



IBM Linux Technology Center

Parallel C Experience in the Linux Kernel

Paul E. McKenney



IBM Linux Technology Center
April 17, 2007

Agenda

- Desiderata
 - Review of Parallel C Use Cases in Linux
 - Explicit Memory Fence Operations
 - Dependency Ordering
 - Compiler Fence Operations
 - Atomicity of Aligned Basic Types
 - Conclusions and Recommendations
-
- Summary of N2153 and cpp-threads discussion



Desiderata

- Solid and well-defined primitives are available for parallel code
 - ▶ Don't force programmers to “hack” without good reason
- Support/codify existing parallel C/C++ practice
 - ▶ Don't invalidate existing programs without good reason
 - ▶ There is a *lot* of existing parallel C/C++ code
- Leverage existing compilers and optimizations
 - ▶ Don't invalidate existing compilers without good reason



Desiderata: APIs

- Memory Fences:
 - ▶ `void acquire_fence(void);`
 - ▶ `void release_fence(void);`
 - ▶ `void ordered_fence(void);`
 - ▶ `T *load_dependent_acquire(T *p);`
 - ▶ `void store_dependent_release(T *p, T *newval);`

- Tag Arguments and Return Values to Propagate Dependencies
 - ▶ `dependence_propagate`

- Compiler Fences:
 - ▶ `void compiler_fence(void);`
 - ▶ `T compiler_load_dependent_acquire(T *p);`
 - ▶ `void compiler_store_dependent_release(T *p, T newval);`



Review of Parallel C Use Cases in Linux



Linux Kernel Use Cases

- Split counters
- Hash tables with lockless readers
- Communication with signal/interrupt/exception handlers
- Additional RCU use cases
 - ▶ Dereference field in struct from pointer
 - ▶ Dereference field referencing list of structs from pointer
 - ▶ Dereference reference counter field from pointer
 - ▶ Dereference array field from pointer
 - ▶ Dereference lock field from pointer



Use Case: Split Counter

- Per-CPU counters (e.g., “int”)
- Each CPU modifies its own counter
- To approximate value of counter, sum all per-CPU counters
 - ▶ Requires only atomicity of loads and stores: no locks, special atomic instructions, or memory fences.
 - ▶ Non-SC results perfectly permissible: who cares which of two concurrent packets arrived first?
 - ▶ Commonly implemented using normal loads/stores to aligned/padded variables.
- Code is trivial (see next slide)



Use Case: Split Counter

```
DEFINE_PER_CPU(unsigned long, mycounter) = {0};

void counter_add(unsigned long *cp, unsigned long v) {
    __get_cpu_var(cp) += v;
}

unsigned long counter_value(unsigned long *cp) {
    int cpu;
    unsigned long sum;

    for_each_possible_cpu(cpu) {
        sum += per_cpu(cp, cpu);
    }
    return sum;
}
```



Use Case: Hash Tables With Lockless Readers

- Dense array for hash table with linked lists for each bucket
- Updates (insertion and deletion) concurrent with search
- Search much more frequent than updates
 - ▶ Routing tables: search per-packet, update infrequently if ever
 - ▶ Security policy structures: search per operation, update infrequently if ever (permissions, firewalls, ...)
 - ▶ Filesystem directory cache: search per filesystem operation, update on modification or on cache miss/flush
- Use memory ordering, memory barriers, or dependency ordering for extremely low-overhead search



Use Case: Hash Tables With Lockless Readers

```
struct foo {  
    struct foo *next;  
    int key;  
};  
struct foo *hashtable[NUM_BUCKETS]  
DEFINE_SPIN_LOCK(foo_lock);
```



Use Case: Hash Tables With Lockless Readers

```
int find_foo(int key) {
    struct foo *p;
    int retval;

    rcu_read_lock();
    p = rcu_dereference(hashtable[foo_hash(key)]);
    while (p != NULL && p->key < key)
        p = rcu_dereference(p->next);
    retval = p != NULL && p->key == key;
    rcu_read_unlock();
    return retval;
}
```



