Userspace RCU Library:

What Linear Multiprocessor Scalability Means for Your Application

Linux Plumbers Conference 2009

Mathieu Desnoyers
École Polytechnique de Montréal

> Mathieu Desnoyers

- Author/maintainer of :
  - LTTV (Linux Trace Toolkit Viewer)
    - 2003-...
  - LTTng (Linux Trace Toolkit Next Generation)
    - 2005-...
  - Immediate Values
    - 2007...
  - Tracepoints
    - 2008-...
  - Userspace RCU Library
    - 2009-...

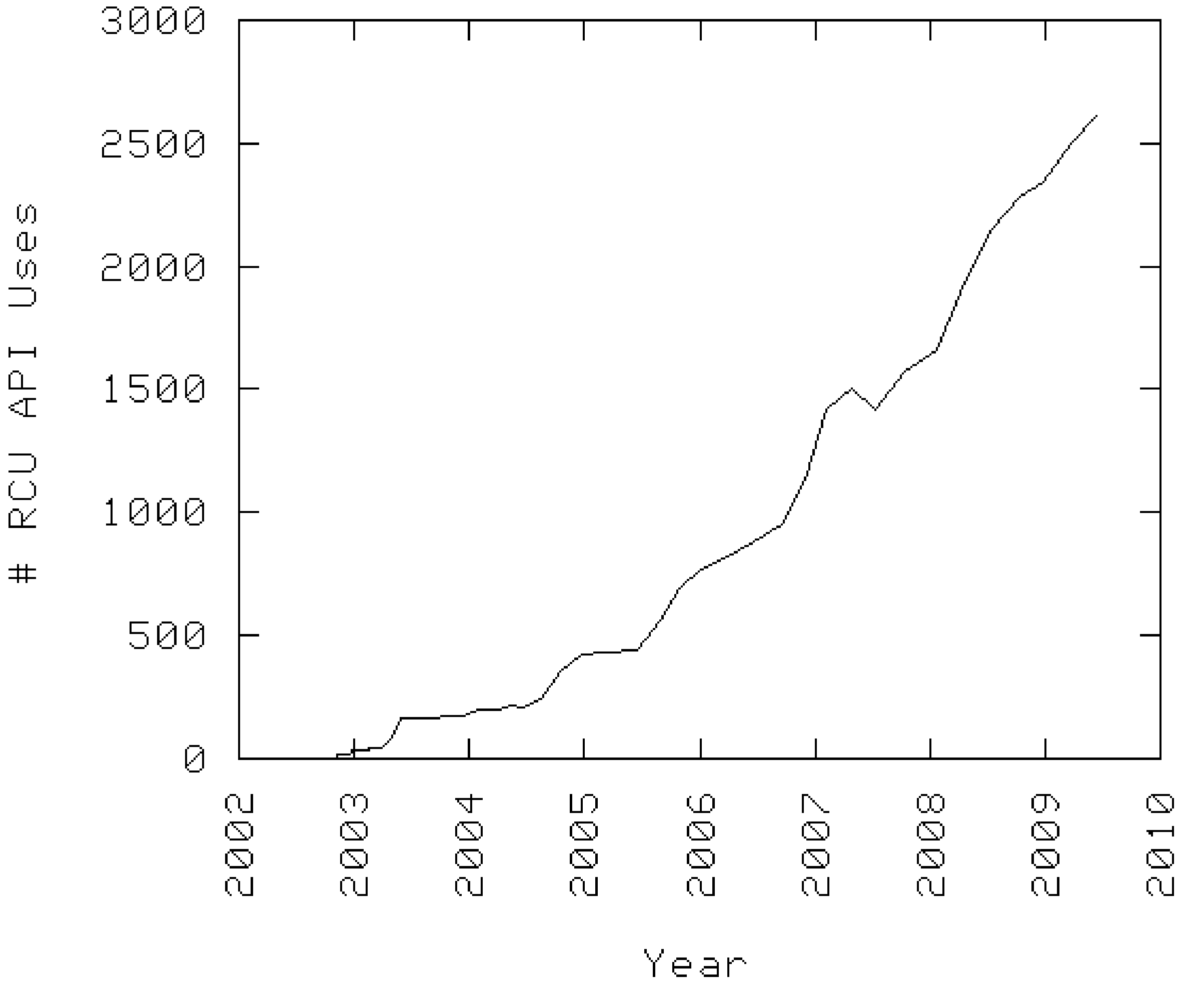
> Contributions by

- Paul E. McKenney
  - IBM Linux Technology Center
- Alan Stern
  - Rowland Institute, Harvard University
- Jonathan Walpole
  - Computer Science Department, Portland State University
- Michel Dagenais
  - Computer and Software Engineering Dpt., École Polytechnique de Montréal

