# Tricks and Tips With NIO Using the Grizzly Framework

**Jeanfrancois Arcand, Senior Staff Engineer**
**Charlie Hunt, Senior Staff Engineer**
**Scott Oaks, Senior Staff Engineer**

SUN Microsystems

TS-2992

# Goal of Your Talk

## Tricks and Tip with NIO

In this session, we discuss tricks and tips we have learned working on the Project Grizzly NIO Framework. We will describe how to efficiently manage ByteBuffers, how to properly handle SelectionKeys, recommend threading and buffer management techniques, and talk about an efficient NIO/SSL implementation.

# Agenda

Introduction
What Is Project Grizzly
Project Grizzly Performance
Overview of NIO
Tricks and Tips
Summary

# Agenda

**Introduction**

What Is Project Grizzly

Project Grizzly Performance

Overview of NIO

Tricks and Tips

Summary

# Introduction

- Implementing scalable servers in Java™ platform can be difficult
- Almost impossible prior to Java technology-based NIO
- Using Java technology-based NIO effectively can be a challenge
- Project Grizzly eliminates many of these challenges
- In this session we present many of the tips and tricks we learned while implementing Project Grizzly to build scalable servers using Java technology-based NIO

# Introduction

## What we will talk about in this session

- Give a brief history of Project Grizzly
  - How Project Grizzly was born
  - Current state of Project Grizzly
- Show Project Grizzly's performance and scalability
- Familiarize ourselves with Java technology-based NIO package
- Present several of the tips-and-tricks effectively using Java technology-based NIO as we learned implementing Project Grizzly

# Agenda

Introduction

**What Is Project Grizzly**

Project Grizzly Performance

Overview of NIO

Tricks and Tips

Summary

# What Is Project Grizzly

## Project Grizzly (current Grizzly Framework)

- Uses Java technology-based NIO primitives and hides the complexity of programming with Java technology-based NIO
- Easy-to-use high-performance APIs for TCP, UDP and SSL communications
- Brings non-blocking sockets to the protocol processing layer
- Utilizes high-performance buffers and buffer management
- Choice of several different high-performance thread pools

# What Is Project Grizzly

## History of Project Grizzly

- Project Grizzly was born in 2004 under the GlassFish™ project, (https://glassfish.dev.java.net)
- Initially built as an HTTP Web Server, replacing Tomcat's Coyote Connector and Sun WebServer 6
  - Later became known as Grizzly 1.0.
- Grizzly 1.0 shipped with Sun Java System Application Server 8.1 PE, 8.2 PE/EE and all GlassFish project distributions, replacing native Sun WebServer software runtime

# What Is Project Grizzly

## History of Project Grizzly

- Grizzly 1.0 became extremely popular in 2006; Multiple protocol implementations were built on top of it
- But Grizzly 1.0 had HTTP protocol specific implementation details included in its transport logic
- The main class, SelectorThread, contained several artifacts specific to http such as file caching, request monitoring, etc.

# What Is Project Grizzly

## History of Project Grizzly

- Several classes needed to be extended in order to use the framework
- Example: JettySelectorThread extends SelectorThread
- Example: SSLSelectorThread extends SelectorThread
- The Grizzly 1.0 mixed ‘extension’ and ‘implementation’

# What Is Project Grizzly

## History of Project Grizzly

- But, Grizzly 1.0 was still a good choice for nearly all TCP/HTTP-based protocols
- Projects successfully utilizing Grizzly 1.0:
  - JRuby On Grizzly
  - Project Alaska's HTTP BC component
  - GlassFish build v3 micro kernel
  - Phobos in NetBeans™ software
  - SOAP over TCP integration in GlassFish project
  - Comet / Cometd
  - AsyncWeb on Grizzly
  - GlassFish build v2
  - Sun Web 2.0 Developer pack (REST HTTP Server)

# What Is Project Grizzly

## Open Source Grizzly

- Grizzly 1.5 began development in 2006
- Grizzly 1.5 objectives
  - Remove all dependencies on HTTP and/or GlassFish project
  - All 1.0 applications must still work with 1.5
  - Support all tricks and tips learned during development of Grizzly 1.0 (performance, NIO performance gotchas, etc.)
  - **Keep it simple!!**
- Grizzly 1.5 Open Sourced February 6, 2007, https://grizzly.dev.java.net
- Grizzly 1.5 officially released as of 2007 JavaOne$^{SM}$ Conference!!

# What Is Project Grizzly

## Who's looking at Grizzly 1.5 ?

- Sun JDK™ Software ORB
- GlassFish Project ORB
- Sun Java System Message Queue Software
- Sun Labs Project(s)
- Several enterprise/middleware companies
- …and many more (subscribe to Project Grizzly's mailing list to learn who!!!)

