Understanding and Tuning the Performance of Critical Sections with Program Analysis and Software Visualization Tools

Michael Dilip Shah

Advisor: Samuel Z. Guyer

Monday July 31, 2017

Why Care About Performance

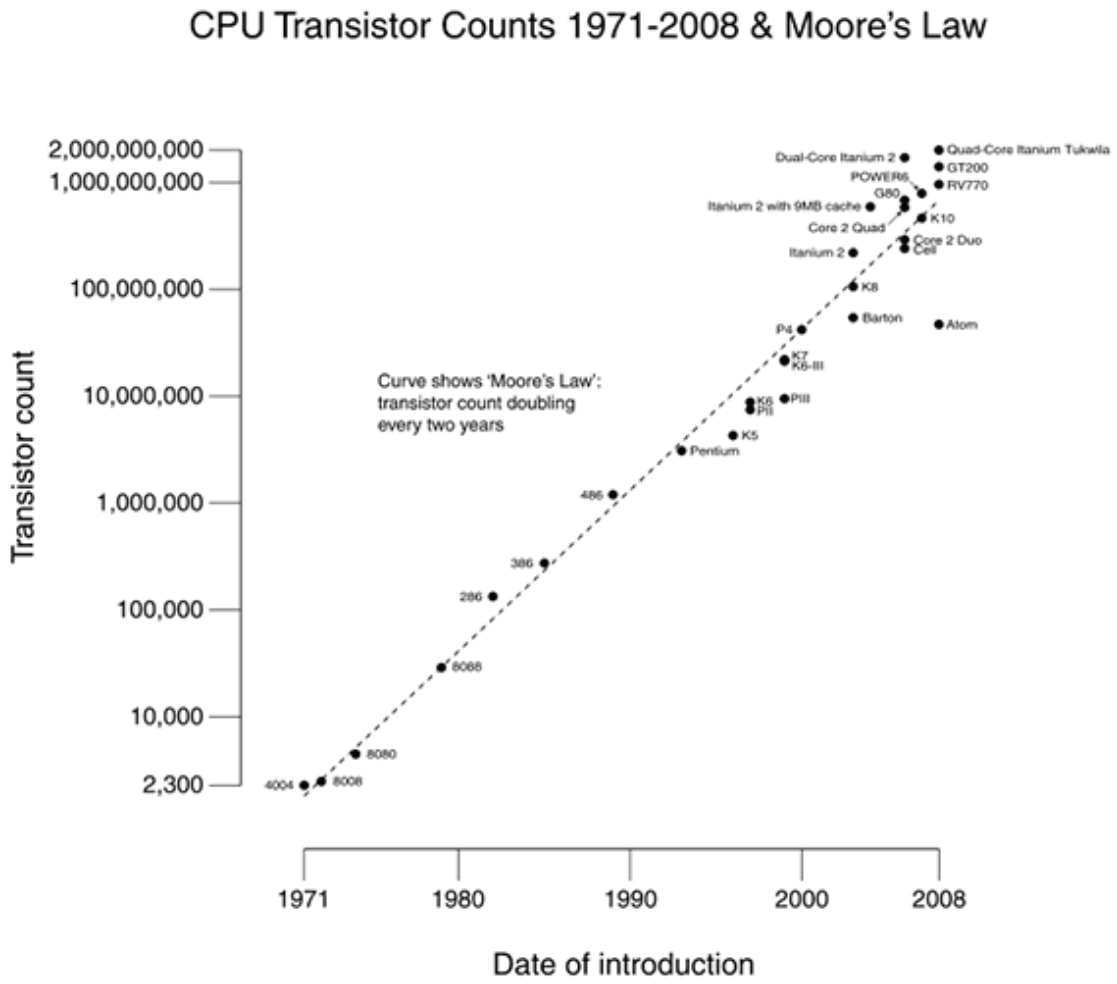
- Servers
- Mobile
- Games

Image Sources:
www.facebook.com
http://www.techcrok.com/
http://modloader-for-minecraft.en.softonic.com/

Moore's Law

"The number of transistors incorporated in a chip will approximately double every 24 months."

--Gordon Moore, Intel co-founder

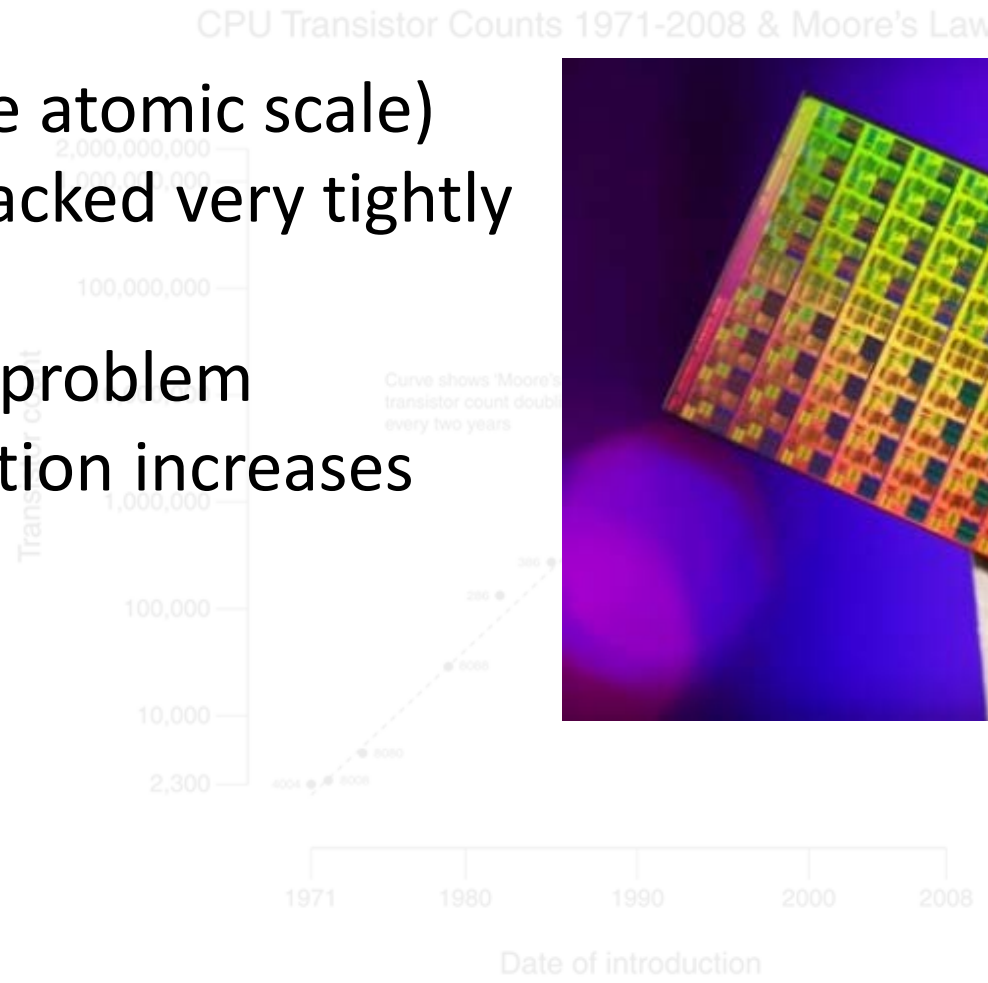


http://www-cs-faculty.stanford.edu/~eroberts/cs181/projects/2010-11/TechnologicalSingularity/pageviewa478.html?file=forfeasibility.html

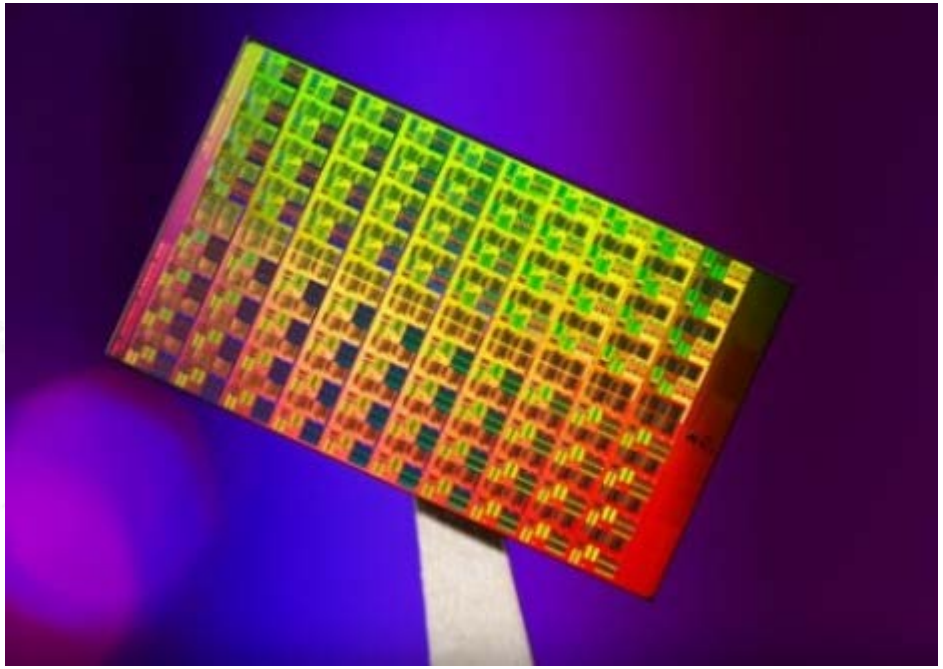
Moore's Law

"The number of transistors incorporated in a chip will approximately double every 24 months."

--Gordon Moore, Intel co-founder



- Physically (on the atomic scale) transistors are packed very tightly together
- Heat becomes a problem
- Energy consumption increases



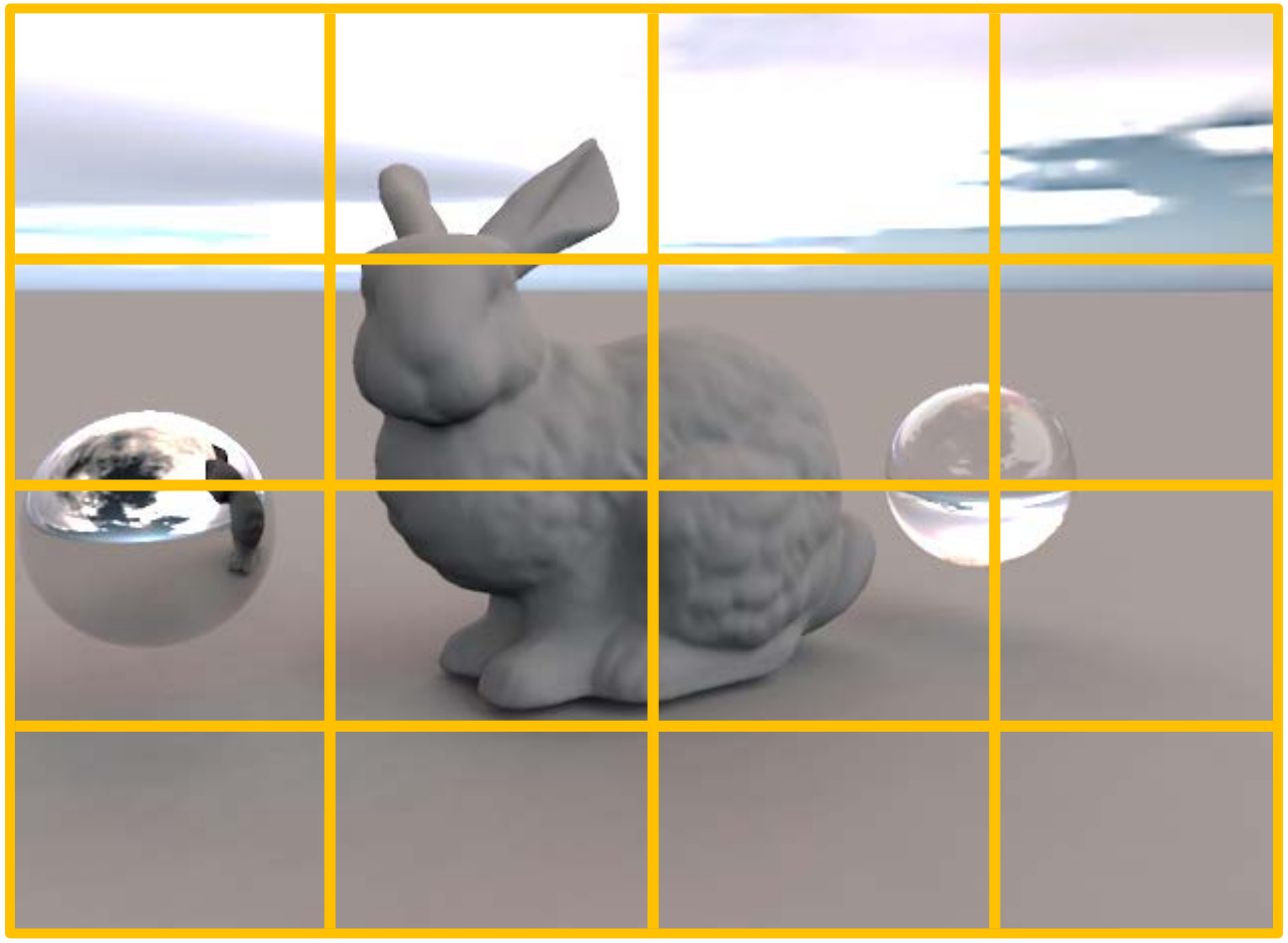
http://www-cs-faculty.stanford.edu/~eroberts/cs181/projects/2010-11/TechnologicalSingularity/pageviewa478.html?file=forfeasibility.html

