A Methodology for Implementing Highly Concurrent Data Objects

Written by Maurice Herlihy
Presented by Caylee Hogg

October 12, 2011

Defining Our Terms

- Concurrent object

Defining Our Terms

- Concurrent object
- Non-blocking

Defining Our Terms

- Concurrent object
- Non-blocking
- Wait-free

Defining Our Terms

- Concurrent object
- Non-blocking
- Wait-free
- Linearizable

Our Problem

- Concurrent code is hard to write

Our Problem

- Concurrent code is hard to write
- Concurrent code is harder to *understand*

Our Problem

- Concurrent code is hard to write
- Concurrent code is harder to *understand*
- Want to reason about concurrent code as easily as sequential

Our Problem

- Concurrent code is hard to write
- Concurrent code is harder to *understand*
- Want to reason about concurrent code as easily as sequential
- Want to make it easy to write correct concurrent code, i.e. linearizable

Our Problem

- Concurrent code is hard to write
- Concurrent code is harder to *understand*
- Want to reason about concurrent code as easily as sequential
- Want to make it easy to write correct concurrent code, i.e. linearizable
- Don't want the performance to be too painful!

The Plan

- Write our sequential operations first

The Plan

- Write our sequential operations first
- Transform the sequential code to concurrent code in principled, possibly automated, way

The Plan

- Write our sequential operations first
- Transform the sequential code to concurrent code in principled, possibly automated, way
- Use `load_linked` and `store_conditional` for non-blocking implementation

The Plan

- Write our sequential operations first
- Transform the sequential code to concurrent code in principled, possibly automated, way
- Use `load_linked` and `store_conditional` for non-blocking implementation
- Naturally linearizable because only interaction point is `store_conditional`

The Plan

- Write our sequential operations first
- Transform the sequential code to concurrent code in principled, possibly automated, way
- Use `load_linked` and `store_conditional` for non-blocking implementation
- Naturally linearizable because only interaction point is `store_conditional`
- Can further transform non-blocking algorithms to wait-free algorithms

Sequential Qualifiers

- Not just any sequential algorithm can be transformed

Sequential Qualifiers

- Not just any sequential algorithm can be transformed
- Sequential algorithm must be total

Sequential Qualifiers

- Not just any sequential algorithm can be transformed
- Sequential algorithm must be total
- Total means well-defined on all valid states of the data

Sequential Qualifiers

- Not just any sequential algorithm can be transformed
- Sequential algorithm must be total
- Total means well-defined on all valid states of the data
- Only side-effects can be modifying the blocks it was passed

Sequential Qualifiers

- Not just any sequential algorithm can be transformed
- Sequential algorithm must be total
- Total means well-defined on all valid states of the data
- Only side-effects can be modifying the blocks it was passed
- Why?

Loading & Storing

- `load_linked` copies the data into a provided variable

Loading & Storing

- `load_linked` copies the data into a provided variable
- `store_conditional` only successfully writes if no one else has written

Loading & Storing

- `load_linked` copies the data into a provided variable
- `store_conditional` only successfully writes if no one else has written
- If no one else has written, does that mean it will succeed?

Loading & Storing

- `load_linked` copies the data into a provided variable
- `store_conditional` only successfully writes if no one else has written
- If no one else has written, does that mean it will succeed?
- Can have "spurious" failures

Spurious Failures

- In real implementations, failures occur for reasons invisible to the semantics

Spurious Failures

- In real implementations, failures occur for reasons invisible to the semantics
- Modifying other portion of the cache line
- Context switching
- Execution of another `load_linked` on a different location

Differences From CAS?

- Stronger conditional

Differences From CAS?

- Stronger conditional
- No `ABA` problem is possible (Why?)

Differences From CAS?

- Stronger conditional
- No `ABA` problem is possible (Why?)
- Many ways to spuriously fail

Loading & Storing

```
int double_it(int *n){
  while(1){
    int local = load_linked(n);
    local = local * 2;
    if (store_conditional(n,local)) break;
}
```

Small Objects

- Fits into a contiguous block of memory

Small Objects

- Fits into a contiguous block of memory
- Can be copied efficiently
- Memory management is simpler
- For n threads and m objects, how many blocks do we need?