Use Case: Hash Tables With Lockless Readers

```
int insert_foo(int key) {
    struct foo *newp, *p, **plast;
    int retval = 0;

    spin_lock(&foo_lock);
    plast = &hashtable[foo_hash(key)];
    p = *plast;
    while (p != NULL && p->key < key) {
        p = p->next;
        plast = &p->next;
    }
    if (p == NULL || p->key != key) {
        newp = kmalloc(sizeof(*newp), GFP_KERNEL);
        if (newp != NULL) {
            newp->key = key;
            newp->next = p;
            rcu_assign_pointer(*plast, newp);
            retval = 1;
        }
    }
    spin_unlock(&foo_lock);
    return retval;
}
```



Use Case: Hash Tables With Lockless Readers

- `rcu_read_lock()` and `rcu_read_unlock()`:
 - ▶ Compile to nothing on non-preemptive kernels
 - ▶ Deterministic implementation with no high-overhead instructions for preemptive and realtime cases
- `rcu_dereference()`:
 - ▶ Simple assignment, possible with compiler control
 - Relies on either dependency ordering or strong memory model
 - ▶ Memory barrier required on DEC Alpha
- `rcu_assign_pointer()`:
 - ▶ Memory barrier preceding assignment
- Removal not discussed here: see RCU papers for full details
 - ▶ Basic trick: wait until all readers are done before freeing



Use Case: Communication With Handler

- Memory fences not needed: only one CPU involved
- Compiler control *is* needed
- Example: NMI-based profiling

```
struct profile_buf {  
    unsigned long size;  
    int count[0];  
} *pb;
```



Use Case: Communication With Handler

```
int start_profile(int size) {
    struct profile_buf *p;
    p = kzalloc(sizeof(*p) + size * sizeof(p->size), GFP_KERNEL);
    if (p == NULL) return 0;
    p->size = size;
    barrier();
    pb = p;
}

void nmi_prof(unsigned long pc) {
    struct profile_buf *p;

    p = rcu_dereference(pb);
    if (p == NULL) return;
    if (pc > p->size) return;
    p->count[pc]++;
}
```



Use Case: Communication With Handler

- `barrier()` primitive prohibits optimizations that would reorder memory references across the `barrier()` call
- Definition in gcc for Linux kernel:

```
#define barrier() __asm__ __volatile__("":::"memory");
```



Use Case: RCU

- Generic RCU read-side code usage pattern:

```
rcu_read_lock();  
do_something();  
p = rcu_dereference(gp);  
do_something_with(p);  
rcu_read_unlock();
```

Different RCU use cases have different `do_something_with()` definitions.



Use Case: RCU

Pattern	# Uses	Expansion of do_something_with()
field	229	Dereference a field: <code>p->b</code>
field-list	49	Traverse multiple structures: <code>p->q->a</code>
refcnt	47	Acquire a reference: <code>atomic_inc(&p->refcnt)</code>
field-array	38	Dereference an array field: <code>p->a[i]</code>
lock	10	Acquire a per-struct lock: <code>spin_lock(&p->lock)</code>
cast	6	Casts the pointer: <code>((struct foo *)p)->f</code>



Linux 2.6.20 kernel

Explicit Memory Fence Operations



Explicit Memory Fence Operations

- `void acquire_fence(void);`
 - ▶ Lea/Click `postLoadFence()`
- `void release_fence(void);`
 - ▶ Lea/Click `preStoreFence()`
- `void ordered_fence(void);`
 - ▶ Full fence
- `T load_dependent_acquire(T *p);`
 - ▶ Lea/Click `perObjectPostLoadFence()`
 - ▶ `Dimov atomic_load_address()`, but constrained
 - ▶ Implements Linux `rcu_dereference()`
- `void store_dependent_release(T *p, T *newval);`
 - ▶ Lea/Click `perObjectPreStoreFence`
 - ▶ Implements Linux `rcu_assign_pointer()`
- Expand on last two in subsequent slides



load_dependent_acquire() / perObjectPostLoadFence

- Ensure that a load of a variable is ordered before any operations dependent on that load
 - ▶ Many use cases, as seen on earlier slides
- Strategies:
 - ▶ Prohibit compiler optimizations that break dependency chains
 - Generally believed overly restrictive
 - ▶ Classify dependency-breaking optimizations into permitted and prohibited classes
 - Generally believed to be infeasible, especially for yet-to-be-discovered optimizations
 - ▶ Emit a full acquire fence
 - Appropriate for strongly ordered memory models
 - ▶ Provide mechanisms to allow the programmer to flag particular dependencies that must be maintained
 - Appropriate for weakly ordered memory models



