

Do you wish you could hear the audio and read the transcription of this session?

Then come to JavaOneSM Online where this session is available in a multimedia tool with full audio and transcription synced with the slide presentation.

JavaOne Online offers much more than just multimedia sessions. Here are just a few benefits:

- 2003 and 2002 Multimedia JavaOne conference sessions
- Monthly webinars with industry luminaries
- Exclusive web-only multimedia sessions on Java technology
- Birds-of-a-Feather sessions online
- Classified Ads: Find a new job, view upcoming events, buy or sell cool stuff and much more!
- Feature articles on industry leaders, Q&A with speakers, etc.

For only \$99.95, you can become a member of JavaOne Online for one year. Join today!

Visit <http://java.sun.com/javaone/online> for more details!



JavaOneSM
Sun's 2003 Worldwide Java Developer Conference

Performance Myths Exposed

Dr. Cliff Click

Senior Staff Engineer
Azul Systems

Writing Fast Java Code

Learn about some “performance tips”
that are now irrelevant

Avoid premature optimization

Write portably performant Java code

Speaker's Qualifications

- **Dr. Click** is a Senior Staff Engineer now at Azul Systems
- Dr. Click was until recently a Senior Staff Engineer at Sun Microsystems
- Dr. Click architected the HotSpotTM Server Compiler
- Dr. Click wrote his first compiler at age 16 and has been writing:
 - Runtime compilers for 15 years, and
 - Optimizing compilers for 10 years

Performance Myths Exposed

There are several Java™ coding styles promoted to help performance, that are now plainly irrelevant.

Agenda

- Compare 7 Java™ VMs against 6 Performance Hacks

The JVM™ Machines

- Running on a 2.4Ghz P4
 - BEA 8.0 (Jrocket)
 - HotSpot -client 1.4.2
 - HotSpot -server 1.4.2
 - IBM 1.3.0
 - IBM 1.4.0 (linux)
- Running on a Sparc 400Mhz US2
 - HotSpot -client 1.4.2
 - HotSpot -server 1.4.2

The Performance Myths

- Using exceptions to exit loops
- Use “final” everywhere
- Avoid try/catch blocks
- Use RTTI to avoid virtual calls
- Avoid synchronization
- Object Pooling

Comparing Performance Myths

- Will present alternate coding styles
 - Clean code vs.
 - Performance-hacked code
- Time both on 7 Java™ VMs
- Looking for relative, not absolute, performance
- No one JVM wins all speed benchmarks
 - JVMs have different strengths, weaknesses
- Performance vs. maintenance tradeoff

Exceptions Should Be Exceptional

- Myth 1: end loop with exception, not test
- Idea: range check happens anyways
 - Why duplicate check in for loop?
 - Run off end of array and catch AIOOB
- BUT: defeats JIT optimizations

Good

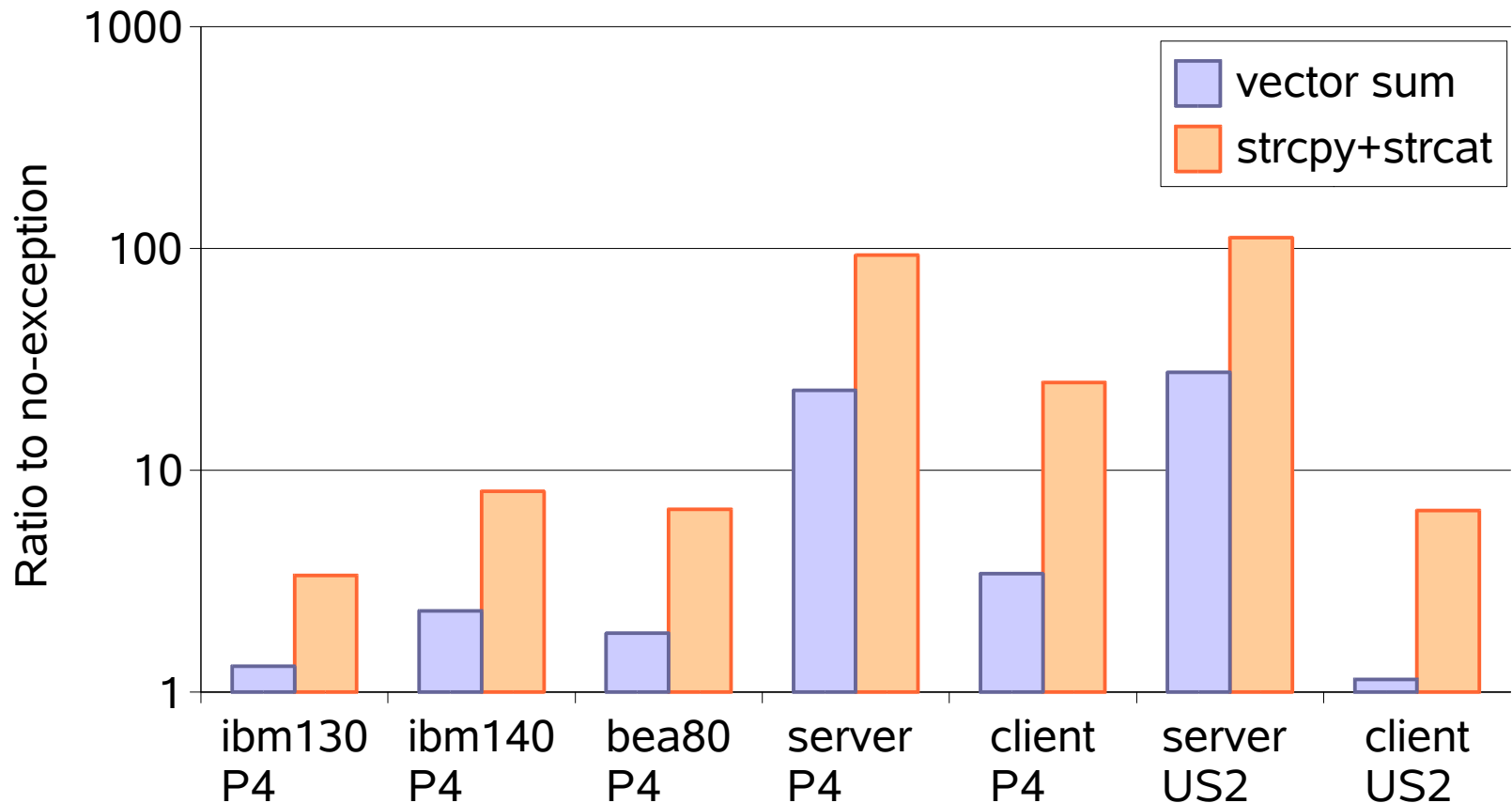
```
for( int i=0; i<A.length; i++ )  
    ...A[i]...
```

