Missing optimizations in node-based containers

ACCU 2017
April 28, 2017

Elliot Goodrich, Software Engineer
egoodrich4@bloomberg.net

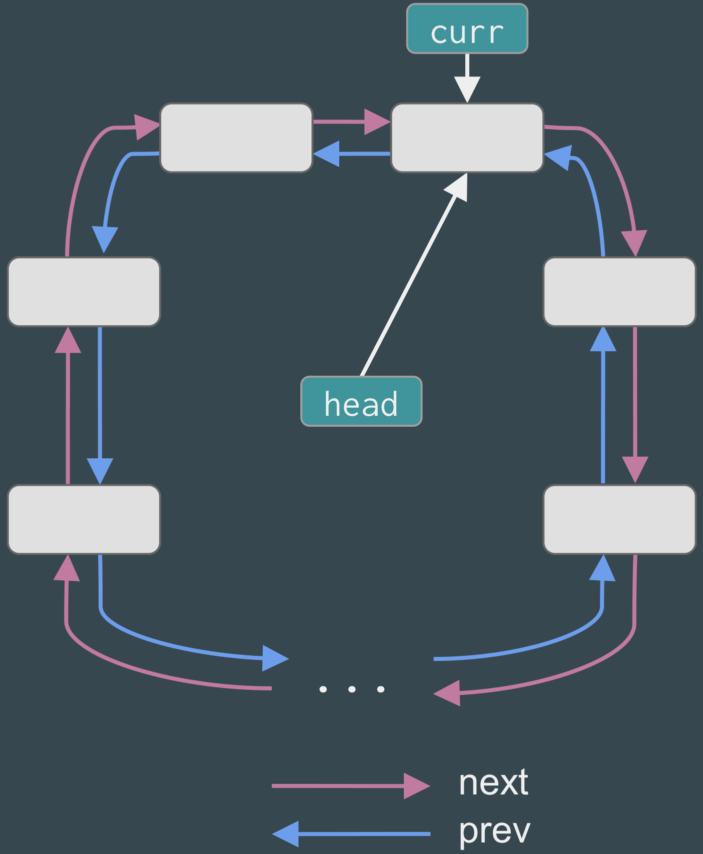
techatbloomberg.com

Bloomberg

How fast can we reverse a doubly linked list?

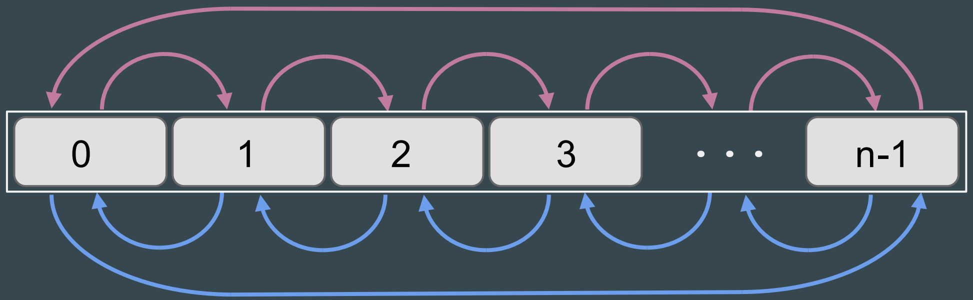
std::list::reverse

```
struct node {
    node* next;
    node* prev;
};

void reverse(node* head) {
    node* curr = head;

    do {
        std::swap(curr->prev, curr->next);
        curr = curr->prev;
    } while (curr != head);
}
```

