AT&T
Network Design and Performance Analysis Consulting

Understanding and Analyzing Java Performance

Tutorial - MASCOTS 2001

Varsha Mainkar

Network Design and Performance Analysis Department, AT&T Labs, New Jersey

www.att.com/networkandperformance

July 27, 2001

Acknowledgements

- P. Reeser
- A. Karasaridis
- K. Futamura
- M. Hosseini-Nasab
- R. Farel
- K. Meier-Hellstern
- D. Lynch
- Tony Hansen
- D. Cura
- W. Ehrlich
- A. Avritzer
- A. Bondi

Presenters at the Java Performance Seminar Series - implicit contributors to this tutorial.

Outline

- **Introduction to Java (10 mts)**
- **Motivation for studying Java Performance (15 mts)**
- **Overview of Java Architecture (30 mts)**
- **Break (5 mts)**
- **Impact of Java Architecture on Performance (45 mts)**
- **Analyzing Java Programs (15 mts)**

AT&T Network Design and Performance Analysis Consulting

Background

Java : A Brief History

- **Appeared in 1993**
- **Initially developed for**
  - **Networked, handheld devices**
- **Coincided with emergence of WWW**
  - **Target market shifted to Internet**
- **First showcased in the form of *applets***

...Java- A Brief History

- Server-side Java applets or Servlets introduced as demand increased for dynamic page generation
- Java Beans - for reusable software components
- Java Server Pages - for decoupling dynamic data from HTML presentation
- "Java 2" - Java 1.3
  - HotSpot Compiler, enhancements

Java - Features

- Platform - independence
- Security
- Robustness
- Network mobility
- Multi-threaded
- Built - in memory management
- Rich API for Internet and Web Programming

Why Java ?

- Faster, less troublesome development
- Easy porting to multiple platforms
- Easier software distribution
- Security features
- Rich APIs (Internet, Web,…)

API = Application Programmer's Interface

Where's the catch ?

- Performance !
- Generally true : rich programming features come at the cost of performance. Solutions:
  - Do not use rich environments ✗
  - Understand the environment and do "enlightened" development √

AT&T Network Design and Performance Analysis Consulting

Other Disadvantages

- Buggy Virtual Machines
- "Write once - Debug Everywhere"
- Platform Independence → "independence" from useful OS features

Bottom Line: Java has become extremely popular, equally among new programmers as well as seasoned C++ gurus.

Motivation

Is Java “slow”?

- Simple Example: Java vs. C++
  - Java with “Just-in-time” compilation and without

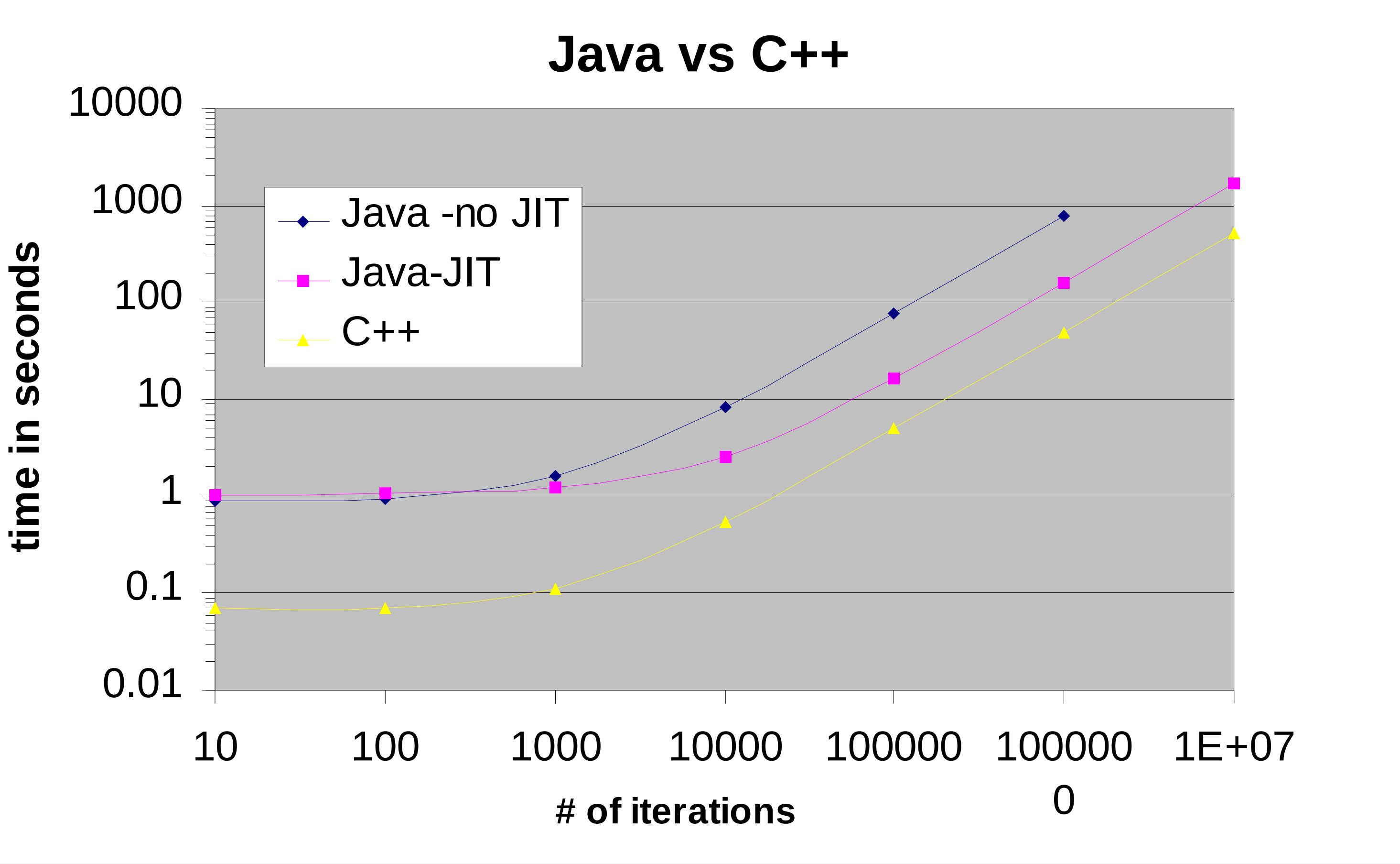
```
class Salutation {

   private static final String hello = "Hello, world!";
   private static final String greeting = "Greetings,
planet!";
   private static final String salutation = "Salutations,
orb!";

   private static int choice;

   public static void main(String[] args) {

      int i;

      for (i = 0; i <= 10000; i ++) {

        choice = (int) (Math.random() * 2.99);
        String s = hello;
        if (choice == 1) {
            s = greeting;
        }
        else if (choice == 2) {
            s = salutation;
        }
        System.out.println(s);
      }
   }

}
```

Java

```
#include<stdio.h>
#include<iostream.h>
#include<stdlib.h>
#include<string.h>

void main () {

  char* hello = "Hello World!";
  char* greeting = "Greetings, Planet!";
  char* salutation = "Salutation, Orb!";
  char* s;

  int choice;

  int i;

  for (i =0; i <= 10000; i ++) {
    choice = ((int) rand()) % 3;
    s = hello;
    if (choice == 1) {
      s = greeting;
    }
    else if (choice == 2) {
      s = salutation;
    }
    cout << s << endl;
  }

}
```

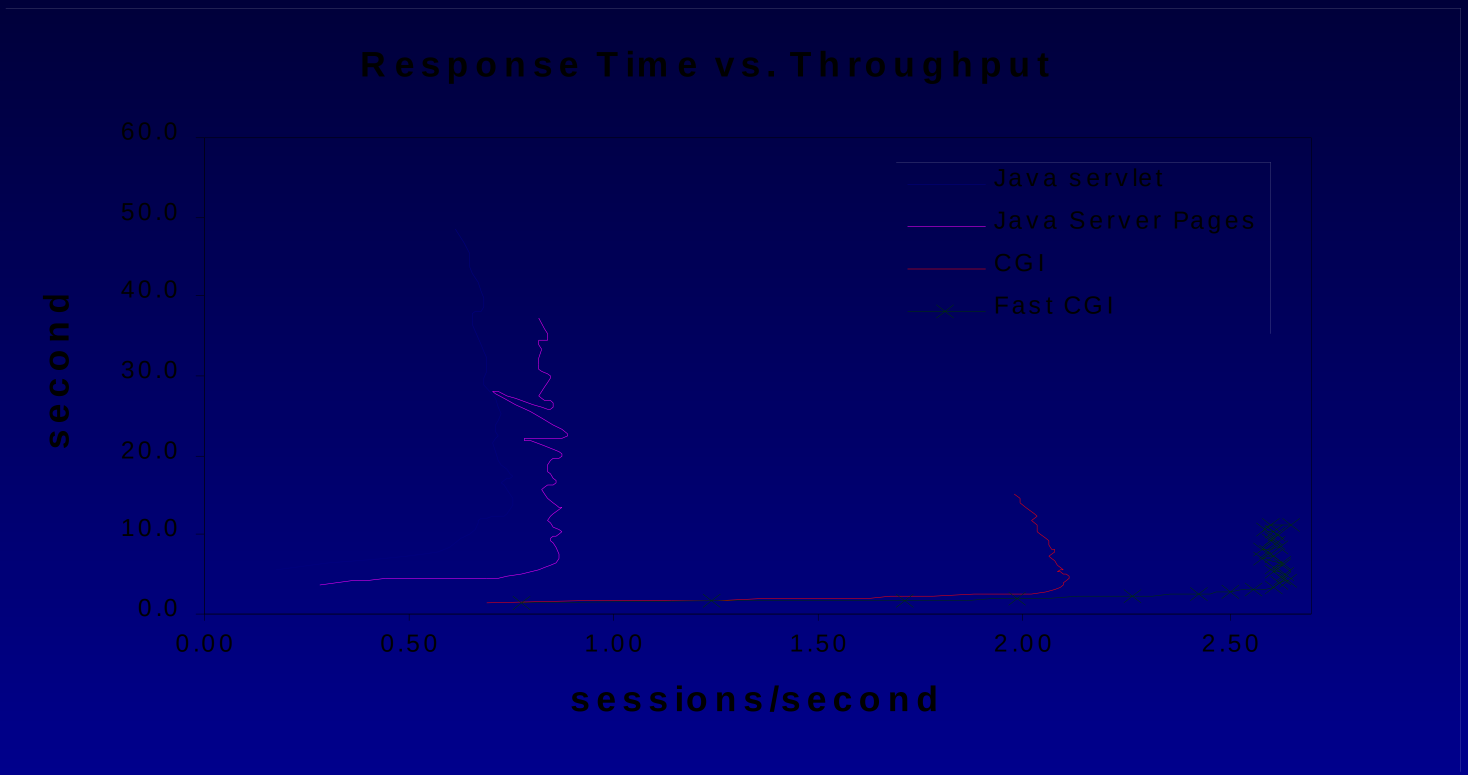
C++

Java vs C++
Simple Example



Is Java Slow ?

Realistic Example : A messaging application implemented in Java (servlets and JSP) and C++ (CGI and FastCGI) [5]



Does Java have scalability problems?

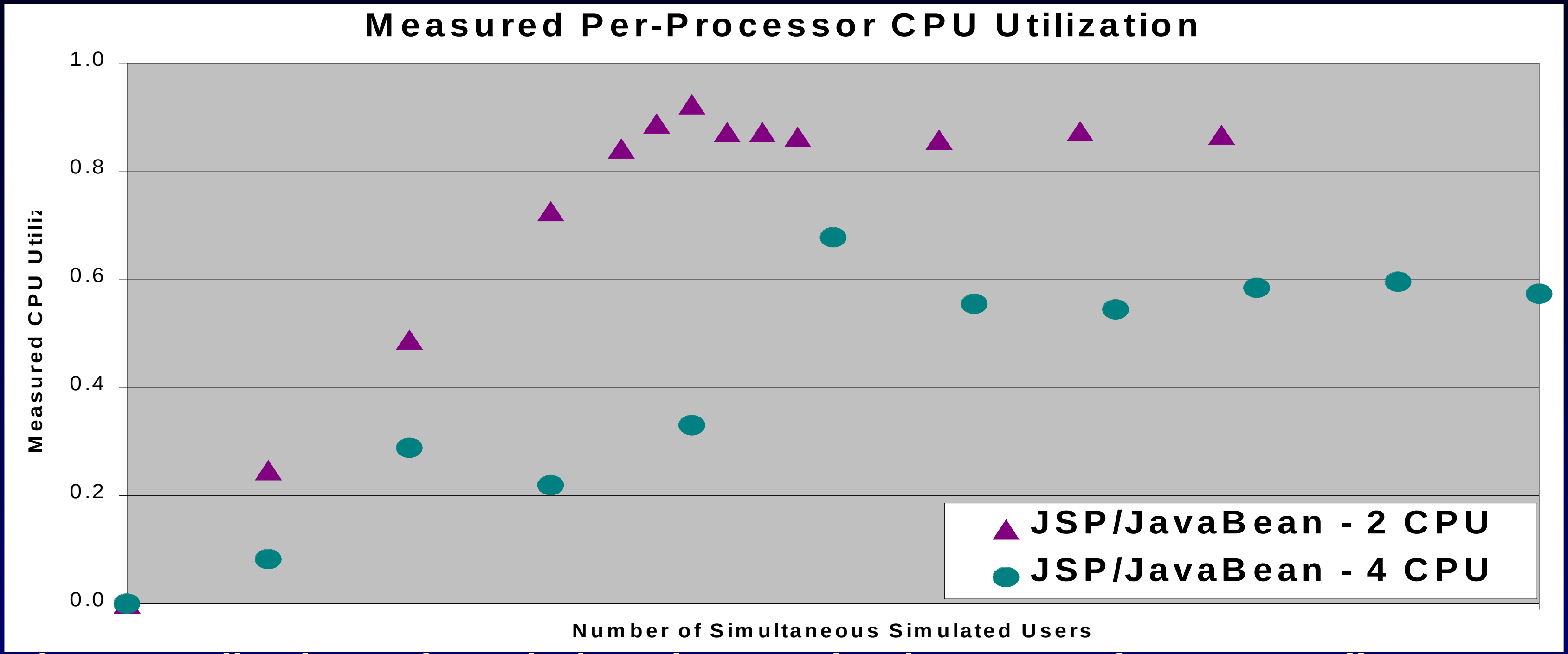


Figure from "Implications of Servlet/Javabean technology on Web server scaling": Cura, Ehrlich, Gotberg, Reeser

- **Bottleneck prevents use of multiple CPUs efficiently**
- **Thorough analysis pointed to inherent Java bottleneck**

Java scalability

- Some history of poor scalability: e.g. Java 1.1.7
  - Article in JavaWorld, August 2000 - "Java Threads may not use all your CPUs", P. Killelea.
    - Two programs:one in C, that does an empty loop, same in Java.
    - Run the program as multiple processes on 12-CPU machine scalabilityof C++ process
    - Run the Java program as multiple threads

Java Scalability

The C program:

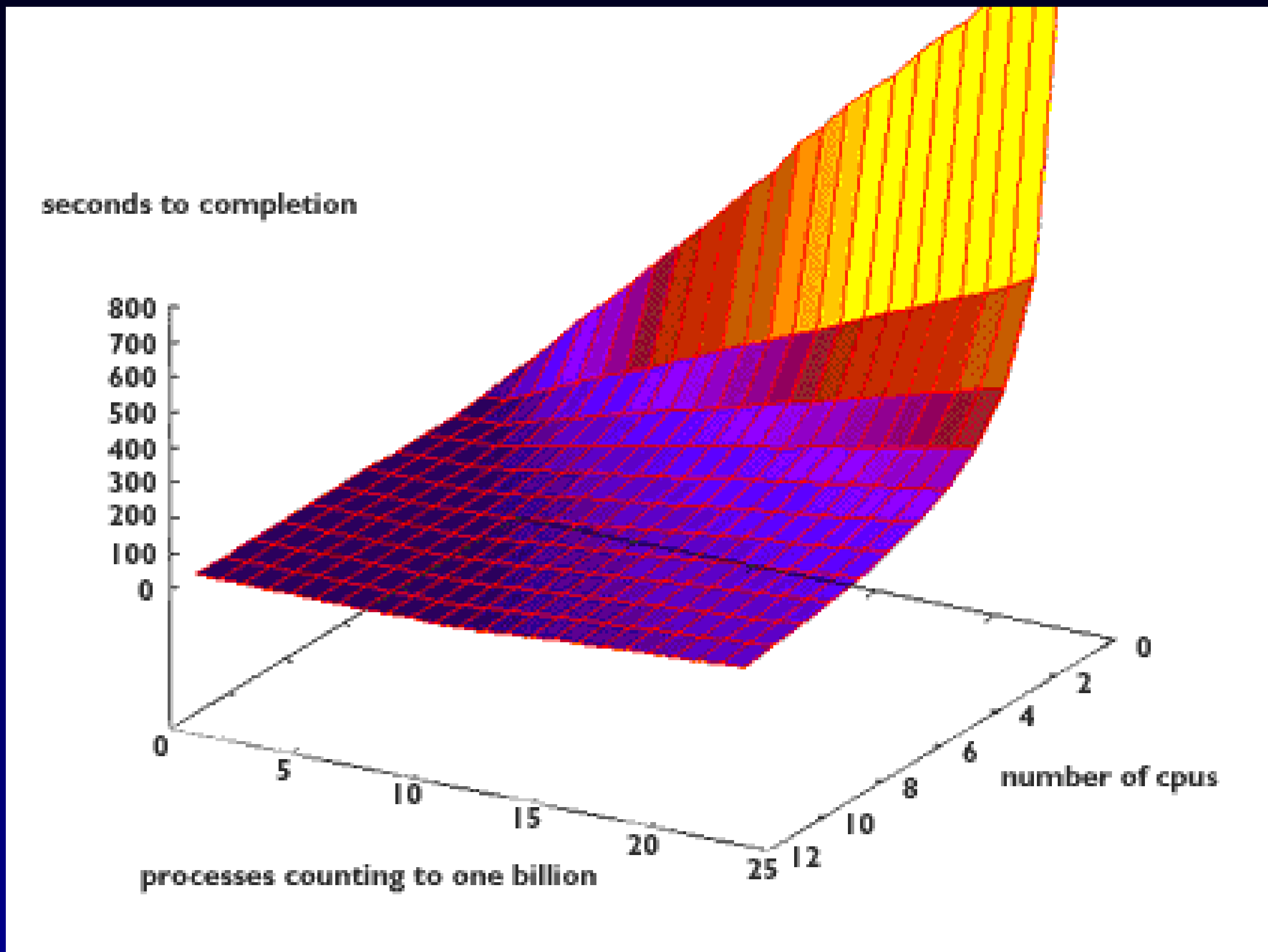
```
main() {
  unsigned long i
   for (i = 0; i < 1000000000; i ++);
}
```

Perl Wrapper creates multiple processes

The Java program:

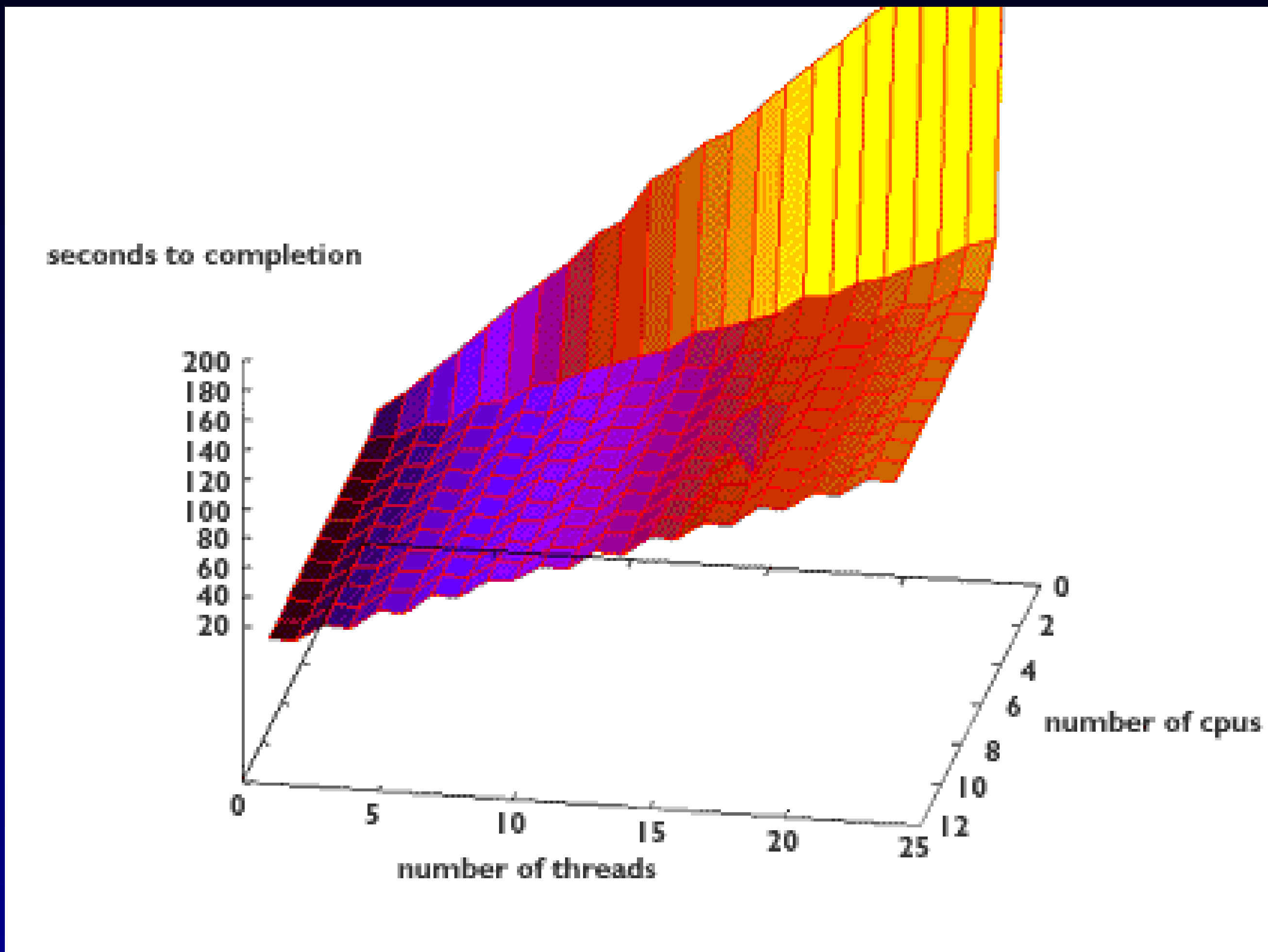
```
class Loop implements Runnable {
public static void main (String[] args) {
    for (int t = 0; t < Integer.parseInt(args[0]); t++)
            new Thread(new Loop()).start();
}
public void run() {
   for (int i = 0; i < 1000000000; i ++);
}
```

CPU Scalability - C processes



3 from
by
ea,
orld,

CPU scalability- Java Threads



from article by P.Killelea, JavaWorld, August 2000.

Initial Conclusion

- Java has performance problems
  - Root cause often hard to understand
- But Java has immense technical and business advantages
  - Use of Java for server programs will continue increasing
- Developers and analysts need to educate themselves on Java architecture and performance

Tutorial Goal

- Basic understanding of how Java works
- Identify elements of Java architecture that impact performanc
- Intro to issues in performance analysis of Java programs
- Guidelines to improving Java performance (references, papers, etc)

AT&T Network Design and Performance Analysis Consulting

Java Architecture

July 27, 2001

How Java Works

1. Write code in Java : *foo1.java, foo2.java*

2. Compile:
   - javac foo1.java foo2.java

   (javac is the *Java compiler)*

   - generates *bytecodes* in a *class* file:
     - foo1.class, foo2.class

3. Run:
   - java foo1.class

   ("java" is the JVM: *Java virtual machine)*

Note: No linked executable

Each application runs inside its own JVM

Java Platform Components

- Programming Language
- Class file format
- API
- JVM
- JVM+API = platform for which Java programs are compiled

Programming Language

- Object Oriented
- Robustly checked (type checking, array bounds, memory references…)
- No explicit memory management functions (no *free(), destroy()*)
- Syntactically like C++
- Has a rich class library - vectors, hastables, Internet, Web, …
- Naturally multithreaded

Java Class File

- Binary file format of Java programs
- Completely describes a Java class
- Contains bytecodes - the “machine language” for a Java virtual machine
- Designed to be compact
  – minimizes network transfer time
- Dynamically Linked
  – can start a Java program without having all classes - good for applets

The Java Virtual Machine

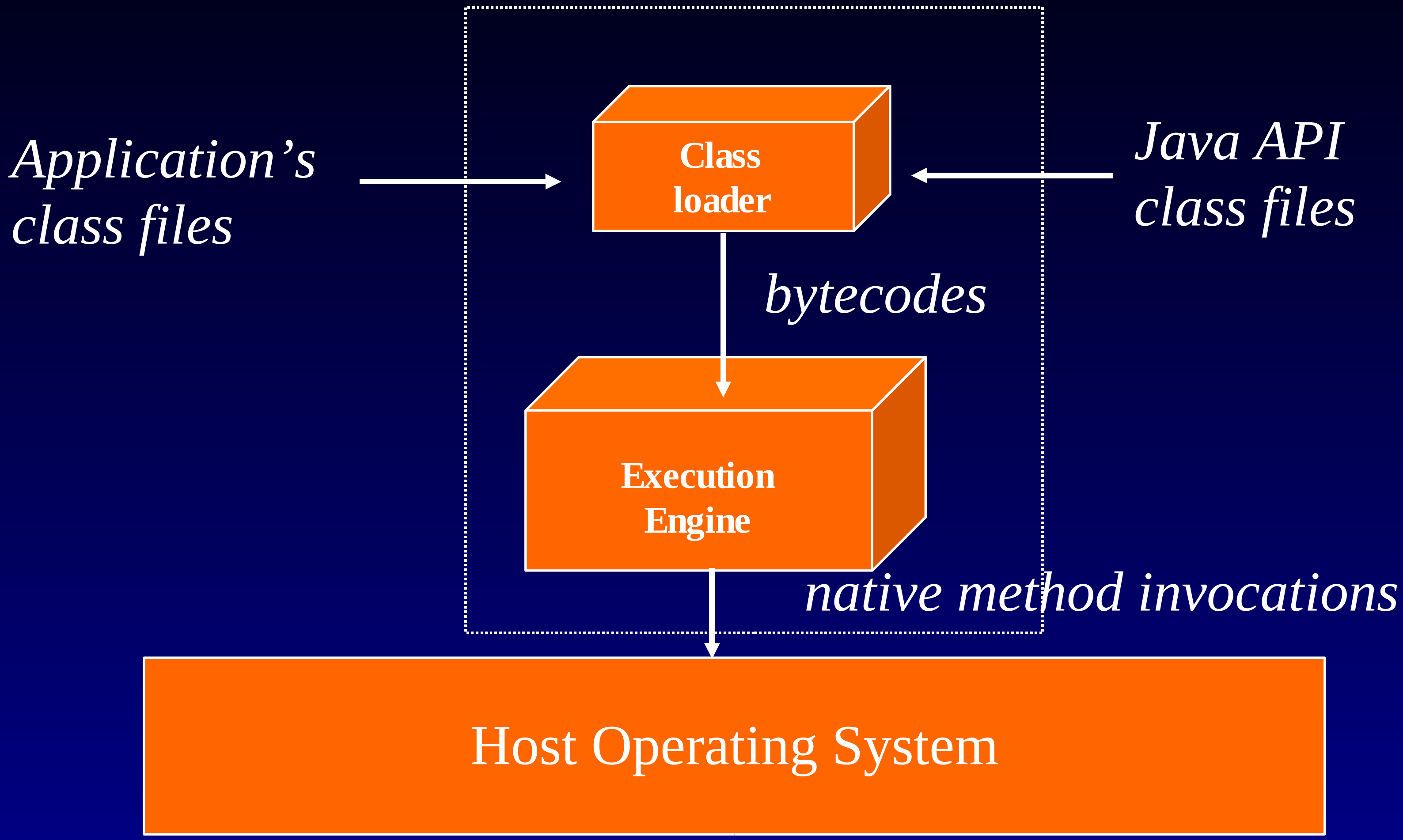


**Figure 1-4, from Venners[1]*

JVM (Java Virtual Machine)

- JVM Class loader loads classes from the program and the Java API
- Bytecodes are executed in the execution engine
- Interpreted or
- *just-in-time complied* : method compiled to native instructions when first compiled, then cached

The Java API

- Set of runtime libraries that provide a standard way to access system resources on a host machine
- JVM+Java API are required components of the Java Platform
- The combination of loaded class files from a program, the Java API and any DLLs constitutes a full program executed by the JVM

AT&T Network Design and Performance Analysis Consulting

Java Under the Hood

July 27, 2001

Java VM architecture

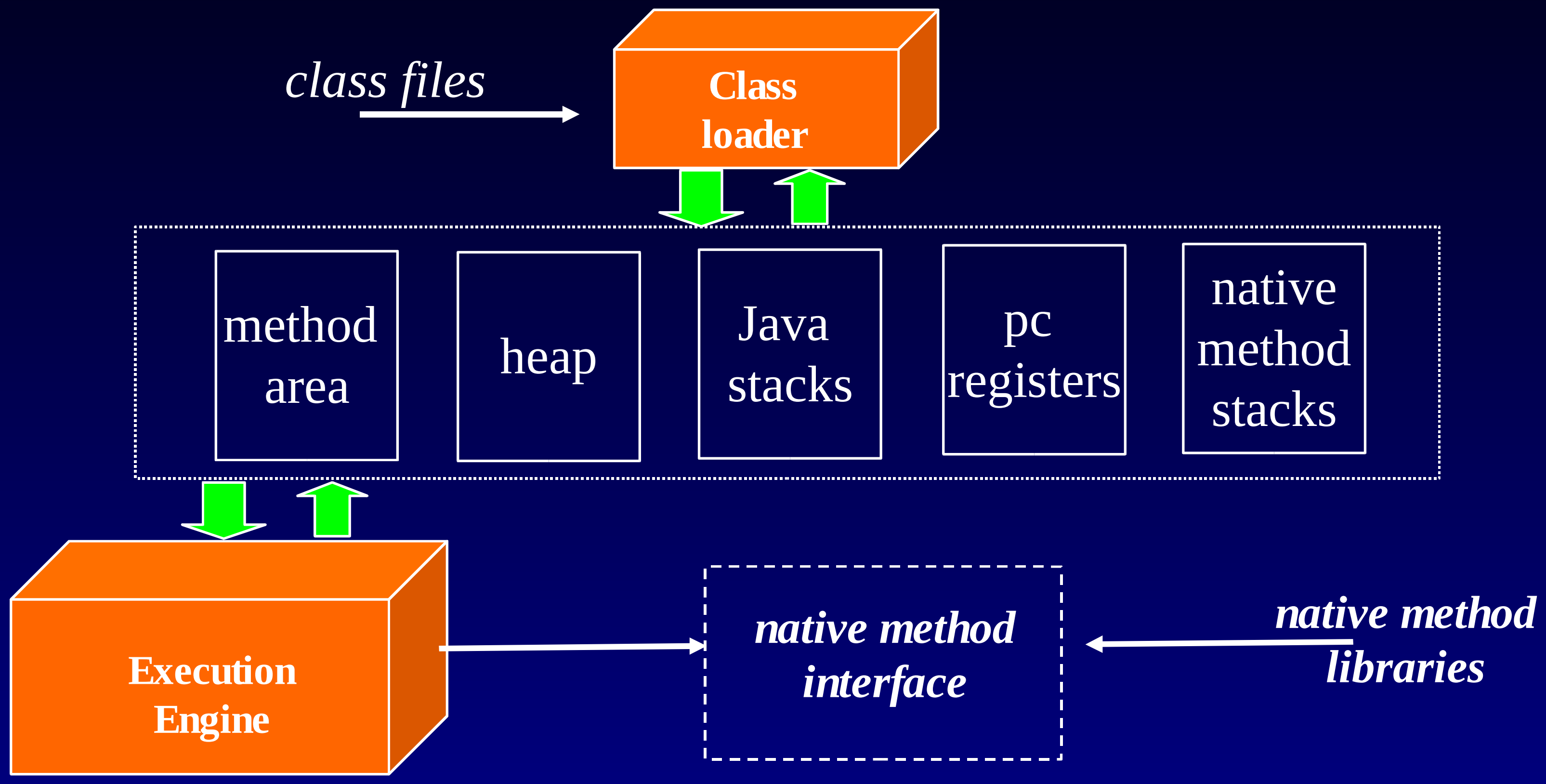


Figure 5-1, from Venners[1]

