6.172
Performance
Engineering of
Software Systems

LECTURE 16
Synchronizing without Locks

Charles E. Leiserson

November 4, 2010

OUTLINE

- Memory Consistency
- Lock-Free Protocols
- The ABA Problem
- Reducer Hyperobjects

OUTLINE

- Memory Consistency
- Lock-Free Protocols
- The ABA Problem
- Reducer Hyperobjects

Memory Models

Initially, a = b = 0.

Processor 0

```
mov 1, a      ;Store
mov b, %ebx   ;Load
```

Processor 1

```
mov 1, b      ;Store
mov a, %eax   ;Load
```

Q. What are the final possible values of %eax and %ebx after both processors have executed?

A. It depends on the *memory model*: how memory operations behave in the parallel computer system.

Sequential Consistency

"[T]he result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

— *Leslie Lamport [1979]*

- The sequence of instructions as defined by a processor's program are *interleaved* with the corresponding sequences defined by the other processors's programs to produce a global *linear order* of all instructions.
- A load instruction receives the value stored to that address by the most recent store instruction that precedes the load, according to the linear order.
- The hardware can do whatever it wants, but for the execution to be sequentially consistent, it must *appear* as if loads and stores obey some global linear order.

Example

Initially, a = b = 0.

Processor 0

```
1  mov 1, a      ;Store
2  mov b, %ebx   ;Load
```

Processor 1

```
3  mov 1, b      ;Store
4  mov a, %eax   ;Load
```

	Interleavings					
	1	1	1	3	3	3
	2	3	3	1	1	4
	3	2	4	2	4	1
	4	4	2	4	2	2
%eax	1	1	1	1	1	0
%ebx	0	1	1	1	1	1

Sequential consistency implies that no execution ends with %eax= %ebx = 0.

Mutual-Exclusion Problem

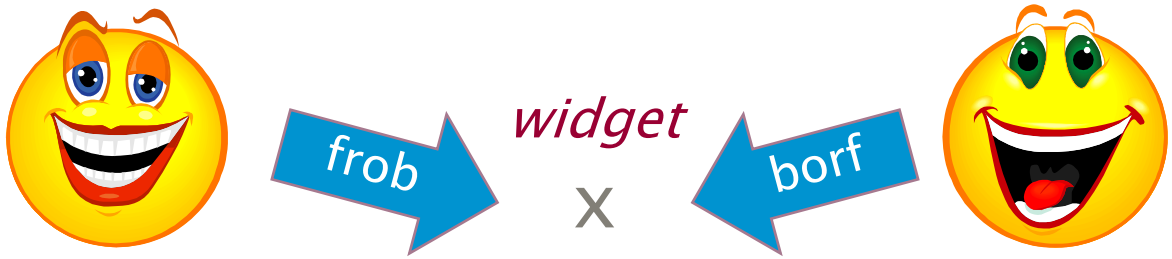
Most implementations of mutual exclusion employ an *atomic read-modify-write* instruction or the equivalent, usually to implement a lock:

- e.g., xchg, test-and-set, compare-and-swap, load-linked-store-conditional.

Q. Can mutual exclusion be implemented with only atomic loads and stores?

A. Yes, Dekker and Dijkstra showed that it can as long as the computer system is sequentially consistent.

Peterson's Algorithm




```
widget x; //protected variable
bool she_wants(false);
bool he_wants(false);
enum theirs {hers, his} turn;
```

Her Thread

```
she_wants = true;
turn = his;
while(he_wants && turn==his);
frob(x); //critical section
she_wants = false;
```

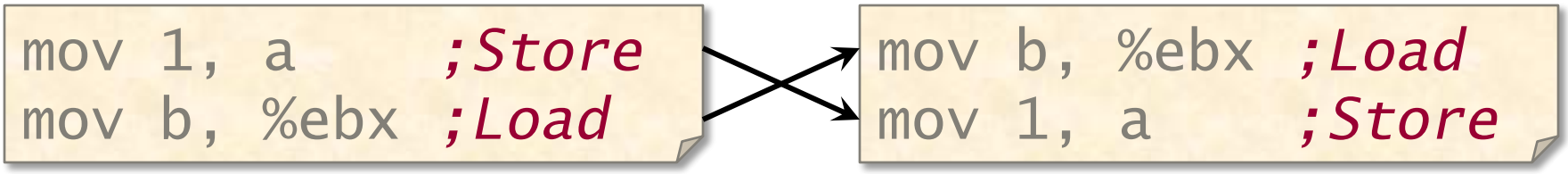
His Thread

```
he_wants = true;
turn = hers;
while(she_wants && turn==hers);
borf(x); //critical section
he_wants = false;
```

Memory Models Today

- No modern-day processor implements sequential consistency.
- All implement some form of relaxed consistency.
- Hardware actively reorders instructions.
- Compilers may reorder instructions, too.

