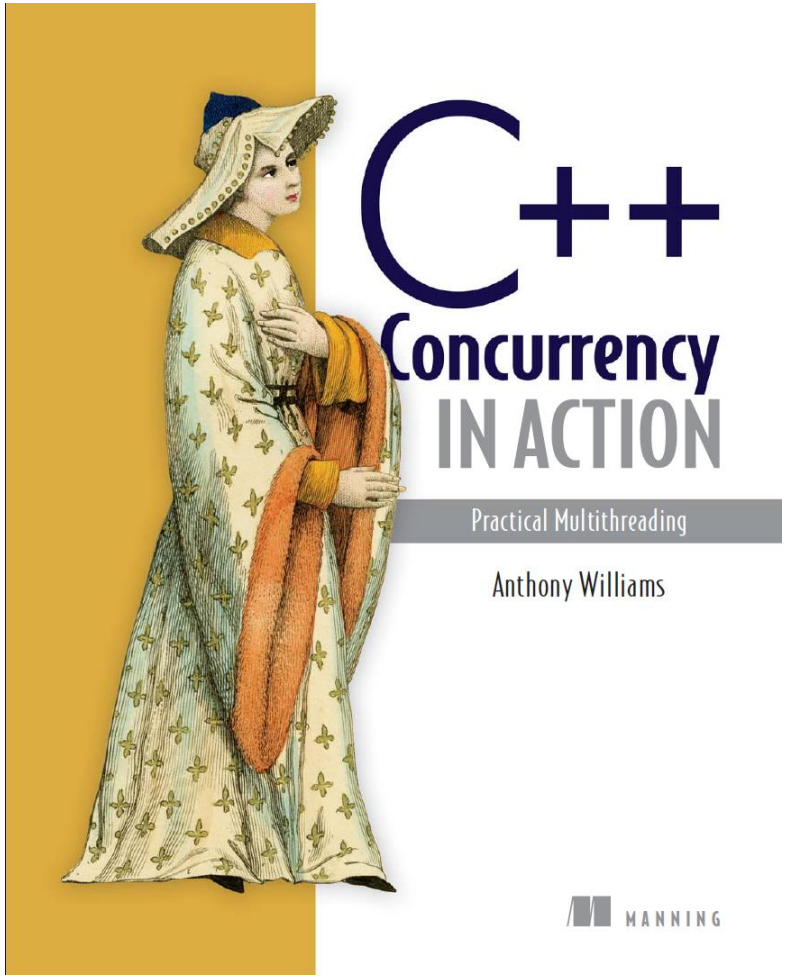
Implementing A Lock-free `atomic_shared_ptr<>`

Michael McCarty

Siemens PLM Software


C++
Concurrency
IN ACTION
Practical Multithreading
Anthony Williams
MANNING

Background

- The `shared_ptr<>` analogue to `atomic<T>` types
- Originally proposed by Herb Sutter (N4162)
  - Revision of rejected `atomic<shared_ptr<>>` proposal (N4058)
- Accepted as part of C++17's Concurrency TS
  - along side `atomic_weak_ptr<>`

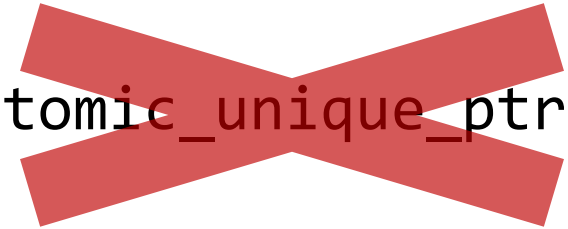
atomic_shared_ptr<> API

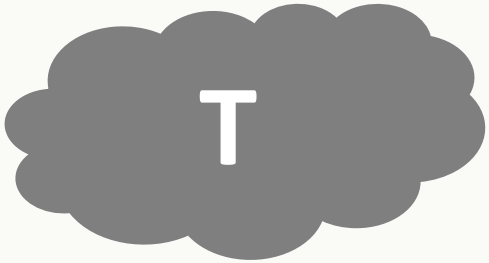
```
namespace std { namespace experimental {
template <typename T> class atomic_shared_ptr {
        void            store(shared_ptr) noexcept;
        shared_ptr      load() const noexcept;
        shared_ptr      exchange(shared_ptr) noexcept;

        bool            compare_exchange_weak(shared_ptr&,
                                              const shared_ptr&) noexcept;
        bool            compare_exchange_strong(shared_ptr&,
                                              const shared_ptr&) noexcept;
};
} }
```

The anatomy of a `shared_ptr<>`

What you won't see today…

```
template <typename T> …

…, memory_order order = memory_order_seq_cst)

atomic_weak_ptr<>

atomic_unique_ptr<>
```






```
struct SharedCounts {
    atomic<long> sCount;  // strong count
    atomic<long> wCount;  // weak count
};
```

```
class shared_ptr {
    SharedCounts* pC_;
    T*            pT_;
};
```

```
class weak_ptr {
    SharedCounts* pC_;
    T*            pT_;
};
```

Q: Why does shared_ptr hold two pointers?

```
struct X { …; };
struct Y { …; };
struct Z : X, Y { …; };


shared_ptr<X> pX = make_shared<Z>(…);


shared_ptr<Y> pY = dynamic_pointer_cast<Y>(pX);
```

A lock-based solution

```
class atomic_shared_ptr {
    typedef … Mutex;
    typedef unique_lock<Mutex> Lock;

    Lock  guard() { return Lock{mtx_}; }

    Mutex          mtx_;
    shared_ptr     sp_;
};
```

```
shared_ptr  atomic_shared_ptr::load()
{
    auto _ = guard();
    return sp_;
}
```

```
shared_ptr  atomic_shared_ptr::exchange(shared_ptr x)
{
    auto _ = guard();
    swap(sp_, x);
    return x;
}
```

```
void  atomic_shared_ptr::store(shared_ptr x)
{
    exchange(x);
}
```

```
bool atomic_shared_ptr::compare_exchange_weak(shared_ptr& x,
                                              shared_ptr  v)
{
   shared_ptr gc;

   auto _ = guard();

   const bool matched = ( x == sp_ );
   if ( matched )
        swap(sp_, v);
   else
   {
      swap(x, gc);
        x = sp_;
   }

   return matched;
}
```

Critical Section

A lock-*free* solution

What does lock-free actually mean?