Now we use multiple processors to increase performance

Rendering an Image in Parallel

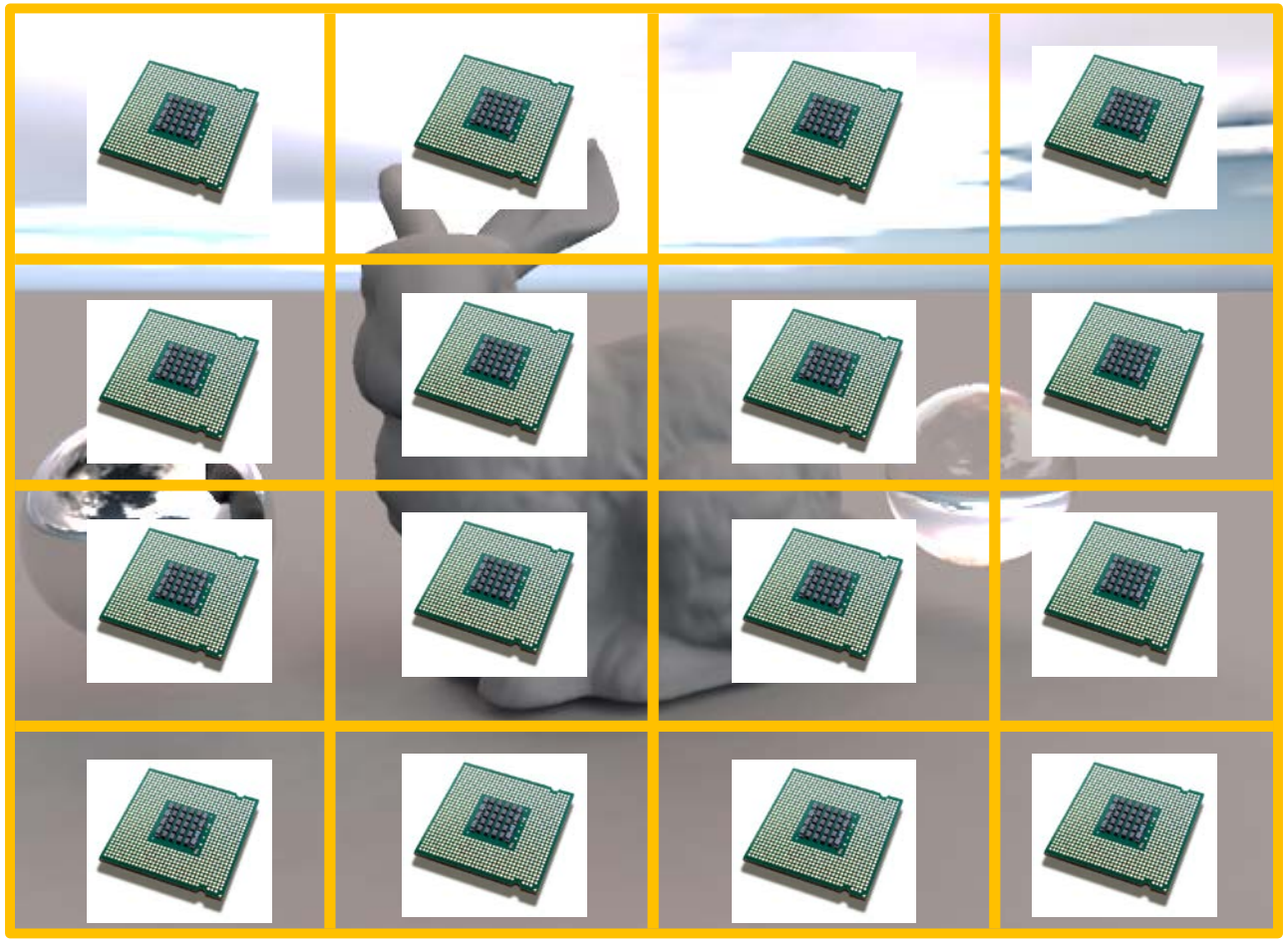
Sunflow – Java Multithreaded Raytracer

Setup 16 threads



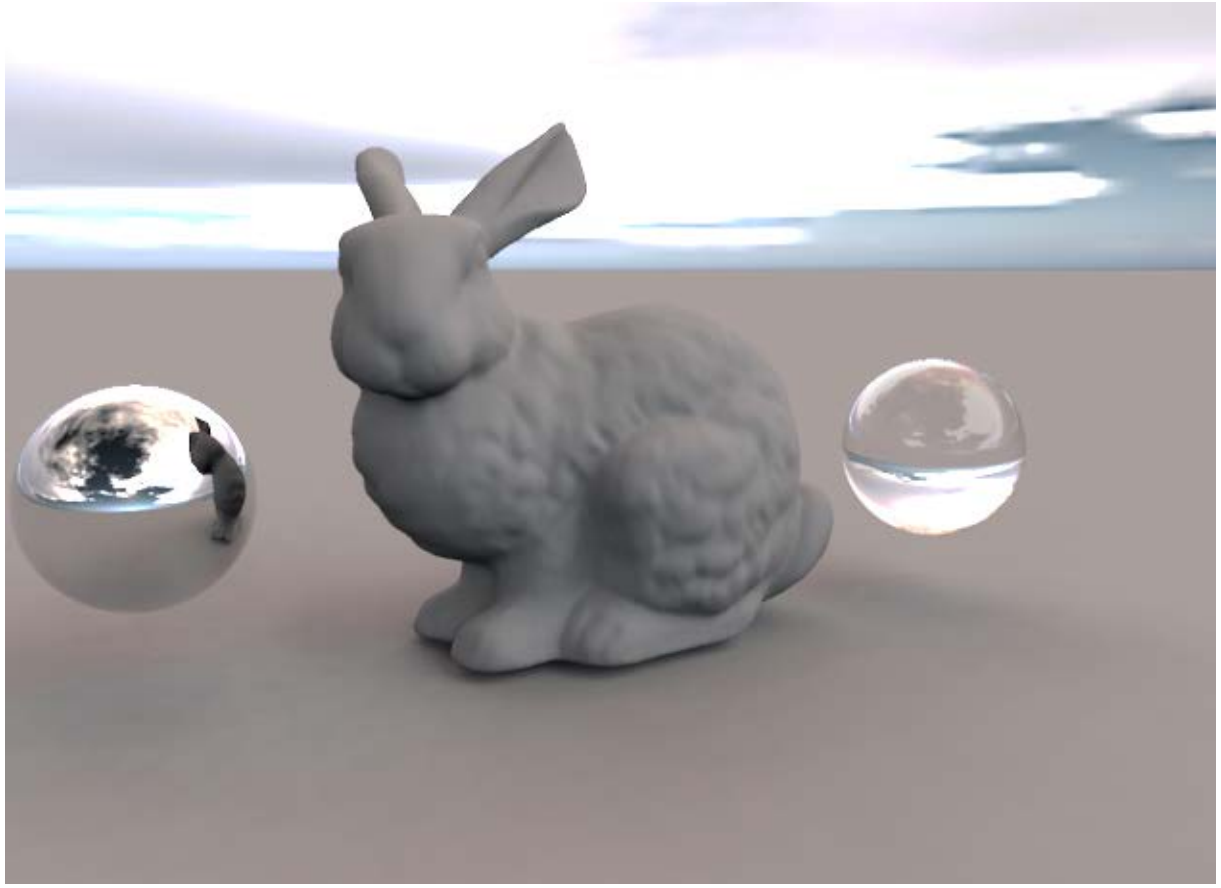
Sunflow – Java Multithreaded Raytracer

Divide and Conquer



Measure the performance

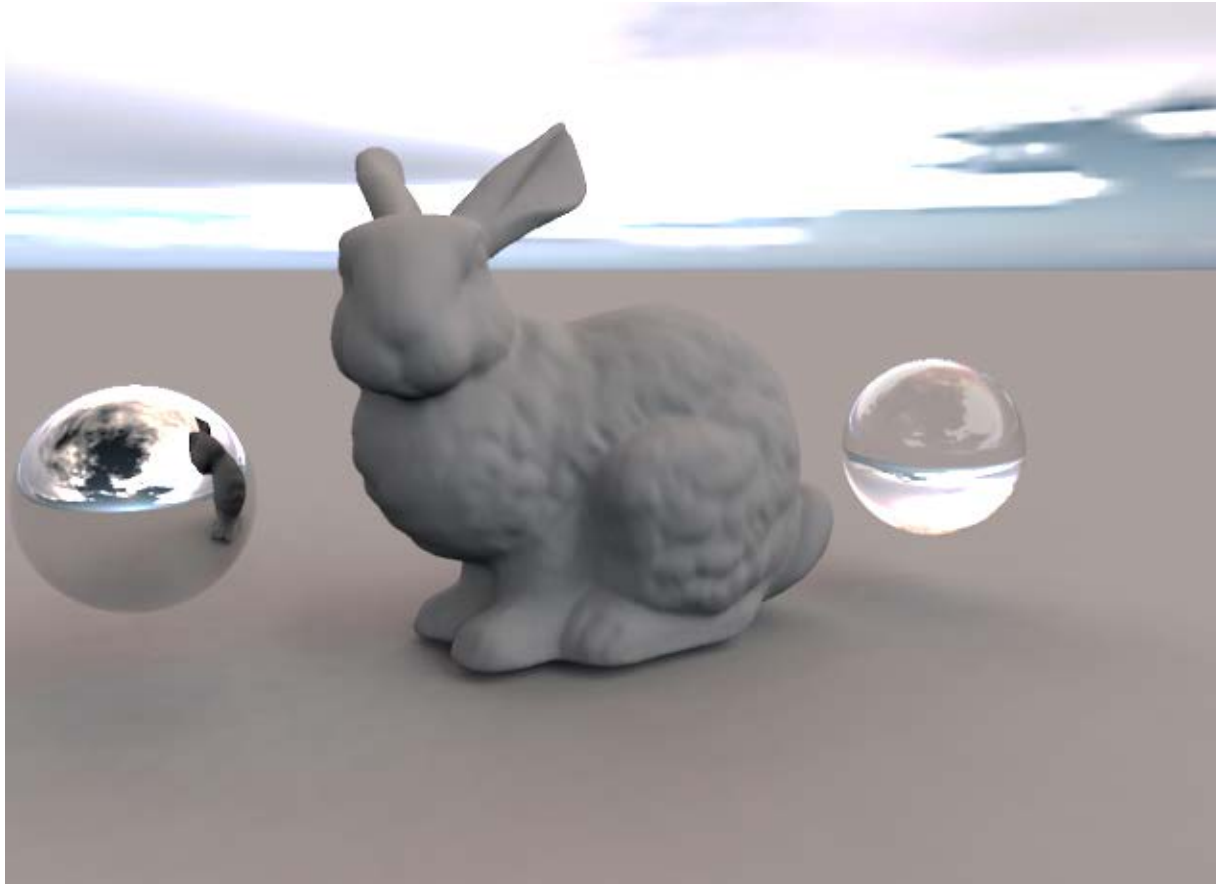
Threads	Time per frame
1	20 seconds
16	6 seconds



Measure the performance

Threads	Time per frame
1	20 seconds
16	6 seconds

- Why is this not 16 times faster?



Amdahl's Law

- We are limited in performance by the number of serial tasks in a program
- Ratio of serial tasks to parallel tasks dictates the maximum speedup.

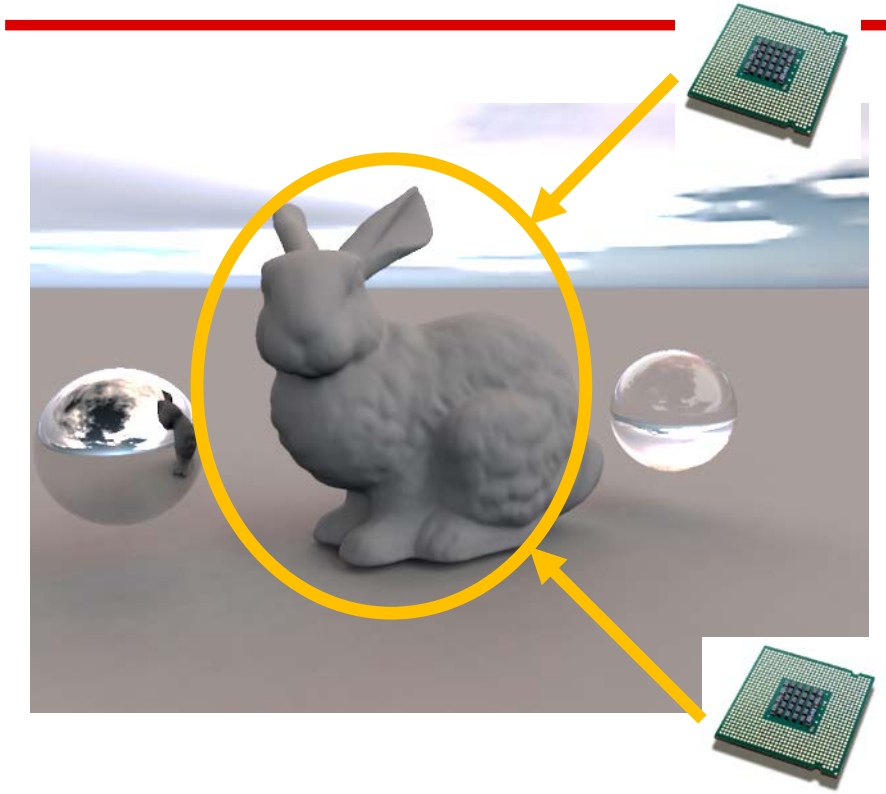
Amdahl's Law

$$\text{Speedup} = \frac{T_{\text{Serial runtime}}}{T_{\text{Parallel runtime}}}$$

