Distributed Operating Systems
SS2008

Multiprocessor Synchronization using Read-Copy Update

Torsten Frenzel

TU Dresden Operating Systems Group

Outline

- Basics
  - Introduction
  - Examples
- Design
  - Grace periods and quiescent states
  - Grace period measurement
- Implementation in Linux 2.6.25
  - Data structures and functions
  - Examples
- Evaluation
  - Scalability
  - Performance
- Conclusion

Torsten
Frenzel

TU Dresden
Operating
Systems Group

Introduction

- Multiprocessor OS's need to synchronize access to data structures
- → Synchronization primitive is crucial for performance and scalability
- Two important facts
  - Small critical sections (, that access data structures)
  - Data structures with many reads and few updates
- Goals
  - Reducing synchronization overhead
  - Reducing lock contention
  - Deadlock avoidance

Torsten
Frenzel

TU Dresden
Operating
Systems Group

Synchronization Primitives

- Coarse-grained locking (code-based locks)
  - Spinlock (called 'Big kernel lock' in Linux)
  - Reader-writer lock (called 'Big reader lock' in Linux)
- Fine-grained locking (data-based locks)
  - Spinlock
  - Reader-writer lock
  - Per-cpu reader-writer lock
- Lock-free synchronization
  - ➔ Fine grained
  - Avoids disadvantages of locks
  - Hard (to do right) for complex data structures

Torsten Frenzel

TU Dresden Operating Systems Group

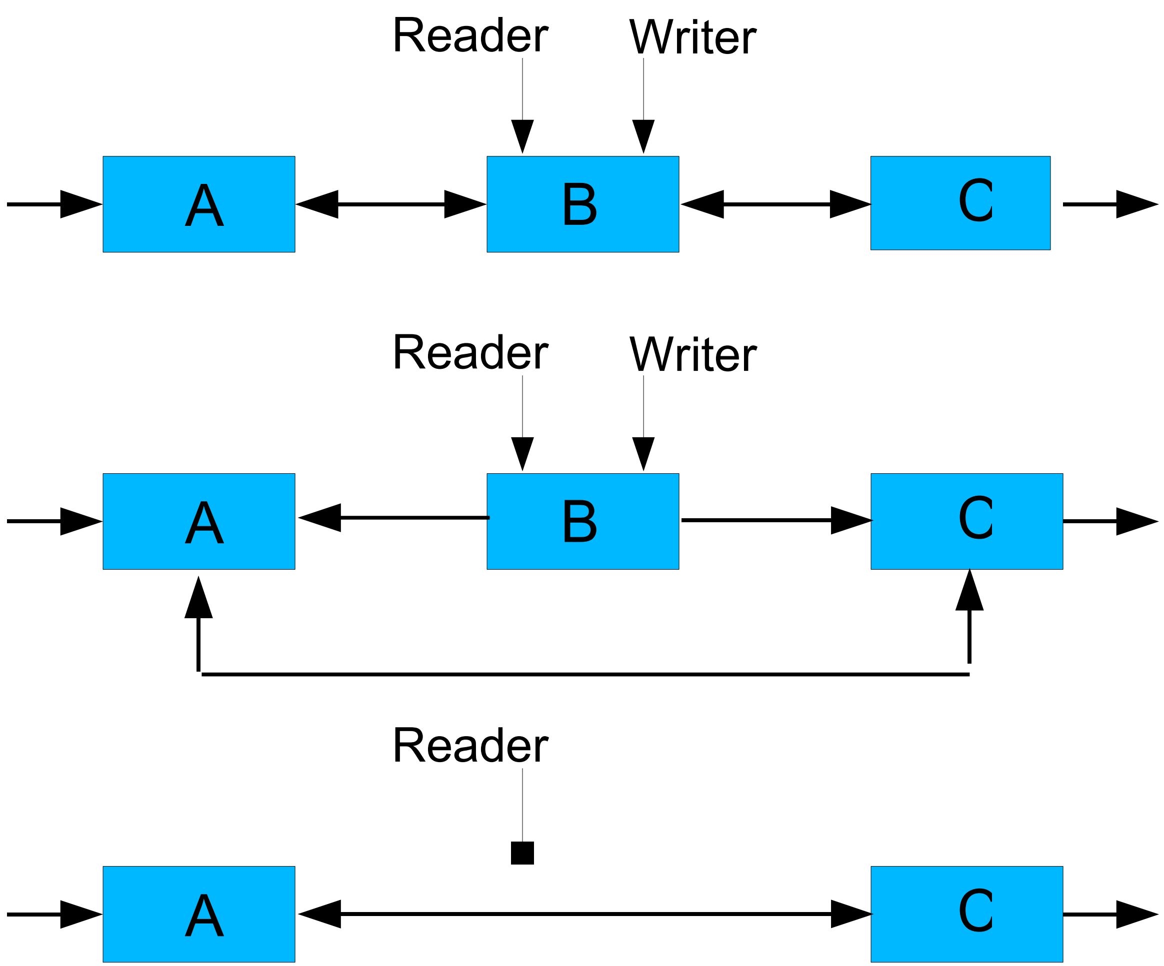
Lockless Synchronization

- Idea
  - Combine advantage of reader-writer locks with lock-free synchronization techniques
  - No locks on reader side
  - Locks only on writer side (no concurrent write operations)
- Prerequisites
  - Many readers and few writers on data structure
  - Short critical sections
  - Properly designed data structure
  - Stale data tolerance for readers
- Problem
  - **When to reclaim memory after update?**
- Solution
  - **Deferred memory reclamation**
  - Two-phase update protocol

Torsten
Frenzel

TU Dresden
Operating
Systems Group

Two-Phase Update - Example



Two-Phase Update - Principle

- Phase 1:
  - Update data structure and make new state visible
- Wait period:
  - Allow existing read operations to proceed on the old state until completed
- Phase 2:
  - Remove old (invisible) state of data structure
- **RCU uses pessimistic approach:**