JVM: Run-Time Data Areas

Shared:

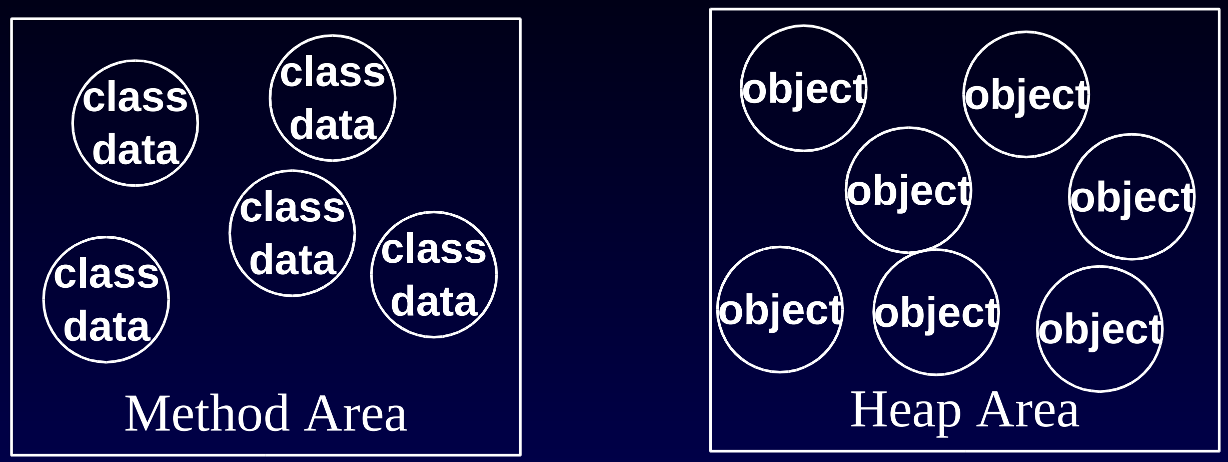


Figure 5-2, from Venners[1]

Exclusive for each thread:

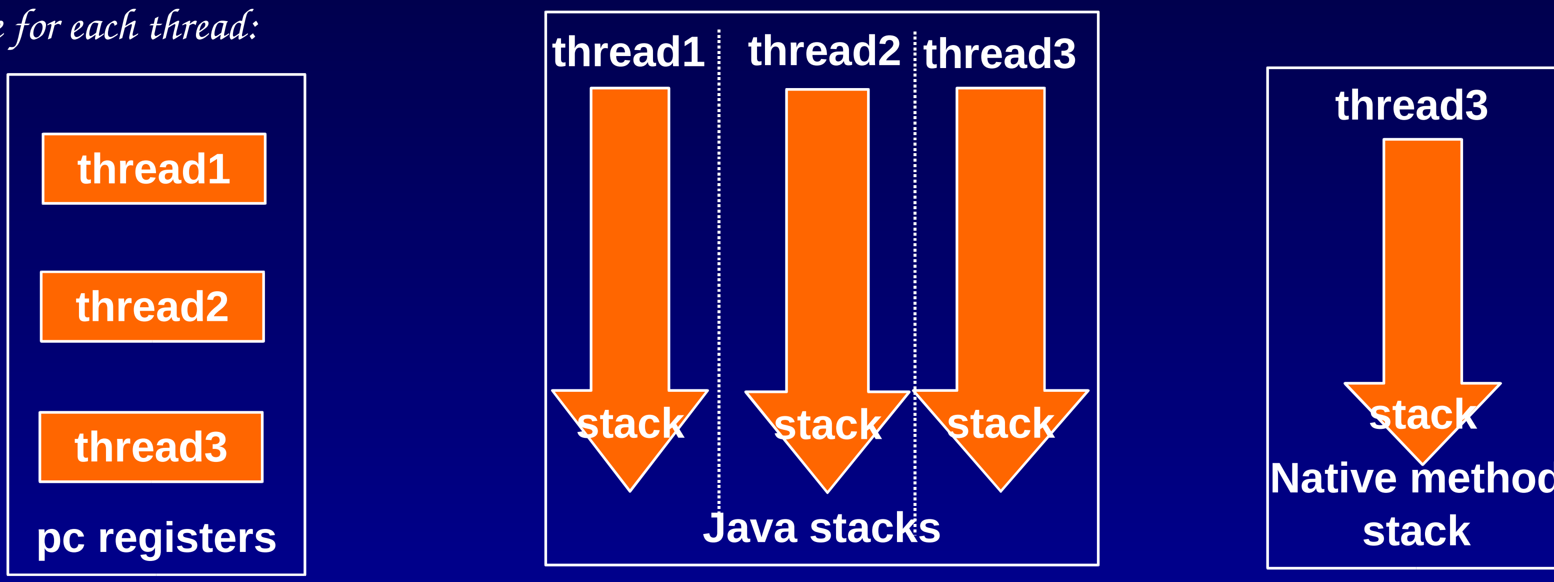
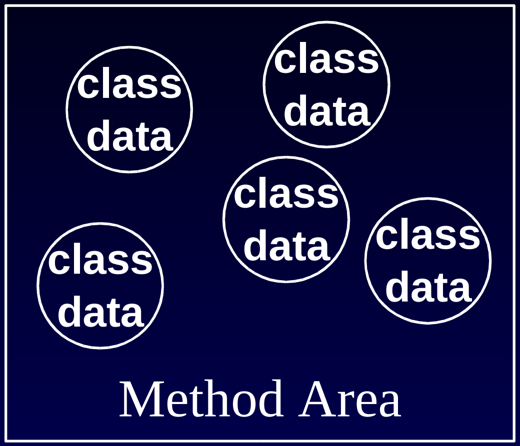


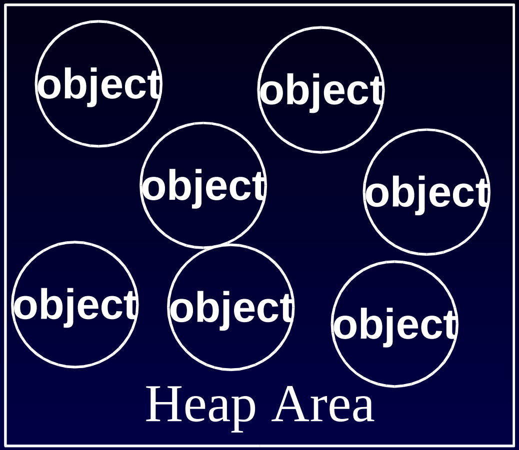
Figure 5-3, from Venners[1]

Method Area:



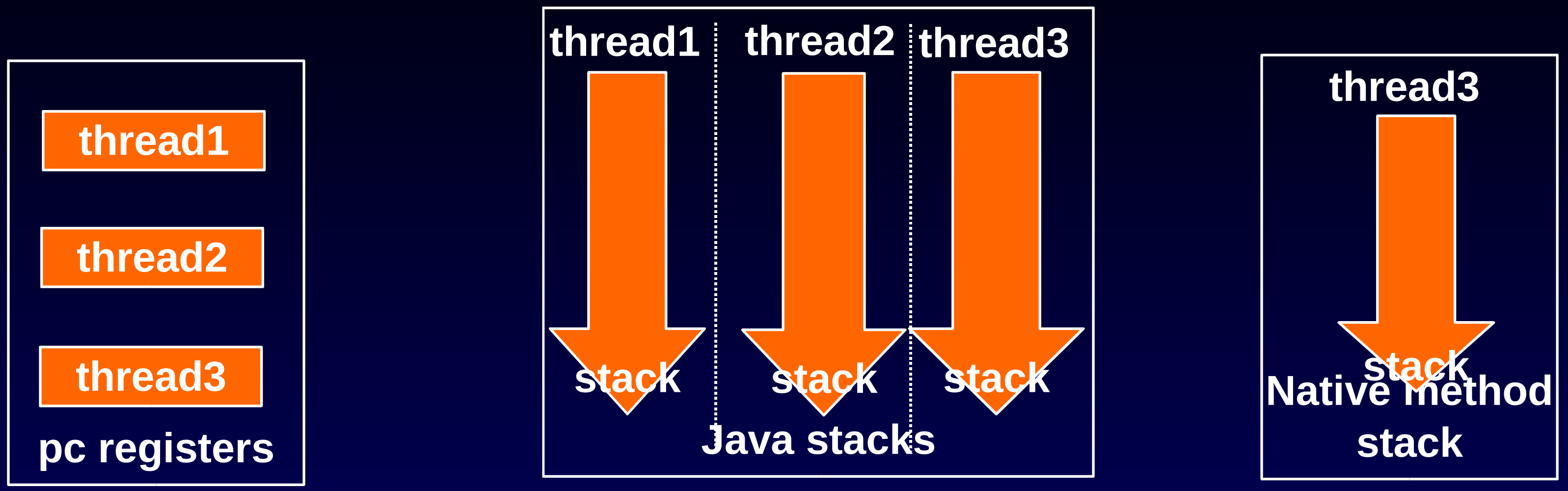
- Class loader loads class information in this area
- All threads share the same method area - must be thread-safe
  - If one thread is loading a class, the other must wait
- Method area could be allocated on the heap also
- Can be garbage collected
  - Collect unreferenced classes
- Type information:name, superclass name,field info, method info, method bytecodes, a reference to class "Class",...

The Heap



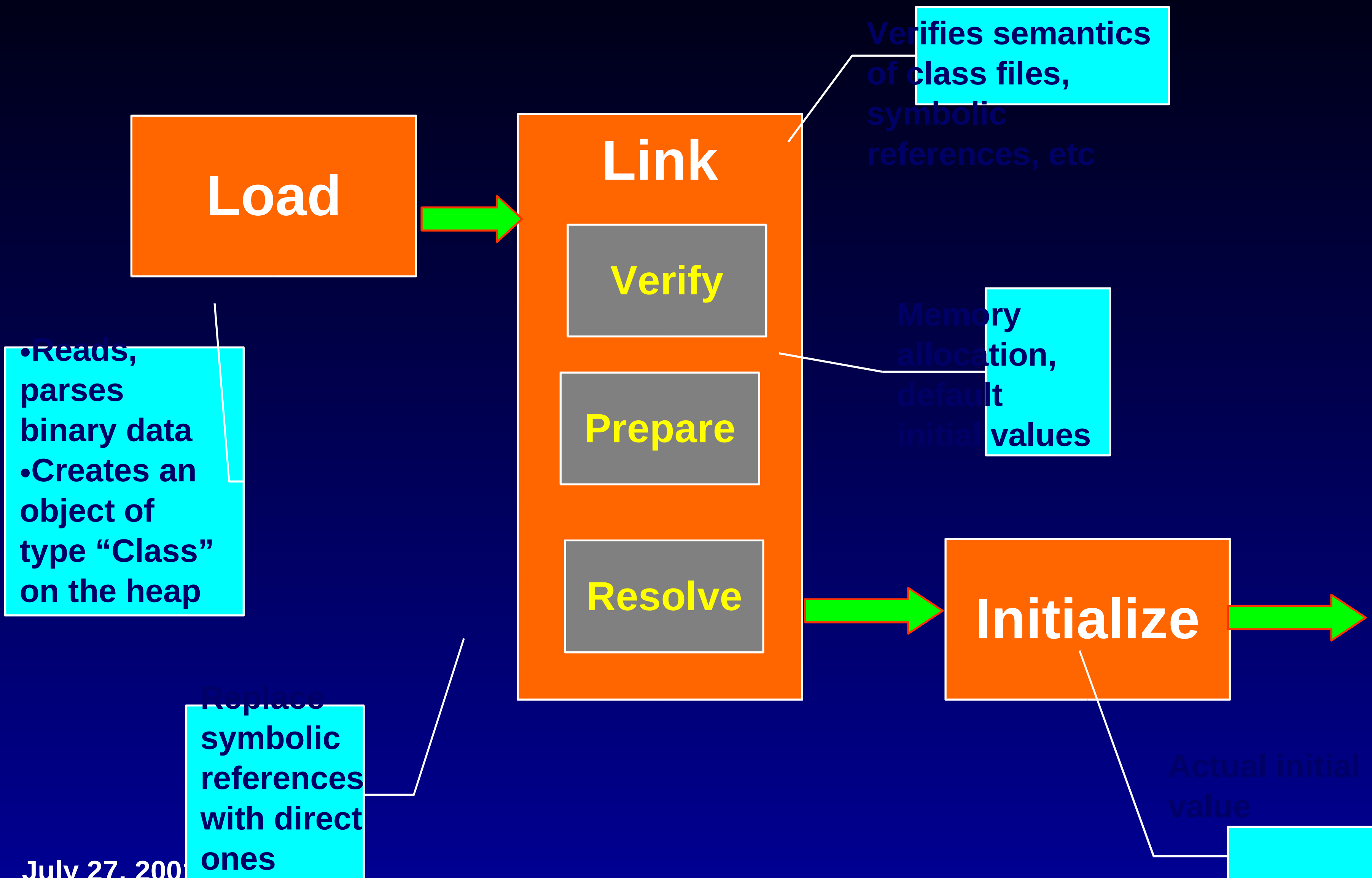
- Area where memory is allocated for objects created during run-time
- Each object has instance data, and pointer to class data in the method area
- Not shared between two applications (each runs inside its own JVM)
- Shared between multiple threads of the same application
  - Access to heap must be thread-safe
  - Access to objects must be thread-safe
- Is managed by JVM using automatic garbage collection (GC)
  - Memory from unreferenced objects is reclaimed
- May have an associated handle pool that points to the actual objects
  - Object reference: Pointer into handle pool
  - When objects are moved during GC - update only the handle pool