Resources in a program are shared

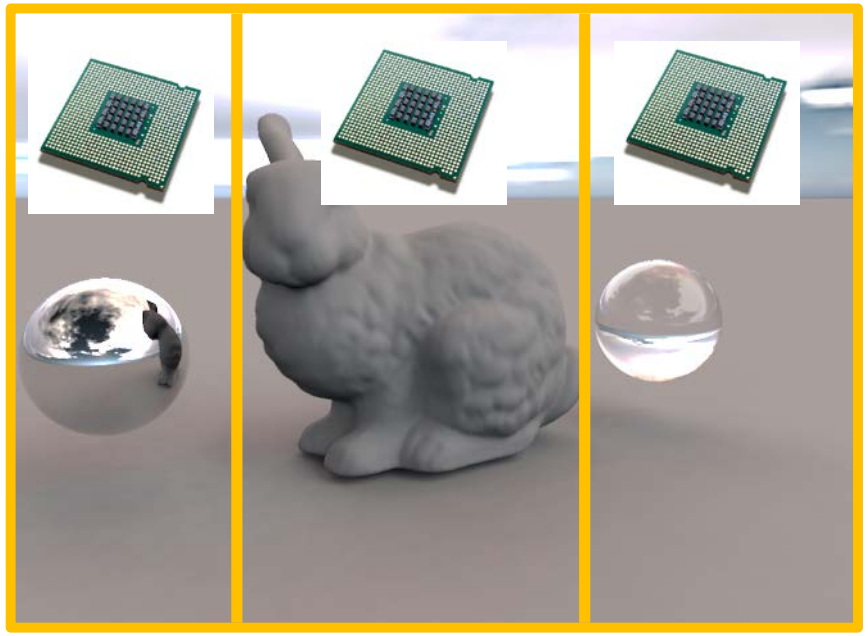
- Only 1 bunny in this scene

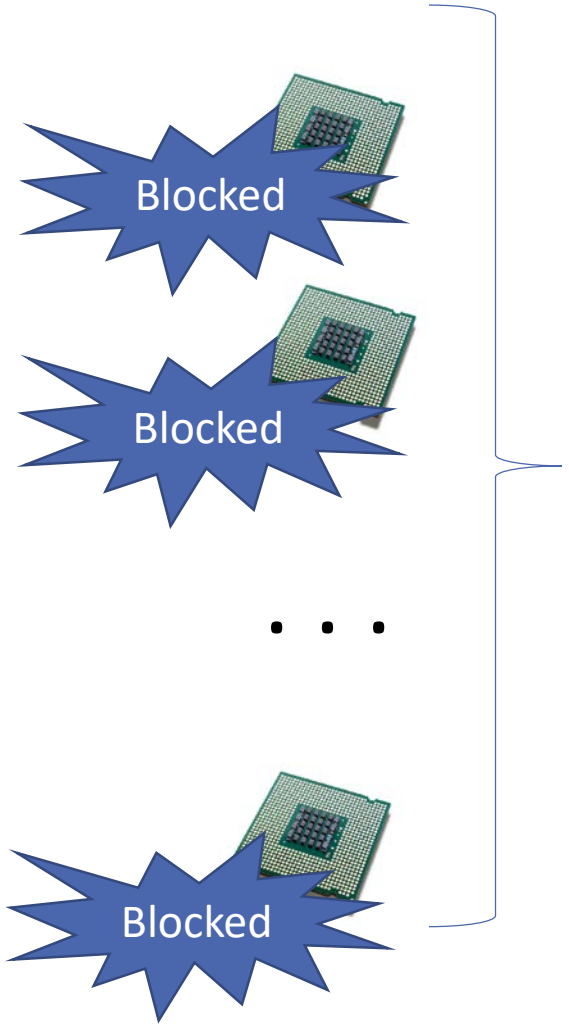
Resources in a program are shared



- Only 1 bunny in this scene
- Attempting to update a shared resource by 2 or more threads at the same time results in a data race

Threads put in a waiting queue





- A few threads work
- Threads are blocked in order to enforce correctness

Java Concurrency – Synchronized Method Example

```
synchronized void modifyBunny()
{

    //  . . .
    //  modify geometry for the bunny
    //  . . .

}
```

Synchronized – puts a lock over shared resources

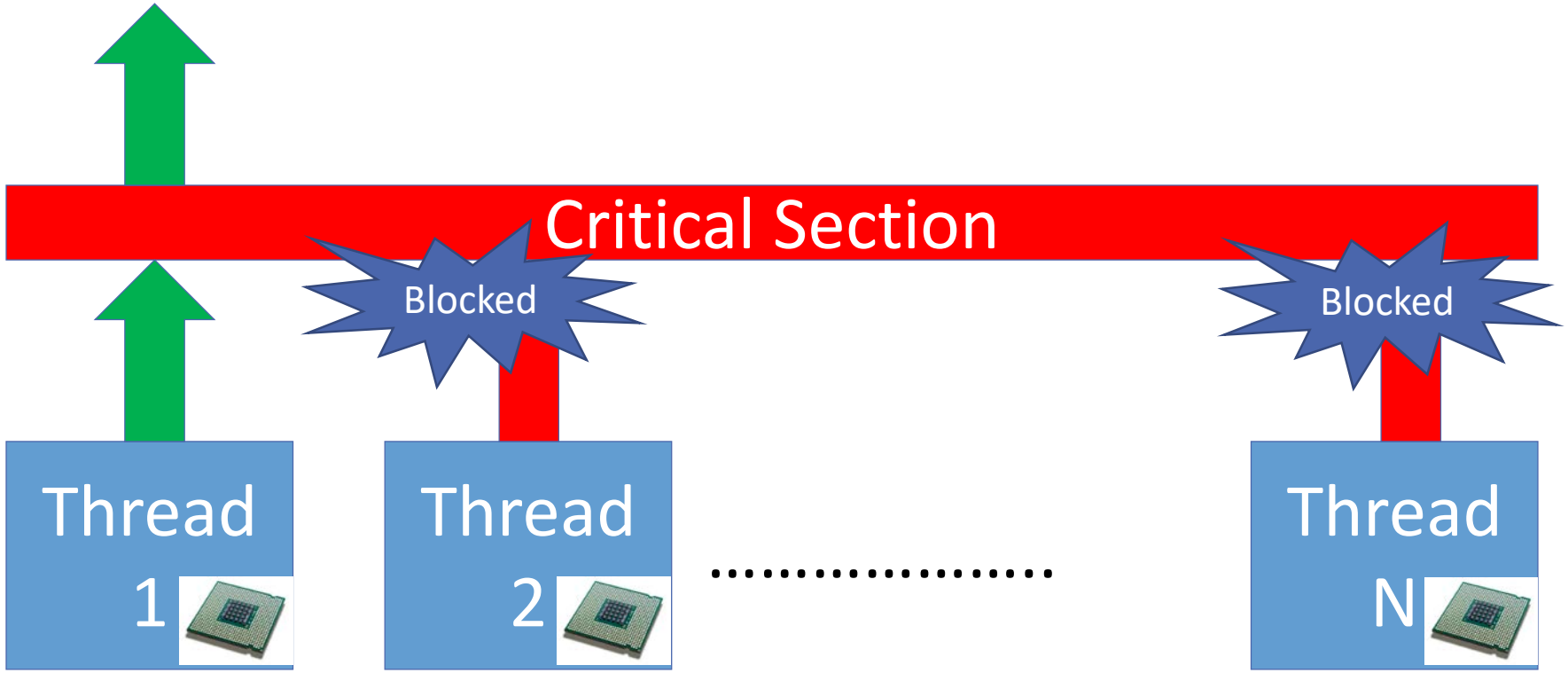
```
synchronized void modifyBunny()
{

        //  . . .
        //  modify geometry for the bunny
        //  . . .

}
```

Critical Sections Defined

- A section of code that is executed by only one thread at a given time.



Correctness (can be) Easy
Performance Hard

```
public class DrawPicture{
        DrawPicture(…) {…}
        lighting (…) {…}
        tesselate(…) {…}
        shadows (…) {…}
        geometry(…) {…}
        getPixel (…) {…}
        getNumLights (…) {…}
}
```

Correctness (can be) Easy

Performance Hard

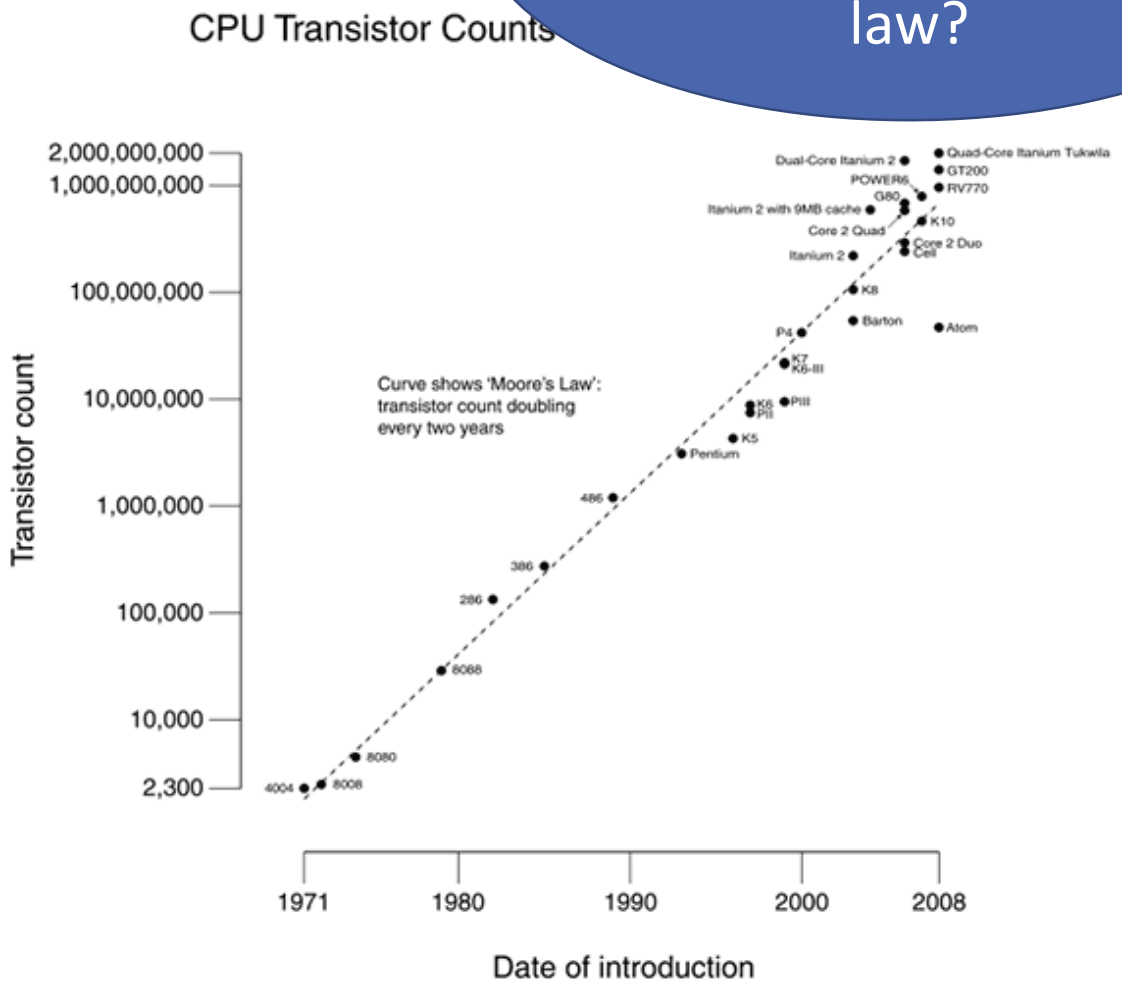
Good job—no data races here!

```
public class DrawPicture{
                    DrawPicture(…) {…}
 synchronized       lighting (…) {…}
 synchronized       tesselate(…) {…}
 synchronized       shadows (…) {…}
 synchronized       geometry(…) {…}
 synchronized       getPixel (…) {…}
 synchronized       getNumLights (…) {…}
}
```

Correctness (can be) Easy

Performance Hard

Your program runs sequentially– did you forget about Amdahl's law?



http://www-cs-faculty.stanford.edu/~eroberts/cs181/projects/2010-11/TechnologicalSingularity/pageviewa478.html?file=forfeasibility.html

The Big Picture With Multithreaded Code

- We want our software to run fast
- Writing multithreaded code correctly is difficult
- We use **synchronized** code when a common resource is shared amongst threads.

The Problem

Real world programmers do not always understand the **performance** of their code in **critical sections**.

Related Work

- 2012, PLDI - Understanding and Detecting Real-World Performance Bugs
  - 332 previously unknown performance problems are found in the latest versions of MySQL, Apache, and Mozilla applications
  - “Developers frequently use inefficient code sequences that could be fixed by simple patches. These inefficient code sequences can cause significant performance degradation and resource waste, referred to as performance bugs. Meager increases in single threaded performance in the multi-core era and increasing emphasis on energy efficiency call for more effort in tackling performance bugs. “

Related Work

- 2012, PLDI - Understanding and Detecting Real-World Performance Bugs
  - **332 previously unknown performance problems** are found in the latest versions of MySQL, Apache, and Mozilla applications
  - "Developers frequently use **inefficient code sequences that could be fixed by simple patches**. These inefficient code sequences can **cause significant performance degradation and resource waste**, referred to as performance bugs. Meager increases in single threaded performance in the multi-core era and increasing emphasis on energy efficiency call for more effort in tackling performance bugs. "

Related Work

- 2012, PLDI - Understanding and Detecting Real-World Performance Bugs
  - **332 previously unknown performance problems** are found in the latest versions of MySQL, Apache, and Mozilla applications
  - "Developers frequently use **inefficient code sequences that could be fixed by simple patches**. These inefficient code sequences can **cause significant performance degradation and resource waste**, referred to as performance bugs. **Meager increases in single threaded performance in the multi-core era** and increasing emphasis on energy efficiency **call for more effort in tackling performance bugs.** "

Related Work

- 2013, ICSE - Toddler: Detecting Performance Problems via Similar Memory-Access Patterns
  - “detecting performance bugs usually requires time-consuming, manual analysis of execution profiles. The human effort for performance analysis limits the number of performance tests analyzed and enables performance bugs to easily escape to production. “

Related Work

- 2013, ICSE - Toddler: Detecting Performance Problems via Similar Memory-Access Patterns
  - "**detecting performance bugs usually requires time-consuming, manual analysis of execution profiles**. The human effort for performance analysis limits the number of performance tests analyzed and **enables performance bugs to easily escape to production**. "

Thesis Statement

Static, dynamic, and software visualization analysis tools focused on critical sections are needed to uncover performance variability in critical sections to avoid unintended software hangs

Thesis Statement

Static, dynamic, and software visualization analysis tools focused on critical sections are needed to uncover performance variability in critical sections to avoid unintended software hangs

A potential bottleneck – remember only 1 thread of execution

Thesis Statement

Static, dynamic, and software visualization analysis tools focused on critical sections are needed to **uncover performance variability** in critical sections to avoid unintended software hangs

If we cannot estimate time accurately – does that impact user experience?