> Summary

- RCU Overview
- Kernel vs Userspace RCU
- Userspace RCU Library
- Benchmarks
- RCU-Friendly Applications

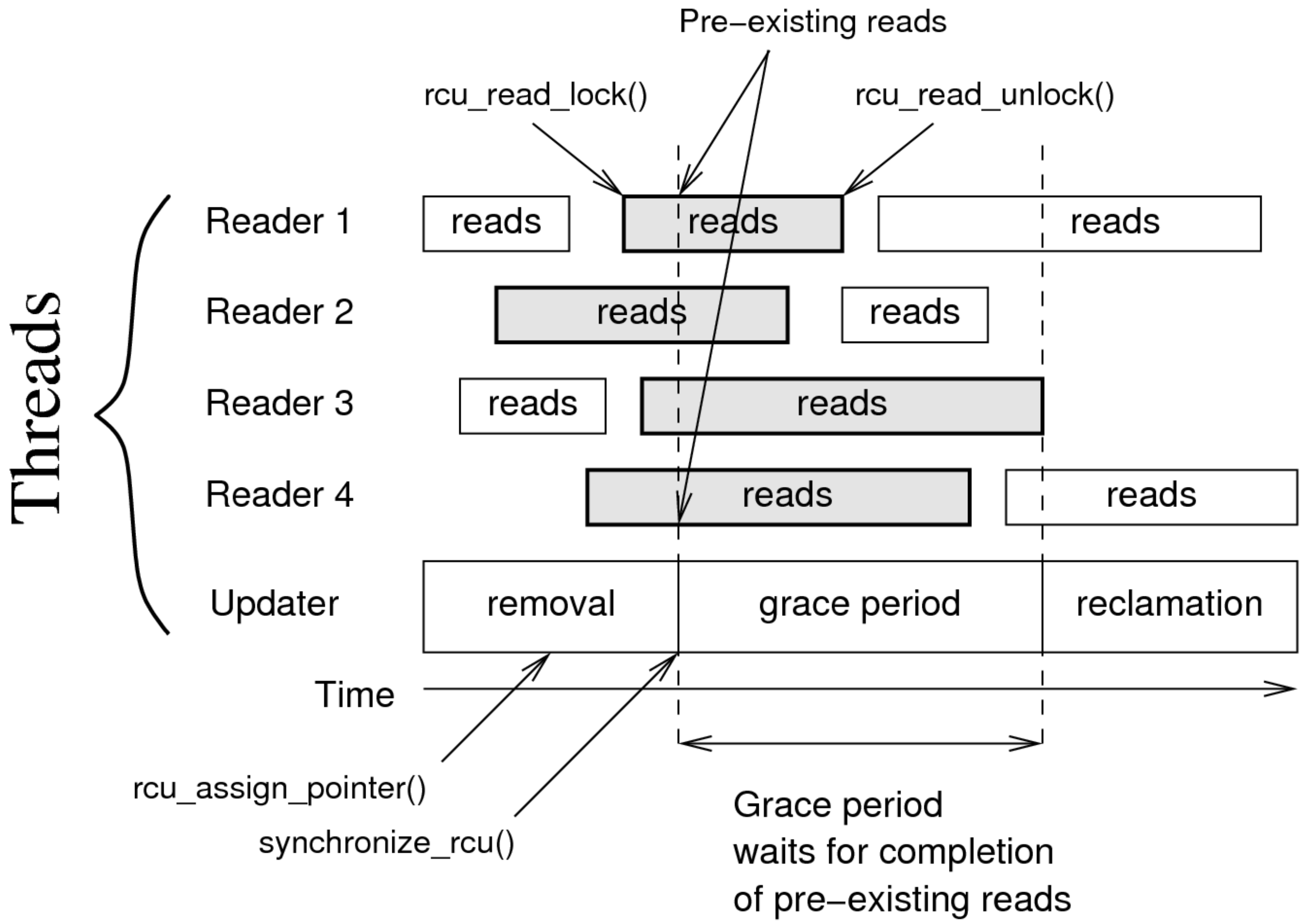
> Linux Kernel RCU Usage



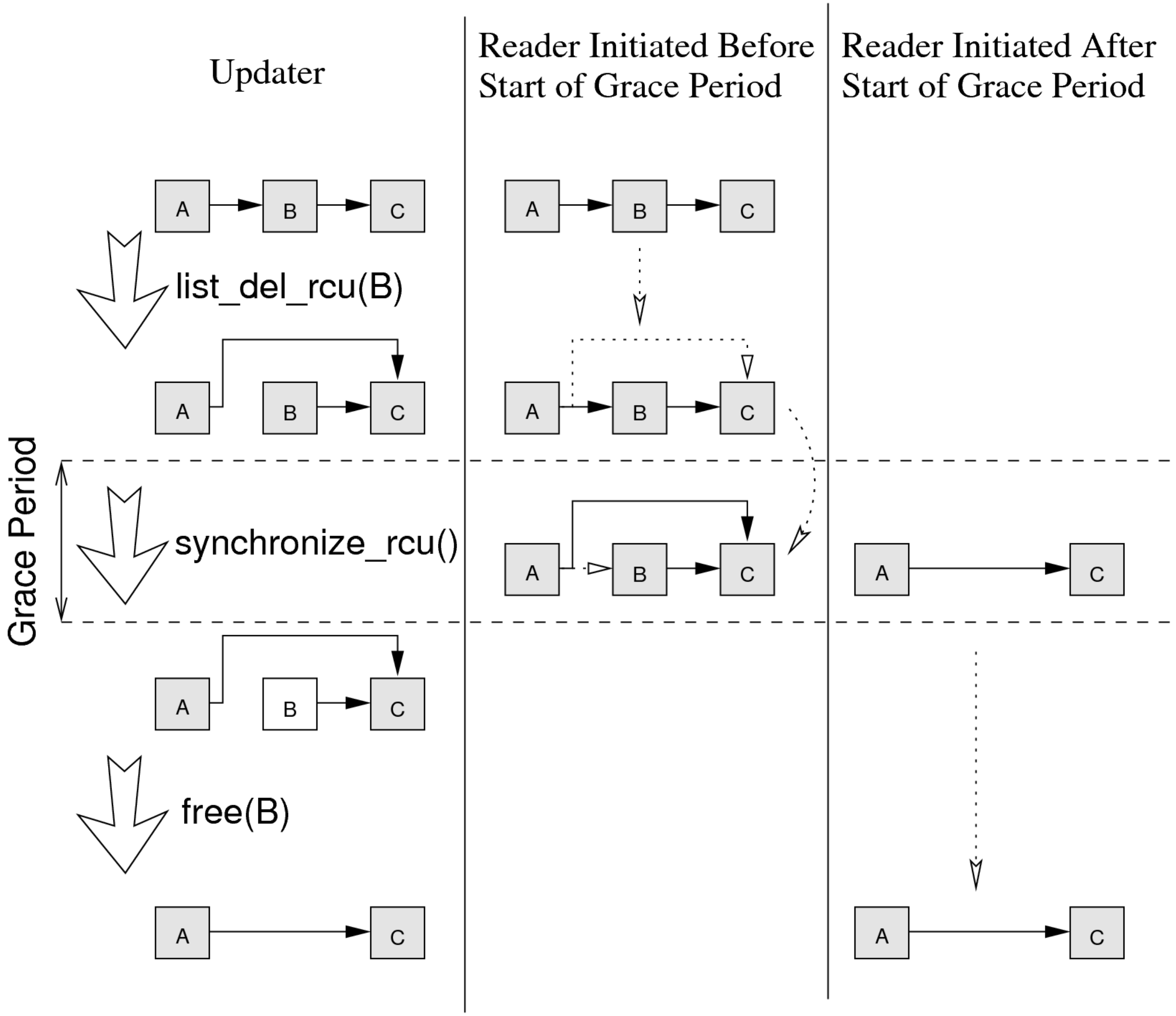
> RCU Overview

- Relativistic programming
  - Updates seen in different orders by CPUs
  - Tolerates conflicts
- Linear scalability
- Wait-free read-side
- Efficient updates
  - Only a single pointer exchange needs exclusive access

> Schematic of RCU Update and Read-Side C.S.