Stacks, PCs

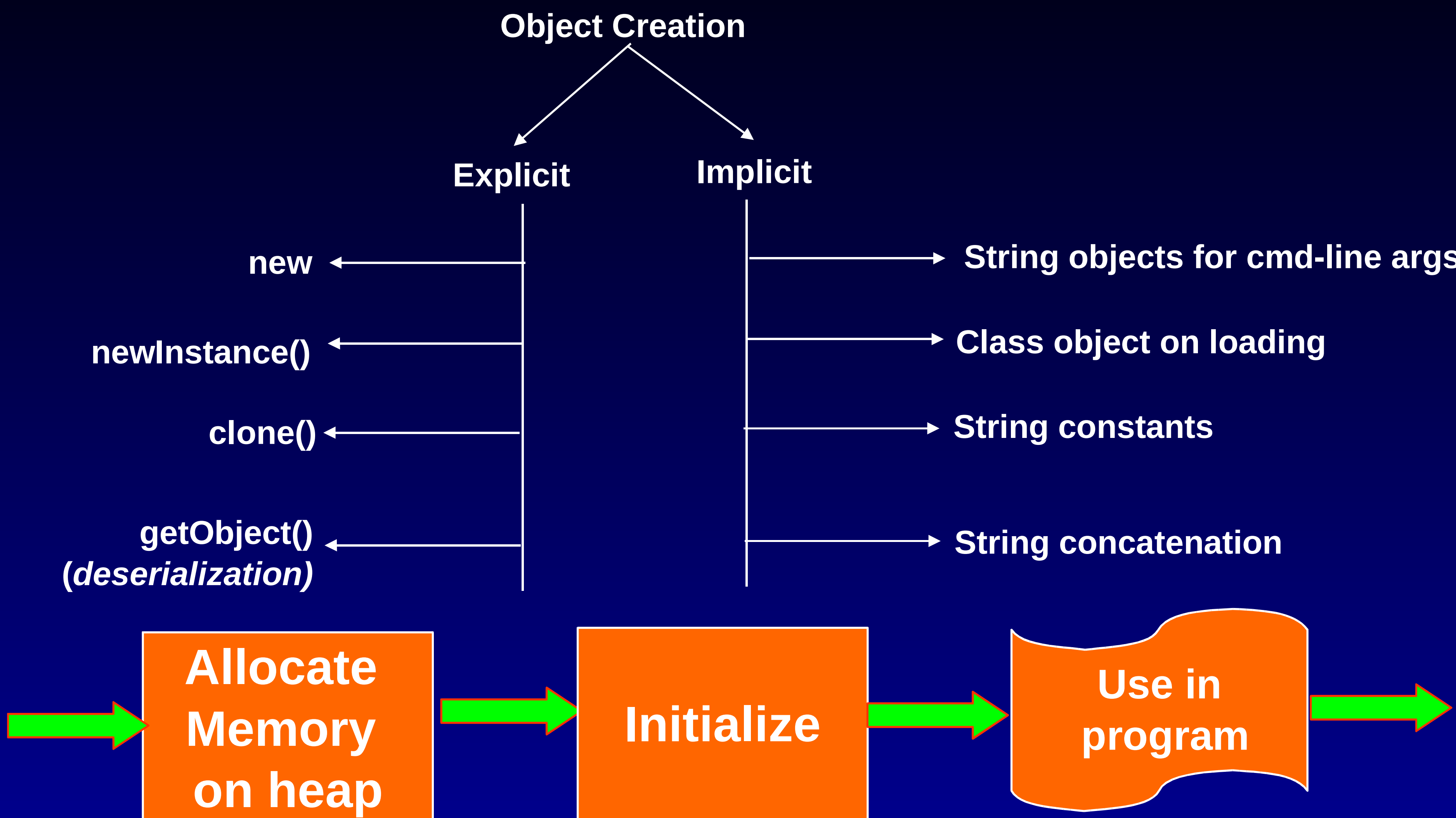


- Each thread has separate stack -
  - no danger of access by another thread
- Method calls generate stack frames - containing parameters, local variables etc
  - may also be allocated on the heap

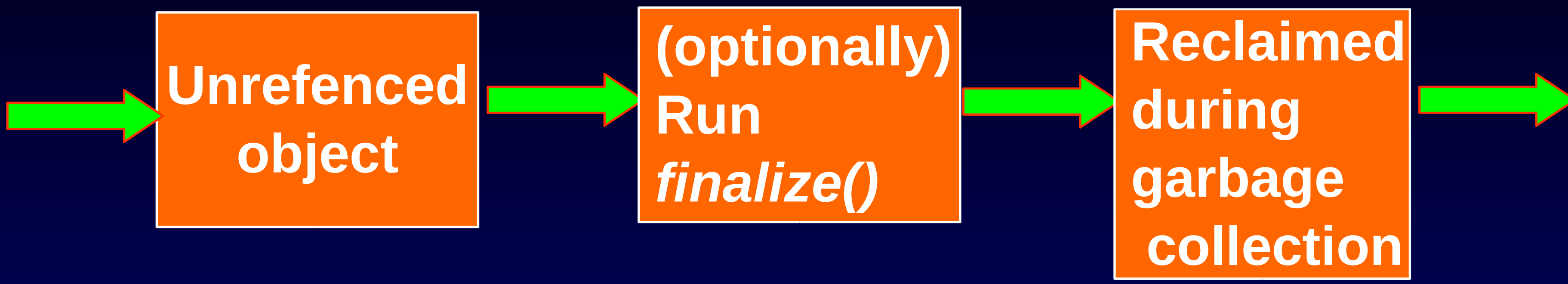
Lifetime of a class



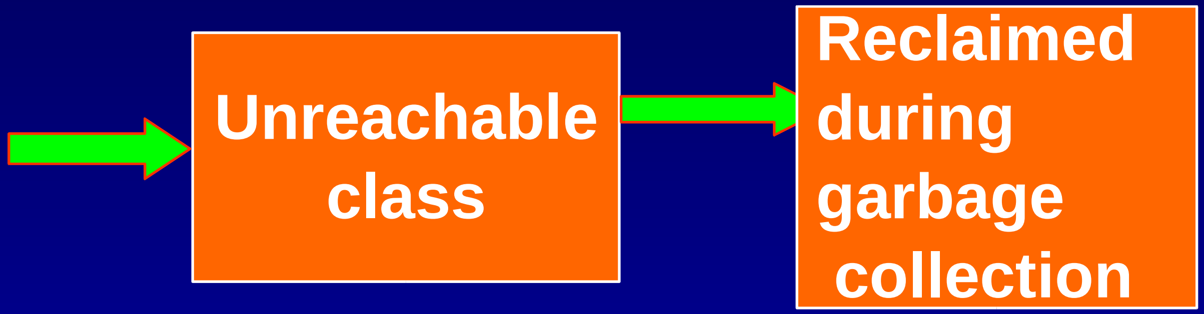
Class Instantiation



Discarding objects



Discarding classes



Garbage Collection

- JVM recycles memory used by objects that are no longer referenced
- GC needs to
  - Determine which objects can be freed, free them
  - Take care of heap fragmentation
- Various algorithms for GC, JVM specification doesn't force any one.

Garbage Collection Algorithms

- Tracing Collectors:
  – Trace from roots (e.g. local variables, operands) down the reference graph. Collect unreachable objects
- Counting Collectors:
  – Maintain reference count for objects.
  – Collect when count goes down to zero.
    - Cannot detect circular references

Garbage Collection -Heap Compaction

- Compacting Collectors:
  - Slide live objects over to occupy free space
- Copying Collectors

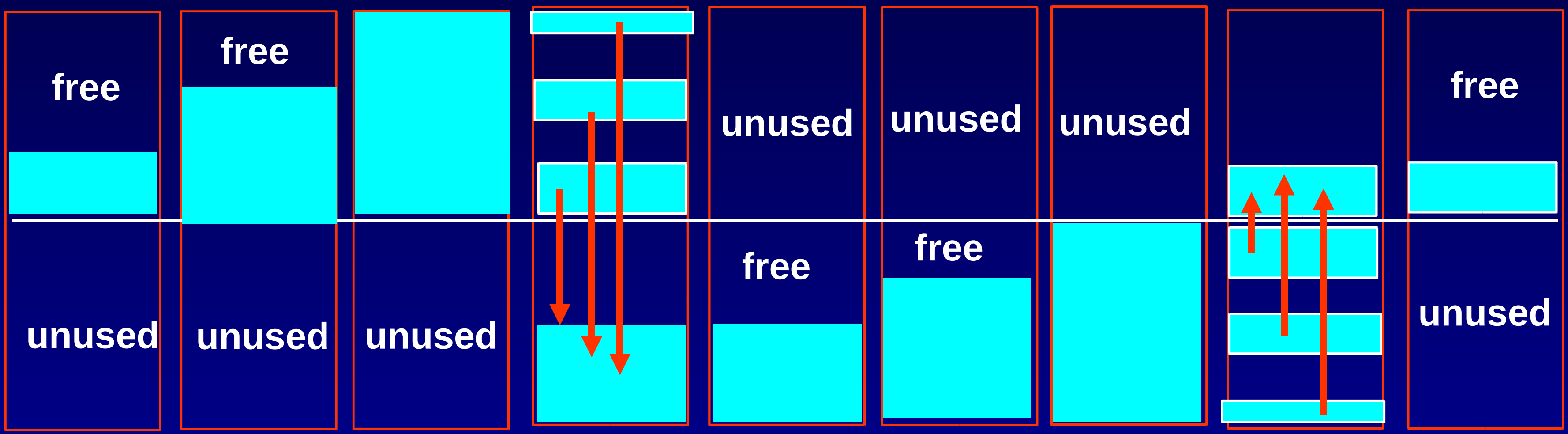


Figure 9-1- from Venners[1]

Garbage Collection -Compaction

- **Generational Collectors: Two observations:**
  1. **Most objects are short-lived**
  2. **Some objects have long lives**
  - **Group objects by age or “generations”**
  - **GC younger generation more frequently**
  - **Surviving objects move up generations**

Synchronization

- Java has a multi-threaded architecture
  - Easy to write code that will not work well with multiple threads
- Use synchronization constructs for
  - Mutual Exclusion: For coherent use of shared data
    - Synchronized statements
    - Synchronized methods
  - Co-operation
    - Working together towards a common goal
    - *wait* and *notify* commands

...Synchronization

- Implemented by acquiring locks on objects
  - Synch statements - lock any object

```
class someClass {
     int someVar;
     synchronized(anObject) {
         someVar++;}
     }
}
```

  - Synch methods - lock the object on which the method was called

```
class someClass {
     int someVar;
     synchronized void incr {
         someVar++;
     }
}
```

Exceptions

- Error handling mechanism
  - programmer can “throw” exception
  - Exception object is created with string comment and stack trace
- Involves object creation, initialization

Security

- Security achieved by:
  - Strict rules about class loading (will prevent loading malicious classes)
  - verification of class files
  - run-time checking by JVM
  - Security manager and the Java API (manages access to resources outside the JVM)

AT&T Network Design and Performance Analysis Consulting

Performance Impact of Java Architecture

July 27, 2001

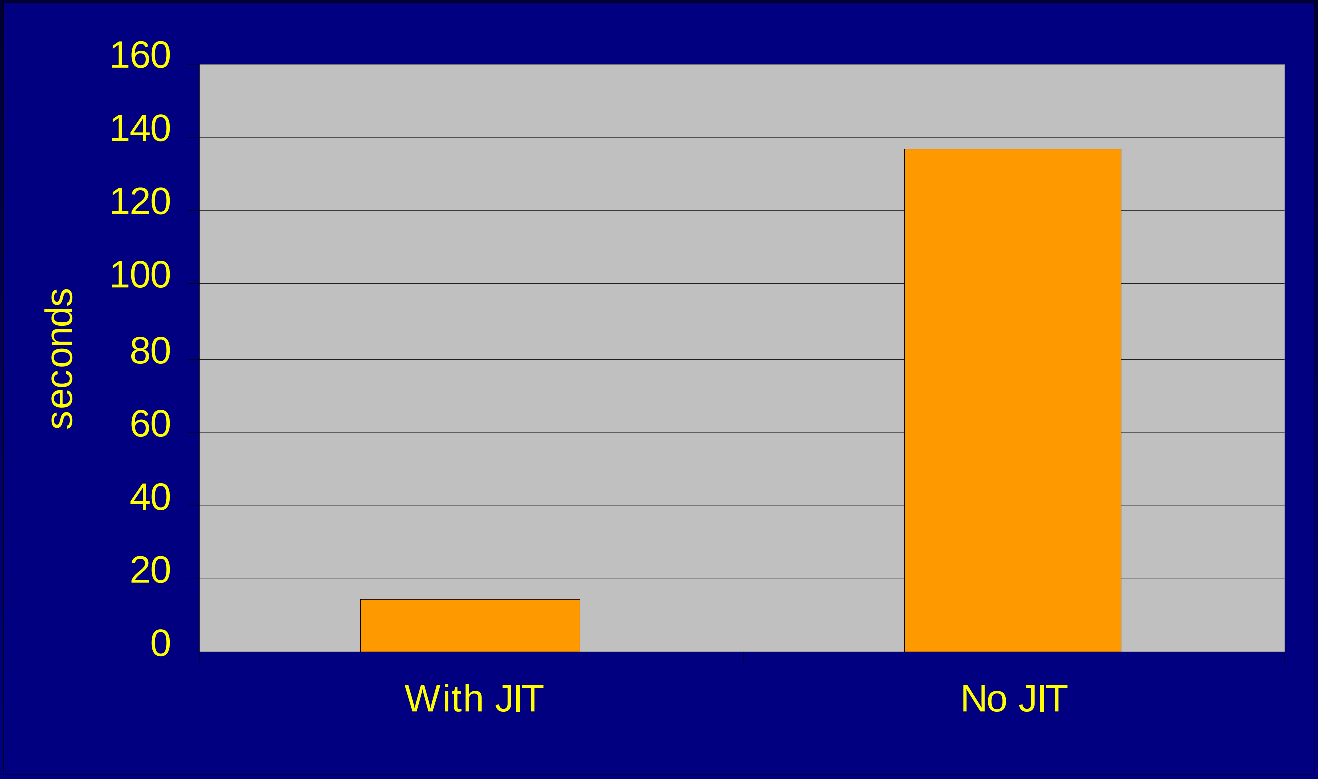
Why is Java slow ?

"Obvious" contributors :

- Bytecode Interpretation (if not jit-ed)
  - Server-side applications may spend only 10-20 % of time executing Jit-ed code (IBM Systems Journal Paper[3].)
- If jit-ed, compilation cost (one-time), footprint cost
  - OS memory management overhead (paging, scanning etc)

Example

- M/M/1 Queue Simulation: Factor of 10 difference in execution time



More Basic Features Impacting Performance

- Dynamic Linking
- Checking of array bounds on each access
- Checking for null references
- Primitive types are the same- not adjusted to the most efficient type for each platform
- ….

Why is Java slow? - Major contributors

- **Non-obvious, but deeply impacting performance:**
  - **Object creation**
  - **Garbage collection**
  - **Synchronization**
  - **API classes too general**
    - ***General-purpose design <u>always</u> implies performance penalty***
    - **Improper use of Classes and APIs**

Performance Impact of Object Creation

Object Creation involves:

- Allocating memory
  - including for superclasses
- Initializing instance variables to Java defaults
- Calling Constructors
  - including superclass constructors
- Initializing instance variables as programmed

Can be expensive!