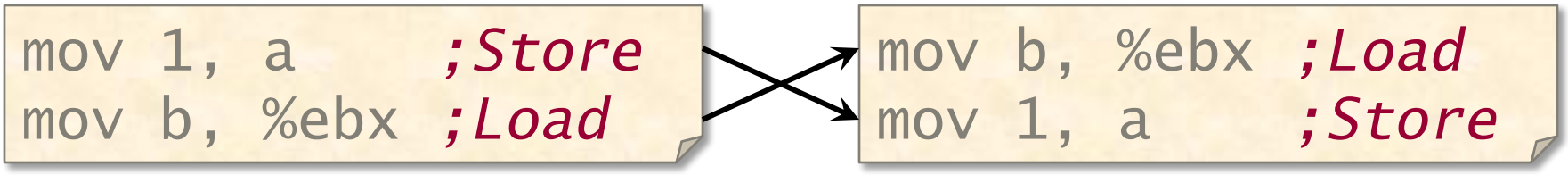
Instruction Reordering



Q. Why might the hardware or compiler decide to reorder these instructions?

A. To obtain higher performance by covering load latency — ***instruction-level parallelism***.

Instruction Reordering

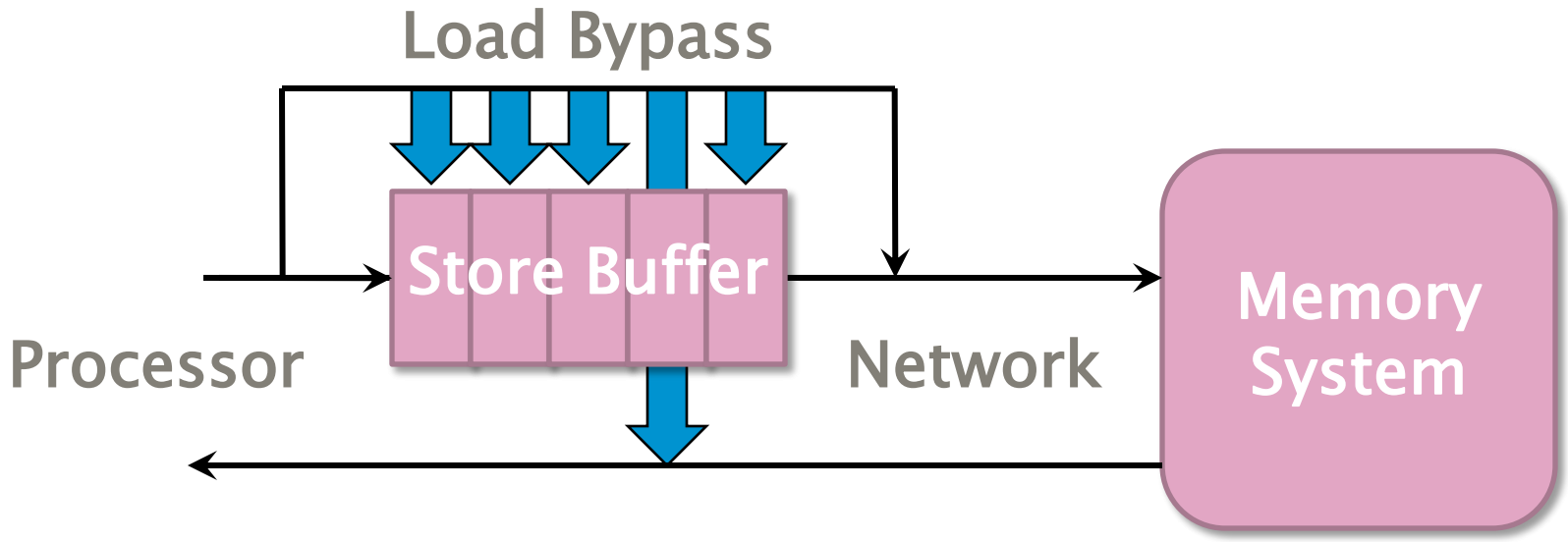


Q. When is it safe for the hardware or compiler to perform this reordering?

A. When a ≠ b.

A′. And there's no concurrency.

Hardware Reordering



- The processor can issue stores faster than the network can handle them ⇒ store buffer.
- Since a load may stall the processor until it is satisfied, loads take priority, bypassing the store buffer.
- If a load address matches an address in the store buffer, the store buffer returns the result.
- Thus, a load can bypass a store to a different address.

x86 Memory Consistency

1. Loads are *not* reordered with loads.
2. Stores are *not* reordered with stores.
3. Stores are *not* reordered with prior loads.
4. A load *may* be reordered with a prior store to a different location but *not* with a prior store to the same location.
5. Loads and stores are *not* reordered with lock instructions.
6. Stores to the same location respect a global total order.
7. Lock instructions respect a global total order.
8. Memory ordering preserves transitive visibility ("causality").

Impact of Reordering

Processor 0

```
1  mov 1, a      ;Store
2  mov b, %ebx   ;Load
```

Processor 1

```
3  mov 1, b      ;Store
4  mov a, %eax   ;Load
```

Impact of Reordering

Processor 0

```
1  mov 1, a      ;Store
2  mov b, %ebx   ;Load
```

Processor 1

```
3  mov 1, b      ;Store
4  mov a, %eax   ;Load
```

```
2  mov b, %ebx   ;Load
1  mov 1, a      ;Store
```

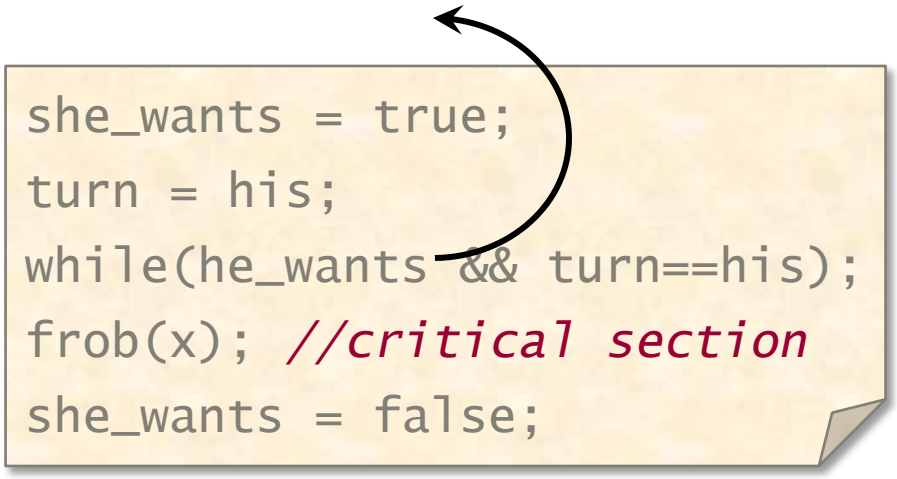
```
4  mov a, %eax   ;Load
3  mov 1, b      ;Store
```

The ordering ⟨2, 4, 1, 3⟩ produces %eax = %ebx = 0.

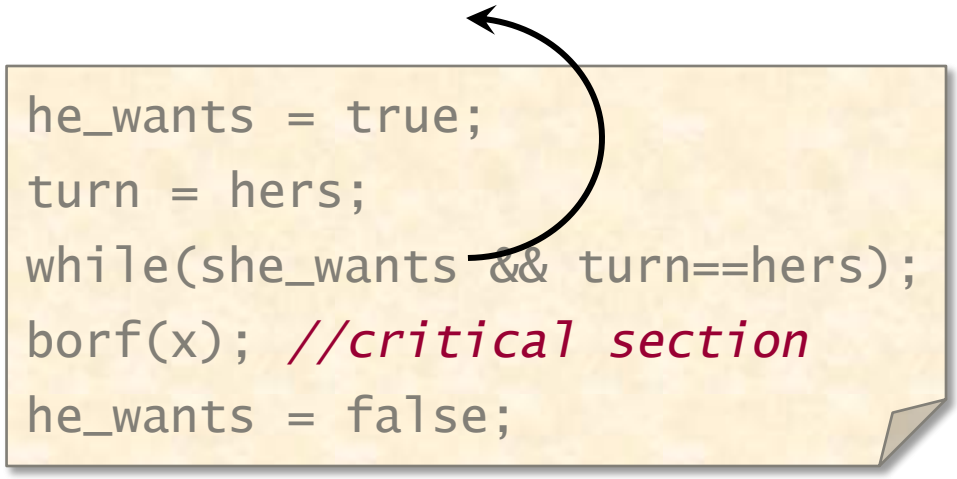
Instruction reordering violates sequential consistency!