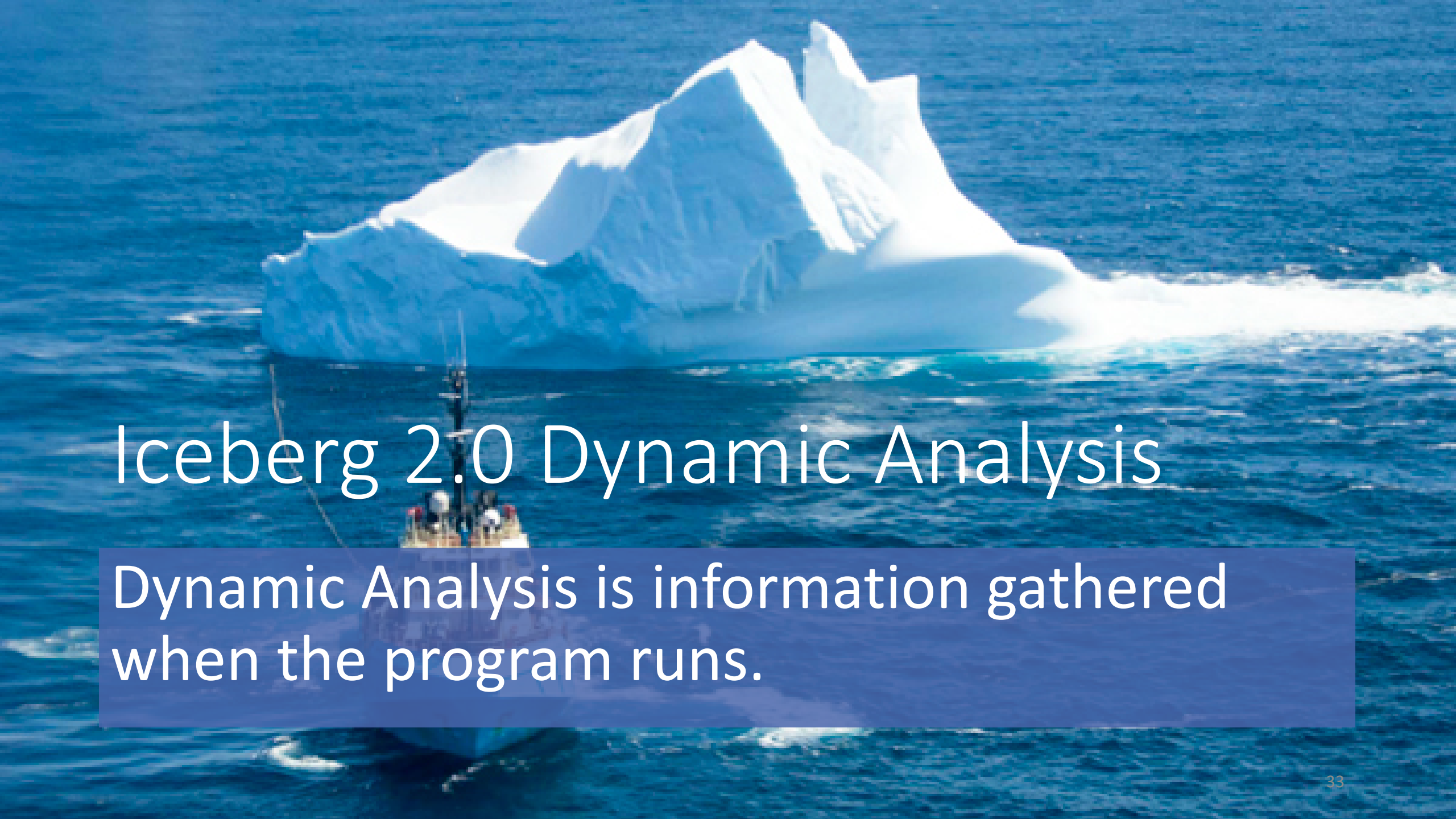
Thesis Statement

Static, dynamic, and software visualization analysis tools focused on critical sections are needed to uncover performance variability in critical sections to avoid unintended software hangs

New tools and analysis will provide insights into how to solve this problem.

Program Analysis

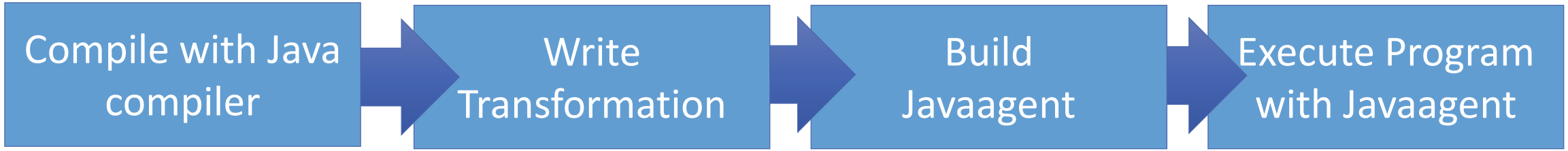
- Static Analysis
- Dynamic Analysis



Iceberg 2.0 Dynamic Analysis

Dynamic Analysis is information gathered when the program runs.

Bytecode Instrumentation with Javassist

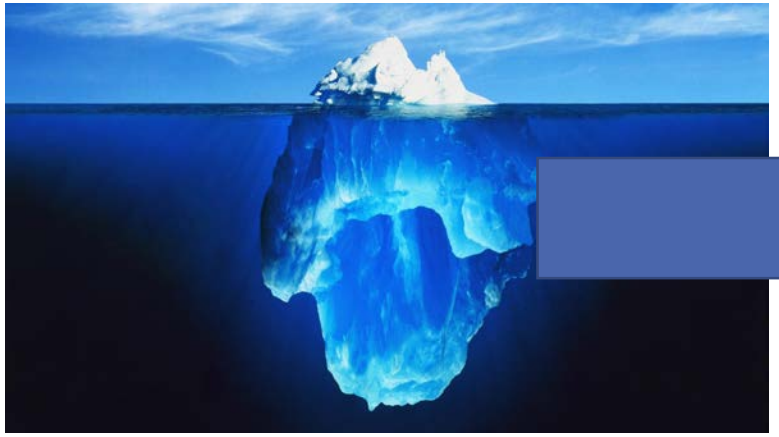


Compile with Java compiler

```
Javac -classpath ./*.java
```

Compile with Java compiler → Write Transformation

- Leverage our previous static analysis to feed our dynamic analysis which methods to instrument



```
sun.awt.X11GraphicsConfig
java.security.cert.Certificate
sun.java2d.loops.DrawRect
org.sunflow.core.Camera
javax.swing.JRadioButton
java.util.EventObject
java.awt.event.ComponentEvent
java.lang.RuntimeException
```

Compile with Java compiler → Write Transformation

- Use the Javassist bytecode engineering library to transform actual Java bytecode .
  - Code that will be injected into critical sections
  - Care taken to minimally perturb the system

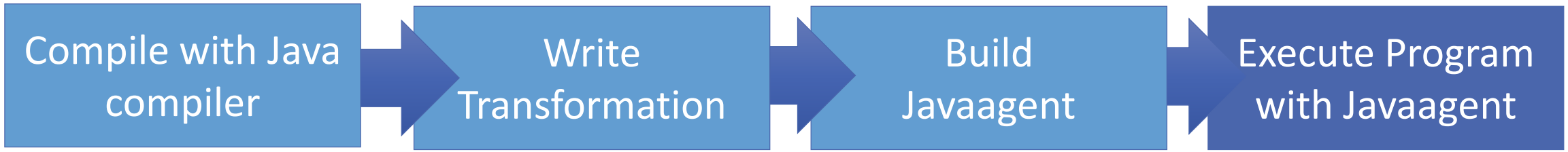
```
1  void synchronized foo(){
2      elapsedTime = System.nanoTime(); // Added Instrumenta
3      // Body of code
4      // ..
5      //
6      elapsedTime = System.nanoTime() - elapsedTime; // Adde
7      ProfilingController.log(\""+m_name+"\",elapsedTime);
8  }
```

Method Entry Probe

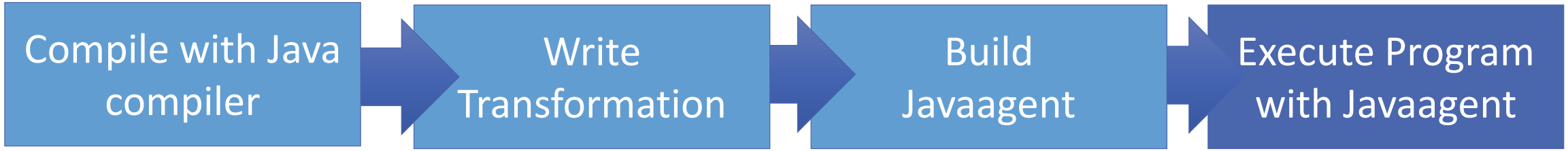
Method Exit Probe

- Build transformation into a .jar file.

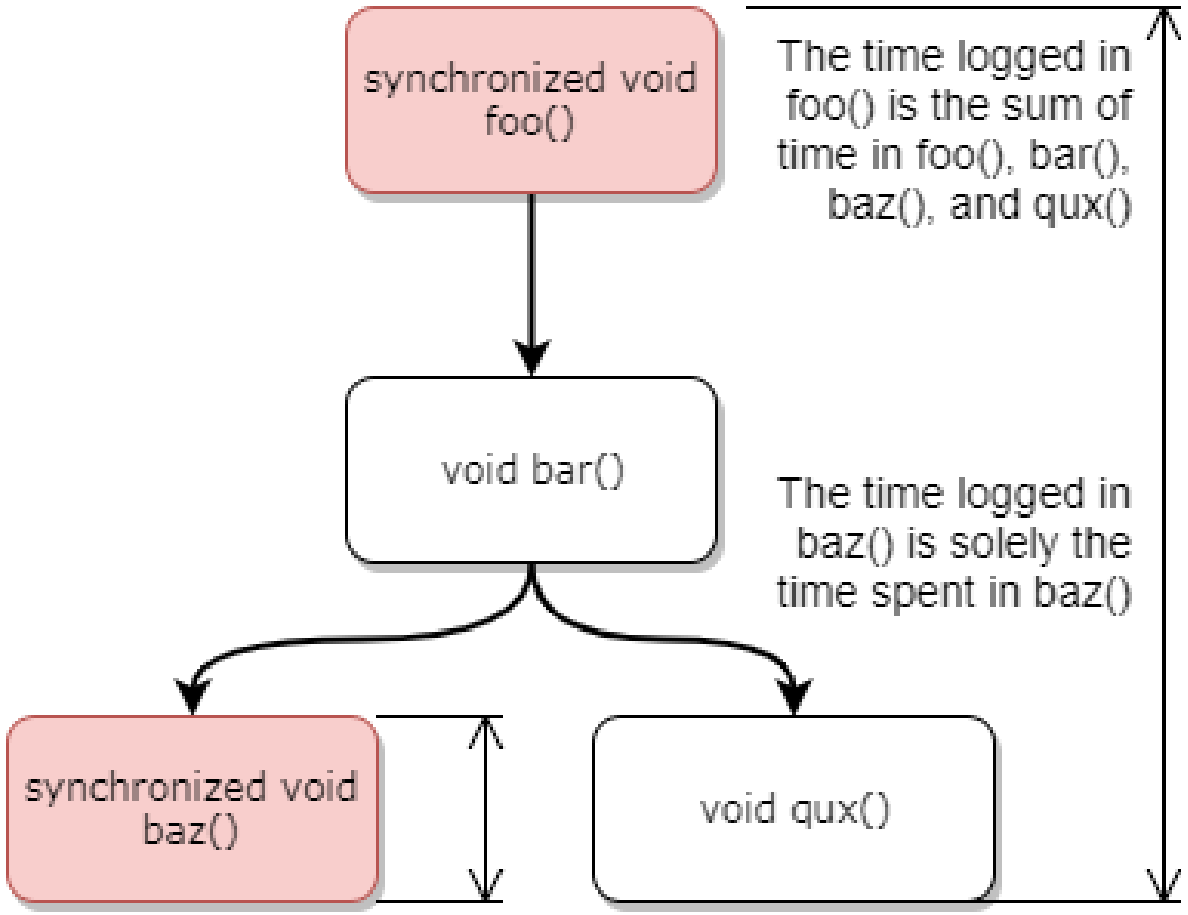
```
javac -cp ./javassist.jar:. *.java
```

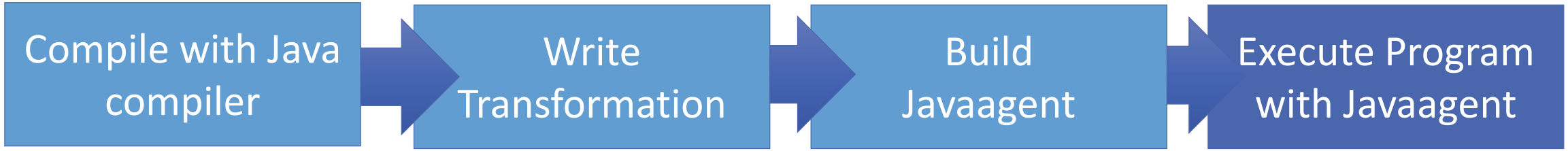



```
java -cp . -javaagent:/iceberg2.jar -jar sunflow.jar
```