- *"… if it satisfies that when the program threads are run sufficiently long at least one of the threads makes progress (for some sensible definition of progress)."*

-- *wikipedia.org*

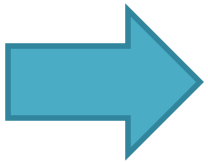
Benefits of lock-free

- Scalability
- Avoids certain classes of deadlock bugs
- Higher performance (maybe)

A first attempt

```
struct SharedCounts {
    atomic<long> sCount;  // strong count
    atomic<long> wCount;  // weak count
};
```

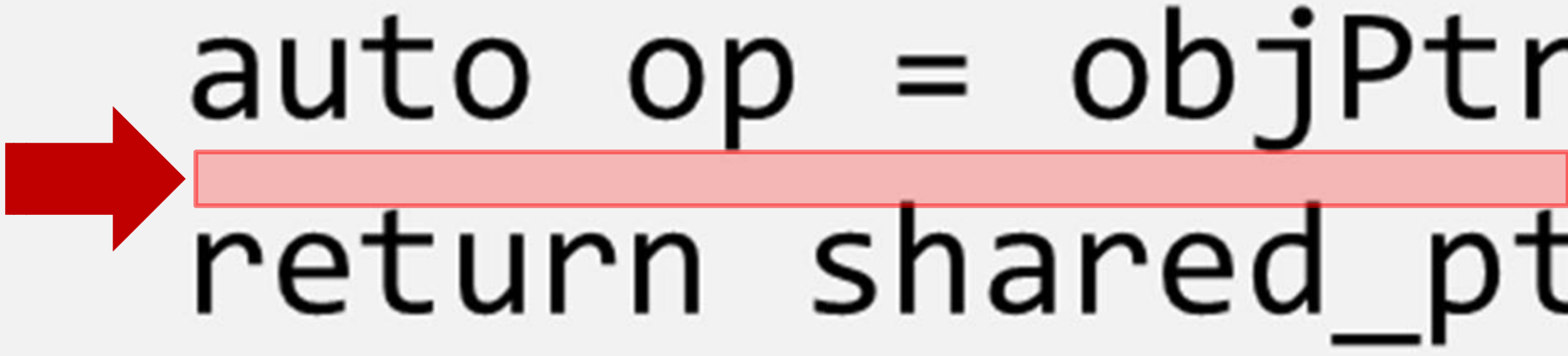
```
class shared_ptr {
    SharedCounts* pC;
    T*            pT;
};
```



```
class atomic_shared_ptr {
    struct ObjPtr {
        SharedCounts* pC;
        T*            pT;
    };

    atomic<ObjPtr>    objPtr;
};
```

```
shared_ptr  atomic_shared_ptr::load()
{
    auto op = objPtr.load();
    return shared_ptr{op.pT, op.pC};
}
```

```
shared_ptr  atomic_shared_ptr
{
    auto op = objPtr.load();
    return shared_ptr{op.pT,
}
```

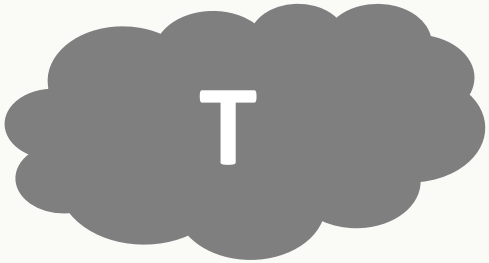

auto op = objPtr
return shared_pt

Differential Reference Counting

Step #1:

CHEAT

Intrusive shared_ptr<>




```
struct SharedCounts {
    atomic<long> sCount;
    atomic<long> wCount;
};
```

```
class shared_ptr {
    SharedCounts* pC_;
    T*            pT_;
};
```

```
class weak_ptr {
    SharedCounts* pC_;
    T*            pT_;
};
```

```
class T : public SharedObject
```

```
SharedObject {
    atomic<long> sCount;
};
```

```
class shared_ptr {

    T*              pT_;
};
```

```
class weak_ptr {

    T*              pT_;
};
```

```
struct SplitCount {
    long            tDiff;  // transient differential
    long            sCount; // strong count
};
```

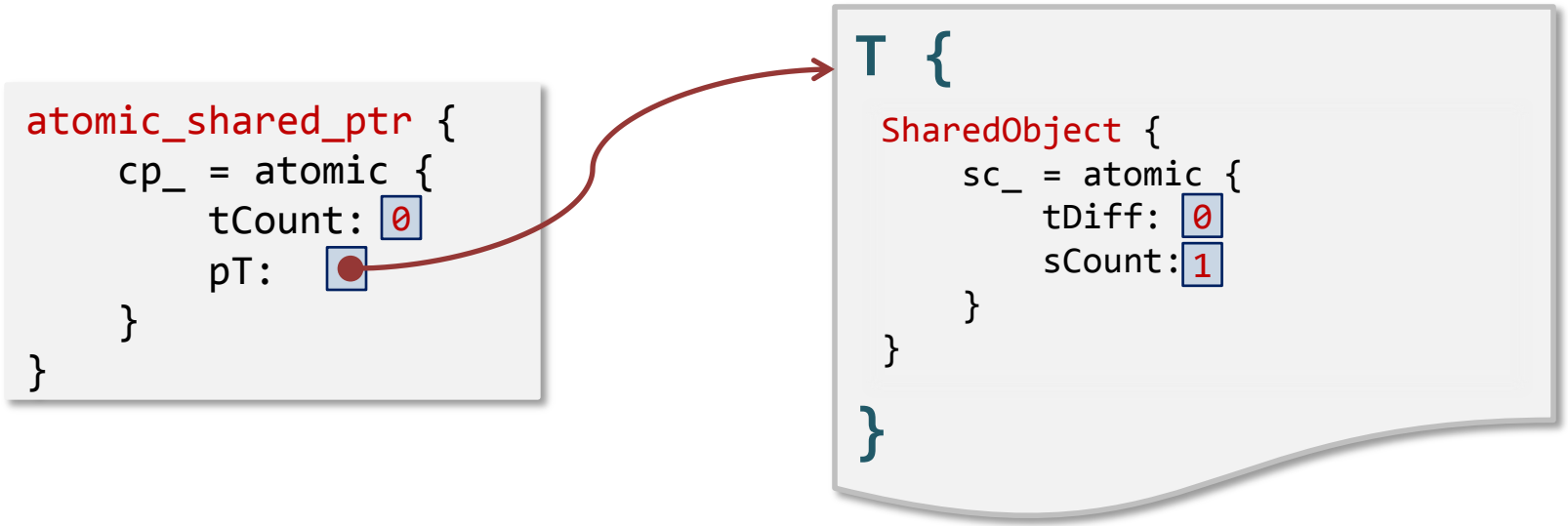
```
struct SharedObject {
    atomic<long>          sc_;
};
```

```
struct SplitCount {
    long           tDiff;  // transient differential
    long           sCount; // strong count
};
```

```
struct SharedObject {
    atomic<SplitCount>  sc_;
};
```

```
struct CountedPtr {
    long            tCount;  // transient count
    T*              pT;
};
```

```
class atomic_shared_ptr {

    atomic<CountedPtr>    cp_;

};
```


atomic_shared_ptr {
cp_ = atomic {
tCount: 0
pT:
}
}
T {
SharedObject {
sc_ = atomic {
tDiff: 0
sCount: 1
}
}
}

```
shared_ptr  atomic_shared_ptr::load()
{
    CountedPtr cp = acquireTransientRef();




}
```