Bad

```
try {  
    while( true )  
        ...A[i++]...  
} catch ( AIOOBException e ) {}
```

Cost of Using Exceptions

(bigger favors no-exception code)



Exceptions Should Be Exceptional

- Myth 1': same trick for linked list traversal
- Idea: null check happens anyways
 - Why duplicate check in for loop?
 - Instead, run off end of list and catch NPE
- BUT: defeats JIT optimizations

Good

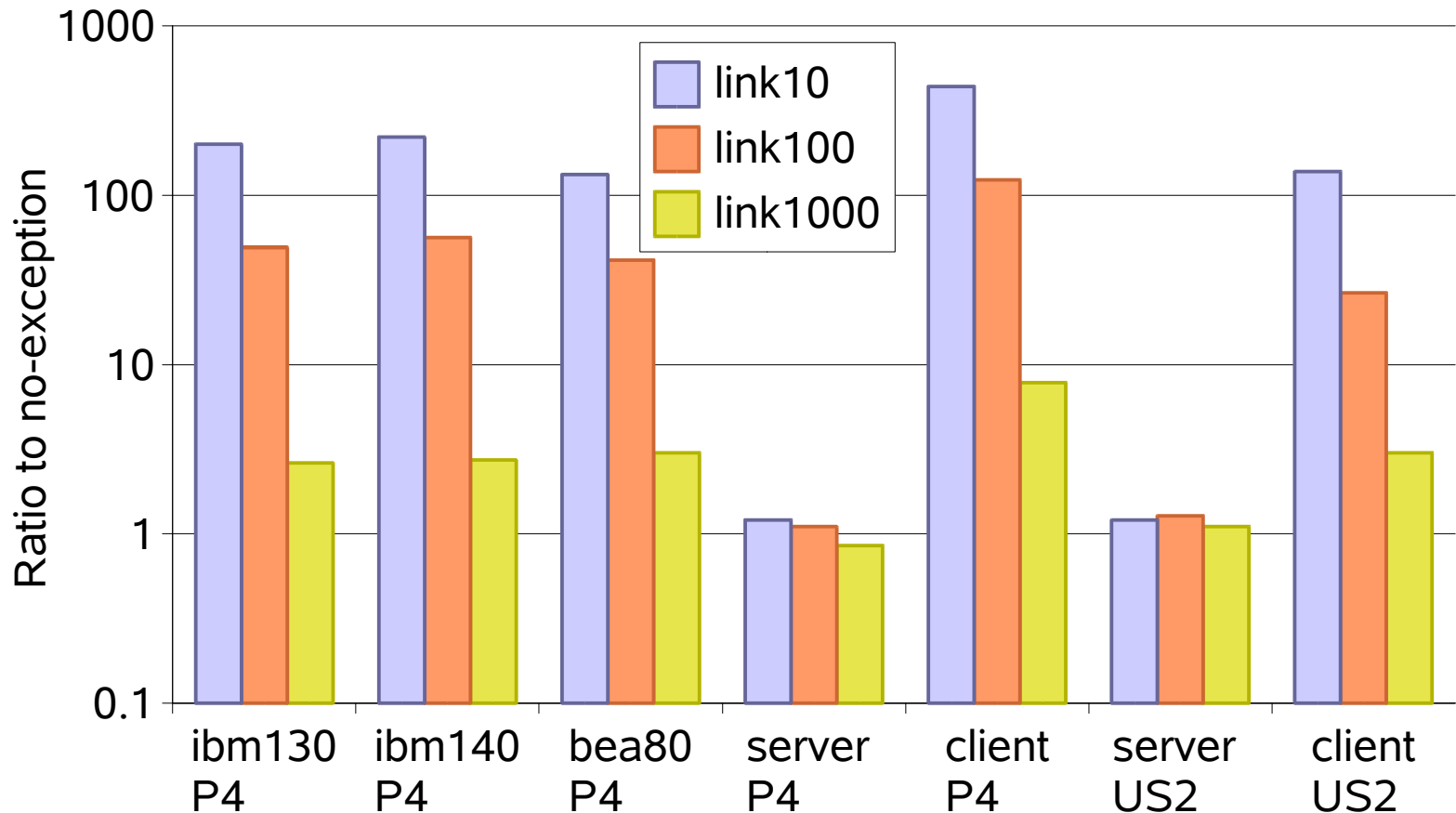
```
while( p != null ) { p.f();  
    p = p.next; }
```

