TRACING
and
MONITORING

Distributed Multi-Core Systems

Adaptative Fault Probing
Progress Report

September 17, 2009
École Polytechnique, Montreal

Report contents

- **Architecture** (Michel Dagenais)
- **Recent Progress** (Mathieu Desnoyers)
  - ***Locking Primitives***
  - ***Wait-Free Buffers***
  - ***User-space RCU***
  - ***Multi-Core Architecture Modeling for Formal Verification***

Architecture

Michel Dagenais
Project lead and professor

Recent Progress

Mathieu Desnoyers
Ph.D. candidate
LTTng project lead

Synchronization for Fast and Reentrant Operating System Kernel Tracing

- Synchronization Benchmarks
- Synthetic Trace Clock

Synchronization Benchmarks

Table Speedup of tracing synchronization primitives compared to disabling interrupts and spin lock

Architecture	Spin lock disabling interrupts (Speedup)	Sequence lock and CAS (Speedup)	Preempt disabled and local CAS (Speedup)
Pentium 4	1	3.2	8.1
AMD Athlon(tm)64 X2	1	3.2	5.3
Intel Core2	1	3.7	5.0
Intel Xeon E5405	1	3.8	4.4
ARMv7 OMAP3[a]	1	1.2	8.4

[a] Forced SMP configuration for test module.

- Benchmarks justifying
  - Use of RCU for read-side tracer synchronization
  - Use of local atomic operations for buffering scheme

Synthetic Trace Clock

- Propose an RCU-based synthetic trace clock
  - Extend limited cycle counters to 64 bits.

Lockless Multi-Core High-Throughput Buffering Scheme for Kernel Tracing

- Details LTTng wait-free buffering scheme
- Benchmarks

Benchmarks (LTTng probe)

Table Cycles taken to execute a LTTng 0.140 probe, Linux 2.6.30.

Architecture	Cycles	Core freq. (GHz)	Time (ns)
Intel Pentium 4	545	3.0	182
AMD Athlon64 X2	628	2.0	314
Intel Core2 Xeon	238	2.0	119
ARMv7 OMAP3	507	0.5	1014

Benchmarks (IRQ off vs Lockless)

Table Comparison of lockless and interrupt disabling LTTng probe execution time overhead, Linux 2.6.30.

Architecture	IRQ-off (ns)	Lockless (ns)	Speedup (%)
Intel Pentium 4	212	182	14
AMD Athlon64 X2	381	314	34
Intel Core2 Xeon	128	119	7
ARMv7 OMAP3	1108	1014	8

Benchmarks (Linux kernel build)

Table Linux kernel compilation tracing overhead.

Test	Time (s)	Overhead (%)
Mainline Linux kernel	85	0
Dormant instrumentation	84	-1
Overwrite (flight recorder)	87	3
Normal tracing to disk	90	6

Benchmarks (tbench)

Table `tbench` client network throughput tracing overhead.

Test	Throughput (MB/s)	Overhead (%)
Mainline Linux kernel	12.45	0
Dormant instrumentation	12.56	0
Overwrite (flight recorder)	12.49	0
Normal tracing to disk	12.44	0

