Concurrent Java

Java Programming in a Multicore World

Angelika Langer

Trainer/Consultant

http://www.AngelikaLanger.com/

objective

- take look at current trends in concurrent programming
- explain the Java Memory Model
- discuss future trends such as lock-free programming and transactional memory

speaker's relationship to topic

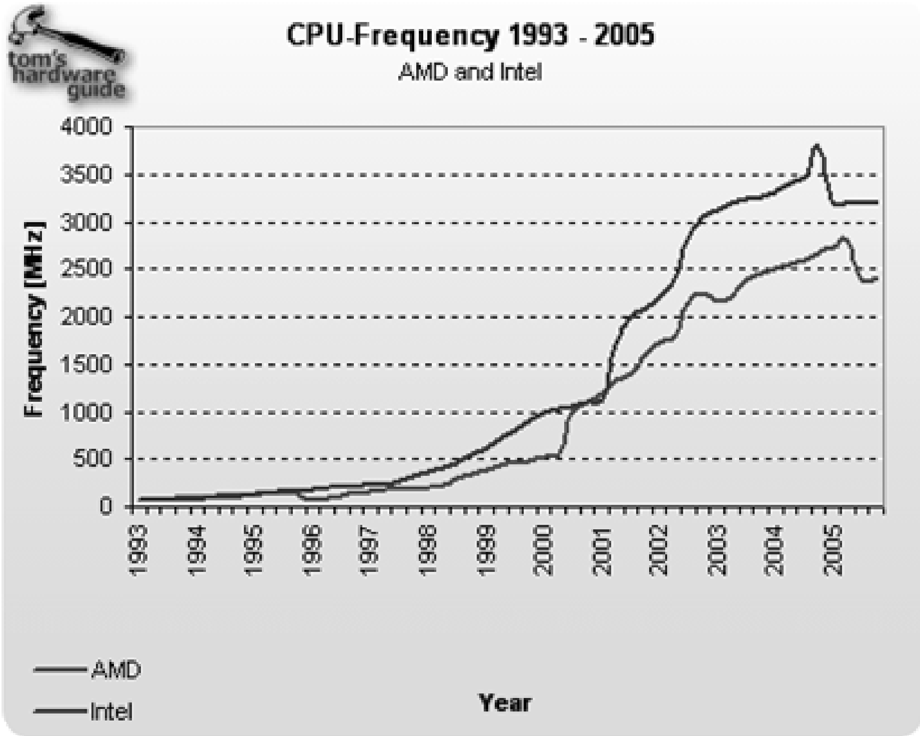
- independent trainer / consultant / author
  - teaching C++ and Java for 10+ years
  - curriculum of a dozen challenging courses
  - co-author of "Effective Java" column
  - author of Java Generics FAQ online
  - Java champion since 2005

agenda

- **history of concurrency & concurrency trends**
- synchronization and memory model
- fight the serialization – improve scalability
- future trends

CPU development

- Moore's law:
  number of transistors doubles every two years
  - since 2004: more cores
  - until 2004: faster ones
  - main reason: heat
- 2 cores became standard 2007
  - 6-12 in 2009 (AMD)
- more complex caches
  - hierarchy



CPU development implies

- new CPU will not solve your performance problems
  - if your program does not scale (well) to multiple cores
  - i.e.: find (and fight) the serialization

- existing programs
  - undetected errors might pop up
  - multi-core + caching uncovers synchronization problems

- Java environment
  - more and more complex work for
    - the byte code compiler, and
    - the JIT compiler

Java history – initial MT support

- mostly built into the language (not into the library)
  - `synchronized` block/method – lock in every object
  - `Object.wait()`, `Object.notify()` – condition in every object
  - `...`
- mainly low level functionality
  - no thread pool, no blocking queue, …
- memory model
  - chapter 17 of the Java Language Specification: Threads and Locks
    - hard to understand,
    - incomplete,
    - violated by JVM implementations

Java history – JDK 5.0 MT support

- rework of existing locks and conditions
  - into the library: `java.util.concurrent.locks`
  - extended functionality
    - timeout for existing locks
    - new locks: read-write-lock
  - approach changed: library is more flexible than language
    - think of C
- high-level abstractions
  - thread pool: `ThreadPoolExecuter`, …
  - synchronizers: `BlockingQueue`, `CyclicBarrier`, …
  - support for asynchronous programming: `Future`, …

(cont.)