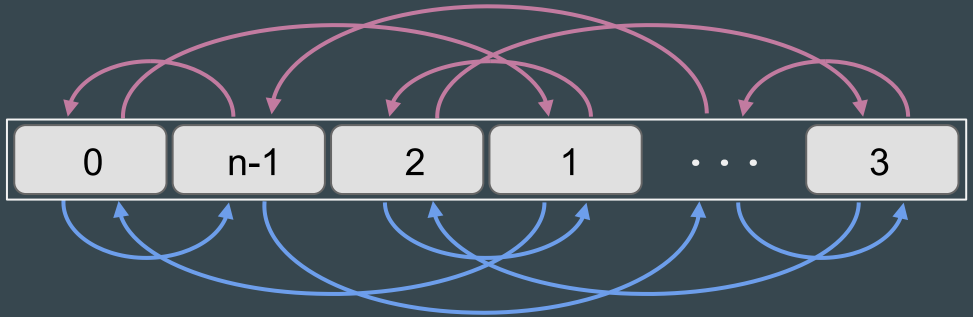


std::list::reverse benchmark results

Sequentially Placed Nodes 119.46 million nodes per second



Randomly Placed Nodes 10.47 million nodes per second

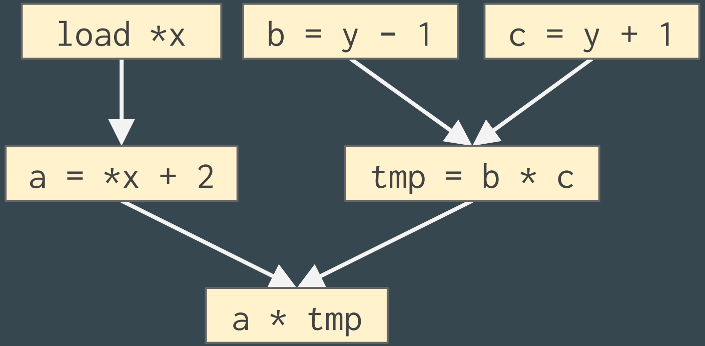


- Memory layout matters!
- Accessing memory from RAM is slow (100+ CPU cycles)
- Predictable memory access allows cache prefetcher to fetch memory in advance
- Nodes have been padded to 64 bytes
- i5 3750K CPU (2012)

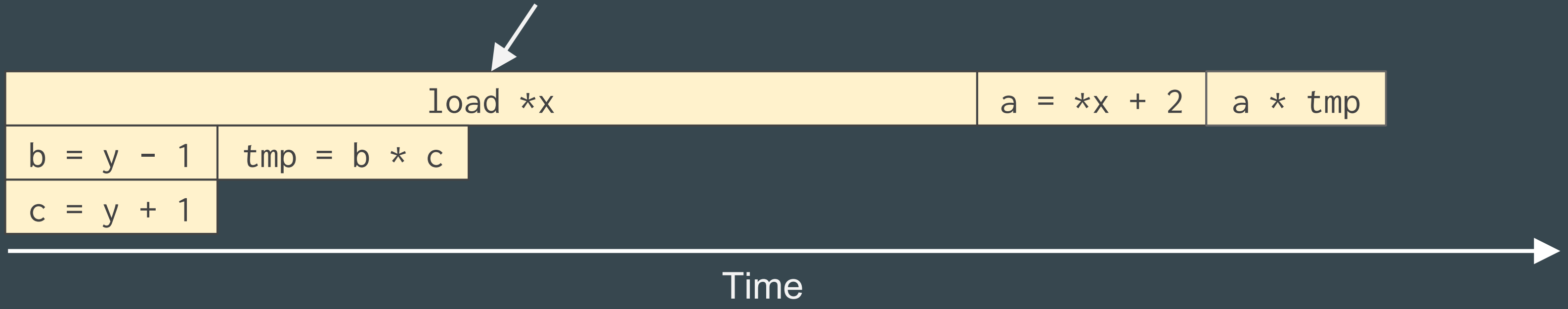
Instruction-Level Parallelism

- Modern CPUs have **Out-of-Order Execution**
- Instructions can be executed when their inputs are ready, not based on the instruction order
- Independent instructions can be executed in parallel (still on one thread)

```
int f(int* x, int y) {
    int a = *x + 2;
    int b =  y - 1;
    int c =  y + 1;
    return a * b * c;
}
```



If *x is not in the cache, this operation dominates



Goal: Remove data dependencies!

