Outline

Why does the code in java.util.concurrent look so different than that in an academic paper?

- Programming on layered, virtualized substrates
  - Performance anomalies are common
  - Libraries like j.u.c are “middle” members of substrate
  - Focus on Java, but many issues generalize
- Sample sets of problems and solutions
  - Adaptive waiting
  - Fine-grained memory access ordering
  - Memory placement and contention
  - Applications to bulk parallel operations
- Illustrate some general issues, some hacks
  - Performance impact can range from 1000X down to 1%
    - Usual… case

lots of stuff so this is a story about

Outline

Why does the code in java.util.concurrent look so different than that in an academic paper?

- Programming on layered, virtualized substrates
  - Performance anomalies are common
  - Libraries like j.u.c are “middle” members of substrate
  - Focus on Java, but many issues generalize
- Sample sets of problems and solutions
  - Adaptive waiting
  - Fine-grained memory access ordering
  - Memory placement and contention
  - Applications to bulk parallel operations
- Illustrate some general issues, some hacks
  - Performance impact can range from 1000X down to 1%
    - Usually don't know range in advance in any given case

Warmup: Sample Issues

Some questions without simple answers:

- How long does it take to block/unblock a thread?
- How long to re-read a variable inside a spin-wait?
- How to ensure you {always, never} re-read a field?
- How to minimize load → CAS window?
- How to isolate “thread local” variables?
- How to ensure that unused objects are reclaimable?
- How to ensure a method is compiled (not interpreted)

java.util.concurrent

- Atomic Variables
  - supporting compareAndSet (CAS), striped forms, etc
- Locks
  - including Conditions, ReadWriteLocks
- Synchronizers
  - Semaphores, barriers, etc
- Data Exchange
  - Queues, etc
- Concurrent Collections
  - Maps, Sets, Lists serving as shared resources
- Executor Frameworks
  - Tasks, Thread pools, Futures, work-stealing, completions...

there up as far as we can go without
without biasing

Layered, Virtualized Systems

Lines of source code make many transitions on their way down layers, each imposing unrelated-looking ...

- policies, heuristics, bookkeeping

... on that layer's representation of ...

- single instructions, sequences, flow graphs, threads

... and ...

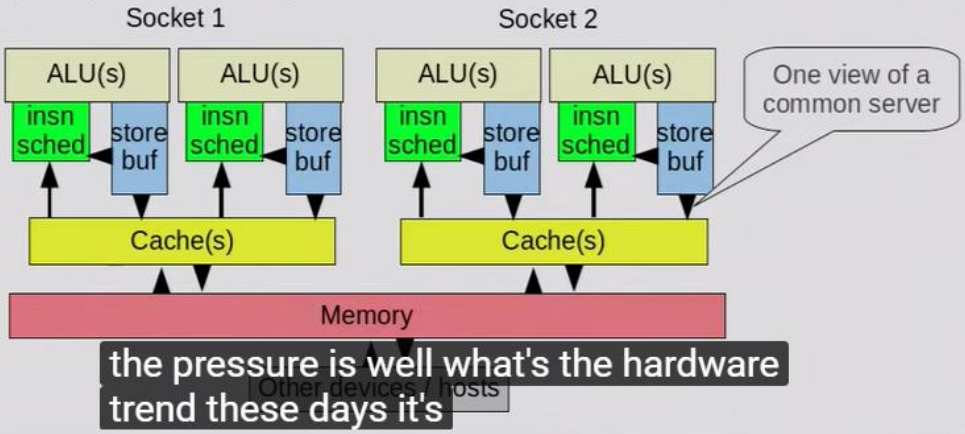
- variables, objects, aggregates

One resu
tier so we sit right in the middle of
all the actions there's
Poor me ffects of any line of code