- support for lock free programming
  - low-level
    - abstractions from `java.util.concurrent.atomic`
  - high-level
    - 'concurrent' collections: `ConcurrentHashMap`, …

- reworked memory model
  - cleared up what `volatile` and `final` mean in a MT context
  - defines requirements regarding atomicity, visibility and ordering of operations

Java keeps up to …

- … the needs and requirements of the changing MT uses
- more people build MT programs
  - MT patterns and idioms become common knowledge
  - need for high-level abstractions
- more people build Java MT programs for multiprocessor platforms
  - need for clear and exact memory model
  - wish for better scaling MT abstractions
    - need for lock free programming
- former niche becomes main stream with multi core CPUs

architecture history – mid 90ies

- no or insufficient support of threads
  - on many proprietary unices thread implementation in the user space:
    - blocking `read()` in one thread blocked all threads of the process
  - Java started with a similar model: green threads
    - non-preemptive, sometimes obscure behavior

- threads used to structure programs – not to achieve more through put
  - scalability was not an issue, multi-processor systems were rare
    - multi-core unknown
  - user space threads scalability limited with multi-processors

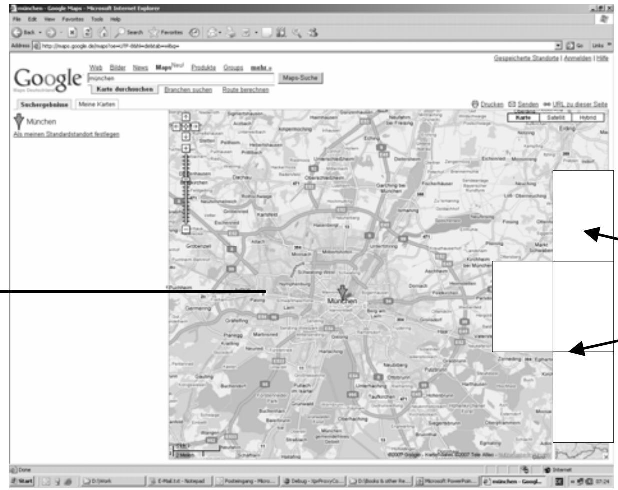
architecture history – since then

- trend to asynchronous and parallel computing
  to increase throughput

- Java examples
  - asynchronous I/O
    - 1.4 socket, 5.0 sockets + SSL, 7.0 sockets + file system
    - essential: frees you from one thread per socket
    - but: program structure gets more complex and technical
  - JMS introduced 2001
    - much later than RMI which was part of Java from the beginning
    - effect: EJB became *message-driven beans*

(cont.)

- general example:
  - AJAX (**A**synchronous **J**avaScript **a**nd **X**ML)
  - means: user interaction decoupled from HTTP requests
  - traditionally
    - you select a link / push a button / etc. , and
    - a new page gets loaded into your browser
  - AJAX example: Google Maps



user interaction

e.g. pull the map to the left

map elements are asynchronously pulled from the server via Javascript

more AJAX

- Google Maps
  - user interaction decoupled from HTTP requests

- more asynchronicity
  - HTTP push via Ajax
    - signal an asynchronous event in the browser
    - e.g. incoming telephone call

- alternatives:
  - Christian Gross: Ajax Pattern and Best Practices,
    chapter 8: Persistent Communication Pattern
  - Alex Russell: Comet – Low Latency Data for the Browser
    - http://alex.dojotoolkit.org/?p=545

(cont.)

- both solutions boil down to
  a 'long-lived' HTTP request from the browser

- persistent communication / long polling / hybrid polling:
  - request lives, until the event occurs
    - or a 'long' timeout occurs (5-10 minutes)
  - event is signaled in the response
    - or the timeout
  - new request to poll the next event

- comet style / HTTP streaming:
  - request lives, until the client goes away
  - all data is send from the server to the client in the same response

a small problem

- traditional servlet programming
  - one thread:
    - receives HTTP request
    - determines what has to be done
    - gathers the data (and renders the new page)
    - sends all this back to the client in a response
- what about an long-lived open HTTP request ?
  - that waits for an external event
    - e.g. the incoming telephone call
- allocates a thread until
  the event occurs / client goes away !
  - with 50000 users on the server !

asynchronous web servers

- decouple the request from the response
- Jetty 6 Continuation
    - `Continuation.suspend()`, `Continuation.resume()`
    - http://docs.codehaus.org/display/JETTY/Continuations
- Tomcat 6.0
    - Comet module allows to process I/O asynchronously
    - http://tomcat.apache.org/tomcat-6.0-doc/aio.html
- Java standard for asynchronous web server
    - JSR 315 = Servlet 3.0 specification
    - scheduled finish by the end of 2008
- underlying concept: asynchronous I/O

more and more asynchronicity

- not only web server – other servers too
- SOA (**s**ervice **o**riented **a**rchitecture)
  - service -> service -> service …
  - you don't want to have a waiting thread in each of the server
  - i.e.
    - asynchronous handling of the request
    - MOM (message oriented middleware), means often JMS in Java
- all this means:
  - you need multiple threads and some synchronization of these to tie the external asynchronous channels to your program

agenda

- history of concurrency & concurrency trends
- synchronization and memory model
- fight the serialization – improve scalability
- future trends

motivation - why does JMM matter?