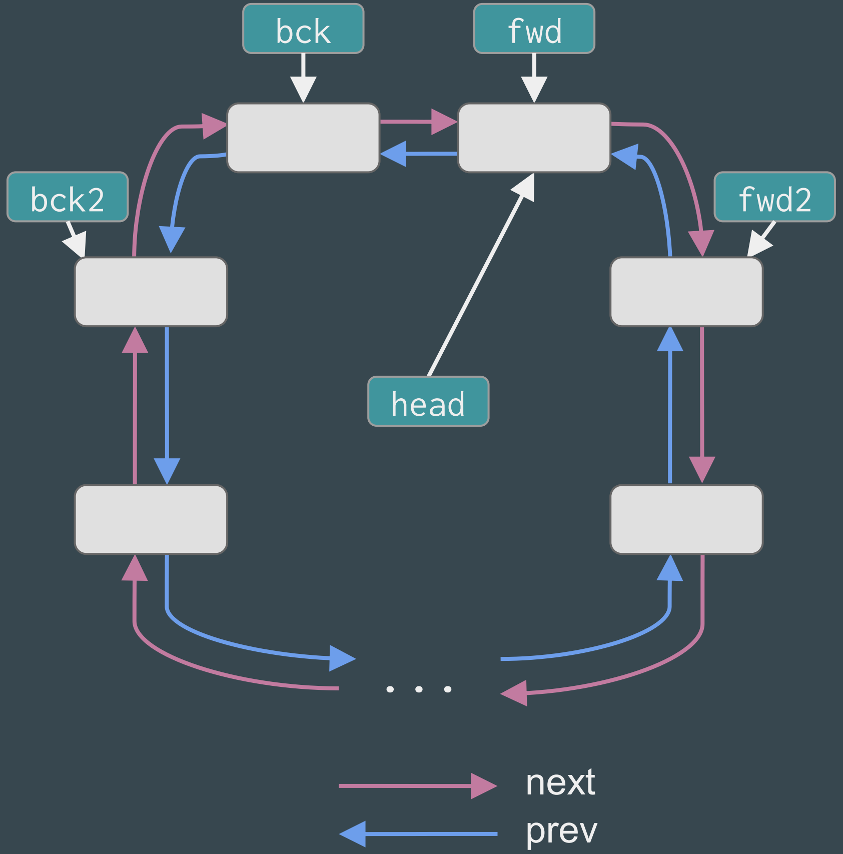
Data dependencies in std::list::reverse

```
void reverse(node* head) {
    node* curr = head;

    do {
        std::swap(curr->prev, curr->next);
        curr = curr->prev;
    } while (curr != head);
}
```

- Pointer chasing has data dependencies
- Modern CPUs have **Memory-Level Parallelism**
- Idea – follow the `prev` and `next` pointers together

std::list::reverse (unrolled)

```
void reverse(node* head) {
    node* fwd = head;
    node* bck = head->prev;
    if (fwd == bck) return;
    while (true) {
        node* fwd2 = fwd->next;
        node* bck2 = bck->prev;
        std::swap(fwd->next, fwd->prev);
        std::swap(bck->next, bck->prev);

        if (fwd2 == bck2) {
            std::swap(fwd2->next, fwd2->prev);
            return;
        }
        if (fwd2 == bck) return;

        fwd = fwd2;
        bck = bck2;
    }
}
```



std::list::reverse (unrolled)

```
void reverse(node* head) {
    node* fwd = head;
    node* bck = head->prev;
    if (fwd == bck) return;
    while (true) {
        node* fwd2 = fwd->next;
        node* bck2 = bck->prev;
        std::swap(fwd->next, fwd->prev);
        std::swap(bck->next, bck->prev);

        if (fwd2 == bck2) {
            std::swap(fwd2->next, fwd2->prev);
            return;
        }
        if (fwd2 == bck) return;

        fwd = fwd2;
        bck = bck2;
    }
}
```

- Twice as much work per iteration
- CPU can fetch `fwd2` and `bck2` in parallel since they are independent instructions
- **The optimisation is from the CPU!**
- Language agnostic