  „Wait until every concurrent read operations has completed and no pending references to the data structure exist“

Torsten
Frenzel

TU Dresden
Operating
Systems Group

Applications

- Scenarios
  - File descriptor table
  - Routing cache
  - Network subsystem policy changes
  - Hardware configuration
  - Module unloading
- Implementation
  - DYNIX
    - UNIX-based operating system from Sequent
  - Tornado
    - Operating system for large scale NUMA architectures
  - K42
    - Operating system from IBM for large scale architectures
  - Linux

Example 1: List - Read

```
void read(long addr)
{
  read lock(&list lock);
  struct elem *p = head->next;
  while (p != head)
  {
     if (p->address == addr)
     {
        /*read-only access to p */
        read unlock(&list lock);
        return;
     }
     p = p->next;
  }
  read unlock(&list lock);
  return;
}
```

```
void read(long addr)
{

  struct elem *p = head->next;
  while (p != head)
  {
     if (p->address == addr)
     {
        /*read-only access to p */

        return;
     }
     p = p->next;
  }

  return;
}
```

Torsten
Frenzel

TU Dresden
Operating
Systems Group

Example 1: List - Delete

```
void delete(struct elem *p)
{
  struct elem *p = head->next;
  write_lock(&list_lock);
  while (p != head)
  {
    if (p->address == addr)
    {
      p->next->prev = p->prev;
      p->prev->next = p->next;
      write_unlock(&list_lock);

      kfree(p);
      return;
    }
    p = p->next;
  }
  write_unlock(&list_lock);
  return;
}
```

```
void delete(struct elem *p)
{
  struct elem *p = head->next;
  spin_lock(&list_lock);
  while (p != head)
  {
    if (p->address == addr)
    {
      p->next->prev = p->prev;
      p->prev->next = p->next;
      spin_unlock(&list_lock);
      wait_for_rcu();
      kfree(p);
      return;
    }
    p = p->next;
  }
  spin_unlock(&list_lock);
  return;
}
```

Example 2: Filedescriptor Table

```
spin_lock(&files->file_lock);
nfds = files->max_fdset + FDSET INC VALUE;
/* prepare new openset */
/* prepare new exec set */
...
old_openset = xchg(&files->open_fds, new_openset);
old_execset = xchg(&files->close_on_exec, new_execset);
...
nfds = xchg(&files->max_fdset, nfds);
spin_unlock(&files->file_lock);
wait for rcu();
free_fdset(old_openset, nfds);
free_fdset(old_execset, nfds);
```

Grace Periods and Quiescent States

- Definition of a grace period
  - Intuitive: duration until references to data are no longer hold by any thread
  - More formal: duration until every CPU has passed through a quiescent state
- Definition of a quiescent state
  - State of a CPU without any references to the data structure
- How to measure a grace period?
  - Enforcement: induce quiescent state into CPU
  - Detection: Wait until CPU has passed through quiescent state