Hardware Trends

Opportunistically parallelize anything and everything

- More gates → More parallel computation
  - Dedicated functional units, multicores
- More communication → More asynchrony
  - Async (out-of-order) instructions, memory, & IO



the pressure is well what's the hardware
trend these days it's

Interfaces

- Interface-like constructions hide implementations
  - Often get only one shot to define API at each layer
- Tension between **Over- and Under- Abstraction**
  - APIs impose consistency, composability constraints
  - More specialized APIs/components make fewer implementation tradeoffs
    - But require more specialized usages
    - Can lead to too many ways of doing almost the same thing
  - Example: j.u.c.Queue extends java.util.Collection
    - Improves learnability; simplifies most usages
    - But requires that all implementations somehow allow removal of elements not at head of queue
    - Arbitrary removal is an unnatural act for many queue implementations; can lead to "unrelated" methods

that pretty soon so the classic way to deal with this also what's

Some Sources of Anomalies

- **Fast-path / slow-path**
  - "Common" cases fast, others slow
  - Ex: Caches, hash-based, JITs, exceptions, net protocols
  - Anomalies: How common? How slow?
- **Lowering representations**
  - Translation need not preserve expected performance model
    - May lose higher-level constraints; use non-uniform emulations
  - Ex: Task dependencies, object invariants, pre/post conds
  - Anomalies: Dumb machine code, unnecessary checks, traps
- **Code between the lines**
  - Insert support for lower-layer into code stream
  - Ex: VMM code rewrite, GC safepoints, profiling, loading
  - Anomalies: [illegible] with user code

nobody can get their interfaces right so most common anomaly is fast

Leaks Across Layers

Higher layers may be able to influence policies and behaviors of lower layers

- Sometimes control is designed into layers
  - Components provide ways to alter policy or bypass mechanics
    - Sometimes with explicit APIs
    - Sometimes the "APIs" are coding idioms/patterns
    - Ideally, a matter of performance, not correctness
  - Underlying design issues are well-known
    - See e.g., Kiczales "open implementations" (1990s)
  - Leads to eat-your-own-dog-food development style
- More often, control arises by accident
  - Designers (defensibly) resist specifying or revealing too much
    - Sometimes even when "required" to do so (esp hypervisors)
  - Effective control becomes a black art
    - Fragile; unguaranteed byproducts of development history

last part

Theme: Data-Parallel Composition

Tiny map-reduce example: sum of squares on array

- Familiar sequential code/compilation/execution

```
s = 0; for (i=0; i<n; ++i) s += sqr(a[i]); return s;
```

... or ...

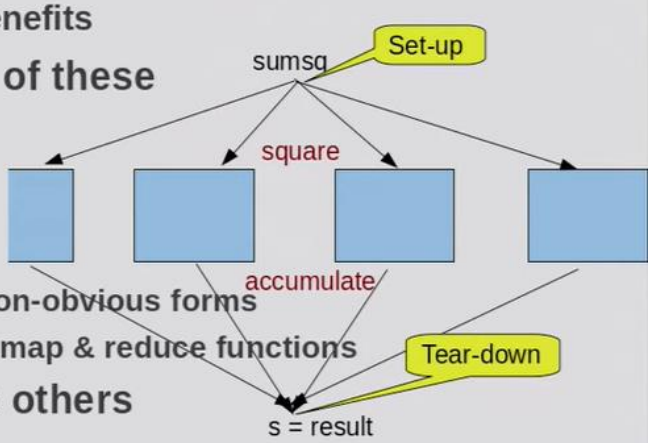
```
reduce(map(a, sqr), plus, 0);
```

  - May be superscalar even without explicit parallelism
- Parallel needs algorithm/policy selection, including:
  - Split work: Static? Dynamic? Affine? Race-checked?
  - Granularity: #cores vs task overhead vs memory/locality
  - Reduction: Tree joins? Async completions?
  - Substrate: Multicore? GPU? FPGA? Cluster?
- Results in families of code skeletons
  - Some of them can be even faster than sequential

okay I am gonna delve into some mechanics

Bulk Operations and Amdahl's Law

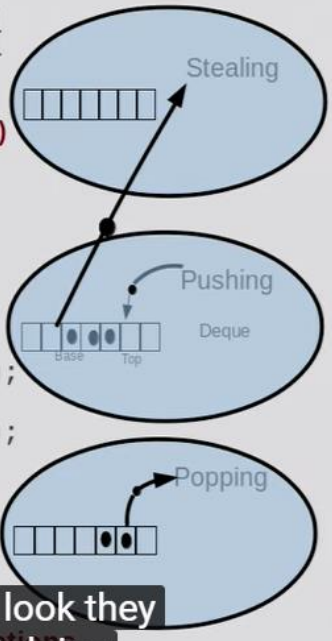
- Sequential set-up/tear-down limits speedup
  - Or as lost parallelism = (cost of seq steps) * #cores
  - Can easily outweigh benefits
- Can parallelize some of these
  - Recursive forks
  - Async Completions
  - Adaptive granularity
    - Best techniques take non-obvious forms
    - Some rely on nature of map & reduce functions
- Cheapen or eliminate others
  - Static optimization
    - Jamming/fusing across operations; locality enhancements
  - Share (concurrent) collections to avoid copy / merge