If we know the size of the linked list (required in C++11 onwards) then one of these branches can be removed

std::list::reverse (unrolled single if)

```
void reverse(node* head, int size) {
    node* fwd = head;
    node* bck = head->prev;
    if (size % 2 == 1) {
        std::swap(fwd->prev, fwd->next);
        fwd = fwd->prev;
    }
    if (size == 1) return;

    while (true) {
        node* fwd2 = fwd->next;
        node* bck2 = bck->prev;
        std::swap(fwd->next, fwd->prev);
        std::swap(bck->next, bck->prev);

        if (fwd2 == bck) return;
        fwd = fwd2;
        bck = bck2;
    }
}
```

Benchmarks

	Sequential	Speedup	Random	Speedup
reverse	119.46 M	1.00x	10.47 M	1.00x
unrolled	125.42 M	1.05x	19.23 M	1.84x
unrolled single if	125.42 M	1.05x	19.23 M	1.84x

Other functions that could equally benefit

- `list::merge`
- `list::splice` (O(N) overload splicing one iterator range to another list)
- `list::remove`/`list::remove_if`
- `list::merge`/`map::merge`
- `list::unique`
- `list::sort`
- `distance` (on `list`/`map` iterators)
- `any_of`/`all_of`/`none_of` (on `list`/`map` iterators)
- Any hash table (e.g. `unordered_map`) implementation that uses a doubly-linked list to solve collisions

Why does it fall short of 2x improvement?

```
node* fwd2 = fwd->next;
node* bck2 = bck->prev;
std::swap(fwd->next, fwd->prev);
std::swap(bck->next, bck->prev);
```

Read-After-Write (RAW) dependency between these swaps

- `fwd` and `bck` could alias
- CPU must wait for first `swap` before executing the second

Optimizing swaps

read

write

```
node* fwd2 = fwd->next;
node* bck2 = bck->prev;
// std::swap(fwd->next, fwd->prev);
fwd->next  = fwd->prev;
fwd->prev  = fwd2;
// std::swap(bck->next, bck->prev);
bck->prev  = bck->next;
bck->next  = bck2;
```

asm

```
mov     rax, QWORD PTR [rdi+8]
mov     rdx, QWORD PTR [rdi]
mov     QWORD PTR [rdi], rax
mov     QWORD PTR [rdi+8], rdx
mov     rdx, QWORD PTR [rsi]
mov     rax, QWORD PTR [rsi+8]
mov     QWORD PTR [rsi+8], rdx
mov     QWORD PTR [rsi], rax
```

rewrite

```
node* fwd2 = fwd->next;
node* bck2 = bck->prev;
node* tmp  = bck->next;
fwd->next  = fwd->prev;
fwd->prev  = fwd2;
bck->next  = bck2;
bck->prev  = tmp;
```

asm

```
mov     rcx, QWORD PTR [rdi+8]
mov     rdx, QWORD PTR [rsi]
mov     rax, QWORD PTR [rsi+8]
mov     r8, QWORD PTR [rdi]
mov     QWORD PTR [rdi], rcx
mov     QWORD PTR [rdi+8], r8
mov     QWORD PTR [rsi+8], rdx
mov     QWORD PTR [rsi], rax
```

std::list::reverse (double_swap)

```cpp
void reverse(node* head, int size) {
    node* fwd = head;
    node* bck = head->prev;
    if (size % 2 == 1) {
        std::swap(fwd->prev, fwd->next);
        fwd = fwd->prev;
    }
    if (size == 1) return;

    while (true) {
        node* fwd2 = fwd->next;
        node* bck2 = bck->prev;
  →     double_swap(fwd, bck);

        if (fwd2 == bck) return;
        fwd = fwd2;
        bck = bck2;
    }
}
```

Benchmarks

	Sequential	**Speedup**	**Random**	**Speedup**
reverse	119.46 M	1.00x	10.47 M	1.00x
unrolled	125.42 M	1.05x	19.23 M	1.84x
unrolled single if	125.42 M	1.05x	19.23 M	1.84x
double_swap	128.21 M	1.07x	20.40 M	1.95x