Performance Impact of Object Creation

- **Example 1: Code piece A is 95 % faster than Code piece B**

A :

```
boolean  bool =
a.equalsIgnoreCase(b);
```

B :

```
ucA = a.toUpperCase();
ucB = b.toUpperCase();
boolean bool = ucA.equals(ucB);
```

- **Example 2: Code piece A is 60 % faster than Code piece B**

A :

```
Vector  v = new Vector();
  for (i=0; i<n; i++)
{     v.clear(); v.addElement… }
```

B:

```
  for (i=0; i<n; i++)
{Vector v = new Vector(); v.addElement..}
```

Object Creation

Two Overheads:

- Creating the object in the heap (previous slide)
- Since the heap is shared by all threads -
  - overhead due to contention for the heap

Object Creation… Scalability

- **Concurrency efficiency of object creation across threads**
  - **Program that creates 500,000 objects, on 6-cpu machine**

```
public void run () {
int i;
   myObj obj;
   Thread ct = Thread.currentThread();
   String thrName = ct.getName()+ ":";
  obj = new myObj();
  for (i = 0; i < mt; i++) {
      if (c == 1) obj = new myObj();
  }
}
```

Object Creation… Scalability

- Time program with varying number of threads- but total # of objects created is always 500,000.

# threads	execution time	"ideal" time
1	15 s	15 s
2	10 s	7.5 s
3	8.5 s	5 s
4	7.6 s	3.75 s
5	8.5 s	3 s
6	8.3 s	2.5 s

>

Scalability : Sanity check

Concurrency efficiency of cpu-bound program

```
for (i = 0; i < mt; i++) {
    for (j = 0; j< 100; j++)
      f =  (i)/ (j+1);
 }
```

– Timings with varying # of threads (# of loop iterations is constant)

# threads	execution time	"ideal" time
1	18.9 s	18.9 s
2	9.9 s	9.4 s
3	6.9 s	6.3 s
4	5.6 s	4.7 s
5	4.6 s	3.8 s
6	4.1 s	3.1 s

≈

Object Creation

- Observations
  - Has a basic overhead
  - Programs doing lot of object creation (explicit/implicit) will have unexpected scalability problems
  - Each created object adds to garbage collection overhead
    - must be traversed
    - must be collected, when unreferenced.
    - Having many short-lived objects can be a performance bottleneck

Performance Impact of Garbage Collection

- Garbage collection adds a run-time overhead
  - In older JVMs GC could stop all processing
    - GC could result in user perceivable delays
    - Delays could be 5-10 seconds for large heaps (100-500 MB)[3]

Performance Impact of Garbage Collection

- Newer JDKs have improved algorithms
  - Sun JDK 1.3 has
    - Generational garbage collection
    - Train algorithm for the old generation sub-heap
  - Overhead is now smaller
    - e.g. Queue simulation example : 53 ms out of 13 s running time. Heap size b/w 160KB and 2MB
  - Is larger if heap is large

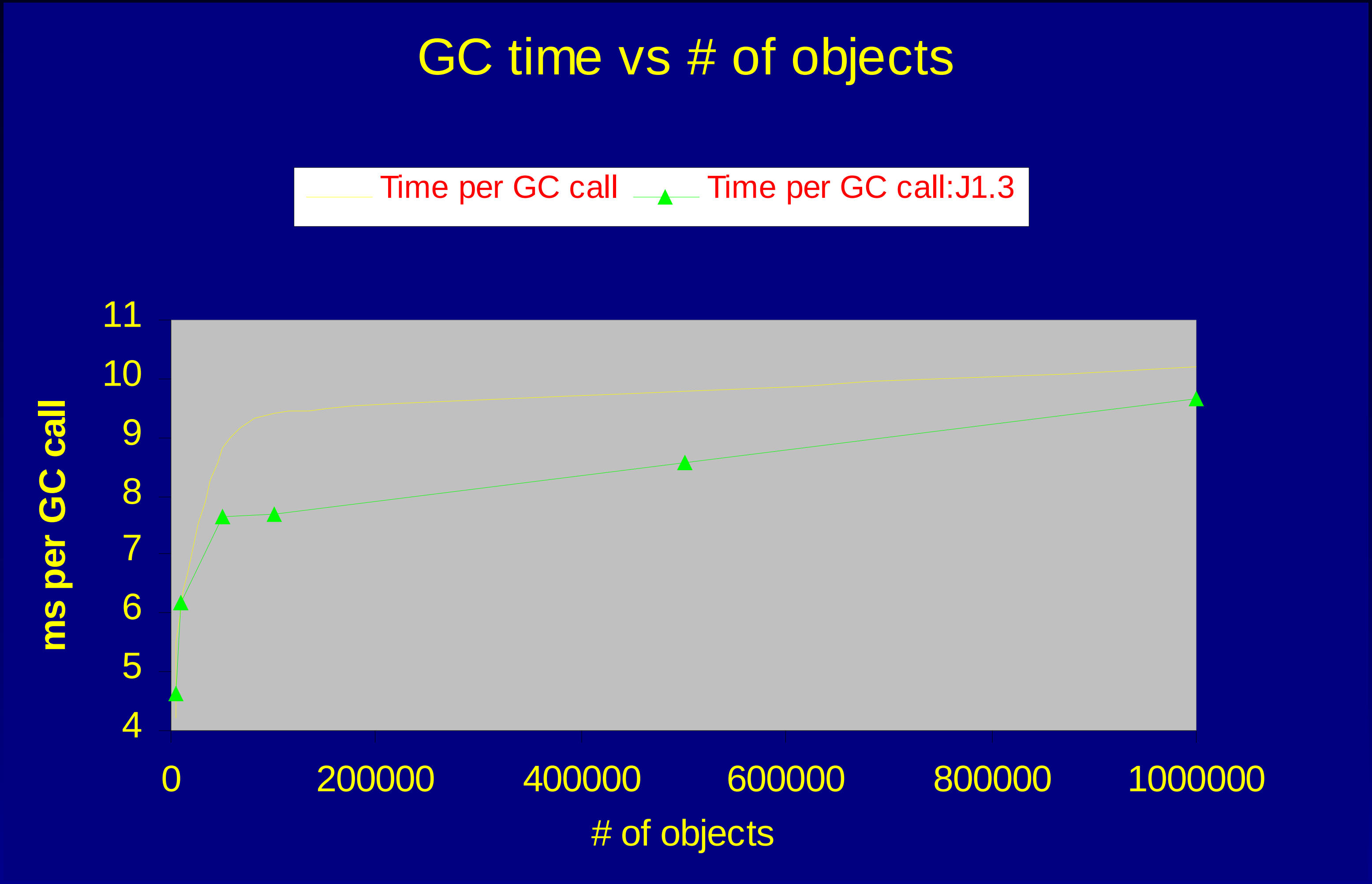
Performance Impact of Garbage Collection

- Garbage collection can be timed (*java -verbosegc)*
- Test GC in a program in which number of objects, and heap size keep increasing

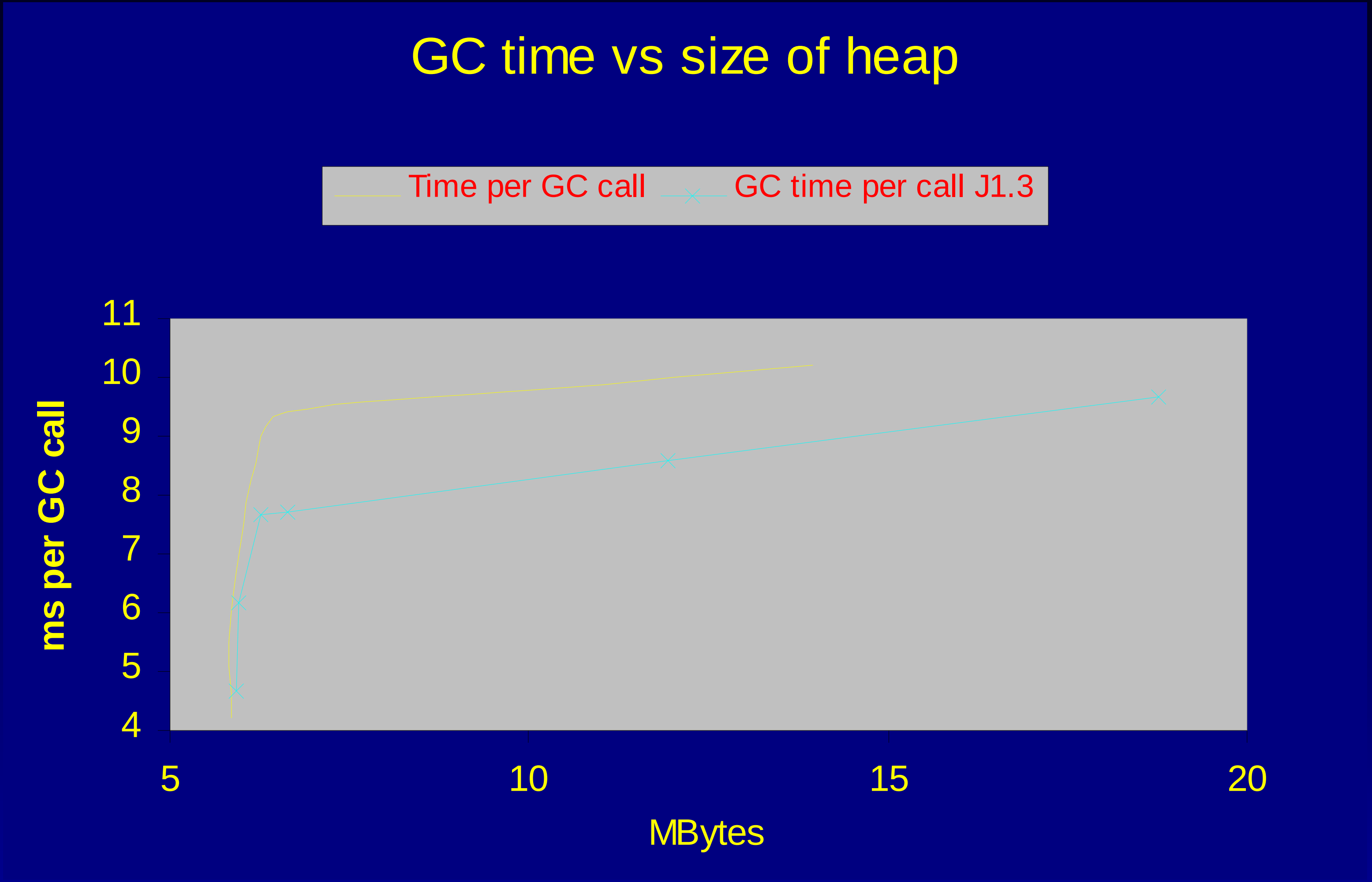
```
class create implements Runnable {
static int m, c, mt;
 public void run () {

   int i;
   myObj obj[]= new myObj[1000000];
   Thread ct = Thread.currentThread();
   String thrName = ct.getName()+ ":";
```

```
long st = System.currentTimeMillis();
   for (i = 0; i < mt; i++) {
     if (c == 1) obj[i] = new myObj();
     //System.out.println(thrName+obj);
   }
   long diff = System.currentTimeMillis() -st;
   System.out.println("Time: "+ diff);
 }
```

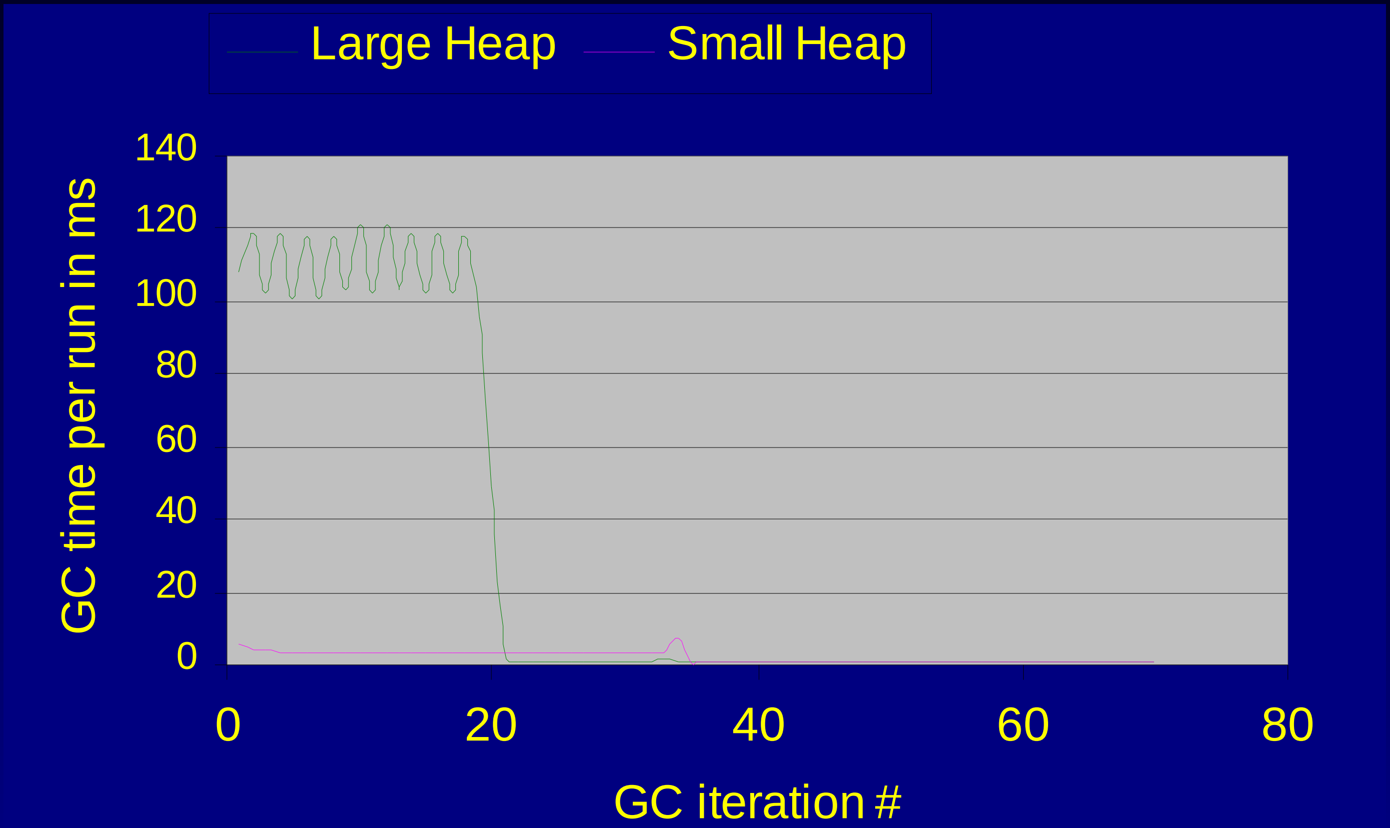
Performance Impact of Garbage Collection



Performance Impact of Garbage Collection



Performance Impact of Garbage Collection



Test queue simulation program, after allocating a large array of objects in the beginning, and then running the simulation as usual.