faster where the challenges there's a bunch of challenges one

Sample ForkJoin Sum Task

```
class SumSqTask extends RecursiveAction {
  final long[] a; final int l, h; long sum;
  SumSqTask(long[] array, int lo, int hi) {
    a = array; l = lo; h = hi;
  }
  // (One basic form; many improvements possible)
  protected void compute() {
    if (h - l < THRESHOLD) {
      for (int i = l; i < h; ++i)
        sum += a[i] * a[i];
    }
    else {
      int m = (l + h) >>> 1;
      SumSqTask rt = new SumSqTask(a, m, h);
      rt.fork(); // pushes task
      SumSqTask lt = new SumSqTask(a, l, m);
      lt.compute();
      rt.join(); // pops/runs or helps or waits
      sum = lt.sum + rt.sum;
    }
  }
}
```



Tediously similar code for many other bulk operations

detail so how do these things look they
you put them up so this is just doing

Case 1: Adaptive Waiting

Goal: When waiting for a change in a variable that is outside of local ability to modify, choose scheme to:

- Maximize **throughput**, and/or minimize **latencies** or their **variance** (fairness), and/or minimize **interference** (impact on other activities), and/or minimize **energy** requirements
- Impossible in general
  - Among main motivations for non-blocking approaches
    - But waiting often unavoidable
  - **Huge performance impact for some bad decisions**
    - The same issues arise at multiple layers → 1000X variance
- Classic push/pull tradeoffs
  - Spin polling for change, vs "do nothing" until signalled
  - Best-pra[illegible] push
    - j.u.c po[illegible]ays eventually do so (no unbounded spins)

say I can't do anything to those joins
but it's

Spinning

Obvious form:

```
volatile boolean cond; …
while (!cond) { /* skip */ }
```

- Some problems:
  - Processor may detect as an idle-loop and power down core
  - Processor may combine/elide instructions that overwhelm memory controller, so act superscalar. Or the opposite.
  - Memory system may be saturated, slowing down other threads, especially if cache line shared with other variables
  - JVM may insert execution counter and/or safepoint check into loop → more coherence traffic
  - OS and/or VMM may context-switch out thread, thus blocking when user code expects spinning
  - No transition mechanics to blocking forms

amazingly

Blocking

Obviously wrong form:

```
T1: if (!cond) suspend();
T2: cond = true; t1.resume();
```

- Each layer must deal with suspend-resume race
  - Usually, internal semaphore-like mechanics
    - Java intrinsics: LockSupport.{park, unpark}
    - Park returns immediately if semaphore set, so a near no-op
      - Sometimes set "accidentally", so calls must use `while`, not `if`
- Each layer feels free to pile on bookkeeping
  - Reasoning: "the thread isn't doing anything useful anyway"
  - Even for signalling/resumption (ignoring Amdahl's law)
- Sometimes leads to fall-off-cliff discontinuities
  - Can be hundreds of thousands of cycles to block/unblock

hold that thought oh

Transition Policies

Goal: Spin a while, then block

- Common ideas/approaches:
  - Backoffs: Do “nothing” for increasing periods before block
    - If “nothing” != spinning, entails blocking + OS timer wakeups
      - Timer granularity too coarse for many uses
  - Bound the number of cycles possibly wasted spinning, to guarantee energy/interference bounds
    - Decision is arbitrary, but users want a “good” choice
  - Rate Predictive: If average wait time < block+unblock time, then only block if (sufficiently) past average
    - Entails statistical decision mechanics & randomization
- Information required for implementing these (and others) spans layers, and/or is unreliable
  - Requires check-then-act decisions based on stale data