Other examples of unnecessary RAW dependencies

```
// Visual Studio
node* _Unlinknode(const_iterator _Where) {
    node* _N = _Where.node();
    _N->prev->next = _N->next;
    _N->next->prev = _N->prev;
    --this->size;
    return _N;
}
```

```
// libc++
void __unlink_nodes(node* __f, node* __l) {
    __f->prev->next = __l->next;
    __l->next->prev = __f->prev;
}
```

```
// Visual Studio
node* _Unlinknode(const_iterator _Where) {
    node* _N = _Where.node();
    node* _Next = _N->next;
    node* _Prev = _N->prev;
    _Prev->next = _Next;
    _Next->prev = _Prev;
    --this->size;
    return _N;
}
```

```
// libc++
void __unlink_nodes(node* __f, node* __l) {
    node* __next = __l->next;
    node* __prev = __f->prev;
    __prev->next = __next;
    __next->prev = __prev;
}
```

Does the `restrict` keyword help?

- `restrict` qualifier (and `__restrict`/`__restrict__`) restricts **pointer aliasing**
- `restrict` would give more information to the compiler (e.g. our writes are not dependent)
- Success differs between compilers and what function you’re optimizing
- Rewriting functions gives more consistent results

Further (and slightly crazier) Optimizations

- Need more pointers to chase in parallel - requires making changes to the linked list
- If our linked list held a pointer to a node midway then this would give us 4 independent pointers to chase
    - Following `next` from `head`
    - Following `prev` from `head`
    - Following `next` from `midway`
    - Following `prev` from `midway`
- Problem! Can’t update midway while splicing
- Require O(N) time to update midway, but splice within a list must be O(1)

Loopidly linked lists

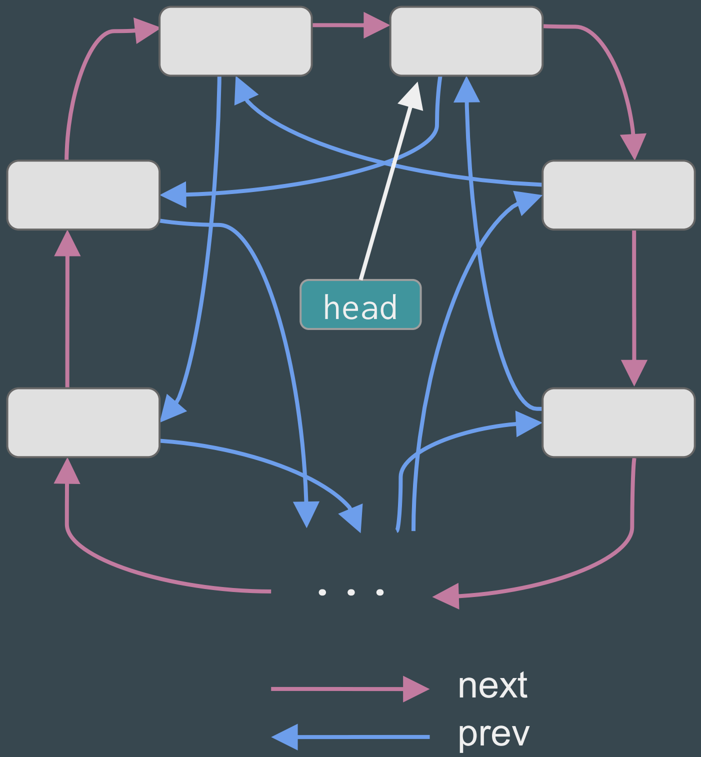
- `next` pointer is unchanged
- `prev` pointer points to the node **two positions back** in the list