# What Is Project Grizzly

## Where to find Project Grizzly

- Open Source Project on java.net, (https://grizzly.dev.java.net)
- Open Sourced under CDDL license
- Very open community policy
  - All project communications are done on Project Grizzly mailing list; No internal, off mailing list conversations
  - Project meetings open to anyone, (public conference call)
- Project decisions are made by project member vote
  - No project member has more voting power than any other project member

# Agenda

Introduction

What Is Project Grizzly

**Project Grizzly Performance**

Overview of NIO

Tricks and Tips

Summary

# Project Grizzly Performance

## How did we performance test?

- Client load generated via faban
  - faban.sunsource.net
- Two modes of measurement
  - Throughput
    - Limited # of clients
    - No Think Time
  - Scalability
    - Max # of clients with 90% response time metric
    - Think time

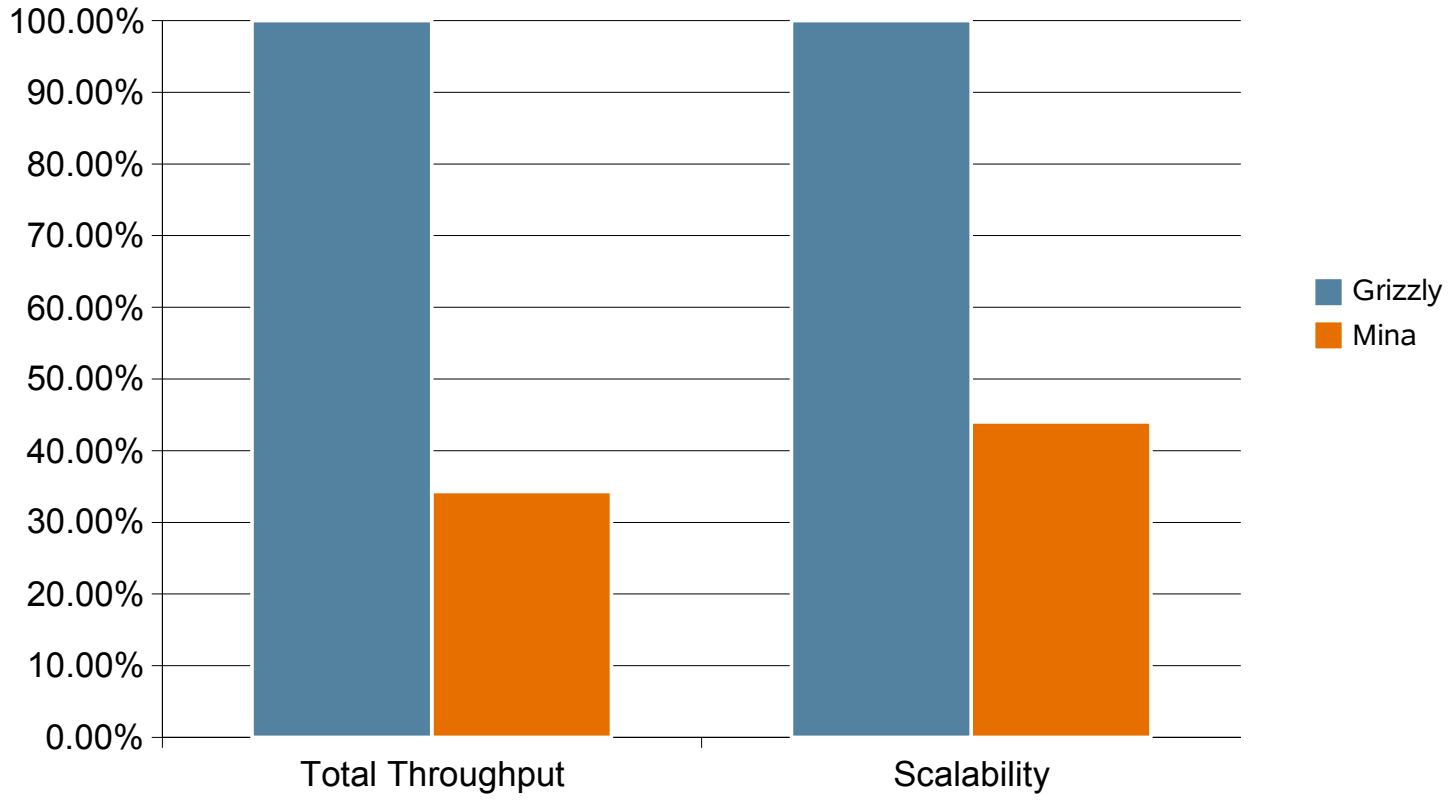
# Project Grizzly Performance

## Comparing Project Grizzly to Apache MINA both running AsyncWeb

- What is MINA?
  - Apache MINA (Multipurpose Infrastructure for Network Applications) is a network application framework which helps users develop high performance and high scalability network applications easily
- What is AsyncWeb?
  - AsyncWeb is a high-throughput, non-blocking Java platform HTTP engine—Designed throughout to support asynchronous request processing; AsyncWeb is built on top of MINA