> RCU Linked-List Deletion


Updater

Reader Initiated Before Start of Grace Period

Reader Initiated After Start of Grace Period

> Kernel vs Userspace RCU

- Quiescent state
  - Kernel threads
    - Wait for kernel pre-existing RCU read-side C.S. to complete
  - User threads
    - Wait for process pre-existing RCU read-side C.S. to complete

> Userspace RCU Library

- QSBR
  - liburcu-qsbr.so
- Generic RCU
  - liburcu-mb.so
- Signal-based RCU
  - liburcu.so
- call_rcu()
  - liburcu-defer.so

> QSBR

- Detection of quiescent state:
  - Each reader thread calls rcu_quiescent_state() periodically.
- Require application modification
- Read-side with <u>very</u> low overhead

> Generic RCU

- Detection of quiescent state:
  - rcu_read_lock()/rcu_read_unlock() mark the beginning/end of the critical sections
  - Counts nesting level
- Suitable for library use
- Higher read-side overhead than QSBR due to added memory barriers

> Signal-based RCU

- Same quiescent state detection as Generic RCU
- Suitable for library use, but reserves a signal
- Read-side close to QSBR performance
  - Remove memory barriers from rcu_read_lock()/rcu_read_unlock().
  - Replaced by memory barriers in signal handler, executed at each update-side memory barrier.

> call_rcu()

- Eliminates the need to call synchronize_rcu() after each removal
- Queues RCU callbacks for deferred batched execution
- Wait-free unless per-thread queue is full
- “Worker thread” executes callbacks periodically
- Energy-efficient, uses sys_futex()

> Example: RCU Read-Side

```
struct mystruct *rcudata = &somedata;

/* register thread with rcu_register_thread()/rcu_unregister_thread() */
void fct(void)
{
	struct mystruct *ptr;

	rcu_read_lock();
	ptr = rcu_dereference(rcudata);
	/* use ptr */
	rcu_read_unlock();
}
```

> Example: exchange pointer

```
struct mystruct *rcudata = &somedata;

void replace_data(struct mystruct data)
{
    struct mystruct *new, *old;

    new = malloc(sizeof(*new));
    memcpy(new, &data, sizeof(*new));
    old = rcu_xchg_pointer(&rcudata, new);
    call_rcu(free, old);
}
```