- JMM = **J**ava Memory **M**odel

- understanding JMM reveals errors in existing programs
  - undetected errors might pop up
  - multi-core + caching uncovers synchronization problems

Java Memory Model (JMM)

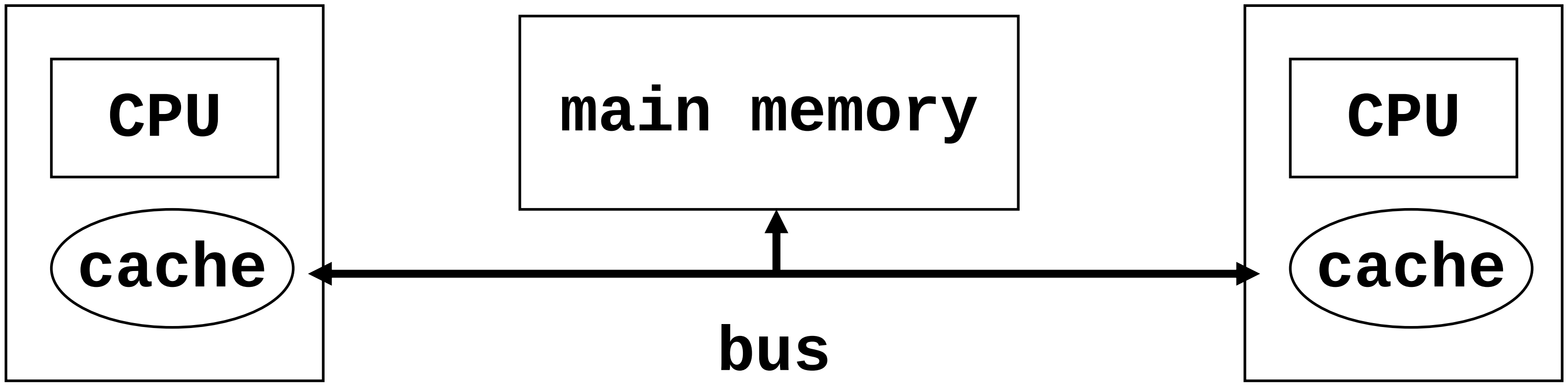
- specifies minimal guarantees given by the JVM
  - about when writes to variables become visible to other threads
- is an abstraction on top of hardware memory models

Java Memory Model
- threads read and write to variables

Hardware Memory Model
- processors read and write to caches, registers, main memory

Java memory model

- JMM resembles abstract SMP (symmetric multi processing) machine
- key ideas:
  - all threads share the main memory
  - each thread uses a local working memory
  - flushing or refreshing working memory to/from main memory must comply to JMM rules



Java memory model

JMM rules address 3 intertwined issues:

- atomicity
  - which operations must have indivisible effects ?
- visibility
  - under which conditions are the effects of operations taken by one thread visible to other threads ?
- ordering
  - under which conditions can the effects of operations appear out of order to any given thread ?

"operations" means:

- reads and writes to memory cells representing Java variables

JMM in practice

- examples:

atomicity

- access to variables of primitive type (except `long`/`double`) are atomic
- execution of operations in a synchronized block is atomic

visibility

- values written to a volatile variable are visible to other threads

ordering

- effects of operations in a synchronized block appear in order
- accesses to volatile variables appear in order

sequential consistency

- ***sequential consistency*** is a convenient (yet unrealistic) mental model:
  - imagine a single order of execution of all programm operations (regardless of the processors used)
  - each read of a variable will see the last write in the execution order

- JMM does NOT guarantee sequential consistency
  - reordering is generally permitted
  - specific rules for synchronization, thread begin/end, volatile and final variables

hardware memory models

- JVM maps JMM to hardware memory model

- in shared-memory multiprocessor architectures:
  - each processor (or processor core) has its own cache (or even several layers of caches)
  - cache is periodically reconciled with main memory
  - cache strategies vary among architectures

=> *hardware memory model*

barriers and fences

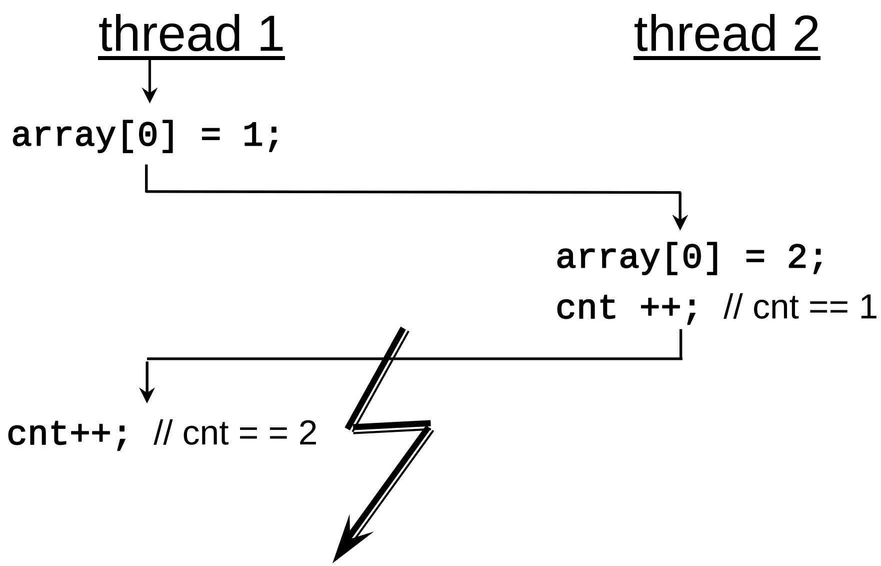
- JVM must use special instructions for memory coordination (called *memory barriers* or *fences*)
  - to shield developers from hardware differences
  - to implement the JMM rules

