Advanced Java: Language, Internals, Techniques

Bob Singh

Learning JNI

Texts:

- **JNI**
  - **Java Native Interface Sheng Liang**
    - **java.sun.com/docs/books/jni/**
  - **JDC Training Writing Advanced Applications**
    - **Chap 5 JNI Examples**
  - **JNI FAQ**
    - **java.sun.com/products/jdk/faq/jnifaq.html**

Multilanguage Java Programming

Motivation

- **Wrappers to legacy platform libraries e.g Java wrappers for MPI, OpenGL etc**
- **Embedding JVM in Server environments for additional functionality**
- **Performance when writing time-critical portion in low-level efficient languages**

Multilanguage Java Programming

Issues

- Type-safety of Java can be compromised
  - Extra care needed
- Portability of Java not there
  - May not run on multiple host environments

JNI

Objectives

- **Compatibility**
- **Compromise between VM independence and efficiency**
- **Functionality**
  - **Expose sufficient JVM details**

Learning JNI

JNI functionality

- **Calling into External Native Libraries**
  - **Legacy Libraries**
- **Embedding JVM into application**
  - **Web Browsers, Web Servers, Databases etc**
- **Some JVM exposure**
  - **GetCritical etc stops GC**
  - **Does VM create a copy of Object or not?**

Learning JNI

JNI Overview

- **From Java code**
  - **Call native code written in C, C++**
- **From C/C++ code**
  - **Create, Update Java objects**
  - **Call Java Methods**
  - **Load classes and obtain class information**

Learning JNI

Data types

- **Primitive types map straightforward**
  - **Java type `int` maps to JNI C/C++ type `jint`**
- **Reference types passed as opaque references**
  - **C pointer types that refer to internal data structures in the JVM**
  - **Manipulate them through JNI functions**

Learning JNI

HelloWorld

```
class HelloWorld {
    private native void print();
    public static void main(String[] args) {
        new HelloWorld().print();
    }
    static {
        System.loadLibrary("HelloWorld");
    }
}
```

Learning JNI

HelloWorld

```
#include <jni.h>
#include <stdio.h>
#include "HelloWorld.h"

JNIEXPORT void JNICALL
Java_HelloWorld_print(JNIEnv *env, jobject obj)
{
    printf("Hello World!\n");
    return;
}
```