Running Example: Counter

```
struct counter_type {
  int counts;
  int value;
};

int counter_inc(struct counter_type* c) {
  c->counts = c->counts + 1;
  c->value = c->value + 1;
  return c->value;
}

int counter_dec(struct counter_type* c) {
  c->counts = c->counts + 1;
  c->value = c->value - 1;
  return c->value;
}
```

Making It Concurrent: Take 1

- Does the previous code fit our sequential criterion?
- Write wrapper functions that use `load_linked` & `store_conditional`
- Each process has its own local sandbox counterxs

Making It Concurrent: Take 1

```
static counter_type *new_counter; //subtle deceit here

int Counter_inc(struct counter_type **C){
  struct counter_type *old_counter;
  int result;
  while (1) {
    old_counter = load_linked(C);
    copy(old_counter,new_counter);
    result = counter_inc(new_counter);
    if (store_conditional(C,new_counter)) break;
  }
  new_counter=old_counter;
  return result;
}
```

Making it Concurrent: Take 2

- Still a potential problem. Do you see it?

Making it Concurrent: Take 2

- Still a potential problem. Do you see it?
- Sequential code might observe a state that breaks invariants of data

Making it Concurrent: Take 2

- Still a potential problem. Do you see it?
- Sequential code might observe a state that breaks invariants of data
- Introduce `check` array in order to pre-emptively remove possibility of viewing a bad state

Making it Concurrent: Take 2

- Still a potential problem. Do you see it?
- Sequential code might observe a state that breaks invariants of data
- Introduce `check` array in order to pre-emptively remove possibility of viewing a bad state
- Wait, aren't we solving the same problem twice?

Making it Concurrent: Take 2

- Still a potential problem. Do you see it?
- Sequential code might observe a state that breaks invariants of data
- Introduce `check` array in order to pre-emptively remove possibility of viewing a bad state
- Wait, aren't we solving the same problem twice?
- `store_conditional` would fail in either case, but `check` only stops inconsistencies not races

Making It Concurrent: Take 2

```
struct Counter_type {
  struct counter_type* version;
  int check[2];
};

static Counter_type *new_counter;

int Counter_inc(struct Counter_type** C){
  Counter_type *old_counter;
  struct counter_type *old_version, *new_version;
  int result;

  while (1) {
    old_counter = load_linked(C);
    old_version = &old_counter->version;
    new_version = &new_counter->version;
    new_counter->check[0] = new_counter->check[1]+1;
    first = old_counter->check[1];
    copy(old_version,new_version);
    last = old_counter->check[0];
    if (first == last) {
      result = counter_inc(new_version);
      new_counter->check[1]++;
      if (store_conditional(C,new_counter)) break;
    }
  }
  new_counter = old_counter;
  return result;
}
```

Making It Wait-Free

- Some operations are slower than others

Making It Wait-Free

- Some operations are slower than others
- Starvation is bad!

Making It Wait-Free

- Some operations are slower than others
- Starvation is bad!
- Prevent starvation by sharing resources

Making It Wait-Free

- Some operations are slower than others
- Starvation is bad!
- Prevent starvation by sharing resources
- “From each according to their ability, to each according to their need”

Work-sharing

- Reify the operations needed
- Have threads declare what they're attempting to accomplish
- Other processes declare their own work and if there's other work outstanding, they finish that instead
- Assuming no spurious failures, every process will have finished in two attempts

New data structures

```
#define MAX_PROCS = 10000;
#define INC_CODE = 0;
#define DEC_CODE = 1;

struct res_type {
  bool toggle;
  int value;
}

struct Counter_type {
  struct counter_type* version;
  int check[2];
  struct res_type res_types[MAX_PROCS];
};

struct inv_type {
  bool toggle;
  int op_name;
}
```

Apply operation

```
void apply(struct inv_type announce[MAX_PROCS], struct Counter_type *object){
  int i;
  for (i = 0; i < MAX_PROCS; i++){
    if(announce[i].toggle != object->res_types[i].toggle){
      switch (announce[i].op_name) {
      case INC_CODE:
        object->res_types[i].value =
          counter_inc(&object->version);
        break;
      case DEC_CODE:
        object->res_types[i].value = counter_dec(&object->version);
        break;
      default:
        fprintf(stderr, "Unknown operation code\n");
        exit(1);
      };
      object->res_types[i].toggle = announce.toggle[i];
    }
  }
}
```