Bad

```
try {  
    while( true ) { p.f(); // NPE here  
        p = p.next; }  
} catch ( NullPointerException e ) {}
```

Cost of Using Exceptions

(bigger favors no-exception)



Use Final Everywhere

- Myth 2: use “final” everywhere
- Idea: allow more inlining
- Reality: inlining (mostly) happens without it

final

```
public final int get() { return fld; }  
public final void set(int x) { fld=x; }
```

no final

```
public int get() { return fld; }  
public void set(int x) { fld=x; }
```

final class

```
public final class Test {  
    public int get() { return fld; }  
}
```

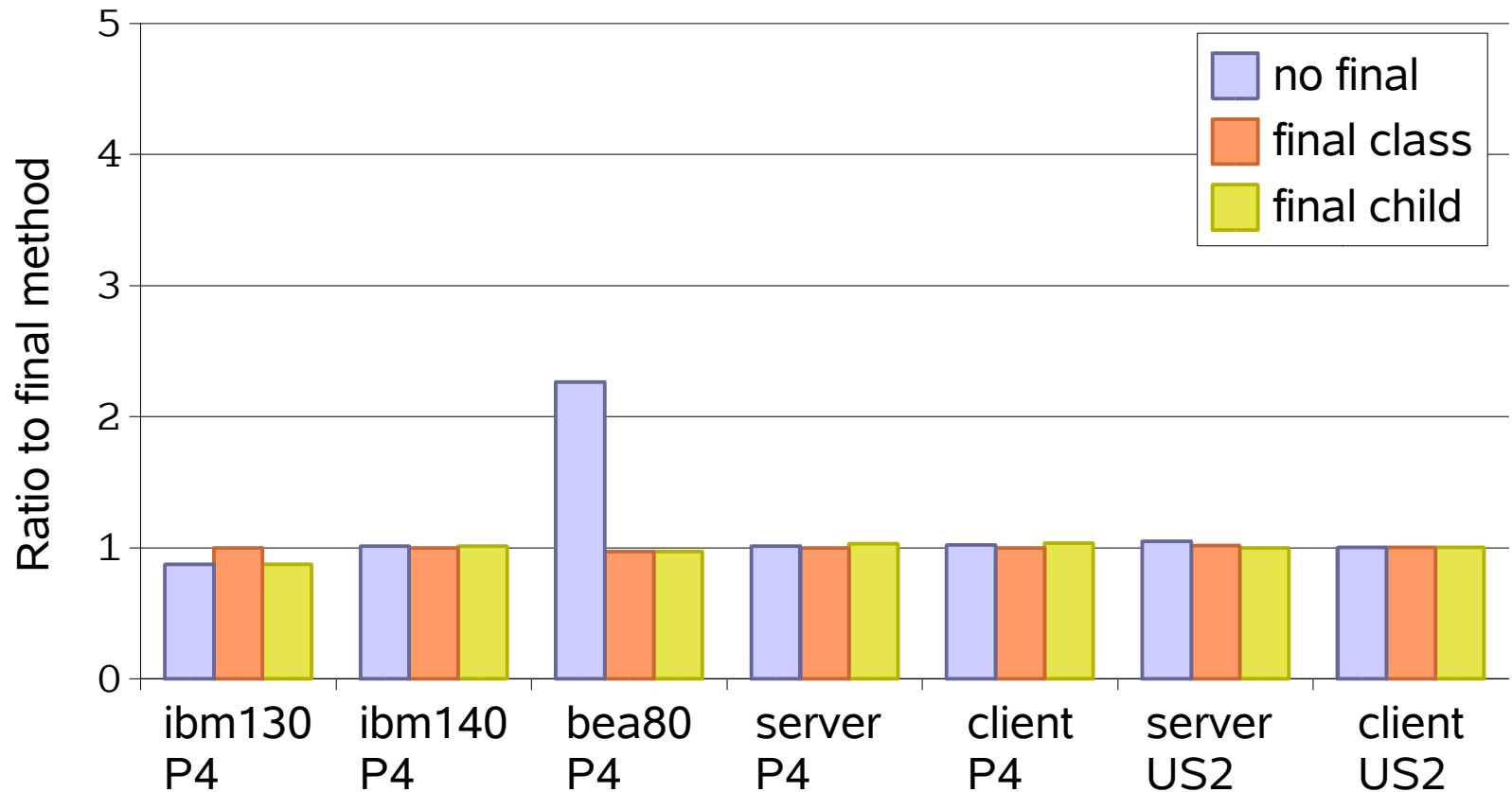
final child

```
public final class Test2 extends Base {
```

Use Final Everywhere

Final vs. non-final accessors

(bigger favors final)



Try/Catch Blocks Are Free (or Not)