agenda

- history of concurrency & concurrency trends
- **synchronization and memory model**
  - **atomicity**
  - visibility
  - ordering
- fight the serialization – improve scalability
- future trends

need for atomicity

- non-atomic operations are a problem in case of race conditions
  - interleaved access to shared resources
    where at least one access is a modification



atomicity guarantees

- explicit synchronization
  - execution of operations in a synchronized block is atomic
  - same for operations between acquisition / release of explicit lock
- unsynchronized field access
  - access to primitive type (except `long`/`double`) is atomic
  - access to references is atomic (does *not* include access to object)
  - access to volatile variables (including `long`/`double`) is atomic
  - access to atomic variables is atomic
- common misconception
  - atomicity means we get the most recent value - wrong!
  - atomic access to a variable just means:
    - we will *not* get some jumble of bits

agenda

- history of concurrency & concurrency trends
- synchronization and memory model
  - atomicity
  - **visibility**
  - ordering
- fight the serialization – improve scalability
- future trends

need for visibility

```
private int[] array;
private int cnt = 0;          ←——— must be volatile
...
public synchronized void push(int elm) { array[cnt++] = elm; }
public synchronized int  pop()         { return(array[--cnt]); }
public              int  size()        { return cnt; }
...
```

- access to `cnt` is atomic
  - no synchronization in `size()` needed
- visibility problem
  - writes performed in one thread need not be visible to other threads
  - i.e. modification of `cnt` in `push()`/`pop()` need not be visible to `size()`
- `volatile` is needed not for atomicity, but for visibility

visibility guarantees

- changes made in one thread are guaranteed to be visible to other threads under the following conditions:
  - explicit synchronization
  - thread start and termination
  - read / write of volatiles
  - first read of finals

visibility guarantee: read / write of volatiles

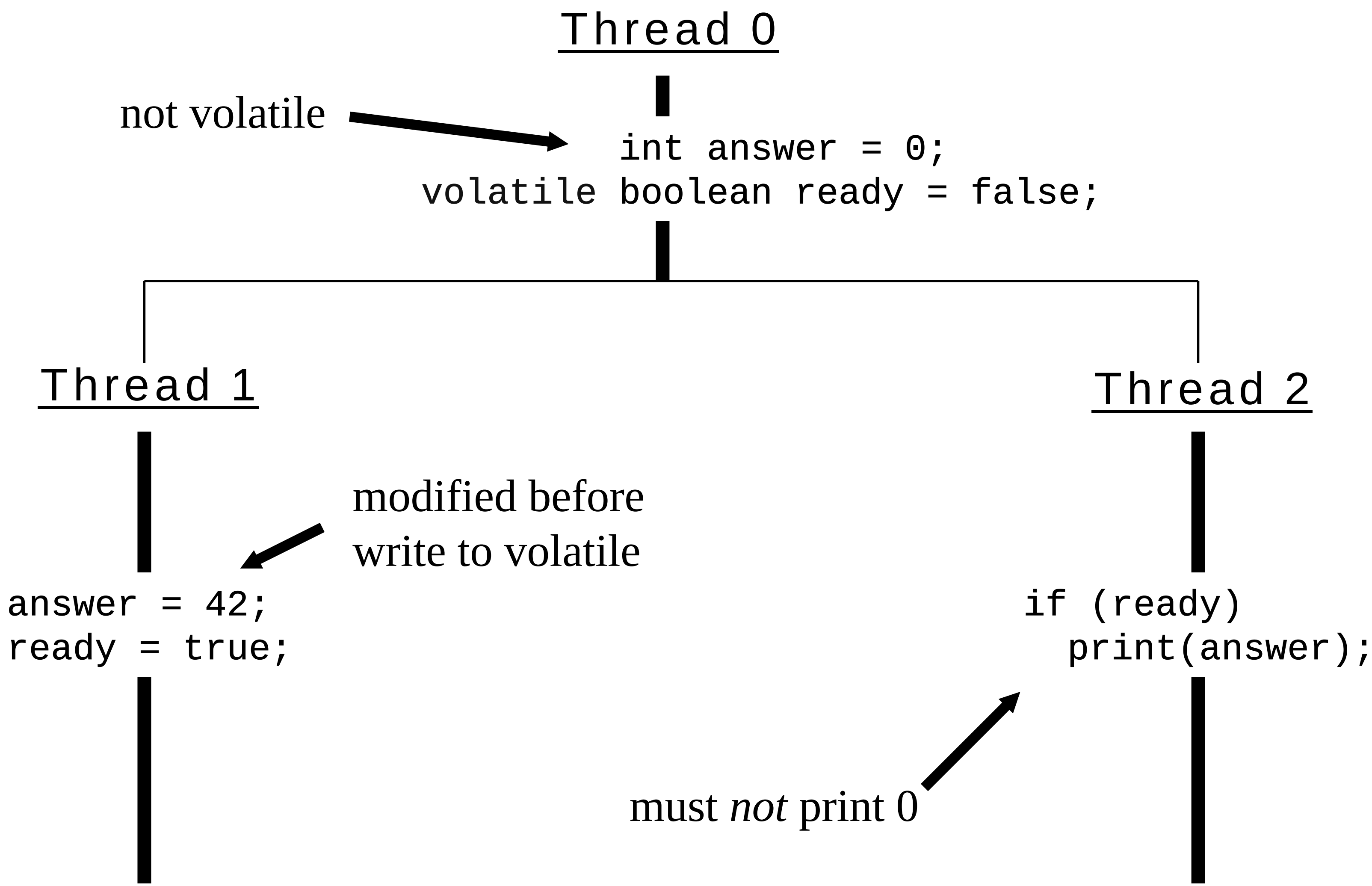
- reading a volatile forces a reload from main memory
- writing to a volatile forces a flush to main memory