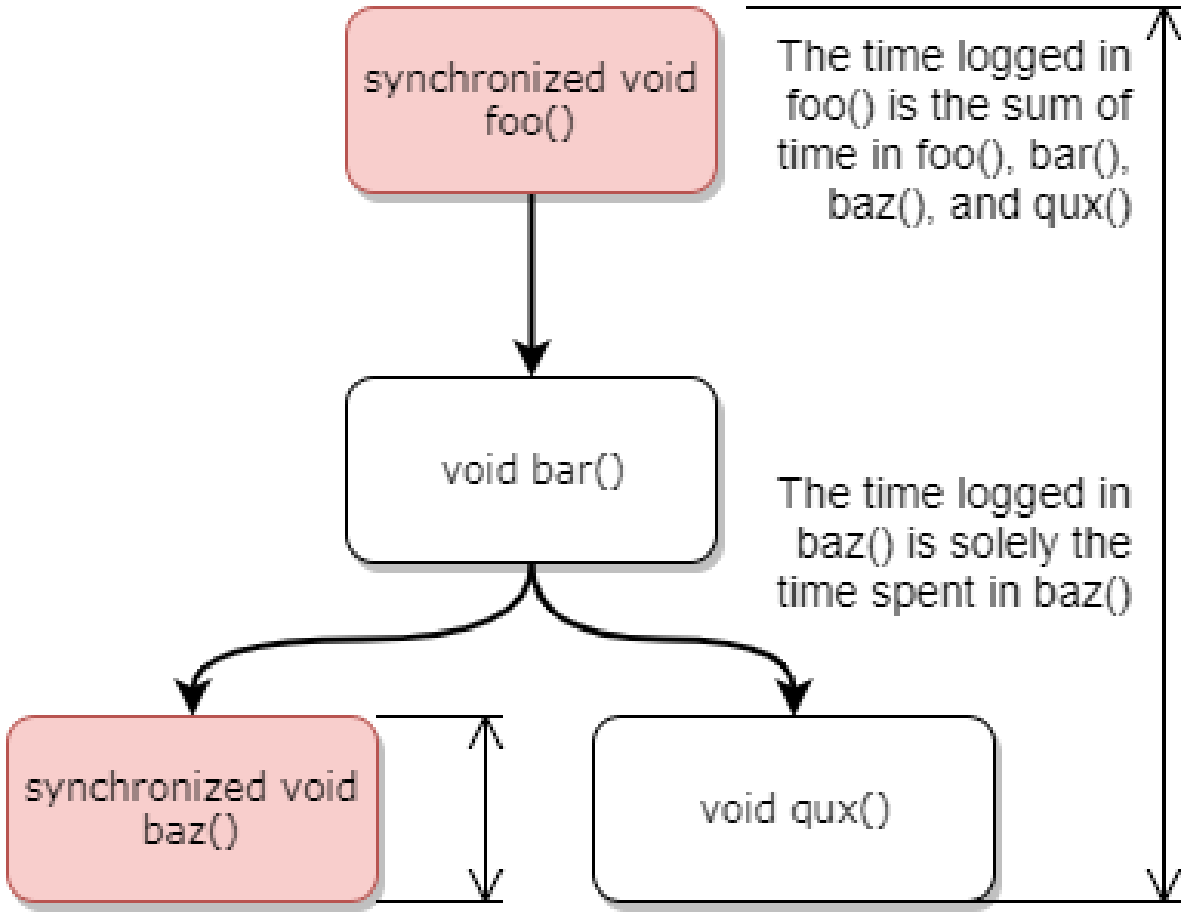


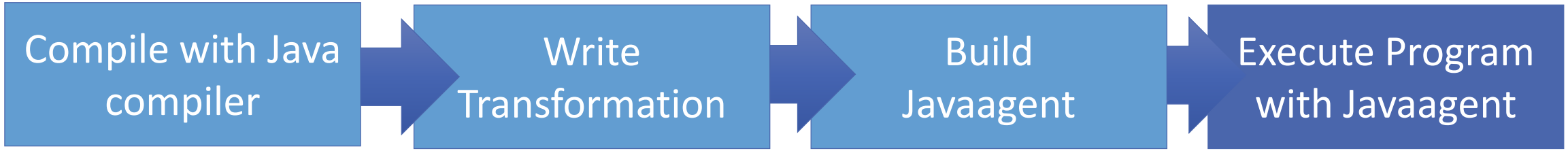
- Record time spent within critical sections



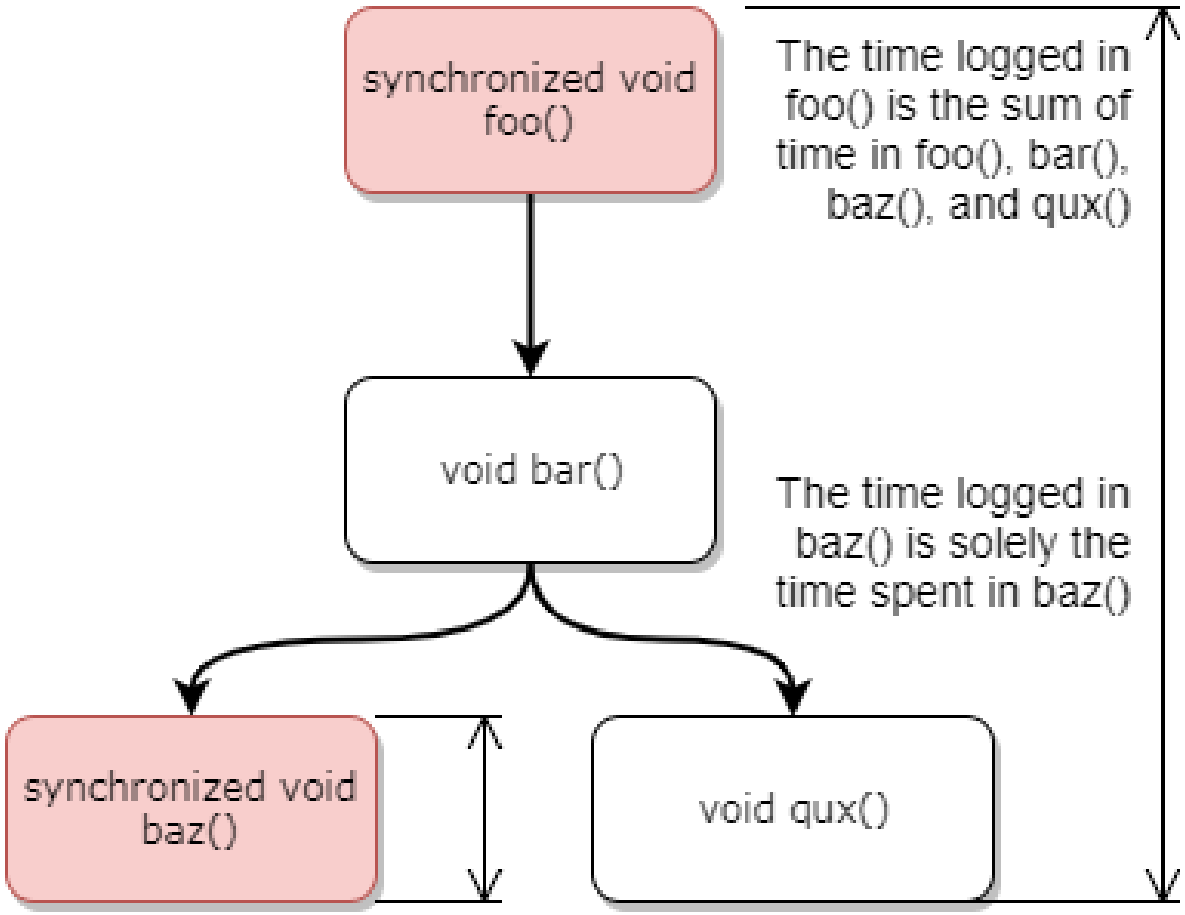


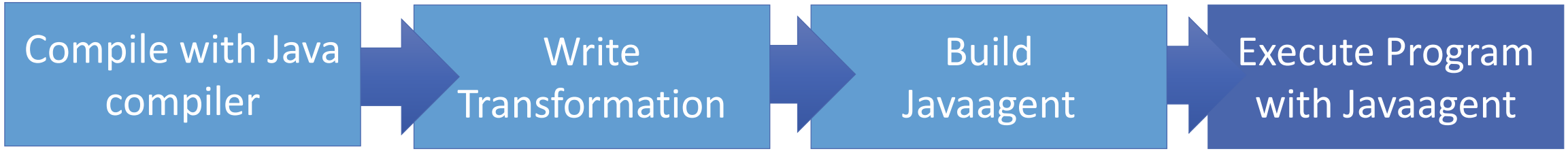
- Record time spent within critical sections
  - Gathering entry and exits from methods



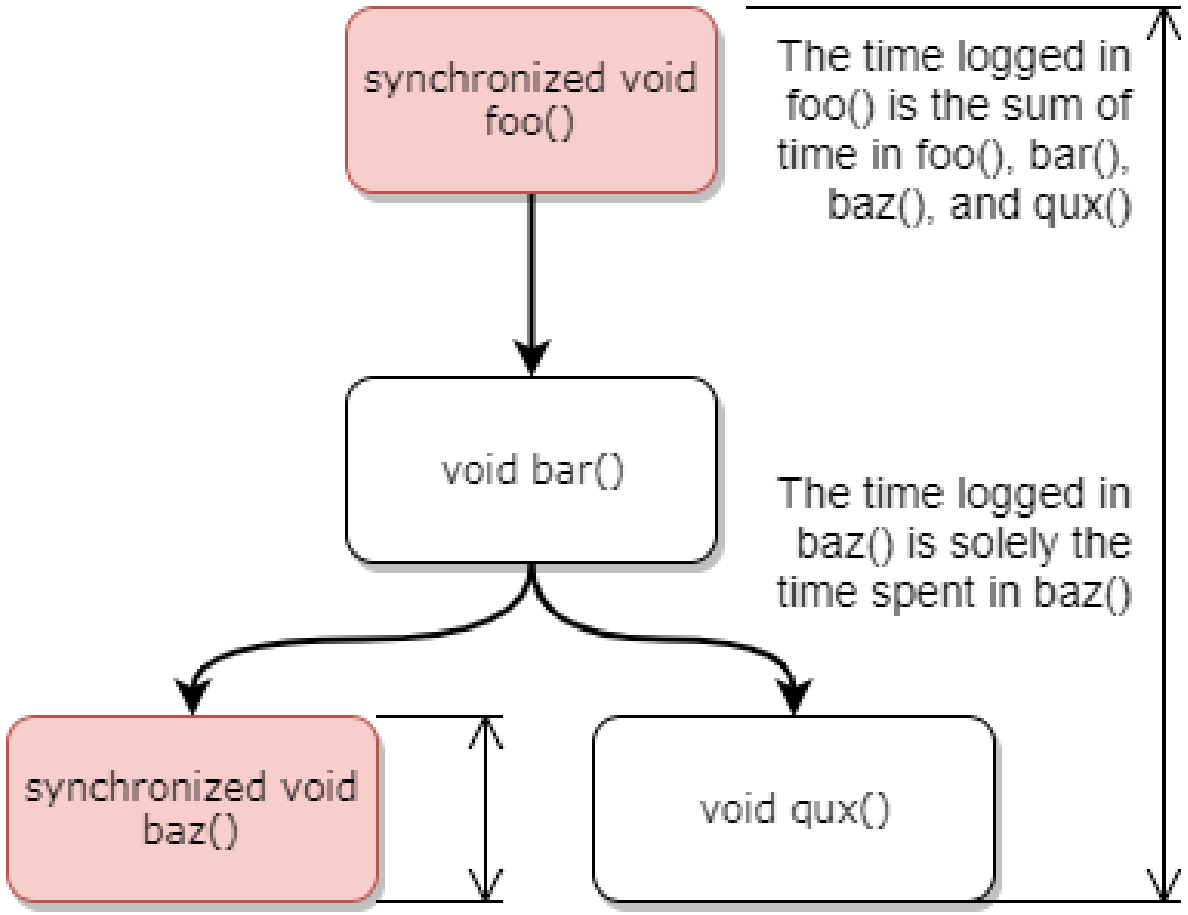


- Record time spent within critical sections
  - Gathering entry and exits from methods
- Variety of power in instrumentation
  - Can record stack
  - Can record thread contention
  - Can record full call tree





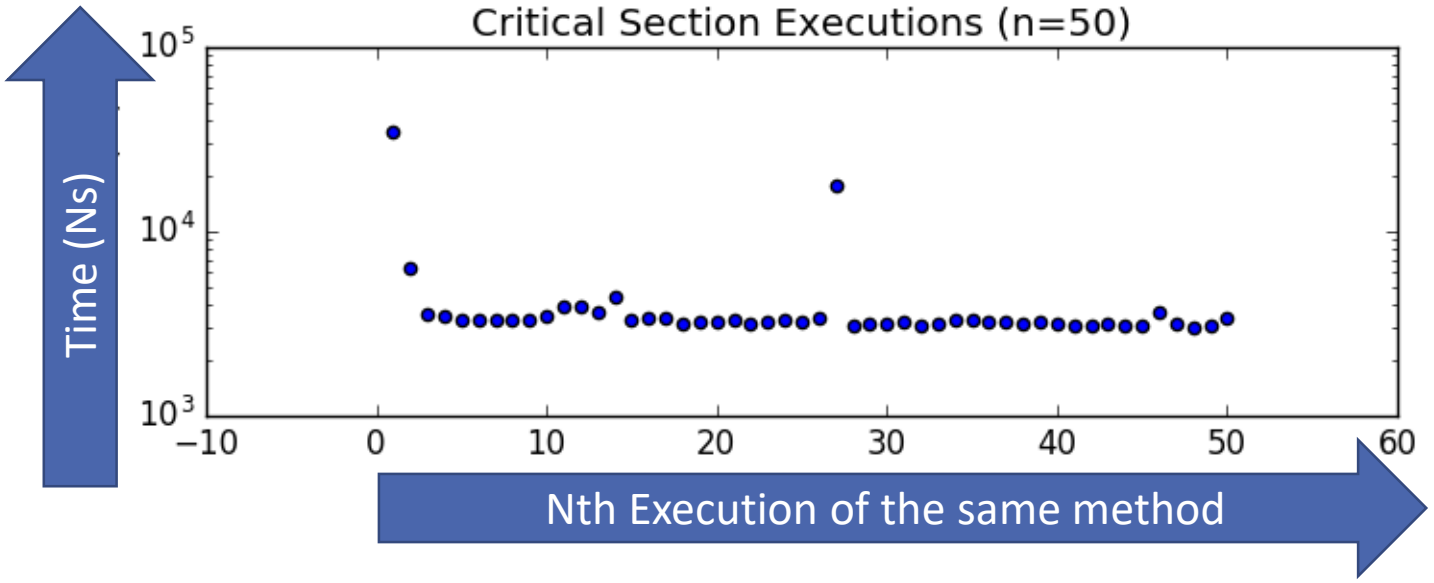
- Record time spent within critical sections
  - Gathering entry and exits from methods
- Variety of power in instrumentation
  - Can record stack
  - Can record thread contention
  - Can record full call tree
- Interested in **time** for now