Further Impact of Reordering

Peterson's algorithm revisited



```
she_wants = true;
turn = his;
while(he_wants && turn==his);
frob(x); //critical section
she_wants = false;
```



```
he_wants = true;
turn = hers;
while(she_wants && turn==hers);
borf(x); //critical section
he_wants = false;
```

- The loads of `he_wants`/`she_wants` can be reordered before the stores of `she_wants`/`he_wants`.
- Both threads might enter their critical sections simultaneously!

Memory Fences

- A *memory fence* (or *memory barrier*) is a hardware action that enforces an ordering constraint between the instructions before and after the fence.
- A memory fence can be issued explicitly as an instruction (x86: mfence) or be performed implicitly by locking, compare-and-swap, and other synchronizing instructions.
- The gcc and Intel compilers implement a memory fence via the built-in function `__sync_synchronize()`.*
- The typical cost of a memory fence is comparable to that of an L2-cache access.

---

*See <http://gcc.gnu.org/onlinedocs/gcc-4.3.4/gcc/Atomic-Builtins.html#Atomic-Builtins>. Some versions of gcc contain a bug which causes the mfence instruction to be omitted, however, but there is a patch: <http://gcc.gnu.org/viewcvs/branches/gcc-4_3-branch/gcc/config/i386/sse.md?r1=142310&r2=142309&pathrev=142310>.

Fixing the Reordering Error

Peterson's algorithm with memory fences

```
she_wants = true;
turn = his;
__sync_synchronize();
while(he_wants && turn==his);
frob(x); //critical section
she_wants = false;
```

```
he_wants = true;
turn = hers;
__sync_synchronize();
while(she_wants && turn==hers);
borf(x); //critical section
he_wants = false;
```

Memory fences can restore consistency.

OUTLINE

- Memory Consistency
- Lock-Free Protocols
- The ABA Problem
- Reducer Hyperobjects

Recall: Summing Problem

```
int compute(const X& v);
int main()
{
    const std::size_t n = 1000000;
    extern X myArray[n];
    // ...

    int result = 0;
    for (std::size_t i = 0; i < n; ++i)
    {
        result += compute(myArray[i]);
    }
    std::cout << "The result is: "
              << result
              << std::endl;
    return 0;
}
```

Summing Example in Cilk++

```
int compute(const X& v);
int main()
{
    const std::size_t n = 1000000;
    extern X myArray[n];
    // ...

    int result = 0;
    cilk_for (std::size_t i = 0; i < n; ++i)
    {
        result += compute(myArray[i]);
    }
    std::cout << "The result is: "
              << result
              << std::endl;
    return 0;
}
```

Race!

Mutex for the Summing Problem

```
int result = 0;
mutex L;
cilk_for (std::size_t i = 0; i < n; ++i)
{
    int temp = compute(myArray[i]);
    L.lock();
    result += temp;
    L.unlock();
}
```

Contention, yes, but it may not be significant if `compute(myArray[i])` takes sufficiently long. Still, in a multiprogrammed setting, there may be other problems….

Mutex for the Summing Problem

```
int result = 0;
mutex L;
cilk_for (std::size_t i = 0; i < n; ++i)
{
    int temp = compute(myArray[i]);
    L.lock();
    result += temp;
    L.unlock();
}
```

Q. What happens if the operating system swaps out a loop iteration just after it acquires the mutex?

A. All other loop iterations must wait.

Compare-and-Swap

Compare-and-swap is provided by the `cmpxchg` instruction on x86. The gcc and Intel compilers implement compare-and-swap via the built-in function `__sync_bool_compare_and_swap()` which operates on values of type `int`, `long`, `long long`, and their unsigned counterparts.*

Implementation logic

```
bool
__sync_bool_compare_and_swap (T *x, T old, T new) {
    if (*x == old) { *x = new; return 1; }
    return 0;
}
```

Executes atomically.

*See <http://gcc.gnu.org/onlinedocs/gcc-4.3.4/gcc/Atomic-Builtins.html#Atomic-Builtins>.

CAS for Summing

```
int result = 0;
cilk_for (std::size_t i = 0; i < n; ++i)
{
    temp = compute(myArray[i]);
    do {
        int old = result;
        int new = result + temp;
    } while ( !__sync_bool_compare_and_swap
                (&result, old, new) );
}
```

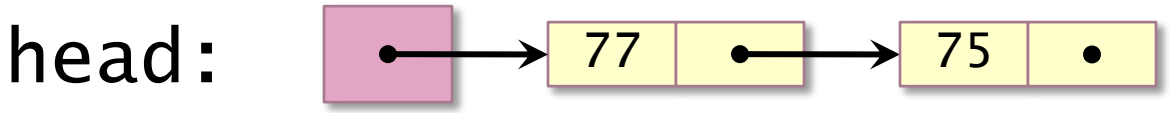
Q. What happens if the operating system swaps out a loop iteration?

A. No other loop iterations need wait.

Lock-Free Stack

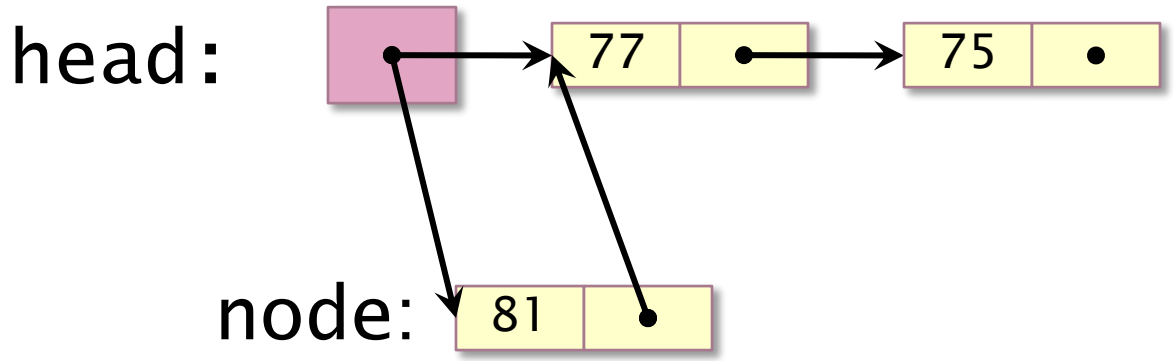
```
struct Node {
    Node* next;
    int data;
};

class Stack {
  private:
    Node* head;
          ⋮
```