Pros

- Independently chase two pointers backwards and one forward

Cons

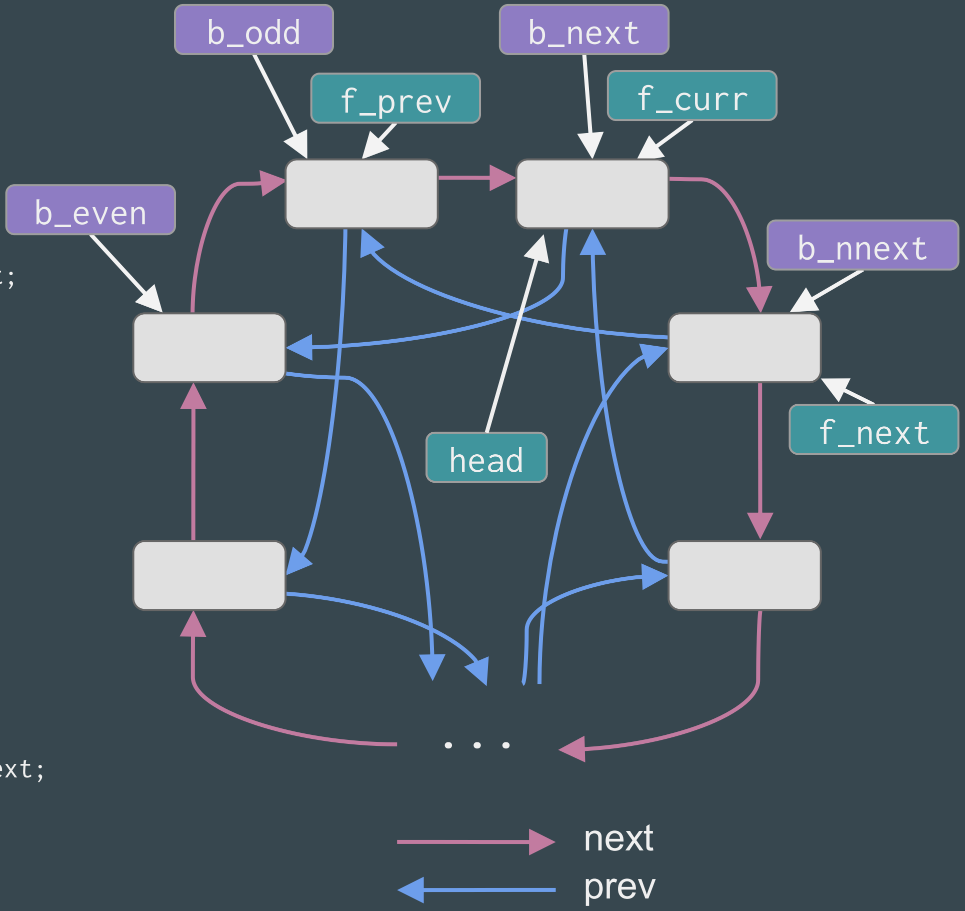
- Given a node `n`, the node before it will be `n->prev->next`, which will take twice as long to access
- Have to modify two extra nodes when inserting or deleting



Loopidly linked lists

```
void reverse(node* head, int size) {
    node* f_prev = head->prev->next;
    node* f_curr = head;
    node* f_next = head->next;
    node* b_nnext = f_next;     node* b_next = head;
    node* b_even  = head->prev; node* b_odd  = b_even->next;
    switch (size / 3) { /* process 0-2 items */ }

    while (true) {
        node* f_nnext = f_next->next;
        node* b_prev  = b_odd->prev;
        node* b_pprev = b_even->prev;

        f_curr->next = f_prev;  f_curr->prev = f_nnext;
        b_odd->next  = b_even;  b_odd->prev  = b_nnext;
        b_even->next = b_prev;  b_even->prev = b_next;

        if (b_prev == f_curr) return;

        f_prev  = f_curr;  f_curr = f_next;  f_next = f_nnext;
        b_nnext = b_odd;   b_next = b_even;
        b_odd   = b_prev;  b_even = b_pprev;
    }
}
```

} Independent loads



Benchmarks

	Sequential	**Speedup**	**Random**	**Speedup**
reverse	119.46 M	1.00x	10.47 M	1.00x
unrolled	125.42 M	1.05x	19.23 M	1.84x
unrolled single if	125.42 M	1.05x	19.23 M	1.84x
double_swap	128.21 M	1.07x	20.40 M	1.95x
loopidly	128.21 M	1.07x	26.83 M	2.56x

Loopidly linked lists

- `prev` pointer could skip three nodes (or four, or five…) giving more independent loads
- Or the `next` pointer could skip ahead as well
- Diminishing returns on skipping more nodes

Thanks for listening!

•••

Elliot Goodrich
egoodrich4@bloomberg.net
elliotgoodrich@gmail.com