- Myth 3: Try/catch blocks are free (or very expensive)
- Reality is more complex
 - Defeats bounds-check opts on some JVMs
 - Free otherwise

vectorsum

```
for( int i=0; i<A.length; i++ )  
    sum += A[i];
```

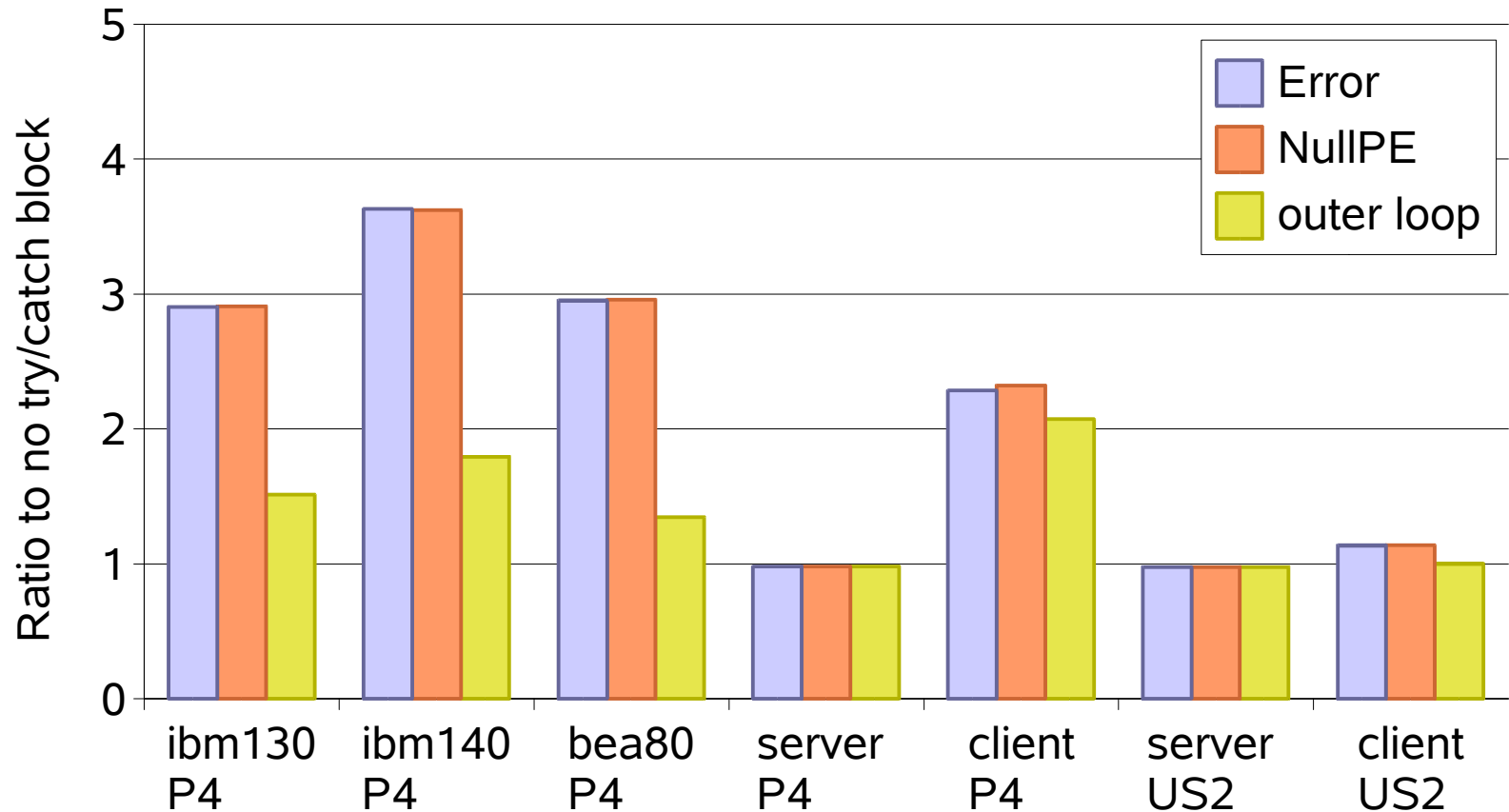
try/catch

```
for( int i=0; i<A.length; i++ ) {  
    try { sum += A[i]; }  
    catch( Error e ){}  
}
```

Try/Catch Blocks Are Free (or Not)