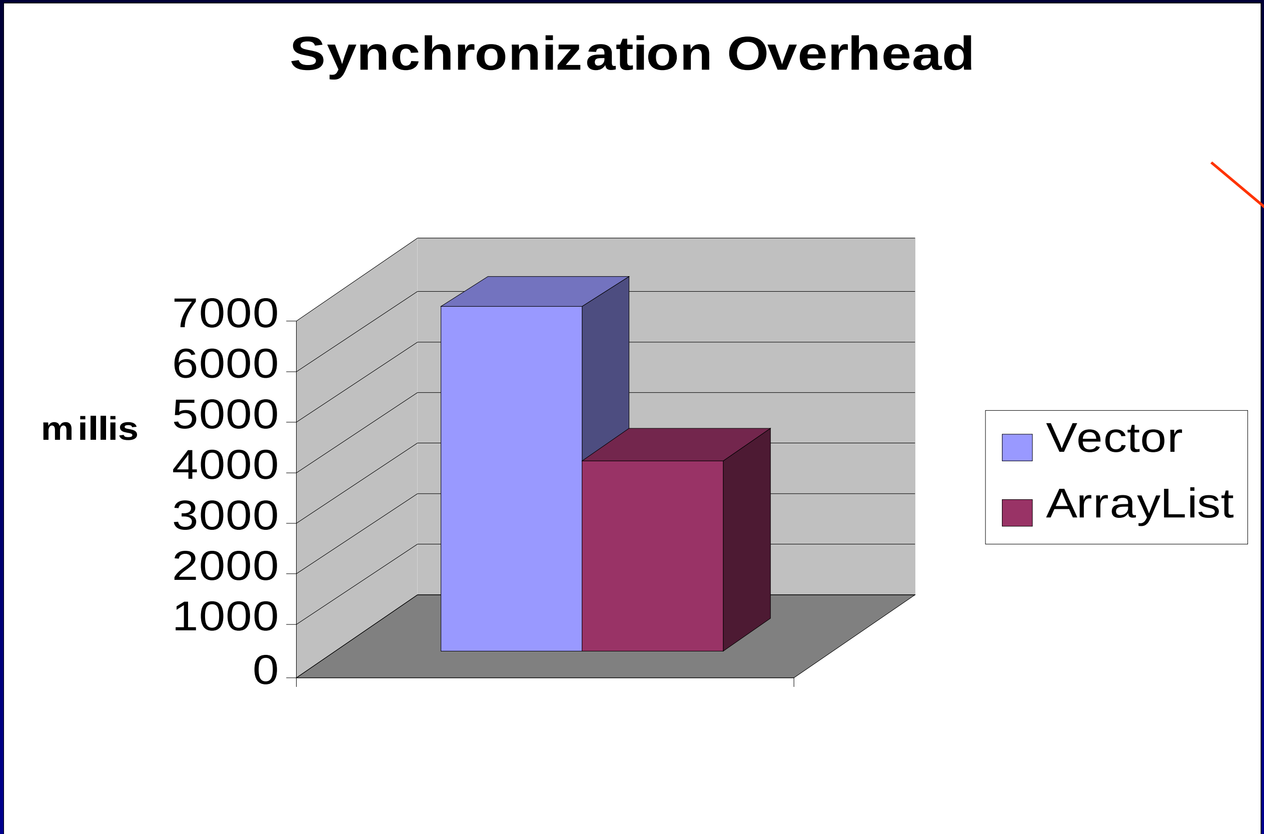
- Looks like GC learns about the long-lived object and does not include that in later GC?

Performance Impact of Synchronization

- "Obvious" :
  - In a multithreaded application, synchronized pieces will be the bottlenecks (Java-independent issue)
- Non-obvious (Java-isms):
  - Big synchronization overhead
  - Java API classes may have synchronized methods - a big overhead in cases where synchronization is not necessary (access only by one thread)
  - Implicitly shared objects internal to the JVM - e.g. heap. Access will be synchronized

Performance Impact of Synchronization

- **Example: Vector vs ArrayList** (example creates vector/array list, adds elements, then accesses them)

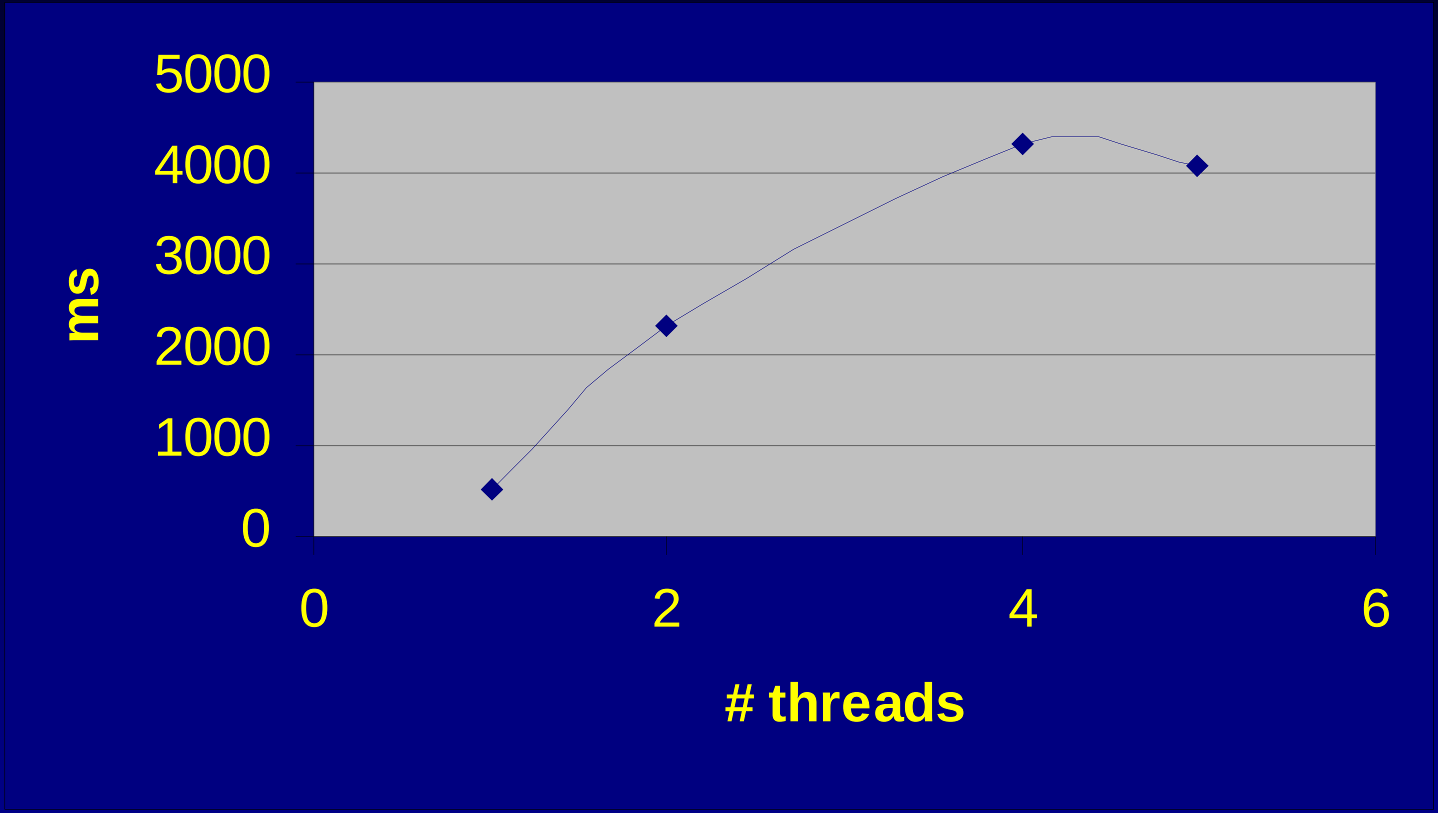


Vector is a synchronized class

From Bulka[2]

Performance Impact of Synchronization

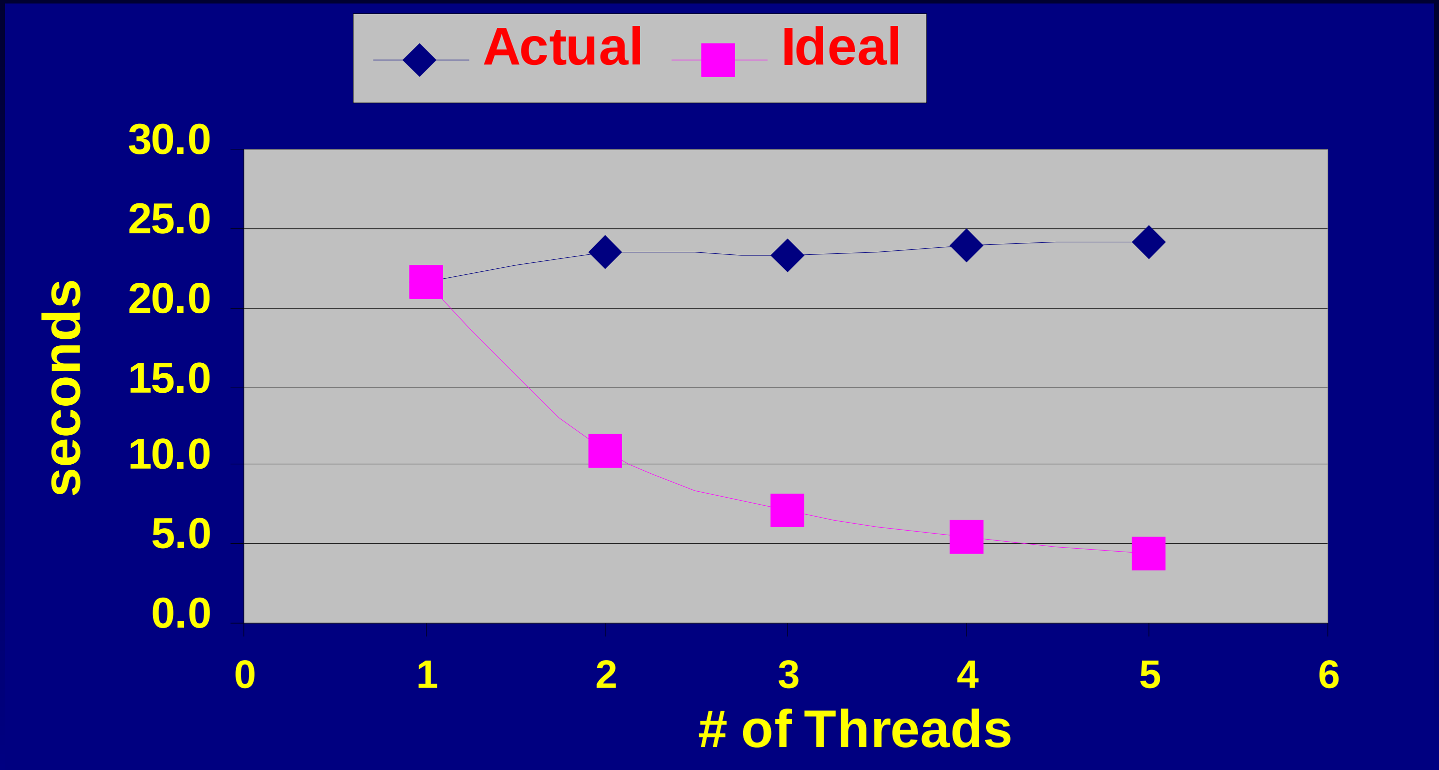
Contention for synchronized code:



Example Bulka[2]: increase a counter using synchronized method. Use increasing # of threads to do the same amount of total work. Results from 6-cpu machine.

Performance Impact of Synchronization

Implicitly synchronized code:



Object creation example, with printing inside the loop (*System.out. Println - not an explicitly synchronized function in Java. Access possibly serialized by OS)*

Performance Impact of Synchronization

AT&T Network Design and Performance Analysis Consulting

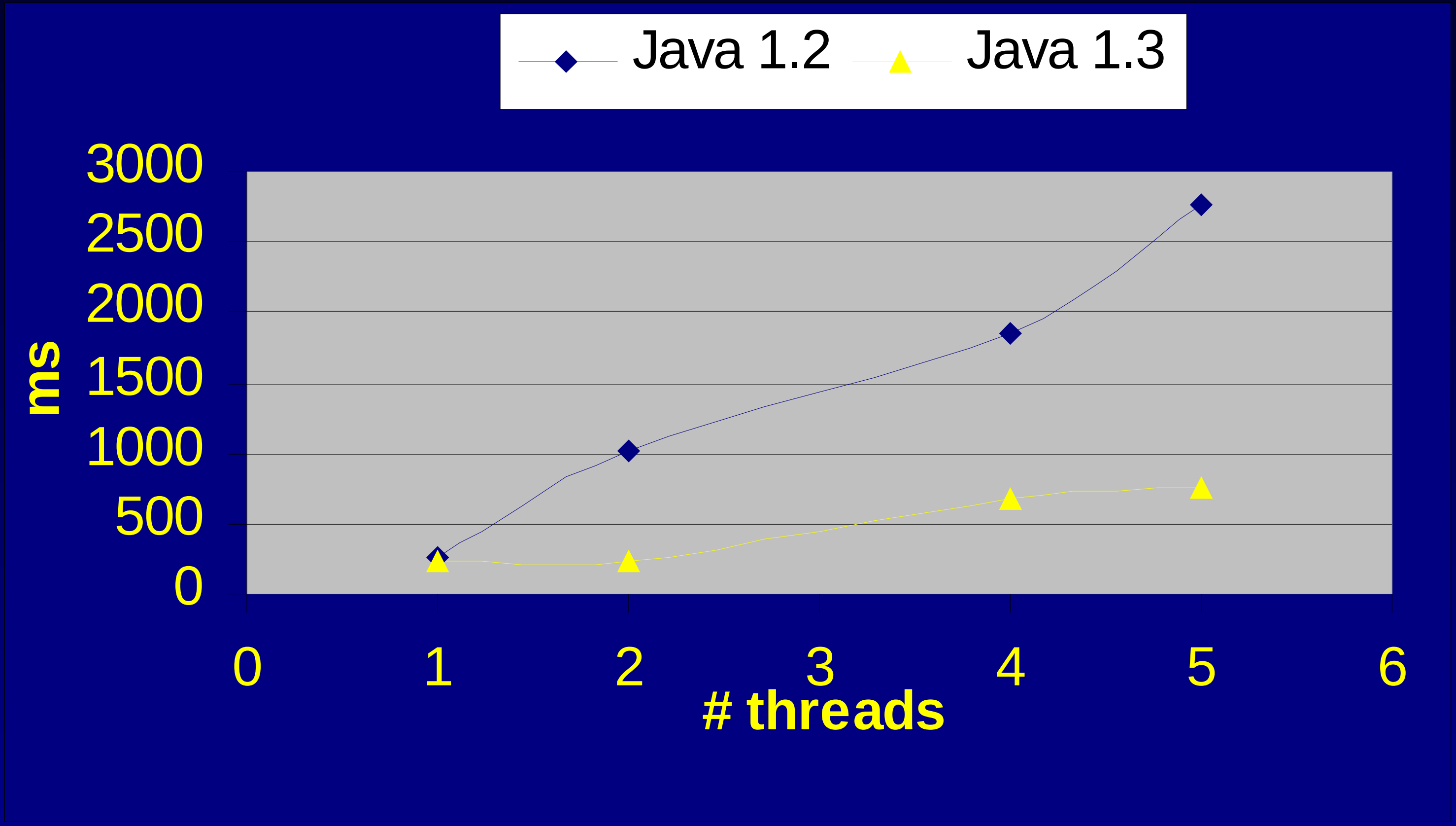
Example: Multiple threads increment a shared variable by calling Math.random()

Run this program with increasing number of threads, keeping the total number of iterations the same - on 6-CPU machine

```
class WorkerThread extends Thread {
   private int iter;
   private int tid;
   private static double num;
   public WorkerThread (int
iterationCount, int id) {
     this.iter = iterationCount;
     this.tid = id;
   }
   public void run() {
     for (int i = 0; i < iter; i++) {
        num += Math.random();
   }
  }
}
```

Performance Impact of Synchronization

- Example of multiple threads calling Math.random() - a *synchronized* method



Performance Impact of Synchronization

AT&T Network Design and Performance Analysis Consulting

- Object creation can be viewed as a special case of access to synchronized data structures and methods
- We saw similar effects there