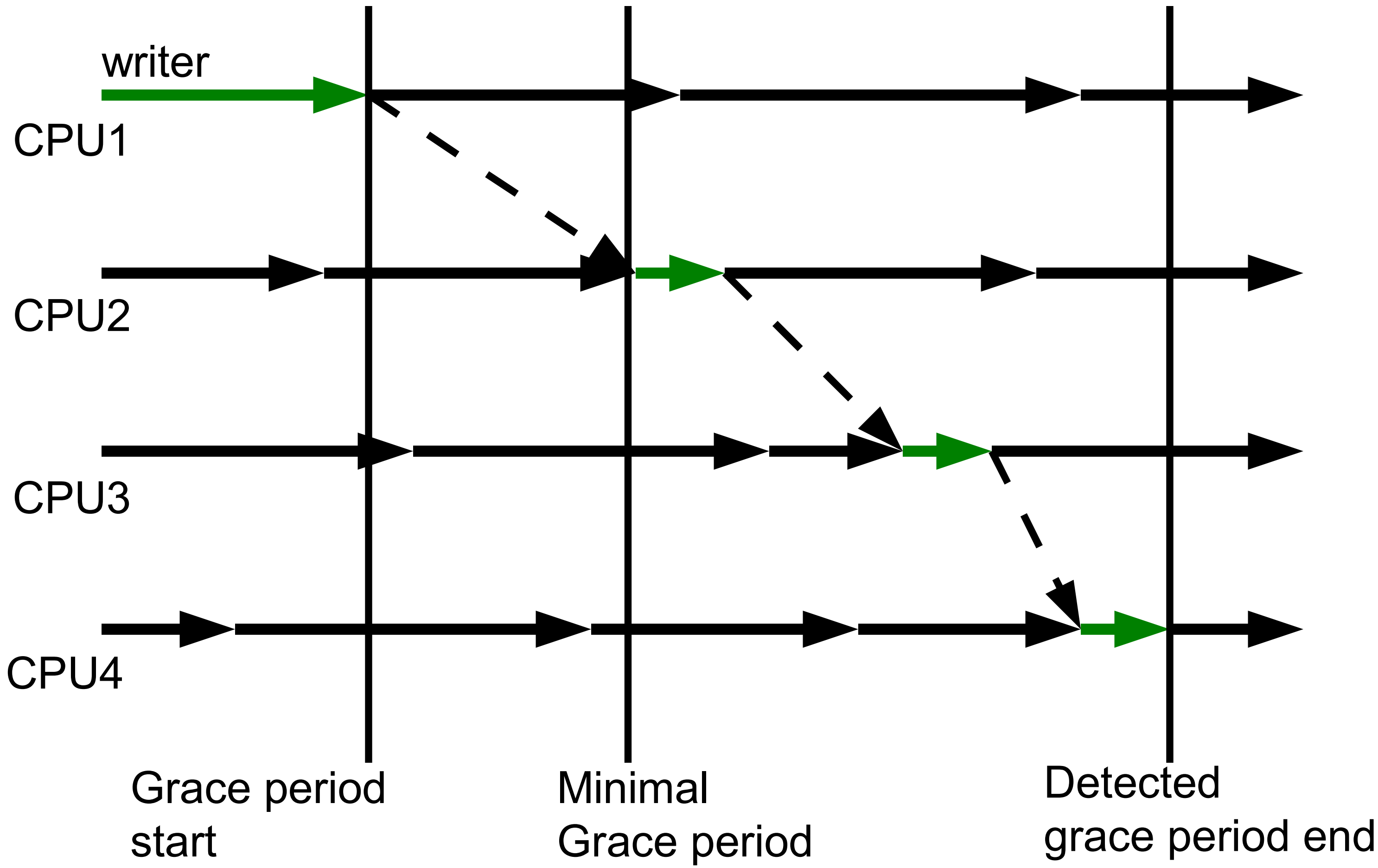
Torsten
Frenzel

TU Dresden
Operating
Systems Group

Quiescent State

- What are good quiescent states?
  - Should be easy to detect
  - Should occur not to frequent or infrequent
- **Per-CPU granularity**
  - For example: context switch, execution in idle loop, kernel entry/exit, CPU goes offline
  - OSs without blocking and preemption in read-side critical sections
- Per-thread granularity
  - OSs with blocking and preemption in read-side critical sections

