Convergence in Concurrency

Doug Lea
SUNY Oswego
Introduction

Motivation

Infrastructure and middleware development evolves from ...

- Make something that works ... to ...
- Make it faster ... to ...
- Make it more predictable

Encounter issues seen in real-time systems
- Can we apply lessons learned in one to the other?

Outline

Present three problem areas, invite discussions
- Avoid GC!  – Controlling allocation and layout
- Avoid blocking! – Memory models, async designs
- Avoid virtualization! – Coping with uncertainty
Concurrent Systems

- Typical system: many mostly-independent inputs; a mix of streaming and stateful processing
- QoS goals similar to RT systems
  - Minimize drops and long latency tails
  - But less willing to trade off throughput and overhead

Diagram:
- Decode
- Process data parallel
- Combine
- Shared state
1. Memory Management

- GC can be ill-suited for stream-like processing:
  - **Repeat**: Allocate → read → process → forget

- RTSJ Scoped memory
  - Overhead, run-time exceptions (vs static assurance)

- Off-heap memory
  - Direct-allocated ByteBuffers hold data
    - Emulation of data structures inside byte buffers
  - Manual storage management (pooling etc)
  - Manual synchronization control
  - Manual marshalling/unmarshalling/layout
    - Project Panama will enable declarative layout control

- Alternatives?
Memory Placement

Memory contention, false-sharing, NUMA, etc can have huge impact

- Reduce parallel progress to memory system rates
  - JDK8 @sun.misc.Contended allows pointwise manual tweaks
  - Some GC mechanics worsen impact; esp card marks
  - When writing a reference, JVM also writes a bit/byte in a table indicating that one or more objects in its address range (often 512bytes wide) may need GC scanning
  - The card table can become highly contended
    - Yang et al (ISMM 2012) report 378X slowdown

JVMs cannot allow precise object placement control

- But can support custom layouts of plain bits (struct-like)
  - JEP for Value-types (Valhalla) + Panama address most cases?
- JVMs oblivious to higher-level locality constraints
  - Including “ThreadLocal”!
2. Blocking

- The cause of many high-variance slowdowns
  - More cores → more slowdowns and more variance
    - Blocking Garbage Collection accentuates impact

- Reducing blocking
  - Help perform prerequisite action rather than waiting for it
  - Use finer-grained sync to decrease likelihood of blocking
  - Use finer-grained actions, transforming ...
    - From: Block existing actions until they can continue
    - To: Trigger new actions when they are enabled

- Seen at instruction, data structure, task, IO levels
  - Lead to new JVM, language, library challenges
    - Memory models, non-blocking algorithms, IO APIs
Opportunistically parallelize anything and everything

- More gates $\rightarrow$ More parallel computation
- Dedicated functional units, multicores
- More async communication $\rightarrow$ More variance
- Out-of-order instructions, memory, & IO

One view of a common server
Parallelizing Expressions

- Exploits available ALU-level parallelism
- Indistinguishable from sequential evaluation in single-threaded user programs
Parallel Evaluation inside CPUs

- Overcome problem that instructions are in sequential stream, not parallel dag

- Dependency-based execution
  - Fetch instructions as far ahead as possible
  - Complete instructions when inputs are ready (from memory reads or ops) and outputs are available
    - Use a hardware-based simplification of dataflow analysis

- Doesn't always apply to multithreaded code
  - Dependency analysis is shallow, local
  - What if another processor modifies a variable accessed in an instruction?
  - What if a write to a variable serves to release a lock?
Shallow Dependencies

- Assumes current core owns inputs & outputs
- Not always true in concurrent programs
- Special instructions (fences etc) are needed to enforce non-local ordering constraints
- The main reason we need Memory Models
Programmers must explicitly disable unordered instruction executions not already covered by as-if-locally-sequential rules.

Stronger processors (sparc, x86) partially automate by suppressing most violations possibly visible across threads (TSO: all except visible Store → Load reordering).

Weaker processors (ARM, POWER) do not.

Compilers also reorder to reduce stalls (plus other reasons).

Processors support fences and/or special r/w instructions or modes that disable reorderings.

Details & performance annoyingly differ across processors.

Among hardest and messiest parts of formal memory models is characterizing effects of not using them.

Many weird cases; e.g., happens-before cycles.
Main JSR-133 Memory Rules

- **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**
  - Load has same ordering rules as lock; store same as unlock
- **Interactions with plain non-volatile accesses**
  - Prevent, e.g., accesses in lock bodies from moving out
- **First approximation of reordering rules:**

<table>
<thead>
<tr>
<th>1st/2nd</th>
<th>Plain load</th>
<th>Plain store</th>
<th>Volatile load</th>
<th>Volatile store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain load</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Plain store</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Volatile load</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Volatile store</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>
Enhanced Volatiles (and Atomics)

- Support extended atomic access primitives
  - CompareAndSet (CAS), getAndSet, getAndAdd, ...
- Provide intermediate ordering control
  - May significantly improve performance
    - Reducing fences also narrows CAS windows, reducing retries
  - Useful in some common constructions
    - Publish (release) → acquire
      - No need for StoreLoad fence if only owner may modify
    - Create (once) → use
      - No need for LoadLoad fence on use because of intrinsic dependency when dereferencing a fresh pointer
- Interactions with plain access can be surprising
  - Most usage is idiomatic, limited to known patterns
  - Resulting program need not be sequentially consistent
Expressing Atomics