load_dependent_acquire() / perObjectPostLoadFence

- Example use:

```
struct pubdata {
    int a;
    int b;
};

struct pubdata *pd = NULL;
int x;
int y;

/* ... */

localpd = load_dependent_acquire(&pd);
r1 = localpd->a; /* ordered. */
localpd->b = 2; /* ordered. */
x = 3; /* not ordered: no dependency. */
r1 = y; /* not ordered: no dependency. */
```



store_dependent_release() / perObjectPreStoreFence()

- “Publish” for load_dependent_acquire() “subscribe”:

```
p->a = 1;      /* ordered. */  
p->b = 1;      /* ordered. */  
x = 1;        /* not ordered: no dependency. */  
y = 1;        /* not ordered: no dependency. */  
store_dependent_release(&pd, p);
```

- Expect most architectures will emit an explicit release fence



Dependency Ordering



Dependency Ordering

- Options:

- 1) Just say “no” to respecting dependency ordering
- 2) Respect only dynamic dependencies (Peter Dimov)
- 3) Respect only dependency chains within a single compilation unit that are headed by `load_dependent_acquire()`
- 4) Just say “no” to violating dependency ordering
- 5) Permit only specific classes of dependency ordering
- 6) Control dependency ordering based on compiler flag on per-compilation-unit basis
- 7) Generate code that both preserves and breaks dependency, using name-mangling to differentiate between them
- 8) Respect only those dependency chains that are explicitly marked by the programmer
- 9) **#8, but also provide function declaration/definition decorations that indicate that dependencies be respected**



Controlling Dependency Propagation

- **Propagate dependencies:**

```
dependence_propagate int f(dependence_propagate int *y)
{
    return y;
}
```

- **Propagate dependencies in:**

```
int f(dependence_propagate int *y)
{
    return *y;
}
```

- **Propagate dependencies out:**

```
dependence_propagate struct foo *f(void)
{
    return foo_head;
}
```

- **Don't propagate dependencies:**

```
template<T> T *kill_dependency_chain(T *y)
{
    return y;
}
```



Compiler Fence Operations



Compiler Fence Operations

- `void compiler_fence(void);`
 - Prohibit compiler from moving memory references across the fence
 - Similar to Linux kernel's `barrier()`
- `T *compiler_load_dependent_acquire(&x);`
- `void compiler_store_dependent_release(T *p, T *newval);`
 - Similar intent as `compiler_fence()`, but allows compiler more freedom to optimize in cases involving dependencies
 - Examples similar to those for `load_dependent_acquire()` and `store_dependent_release()`, but with communication between mainline and handler (as opposed to among CPUs).



Compiler Fence Operations: compiler_fence()

- Mainline code:

```
critical_section = 1;
compiler_fence();
do_something_critical();
compiler_fence();
critical_section = 0;
if (need_deferred) {
    block_handlers(); /* e.g., block signals. */
    non_critical_processing();
    need_deferred = 0;
    unblock_handlers();
}
```

- Handler code:

```
if (critical_section)
    need_deferred = 1;
else
    non_critical_processing();
```



Atomicity of Aligned Basic Types



Atomicity of Aligned Basic Types

- Much code assumes atomic loads/stores of small aligned basic types, for example:
 - ▶ char on 8-bit machines
 - ▶ char, short, and void* on 16-bit machines
 - ▶ char, short, long, void* on 32-bit machines
 - ▶ char, short, long, void* on 64-bit machines
- By “atomic loads/stores”, we mean:
 - ▶ Any load yields either the initial value or the value supplied to a prior store
- This is not practical in general for large variables or for abstract classes – but existing use is only for basic types



Atomicity of Aligned Basic Types

- Why not abstract classes?
 - ▶ Suppose abstract class A has subclasses S1, S2, and S3
 - ▶ Variable of type class A initially contains value of subclass S1
 - ▶ Two threads are concurrently assigning values of subclasses S2 and S3 (respectively) to the variable
 - ▶ A third thread is concurrently reading this same variable
 - ▶ The third thread could see the type change an arbitrary number of times!!!
- Therefore, programmers expecting atomicity from large/abstract variables must use atomics
- Basic types that fit into the machine word provide atomicity