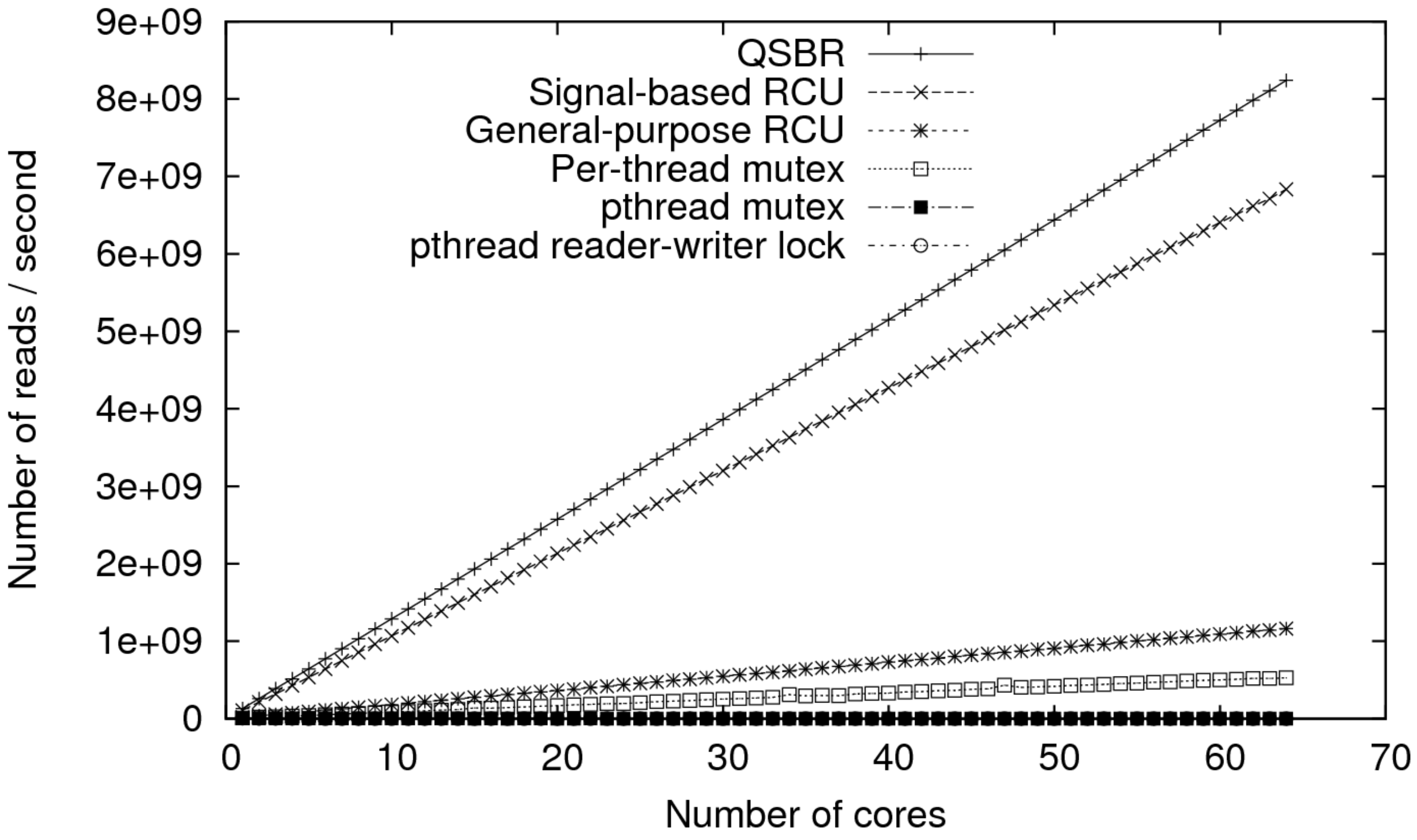
> Example: compare-and-exchange pointer

```
struct mystruct *rcudata = &somedata;

/* register thread with rcu_register_thread()/rcu_unregister_thread() */
void modify_data(int increment_a, int increment_b)
{
    struct mystruct *new, *old;

    new = malloc(sizeof(*new));
    rcu_read_lock();     /* Ensure pointer is not re-used */
    do {
        old = rcu_dereference(rcudata);
        memcpy(new, old, sizeof(*new));
        new->field_a += increment_a;
        new->field_b += increment_b;
    } while (rcu_cmpxchg_pointer(&rcudata, old, new) != old);
    rcu_read_unlock();
    call_rcu(free, old);
}
```

