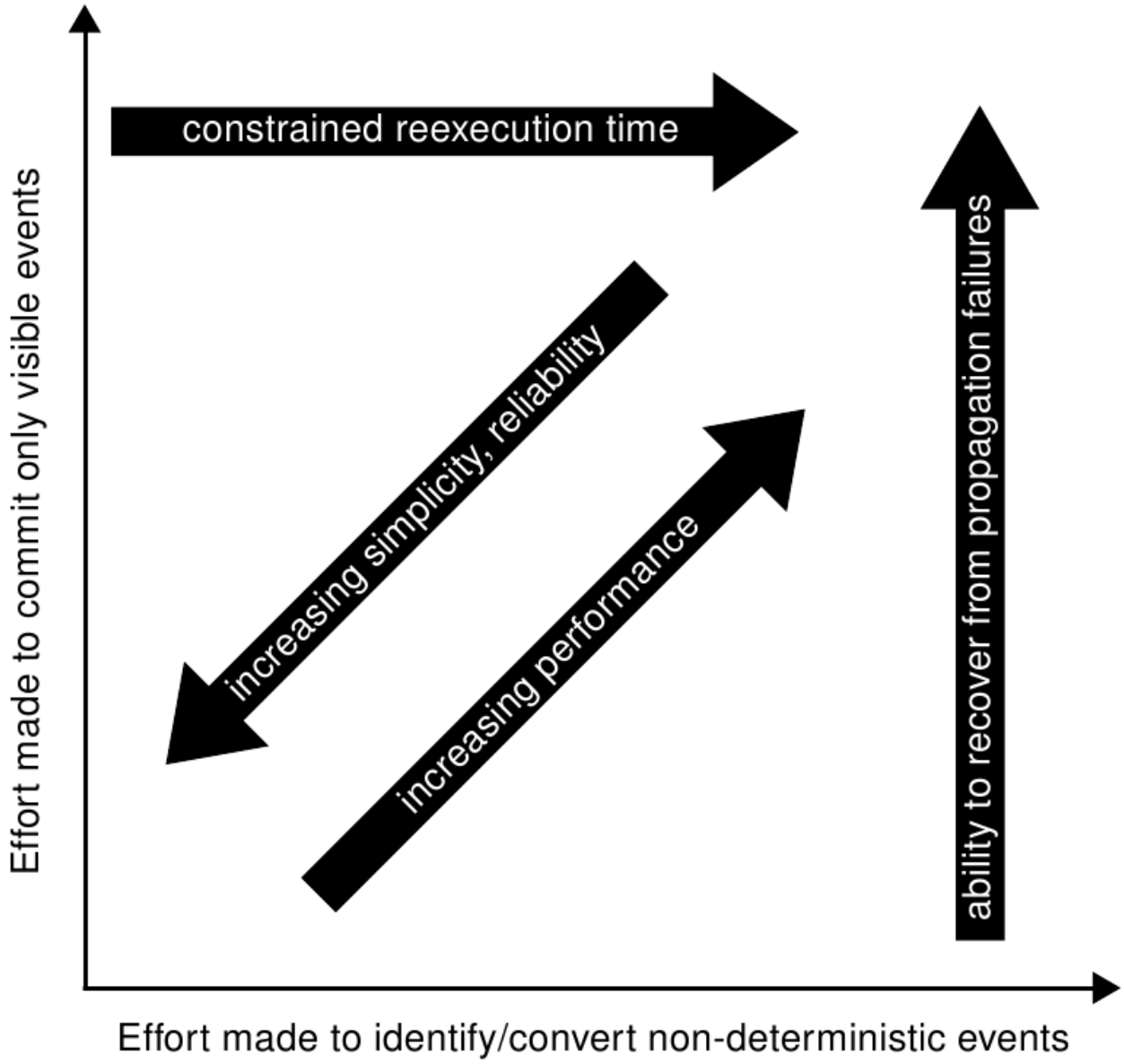
Guaranteeing Failure Transparency: Save Work

- save-work invariant
  - A computation is guaranteed consistent recovery from stop failures if and only if for each executed non-deterministic event e_p^i that causally precedes a visible or commit event e, process p executes a commit event e_p^j such that e_p^j happens-before (or atomic with) e, and $i <= j$.
  - save-work visible and save-work orphan invariant
- many protocols