Conclusions and Recommendations



Conclusions and Recommendations

- Provide explicit memory fence operations
 - ▶ Global and per-object
- Respect dependency ordering when so designated by the programmer
- Provide compiler-fence operations
 - ▶ Global and per-object
- Provide atomicity guarantees for small basic types
 - ▶ Different “profiles” for different machine sizes



Conclusions and Recommendations: APIs

- Memory Fences:
 - ▶ `void acquire_fence(void);`
 - ▶ `void release_fence(void);`
 - ▶ `void ordered_fence(void);`
 - ▶ `T *load_dependent_acquire(T *p);`
 - ▶ `void store_dependent_release(T *p, T *newval);`

- Tag Arguments and Return Values to Propagate Dependencies
 - ▶ `dependence_propagate`

- Compiler Fences:
 - ▶ `void compiler_fence(void);`
 - ▶ `T *compiler_load_dependent_acquire(T *p);`
 - ▶ `void compiler_store_dependent_release(T *p, T *newval);`



BACKUP



LINUX-KERNEL USE-CASE EXAMPLES



Use Case: RCU field

- **Insertion:**

```
p1->a = 1;  
smb_wmb();  
gp = p1;
```

- **do_something_with():**

```
rcu_read_lock();  
p = rcu_dereference(gp);  
x = p->a;  
rcu_read_unlock();
```

- **Can do the above in a loop to traverse a linked structure**



Use Case: RCU field-list

- **Insertion:**

```
p1->a = 1;  
q1->b = 1;  
p1->q = q1;  
smb_wmb();  
gp = p1;
```

- **do_something_with():**

```
rcu_read_lock();  
p = rcu_dereference(gp);  
x = p->q->b;  
rcu_read_unlock();
```

- **Can also do the above in a loop to traverse a linked structure**



Use Case: RCU refcnt

- Data structure:

```
struct cache {  
    struct cache *next;  
    int value;  
    atomic_t refcnt;  
} *head;
```



Use Case: RCU refcnt

- “Peek” function:

```
int peek(void)
{ /* peek function */
    struct cache *p;

    rcu_read_lock();
    p = rcu_dereference(head);
    if (p->value == 0) {
        atomic_inc(&p->refcnt);
        rcu_read_unlock();
        refill_cache(); /* too slow for RCU readers */
        rcu_read_lock();
        if (atomic_dec_and_test(&p->refcnt)) {
            rcu_read_unlock();
            cache_free(head, p);
            return 0;
        }
    }
    rcu_read_unlock();
    return p->value;
}
```



Use Case: RCU field-array

- Data structures:

```
struct mapped_to_object {  
    int data;  
};
```

```
struct bucket {  
    int size;  
    struct mapped_to_object *p[0];  
} *entries;
```



Use Case: RCU field-array

- **Lookup Function:**

```
int lookup(int id) /* lookup function */
{
    struct bucket *ep;
    struct mapped_to_object *q;
    int value;

    rcu_read_lock();
    ep = rcu_dereference(entries);
    q = rcu_dereference(ep[id]);
    if (q == NULL)
        value = -1;
    else
        value = q->data;
    rcu_read_unlock();
    return value;
}
```



Use Case: RCU lock

- Data structures:

```
struct mapped_to_object {  
    spinlock_t lock;  
    int data;  
};
```

```
struct bucket {  
    int size;  
    struct mapped_to_object *p[0];  
} *entries;
```



Use Case: RCU lock

■ Lookup Function:

```
/* lookup function */
struct mapped_to_object lookup_lock(int id) {
    struct bucket *ep;
    struct mapped_to_object *q;

    rcu_read_lock();
    ep = rcu_dereference(entries); /* table grow/shrink */
    q = rcu_dereference(ep[id]); /* entry add/delete */
    if (q == NULL) {
        rcu_read_unlock();
        return NULL;
    }
    spin_lock(&q->lock);
    rcu_read_unlock();
    return q;
}

/* release function */
void lookup_release(struct mapped_to_object *q) {
    spin_unlock(&q->lock);
}
```