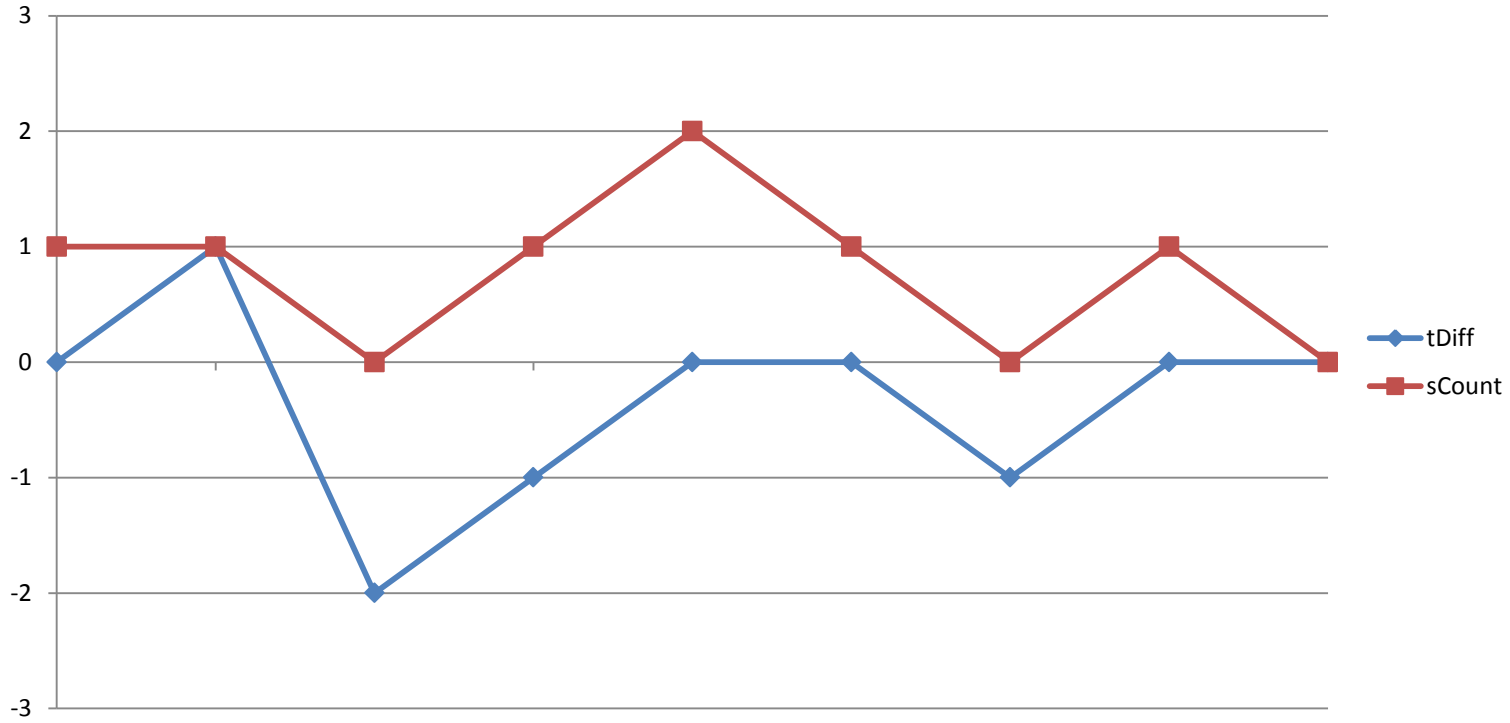
```
CountedPtr atomic_shared_ptr::acquireTransientRef()
{
    CountedPtr next;
    CountedPtr curr = cp_.load();

    do {
        next = curr;
        if ( !next.pT )
            break;

    ➔   next.tCount += 1;
    } while ( !cp_.compare_exchange_weak(curr, next) );

    return next;
}
```

```
shared_ptr  atomic_shared_ptr::load()
{
    CountedPtr cp = acquireTransientRef();




}
```

```
shared_ptr  atomic_shared_ptr::load()
{
    CountedPtr cp = acquireTransientRef();

    if ( cp.pT )
        cp.pT->increment( SplitCount{1,1} );


}
```

```
void SharedObject::increment( SplitCount delta )
{
    SplitCount curr = sc_.load();
    SplitCount total = curr + delta;
    while ( !sc_.compare_exchange_weak(curr, total) )
        total = curr + delta;

    if ( total.allZero() )
        delete this;
}
```