- matches our expectation
  – when a thread reads a volatile, then all writes are visible that any other thread performed prior to a write to the same volatile

- how about volatile references ... ?

- volatile is *not* transitive
  – read/write of a volatile reference affects the reference, but not the referenced object (or array)

volatile (since Java 5) - example



non-transitive volatile - example

Thread 0

```
volatile Name name = null;
```

Thread 1

reference modified before write to fields

```
name = new Name();
name.setFirst("Eva");
name.setLast("Schulz");
```

Thread 2

```
if (name != null)
  print(name);
```

might see empty object

volatile references

- what do we do to also make the modified object visible?
  - make all fields of referenced object volatile
    - problem for arrays: array elements cannot be declared volatile
  - modify elements before assignment to volatile reference
    - all changes made prior to writing to the volatile are flushed
  - use explicit synchronization
    - viable fallback, at the expense of synchronization overhead

agenda

- history of concurrency & concurrency trends
- synchronization and memory model
  - atomicity
  - visibility
  - **ordering**
- fight the serialization – improve scalability
- future trends

need for ordering

- ordering is closely related to visibility

- JMM is specified in terms of *actions*
  - e.g. reads and writes to variables, locks and unlocks of monitors, starting and joining threads
- JMM defines *"happens-before"* rules
  - partial ordering on actions
  - if there is no happens-before ordering between two operations the JVM is free to reorder them
- miconception: "happens-after"
  - there is no "happens-after" rule
  - e.g. an action after a synchronized block can happen before or in the critical section

No!

```
public class PossibleReordering {
  private static int x = 0, y = 0;
  private static int a = 0, b = 0;

  public static void main(String[] args) {
    Thread one = new Thread(new Runnable() {
      public void run() { a = 1; x = b; }
    });
    Thread two = new Thread(new Runnable() {
      public void run() { b = 1; y = a; }
    });
    one.start(); two.start(); one.join(); two.join();
    System.out.println("x="+x+",y="+y);
  }
}
```

- incorrect, due to possible reordering
  - result is unpredictable; even x=0 and y=0 can happen

reordering - example

Thread 0

```
x = y = 0;
a = b = 0;
```

Thread 1

```
a = 1;
x = b;
```

Thread 2

```
b = 1;
y = a;
```

Can even result in x=0 and y=0 !
(as a result of reordering)

ordering

- ordering rules have two aspects:
  - within-thread
    - thread performing actions in a method perceives instructions in normal as-if-serial order
  - between-thread
    - other threads 'spying' on this thread by concurrently running unsynchronized methods might perceive instructions in arbitrary order

ordering guarantees

- ordering of synchronized blocks is preserved
- ordering of read/write of volatile fields is preserved
- ordering of initialization/access to final fields is preserved

- matches our expectation
  - actions in one synchronized block happen (i.e. effects become visible) *before* another thread acquires the same lock
  - effect of writing to a volatile is visible to all subsequent reads
  - all threads will see the correct values of final fields that were set by the constructor