> Benchmarks

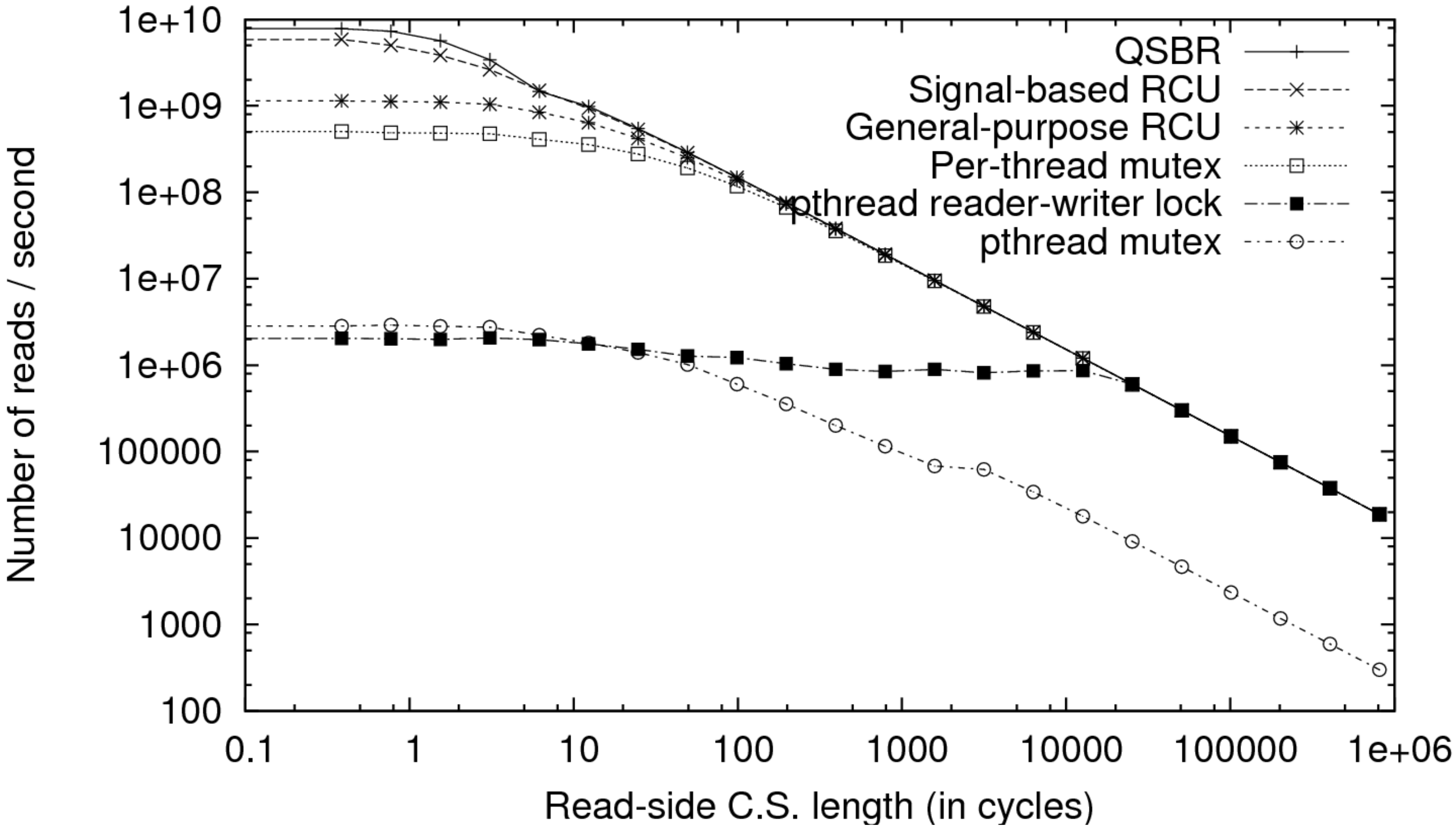
- Read-side Scalability
- Read-side C.S. length impact
- Update Overhead