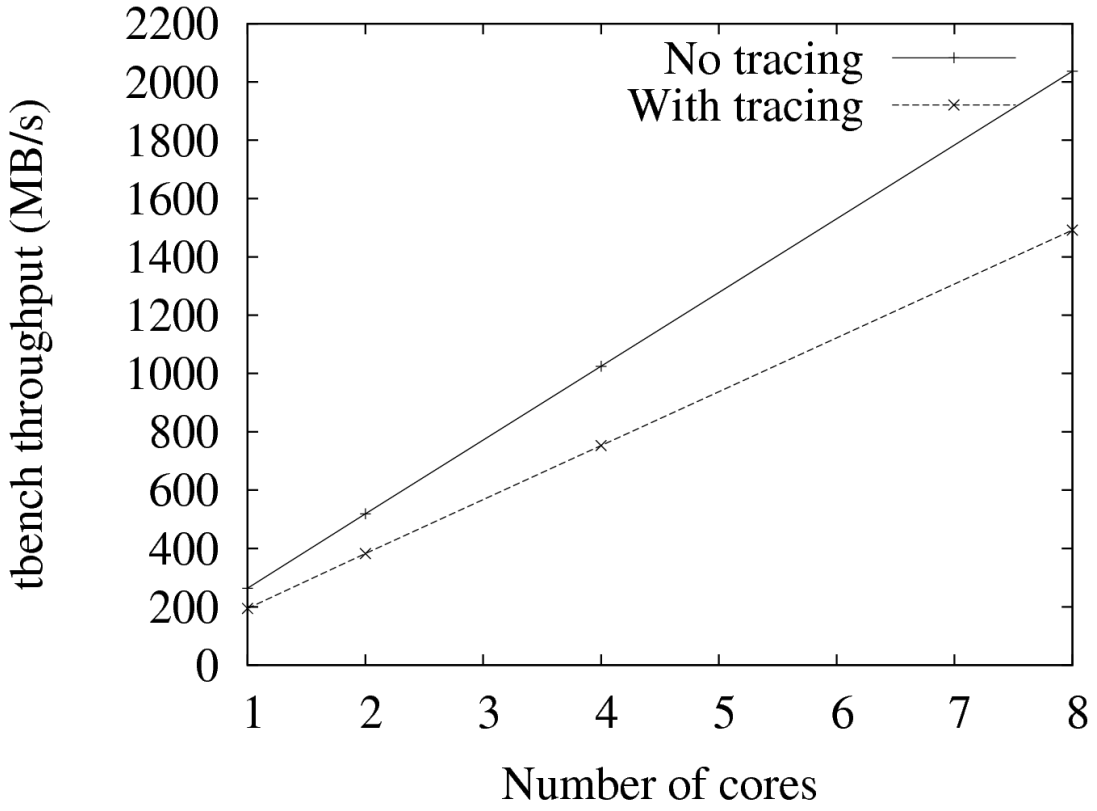
Table `tbench` localhost client/server throughput tracing overhead.

Test	Throughput (MB/s)	Overhead (%)
Mainline Linux kernel	2036.4	0
Dormant instrumentation	2047.1	-1
Overwrite (flight recorder)	1474.0	28
Normal tracing to disk	–	–

Benchmarks (comparison to DTrace)

- Dtrace
  - 1.18µs per event when tracing all system calls to a buffer
- LTTng
  - 0.182µs per event
- Probe speedup: 6.42:1

Scalability



Impact of tracing overhead on local host tbench workload scalability

LTTng Kernel Buffering Scheme

- Low-disturbance
  - Real-time behavior (latency)
    - Wait-free: Strongest type of non-blocking algorithm guarantee.
  - CPU time
    - Low-overhead (e.g. 119ns/event on Intel Xeon)
- NMI-safe

