Atomic Counters

A Lesson on Performance and Hardware Concurrency

Detlef Vollmann

vollmann engineering gmbh, Luzern, Switzerland

ACCU Conference, Bristol, April 2015

Caveat

- This is expert level
  - but I'm not an expert
- Everything is relative
  - depends on hardware
  - depends on usage pattern
- Don't trust the example benchmark
  - "Churchill" benchmark
  - doctored to fit

Overview

Introduction

Stop the World

Uncertainity

Atomic Solutions

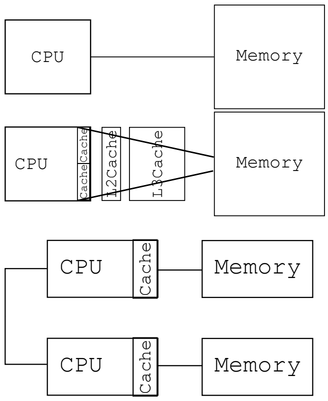
Distributed Solutions

Back to the Future

Wrap-Up

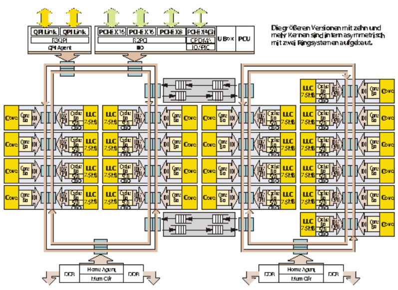
Hardware

- Memory bus
- Getting hierarchical
- Going multi-socket



Modern Hardware

- Modern CPU architectures blur the distinction between SMA and ccNUMA



Picture: Intel, copied from c't Magazin 21/14

- Concurrent access to same data from multiple core costs (a lot)

Counting Problem

- Counting events from different tasks
  - often to detect "hot spots"
- Inherently concurrent
  - really?

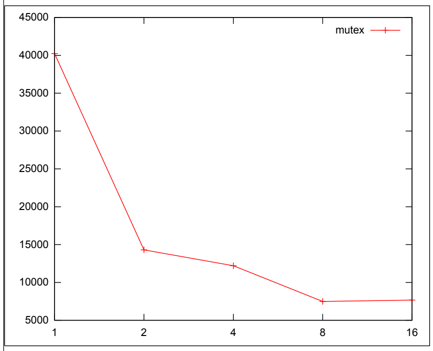
Example: Random Numbers

- Two related counters
  - high frequency
  - lower frequency
- Performance dominated by high frequency
- Very high contention

"Stop the World"

- Proper locking is the only correct solution with exact results

"Stop the World" Results



- Scales badly

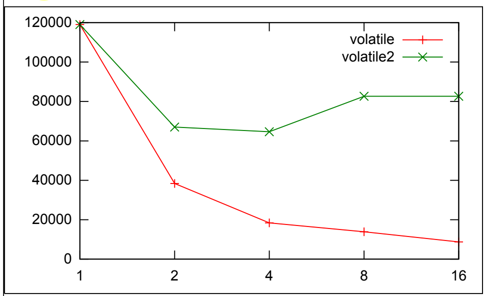
Giving Up Correctness

- Correct version too slow
- Decouple counters
- Possibly loose some counts
  - benign races?

Benign Races

- UB is UB!
- Only actual benign race is write only, same value
- For RMW, old nice CPUs, volatile and good look may help
  - for demo only, never for production!

Volatile Results




```
count: 30870350, matches: 13779698, ratio: 2.24028
```

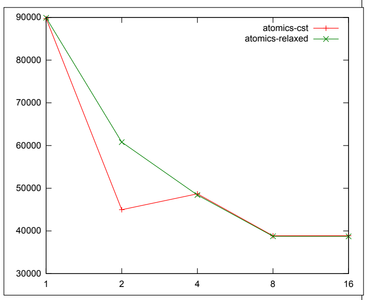
- With high contention, non-synchronized counters loose
- Definitely not worth the risk

Atomic Count

- Correctly synchronize (separately)
- Default atomics
  - sequentially consistent
- Relaxed atomics

Atomic Results

- No measurable difference
  - RMW requires ownership transfer
- Still no real scalability
  - Starvation



(Weak) Compare-And-Swap

- Atomic increment (fetch-add) is RMW
- CAS may be read-only (fail case)
  - no ownership transfer required
- CAS with backoff may reduce cache-line traffic
  - Dice,Lev,Moir "Scalable Statstics Counters"
  - NUMA node based
  - anti-starvation mechanism

Distributed Counters

- Giving up more precision
- Count locally
- Read globally
  - reducer, accumulator
- Proposal N3706 by Lawrence Crowl

```
template<typename Integral>
class bumper;
{
public:
    void operator +=(Integral by);
    void operator -=(Integral by);
    void operator ++();
    void operator --();
};
```