Learning JNI

HelloWorld

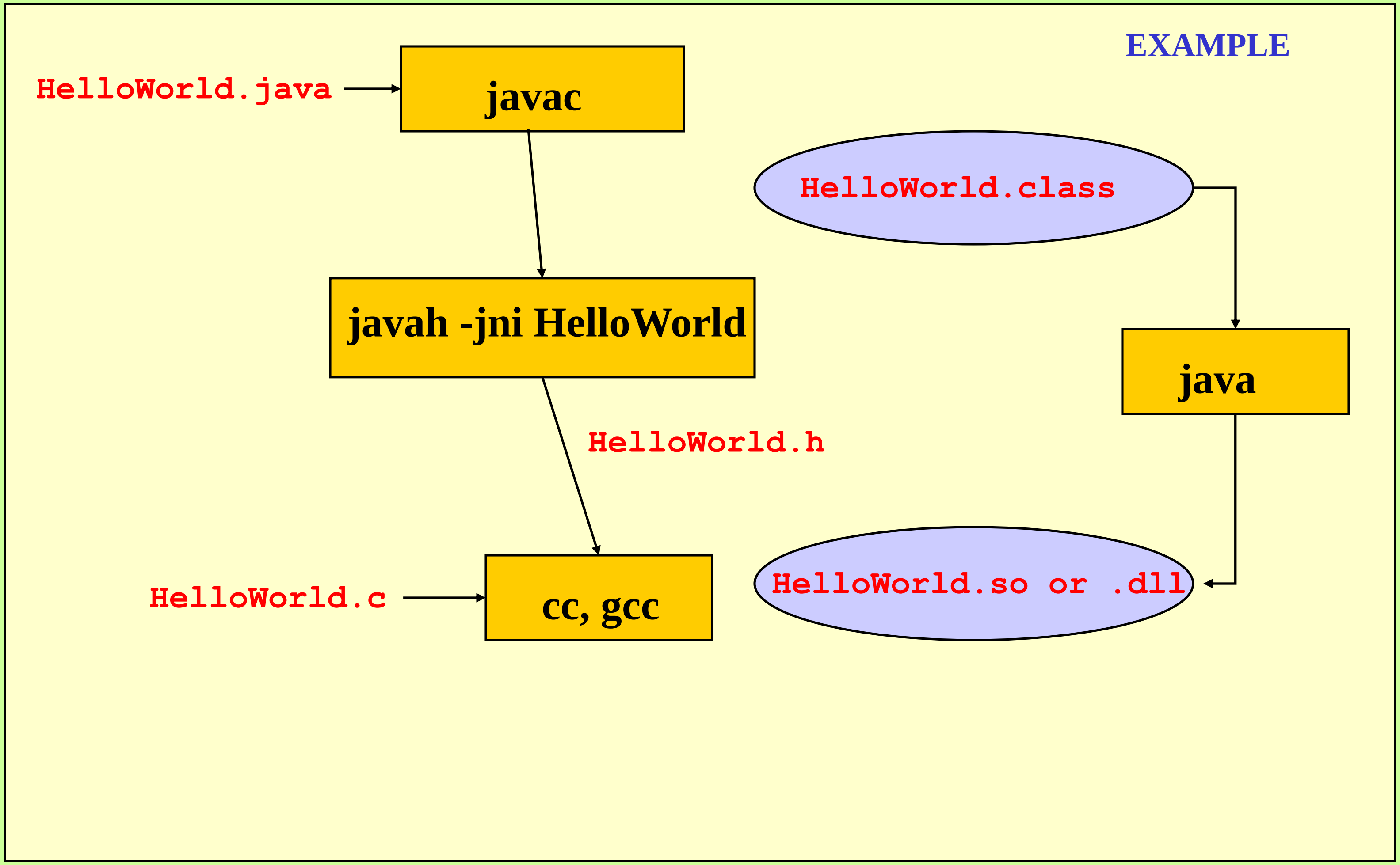
```
Solaris
cc -G -I/java/include -I/java/include/solaris
HelloWorld.c -o libHelloWorld.so

Linux
gcc -o libHelloWorld.so -shared -Wl,-
soname,libHelloWorld.so -I/home/jdk1.2/include -
I/home/jdk1.2/include/linux HelloWorld.c -static -lc

Win32

cl -Ic:\java\include -Ic:\java\include\win32 -MD -LD
HelloWorld.c -FeHelloWorld.dll
```

Learning JNI



Learning JNI

Invocation Overview

```
JavaVM *jvm;
JNIEnv *jni_env;

JNI_CreateJavaVM(&jvm, &jni_env, &args);

jclass cls = jni_env->FindClass(“Hello”);
jmethodID mid = jni_env->GetStaticMethodID(cls, “run”,
“()V”);
jni_env->CallStaticVoidMethod(cls, mid);

jvm->DestroyJavaVM();
```