Array Intensive

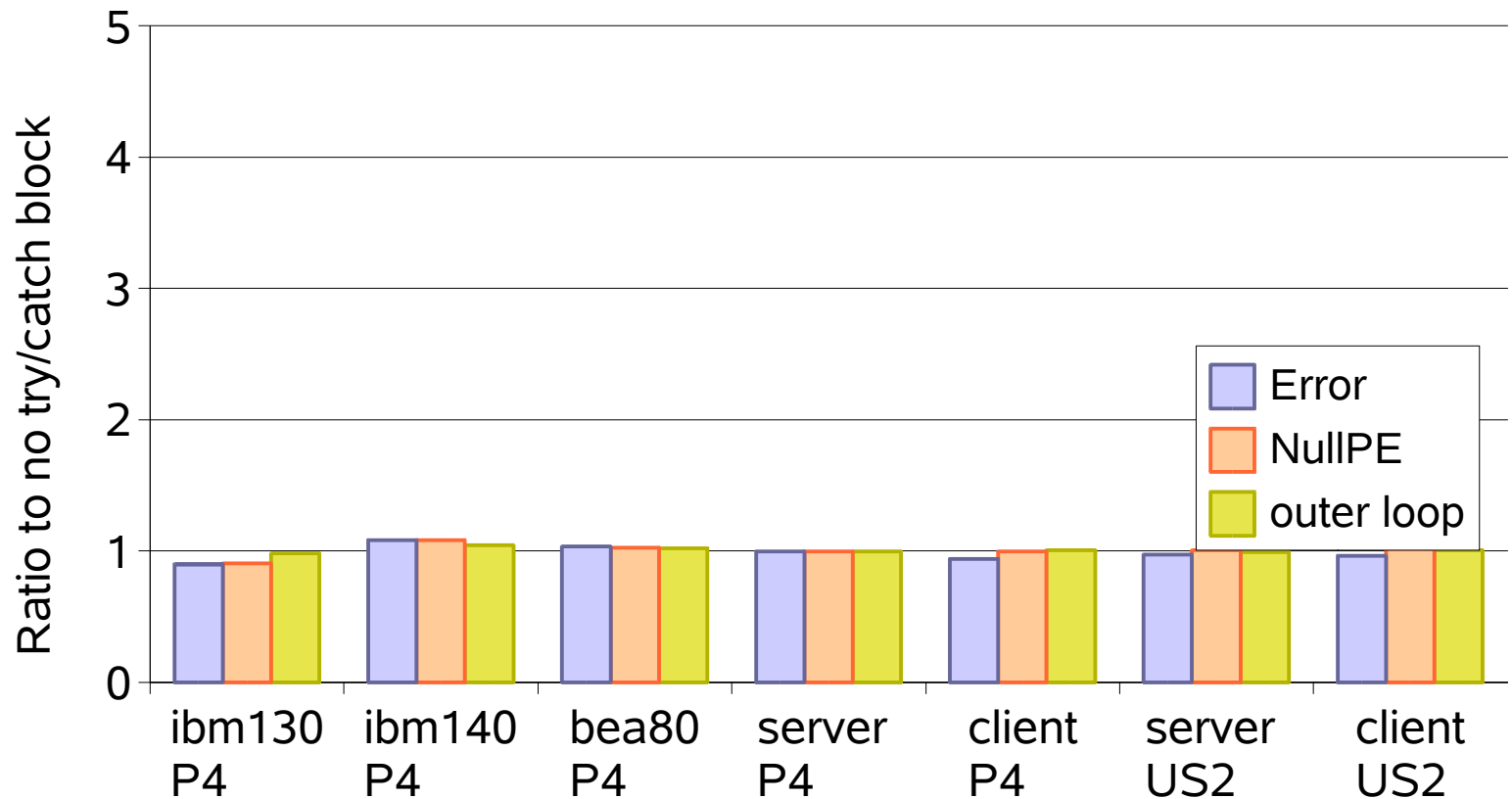
(bigger favors no try/catch)



Try/Catch Blocks Are Free (or Not)

Hashtable Intensive

(bigger favors no try/catch)



RTTI vs. instance-of vs. v-call

- Myth 4: use Run-Time Type Info
 - Replace v-call with instance-of if-tree
 - Or faster switch statement

```
abstract class Base {
    abstract void v_call();
    ...
    v_call(); // dynamic dispatch here
    ...
}
class Child1 extends Base {
    void v_call() { /* Child1-specific */ }
}
class Child2 extends Base {
    void v_call() { /* Child2-specific */ }
}
```

RTTI vs. instance-of vs. v-call

- Myth 4: use Run-Time Type Info
 - Replace v-call with instance-of if-tree

```
abstract class Base {
    ...
    if( this instanceof Child1 )
        ((Child1)this).non_v_call(); // JVM can inline
    else if( this instanceof Child2 )
        ((Child2)this).non_v_call(); // JVM can inline
    else ...
}
class Child1 extends Base {
    void non_v_call() { /*Child1-specific*/ }
}
class Child2 extends Base {
    void non_v_call() { /*Child2-specific*/ }
}
```

RTTI vs. instance-of vs. v-call

- Myth 4: use Run-Time Type Info
 - Small final int per instance
 - Passed into Base constructor

```
abstract class Base {
    final int _rtti;
    Base( int r ) { _rtti=r; }
    ...
}
class Child1 extends Base {
    Child1() { super(1); }
}
class Child2 extends Base {
    Child2() { super(2); }
}
```

RTTI vs. instance-of vs. v-call

- Myth 4: use Run-Time Type Info
 - Use switch statement
 - Hand inline code

```
abstract class Base {
    final int _rtti;
    Base( int r ) { _rtti=r; }
    ...
    switch( _rtti ) {
    case 1: // hand-inline Child1 specifics
    case 2: // hand-inline Child2 specifics
    ...
    }
}
```

RTTI vs. instance-of vs. v-call

- Myth 4: use Run-Time Type Info
 - Replace v-call with instance-of if-tree
 - Or faster switch statement
- But code gets really ugly
 - Instances have more footprint, too