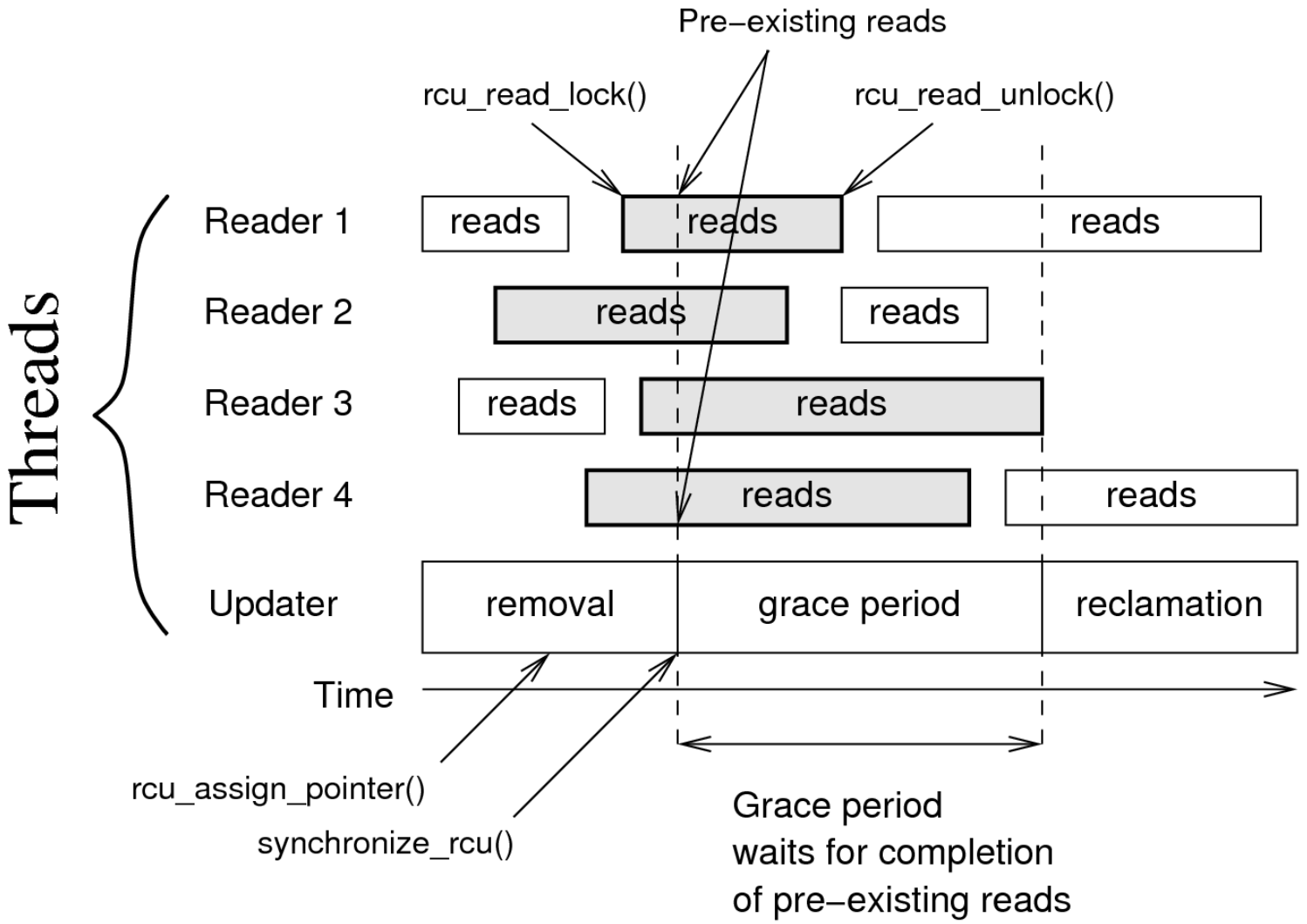
User-Level Implementations of Read-Copy Update

- User-space RCU library
  - Signal-based RCU
  - Memory-barrier RCU
  - QSBR RCU

What is RCU ?