Learning JNI

Data types

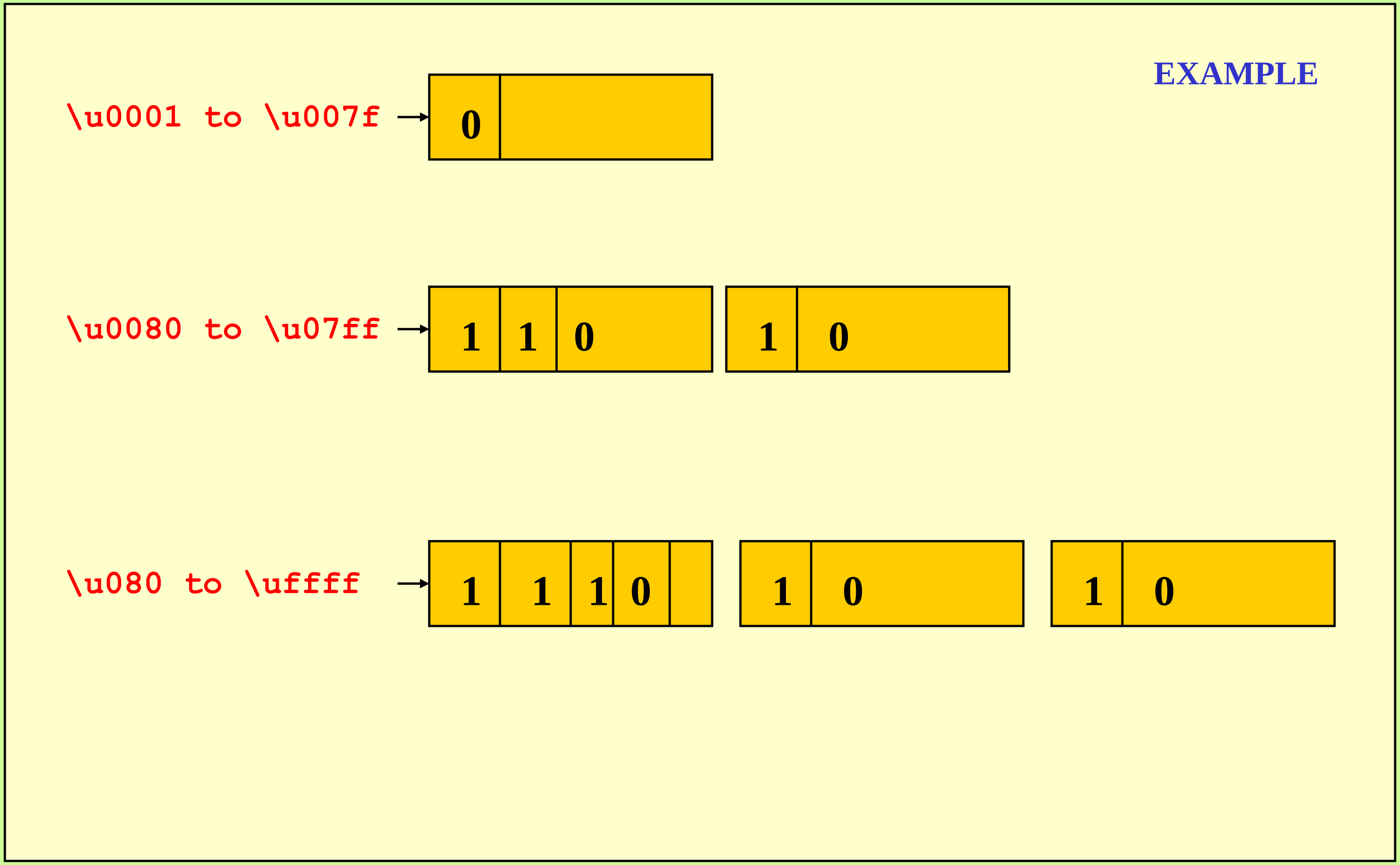
- `jobject` supertype
- subtypes `jstring, jclass, jarray`
  - Commonly used reference types
  - jstring corresponds to java.lang.String
  - jclass corresponds to java.lang.Class
  - jobjectarray, jbooleanarray etc are subtypes of jarray

Learning JNI

String types

- **java strings Unicode**
- **C strings in byte format**
- **UTF allows encoding of Unicode (Unicode Transformation Format)**
  - **non-null ASCII represented as C byte (8 bits)**
    - **\u0001 to \u007f represented by 8 bits**
    - **\u0080 to \u07ff represented by 16 bits**
    - **\u0800 to \uffff represented by 24 bits**

Learning JNI



Other String Functions

- **JDK 1.1.x and Java2**
  - **GetStringChars, ReleaseStringChars, GetStringLength**
  - **GetStringUTFChars, ReleaseStringUTFChars, GetStringUTFLength**
  - **NewString, NewStringUTF**
- **Java2**
  - **GetStringCritical, ReleaseStringCritical**
  - **GetStringRegion, GetStringUTFRegion**

Learning JNI

Prompt

```
class Prompt {

    // native method that prints a prompt and reads a line
    private native String getLine(String prompt);

    public static void main(String args[]) {
        Prompt p = new Prompt();
        String input = p.getLine("Type a line: ");
        System.out.println("User typed: " + input);
    }

    static {
        System.loadLibrary("Prompt");
    }
}
```