Simplex

```
template<typename Integral> class simplex
  : public bumper<Integral>
{
public:
    constexpr simplex();
    constexpr simplex(Integral in);
    simplex(const simplex&);
    simplex& operator=(const simplex&);
    Integral load();
    Integral exchange(Integral to);
};
```

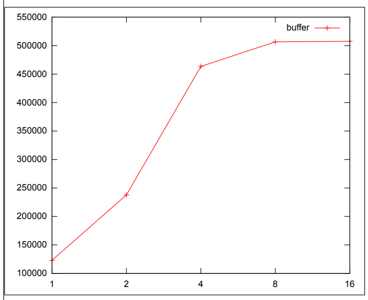
- Not distributed
- Provides interface for other counters as reducer
- Performance like atomics

Buffer

```
template<typename Integral> class buffer
  : public bumper<Integral>
{
    typedef bumper<Integral> prime_type;
public:
    buffer();
    buffer(prime_type& p);
    buffer(const buffer&);
    buffer& operator=(const buffer&);
    void push();
};
```

- Per-task counter
- Provides push (into simplex counter)
- Must push for count being seen

Buffer Results



- This looks good

Broker

```
template<typename Integral> class weak_broker
  : public bumper<Integral>
{
    typedef weak_duplex<Integral> duplex_type;
public:
    weak_broker();
    weak_broker(duplex_type& p);
    weak_broker(const weak_broker&);
    weak_broker& operator=(const weak_broker&);
};
```

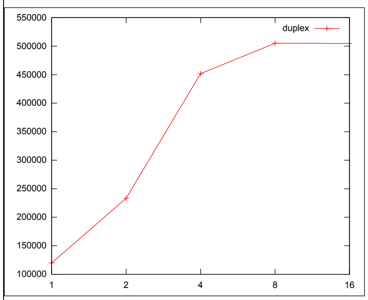
- provides (internal) query interface

Duplex

```
template<typename Integral> class weak_duplex
  : public bumper<Integral>
{
public:
    weak_duplex();
    weak_duplex(Integral in);
    weak_duplex(const weak_duplex&);
    weak_duplex& operator=(const weak_duplex&);
    Integral load();
};
```

- duplex provides the reduction
- provides exact count at time of reading
  - but no "stop the world"
  - no synchronization for related counters

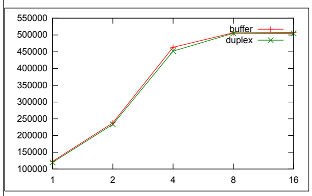
Broker Results



- This also looks good

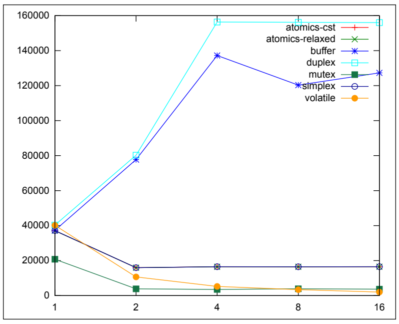
Push vs. Pull

- push only competes with other pushes
- pull competes with actual counts
  - but read only
- pull gives better precision
- performance depends on actual usage



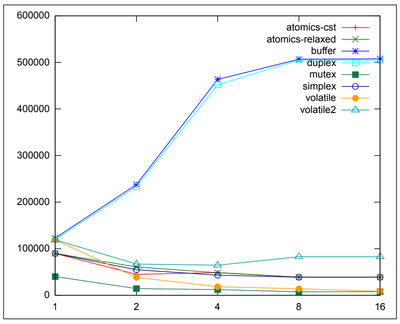
Results From a NUMA Machine

- Dual-core Opterons, dual-socket



Single Socket Machine Results

- Quad-core Haswell, plus hyperthreading



Morris' Counter

- Morris' algorithm provides a counter up to 130000 with 8 bit
  - looses precision
- Dice,Lev,Moir scale that to higher numbers
- Less bit changes mean less updates
- Scales pretty well

Overflow

- Long running counters will overflow
- N3706 provides `exchange` to drain buffers
- `strong_duplex` does this on pull
  - now a read/write operation

Distribution

- The answer to shared state
- Using local counters costs space
- Synchronization for reduction requires thought
- Avoid starvation on reduction
- Avoid false sharing for local counters

Conclusion

- Performant parallel counters are not atomic
- But if you don't want to stop the world, that's the simple truth
- Modifying shared memory is expensive
- Creative solutions may help
- General solution is distribution

References

- N3706, Lawrence Crowl, "C++ Distributed Counters" http://open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3706.html
- "Scalable Statstics Counters"
  Dave Dice, Yossi Lev, Mark Moir; SPAA'13, June 23-25, 2013, Montreal Quebec, Canada