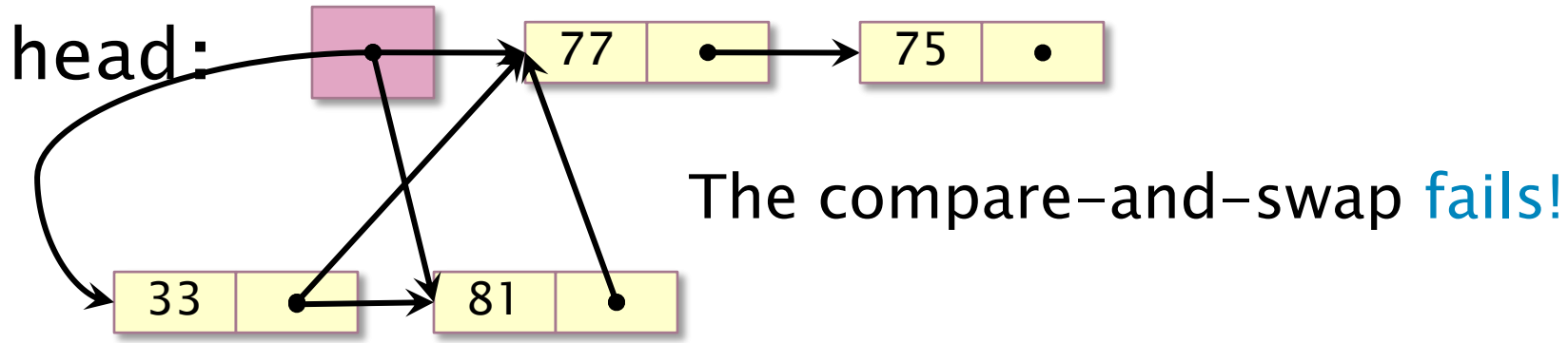
Lock-Free Push

```
public:
  void push(Node* node) {
       do {
            node->next = head;
       } while (!__sync_bool_compare_and_swap
                          (&head, node->next, node));
  }
                                  ⋮
```



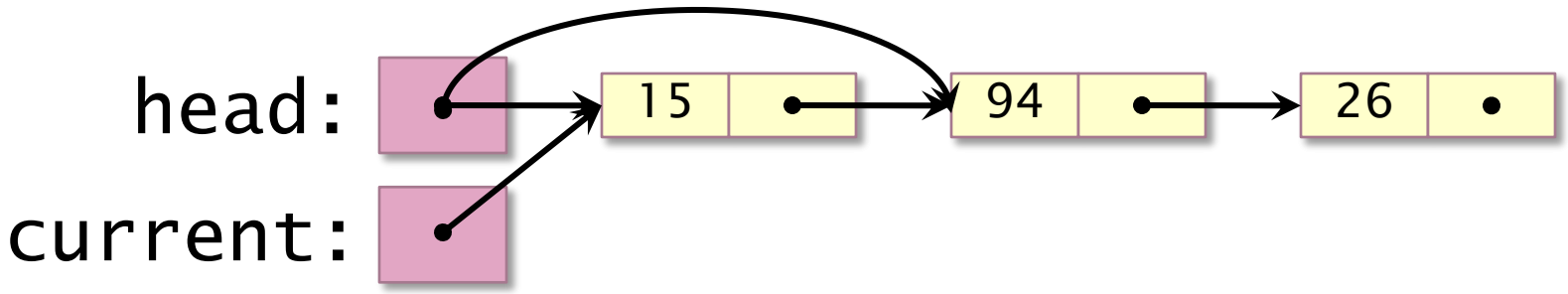
Lock-Free Push with Contention

```
public:
  void push(Node* node) {
       do {
            node->next = head;
       } while (!__sync_bool_compare_and_swap
                          (&head, node->next, node));
  }
                                  ⋮
```



Lock-Free Pop

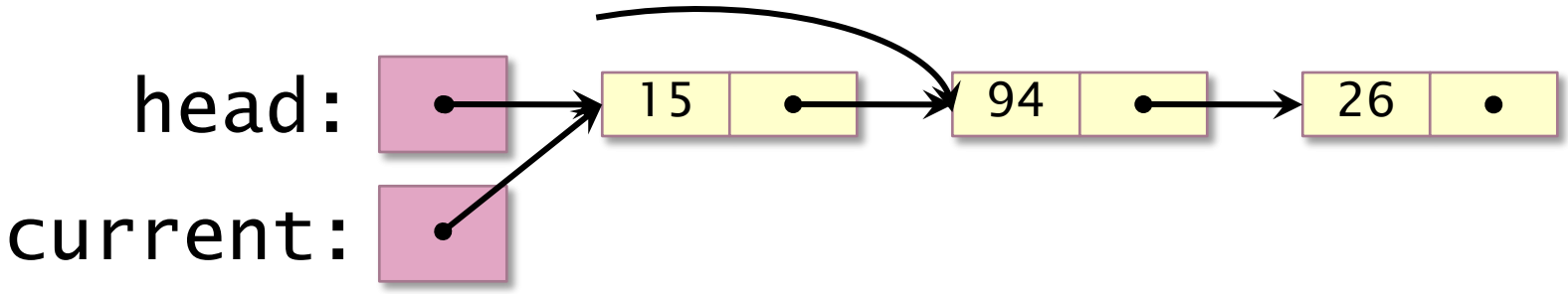
```
  Node* pop() {
      Node* current = head;
      while(current) {
          if(__sync_bool_compare_and_swap
                (&head,
                 current,
                 current->next)) break;
          current = head;
      }
      return current;
  }
}
```