A Wait-Free Inc

```
static struct Counter_type *new_counter;
static struct inv_type announce[MAX_PROCS];
static int P;

int Counter_inc(struct Counter_type **C){
  Counter_type *old_counter;
  counter_type *old_version, *new_version;
  int i, result, new_toggle;
  announce[P].op_name = INC_CODE;
  new_toggle = announce[P].toggle = !announce[P].toggle;
  while ((*C)->response[P].toggle != new_toggle ||
         (*C)->response[P].toggle != new_toggle){
    old_counter = load_linked(C);
    old_version = &old_counter->version;
    new_version = &new_queue->version;
    copy(new_version,old_version);
    result = apply(announce, new_counter);
    if(store_conditional(C,new_counter)) break;
    }
  new_counter = old_counter;
  return result;
}
```

Large Objects

- Anything that needs multiple blocks

Large Objects

- Anything that needs multiple blocks
  - Linked lists

Large Objects

- Anything that needs multiple blocks
  - Linked lists
  - Trees

Large Objects

- Anything that needs multiple blocks
  - Linked lists
  - Trees
  - If you're connecting data together with pointers, it's "large"

Large Objects

- Anything that needs multiple blocks
  - Linked lists
  - Trees
  - If you're connecting data together with pointers, it's "large"
- Key point: you only copy the pieces of the structure you're working on

Large Objects

- Anything that needs multiple blocks
  - Linked lists
  - Trees
  - If you're connecting data together with pointers, it's "large"
- Key point: you only copy the pieces of the structure you're working on
  - Pushing or popping on a linked-list stack shares all but one block with original

Managing Large Objects

- Need explicit pool management

Managing Large Objects

- Need explicit pool management
- Free what you copy, allocate what you create

Managing Large Objects

- Need explicit pool management
- Free what you copy, allocate what you create
- Same basic techniques, just slightly different API

Managing Large Objects

- Need explicit pool management
- Free what you copy, allocate what you create
- Same basic techniques, just slightly different API
- Use `set_alloc` & `set_free`

Performance

- Non-blocking:

Performance

- Non-blocking:
  - The naïve approach has serious memory contention
  - Exponential backoff improves things considerably
  - Experiments indicate within a factor of two of spin-lock implementation

Performance

- Non-blocking:
  - The naïve approach has serious memory contention
  - Exponential backoff improves things considerably
  - Experiments indicate within a factor of two of spin-lock implementation
- Wait-free:

Performance

- Non-blocking:
  - The naïve approach has serious memory contention
  - Exponential backoff improves things considerably
  - Experiments indicate within a factor of two of spin-lock implementation
- Wait-free:
  - Increased sharing between processes means worse performance
  - At *best* a factor of two worse than the non-blocking implementation
  - Use it when you need its safety properties

Our Goals

- Reason about concurrent code as easily as sequential

Our Goals

- Reason about concurrent code as easily as sequential
- Write manifestly linearizable programs

Our Goals

- Reason about concurrent code as easily as sequential
- Write manifestly linearizable programs
- Make it not hurt performance...too much

How Did We Do?

- Our transformed programs are quite clearly linearizable

How Did We Do?

- Our transformed programs are quite clearly linearizable
- What about our reasoning?

How Did We Do?

- Our transformed programs are quite clearly linearizable
- What about our reasoning?
- Performance is *okay*

A Critique of (Im)Pure Reason

- Formal vs. informal reasoning

A Critique of (Im)Pure Reason

- Formal vs. informal reasoning
- A uniform transformation is easier *informally*

A Critique of (Im)Pure Reason

- Formal vs. informal reasoning
- A uniform transformation is easier *informally*
- "It works fine for every program we've tried so far, so by induction..."

A Critique of (Im)Pure Reason

- Formal vs. informal reasoning
- A uniform transformation is easier *informally*
- "It works fine for every program we've tried so far, so by induction..."
- A uniform transformation isn't *necessarily* easier formally

A Critique of (Im)Pure Reason

- Formal vs. informal reasoning
- A uniform transformation is easier *informally*
- "It works fine for every program we've tried so far, so by induction..."
- A uniform transformation isn't *necessarily* easier formally
- In a logic for concurrent systems, one cannot always reason modularly

Conclusions

- “I’m making a note here: huge success”

Conclusions

- “I’m making a note here: huge success”
- Partially achieved goals

Conclusions

- "I'm making a note here: huge success"
- Partially achieved goals
  - Performance isn't too bad for non-blocking
  - Easier for informal reasoning
  - Transformation simple enough to even be automated