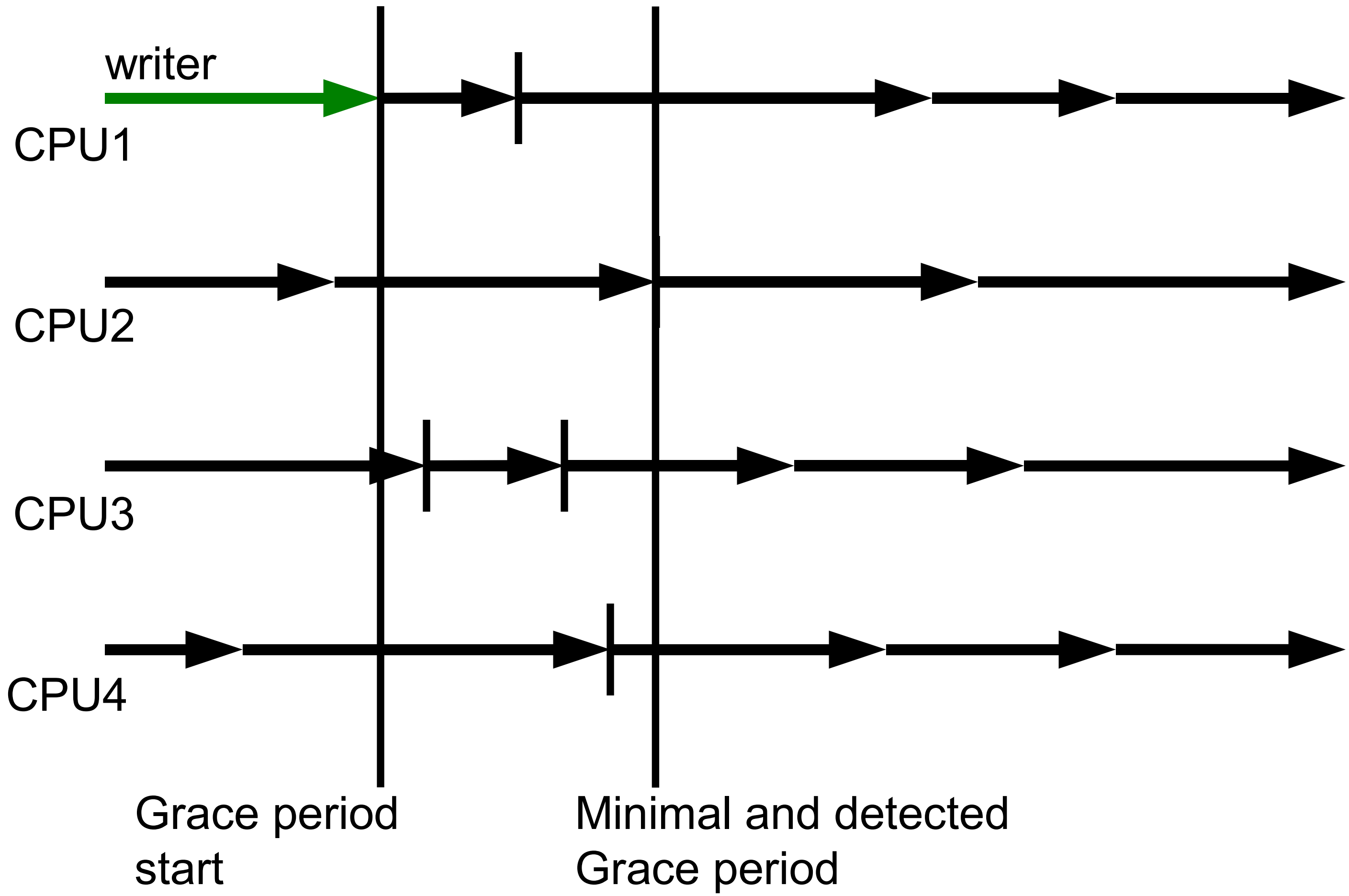
Quiescent State Enforement



Torsten
Frenzel

TU Dresden
Operating
Systems Group

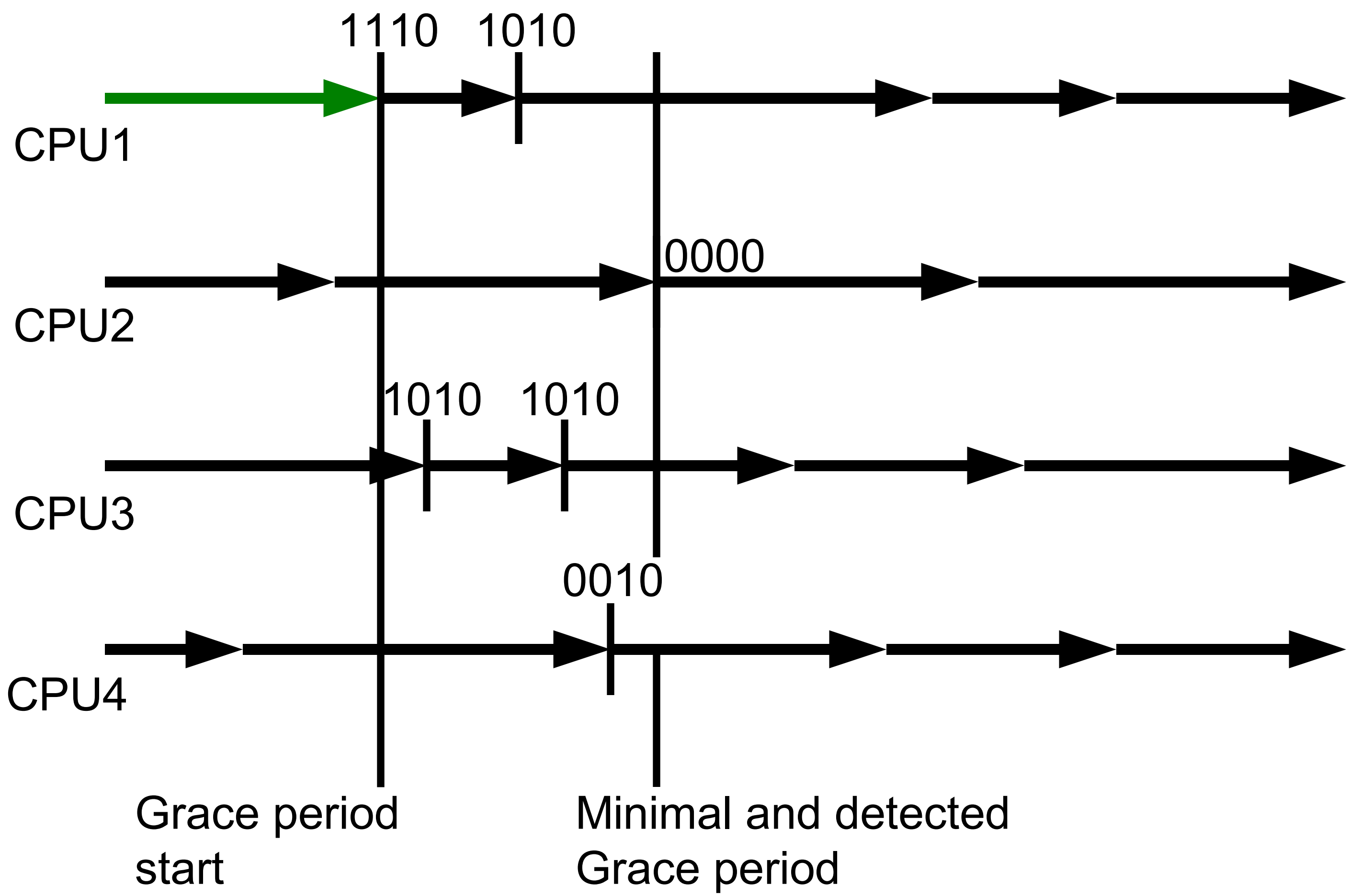
Quiescent State Detection



Torsten
Frenzel

TU Dresden
Operating
Systems Group

Quiescent State Bitmask



Enhancing RCU

- Two observations
  - Measuring grace periods adds overheads
  - Influence on system design
- Consequences
  - Batching of RCU requests
    - Single grace period can satisfy multiple requests
  - Maintaining per-CPU request lists
  - Callback functions for deferred memory reclamation
    - Avoids blocking
  - Low-overhead algorithm for measuring grace periods
  - Measurement framework for long-running critical sections

Linux's RCU Implementation

- Optimized version of RCU
  - Batching with per-CPU request list
  - Separation of CPU-local and global data structures
  - Low overhead if no RCU system is idle
- Grace periods are numbered in increasing order
- One active batch per CPU waiting for completion of current or next grace period
- One next batch per CPU for new requests
- Seperation of quiescent state detection and grace period measurement (RCU core)
- Support for CPU hotplugging
- Support for preemptible read-side critical section
- Support for weak memory consistency

Torsten
Frenzel

TU Dresden
Operating
Systems Group

Data Structures

- Global data: `rcu_ctrlblk`

`cur`	number of current grace period
`completed`	number of recently completed grace period
`next_pending`	flag, requesting another grace period
`cpumask`	bitfield of CPUs, that have to pass through a quiescent state in order complete the ongoing grace period

CPU-local data: `rcu_data`

`quiescbatch`	grace period this CPU thinks as current (should be equally global `cur`)
`qs_pending`	CPU needs to pass through a quiescent
`passed_quiesc`	CPU has passed a quiescent state
`curlist`	closed batch of RCU requests
`batch`	grace period the current batch belongs to
`nxtlist`	open batch of RCU requests

Functional Separation

- Interface
  - `call_rcu()` add RCU callback to batch request list
  - `synchronize_rcu()` wait for grace period to complete
- Tasklet (implements RCU core)
  - Batch processing
    - Invokes callbacks after grace period
  - Finish and start new grace period
  - Quiescent state handling
- Timer-interrupt handler
  - Updates variable passed_quiesc of CPU
  - Schedules tasklet of RCU work is pending
- Scheduler
  - Updates variable passed_quiesc of CPU