DEPENDENCY EXAMPLES



Dependency Ordering: N2176 Example 1

- N2176 example code:

```
r1 = x.load_relaxed();  
r2 = *r1;
```

- Code based on option #9:

```
r1 = load_dependent_acquire(&x);  
r2 = *r1;
```

- Explicit `load_dependent_acquire()` forces dependency ordering to be respected



Dependency Ordering: N2176 Example 2

- N2176 example code:

```
r1 = x.load_relaxed();  
r3 = &a + r1 - r1;  
r2 = *r3;
```

- Could be optimized as follows:

```
r1 = x.load_relaxed();  
r3 = &a;  
r2 = *r3;
```

- Code based on option #9:

```
r1 = load_dependent_acquire(&x);  
r3 = &a + r1 - r1;  
r2 = *r3;
```

- Explicit `load_dependent_acquire()` prohibits this optimization

- ▶ Please note that the opportunity for this optimization might be very non-obvious from the source code



Dependency Ordering: N2176 Example 3

- N2176 example code:

```
r1 = x.load_relaxed();  
if (r1 == 0)  
    r2 = *r1;  
else  
    r2 = *(r1 + 1);
```

- Could be optimized to suppress HW dependency ordering:

```
r1 = x.load_relaxed();  
if (r1 == 0)  
    r3 = r1;  
else  
    r3 = r1 + 1;  
r2 = *r3;
```

- Code based on option #9:

- ▶ Use `load_dependent_acquire()` to force compiler to (a) emit memory fence or (b) create artificial dependency
- ▶ But don't know of any hardware that distinguishes between these two forms



Dependency Ordering: N2176 Example 3'

- N2176 example code email update:

```
if (load_dependent_acquire(&x))  
    ...  
else
```

```
    ...  
    y = 42 * x / 13;
```

- Could be optimized from data to control dependency:

```
if (load_dependent_acquire(&x)) {  
    ...  
    y = 3;  
} else {  
    ...  
    y = 0;  
}
```

- If compiled for machine w/out control dependency (e.g., ARM):
 - ▶ Prohibit optimization,
 - ▶ Emit a memory fence after the load of x, or
 - ▶ Emit artificial dependency recognized by the machine



Dependency Ordering: N2176 Example 4

- N2176 example code:

```
r1 = x.load_relaxed();  
if (r1)  
    r2 = y.a;  
else  
    r2 = y.a;
```

- Could be optimized to suppress HW dependency ordering:

```
r1 = x.load_relaxed();  
r2 = y.a;
```

- Code based on option #9:

```
r1 = load_dependent_acquire(&x);  
if (r1)  
    r2 = y.a;  
else  
    r2 = y.a;
```

- Compiler must preserve dependency, emit memory barrier, or create artificial dependency



Dependency Ordering: N2176 Example 5

- N2176 example code:

```
r1 = load_dependent_acquire(&x);  
if (r1)  
    f(&y);  
else  
    g(&y);
```

- If f() and g() are in separate compilation unit, what to do?
 - ▶ Programmer must also decorate declaration and definition of f() and g() with indication that dependencies must be preserved (per argument and on return value)
- Options for decorating:
 - ▶ gcc attributes
 - ▶ template class for argument that includes appropriate barriers
 - ▶ new storage class similar to volatile, but weaker
- Undecorated unit function/template breaks the chain



Dependency Ordering: N2176 Example 6

- N2176 example code:

```
r2 = load_dependent_acquire(&x);  
r3 = r2 -> a;
```

- `load_dependent_acquire()` prohibits the following optimization:

```
r2 = load_dependent_acquire(&x);  
r3 = r1 -> a;  
if (r1 != r2) r3 = r2 -> a;
```



Dependency Ordering: N2176 Example 7

- N2176 example code:

```
r1 = load_dependent_acquire(&x);  
r2 = a[r1 -> index % a_size];
```

- Suppose that `a_size` is zero. Then `load_dependent_acquire()` prohibits the following optimization:

```
r1 = load_dependent_acquire(&x);  
r2 = a[0];
```

- Please note that the opportunity for this optimization might not be at all obvious from the source code