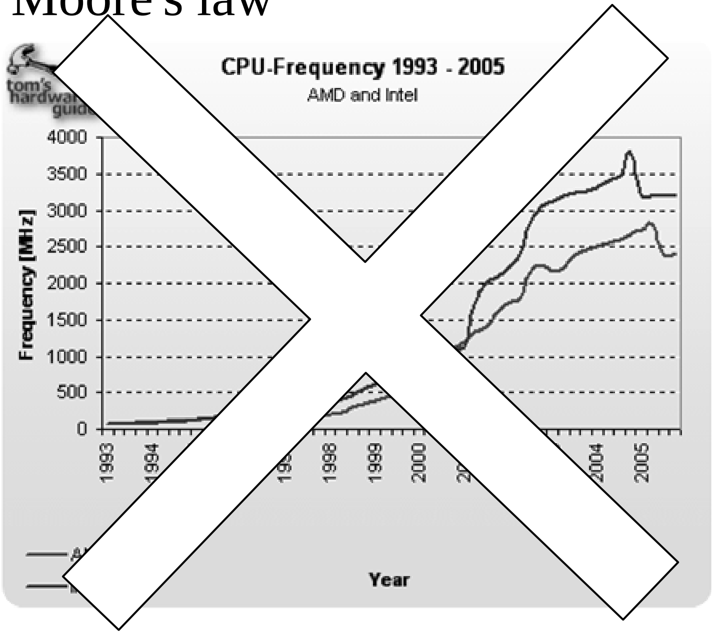
agenda

- history of concurrency & concurrency trends
- synchronization and memory model
- **fight the serialization – improve scalability**
- future trends

performance wise

- cannot buy a faster CPU to speed up the program
  - or hope for a faster CPU six/twelve month from now
    - when you program feels slow during development
- software must be designed so that
  - it can take advantage of the additional cores / CPUs
  - can scale with additional cores / CPUs

Moore's law



Amdahl’s law

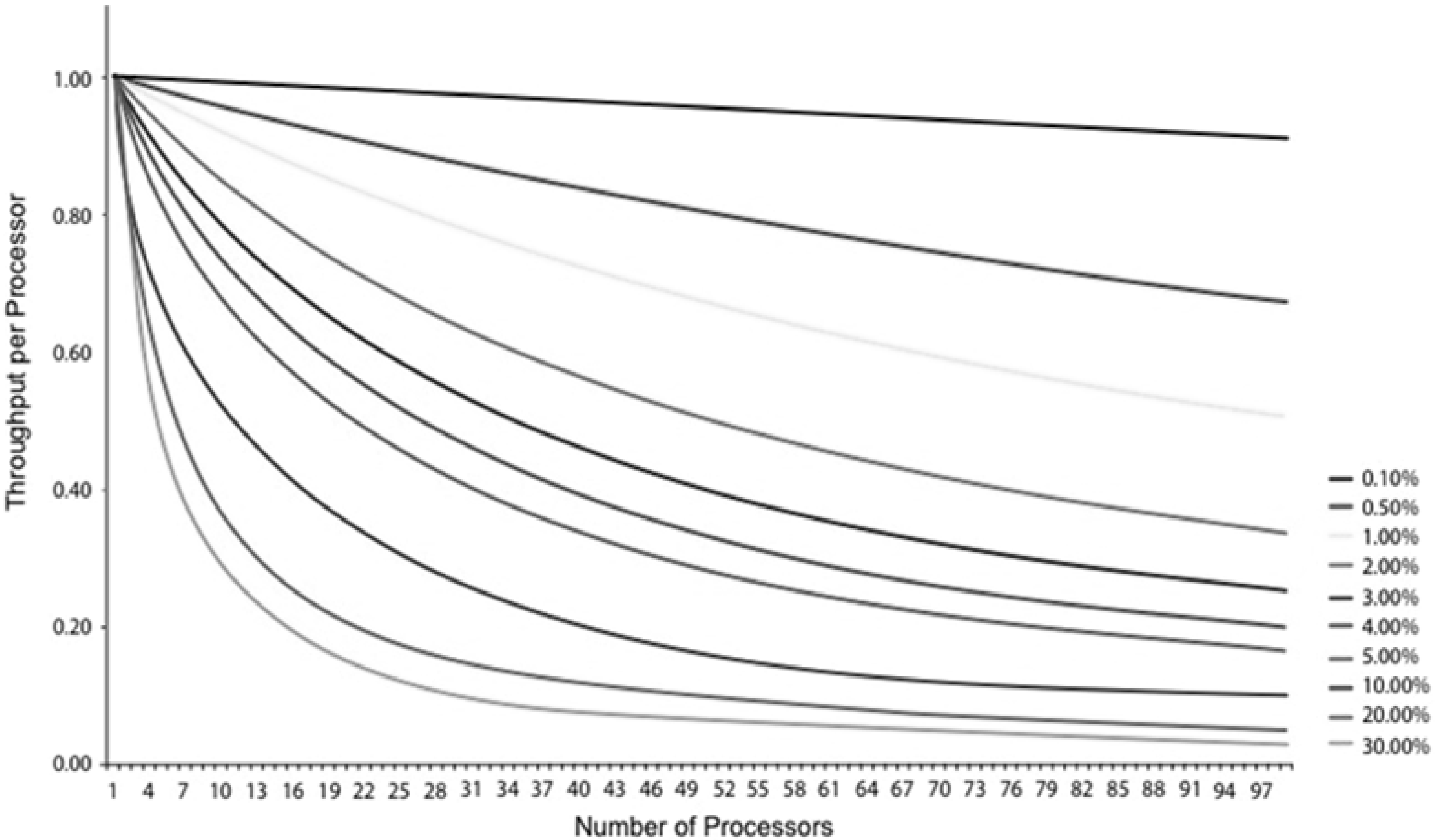
- named after computer architect Gene Amdahl
  - "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Conference Proceedings, (30), pp. 483-485, 1967.
  - Gene Amdahl has approved the use of his complete text in the Usenet comp.sys.super news group FAQ which goes out on the 20th of each month
- used in parallel computing to predict the theoretical maximum speedup using multiple processors

(cont.)