Torsten Frenzel

TU Dresden Operating Systems Group

Batch Processing

```
static void __rcu_process_callbacks(struct rcu_ctrlblk *rcp,
                                    struct rcu_data *rdp)
{
    if (rdp->curlist and                    /* Is the current batch list not empty? */
      (rcp->completed >= rdp->batch))       /* Has grace geriod this batch is waiting for completed? */
    {
        ... move current batch list to temporary batch list ...
    }

    if (rdp->nxtlist and                    /* Is the next batch list empty? */
       not rdp->curlist)                    /* Is the current batch list empty? */
    {
        ... move next batch list to current batch list ...
        rdp->batch = rcp->cur + 1;          /* After the next grace period has completed
                                            this batch can be processed */

        if (not rcp->next_pending)          /* Is a new grace period aleady requested? */
        {
            rcp->next_pending = 1;          /* A new grace period has to be started */
            rcu_start_batch(rcp);           /* Try to start a new grace period immediately */
        }
    }

    rcu_check_quiescent_state(rcp, rdp);    /* Check if this CPU gone through a quiescent state */

    if (rdp->donelist)                      /* is there a completed batch? */
        rcu_do_batch(rdp);                  /* process completed batch */
}
```

Quiescent State Handling

```
static void rcu_check_quiescent_state(struct rcu_ctrlblk *rcp,
                                       struct rcu_data *rdp)
{
        if (rdp->quiescbatch != rcp->cur) {    /* Has a new grace period has started? */
                rdp->qs_pending = 1;            /* Reset, for new grace period */
                rdp->passed_quiesc = 0;         /* Reset, for new grace period */
                rdp->quiescbatch = rcp->cur;    /* Set the grace period this cpu is passing through */
                return;
        }

        if (!rdp->qs_pending)                   /* Is this cpu waiting for quiescent state */
                return;                         /* No, go on with work */

        if (!rdp->passed_quiesc)                /* Has this cpu passed a quiescent state */
                return;                         /* No, come back later */

        rdp->qs_pending = 0;                    /* This cpu has passed through a quiescent state! */

        if (rdp->quiescbatch == rcp->cur)       /* sanity check */
                cpu_quiet(rdp->cpu, rcp);       /* update cpu bitmask and check if global grace period
                                                      completed */
}
```

Finish and Start of Grace Period

```
static void cpu_quiet(int cpu, struct rcu_ctrlblk *rcp)
{
    cpu_clear(cpu, rcp->cpumask);            /* Clear bit of this cpu in cpu bitmask */

    if (cpus_empty(rcp->cpumask))            /* Has a grace period completed? */
    {
        rcp->completed = rcp->cur;           /* Set completed grace period to current grace period */
        rcu_start_batch(rcp);                /* Try to start a new grace period, immediatly */
    }
}

static void rcu_start_batch(struct rcu_ctrlblk *rcp)
{
    if (rcp->next_pending and                /* Should a new grace period be started? */
      rcp->completed == rcp->cur)            /* Is completed grace period equal current grace period? *
    {
        rcp->next_pending = 0;               /* Reset grace period trigger */

        rcp->cur++;                          /* A new global grace period starts */

                                             /* Update cpu bitmask */
        cpus_andnot(rcp->cpumask, cpu_online_map, nohz_cpu_mask);

        rcp->signaled = 0;
    }
}
```