SplitCount time evolution

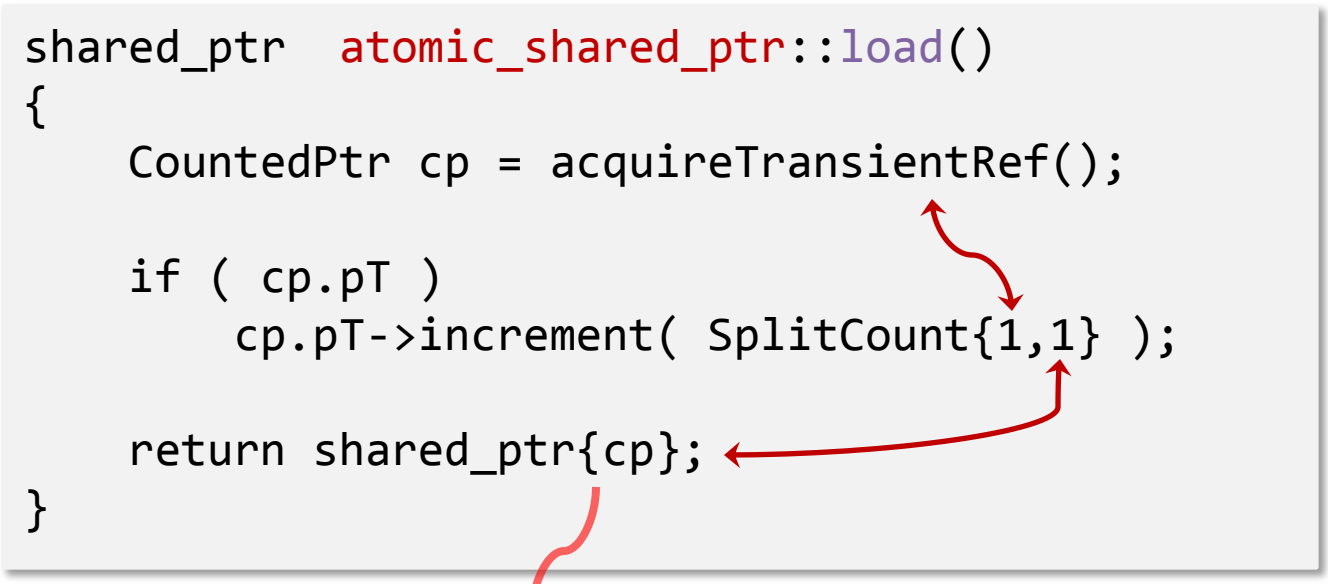


```
void SharedObject::increment( SplitCount delta )
{
    SplitCount curr = sc_.load();
    SplitCount total = curr + delta;
    while ( !sc_.compare_exchange_weak(curr, total) )
        total = curr + delta;

    if ( total.allZero() )
        delete this;
}
```

```
void SharedObject::decrement( SplitCount delta )
{
    SplitCount curr = sc_.load();
    SplitCount total = curr - delta;
    while ( !sc_.compare_exchange_weak(curr, total) )
        total = curr - delta;

    if ( total.allZero() )
        delete this;
}
```

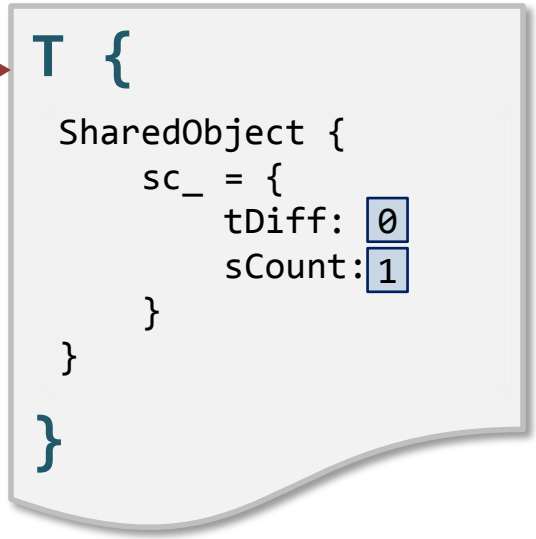
```
shared_ptr  atomic_shared_ptr::load()
{
    CountedPtr cp = acquireTransientRef();

    if ( cp.pT )
        cp.pT->increment( SplitCount{1,1} );


}
```

```
shared_ptr  atomic_shared_ptr::load()
{
    CountedPtr cp = acquireTransientRef();

    if ( cp.pT )
        cp.pT->increment( SplitCount{1,1} );

    return shared_ptr{cp};
}
```



```
shared_ptr(CountedPtr c)
: pT_(c.pT) { }
```

```
atomic_shared_ptr {
    cp_ = {
        tCount: 0
        pT:
    }
}
```

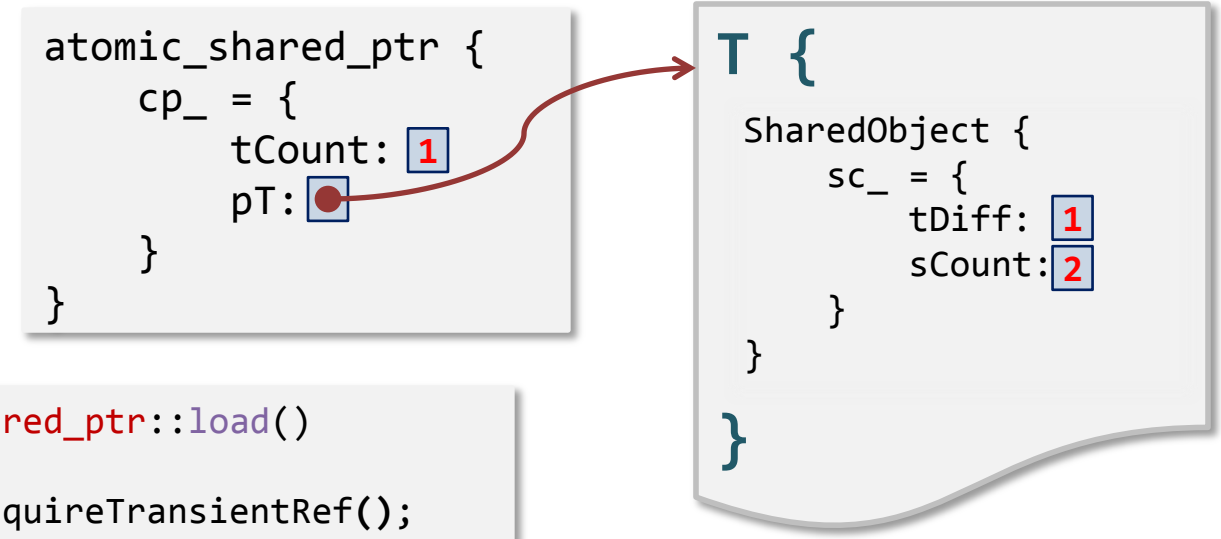



```
shared_ptr  atomic_shared_ptr::load()
{
    CountedPtr cp = acquireTransientRef();

    if ( cp.pT )
        cp.pT->increment( SplitCount{1,1} );

    return shared_ptr{cp};
}
```

```
atomic_shared_ptr {
    cp_ = {
        tCount: 1
        pT:
    }
}
```

```
T {
 SharedObject {
     sc_ = {
         tDiff: 0
         sCount: 1
     }
 }
}
```