Learning JNI

Prompt

```
#include <jni.h>
#include <stdio.h>
#include "Prompt.h"

JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
    char buf[128];
    const char *str;
    str = (*env)->GetStringUTFChars(env, prompt, NULL);
    if (str == NULL) {
        return NULL; /* OutOfMemoryError already thrown */
    }
    printf("%s", str);
    (*env)->ReleaseStringUTFChars(env, prompt, str);
    /* We assume here that the user does not type more than
     * 127 characters */
    scanf("%s", buf);
    return (*env)->NewStringUTF(env, buf);
}
```

Learning JNI

Prompt

```
#include <jni.h>
#include <stdio.h>
#include "Prompt.h"

JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
    char buf[128];
    const char *str;
    str= prompt;
    printf("%s", str);
    scanf("%s", buf);
    return (*env)->NewStringUTF(env, buf);
}
```

Learning JNI

Prompt

C

```
str = (*env)->GetStringUTFChars(env, prompt, NULL);
```

C++

```
str = env->GetStringUTFChars(env, prompt, NULL);
```

Learning JNI

Array types

- **Primitive Arrays**
  - **Similar to Strings**
  - **JDK 1.1.x**
    - **Get<Type>ArrayRegion, Set<Type>ArrayRegion**
    - **Get<Type>ArrayElements, Release<Type>ArrayElements**
    - **GetArrayLength**
    - **New<Type>Array**
  - **Java2 (finer control)**
    - **GetPrimitiveArrayCritical, ReleasePrimitiveArrayCritical**
- **Object Arrays**
    - **GetObjectArrayElement, SetObjectArrayElement**

Learning JNI

Summation

```
class Summation {

    // native method that sums the given array
    private native int sum(int[] arr);

    public static void main(String args[]) {
        Summation p = new Summation();
        int arr[] = new int[10];
        for(int j=0; j < arr.length; j++)
           arr[j] = j;
        int sum = p.sum(arr);
        System.out.println(”Sum = " + sum);
    }

    static {
        System.loadLibrary(”Summation");
    }
}
```

Learning JNI

Summation Version #1

```
#include <jni.h>
#include <stdio.h>
#include "Summation.h"

JNIEXPORT jint JNICALL
Java_Summation_sum(JNIEnv *env, jobject obj, jintArray arr)
{
    jint buf[10];
    jint j, sum = 0;
    (*env)->GetIntArrayRegion(env, arr, 0, 10, buf);
    for(j=0; j < 10; j++)
      sum += buf[j];
    return sum;
}
```

Learning JNI

Summation Version #2

```
#include <jni.h>
#include <stdio.h>
#include "Summation.h"

JNIEXPORT jint JNICALL
Java_Summation_sum(JNIEnv *env, jobject obj, jintArray arr)
{
    jint *carr;
    jint j, sum = 0;
    carr = (*env)->GetIntArrayElements(env, arr, NULL);
    if (carr == NULL)
       return 0;
    for(j=0; j < 10; j++)
      sum += carr[j];
    (*env)->ReleaseIntArrayElements(env, arr, carr, 0);
    return sum;
}
```

Learning JNI

Field Access

```
class StringArrayInit {

    // native method that initializes an Array of Strings
    private native String[] initStringArr();

    public static void main(String args[]) {
        StringArrayInit p = new StringArrayInit();
        String[] arr = p.initStringArr();
        for(int j=0; j < arr.length; j++) {
           System.out.println(arr[j]);
        }
    }

    static {
        System.loadLibrary(”StringArrayInit");
    }
}
```

Learning JNI

ObjectInitialization

```
#include <jni.h>
#include <stdio.h>
#include "StringArrayInit.h"

JNIEXPORT jobjectArray JNICALL
Java_StringArrayInit_initStringArr(JNIEnv *env, jobject obj) {
    jobjectArray ret;
    jint j;
    char *msg[3] = {"abc”, “def”, “ghi”};
    jclass strCls = (*env)->FindClass(env, “java/lang/String”);
    jstring str = (*env)->NewStringUTF(“”);
    ret = (*env)->NewObjectArray(3, strCls, str);
    for(j=0; j < 3; j++)
      (*env)->SetObjectArrayElement(ret, j,
                                             (*env)->NewStringUTF(msg[j]));
    return ret;
}
```