OUTLINE

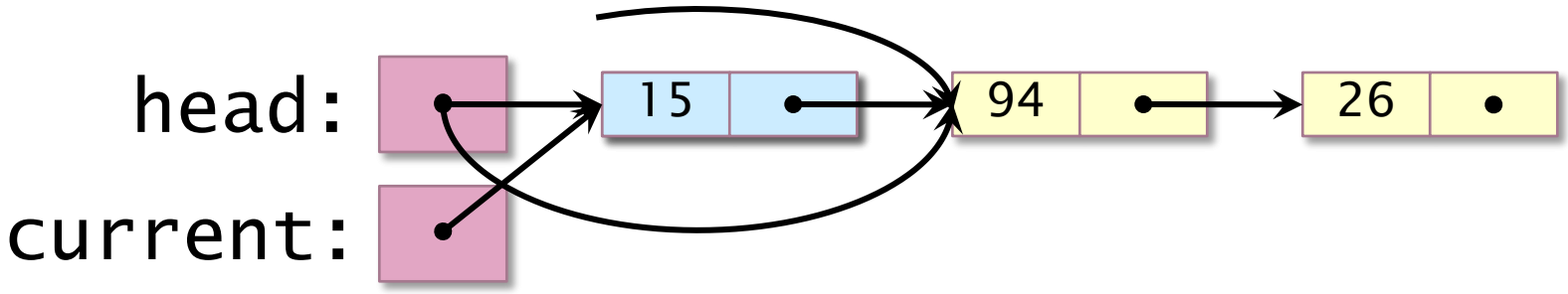
- Memory Consistency
- Lock-Free Protocols
- The ABA Problem
- Reducer Hyperobjects

ABA Problem



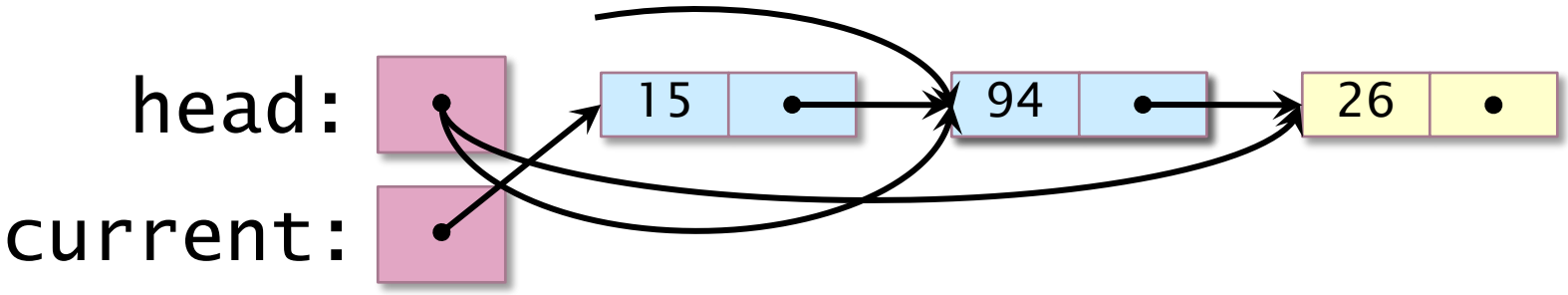
1. Thread 1 begins to pop `15`, but stalls after reading `current->next`.

ABA Problem



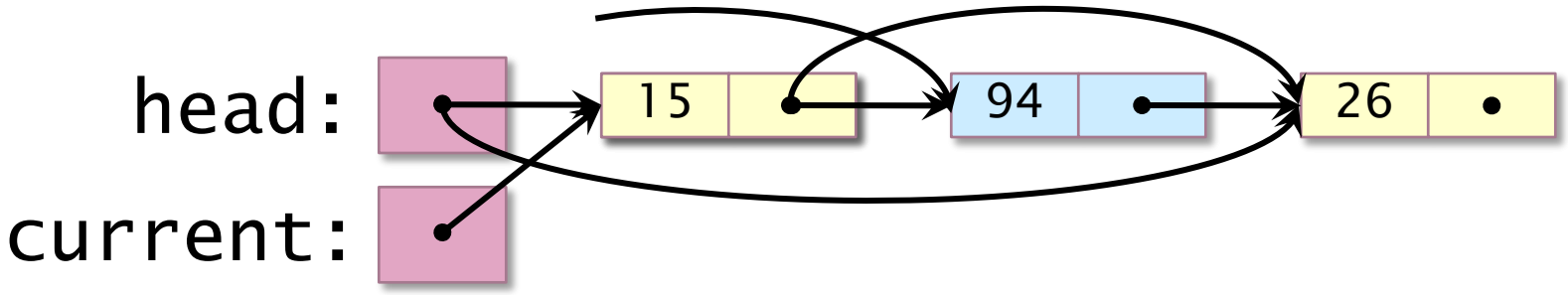
1. Thread 1 begins to pop 15, but stalls after reading `current->next`.
2. Thread 2 pops 15.

ABA Problem



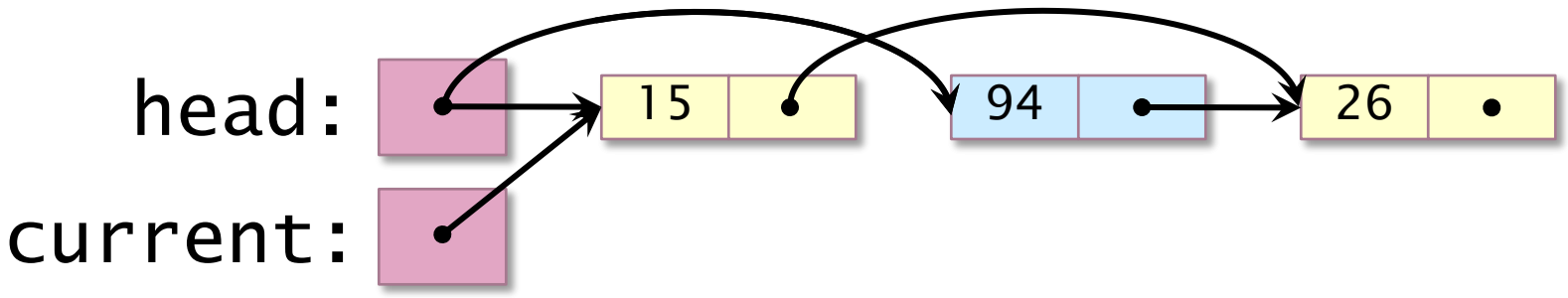
1. Thread 1 begins to pop 15, but stalls after reading `current->next`.
2. Thread 2 pops 15.
3. Thread 2 pops 94.

ABA Problem



1. Thread 1 begins to pop 15, but stalls after reading `current->next`.
2. Thread 2 pops 15.
3. Thread 2 pops 94.
4. Thread 2 pushes 15 back on.

ABA Problem



1. Thread 1 begins to pop 15, but stalls after reading `current->next`.
2. Thread 2 pops 15.
3. Thread 2 pops 94.
4. Thread 2 pushes 15 back on.
5. Thread 1 resumes, and the compare-and-swap completes, removing 15, but putting the garbage 94 back on the list.