# Project Grizzly vs. Apache MINA Both Running AsyncWeb

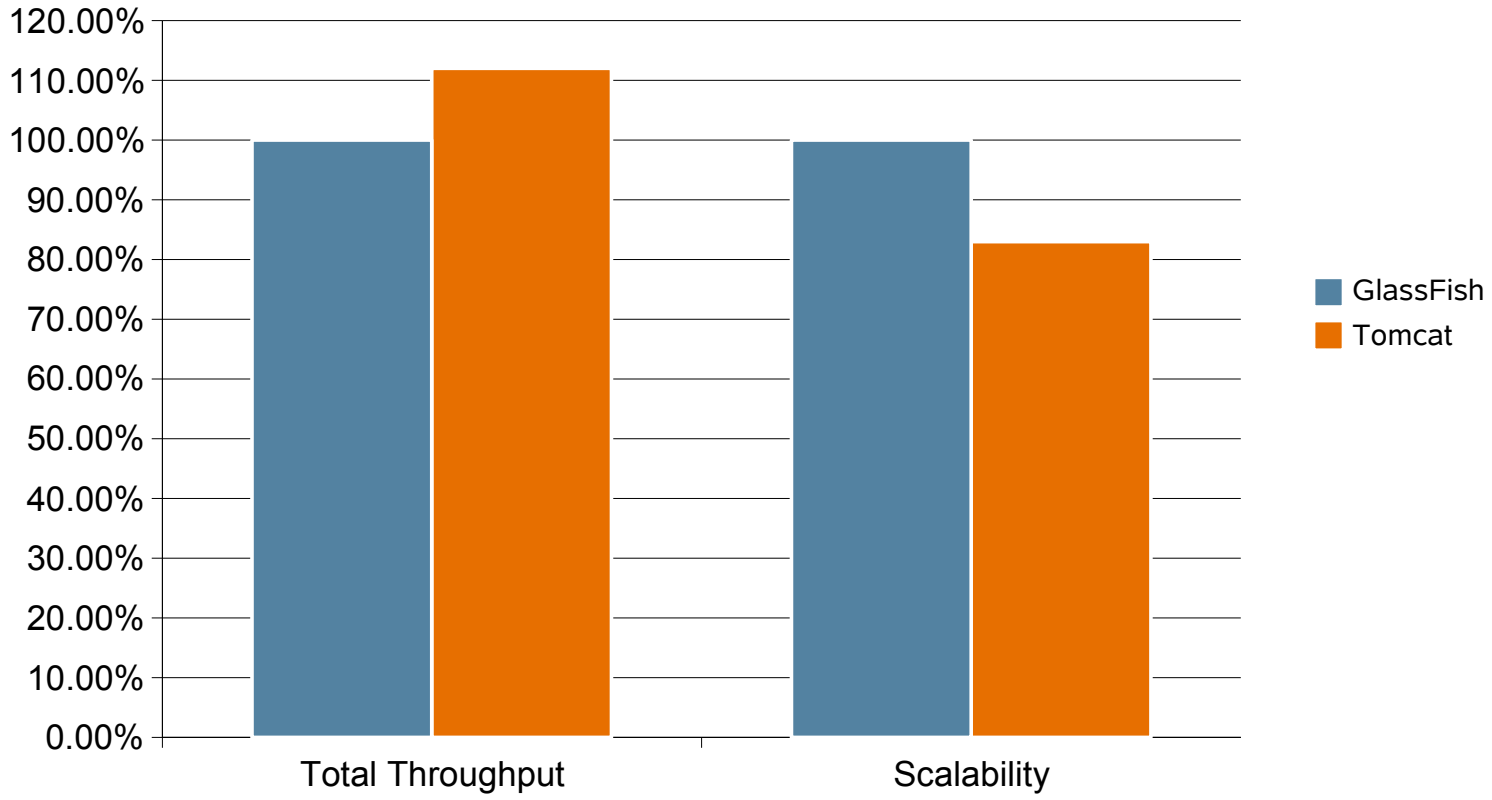
## Higher is better, normalized to Grizzly score