discontinuities so what do you do you you had three

Spin Implementations

- No single implementation in j.u.c
  - Some mechanics shared in AbstractQueuedSynchronizer
  - Many instead use specializations of common patterns
    - Sometimes inconsistently; only improved when updated
- Some common ingredients
  - Avoid loops of form `b: load cond; branchIfZero b;`
    - Usually achieve more uniform rate with less uniform branches
    - If nothing better, generate simple random numbers inside loop – often Marsaglia XorShift (cheap and good enough)
      - Test some bits to randomize control paths
  - Add one or two voluntary context switches (Thread.yield)
    - A rough approximation to early blocking if others runnable
    - May avoid some of the heavier blocking mechanics
    - Thread.yield spec is very weak, so this is very heuristic

within an order of magnitude it's gonna be okay so if

Spin → Block Thresholds

- Estimation
  - Static: Find threshold associated with max target wastage on common machines for highly contended use cases
    - Pick compromise value good enough across platforms
  - Dynamic: Adjust static estimate to improve throughput by tracking loop counts/times
    - But adjustments tend to stabilize at max value
      - Not often worthwhile compared to instrumentation overhead
- Semantics-dependent add-ons, including
  - Queues: Block sooner if head of queue already blocked
  - Allow “barging” for lock-like sync
    - Let another polling thread get lock instead of the one signalled
      - Signallee then restarts entire process with lower threshold
    - Improves throughput, minimizes time lock not held but wanted
    - But weakens fairness (latency variance)

break branch prediction to it it works
it works amazingly well

Case 2: Memory Access Ordering

- Java (also C++, C) Memory Model for locks
  - Sequentially Consistent (SC) for data-race-free programs
    - A requirement for implementations of locks and synchronizers
- Java volatiles (and default C++ atomics) also SC
  - Generate fence instructions and/or compiler constraints
    - (Some) can be CAS'ed, and used to implement locks
- Interactions with non-volatile accesses complicated
  - First approximation of reordering rules:

1st/2nd	Plain load	Plain store	Volatile load	Volatile store
Plain load				NO
Plain store				NO
Volatile load	NO	NO	NO	NO
Volatile store		NO	NO	NO

all the other guys okay so that was a
little vignette one of

  - Plus interactions with dependencies, causality, etc

Relaxed Atomics

- Provide intermediate ordering control
  - May greatly reduce overhead in performance-critical code
    - But must consider r/w before and after library method call
      - One approach is to view in terms of invalidations (next slide)
  - Even more complex interactions with plain & volatile access
- C++ standardized access modes (acquire, release, ...)
- Java: Unstandardized JVM “internal” intrinsics
  - Ideally, a bytecode for each mode of (load, store, CAS)
    - Would fit with No L-values (addresses) Java rules
  - Instead, intrinsics take object + field offset arguments
    - Establish on class initialization, then use in `Unsafe` API calls
    - Non-public; truly “unsafe” since offset args can't be checked
      - Can be used outside of JDK using odd hacks if no security mgr
      - j.u.c supplies public wrappers that interpose (slow) checks

Publication and Transfers

Weaker protocols avoid more invalidation

```
Class X { int field; X(int f) { field = f; } }
```

- For shared var v (other vars are locals):

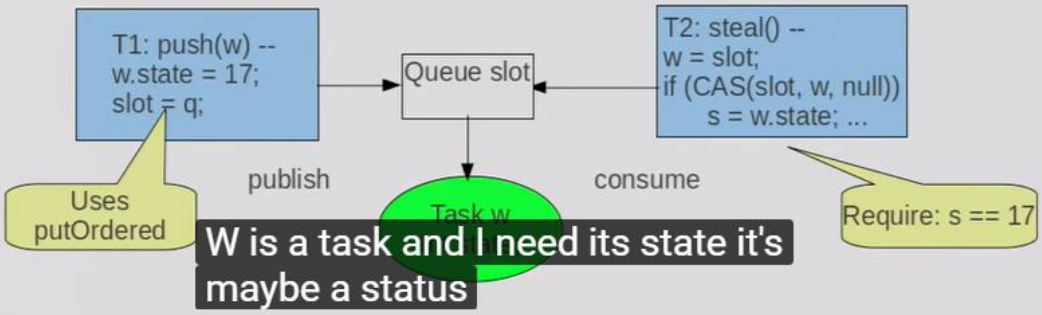
```
P: p.field = e; v = p;  ||
C:              c = v; f = c.field;
```

