Optimistic Parallelization Support for Event Stream Processing Systems

Andrey Brito

TU Dresden

MDS'08, December 1st, 2008
Goal

- **Goal**
  - Investigate usage of speculation in ESP systems

- **Contributions so far**
  - Optimistic parallelization of stateful operators
  - Early processing of events in distributed ESP systems
  - Concept of conflict predictors to guide speculation
Motivation (1)

- Event stream applications
  - Directed acyclic graph of operators
  - Some operators don't keep state
    ⇒ trivially parallelizable
  - Some do keep state
    ⇒ not trivially parallelizable
  - Sometimes there are dependencies between the events
    ⇒ need to process events sequentially, waiting for the order to be restored
Application example

**Incoming stream of events (high rate, time-stamped)**

**Stateless** (e.g., parsing and transformation) ⇒ trivially parallelizable

**Stateful** (e.g., stream query operators)

**Stateful** (low overhead) ⇒ no need to parallelize

**Outgoing stream of events**
Application example
Motivation (2)

- Fault-tolerant distributed event stream applications
  - Some applications require precise recovery in case of failures
  - Some components use non-determinism
    → must log non-deterministic decisions to provide repeatability
  - Also applies for applications that require determinism for postmortem analysis
Application example

Events based on non-det. decision

Events are out!
Application example

Input Adapter → Filter → Processor 1 → Aggregator → Output Adapter

Crash!
Application example

Restore checkpoint: 1
Application example

Ask upstream node to replay events.
Application example

Upstream node replays events.

Processes events again.
Application example

If Processor 1 fails, it will not be able to repeat the events unless it had logged all non-deterministic decisions.

Events reflect different decisions.

What are you talking about?
Outline

- How is speculation used?
- Guiding speculation
- Experiments
- Final remarks
Speculative Event Processing

• Start processing events as soon as they are available
  – But do not expose effects until the right moment

• Commit in the correct order
  – Validate before commit: confirm if processing used the same state that it would use, re-execute if not
  – Indistinguishable from sequential execution
Updated example

Ordered commits (next to commit)

Ordered commits (waiting events)
Speculative execution example
Speculative execution example
Speculative execution example
Speculative execution example
Speculative execution example
Speculative execution example
Speculative execution example
Speculative execution example
Speculative execution example
Speculative execution example
Speculative execution example
Distributed Speculative Event Processing

• **Assume**
  - no parallelization (so no reordering)
  - have to save the logs in stable storage

• **When finished processing**
  - forward results as *speculative*
  - **Do not wait for the log, but ...**
    - do not let speculative results get out of the system
    - do not commit modifications from speculative events
Outline

• How is speculation used?

• Guiding speculation

• Experiments

• Final remarks
Conflict Predictors

- Too optimistic may be counterproductive
  - Do not process events too much out-of-order
  - When given a choice, pick events that are likely to have few dependencies with missing events
  - Provided by user, or from static/dynamic analysis

- Use **conflict predictor** to pick next event
  - Either a new one or one that should be re-executed
  - “Next=9, but should I speculatively execute event 13?”
Predictor Interface

- Predictor.classify(e)
- Predictor.updateStats(e, stats)
- Predictor.getEvent()
- Predictor.init()
Predictor example

• Default predictor

```java
init(): class[0].Horizon = 1
classify(event): {}
updateStats(event, stats): if avg(stats.aborts,100) < 2%
then Horizon++
else Horizon--
```

• Now we can answer the question
  “Should I speculatively execute event 13?”
Outline

- How is speculation used?
- Guiding speculation
- Experiments
- Final remark
Experiments

- Local optimistic parallelization
  - Overheads & speedups
- Conflict predictors
  - Auto-adaptation
  - Advantages & Pitfalls
- Distributed speculation
  - Hiding latencies
Local parallelization: Speculation costs & speed-ups

![Graph showing speed-up vs. task duration for non-speculative and speculative parallelization with 4 and 8 threads on Sun T1000 hardware.]

- Speculation creation-commit-disposal overheads.
- Few shared-memory accesses.
- Amdahl's law influence.

Hardware: Sun T1000
Local parallelization: Costs & speed-up with sketch operator

Sketch: a limited representation of the stream.
Here, used to compute top-k.
Hundreds of shared-state accesses/per processing.
Conflict predictor: Auto-adaptation

State size varies between 1 and 20.

Sie=1: concurrent executions will conflict.

Size=20: considerable parallelism.
Conflict predictor: Advantage

Advantage of the conflict predictors over a fixed speculation amount.
Conflict predictor: Pitfalls

The auto-adapting predictor performs worse than the fixed horizons.

It optimizes the wrong parameter.
Distributed speculation: motivation - logging costs

Non-speculative: only stable events are sent.

Speculative: send events before logging is finished.

2 components do logging.
Outline

• How is speculation used?
• Guiding speculation
• Experiments
• **Final remarks**
Ongoing/future work

- **Practical work**
  - Update speculation core (i.e., STM)
  - Applications
    - Implement other stream processing operators
  - Conflict predictors
    - Default is too simple, it does not see that aborts are not always bad
    - Try simple machine learning algorithms
Ongoing/future work

- **Conceptual work**
  - Speculation core
    - What concepts are not beneficial to STM in general but are here?
  - Distributed speculation
    - In a distributed ESP system, where could speculation help?
  - Conflict predictors
    - Is it worth to use user-provided information in predictors?
    - Distributed equivalent of conflict predictors
Conclusion

• Transparent way to...
  – parallelize operators, specially useful for user-defined operators
  – hide latency costs (e.g., logging, waiting events)

• Overhead can be order(s) of magnitude lower
  – Hardware support for TM (e.g., ROCK, ASF)
  – Independent STM research
Thank you!

http://streammine.inf.tu-dresden.de
http://wwwse.inf.tu-dresden.de
Distributed Speculative Event Processing

- When speculative event is received
  - Process it, propagate results also speculatively and start writing any decisions to disk
  - When receive notification that event is stable, propagate immediately and then write that to disk
    ⇒ logs will be written in parallel
  - If during the recovery there is something marked as speculative in the log, check with downstream neighbor
Software Transactional Memory

- Normally for optimistic synchronization
  - Isolate effects from instructions inside the transactions until commit
  - So that it seems **atomic**

- TinySTM
  - Word-based lock-based STM implementation
  - Written in portable C, 32/64-bit
Software Transactional Memory

• Advantages
  – Conflicts/dependencies detected dynamically
    • Unveil hidden parallelism
  – Lot of research going on
    • Overhead is dropping
    • Compensation of external actions (aborts)
  – Reduce implementation effort
Software Transactional Memory

- Required changes
  - Ordered commits
  - Timestamp based conflict solver
  - Pause transactions
  - Work (transaction) stealing
  - Predictors
Application example
Stateless (e.g., parsing and transformation) ⇒ trivially parallelizable

Stateful (e.g., stream query operators) ⇒ no need to parallelize

Our focus

Incoming stream of events (high rate, time-stamped)

Ordered

Out-of-order

Ordered

Ordered

Outgoing stream of events

Stateless (low overhead)
Distributed speculation: latency gains

X axis: components that do logging.

Even in a SAN/WAN, the shapes would look similar.

For deterministic components: add “fixed” latency.
Local speculation: Top-k

2 components do logging.