Guaranteeing Failure Transparency: Protocol Space



Guaranteeing Failure Transparency: Protocol Space



Guaranteeing Failure Transparency: Propagation Failures

- non-determinism helpful
- fixed vs transient events

Guaranteeing Failure Transparency: Lose Work

- lose-work invariant
  - Application-generic recovery from propagation failures is guaranteed to be possible if and only if the application executes no commit event on a dangerous path.
- single/multi-process dangerous paths algorithms
- techniques for adding non-determinism

Evaluation: Performance Penalty

- 4 applications: nvi, magic, xpilot, TreadMarks
- failure transparency: Discount Checking modified to intercept non-deterministic system calls
- reliable memory: Rio File Cache
- transactions: Vista transaction library
- 7 protocols: CAND, CPVS, CBNDVS, CAND-LOG, CBNVS-LOG, CPV-2PC, CBNDV-2PC

Evaluation: Performance Penalty - Results

- 24 performance analyses (no. of checkpoints, increase in execution time)
- interesting observations:
  - commit frequency decreases and performance increases with increasing coordinates (except for xpilot)
  - at least 1 protocol performs well for each app
  - disk-based recovery with reasonably low overhead for interactive apps

Evaluation: Conflicts (Lose-work vs Save-work)

- sample: 50 crashes for each of 7 fault types for 2 applications (nvi, postgres)
- application faults (commit on dangerous path after fault activation)
  - lose-work violation: 35%
  - guess for conflicts based on non-deterministic bug distribution in other applications: 90%
- OS faults
  - failure to recover: 3% (postgres) - 15% (nvi)
  - propagation failures: 10% (postgres) - 41% (nvi)

Related work

- lot of work on transactions, fault tolerance and recovery from stop failures
- notable papers:
  - E. N. Elnozahy et al.: A Survey of Rollback-Recovery Protocols in Message-Passing Systems (thorough analysis of checkpointing & log-based recovery protocols, over 240 references)
  - S. Chandra: Evaluating the Recovery-Related Properties of Software Faults (PhD thesis explaining most of the failure-recovery terms used in the paper)
  - D. E. Lowell: Theory and Practice of Failure Transparency (consistent recovery, commit invariant, protocol space, discount checking)
  - D. E. Lowell, P. M. Chen: Discount Checking (fast light checkpointing system)

Conclusions & Contributions

- extending the concept of the stop failures to propagation failures
- invariant for surviving propagation failures
- evaluation of propagation failures for which consistent recovery is not possible
- failure transparency for stop failures possible, help from the application needed for propagation failures

Questions?

Discount Checking system used in evaluations is built on top of Vista transaction library. Why are transactions needed?

What are the transactions needed for?

- DC doesn't know anything about semantics of application operations and cannot use it for rollback
- transactions can be easily used to implement checkpointing
  - interval between checkpoints = body of a transaction
  - taking a checkpoint = transaction commiting
  - process state rollback to the last checkpoint = transaction aborting

Applications failed to recover in up to 15% cases using Discount Checking with the possibility to roll back to the last checkpoint. Why don't they use multiple checkpoints? Would it help?

Would using multiple checkpoints help? Why don't they use them?

- it would definitely help to decrease the recovery failure rate
- but it wouldn't solve all the problems (starting states)
- it's not used probably because of implementation difficulty (system based on transactions, this kind of action would require another encapsulation)
- performance overhead would increase

Failure transparency is great. What are the drawbacks?

What are the drawbacks of failure transparency?

- increased cost
- complicated debugging (masked failures)
- interference with other components
- lower priority of fault correction

How representative are the results of the evaluations?

How representative are the results of the evaluations?

- small sample of different types of applications with prevailing types of events
- even though only 2 applications tested for performance, significantly different results
- only 1 checkpointing system
- part of the statistics inferred from other programs
- 12 years ago - new HW and possibly different bottlenecks, new possibly more efficient protocols etc.

As a developer trying to make an application transparent to failures, would you use the same approach as the authors did for they evaluations (Discount Checking)? Why?

Would you use Discount Checking?

- good solution for many use cases
- amazingly flexible, very easy setup for existing applications compared to other systems we mentioned
- works out of the box, but extensible if needed
- low performance overhead