- Use weakest construction still ensuring that C:f is usable, considering (among other issues):
  - “Usable” can be algorithm- and API-dependent
    - Consistency with reads/writes of other shared vars?
  - Is write to v final (i.e., the last-ever write)? including:
    - Write Once (null → x), Consume Once (x → null)
  - Is write to x.field final?
    - Is there a unique uninitialized value for field?

well it's this it says I want to shove this thing in

Example: Transferring Tasks

- Work-stealing Queues perform ownership transfer
  - Push: make task available for stealing or popping
    - Needs lightweight store-fence
  - Pop, steal: make task unavailable to others, then run
    - Needs CAS with at least acquire-mode fence



Documenting Consistency Properties

Example: ForkJoinTask.fork API spec

"Arranges to asynchronously execute this task. While it is not necessarily enforced, it is a usage error to fork a task more than once unless it has completed and been reinitialized. Subsequent modifications to the state of this task or any data it operates on are not necessarily consistently observable by any thread other than the one executing it unless preceded by a call to join() or related methods, or a call to isDone() returning true."

- The no-refork rule ultimately reflects internal relaxed consistency mechanics based on ownership transfer
  - The mechanics leverage fact that refork before completion doesn't make sense anyway
- The inconsistent-until-join rule reflects arbitrary state of, e.g., the elements of an array while it is being sorted
  - Also enables weaker ordering (more parallelism) while running
- Would be nicer to statically enforce
  - Secretly, the no-refork rule cannot now be dynamically enforced

sure

Composition and Consistency

- Consistency policies are intrinsic to systems with multiple readers or multicast (so: part of API design)
- Most consistency properties do not compose
  - IRIW Example: vars x,y initially 0 → events x, y unseen
    - Activity A: send x = 1; // (multicast send)
    - Activity B: send y = 1;
    - Activity C: receive x; receive y; // see x=1, y=0
    - Activity D: receive y; receive x; // see y=1, x=0 ? Not if SC
- For vars, can guarantee sequential consistency
  - JMM: declare x, y as volatile
- Doesn't necessarily extend to component operations
  - e.g., if x, y are two maps, & the r/w operations are put/get(k)
- Doesn't [illegible]
  - Even for fault-tolerant systems (CAP theorem)

all right I invite you to find this sometime or jot

Optimization and Ordering

- Orderings inhibit common compiler optimizations
  - Inhibiting wrong ones may also inhibit those you want
- Requires "manual" dataflow optimizations
  - Manually hoisting reads, exception & indexing checks, etc
  - Manually inlining to avoid call opaqueness effects
    - JIT inlining rules usually not too smart here
  - Can be challenging to express in source code
    - Some resort to other intrinsics to bypass redundant checks
- Use odd loop forms to better position safepoints etc
  - Example: increment x.field by:

```
int c;
do {} while (!CAS(x, fieldOffset, c = x.field, c+1));
```

    - Usually better than other forms because field read more likely to be adja[illegible]

all this sort of weak ordering oh no the compiler doesn't understand what

Coping with *Idiot Savant* Compilers

A sampling of tiny, sometimes transient, issues:

- Dispatching/Inlining
  - Recasting code to save a few bytecodes in front-end compiled form improves chances of inlining
    - So, e.g., using Assert statements may impede inlining
  - Inlining only fast paths to funnel slower dispatch to single virtual call points
- Avoiding or exploiting Invisible code
  - Rely on default-zero/null initialization
  - Leverage cases where null-checks are required anyway
    - Null is almost always the best "special" value to check
  - Minimize boxing, class-loading, synthetic access methods, etc
- Compilation Plans
  - Forcing code warmups combats oblivousness to Amdahl's law for sequential portions
  - Avoiding decompilation, "rare" traps, JNI, and other slow stuff

Jets is that you know sometimes they're idiots

Determinism á la carte

- Common components entail algorithmic randomness
  - Hashing, skip lists, crypto, numerics, etc
    - Fun fact: The Mark I (1949) had hw random number generator
  - Visible effects; e.g., on collection traversal order
    - API specs do not promise deterministic traversal order
      - Bugs when users don't accommodate
  - Randomness more widespread in concurrent components
    - Adaptive contention reduction, work-stealing, etc
- Plus non-determinism from multiple threads
  - Visible effects interact with consistency policies
- Main problem across all cases is bug reproducibility
  - A design tradeoff across languages, libraries, and tools
  - Non-deterministic performance bugs exist independently

code is compiled can matter it does um alright