> Read-Side Scalability



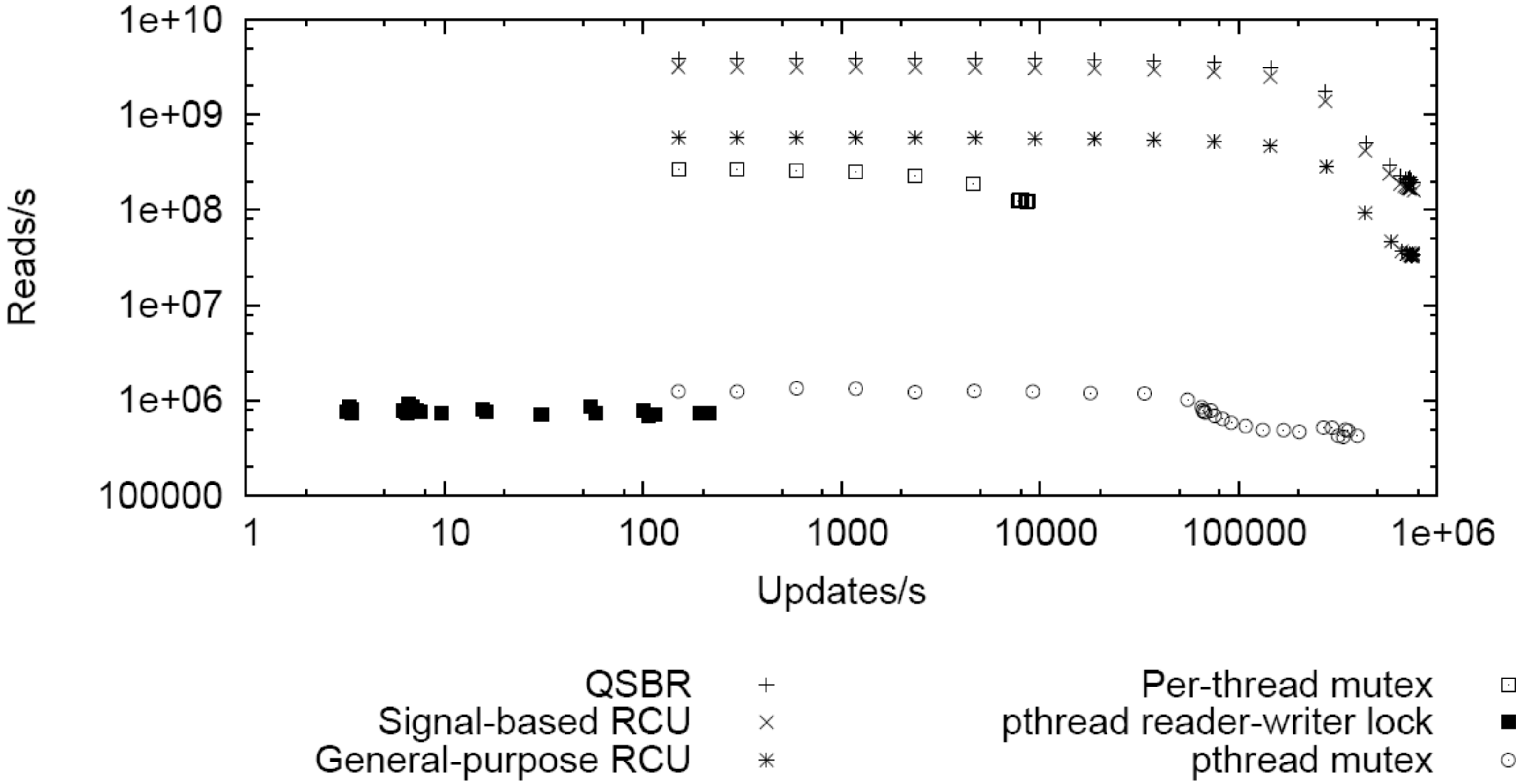
64-cores POWER5+

> Read-Side C.S. Length Impact



64-cores POWER5+, logarithmic scale (x, y)

> Update Overhead



64-cores POWER5+, logarithmic scale (x, y)

> RCU-Friendly Applications

- Multithreaded applications with read-often shared data
  - Cache
    - Name servers
    - Proxy
    - Web servers with static pages
  - Configuration
    - Low synchronization overhead
    - Dynamically modified without restart

> RCU-Friendly Applications

- Libraries supporting multithreaded applications
  - Tracing library, e.g. lib UST (LTTng port for userspace tracing)
    - http://git.dorsal.polymtl.ca/?p=ust.git

> RCU-Friendly Applications

- Libraries supporting multithreaded applications (cont.)
  - Typing/data structure support
    - Typing system
      - Creation of a class is a rare event
      - Reading class structure happens at object creation/destruction (_very_ often)
      - Applies to gobject
        - Used by: gtk/gdk/glib/gstreamer...
    - Efficient hash tables
    - Glib “quarks”

> RCU-Friendly Applications

- Routing tables in userspace
- Userspace network stacks
- Userspace signal-handling
  - Signal-safe read-side
  - Could implement an inter-thread signal multiplexer
- Your own ?

> Info / Download / Contact

- Mathieu Desnoyers
  - Computer and Software Engineering Dpt., École Polytechnique de Montréal
- Web site:
  - http://www.lttng.org/urcu
- Git tree
  - git://lttng.org/userspace-rcu.git
- Email
  - mathieu.desnoyers@polymtl.ca