v-call

```
v_call(); // dynamic dispatch here
```

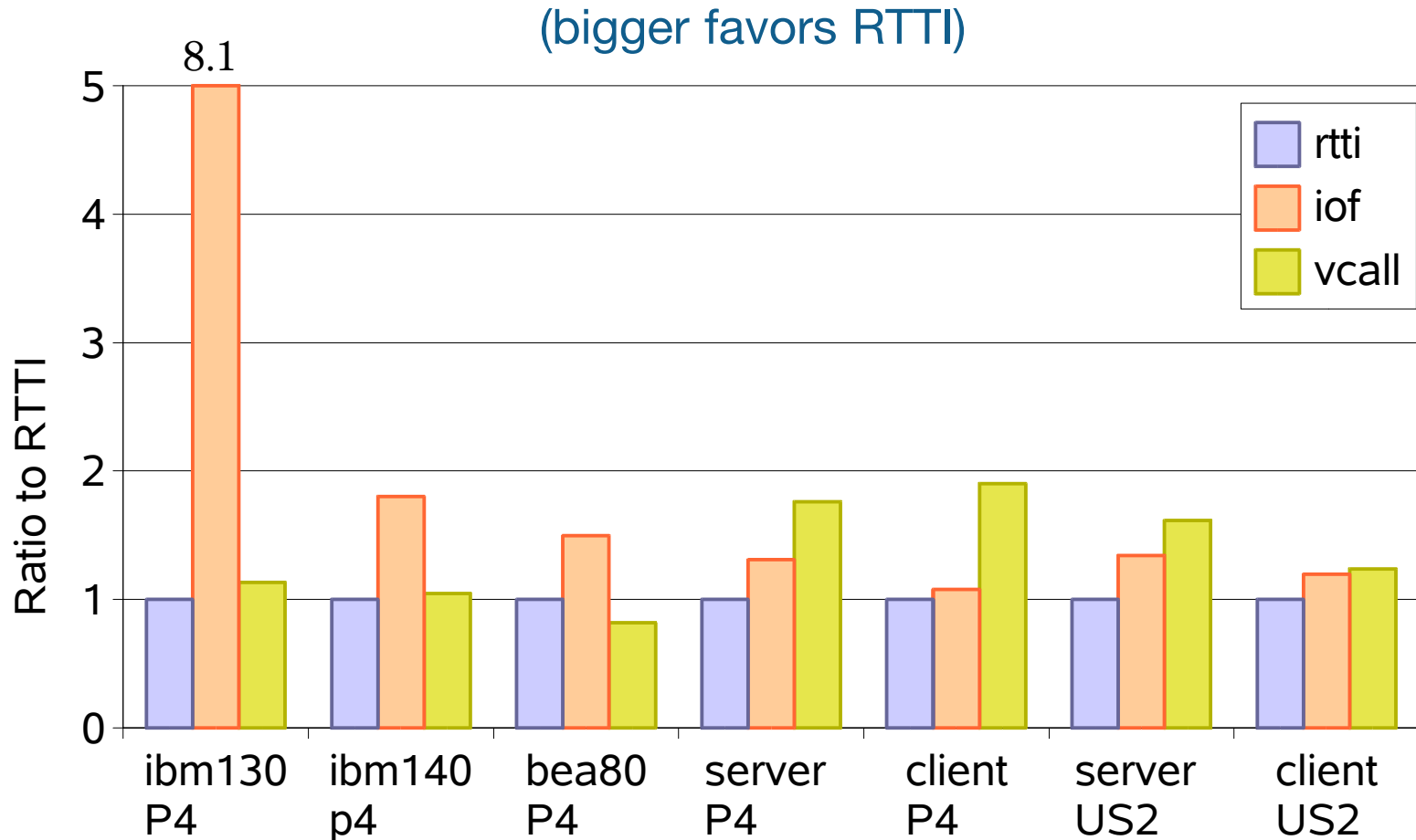
i-of

```
if( this instanceof Child1 )  
    ((Child1) this).non_v_call();
```