Source: Internal Benchmark Tests

# GlassFish vs. Tomcat

## Higher is better, normalized to Grizzly score




Source: Internal Benchmark Tests

# Project Grizzly Performance

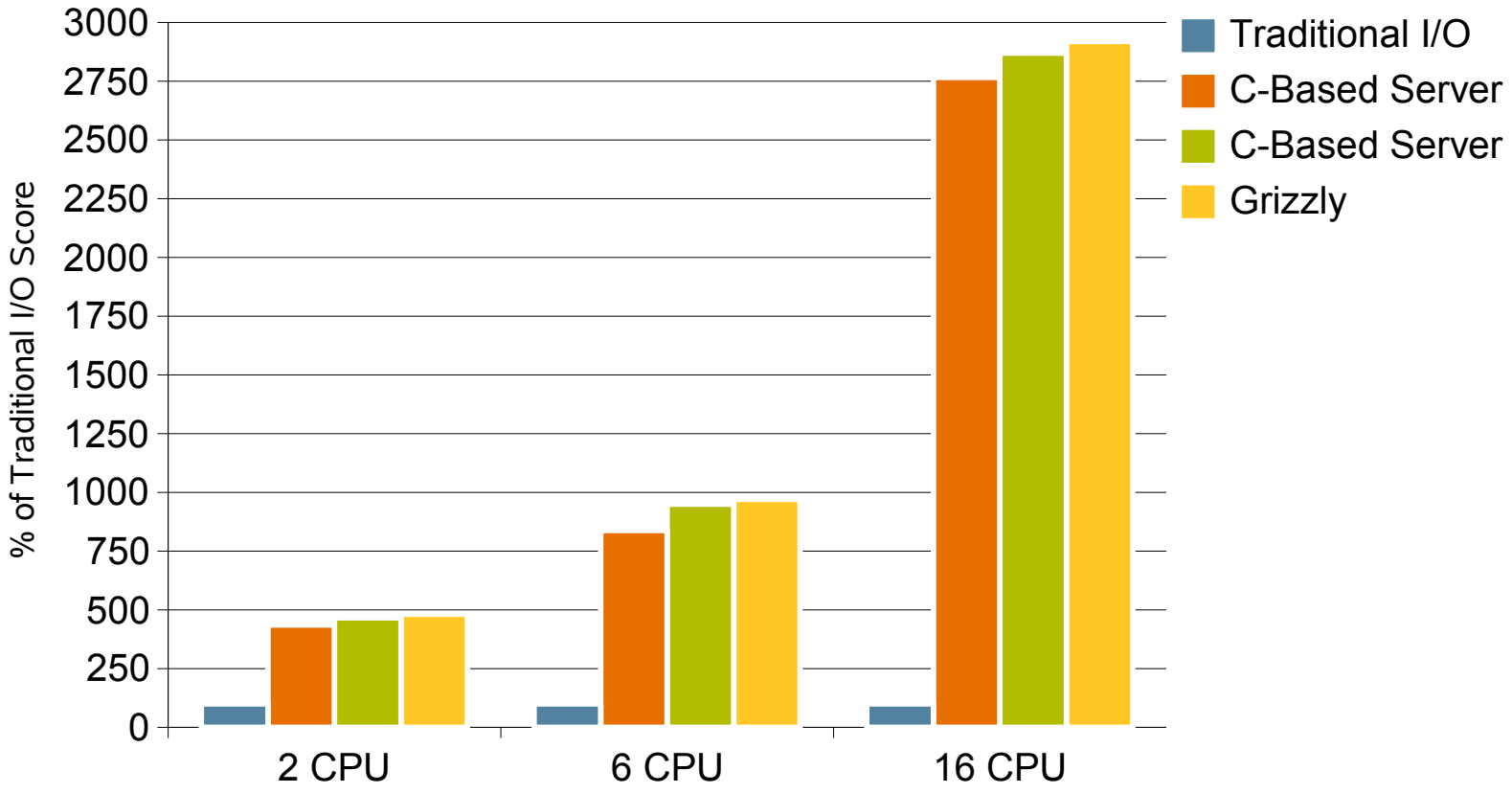
## Project Grizzly HTTP performance

Tested against a benchmark designed to:

- Measure scalability, specifically to measure how many concurrent clients can be supported with:
  - Average client think time of 8 seconds
  - 90% response time within 3 seconds
  - Error rate < 0.1%

# Project Grizzly HTTP vs. Other HTTP Servers

## Higher is better



# Agenda

Introduction

What Is Project Grizzly

Project Grizzly Performance

**Overview of NIO**

Tricks and Tips

Summary

# Overview of NIO

## Blocking or non-blocking?

- Traditional java.net.Sockets are blocking—They block, wait/sleep for data
- NIO Channels can be blocking or non-blocking
  - Non-blocking Channels never put the invoking thread to sleep; Operations either complete right away or return a result indicating nothing was done
  - Non-blocking makes it easy to manage many channels simultaneously
  - Blocking channels encourages “one thread per connection” paradigm
- NIO Channels must be configured, blocking or non-blocking

# Overview of NIO

## Selector—Basic abstraction to enable multiplexed I/O

- Register one or more ‘selectable’ channels (i/o)
- Relationship between channel and selector represented by a selection key (SelectionKey)
- Selection key remembers the events you are interested in
- Selector’s select() method updates the keys which are “ready”
- Service each channel by iterating over the keys which are “ready”

# Overview of NIO

## Buffer and ByteBuffer

- Containers for handling data
- Work very well together with Channels
- High performing if done right!
- To use Buffers, you must understand
  - Capacity—Max number of elements
  - Limit—Count of “live” elements, don’t read/write beyond
  - Position—Index of next element to read/write
  - Mark—A remembered position
  - 0 ← mark ← position ← limit ← capacity

# Overview of NIO

## Two Flavors of ByteBuffers

- ByteBuffers—two flavors
- Direct ByteBuffer and non-direct ByteBuffer
- Non-direct ByteBuffer
  - Underlying storage maintained in a Java platform byte[]
  - Implemented in JDK™ software for Java HotSpot™ technology class libraries as HeapByteBuffer
- Direct ByteBuffer
  - Underlying storage maintained in native code, not in the Java platform heap
  - Implemented in JDK software for Java HotSpot technology class libraries as DirectByteBuffer