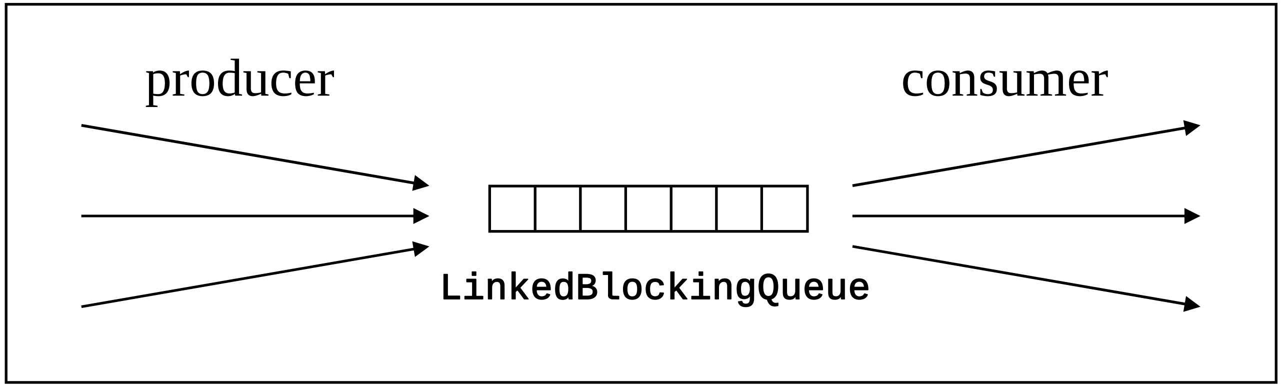
- idea: divide work into **serial** and **parallel** portions
  - serial work cannot be sped up by adding resources
  - parallelizable work can
- Amdahl's Law: $\texttt{speedup} \le \dfrac{1}{\left(F + \dfrac{(1 - F)}{N}\right)}$
  - F is the fraction that must be serialized
  - N is the number of CPUs
- with N **->** ∞**,** speedup **->** 1/F
  - with 50% serialization,
    - your program can only speed up by a factor of 2 (with: ∞ CPUs)
- naïve idea: from 1 to 2 CPUs = factor of 2 ?

(cont.)

- fight serialization to improve performance



example



- looks highly parallelizable
  – (if producers are slow increase their thread pool)

- 0% serialized ?
  – no!
    - need synchronization to maintain the queue's integrity

LinkedBlockingQueue.offer()

```
public boolean offer(E o) {
    if (o == null) throw new NullPointerException();
    final AtomicInteger count = this.count;
    if (count.get() == capacity)
        return false;
    int c = -1;
    final ReentrantLock putLock = this.putLock;
    putLock.lock();
    try {
        if (count.get() < capacity) {
            insert(o);
            c = count.getAndIncrement();
            if (c + 1 < capacity)
                notFull.signal();
        }
    } finally {
        putLock.unlock();
    }
    if (c == 0)
        signalNotEmpty();
    return c >= 0;
}
```

(cont.)

- Doug Lea did an excellent job with the implementation
  - highly optimized
    - split lock: put / take
    - `count` guarded lock-free
    - stack-local variables to speed up the execution inside the critical region
    - …
- structural problem
  - serialization of offering threads (producers)
  - similar serialization of getting threads (consumers)

serialization

- where/when threads demand concurrent access
- often hidden
  - in frameworks / third party abstractions
- other area: asynchronous service architecture
  - example: `java.nio.channels.Selector`
    - section on concurrency in the respective JavaDoc
  - management to send back the result asynchronously
    - Jetty continuation

fight the serialization …

… try to reduce lock induced serialization

- smallest critical region possible
  - `synchronized` block vs. `synchronized` method
    - or use explicit locks
  - speed up execution inside the critical region
  - replace synchronized counters with `AtomicInteger`

- lock splitting / striping
  - guard different state with different locks
  - reduces likelihood of lock contention

fight the serialization …

… try to eliminate locking entirely

- replace mutable objects with immutable ones
- replace shared objects with thread-local ones
  - e.g. make a copy before passing it to a concurrent thread

- lock-free programming

agenda

- history of concurrency & concurrency trends
- synchronization and memory model
- fight the serialization – improve scalability
- **future trends**

trends

- lock-free programming
  - supported in Java since JDK 5.0
    - `java.util.concurrent.atomic`, and
    - `Concurrent` collections in `java.util.concurrent`
- transactional memory
  - neither supported in Java nor in any popular programming language at the moment
- commonality
  - avoid locking to avoid serialization

agenda

- history of concurrency & concurrency trends
- synchronization and memory model
- fight the serialization – improve scalability
- future trends
  - **lock free programming**
  - transactional memory