RTTI

```
switch( _rtti ) {  
case 1: // hand-inline Child1 specific
```

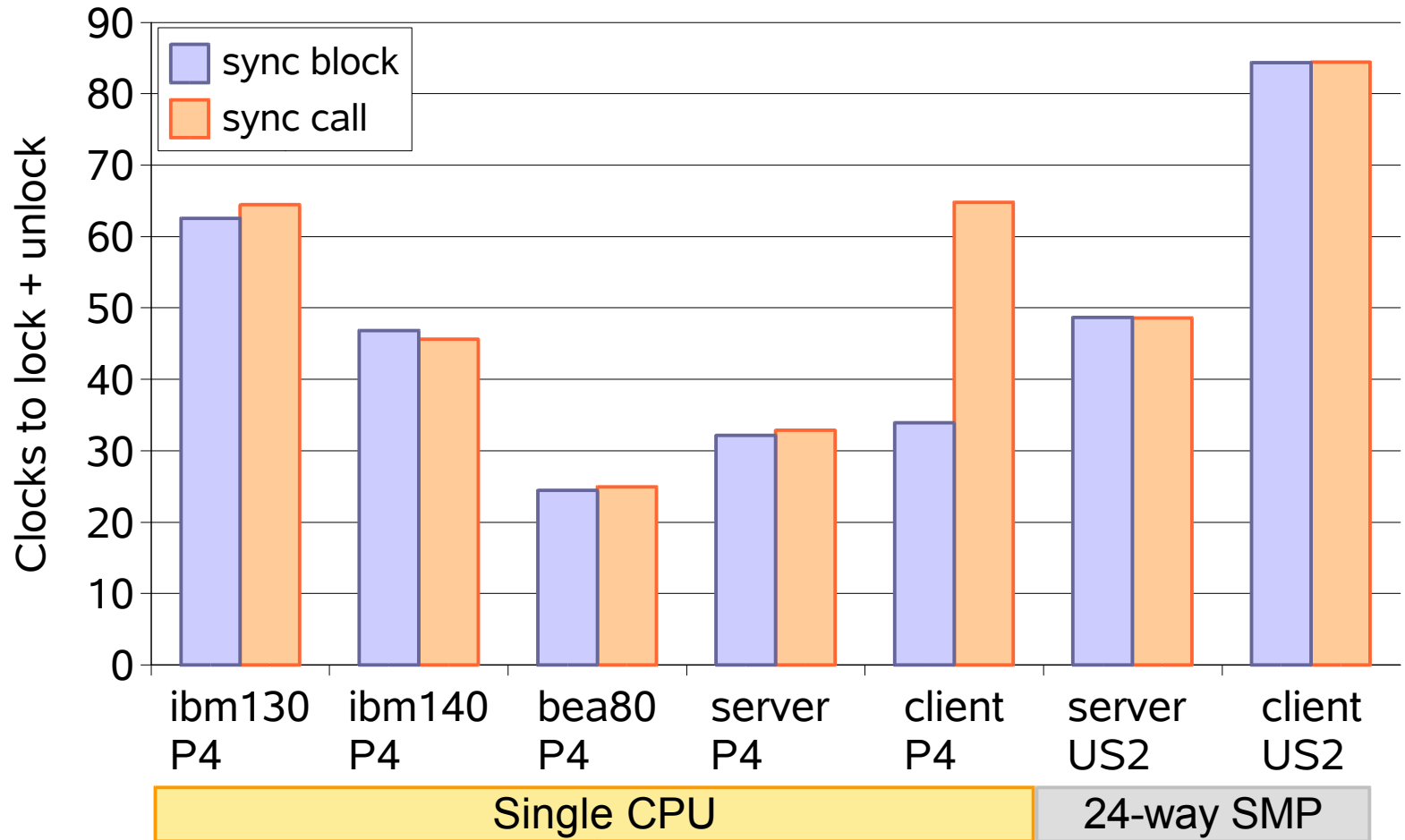
RTTI vs. instance-of vs. v-call



Avoid Synchronization

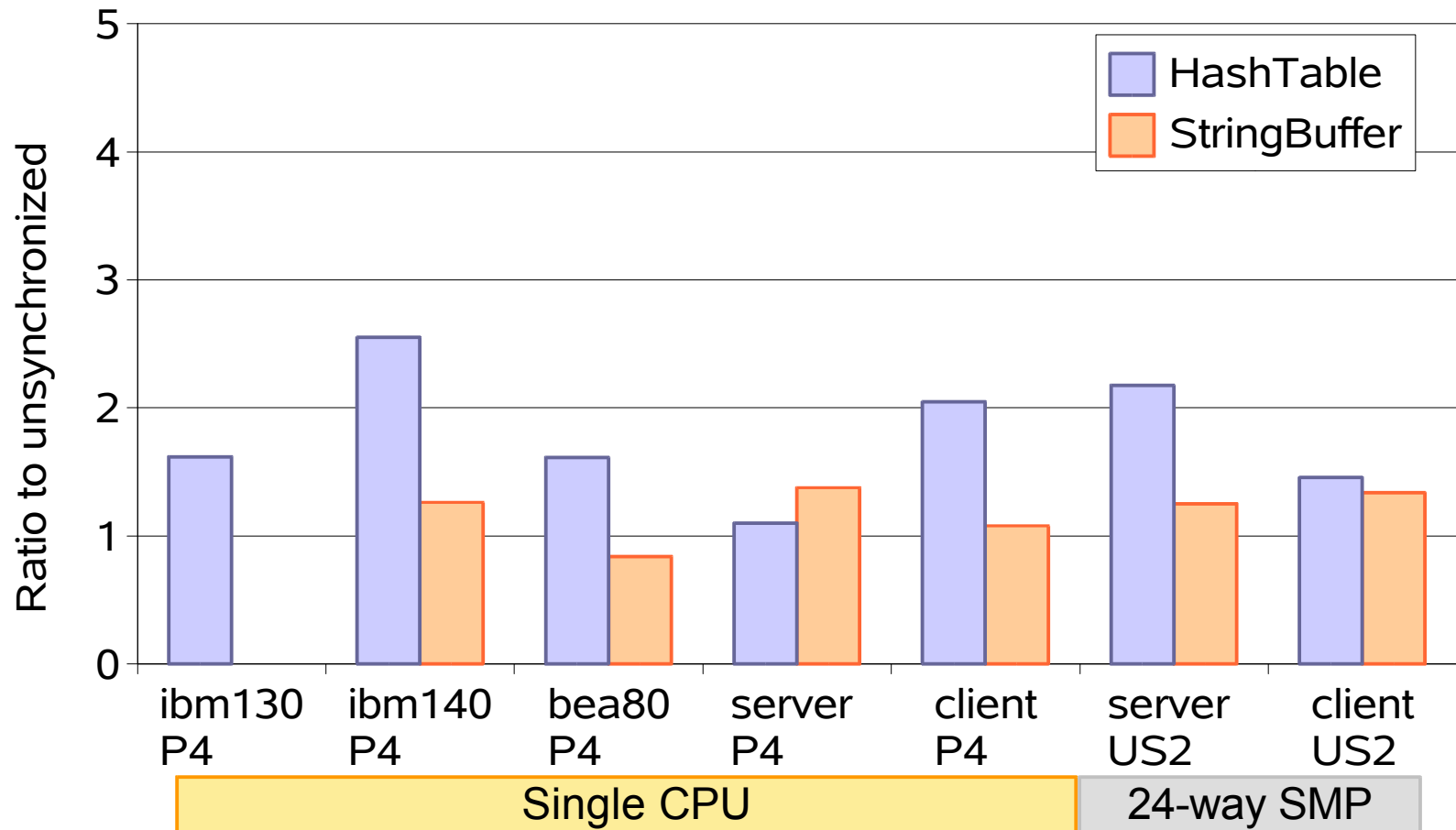
- Myth 5: synchronization is expensive
- Reality: 30–50 clocks for uncontended lock+unlock, on a 2.4Ghz P4
 - Not free, but not really expensive anymore
- Race-condition bugs are devilishly hard to find, fix and debug
- A little over-synchronization isn't fatal
- If you see heavy contention:
 - Maybe need a different algorithm

Raw Synchronize Cost (No Contention)



Hashtable vs. HashMap StringBuffer vs. no-sync SB

(bigger favors no-sync)