# Overview of NIO

## NIO Myths

- Using NIO SocketChannels and ByteBuffers is easy

# Overview of NIO

## NIO Myths

- Using NIO SocketChannels and ByteBuffers is easy
  - TRUE

# Overview of NIO

## NIO Myths

- Using NIO SocketChannels and ByteBuffers is easy
  - TRUE
- Building a high performing and highly scalable application with NIO is easy

# Overview of NIO

## NIO Myths

- Using NIO SocketChannels and ByteBuffers is easy
  - TRUE
- Building a high performing and highly scalable application with NIO is easy
  - FALSE

# Overview of NIO

## NIO Myths

- Using NIO SocketChannels and ByteBuffers is easy
  - TRUE
- Building a high performing and highly scalable application with NIO is easy
  - FALSE
- Non-blocking NIO is for server-side applications only

# Overview of NIO

## NIO Myths

- Using NIO SocketChannels and ByteBuffers is easy
  - TRUE
- Building a high performing and highly scalable application with NIO is easy
  - FALSE
- Non-blocking NIO is for server-side applications only
  - FALSE

# Agenda

Introduction
What Is Project Grizzly
Project Grizzly Performance
Overview of NIO
**Tricks and Tips**
Summary

# Tip #1: Use of SelectionKey.attach()

## Use care when attaching an object

- SelectionKey(s) are a token representing the registration of a SelectableChannel (i.e. SocketChannel) with a Selector
- A SelectionKey is created each time a SocketChannel is registered with a Selector and remains valid until it is cancelled
- A SelectionKey has “interest ops” which indicate when a SocketChannel is “ready” for type of operation operation, (i.e. read, write, accept, connect)

# Tip #1: Use of SelectionKey.attach()

## Use care when attaching an object

- The SelectionKey API also provides the ability to attach an object using *SelectionKey.attach*(Object obj), and later retrieve that object using Object *SelectionKey.attachment*()
- This is very convenient construct to use on a SelectionKey to pass or hold state information between “interest op” operations
- But, use care when using this construct

# Tip #1: Use of SelectionKey.attach()

## Use care when attaching an object

- What you are attaching to the SelectionKey could lead to unintended consequences
- Why? Because there is some probability that your SelectionKey might never return to a ready-operation state, leaving the SelectionKey and its attachment forever inside the Selector key set
- But what's the point, nobody will ever do that! We are all very talented engineers, and we always take special care in managing our SelectionKey(s), right?

# Tip #1: Use of SelectionKey.attach()

## Use care when attaching an object

- In a software architecture where your (NIO) framework needs to handle thousands of connections, and you need to keep-alive those connections for a very long time (from 60 seconds to 5 minutes)
- Most frameworks (and unfortunately a lot of tutorials, articles and presentations) attach their framework object to the SelectionKey (Reactor pattern)

# Tip #1: Use of SelectionKey.attach()

## Can lead to a memory leak

- Those framework objects will likely include one or more of the following:
  - A ByteBuffer
  - Some keep-alive object (let's assume a Long)
  - A SocketChannel
  - A Framework Handler (like the Reactor pattern)
  - etc.

# Tip #1: Use of SelectionKey.attach()

## Can lead to a memory leak

- So you can end up with thousand of objects taking vacations, enjoying idle time inside the Selector key set
- If you did not implement any mechanism to take a periodic look inside the Selector key set, then you may end up with a memory leak (or your framework performance will be impacted)
- Worse, you might never notice the problem

# Tip #1: Use of SelectionKey.attach()

## Common misuse

- **How do I retrieve the SocketChannel if I don't attach it to my framework objects?**
- Most existing frameworks include inside their framework objects, a SocketChannel associated with the SelectionKey; This is not needed because a SocketChannel can always be retrieved using SelectionKey.channel()

# Tip #1: Use of SelectionKey.attach()

## Common misuse

- **How do I deal with incomplete SocketChannel read?**
- When you do non-blocking SocketChannel.read(), you can never predict when all bytes are read from the socket buffer
- Common practice is to register the SelectionKey back with the Selector and attach the incomplete ByteBuffer to the SelectionKey
- Instead, I would recommend you use a temporary Selector, (next trick)

# Tip #1: Use of SelectionKey.attach()

## Recommendations

- If you are building a framework, try to avoid attaching anything to the SelectionKey.attach()
- Allow the user of the framework to use SelectionKey.attach(). But, be sure to educate your users about potential pitfalls
  - Possible memory leak(s)
  - Performance throughput and/or scalability degradation
- Tip #2 helps with one case of wanting to use SelectionKey.attach()