General-Purpose API classes

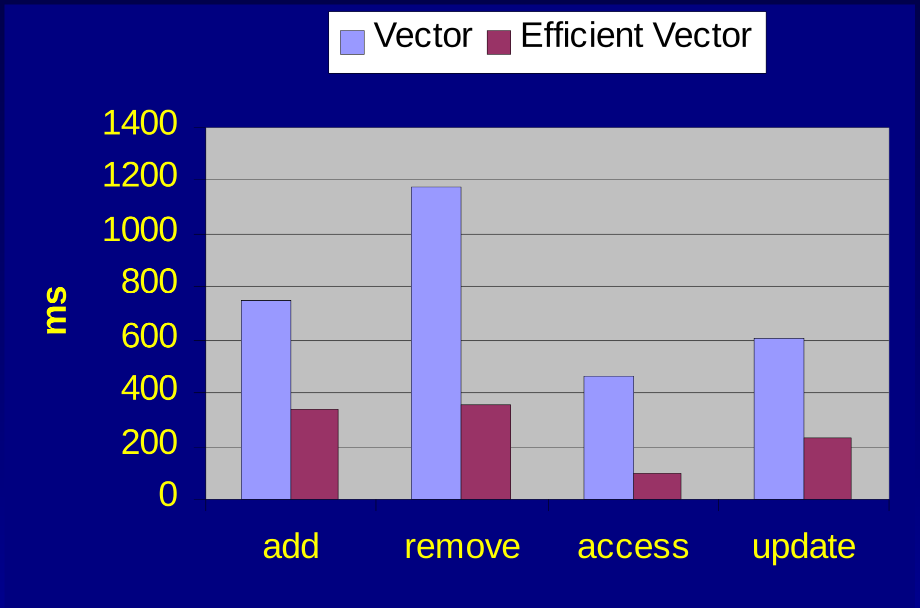
- Generally true: When a class/API provides maximum flexibility and features, there will be an associated performance cost. Examples:
  - Vector Class
    - Some applications may need their own efficient vector implementation
  - Date
    - Using native Date functions thru JNI might prove better performing

General-Purpose API classes

- **Example 1: Vector class provides basic access/update functions, growing capacity if needed, range checking, synchronization, iterator**

Example from Bulka[2]:

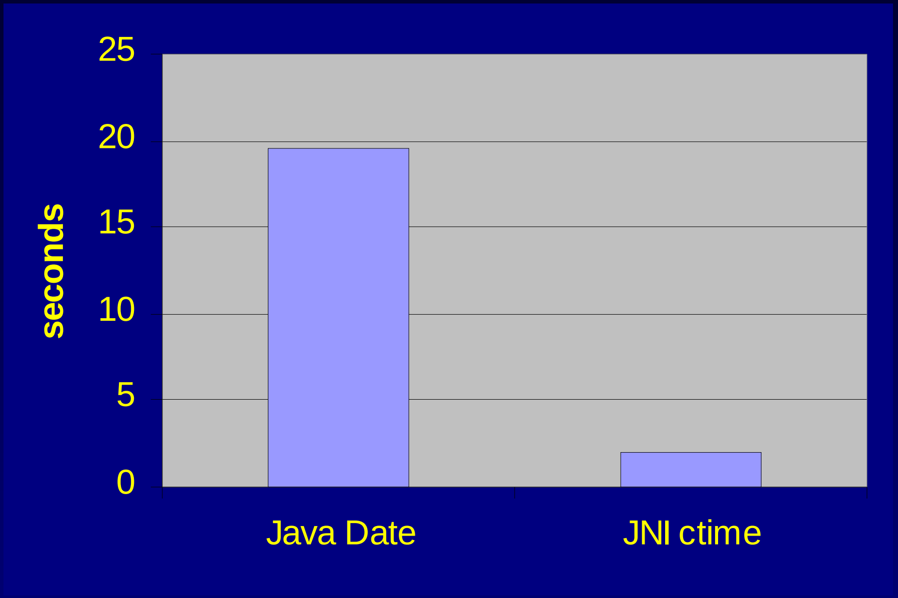
Speed up due to a "light" implementation of Vector class, offering few features.



Performance Impact of "Heavy" API classes

- **Date is a computationally expensive class**

Example from Bulka[2]:



Speed up due to a use of native call instead of the Java Date class

Java Memory Issues

AT&T Network Design and Performance Analysis Consulting

- Contributors to memory usage in Java:
  - Objects
  - Classes
    - Bytecode
    - JIT compiled code
    - Constant pool entries
    - Data structures representing methods and fields
  - Threads
  - Native data structures
    - e.g. OS-specific structures
- Too much memory usage will result in OS virtual memory overheads - and possible slow down in garbage collection

Java Memory Issues

- No method for calculating object size
  - Methods returning total memory and free memory of heap
  - Object size can be estimated indirectly using garbage collection, and heap memory methods
- Class loading can be tracked with *java -verbose:* lists all the classes being loaded

Key Recommendations

- Limit object creation (various techniques…)
- Do not use synchronized API classes if not needed
- Rewrite “heavy” API classes, if light ones are needed
- Apply various optimizations (books, papers).

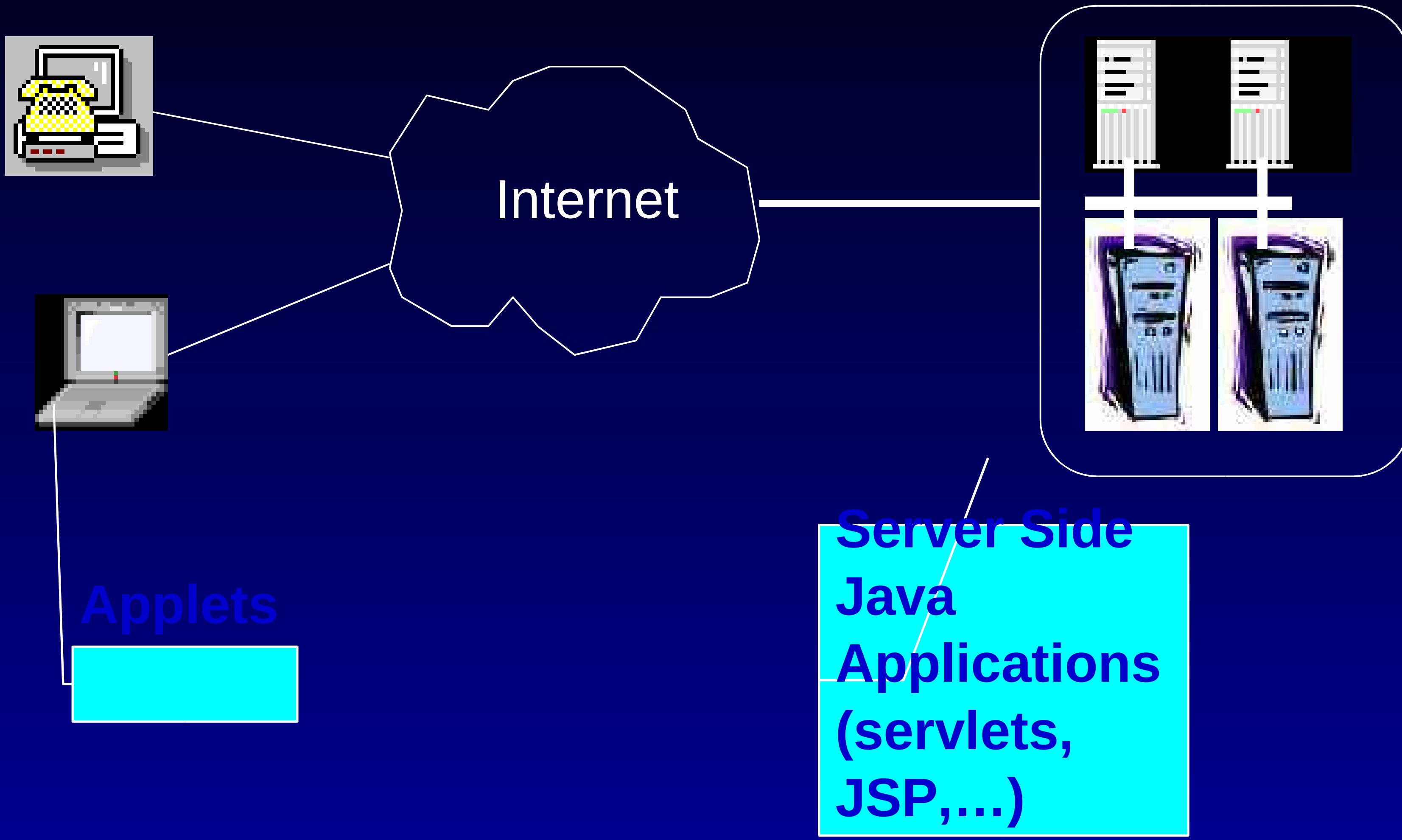
AT&T

Network Design and Performance Analysis Consulting

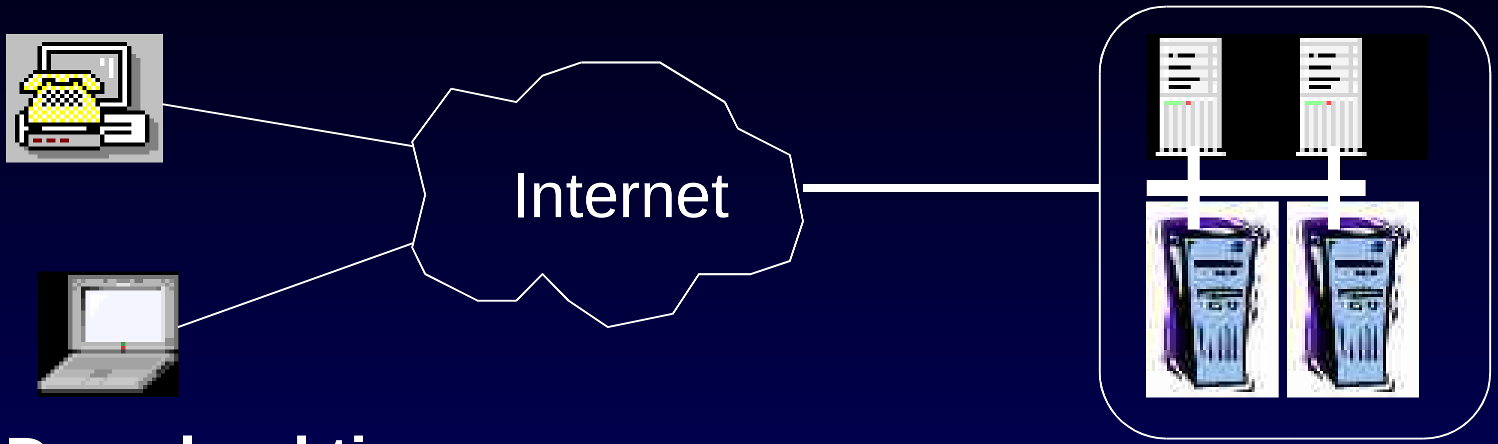
Performance/Capacity Analysis of Java Applications

July 27, 2001

Two kinds of Java apps



Applet performance Issues



- **Download time**
  - **downloads can be sped up using *jar* files instead of individual class files**
- **Dynamically linked classes that are downloaded when needed (will affect user response time on first use)**
- **Needs to be fast (usually used as a GUI)**
- **Usually no thread contention issues**

Capacity Analysis for Server Applications

- Typical industry problem:
  - Given a Java server application, size the server center to support volume of *N* requests per second.
  - Available data: measurement data from load testing at smaller volume and on systems smaller than “production” systems.

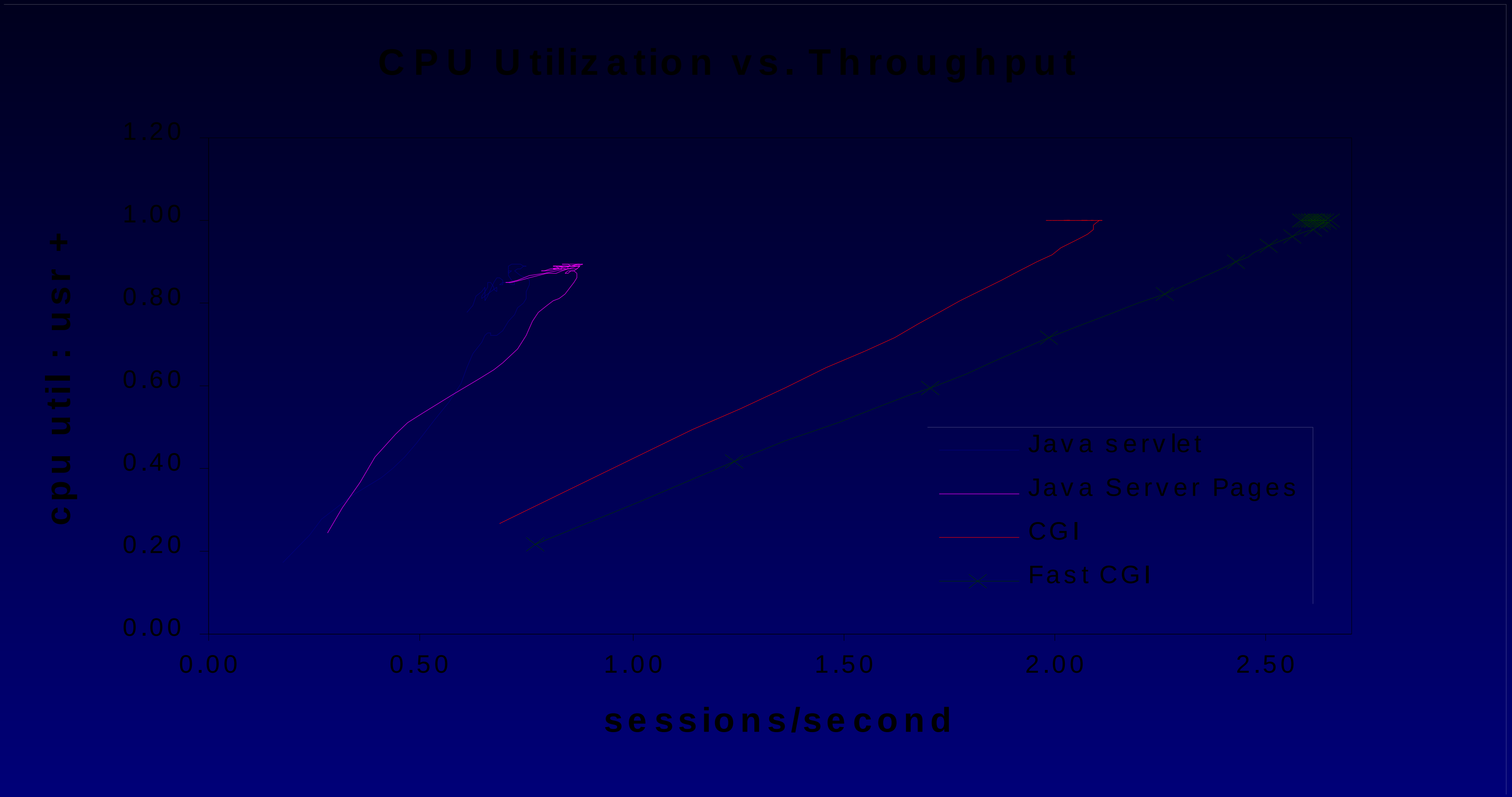
Issues in Java App capacity analysis

- Bottleneck capacity may not be that of a hardware resource
- Bottleneck may be
  - a piece of synchronized code
  - object creation, if a large number of objects are being created.
  - garbage collection, if large number of short-lived objects.
  - I/O (poorly coded)