Solution to ABA

Versioning

- Pack a version number with each pointer in the same atomically updatable word.
- Increment the version number every time the pointer is changed.
- Compare-and-swap both the pointer and the version number as a single atomic operation.

Issue: Version numbers may need to be very large.

As an alternative to compare-and-swap, some architectures feature a *load-linked, store conditional* instruction.

OUTLINE

- Memory Models
- Lock–Free Synchronization
- The ABA Problem
- **Reducer Hyperobjects**

Recall: Summing Problem

```
int compute(const X& v);
int main()
{
    const std::size_t n = 1000000;
    extern X myArray[n];
    // ...

    int result = 0;
    for (std::size_t i = 0; i < n; ++i)
    {
        result += compute(myArray[i]);
    }
    std::cout << "The result is: "
              << result
              << std::endl;
    return 0;
}
```

Reducer Solution

```
int compute(const X& v);
int main()
{
    const std::size_t ARRAY_SIZE = 1000000;
    extern X myArray[ARRAY_SIZE];
    // ...

    cilk::reducer_opadd<int> result;
    cilk_for (std::size_t i = 0; i < ARRAY_SIZE; ++i)
    {
        result += compute(myArray[i]);
    }
    std::cout << "The result is: "
              << result.get_value()
              << std::endl;
    return 0;
}
```

Reducer Solution

```
int com
int mai
{
    const std.:size_t ARRAY_SIZE = 1
    extern X myArray[ARRAY_SIZE];
    // ...

    cilk::reducer_opadd<int> result;
    cilk_for (std::size_t i = 0; i < ARRAY_SIZE; ++i)
    {
        result += compute(myArray[i]);
    }
    std::cout << "The result is: "
              << result.get_value()
              << std::endl;
    return 0;
}
```

Declare result to be a *summing reducer* over int.

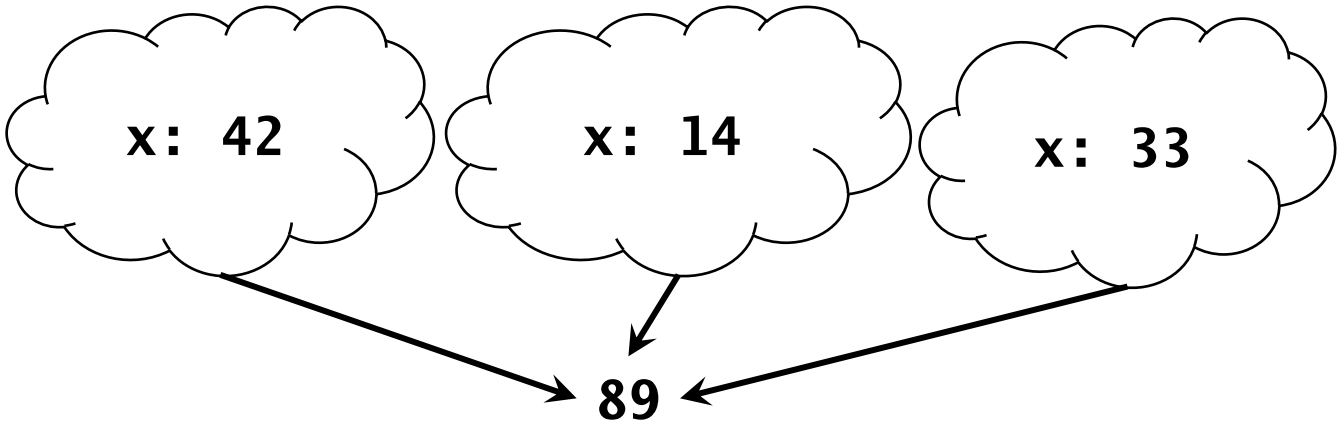
Updates are resolved automatically without races or contention.

At the end, the underlying int value can be extracted.

Reducers

- A variable x can be declared as a ***reducer*** over an *associative* operation, such as addition, multiplication, logical AND, list concatenation, etc.
- Strands can update x as if it were an ordinary nonlocal variable, but x is, in fact, maintained as a collection of different copies, called ***views***.
- The Cilk++ runtime system coordinates the views and combines them when appropriate.
- When only one view of x remains, the underlying value is stable and can be extracted.

Example: summing reducer



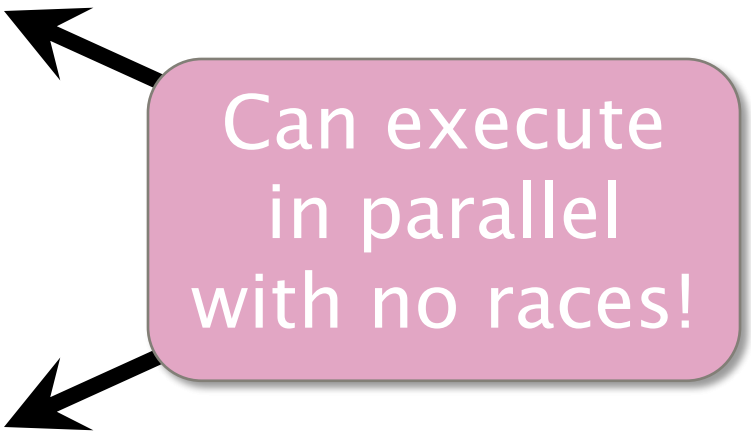
Conceptual Behavior

original

```
x = 0;
x += 3;
x++;
x += 4;
x++;
x += 5;
x += 9;
x -= 2;
x += 6;
x += 5;
```

equivalent

```
x1 = 0;
x1 += 3;
x1++;
x1 += 4;
x1++;
x1 += 5;
x2 = 0;
x2 += 9;
x2 -= 2;
x2 += 6;
x2 += 5;
x = x1 + x2;
```



If you don't "look" at the intermediate values, the result is ***determinate***, because addition is ***associative***.

Conceptual Behavior

original

```
x = 0;
x += 3;
x++;
x += 4;
x++;
x += 5;
x += 9;
x -= 2;
x += 6;
x += 5;
```

equivalent

```
x1 = 0;
x1 += 3;
x1++;
x1 += 4;
x1++;
x1 += 5;
x2 = 0;
x2 += 9;
x2 -= 2;
x2 += 6;
x2 += 5;
x = x1 + x2;
```

equivalent

```
x1 = 0;
x1 += 3;
x1++;
x2 = 0;
x2 += 4;
x2++;
x2 += 5;
x2 += 9;
x2 -= 2;
x2 += 6;
x2 += 5;
x = x1 + x2;
```

If you don't "look" at the intermediate values, the result is ***determinate***, because addition is ***associative***.