# Tip #2: Using Temporary Selectors

## Avoiding Thread context switching

- Using multiple Selectors instances can significantly improve performance
- Why? Remember that Selectors detect when channels are ready for some operation of interest; When Selectors detect these “ready” operations, the operations themselves are usually carried out or dispatched to some other thread which results in a thread context switch
- Thread context switching can be expensive

# Tip #2: Using Temporary Selectors

## Solution

- Most implementations or frameworks utilize a single Selector where SocketChannels register a SelectionKey; Let's call this the "main" Selector
- When you are expecting to read or write more bytes from a SocketChannel, instead of registering the SocketChannel's SelectionKey back with the "main" Selector, create a temporary Selctor, register the SelectionKey with it along with enabling your read or write interest op
- Then, read or write more bytes using the temporary Selector

# Tip #2: Using Temporary Selectors

```
int bytesRead = socketChannel.read(byteBuffer);
if (bytesRead == 0) {
    readSelector = SelectorFactory.getSelector();
    tmpKey = socketChannel.register
                        (readSelector,SelectionKey.OP_READ);
    tmpKey.interestOps(tmpKey.interestOps() |
                        SelectionKey.OP_READ);
    int code = readSelector.select(readTimeout);
    tmpKey.interestOps(tmpKey.interestOps() &
                        (~SelectionKey.OP_READ));
    if ( code == 0 ){
       return 0;
    }
    do {
        bytesRead = socketChannel.read(byteBuffer);
    while (bytesRead > 0 && byteBuffer.hasRemaining());
...
```

# Tip #3: Handling the OP_WRITE

## Avoiding 100% CPU consumption

- Handling OP_ACCEPT and OP_READ has been well documented in many NIO tutorials
- However, OP_WRITE is often times not described
- Not handling OP_WRITE correctly can greatly impact your server performance, and on Win32 it can severely impact performance as a result of 100% CPU

# Tip #3: Handling the OP_WRITE (Cont.)

**Many NIO frameworks or applications will write to a non-blocking SocketChannel by doing something very similar if not exactly like:**

```
while (byteBuffer.hasRemaining()) {
    int len = socketChannel.write(byteBuffer);
    if (len < 0){
       throw new EOFException();
    }
 }
```

# Tip #3: Handling the OP_WRITE (Cont.)

- Under a light load, this code may work just fine
- But under heavy load, doing socketChannel.write(..) might return 0 on many write() operations
- That means the socket's outgoing buffer is full, hence all future writes will return 0 until the remote client reads them
- Since SocketChannel.write is returning 0, the CPU(s) will be consumed, looping over and over
- Side note: Socket send buffer can be increased

# Tip #3: Handling the OP_WRITE (Cont.)

## Solution

- Register an OP_WRITE interest opt to a temporary Selector and wait for the temporary Selector to tell you when the SocketChannel is ready for a write operation
- Using a temporary Selector to complete the write operation can significantly improve performance under load, especially on Win32

# Tip #3: Handling the OP_WRITE (Cont.)

```
while (byteBuffer.hasRemaining()) {
     int len = socketChannel.write(byteBuffer);
     if (len == 0) {
         if (writeSelector == null){
            writeSelector = SelectorFactory.getSelector();
         }
         key = socketChannel.register
                 (writeSelector, key.OP_WRITE);

         if (writeSelector.select(30 * 1000) == 0) {
             throw new IOException("Client disconnected");
         }
     }
 }
```

# Tip #4: Choosing the Right ByteBuffer

## Can significantly improve performance

- There are three choices of ByteBuffer:
  - Direct ByteBuffer [ByteBuffer.allocateDirect()]: Given a direct ByteBuffer, the Virtual Machine for the Java platform (JVM™ machine) will make a best effort to perform native I/O operations directly upon it
  - Non-direct ByteBuffer [ByteBuffer.allocate()]: A non-direct ByteBuffer backed by a Java platform byte array
  - View ByteBuffer [ByteBuffer.slice()]: a ByteBuffer whose content is a shared subsequence of direct or non-direct ByteBuffer's content

The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java™ platform.

# Tip #4: Choosing the Right ByteBuffer

## Can significantly improve performance

- Choosing the right one can significantly improve performance
- In some cases, the view buffers from a non-direct ByteBuffer (HeapByteBuffer) can perform better than view buffers from a direct ByteBuffer (DirectByteBuffer)
- Highly recommend you performance test both; Performance may change from JDK release to JDK release and from workload to workload