Issues in Java App capacity analysis

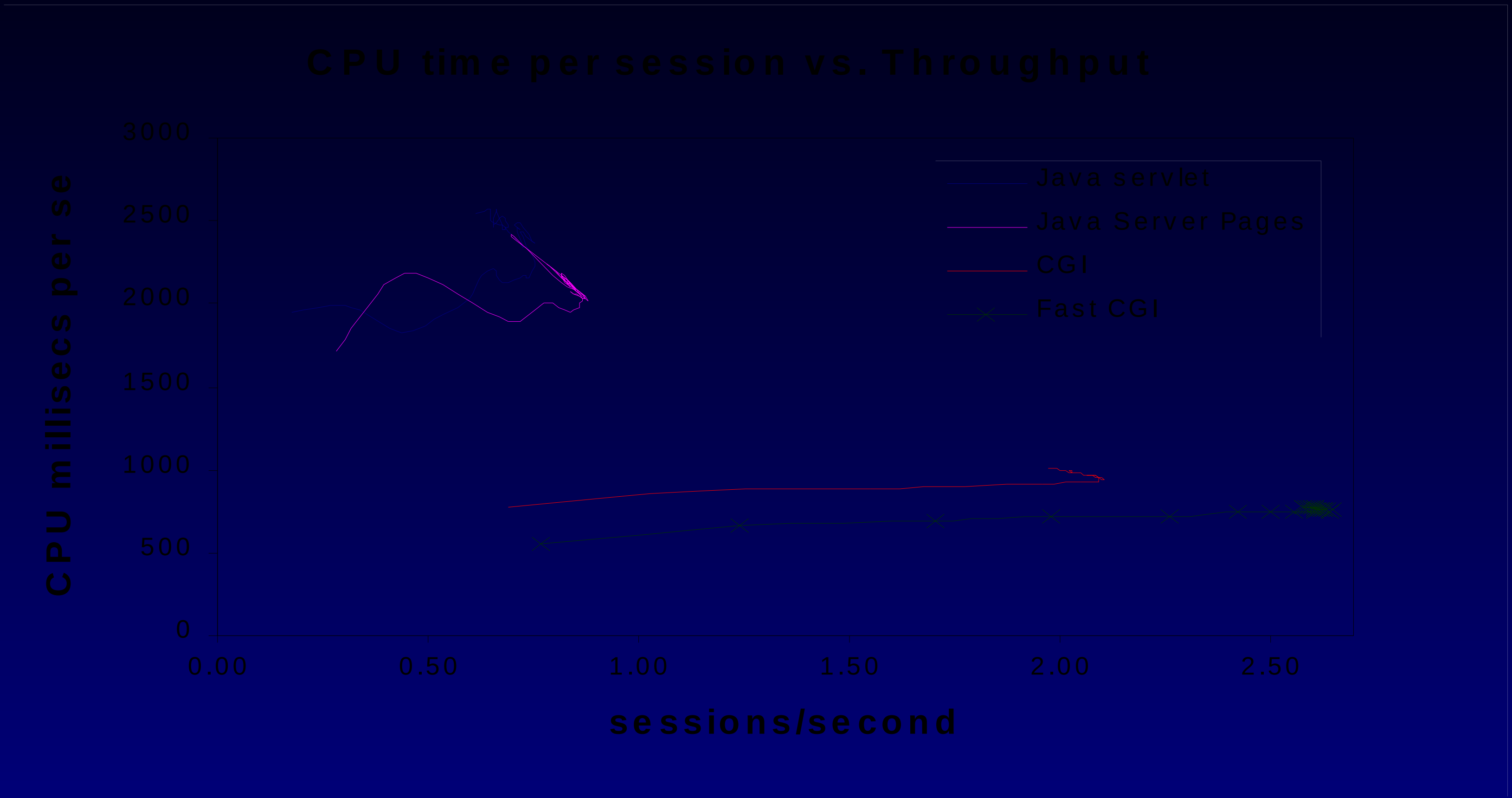
- Possibly no capacity increase with additional processors (threads)
  - CPU may not be the bottleneck
- Speed up due to more memory
  - Configure larger heap size
- Speed up with more servers
- CPU time per transaction may increase going from small to large number of users

Messaging Example



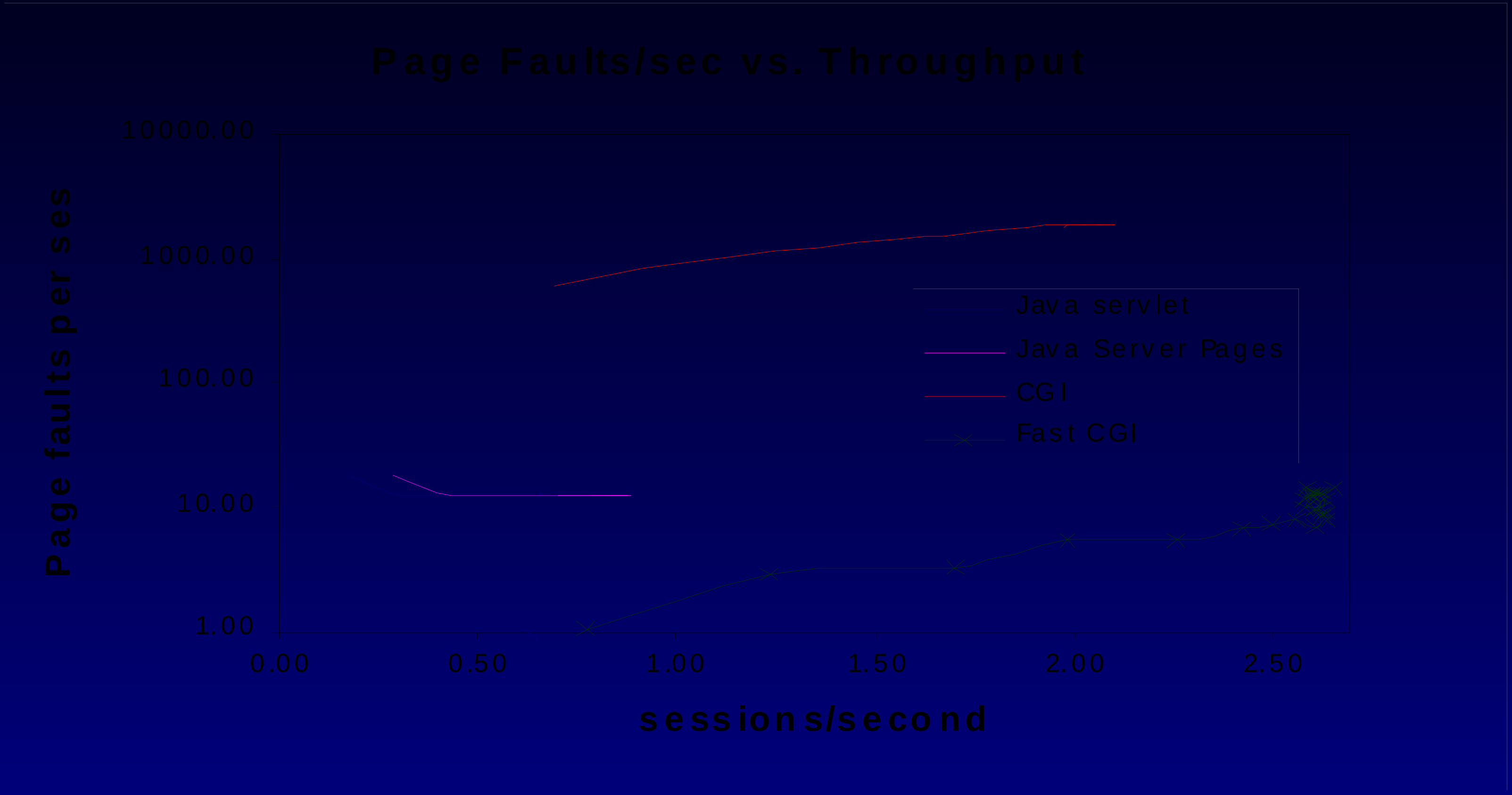
From Hansen, Mainkar, Reeser, 2001 [6]

Messaging Example



From Hansen, Mainkar, Reeser, 2001 [6]

Messaging Example



From Hansen, Mainkar, Reeser, 2001 [6]

Delay Analysis

- Apart from hardware resources, Java's software resources should also be analyzed as queues -
  - should take into account synchronized portion of code, and contention for it in a delay model.
- Should take into account garbage collection - service time in queues may be load-dependent

Previous Work

- Reeser[5] modelled a Java application with software “code lock” as a separate queue
  - “Abstract” bottleneck, - paper does not say which particular Java resource was the bottleneck
  - Model fits well

Reeser model example

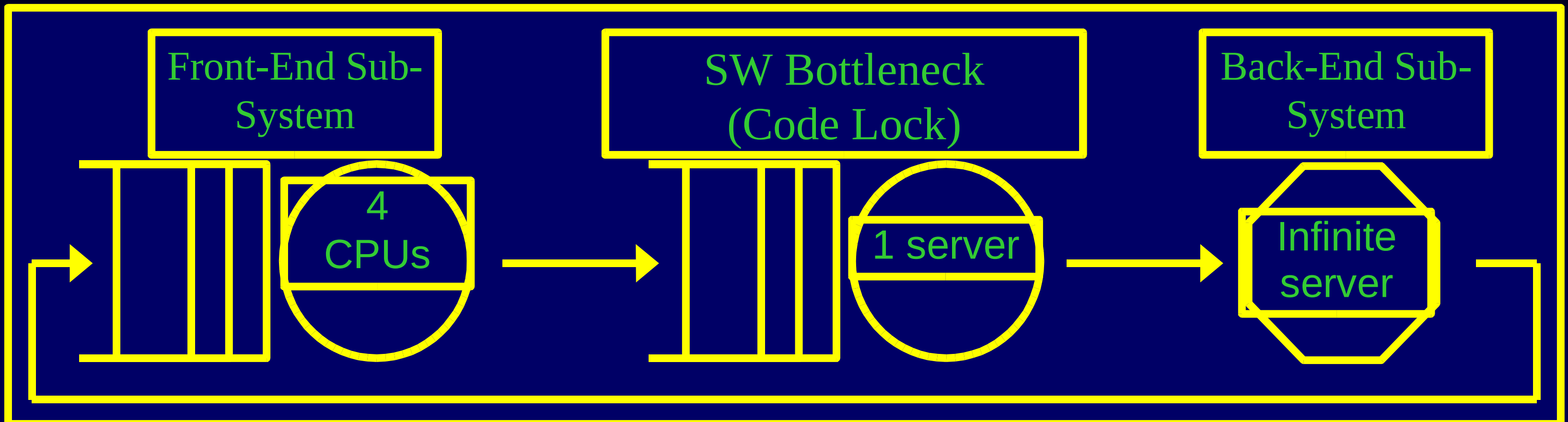


FIGURE 6: QUEUEING MODEL

Figure 6 from "Using Stress Test Results to Drive Performance Modeling: A Case Study in "Gray-Box" Vendor Analysis", ITC-16, Brazil, 2001.

Reeser model example

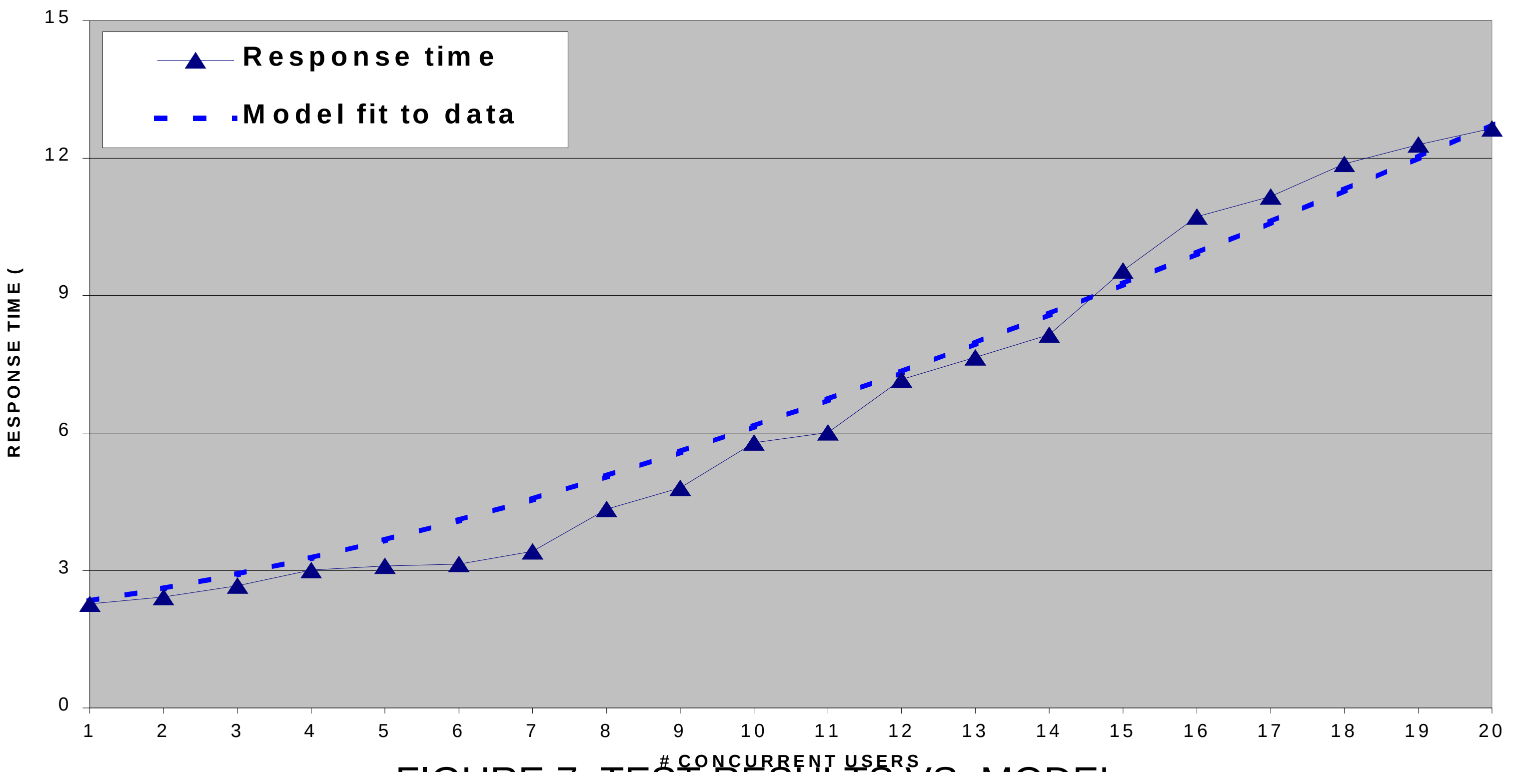


FIGURE 7: TEST RESULTS VS. MODEL

Figure 7 from "Using Stress Test Results to Drive Performance Modeling: A Case Study in "Gray-Box" Vendor Analysis", ITC-16, Brazil, 2001.

Profiling Tools

- Java VM comes with a profiler
  - Can report times spent in method calls, heap data etc.
  - Hard to read and understand
- Commercial Profilers
  - Jprobe, OptimizeIt
- Useful to developers to really tune their code
- Useful to analysts for understanding GC time and other bottlenecks

Future Directions

- Better models and techniques to analyze and predict capacity and performance of Java applications

References

1. B. Venners. *Inside the Java 2 Virtual Machine.* 2nd Ed. McGraw Hill, 1999.
2. D. Bulka. *Java Performance and Scalability, Vol. 1.* Addison-Wesley, 2000.
3. IBM Systems Journal Vol. 39, No.1, 2000. *Special Issue on Java Performance.*
4. J. Shirazi. *Java Performance Tuning.* O'Reilly, 2000.
5. P. Reeser, "Using Stress-Test Results to Drive Performance Modeling: A Case-Study in Vendor Gray-Box Modeling".
6. T. Hansen,V.Mainkar,P.Reeser, "Performance Comparison of Dynamic Web Platforms", SPECTS 2001.