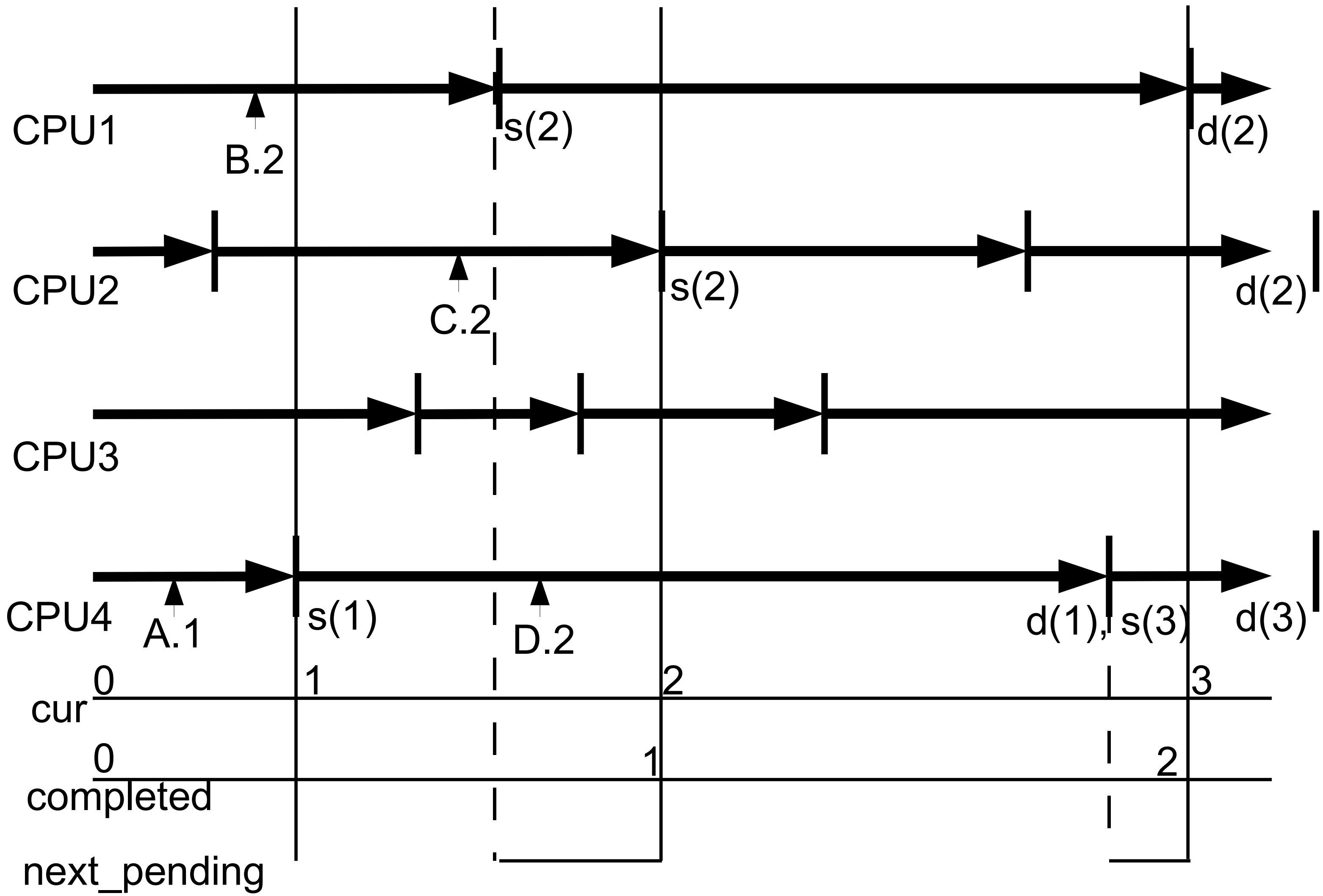
Torsten
Frenzel

TU Dresden
Operating
Systems Group

When to invoke the RCU Core?

```
static int __rcu_pending(struct rcu_ctrlblk *rcp, struct rcu_data *rdp)
{
        /* This cpu has pending rcu entries and the grace period
           for them has completed. */
        if (rdp->curlist and rcp->completed >= rdp->batch)
                return 1;

        /* This cpu has no pending entries, but there are new entries */
        if (not rdp->curlist and rdp->nxtlist)
                return 1;

        /* This cpu has finished callbacks to invoke */
        if (rdp->donelist)
                return 1;

        /* The rcu core waits for a quiescent state from the cpu */
        if (rdp->quiescbatch != rcp->cur or rdp->qs_pending)
                return 1;

        return 0;
}
```