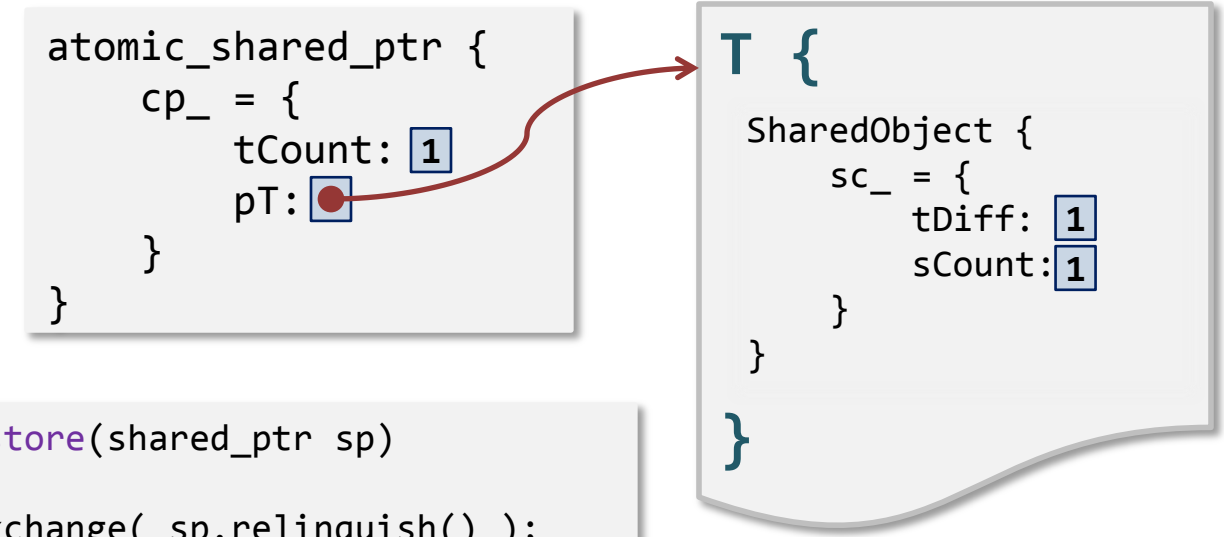
```
shared_ptr  atomic_shared_ptr::load()
{
    CountedPtr cp = acquireTransientRef();

    if ( cp.pT )
        cp.pT->increment( SplitCount{1,1} );

    return shared_ptr{cp};
}
```




```
shared_ptr  atomic_shared_ptr::load()
{
    CountedPtr cp = acquireTransientRef();

    if ( cp.pT )
     ➔ cp.pT->increment( SplitCount{1,1} );

    return shared_ptr{cp};
}
```

```
void  atomic_shared_ptr::store(shared_ptr sp)
{
    CountedPtr cp = cp_.exchange( sp.relinquish() );

    if ( cp.pT )
        cp.pT->decrement(SplitCount{cp.tCount, 1});
}
```

```
CountedPtr relinquish()
{
    CountedPtr rv{ p_, 0 };
    p_ = nullptr;
    return rv;
}
```




```
void  atomic_shared_ptr::store(shared_ptr sp)
{
    CountedPtr cp = cp_.exchange( sp.relinquish() );

    if ( cp.pT )
        cp.pT->decrement(SplitCount{cp.tCount, 1});
}
```

```
atomic_shared_ptr {
    cp_ = {
        tCount: 0
        pT:
    }
}
```

```
T {
 SharedObject {
     sc_ = {
         tDiff: 1
         sCount: 1
     }
 }
}
```

```
void  atomic_shared_ptr::store(shared_ptr sp)
{
➔ CountedPtr cp = cp_.exchange( sp.relinquish() );

    if ( cp.pT )
        cp.pT->decrement(SplitCount{cp.tCount, 1});
}
```

```
T {
```

```
atomic_shared_ptr {
    cp_ = {
        tCount: 0
        pT:
    }
}
```

```
T {
 SharedObject {
     sc_ = {
         tDiff: 0
         sCount: 0
     }
 }
}
```

```
void  atomic_shared_ptr::store(shared_ptr sp)
{
    CountedPtr cp = cp_.exchange( sp.relinquish() );

    if ( cp.pT )
        cp.pT->decrement(SplitCount{cp.tCount, 1});
}
```

```
T {
```

```
shared_ptr  atomic_shared_ptr::exchange(shared_ptr sp)
{
    CountedPtr cp = cp_.exchange( sp.relinquish() );

    if ( cp.pT )
        cp.pT->decrement(SplitCount{cp.tCount, 0});

    return shared_ptr{cp};
}
```

```
bool atomic_shared_ptr::compare_exchange_weak(shared_ptr& expected, shared_ptr value)
{
    const CountedPtr orig = acquireTransientRef();
    CountedPtr cnew = value.relinquish();
    CountedPtr curr = orig;

    bool matched = (expected.data() == orig.pT);
    if ( matched )        matched = cp_.compare_exchange_weak(curr, cnew);

    if ( matched )
        { if ( curr.pT )   curr.pT->decrement( SplitCount{curr.tCount-1,1} ); }
    else
    {
        value = shared_ptr{cnew};
        expected.reset();

        if ( !(curr == orig) )
        {
            if ( orig.pT )  orig.pT->increment( SplitCount{1,0} );
            expected = load();
        }
        else if ( curr.pT )
        {
            curr.pT->increment( SplitCount{1,1} );
            expected = shared_ptr{curr};
        }
    }
    return matched;
}
```

Is it actually *lock-free*?

In theory…

```
bool atomic_shared_ptr::compare_exchange_weak(shared_ptr& expected,
{
    const CountedPtr orig = acquireTransientRef();
    CountedPtr cnew = value.relinquish();
    CountedPtr curr = orig;

    bool matched = (expected.data() == orig.pT);
    if ( matched )          matched = cp_.compare_exchange_weak(curr, cnew);

    if ( matched )
        { if ( curr.pT )   curr.pT->decrement( SplitCount{curr.tCount-1,1} ); }
    else
    {
        value = shared_ptr{cnew};
        expected.reset();

        if ( !(curr == orig) )
        {
            if ( orig.pT )  orig.pT->increment( SplitCount{1,0} );
            expected = load();
        }
        else if ( curr.pT )
        {
            curr.pT->increment( SplitCount{1,1} );
            expected = shared_ptr{curr};
        }
    }
    return matched;
}
```

```
void  atomic_shared_ptr::store(shared_ptr sp)
{
    CountedPtr cp = cp_.exchange(

    if ( cp.pT )
        cp.pT->decrement(SplitCou
}
```

```
shared_ptr  atomic_shared_ptr::exchange(shared_ptr sp)
{
    CountedPtr cp = cp_.exchange( sp.relinquish() );

    if ( cp.pT )
        cp.pT->decrement(SplitCount{cp.tCount, 0});
```

```
CountedPtr atomic_shared_ptr::acquireTransientRef()
{
    CountedPtr next;
    CountedPtr curr = cp_.load();

    do {
        next = curr;
        if ( !next.pT )
            break;

        next.tCount += 1;
    } while ( !cp_.compare_exchange_weak(curr, next) );

    return next;
```

```
shared_ptr  atomic_shared_ptr::load()
{
    CountedPtr cp = acquireTransientRef();

    if ( cp.pT )
        cp.pT->increment( SplitCount{1,1}

    return shared_ptr{cp};
}
```

```
void SharedObject::increment( SplitCount delta )
{
    SplitCount curr = sc_.load();
    SplitCount total = curr + delta;
    while ( !sc_.compare_exchange_weak(curr, total) )
        total = curr + delta;

    if ( total.allZero() )
        delete this;
}
```