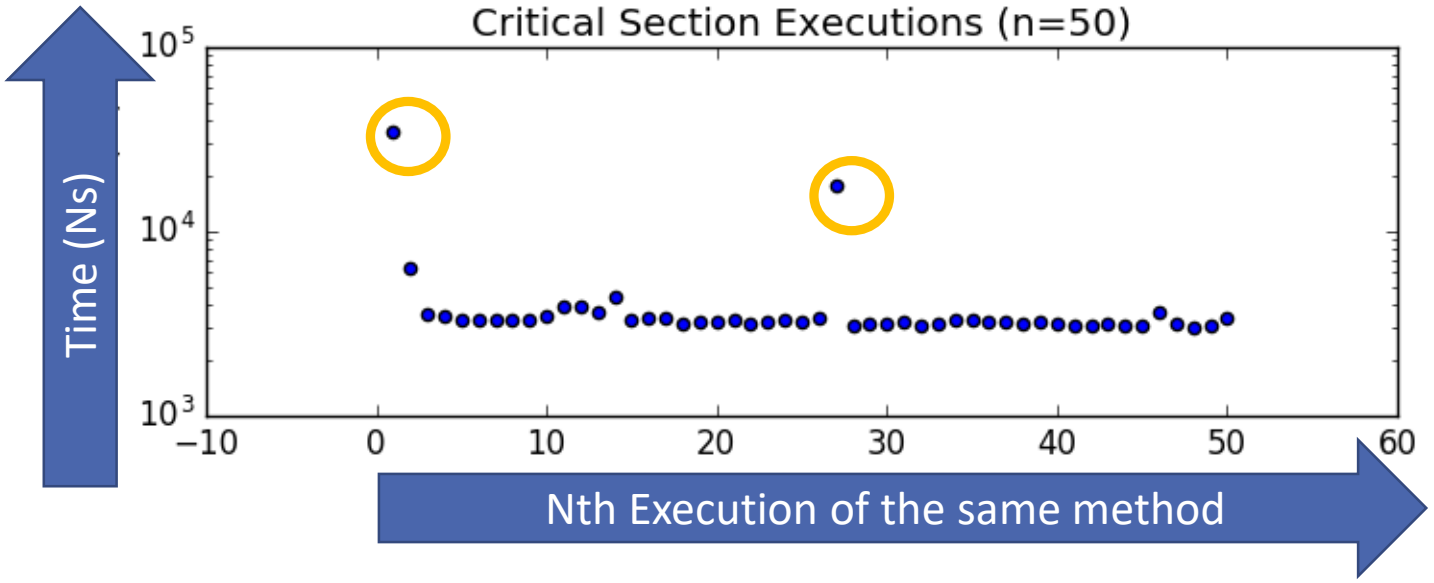
Calibrating our tool with Microbenchmarks

- 28 total microbenchmarks with synchronized methods
  - Input/Output
  - Networking
  - Allocations
  - Data Structures
  - Branching behavior
  - Nested synchronization
  - Sorting
- Purpose to validate our tool works

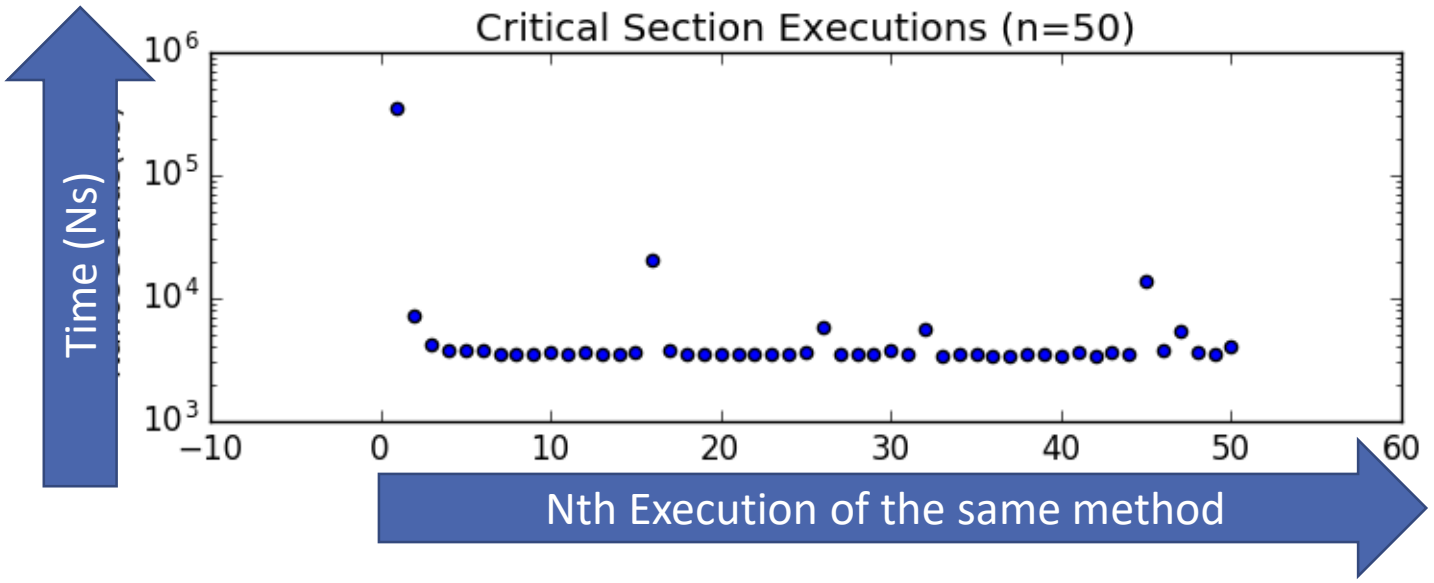
SortArrayListTest.addRandom5LetterWord



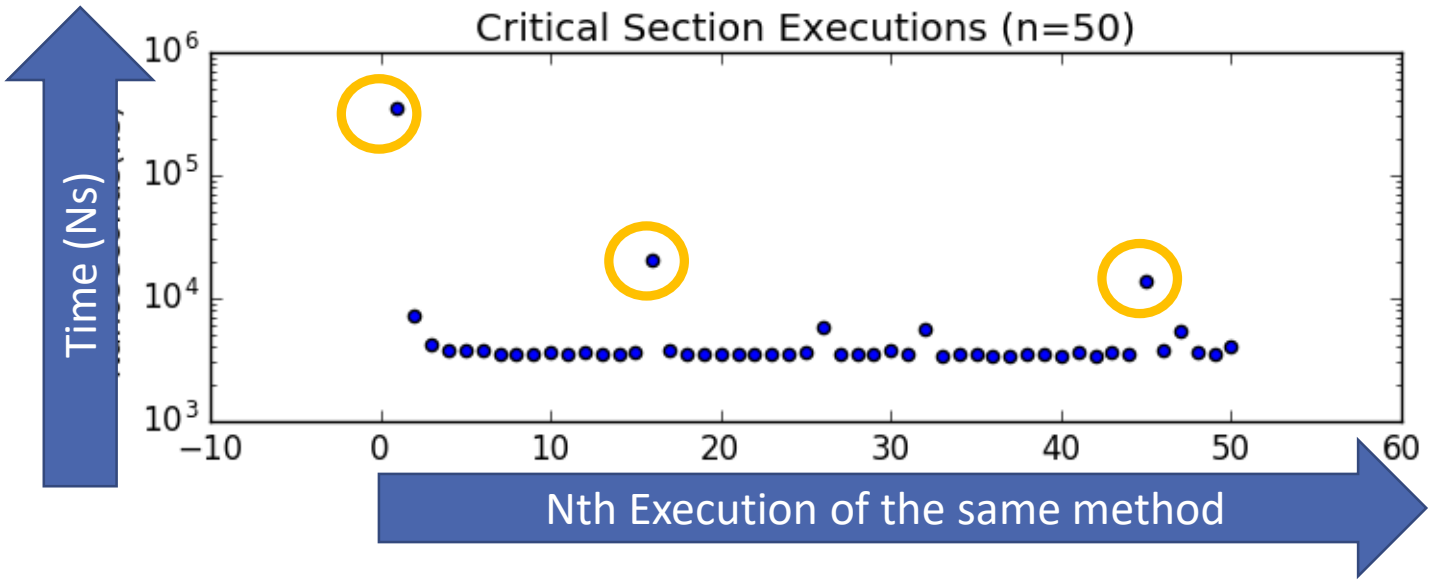
SortArrayListTest.addRandom5LetterWord



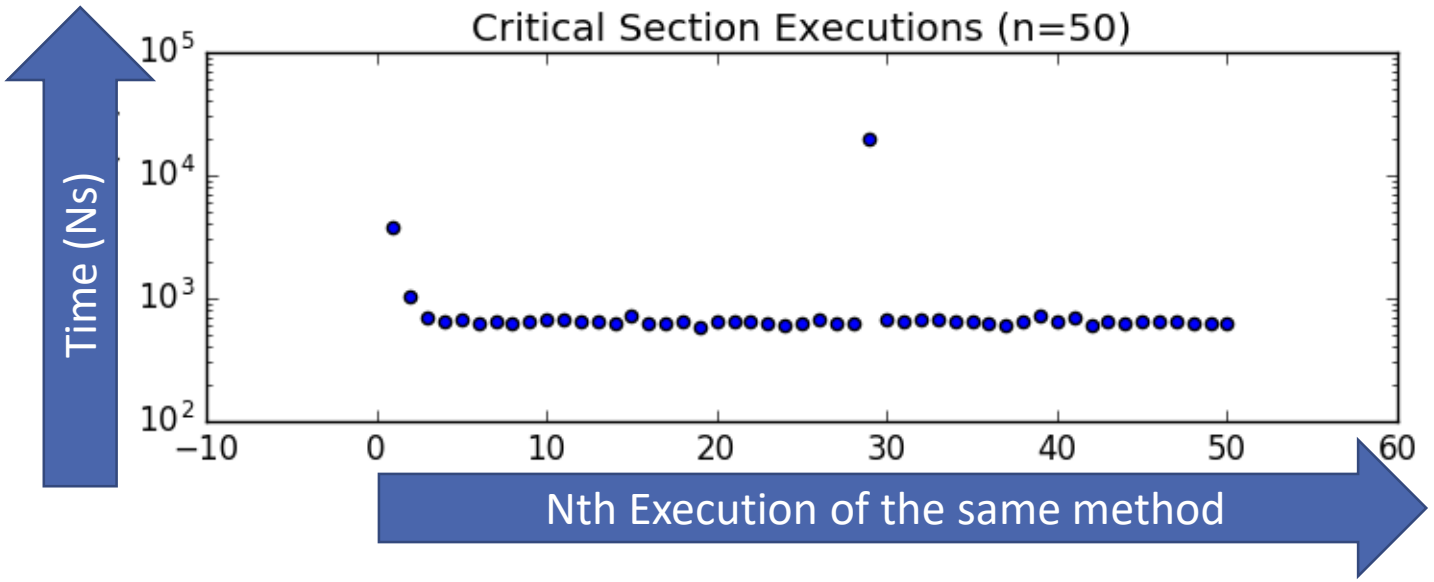
SortLinkedListTest.addRandom5LetterWord



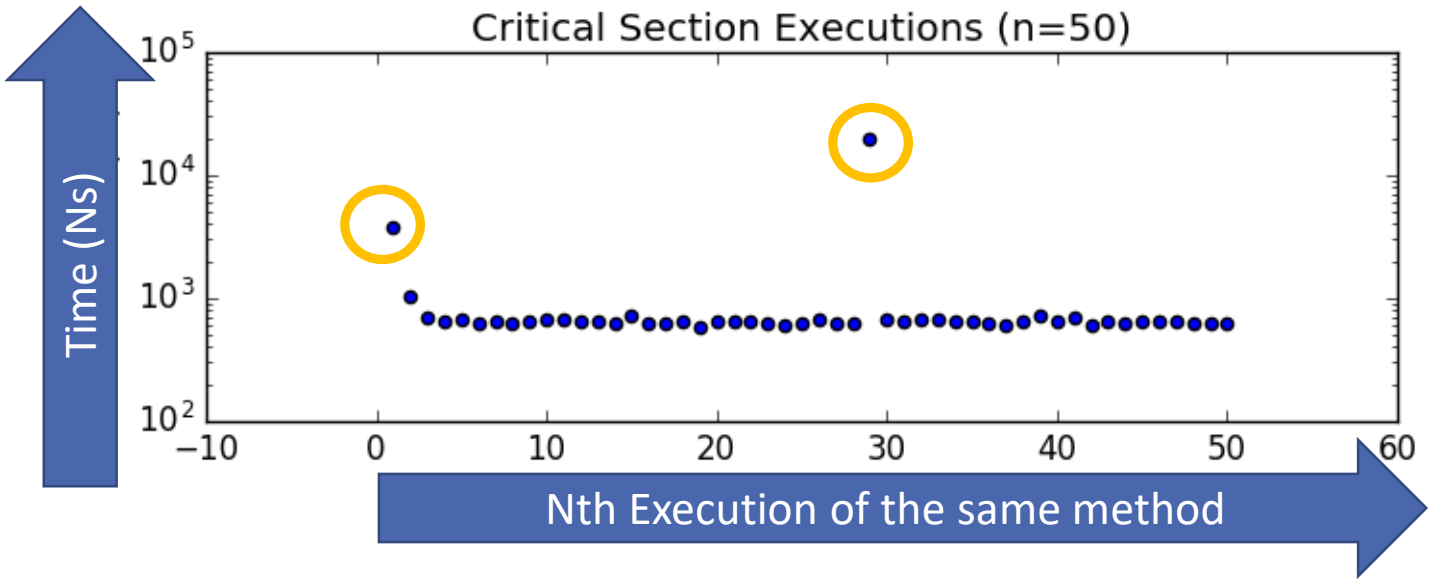
SortLinkedListTest.addRandom5LetterWord



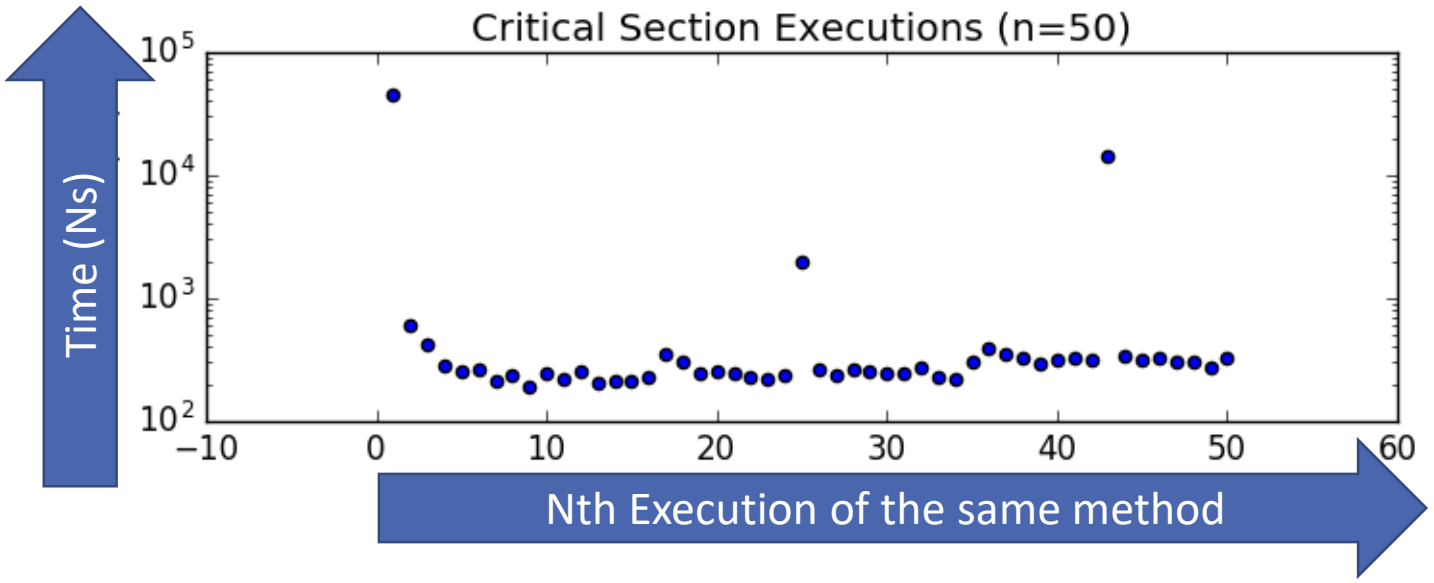
StackTest.remove()