CAS

- modern processors have a primitive called ***c**ompare-**a**nd-**s**wap*, or CAS
- a CAS operation includes three operands
  – a memory location
  – the expected old value
  – a new value
- the processor will atomically update the location to the new value
  – if the value that is there matches the expected old value
  – otherwise it will do nothing
  – it returns the value that was at that location prior to the CAS instruction

CAS permits atomic read-modify-write

- CAS allows an algorithm to execute a read-modify-write sequence
  - without fear of another thread modifying the variable in the meantime
  - if another thread did modify the variable, the CAS would detect it (and fail)
  - and the algorithm could retry the operation

- CAS-like operation are available in JDK 5.0 as "atomic variables“
  - based on the underlying system/hardware/CPU support
  - `java.util.concurrent.atomic`

example - thread-safe counter

```
public class SafeCounter {
 private volatile int value;
 public              int getValue()  { return   value; }
 public synchronized int increment() { return ++value; }
 public synchronized int decrement() { return --value; }
}
```

- `increment()` / `decrement()` are read-modify-write operations and must be atomic
  - atomic read-modify-write cannot be achieved by making instance variable volatile
  - need to be synchronized
- `get()` without synchronization, since **`value`** is **`volatile`**

example - thread-safe counter – lock free

```
public class AtomicCounter {
  private AtomicInteger value;
  public int getValue() { return value.get(); }
  public int increment() {
   int oldValue = value.get();
   while (!value.compareAndSet(oldValue,
                               oldValue + 1))
     oldValue = value.get();
   return oldValue + 1;
  }
  public int decrement() {
   int oldValue = value.get();
   while (!value.compareAndSet(oldValue,
                               oldValue - 1))
     oldValue = value.get();
   return oldValue - 1;
  }
}
```

is atomic

lock-free

- advantages
  - fast (~ 4 times faster than best locks)
  - deadlock immunity
  - …
- disadvantages
  - hard to program !!!
    - no simple straight forward approach as with locks
  - …

hard to program, but what you can do

- some strategies
  - e.g. lock-free counter, ABA problem, …
  - no single best resource of information known
    - best to search the web for ‘lock free programming’

- algorithms for standard data structures
  - map, linked list, …
  - `Concurrent` collections from `java.util.concurrent`
  - use these in your program
    - or these in combination with locks

agenda

- history of concurrency & concurrency trends
- synchronization and memory model
- fight the serialization – improve scalability
- future trends
  - lock free programming
  - **transactional memory**

transactional memory …

- … or **s**oftware **t**ransactional **m**emory (STM)

- similar to optimistic strategies in database transactions
  - e.g. *optimistic locking pattern for EJBs*

strategy

- thread does modifications to shared memory/object
  - without regards what the other threads are doing
- finished modifications
  - commit
    - verification that no other thread made concurrent modifications
  - abort and rollback
    - concurrent modifications occurred
    - error handling: (in most cases) retry of the transactions
- increased concurrency vs. overhead of retrying transactions that failed

conceptual pros

- very intuitive, e.g. update an object in shared memory
  - close to the original Java approach
    - object = monitor, (`public`) mutating methods are synchronized

- language integration proposal
  - Tim Harris and Keir Fraser: *Language Support for Lightweight Transactions*
  - `http://citeseer.ist.psu.edu/harris03language.html`

```
public void addName(String name) {
  atomic {
    nameCount++;
    nameList.add(name);
  }
}
```

conceptual cons

- abort and rollback
  - implies that you can undo every operation
  - what about those that are not memory based ?
    - e.g. unbufferd I/O
  - solution possibilities
    - add a buffer that can at least hold the changes made in the transaction
    - explicit undo operation

want to try STM ?

- no popular programming language supports STM
  - at the moment
- use a more experimental language
  - e.g. Clojure
    - dynamic programming language, Lisp dialect
    - compiles to JVM bytecode
    - "… *general-purpose language, combining the approachability and interactive development of a scripting language with an efficient and robust infrastructure for multithreaded programming* …"
    - `http://clojure.sourceforge.net/`
  - there are others too (not compiling to the JVM)

wrap-up

- a trend towards concurrent, asynchronous computing
  - MT initally for better structure
  - today to overcome synchronicity (messaging, AJAX, ...)
- multicore architecture might reveal yet undetected bugs
  - due to memory model issues (atomicity, visibility, ordering)
- multicore architectures need scalable software to be useful
  - avoid serialization - increase concurrency - Amdahl's law
- a gaze into the crystal ball
  - lock-free programming is already in use (by experts)
  - transactional memory might ease concurrent programming some time in the future

authors

Angelika Langer

Training & Consulting

Klaus Kreft

SEN Group, Munich, Germany

http://www.angelikalanger.com/Forms/Contact.html

Q & A