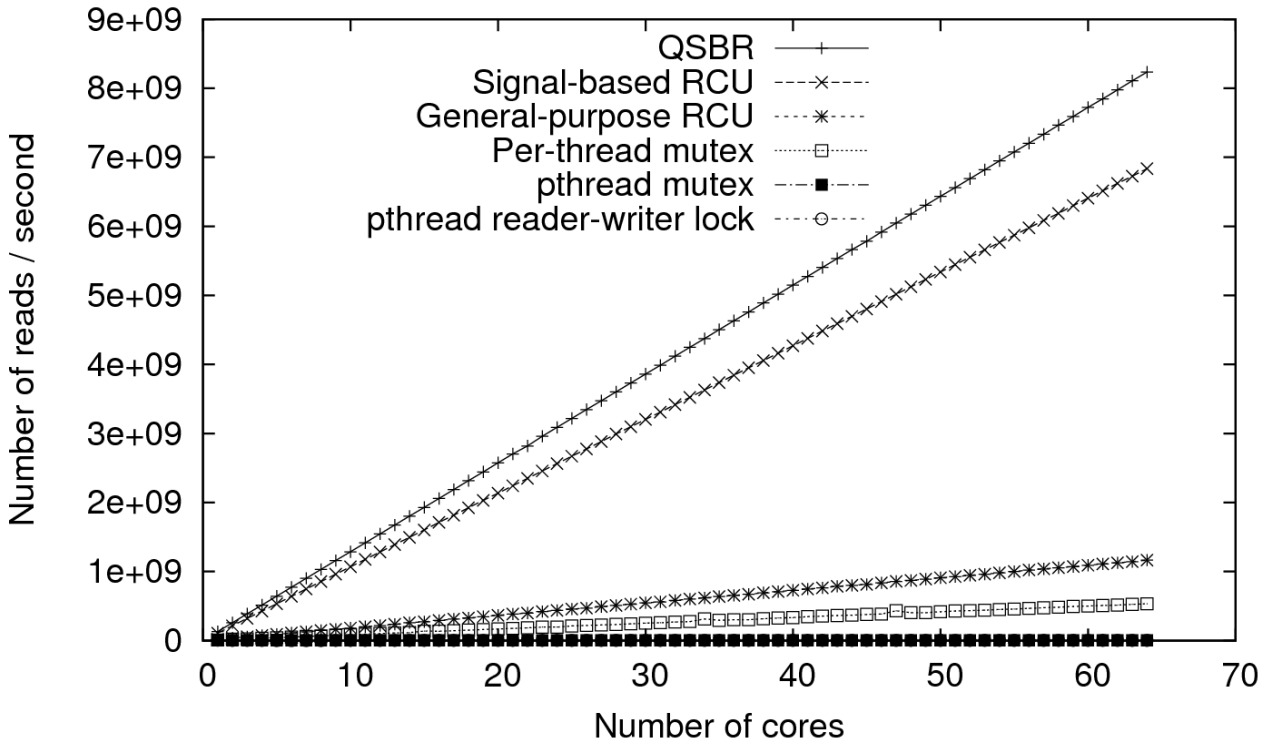
- Method for deferring memory reclamation.
- Permits to perform cheap synchronization without mutual exclusion.
- Benefits from allowing data to be in non-consistent state across processors for a bounded amount of time (grace period).