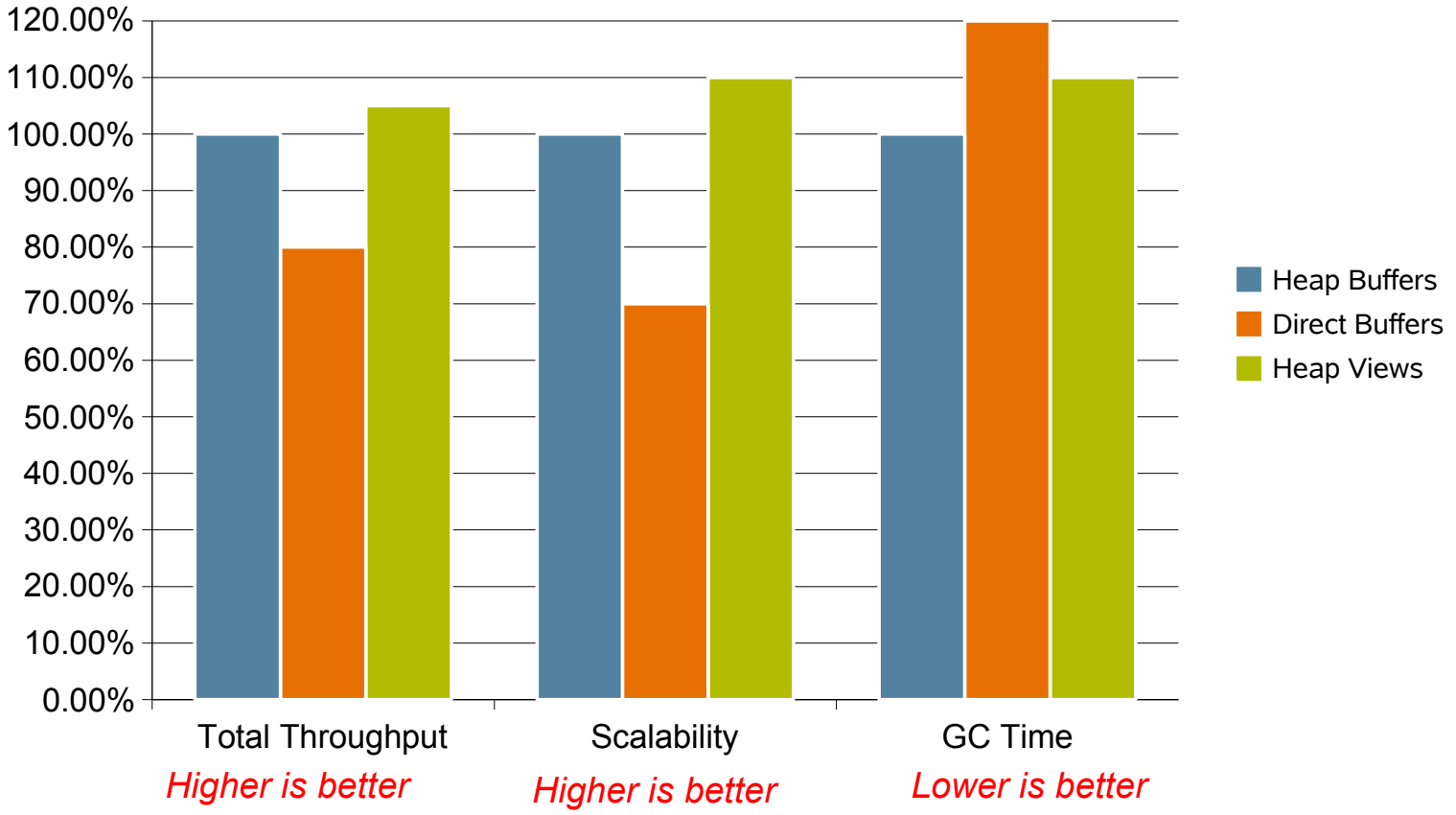
# Tip #4: Choosing the Right ByteBuffer

## Buffer performance

- Some of the workloads we tested to measure scalability and throughput showed some interesting differences in ByteBuffer performance, non-direct ByteBuffer, direct ByteBuffer and “view” buffers using non-direct ByteBuffer
- Some of these workloads tended do a large amount of copying of Java platform heap data, (i.e. Strings) from a byte array into buffers

# Tip #4: Choosing the Right ByteBuffer

## Normalized to non-direct ByteBuffer performance




Source: Internal Benchmarks

# Tip #4: Choosing the Right ByteBuffer

## Buffer performance

- The ‘real’ issue illustrated here is ‘copying bytes’ performance, not poor DirectByteBuffer performance
- Underscores the need to understand what is being tested in your workload(s)
- And, to continuously monitor Buffer performance and scalability with updated releases of the JDK software and JVM machine

# Tip #5: Implementing Non Blocking SSL

## Far from simple…

- The entry point when using SSL is the SSLEngine
- The SSLEngine is associated with the lifetime of the SocketChannel, so you need to take care of re-using the same SSLEngine between registration(s) of a SelectionKey
- For HTTP, it means you will most likely use SelectionKey.attach() for managing this association

# Tip #5: Implementing Non Blocking SSL

## Far from simple…

- That means we would attach something like the following to a SelectionKey:

```
public class Attachment {
        protected SSLEngine sslEngine;
        protected long keepAliveTime;
        protected ByteBuffer byteByffer;
  }
```

- You will likely pool the Attachment object to avoid creating a new instance every time you need to register a SelectionKey with a Selector