Related Work

- *OpenMP's reduction construct*
  - Tied to parallel for loop.
- *TBB's parallel reduce template*
  - Tied to loop construct.
- *Data-parallel (APL, NESL, ZPL, etc.) reduction*
  - Tied to the vector operation.
- *Google's MapReduce*
  - Tied to the map function.

In contrast, Cilk++ reducers are not tied to any control or data structure. They can be named anywhere (globally, passed as parameters, stored in data structures, etc.). Wherever and whenever they are dereferenced, they produce the local view.

Algebraic Framework

Definition. A ***monoid*** is a triple $(T, \otimes, e)$, where

- T is a set,
- $\otimes$ is an associative binary operator on elements of T,
- $e \in T$ is an identity element for $\otimes$.

Associative

$$a \otimes (b \otimes c) = (a \otimes b) \otimes c$$

Identity

$$a \otimes e = e \otimes a = a$$

Examples:

- $(\mathbb{Z}, +, 0)$
- $(\mathbb{R}, \times, 1)$
- $(\{\text{TRUE}, \text{FALSE}\}, \wedge, \text{TRUE})$
- $(\Sigma^*, \|, \epsilon)$
- $(\mathbb{Z}, \text{MAX}, -\infty)$

Representing Monoids

In Cilk++ we represent a monoid over `T` by a C++ class that inherits from `cilk::monoid_base<T>` and defines

- a member function `reduce()` that implements the binary operator $\otimes$ and
- a member function `identity()` that constructs a fresh identity `e`.

Example:

```
struct sum_monoid : cilk::monoid_base<int> {
  void reduce(int* left, int* right) const {
    *left += *right; // order is important!
  }
  void identity(int* p) const {
    new (p) int(0);
  }
};
```

Defining a Reducer

A reducer over `sum_monoid` may now be defined as follows:

```
cilk::reducer<sum_monoid> x;
```

The local view of `x` can be accessed as `x()`.

Issues

- It is generally inconvenient to replace every access to `x` in a legacy code base with `x()`.
- Accesses to `x` are not safe. Nothing prevents a programmer from writing "`x() *= 2`", even though the reducer is defined over `+`.

⇨ A wrapper class solves these problems.

Reducer Library

Cilk++'s hyperobject library contains many commonly used reducers:

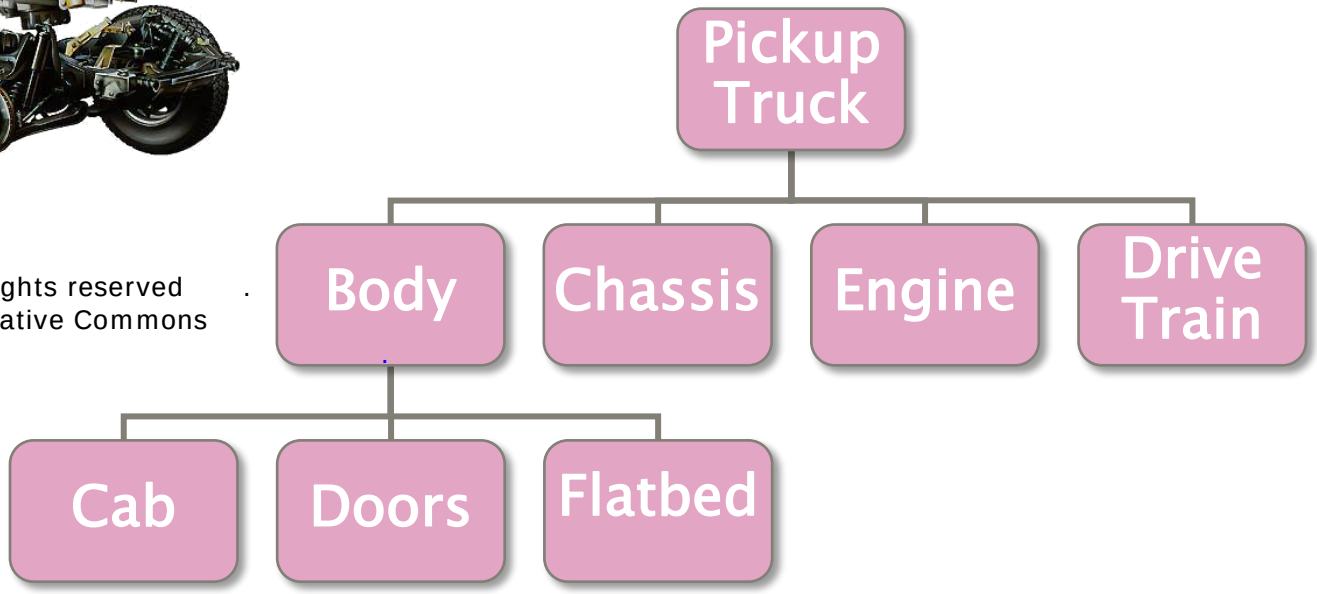
- `reducer_list_append`
- `reducer_list_prepend`
- `reducer_max`
- `reducer_max_index`
- `reducer_min`
- `reducer_min_index`
- `reducer_opadd`*
- `reducer_ostream`
- `reducer_basic_string`
- …

You can also roll your own reducers using `cilk::monoid_base` and `cilk::reducer`.

*Behavior is nondeterministic when used with floating-point numbers.

Real-World Example

A mechanical assembly is represented as a tree of subassemblies down to individual parts.



Collision-detection problem: Find all "collisions" between an assembly and a target object.

C++ Code

Goal

Create a list of all the parts in a mechanical assembly that collide with a given target object.

```
Node *target;
std::list<Node *> output_list;
...
void walk(Node *x)
{
  switch (x->kind) {
  case Node::LEAF:
    if (target->collides_with(x))
      {
        output_list.push_back(x);
      }
    break;
  case Node::INTERNAL:
    for (Node::const_iterator child = x.begin();
                 child != x.end();
                 ++child)
      {
        walk(child);
      }
    break;
  }
}
```

Naive Parallelization

Idea

Parallelize the search by using a `cilk_for` to search all the children of each internal node in parallel.