Object Pools: New vs. Reuse

- Myth 6: reuse objects instead of new + GC
 - Requires synchronized free list
 - Explicit “free” operation (buggy)
 - Object must be reset
- Usefulness varies by cost to initialize object and frequency of allocation
- Tweak GC flags

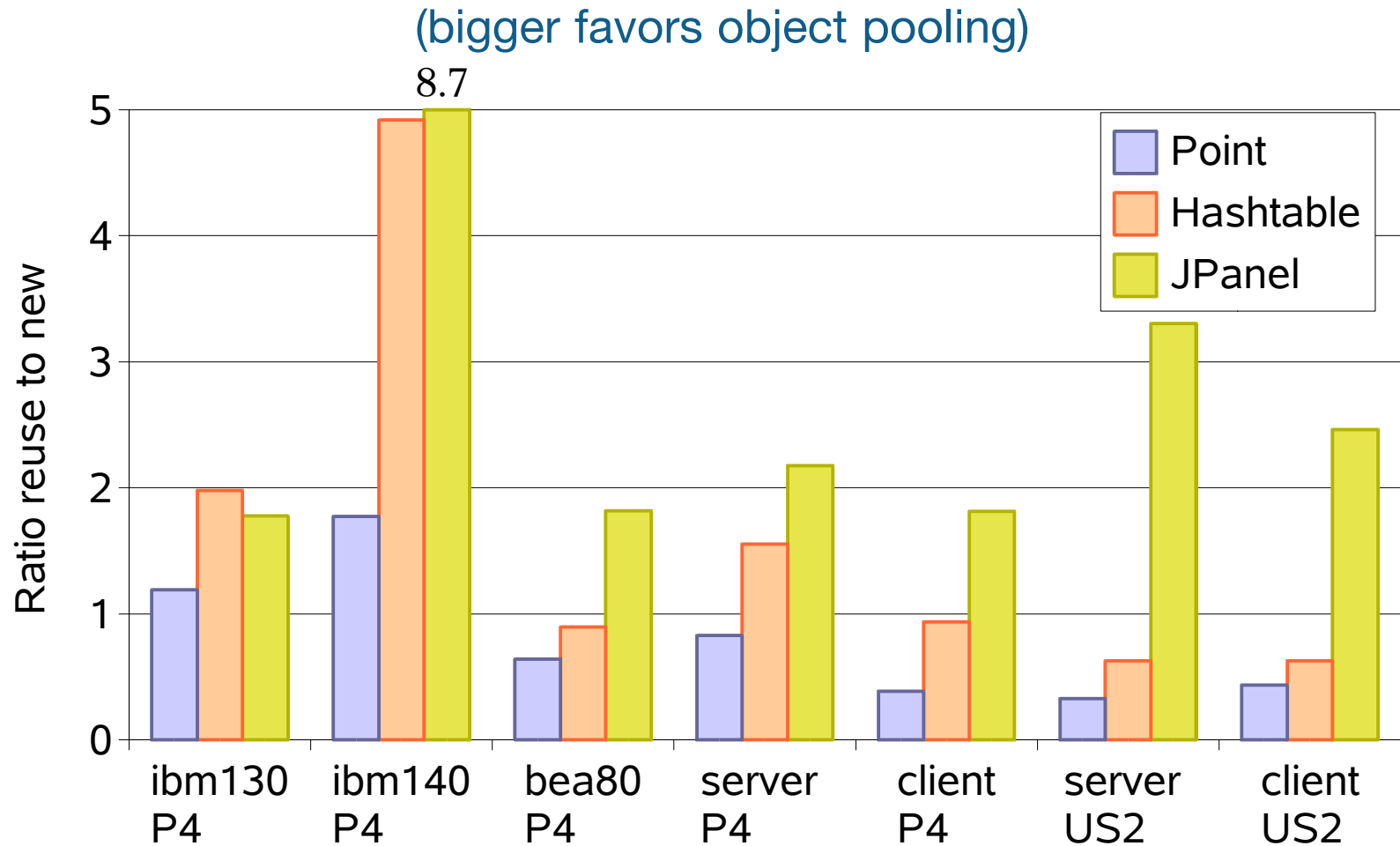
new+GC

```
t = new Hashtable();  
// delete is a no-op; GC gets it
```

Reuse

```
t = Hashtable.make(); // factory  
...  
Hashtable.free(t); // explicit del
```

Object Pools: New vs. Reuse



Object Pools: New vs. Reuse

- Loses for light-weight objects
- A wash for mid-weight objects (Hashtables)
 - (but has maintenance costs)
- A win for big objects
- IBM JIT reuse always wins
 - Appears to lack a generational GC?
 - Bigger heap helps IBM (default is 4M)
 - Bigger heap not a big help with other JVMs

Summary

- Exceptions should be exceptional
- “final” doesn’t help performance
- RTTI is a marginal performance win with a maintenance cost
- try/catch blocks are free*
 - (Except in tight inner loops)
- Synchronization is cheaper than bugs
- GC works: use Object Pools sparingly

Looking Forward

- Escape analysis
 - Short lived local objects will be cheaper
- GC – Concurrent, parallel, pause-less
 - Expect GC to become less overhead
 - And less visible
- Locks, concurrency
 - Multi-threading is easy in Java™
 - And getting cheaper

If You Only Remember One Thing...

Modern Java™ VMs favor
common usage patterns and
clean code.

Java™ Technology Is Slow

- Myth 0: Java™ technology is Slow
- We should all use VB instead
- Please pay your mandatory donation to Redmond on your way out

Q&A

Java™



JavaOneSM

Sun's 2003 Worldwide Java Developer Conference™

JavaTM

java.sun.com/javaone/sf