# Tip #5: Implementing Non Blocking SSL

## Far from simple...

- Having 10,000 connections will possibly mean 10,000 Attachment objects active, and leading to the potential issues described in Tip #1
- Fortunately, instead of creating Attachment objects, you can directly attach the SSLEngine to the SelectionKey and use the SSLSession object:

```
((SSLEngine)selectionKey.attachment()).getSession(
    ).putValue((EXPIRE_TIME,
    System.currentTimeMillis());
```

# Tip #5: Implementing Non Blocking SSL

## Far from simple…

- Notice that we are using SSLEngine.getSession() to store the data structure
- You don't have to synchronize on a pool and/or create your own data structure

# Tip #6: To Thread or not to Thread

## When to use Threads

- Any Selector event can be executed on a thread (OP_READ, OP_WRITE, OP_ACCEPT)
- When we started working on Project Grizzly, we designed the framework with the flexibility to easily add a thread pool mostly anywhere during request processing
- At the time when we started Project Grizzly, we made no recommendations on which events were better to delegate to another thread
- So, we decided to make it configurable!!

# Tip #6: To Thread or not to Thread

## When to use Threads

- First, let's restrict the discussion to OP_ACCEPT and OP_READ
- We have the following options:
  - A) Execute OP_ACCEPT and OP_READ on separate threads as the Selector thread
  - B) Execute OP_ACCEPT and OP_READ on a separate thread as the Selector thread
  - C) Execute OP_ACCEPT using the Selector thread and OP_READ on a separate thread
  - D) Execute OP_READ using the Selector thread and OP_ACCEPT on a separate thread

Source: Please add the source of your data here

# Tip #6: To Thread or not to Thread

## Option A

```
myExecutor = Executors.newFixedThreadPool(maxThreads);
try{
  selectorState = selector.select(selectorTimeout);
} catch (CancelledKeyException ex){
   ;
}
iterator = selector.selectedKeys().iterator();
while (iterator.hasNext()) {
   key = iterator.next();
   if (key.isAcceptable()) {
       myExecutor.execute(getAcceptHandler(key));
   } else if (key.isReadable()) {
       myExecutor.execute(getReadHandler(key));
```

# Tip #6: To Thread or not to Thread

## Option B

```
myExecutor = Executors.newFixedThreadPool(maxThreads);
[...]
iterator = selector.selectedKeys().iterator();
while (iterator.hasNext()) {
   key = iterator.next();
   myExecutor.execute(new Runnable(){
     public void run()
      if (key.isAcceptable()) {
        getAcceptHandler(key);
      } else if (key.isReadable()) {
        getReadHandler(key);
   });
```

# Tip #6: To Thread or not to Thread

## Option C

```
myExecutor = Executors.newFixedThreadPool(maxThreads);
try{
  selectorState = selector.select(selectorTimeout);
} catch (CancelledKeyException ex){
   ;
}
iterator = selector.selectedKeys().iterator();
while (iterator.hasNext()) {
   key = iterator.next();
   if (key.isAcceptable()) {
       getAcceptHandler(key).run();
   } else if (key.isReadable()) {
       myExecutor.execute(getReadHandler(key));
```

# Tip #6: To Thread or not to Thread

## Option D

```
myExecutor = Executors.newFixedThreadPool(maxThreads);
try{
  selectorState = selector.select(selectorTimeout);
} catch (CancelledKeyException ex){
   ;
}
iterator = selector.selectedKeys().iterator();
while (iterator.hasNext()) {
   key = iterator.next();
   if (key.isAcceptable()) {
       myExecutor.execute(getAcceptHandler(key));
   } else if (key.isReadable()) {
       getReadHandler(key).run();
```

# Tip #6: To Thread or not to Thread

## When to use Threads

- We've benchmarked all of the above options and found that the one that perform the best is option C:
  - Execute OP_ACCEPT using the Selector thread and OP_READ on a separate thread
- Reason: OP_ACCEPT is a fast operation, and the overhead of spawning a thread, or context switching to a worker thread significantly increases the time required to execute the accept operation
- Simply a matter of cloning/returning a socket

# Agenda

Introduction

What Is Project Grizzly

Project Grizzly Performance

Overview of NIO

Tricks and Tips

**Summary**

# Summary

*Writing scalable server applications can be a difficult proposition and using Java technology-based NIO effectively presents its own challenges.*

*As a result of the tips-and-tricks we have shared with you as a result of implementing Project Grizzly we hope you will be able to realize some of the same performance and scalability we have realized with Project Grizzly.*

*And, you can always just use Project Grizzly directly if it fits your needs.*

# For More Information

- Project Grizzly
  - https://grizzly.dev.java.net
  - Project Grizzly mailing list
- And our regular blogs
  - http://weblogs.java.net/blog/jfarcand/
  - http://weblogs.java.net/blog/sdo/
  - http://blogs.sun.com/charliebrown/

# Tricks and Tips With NIO Using the Grizzly Framework

**Jeanfrancois Arcand, Senior Staff Engineer**
**Charlie Hunt, Senior Staff Engineer**
**Scott Oaks, Senior Staff Engineer**

SUN Microsystems

TS-2992