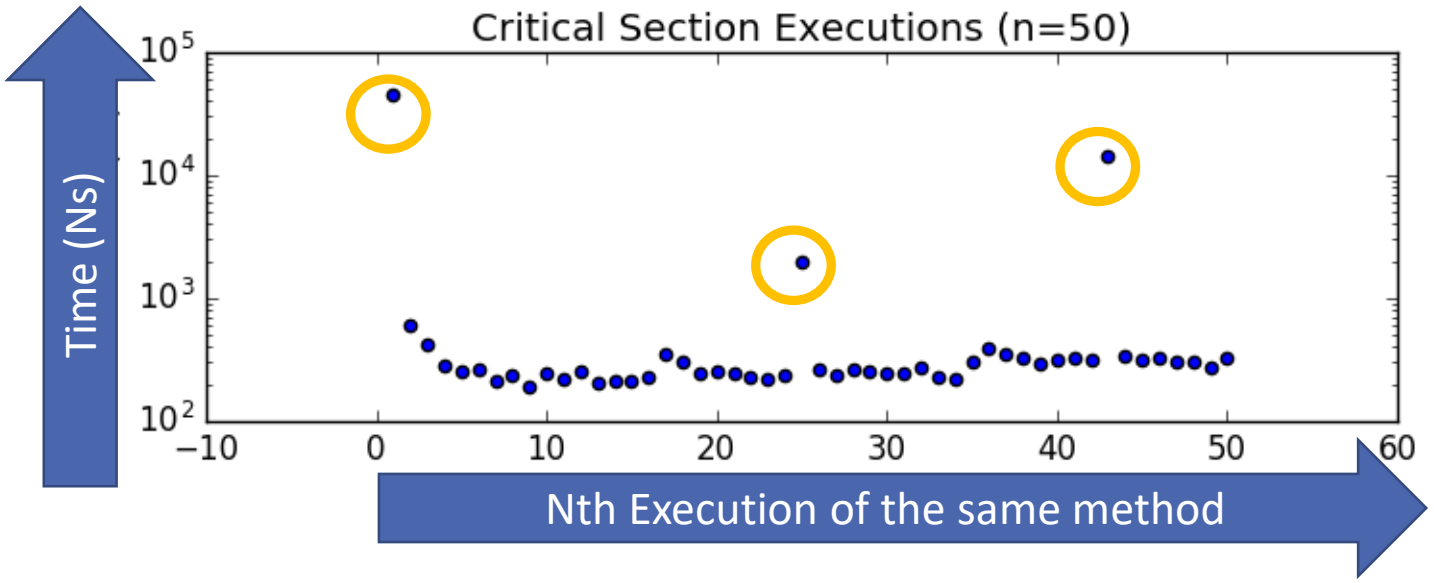
StackTest.remove()



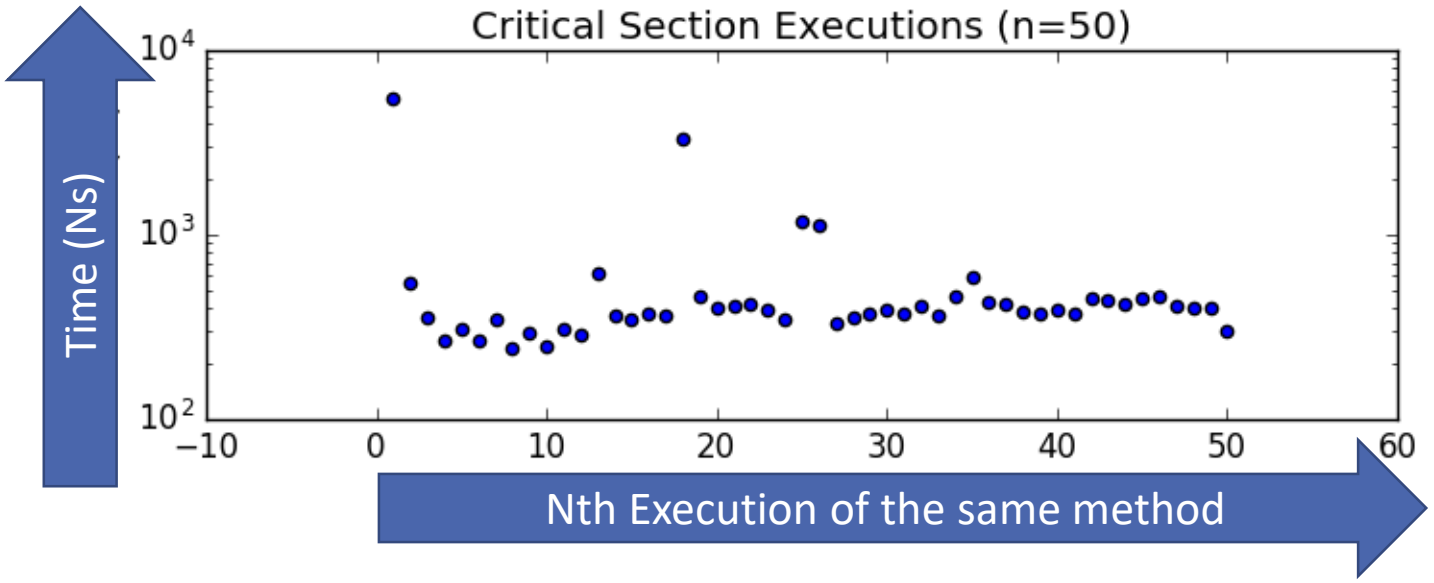
StringTest.append()



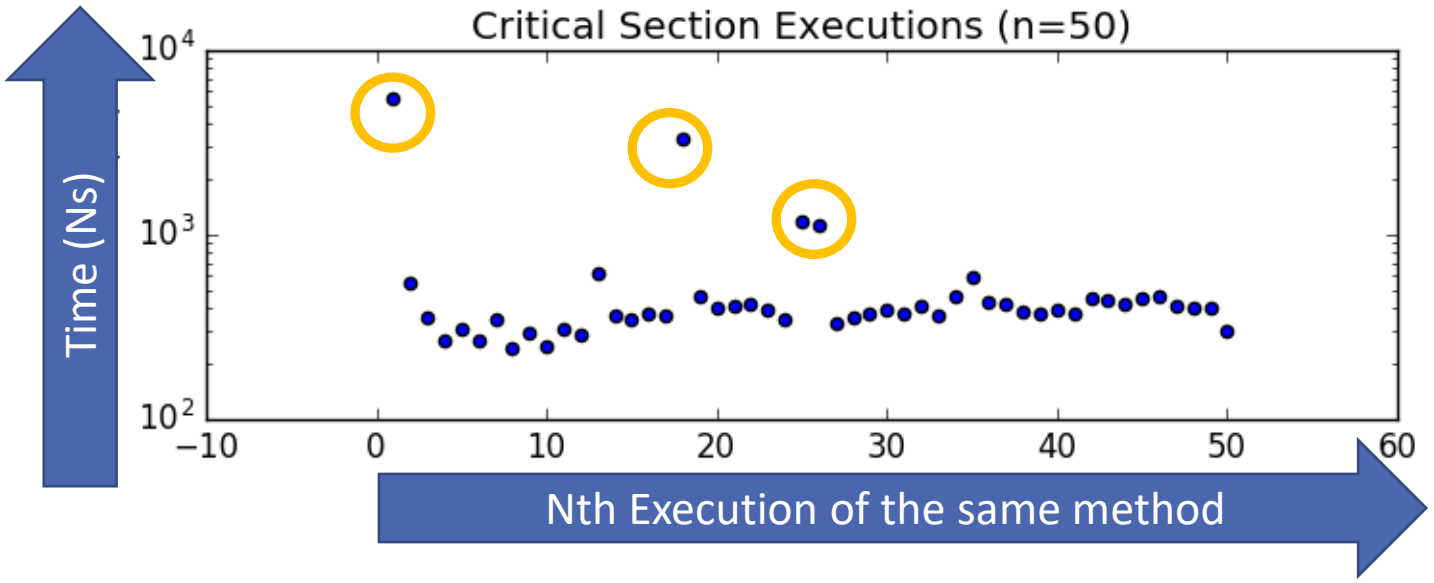
StringTest.append()

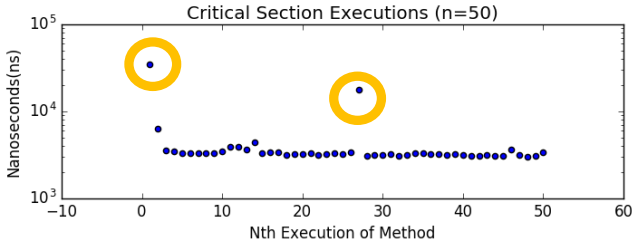


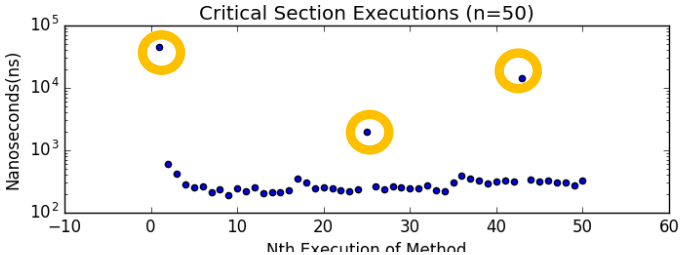
AllocationTest.remove()

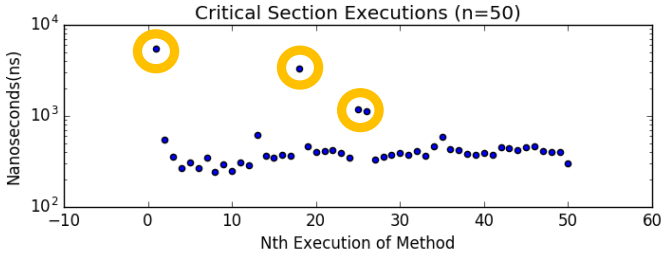


AllocationTest.remove()