In practice…

```
struct CountedPtr {
    long            tCount;
    T*              pT;
};
```

```
class atomic_shared_ptr {
    atomic<CountedPtr>     cp_;
};
```

Lock-free?

```
atomic<CountedPtr>  acp;

cout << acp.is_lock_free() << endl;
```

`acp.is_lock_free()`

(64-bit executables)

Clang 3.6:	`true`
GCC 5.3:	`true`
Visual Studio 2015 Update 1:	`false`

```
struct CountedPtr {
    long            tCount;
    T*              pT;
};
```

sizeof(CountedPtr) == **16**

DWCAS

Double **W**idth **C**ompare **A**nd **S**wap

Intel x64 ISA: cmpxchg16b

CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F C7 /1 *m64* CMPXCHG8B *m64*	M	Valid	Valid*	Compare EDX:EAX with *m64*. If equal, set ZF and load ECX:EBX into *m64*. Else, clear ZF and load *m64* into EDX:EAX.
REX.W + 0F C7 /1 *m128* CMPXCHG16B *m128*	M	Valid	N.E.	Compare RDX:RAX with *m128*. If equal, set ZF and load RCX:RBX into *m128*. Else, clear ZF and load *m128* into RDX:RAX.

NOTES:
*See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA

Description

Compares the 64-bit value in EDX:EAX (or 128-bit value in RDX:RAX if operand size is 128 bits) with the operand (destination operand). If the values are equal, the 64-bit value in ECX:EBX (or 128-bit value in RCX:RBX) is stored in the destination operand. Otherwise, the value in the destination operand is loaded into EDX:EAX (or RDX:RAX).

A lock-free implementation

```
LBB0_1:                                 ## %atomicrmw.start
                                        ## =>This Inner Loop Header: Depth=1
    lock
    cmpxchg16b  (%rdi)
    jne LBB0_1
## BB#2:                                ## %atomicrmw.end
```

A not-lock-free implementation

```
        /* ATOMIC OPERATIONS FOR OBJECTS WITH SIZES THAT
            DON'T MATCH THE SIZE OF ANY INTEGRAL TYPE */
inline void _Atomic_copy(
    volatile _Atomic_flag_t *_Flag, size_t _Size,
        volatile void *_Tgt, volatile const void *_Src,
            memory_order _Order)
    {   /* atomically copy *_Src to *_Tgt with memory ordering */
    _Lock_spin_lock(_Flag);
    _CSTD memcpy((void *)_Tgt, (void *)_Src, _Size);
    _Unlock_spin_lock(_Flag);
    }
```

DWCAS::Atomic<T>

```
namespace DWCAS
{
    template <typename T> class Atomic
    {
#if defined(_M_X64)
        struct alignas(2 * sizeof(int64_t)) DW
        {
            int64_t i_[2];
            operator int64_t* () { return i_; }
            int64_t operator[] (size_t ii) const { return i_[ii]; }
        };

        static void init(DW& dw)    { dw.i_[0] = dw.i_[1] = 0; }
        static void twiddle(DW& dw) { dw.i_[0] ^= -int64_t(1 << 30); }
        static bool dwcas(DW& dest, const DW& value, DW& expected)
        {
            return _InterlockedCompareExchange128(dest, value[1], value[0], expected) != 0;
        }
#else
```

Step #2:

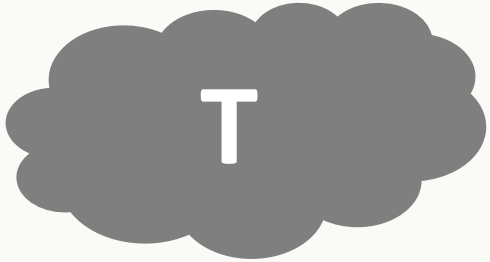
STOP CHEATING

Implement *non-intrusive*

```
class T : public SharedObject
```

```
SharedObject {
    atomic<SplitCount> sc_;
};
```

```
class atomic_shared_ptr {
    atomic<CountedPtr>   cp_;
};
```




```
struct SharedCounts {
    atomic<SplitCount> sc_;
    atomic<long>       wc_;
};
```

```
class atomic_shared_ptr {
    atomic<CountedPtr>   cp_;
};
```

```
struct CountedPtr {
    long           tCount;
    T*             pT;
    SharedCounts* pSC;
};
```

```
static_assert( sizeof(CountedPtr) == 2*sizeof(void*) );
```

```
struct CountedPtr {
    int             tCount;
    int             hPtr;
    SharedCounts* pSC;
};
```

```
static_assert( sizeof(CountedPtr) == 2*sizeof(void*) );
```

```
struct CountedPtr {
    int           tCount;
    int           hPtr;
    SharedCounts* pSC;
};
```

```
static_assert( sizeof(CountedPtr) == 2*sizeof(void*) );
```

```
struct CountedPtr {
    int           tCount;
    int           hPtr;
    SharedCounts* pSC;
};
```

```
struct SharedCounts {
    atomic<SplitCount> sc_;
    atomic<long>       wc_;




};
```

```
struct CountedPtr {
    int           tCount;
    int           hPtr;
    SharedCounts* pSC;
};
```