Oops!

```
Node *target;
std::list<Node *> output_list;
...
void walk(Node *x)
{
  switch (x->kind) {
  case Node::LEAF:
    if (target->collides_with(x))
      {
        output_list.push_back(x);
      }
    break;
  case Node::INTERNAL:
    cilk_for (Node::const_iterator child = x.begin();
              child != x.end();
              ++child)
      {
        walk(child);
      }
    break;
  }
}
```

Problematic Parallelization

Problem

The global variable `output_list` is updated in parallel, causing a *race bug*.

```
Node *target;
std::list<Node *> output_list;
...
void walk(Node *x)
{
  switch (x->kind) {
  case Node::LEAF:
    if (target->collides with(x))
      {
        output_list.push_back(x);
      }
    break;
  case Node::INTERNAL:
    cilk_for (Node::const_iterator child = x.begin();
              child != x.end();
              ++child)
      {
        walk(child);
      }
    break;
  }
}
```

Race!

A Mutex Solution

Locking

Each leaf locks `output_list` to ensure that updates occur atomically. Unfortunately, *lock contention* inhibits speed-up. Also, the list is produced in a jumbled order.

```
Node *target;
std::list<Node *> output_list;
mutex output_list_mutex;
...
Void walk(Node *x)
{
  switch (x->kind) {
  case Node::LEAF:
    if (target->collides_with(x))
      {
        output_list_mutex.lock();
        output_list.push_back(x);
        output_list_mutex.unlock();
      }
    break;
  case Node::INTERNAL:
    cilk_for (Node::const_iterator child = x.begin();
                child != x.end();
                ++child)
      {
        walk(child);
      }
    break;
  }
}
```

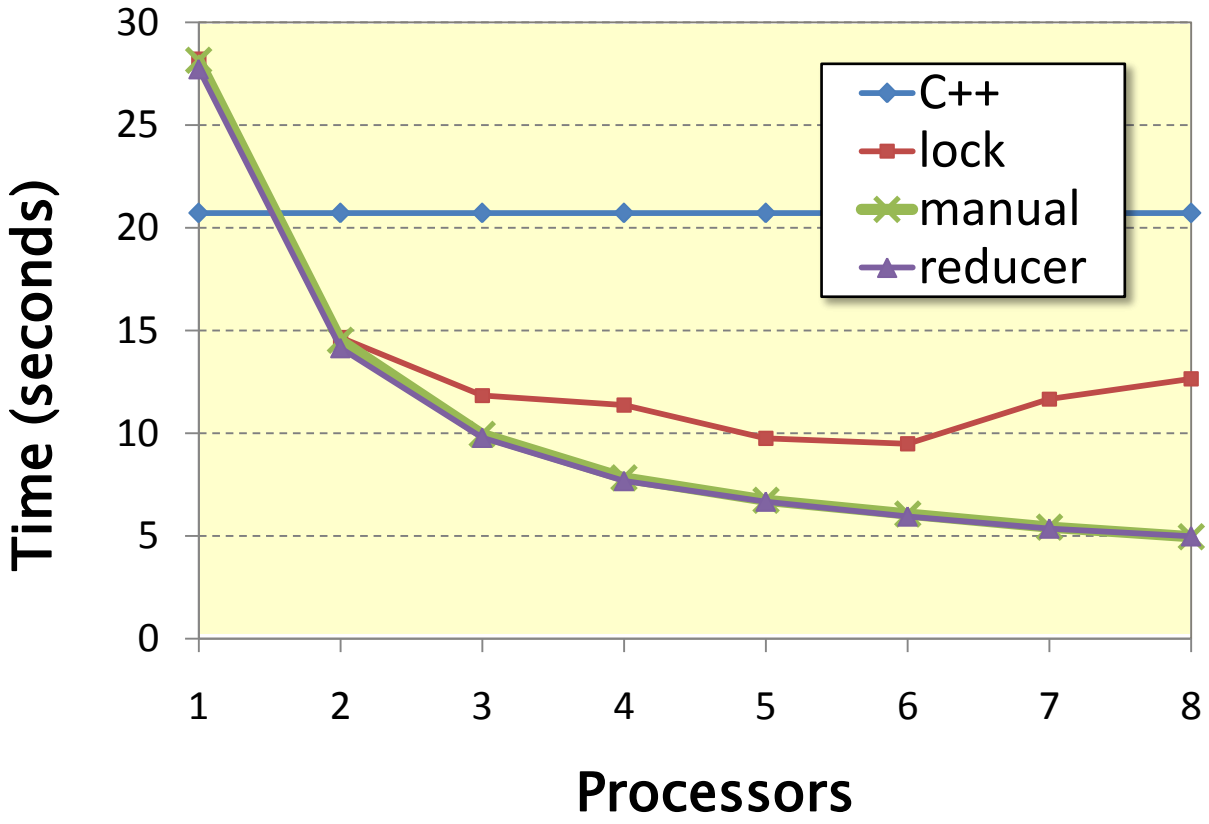
Hyperobject Solution

Declare `output_list` to be a *reducer* whose `reduce()` function concatenates lists.* The `output_list` is produced in the same order as in the original C++.

```
Node *target;
cilk::reducer_list_append<Node *> output_list;
...
void walk(Node *x)
{
  switch (x->kind) {
  case Node::LEAF:
    if (target->collides_with(x))
      {
        output_list.push_back(x);
      }
    break;
  case Node::INTERNAL:
    cilk_for (Node::const_iterator child = x.begin();
              child != x.end();
              ++child)
      {
        walk(child);
      }
    break;
  }
}
```

*List concatenation is associative.

Performance of Collision Detection



Reducer Implementation

- Each worker (processor) maintains a ***hypermap*** as a hash table, which maps hyperobjects into views.*
- An access to a reducer `x()` causes the worker to look up the local view of `x()` in the hypermap.
- If a view of `x()` does not exist in the hypermap, the worker creates a new view with value `e`.
- During load-balancing, when a worker "steals" a subcomputation, it creates an empty hypermap.
- When a worker finishes its subcomputation, hypermaps are combined using the appropriate `reduce()` functions.
- The actual distributed protocol becomes rather tricky to avoid deadlock and ensure rapid completion — see the SPAA 2009 paper or the code itself for details.

*In fact, each worker maintains two additional auxiliary hypermaps to assist in bookkeeping.

Overheads

- For programs with sufficient parallelism, the total cost of performing O(1)-time `reduce()` functions is provably small.
- The cost of an access to a reducer view is never worse than a hash-table look-up.
- If the reducer is accessed several times within a region of code, however, the compiler can optimize look-ups using common-subexpression elimination.
- In this common case, the hash-table look-up is performed only once, resulting in an access cost equal to one additional level of indirection (typically an L1-cache hit).

6.172 Performance Engineering of Software Systems

Fall 2010