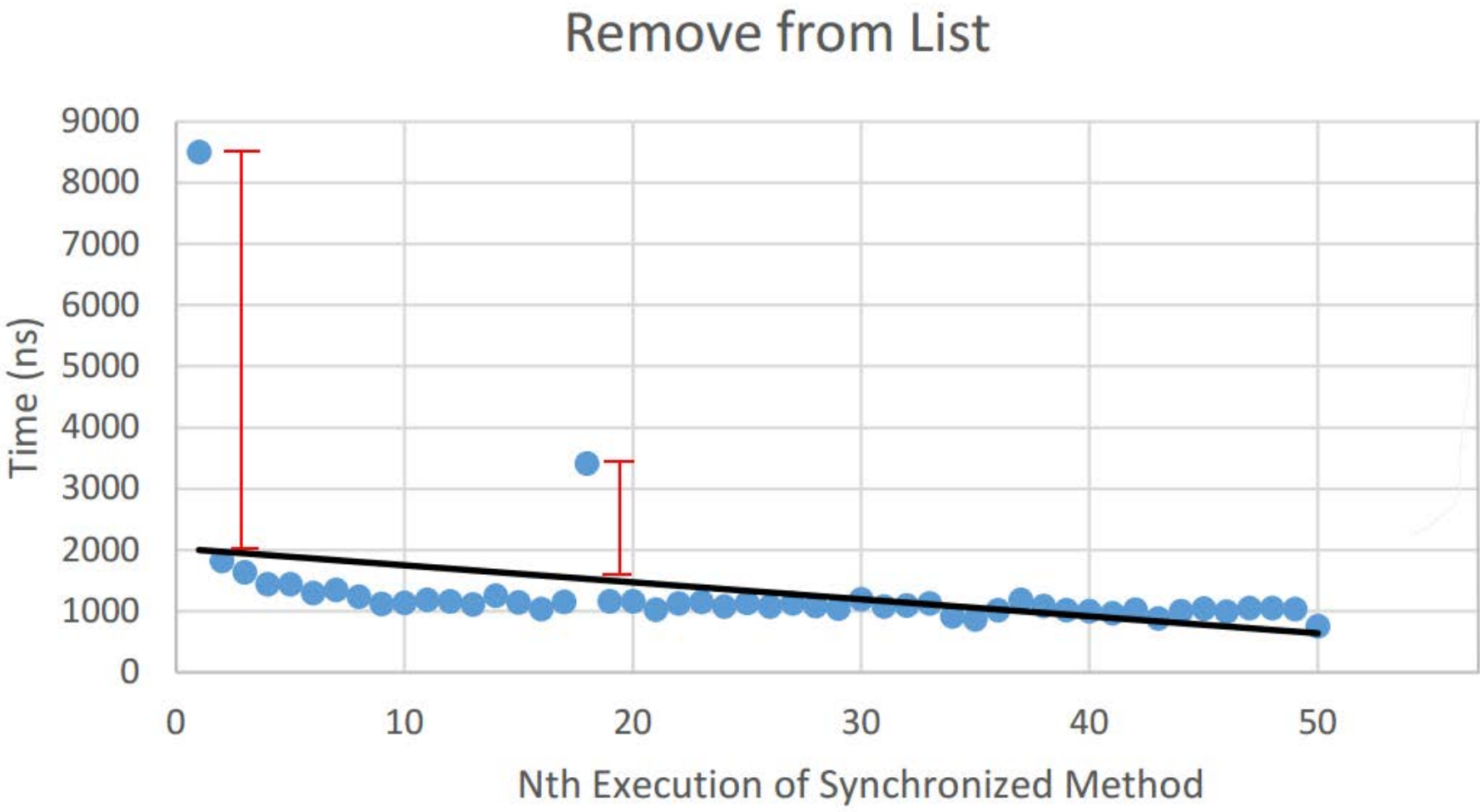




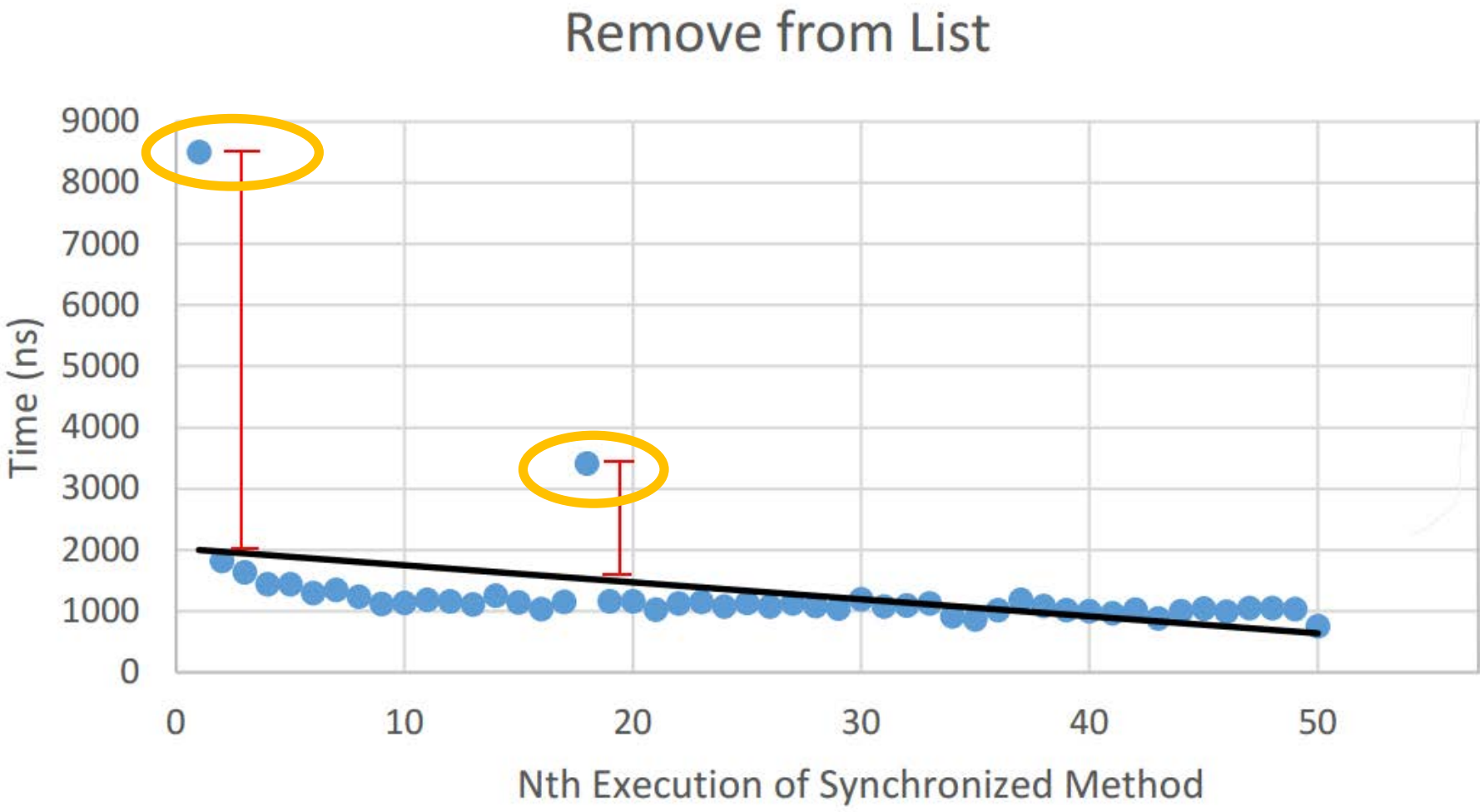
Our tool captures execution information

Variability in execution time exists—these plots are not flat

Statistical model – Find executions greater than one standard deviation away from trend



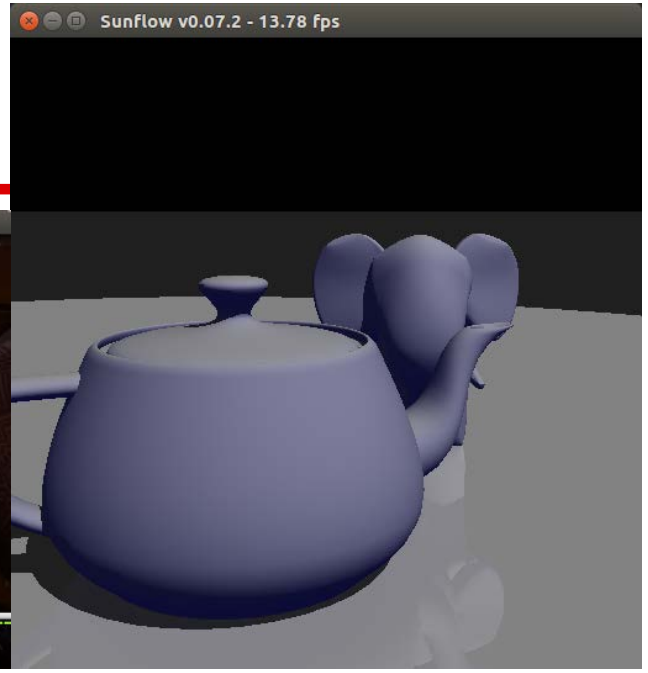
Statistical model – Find executions greater than one standard deviation away from trend



Test on Real World Programs

Skulls – Video Game

Mediaplayer – Hi-resolution .mp4 video player



Sunflow - Raytracer

Metrics collected for each synchronized method

- Number of times method executes

Metrics collected for each synchronized method

- Number of times method executes
- Total time spent in that method
  - Min, max, average ranges

Java Movie Player

- Analyzed 40 different **synchronized** methods
- Iceberg tells us which methods diverged during most of their executions
- Iceberg tells us how much total time we spend within synchronized method
  - So we know if it is worth our time to fix.

Java Movie Player – Example 1 | getRenderer

- A method that returns a renderer to draw to the screen

Java Movie Player – Example 1 | getRenderer

- A method that returns a renderer to draw to the screen
- A big critical section
  - Iteration
  - Control flow changes based on state

```java
public static synchronized Renderer getRenderer(FilterContext fctx) {
    if (fctx == null) {
        throw new IllegalArgumentException("FilterContext must be non-null");
    }

    Renderer r = rendererMap.get(fctx);
    if (r != null) {
        if (r.getRendererState() == RendererState.OK) {
            return r;
        }
        if (r.getRendererState() == RendererState.LOST) {
            // use the backup while the renderer is in lost state, until
            // it is disposed (or forever if it can't be disposed/reset)
            // Note: we don't add it to the cache to prevent permanent
            // association of the backup renderer and this filter context.
            return r.getBackupRenderer();
        }
        if (r.getRendererState() == RendererState.DISPOSED) {
            r = null;
            // we remove disposed renderers below instead of here to cover
            // cases where we never use a context which the disposed
            // renderer is associated with
        }
    }

    if (r == null) {
        // clean up all disposed renderers first
        Collection<Renderer> renderers = rendererMap.values();
        for (Iterator<Renderer> iter = renderers.iterator(); iter.hasNext();)
        {
            Renderer ren = iter.next();
            if (ren.getRendererState() == RendererState.DISPOSED) {
                ren.imagePool.dispose();
                iter.remove();
            }
        }

        r = RendererFactory.createRenderer(fctx);
        if (r == null) {
            throw new RuntimeException("Error creating a Renderer");
        } else {
            if (verbose) {
                String klassName = r.getClass().getName();
                String rname = klassName.substring(klassName.lastIndexOf(".")+1);
                Object screen = fctx.getReferent();
                System.out.println("Created " + rname +
                    " (AccelType=" + r.getAccelType() +
                    ") for " + screen);
            }
        }
        rendererMap.put(fctx, r);
    }
    return r;
}
```

(source code tiny on purpose)

Java Movie Player – Example 1 | getRenderer

- A method that returns a renderer to draw to the screen
- A big critical section
  - Iteration
  - Control flow changes based on state
- Called 2104 times during a 14 minute movie

```java
public static synchronized Renderer getRenderer(FilterContext fctx) {
    if (fctx == null) {
        throw new IllegalArgumentException("FilterContext must be non-null");
    }

    Renderer r = rendererMap.get(fctx);
    if (r != null) {
        if (r.getRendererState() == RendererState.OK) {
            return r;
        }
        if (r.getRendererState() == RendererState.LOST) {
            // use the backup while the renderer is in lost state, until
            // it is disposed (or forever if it can't be disposed/reset)
            // Note: we don't add it to the cache to prevent permanent
            // association of the backup renderer and this filter context.
            return r.getBackupRenderer();
        }
        if (r.getRendererState() == RendererState.DISPOSED) {
            r = null;
            // we remove disposed renderers below instead of here to cover
            // cases where we never use a context which the disposed
            // renderer is associated with
        }
    }

    if (r == null) {
        // clean up all disposed renderers first
        Collection<Renderer> renderers = rendererMap.values();
        for (Iterator<Renderer> iter = renderers.iterator(); iter.hasNext();)
        {
            Renderer ren = iter.next();
            if (ren.getRendererState() == RendererState.DISPOSED) {
                ren.imagePool.dispose();
                iter.remove();
            }
        }

        r = RendererFactory.createRenderer(fctx);
        if (r == null) {
            throw new RuntimeException("Error creating a Renderer");
        } else {
            if (verbose) {
                String klassName = r.getClass().getName();
                String rname = klassName.substring(klassName.lastIndexOf(".")+1);
                Object screen = fctx.getReferent();
                System.out.println("Created " + rname +
                    " (AccelType=" + r.getAccelType() +
                    ") for " + screen);
            }
        }
        rendererMap.put(fctx, r);
    }
    return r;
}
```

(source code tiny on purpose)

Java Movie Player – Example 1 | getRenderer

- A method that returns a renderer to draw to the screen
- A big critical section
  - Iteration
  - Control flow changes based on state
- Called 2104 times during a 14 minute movie
- Diverges 0.03% of the time

```java
public static synchronized Renderer  getRenderer(FilterContext fctx) {
    if (fctx == null) {
        throw new IllegalArgumentException("FilterContext must be non-null");
    }

    Renderer r = rendererMap.get(fctx);
    if (r != null) {
        if (r.getRendererState() == RendererState.OK) {
            return r;
        }
        if (r.getRendererState() == RendererState.LOST) {
            // use the backup while the renderer is in lost state, until
            // it is disposed (or forever if it can't be disposed/reset)
            // Note: we don't add it to the cache to prevent permanent
            // association of the backup renderer and this filter context.
            return r.getBackupRenderer();
        }
        if (r.getRendererState() == RendererState.DISPOSED) {
            r = null;
            // we remove disposed renderers below instead of here to cover
            // cases where we never use a context which the disposed
            // renderer is associated with
        }
    }

    if (r == null) {
        // clean up all disposed renderers first
        Collection<Renderer> renderers = rendererMap.values();
        for (Iterator<Renderer> iter = renderers.iterator(); iter.hasNext();)
        {
            Renderer ren = iter.next();
            if (ren.getRendererState() == RendererState.DISPOSED) {
                ren.imagePool.dispose();
                iter.remove();
            }
        }

        r = RendererFactory.createRenderer(fctx);
        if (r == null) {
            throw new RuntimeException("Error creating a Renderer");
        } else {
            if (verbose) {
                String klassName = r.getClass().getName();
                String rname = klassName.substring(klassName.lastIndexOf(".")+1);
                Object screen = fctx.getReferent();
                System.out.println("Created " + rname +
                    " (AccelType=" + r.getAccelType() +
                    ") for " + screen);
            }
        }
        rendererMap.put(fctx, r);
    }
    return r;
}
```

(source code tiny on purpose)

Resulting Analysis

- 40 synchronized methods analyzed in Movie Player
- 86 synchronized methods analyzed in Skulls
- 44 synchronized methods analyzed in Sunflow

- 10 additional benchmarks in the database, graphics, and game domains analyzed.

Resulting Analysis - 8 different behaviors

- Behavior A: Several execution paths
- Behavior B: high contention
- Behavior C: amortized data structures
- Behavior D: data locality (linked list versus array)
- Behavior E: large number of allocations
- Behavior F: Input/Output events, such as logging
- Behavior G: Generic getter/setter methods
- Behavior H: Critical sections that only execute once

Resulting Analysis

- Our tool picks out synchronized methods with the highest variability

- Resolutions
- Split the function up into special use cases (smaller critical sections)
- Variability important, makes difficult for profile-guided optimizations

Reminder

Static, dynamic, and software visualization analysis tools focused on critical sections are needed to **uncover performance variability** in critical sections to avoid unintended software hangs

We have confirmed performance variability!

Now to focus on root cause

Acknowledgements

The committee for their patience, feedback, guidance, and support!

Understanding and Tuning the Performance of Critical Sections with Program Analysis and Software Visualization Tools

Michael Dilip Shah

Advisor: Samuel Z. Guyer

Monday July 31, 2017

Committee Members

- Remco Chang
- Kathleen Fisher
- Mark Hempstead
- Tao B. Schardl