Linux RCU Example



Torsten
Frenzel

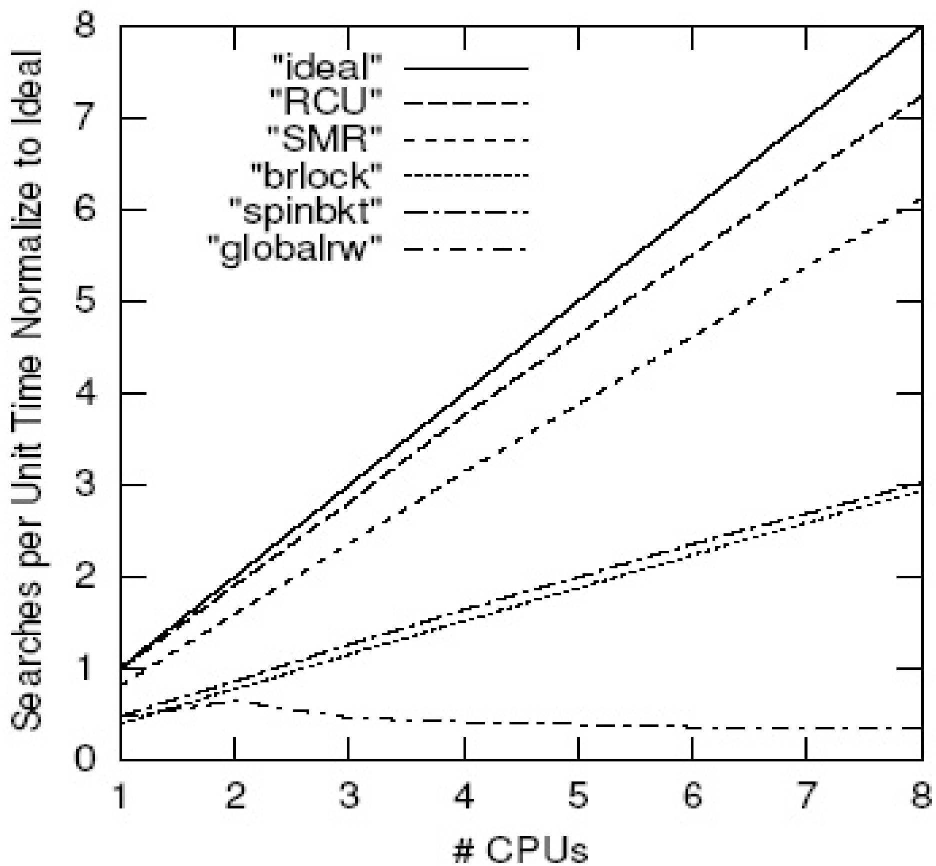
TU Dresden
Operating
Systems Group

Scalability and Performance

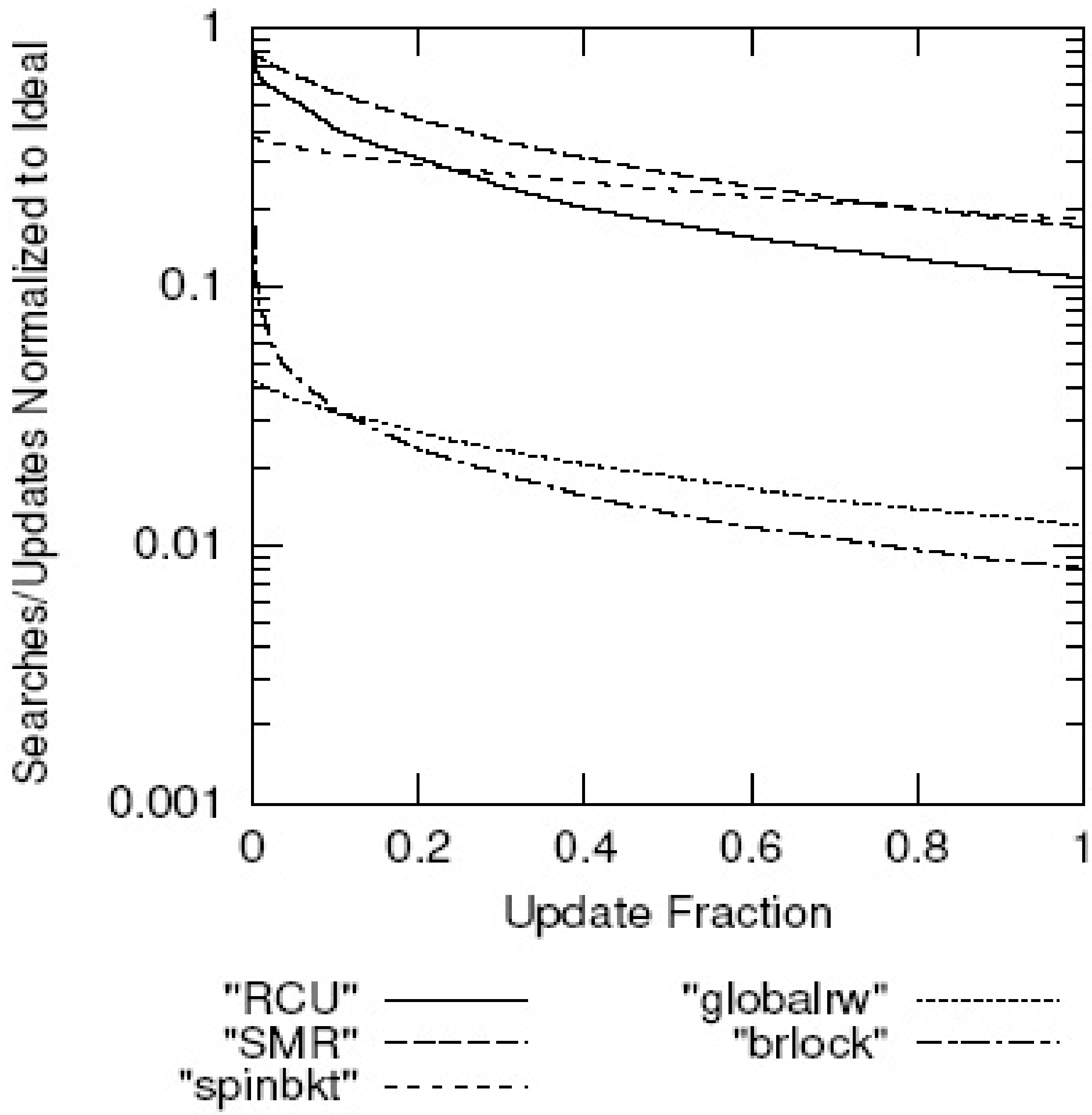
- How does RCU scale?
  - Number of CPUs (n)
  - Number of read-only operations
- How does RCU perform?
  - Fraction of accesses that are updates (f)
  - Number of operations per unit
- What other algorithms to compare to?
  - Global reader-writer lock (globalrw)
  - Per-CPU reader-writer lock (brlock)
  - Data spinlock (spinbkt)
  - Lock-free using safe memory reclamation (SMR)