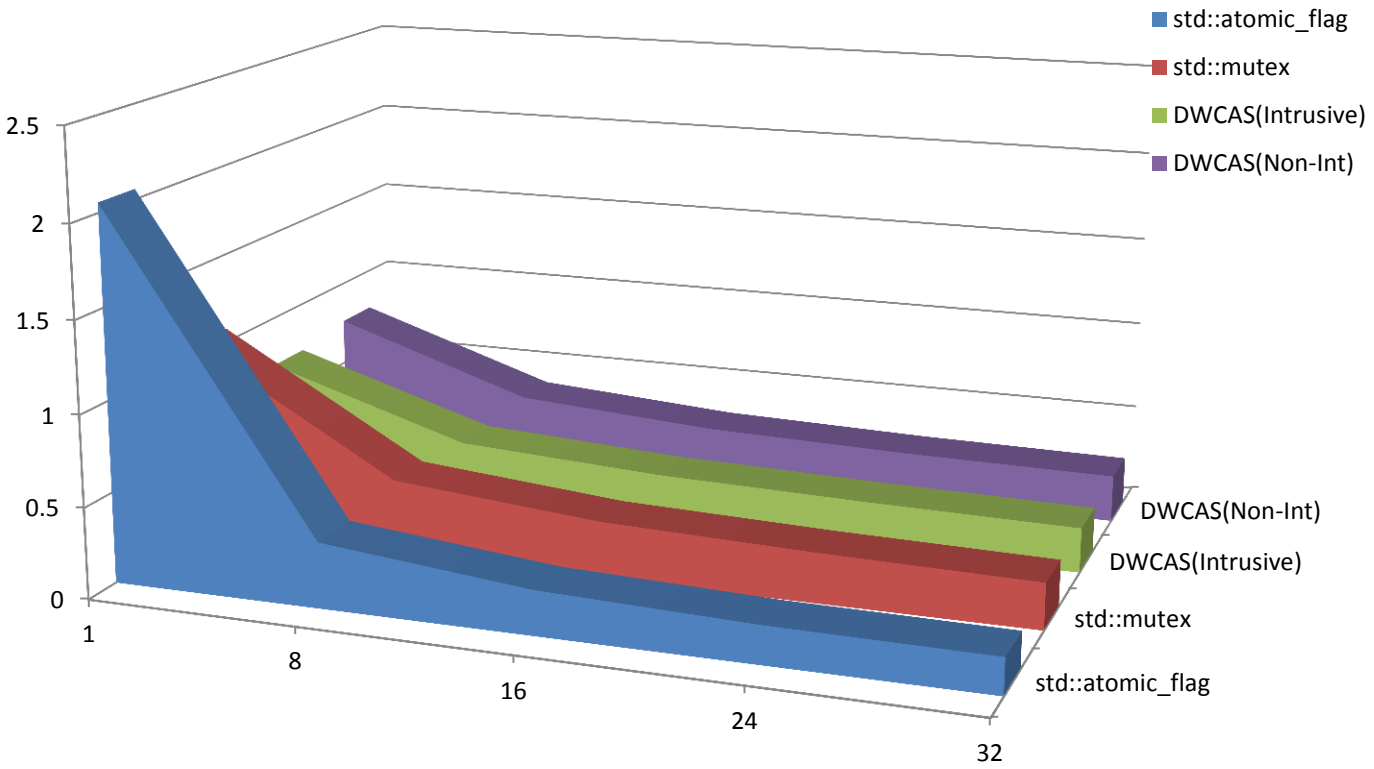
```
struct SharedCounts {
    atomic<SplitCount> sc_;
    atomic<long>       wc_;
    atomic<???>        vp_;

    void* getPtr(int h) { …; }
    int   regPtr(void*) { …; }
};
```

Performance

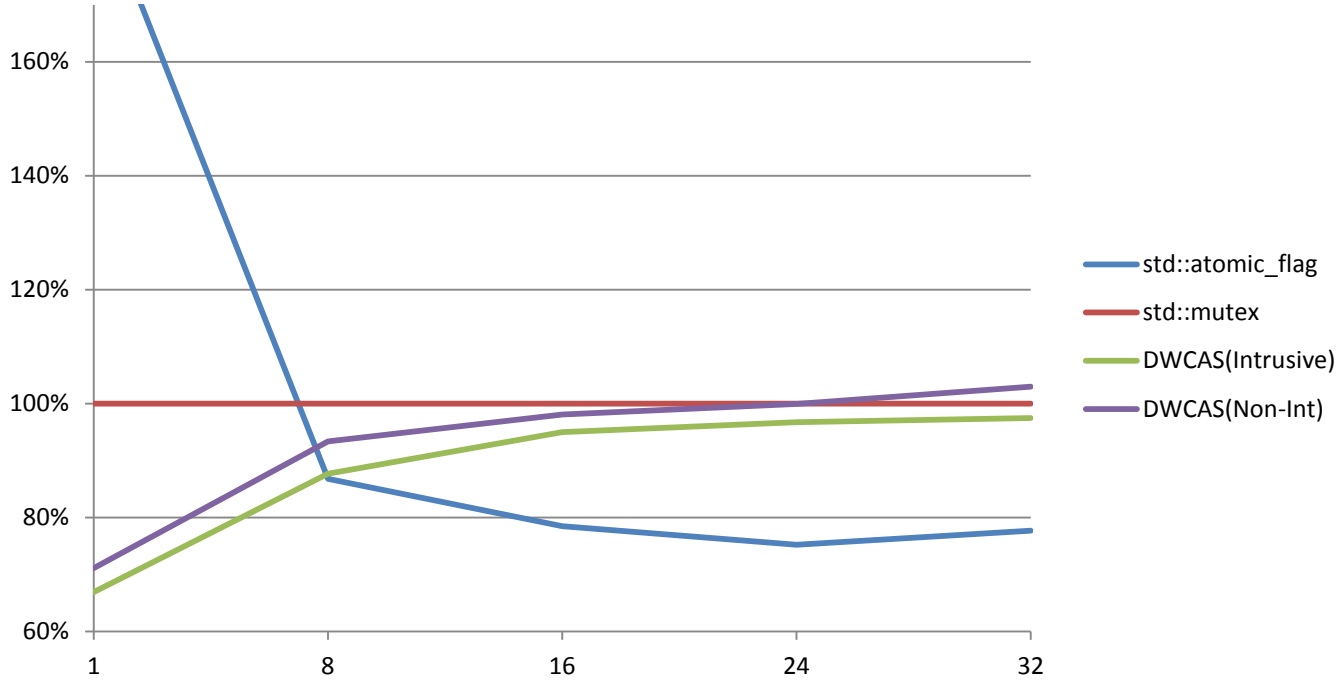
Benchmark #1: Random Operations

- Hardware:
  - Xeon E3-1270, 3.5 Ghz
  - 4 cores, 8 hardware threads
  - 16 Gb RAM
- Benchmark
  - Visual Studio 2015 Update 1, 64-bit build
  - 8 worker threads
  - 1M iterations / thread
    - randomly selected load(),store(), exchange(), CAS operations performed on randomly selected `atomic_shared_ptr<>`.
  - Average of 10 runs at set levels of memory contention