RCU Overview

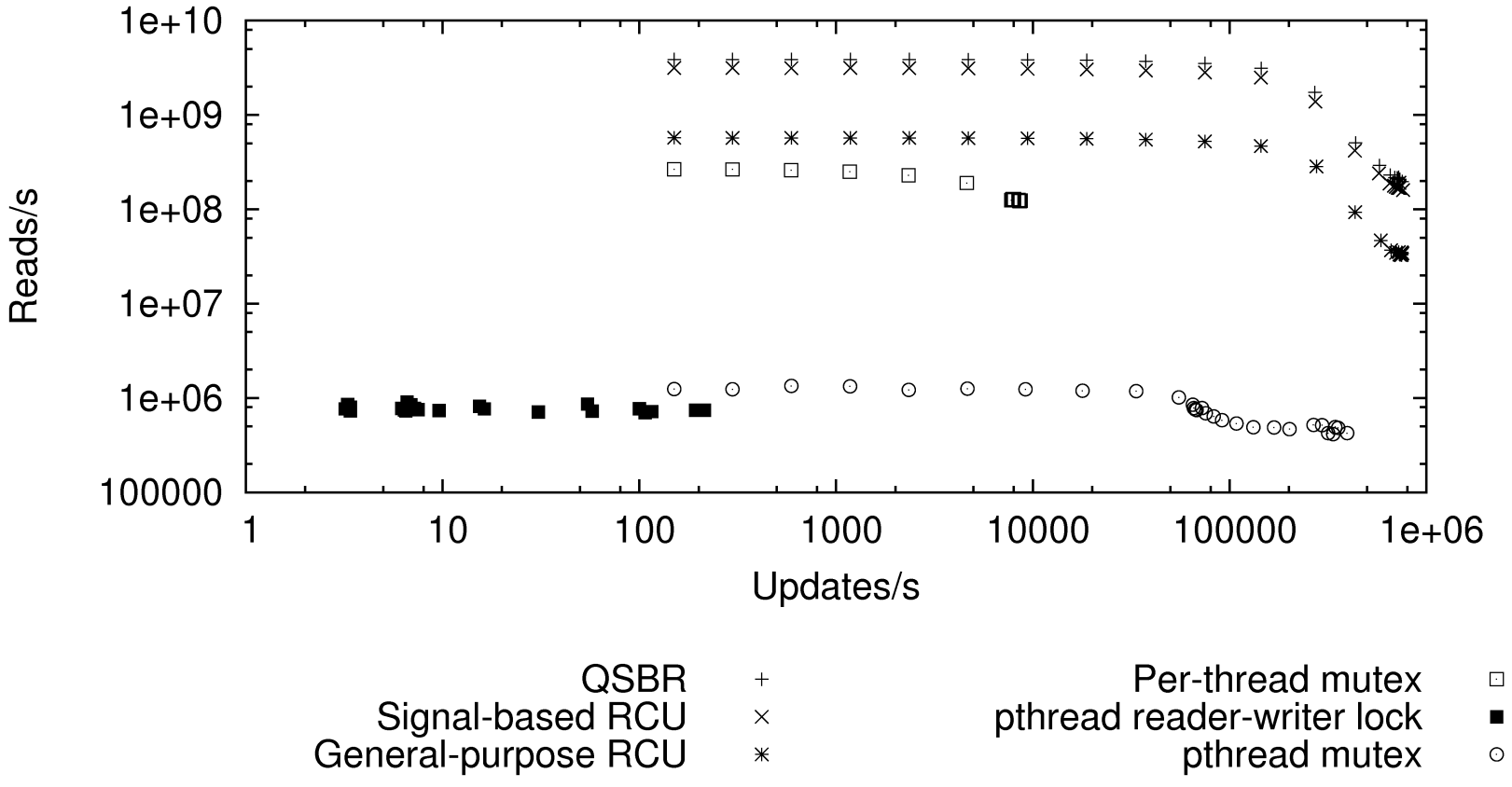


RCU Results (read-side)

- Linear read-side scalability



RCU results (update-side)



64-core Power5+, 32 reader/32 updater threads

Why Using RCU for Tracing ?

- Very efficient read-side
  - Scales linearly as the number of cores increases.
  - Very fast. Does not require atomic operations.
- Real-time properties
  - Wait-free read-side
  - Means: no reader thread can be starved because of this synchronization.
  - Does not affect real-time behavior of a system.

Multi-Core Systems Modeling for Formal Verification of Parallel Algorithms

- Address the problem of modeling synchronization primitive algorithms targeting weakly-ordered multiprocessor and multi-core systems.
- Enable formal verification of these models with the Spin model-checker.

OoOmem Framework

- Created the OoOmem framework to model architectures with:
  - shared-memory multiprocessor,
  - weak memory-ordering.

OoOisched Framework

- Created the OoOisched framework to model
  - compiler-level optimizations,
  - pipelined and superscalar architectures,
  - out-of-order instruction scheduling.

Models

- Created memory exchange scenarios with known behavior to tests the model.
- Created RCU model.
  - Verify grace-period guarantee.
  - Verify publication guarantee.
- Create altered models for error-injection tests.

Model-Checking

- Model-checking successful
  - Fits within available memory and time.
  - Detects errors injected.
  - Full coverage of the proves:
    - Safety
    - Per-process progress (absence of starvation)
      - Proves that RCU model is wait-free.