- C++/C11: standardized access methods and modes
- Java: JVM “internal” intrinsics and wrappers
  - Not specified in JSR-133 memory model, even though some were introduced internally in same release (JDK5)
  - 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
- JEP 188 and 193 (targeting JDK9) will provide first-class specs, and improved APIs
- Should be equally useful in RTSJ
### Example: Transferring Tasks

- **Work-stealing Queues** perform ownership transfer
  - **Push**: make task available for stealing or popping
    - Needs release fence (weaker, thus faster than full volatile)
  - **Pop, steal**: make task unavailable to others, then run
    - Needs CAS with at least acquire-mode

T1: push(w) --
\[ w.\text{state} = 17; \]
\[ \text{slot} = q; \]

T2: steal() --
\[ w = \text{slot}; \]
\[ \text{if} \ (\text{CAS(slot, } w, \text{ null)}) \]
\[ s = w.\text{state}; \ldots \]

- **Require**: \( s == 17 \)
Example: ConcurrentLinkedQueue

- **Extend Michael & Scott Queue (PODC 1996)**
  - CASes on different vars (head, tail) for put vs poll
  - If CAS of tail from t to x on put fails, others try to help
    - By checking consistency during put or take
  - Restart at head on seeing self-link

Poll

```
Poll
```

Put x

```
1: CAS head from h to n
```

```
2: self-link h (relaxed store)
```

```
1: CAS t.next from null to x
```

```
2: CAS tail from t to x
```

```
head tail
```

```
h n t
```

```
x
```
Efficient Ordering Control

- Orderings inhibit common compiler optimizations
  - Inhibiting wrong ones may also inhibit those you want
  - A byproduct of coarse-grained JMM modes/rules
- Can overcome with manual dataflow-like tweaks
  - Hoisting reads, exception & indexing checks, etc
  - Manual inlining to avoid call opaqueness effects
  - Resort to unsafe intrinsics to bypass redundant checks
- Efficient concurrent Java code looks a lot like efficient concurrent C11 code
  - Encapsulate in libraries whenever possible
**IO**

- Long-standing design and API tradeoff:
  - Blocking: suspend current thread awaiting IO (or sync)
  - Completions: Arrange IO and a completion (callback) action

- Neither always best in practice
  - Blocking often preferable on uniprocessors if OS/VM must reschedule anyway
  - Completions can be dynamically composed and executed
    - But require overhead to represent actions (not just stack-frame)
    - And internal policies and management to run async completions on threads. (How many OS threads? Etc)
  - Some components only work in one mode

- Ideally support both when applicable
  - Completion-based support problematic in pre-JDK8 Java
    - Unstructured APIs lead to “callback hell”
Java.util.concurrent Futures hit similar tradeoffs

Completion support hindered by expressibility
- Initially skirted “callback hell” by not supporting any callbacks. But led to incompatible 3\textsuperscript{rd} party frameworks

JDK8 lambdas and functional interfaces enabled introduction of CompletableFutures (CF)

CF supports fluent dynamic composition
CompletableFuture.supplyAsync(()\rightarrow generateStuff()).thenApply(stuff\rightarrow reduce(stuff)).thenApplyAsync(x\rightarrow f(x)).thenAccept(result\rightarrow print(result)); // add .join() to wait

Plus methods for ANDed, ORed, and flattened combinations
- In principle, CF alone suffices to write any concurrent program

Not fully integrated with JDK IO and synchronization APIs
- Adaptors usually easy to write but hard to standardize
- Tools/languages could translate into CFs (as in C# async/await)
3. 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
- variables, objects, aggregates

![](chart)

Each may be internal layered

- Poor predictability of the effects of any line of code
  - Need to know what to look for to cope with anomalies
  - (More details in SPAA 2012 and Philly ETE 2013 talks)
Some Sources of Anomalies

- **Fast-path / slow-path**
  - “Common” cases fast, others slow
  - Ex: Caches, hash-based, exceptions, net protocols
  - **Anomalies:** How common? How slow?

- **Hot / cold**
  - Ex: power management, thread-core mappings, JITs
  - **Anomalies:** slow thread startup, uneven throughput

- **Lowering representations**
  - Translation loses higher-level constraints
  - 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:** Unanticipated interactions with user code
Randomization

- Common components inject 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
- Can be even more useful in concurrency
  - Fight async and system non-determinism with algorithmic non-determinism
    - Hashed striping, backoffs, work-stealing, etc
  - Implicit hope that central limit theorem applies
    - Combining many allegedly random effects → lower variance
    - Often appears to work, but almost never provably
      - Formal intractability is an impediment for some real-time use
Summary

- Full performance determinism is a lost cause on general-purpose platforms
  - Cannot reliably predict properties of fully implemented component using a given design / algorithm
  - Hard-real-time increasingly isolated to custom hardware
- But unpredictability can often be reduced in practice
  - Also usually improving throughput
  - Using ideas from both real-time and non-real-time
  - Need to lift more design and programming techniques from black-art to everyday constructions
Backup slides follow