Scalability

- Hashtable benchmark



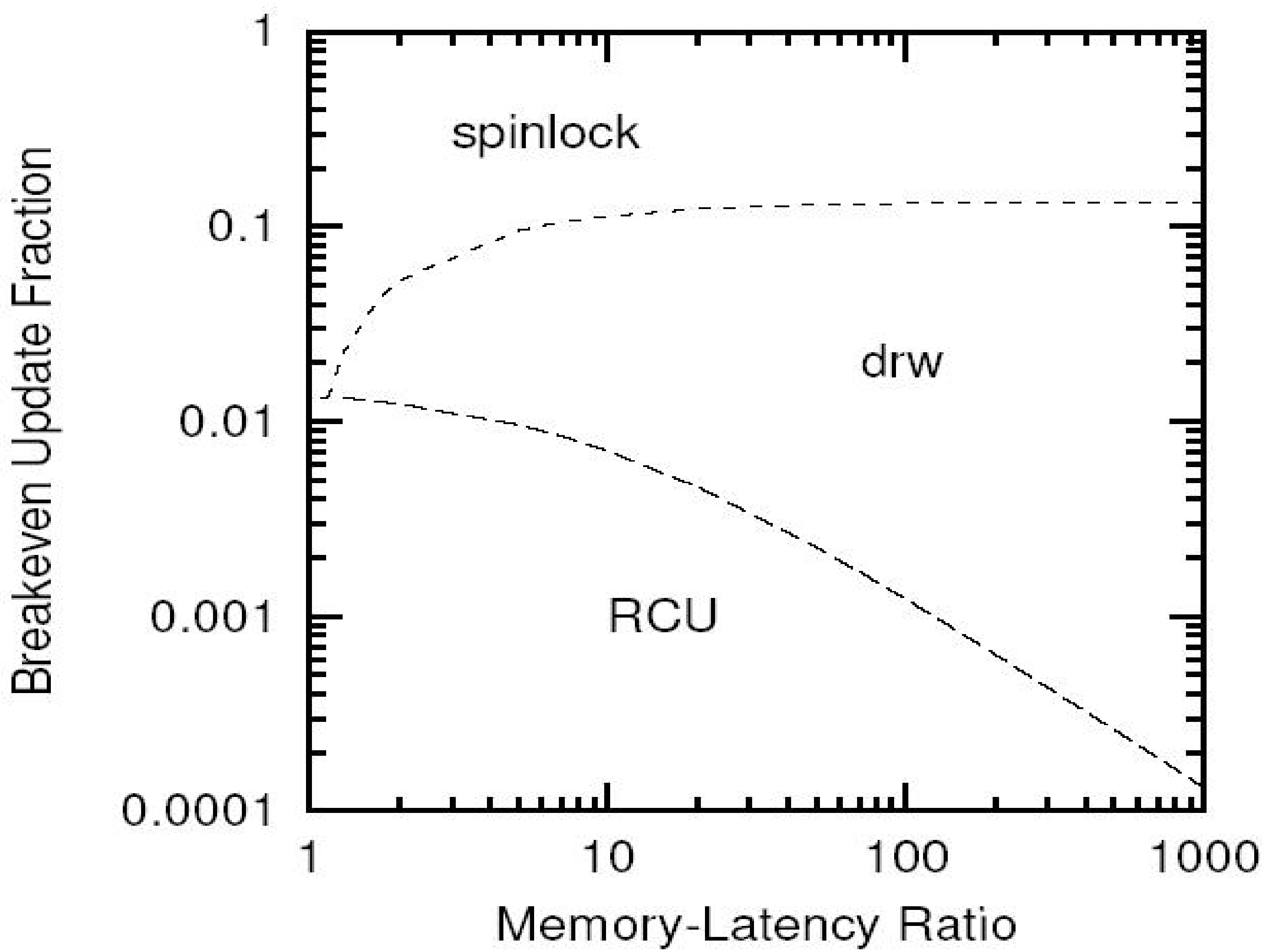
Performance



Performance vs. Complexity

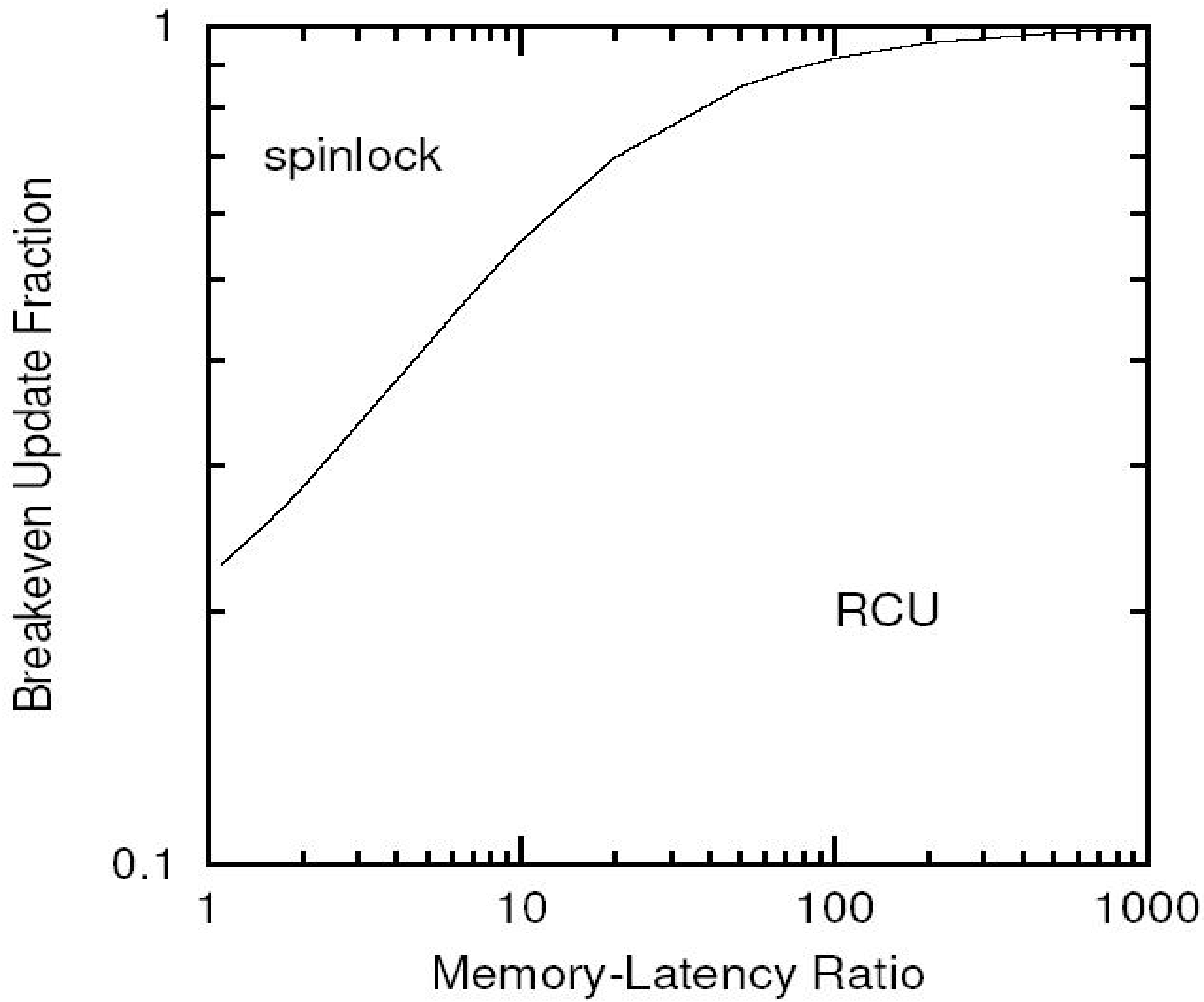
- When should RCU be used?
  - Instead of simple spinlock? (spinlock)
  - Instead of per-CPU reader-writer lock? (drw)
- Under what conditions should RCU be used?
  - Memory-latency ratio (r)
  - Number of CPUs (n=4)
- Under what workloads?
  - Fraction of access that are updates (f)
  - Number of updates per grace period (λ = {small, large})

Bad Case – Small Update Fraction



Distributed
Operating
Systems

Good Case – Large Update Fraction



Torsten
Frenzel

TU Dresden
Operating
Systems Group

Concluding Remarks

- RCU performance and scalability
  - Linear scaling with increasing number of CPUs
  - Very good performance under high contention
- RCU modifications
  - Support for weak consistency models
  - Support for NUMA architectures
  - Without stale data tolerance
  - Support for preemptible critical sections
- Other memory reclamation schemes
  - Lock-free reference counting
  - Hazard-pointer-based recalamation
  - Epoch-based reclamation

References

- Read-Copy Update: Using Execution History to Solve Concurrency Problems; McKenney, Slingwine; 1998
- Read-Copy Update; McKenney, Karma, Arcangeli, Krieger, Russel; 2003
- Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation; Hart McKenney; Brown; 2006
- Linux Journal: Introduction to RCU; McKenney 2004; http://linuxjournal.com/article/6993
- Linux Journal: Scaling dcache with RCU; McKenney; 2004; http://linuxjournal.com/arcticle/7124