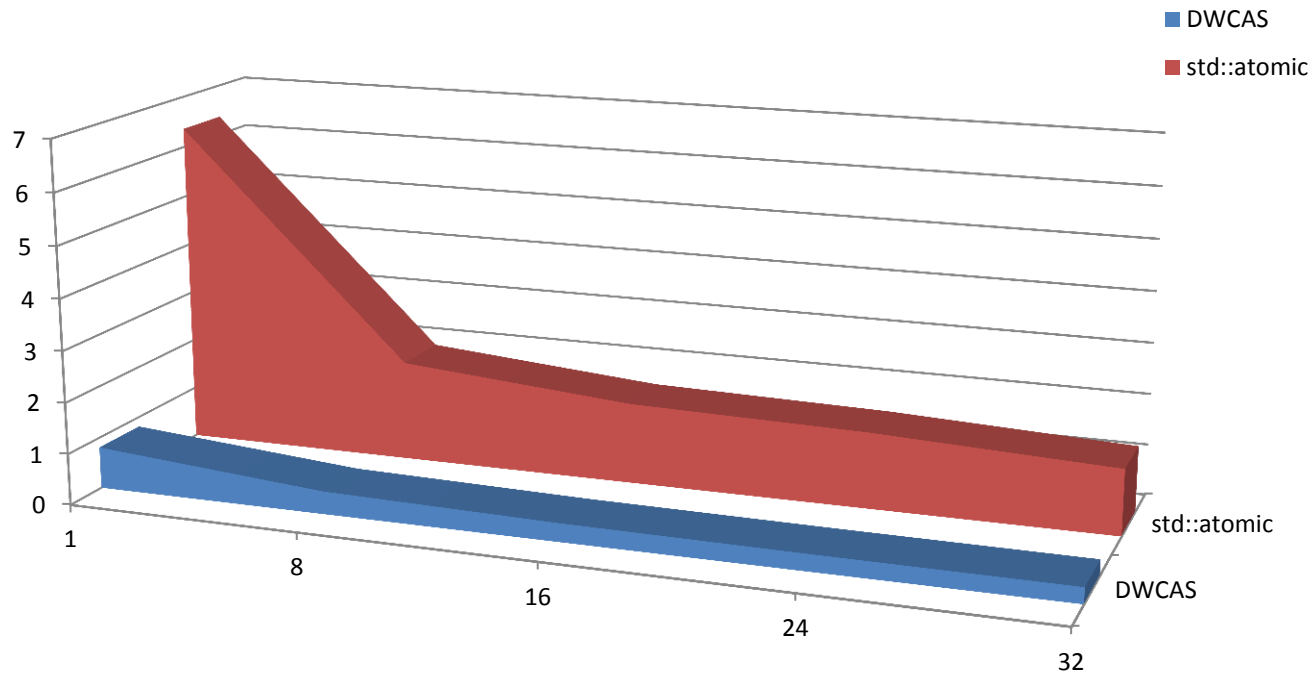
Wall-clock performance (in seconds)



Relative performance



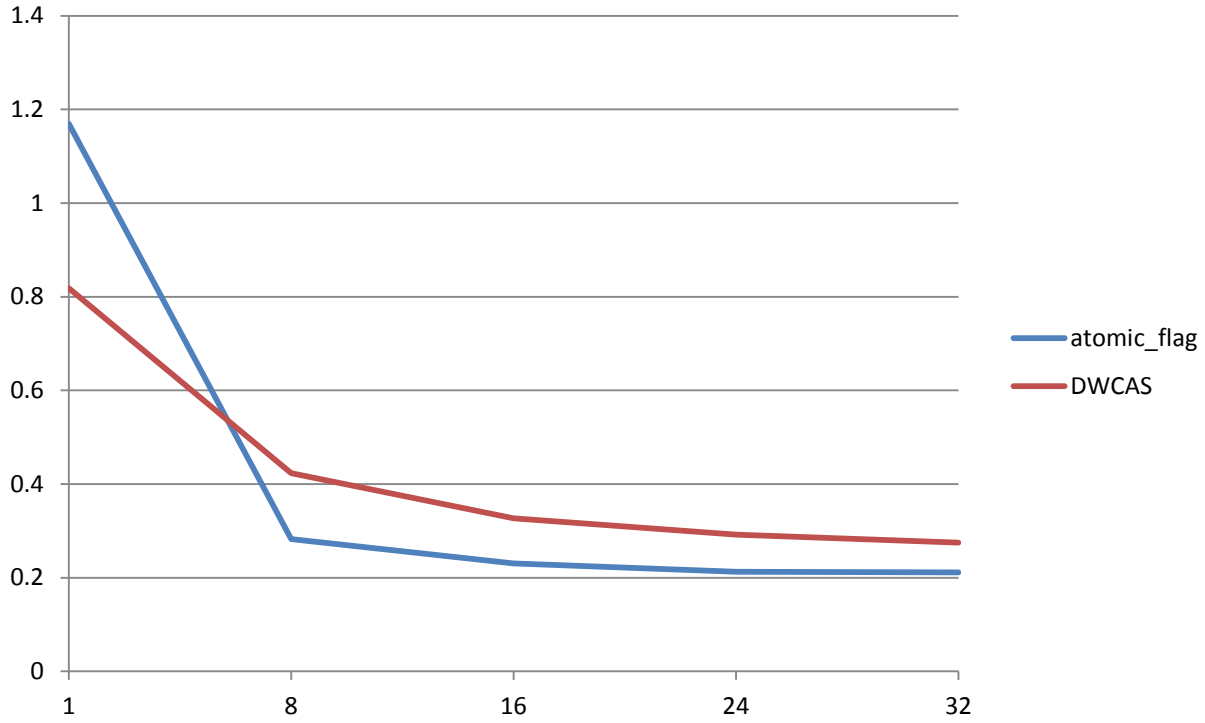
The Value of DWCAS



Benchmark #2: Mostly load()

- Hardware:
  - Xeon E3-1270, 3.5 Ghz
  - 4 cores, 8 hardware threads
  - 16 Gb RAM
- Benchmark
  - 8 worker threads
  - 1M iterations / thread
    - 50% `load()`
    - 5% random `store()`, `exchange()`, CAS operations
    - 45% random `shared_ptr<>` operations
  - Average of 10 runs at set levels of memory contention

Mostly load() performance



Conclusions

Differential Reference Counting

- It works
- Portable
- Mixed performance profile
- Additional complexity and memory overhead

Further Reading

- N4162: atomic<shared_ptr> proposal
- P0159R0: Draft ISO Concurrency TS
- cppreference.com
- Differential reference counting
- Anthony William's blog
- C++ Concurrency in Action