Fields and Methods

- **Fields**
  - **Instance**
    - **GetFieldID**
    - **Get<Type>Field and Set<Type>Field**
  - **Static**
    - **GetStaticFieldID**
    - **GetStatic<Type>Field and SetStatic<Type>**
- **Methods**
    - **Instance: GetMethodID, Call<Type>Method**
    - **Static: GetStaticMethodID, CallStatic<Type>Method**

Learning JNI

Method Calling Pattern

- **Retrieve the class**
  - **FindClass through name**
  - **GetObjectClass from a jobject**
  - **Directly from a jclass**
- **Retrieve the MethodID**
  - **GetMethodID, or GetStaticMethodID**
- **Call the Method**
  - **Using Call<Type>Method**

Learning JNI

Method Calling

- **Constructors**
  - **“<init>” method-name and “V” return type**
- **Method Descriptor**
  - **Use javap -s**
- **SuperClass methods**
  - **CallNonVirtual<Type>Method**

Learning JNI

FieldAccess

```
class InstanceFieldAccess {
    private String s;

    private native void accessField();
    public static void main(String args[]) {
        InstanceFieldAccess c = new InstanceFieldAccess();
        c.s = "abc";
        c.accessField();
        System.out.println("In Java:");
        System.out.println("  c.s = \"" + c.s + "\"");
    }
    static {
        System.loadLibrary("InstanceFieldAccess");
    }
}
```

Learning JNI

FieldAccess

```
#include <jni.h>
#include <stdio.h>
#include "InstanceFieldAccess.h"

JNIEXPORT void JNICALL
Java_InstanceFieldAccess_accessField(JNIEnv *env, jobject obj)
{
    jfieldID fid;   /* store the field ID */
    jstring jstr;
    const char *str;
    jclass cls = (*env)->GetObjectClass(env, obj);
    printf("In C:\n");
   /* Look for the instance field s in cls */
    fid = (*env)->GetFieldID(env, cls, "s",
                             "Ljava/lang/String;");

    /* Read the instance field s */
    jstr = (*env)->GetObjectField(env, obj, fid);
    str = (*env)->GetStringUTFChars(env, jstr, 0);
    printf("  c.s = \"%s\"\n", str);
    (*env)->ReleaseStringUTFChars(env, jstr, str);
    jstr = (*env)->NewStringUTF(env, "123");
    (*env)->SetObjectField(env, obj, fid, jstr);
}
```

Learning JNI

Method and Field Lookup

- **Expensive**
- **Use Caching for Field and Method ID's**
  - **In the static initalizer setup initID's**
- **Need to know about other things for caching**
  - **Stale ID's if Class gets Unloaded and Reloaded**

Learning JNI

Opaque References

- **C pointer types that refer to JVM internal data structures, manipulated through JNI functions**
  - **Local**
    - **Valid for duration of native method**
    - **Illegal to use across threads**
    - **Illegal to cache across invocations**
    - **NewObject, FindClass etc return local Refs**
  - **Global**
    - **Valid until programmer frees**
  - **Weak Global**
    - **new in Java2**

Learning JNI

GlobalRefs from java.net

```
static jclass socketExceptionCls;
JNIEXPORT void JNICALL
Java_java_net_PlainSocketImpl_socketCreate(JNIEnv *env, jobject o) {

   if (socketExceptionClass == NULL) {
      socketExceptionCls = (*env)->FindClass(env,
"java/net/SocketException");
      socketExceptionCls = (jclass)(*env)->NewGlobalRef(env,
socketExceptionCls);
      :
   }
}
```

Learning JNI

LocalRef Handling in Java2

```
#define NREFS

for(j=0; j < len; j++) {
  if ((*env)->PushLocalFrame(env, NREFS) < 0) {
      Error
  }
  jstr = (*env)->GetObjectArrayElement(env, arr, j);
   /* Process jstr */

  (*env)->PopLocalFrame(env, NULL);
}
```

Learning JNI

Exceptions

- **Exception condition encoded in JNI API**
- **ExceptionCheck**
  - **Needed for Call<Type>Method**
- **ExceptionDescribe, ExceptionClear**
- **ThrowNew**
  - **Throwing New Exceptions**

Learning JNI

JNI Gotchas

- Expensive on some JVM's
- Need extensive Error Checking
- JNIEnv, Local Refs etc are per thread
  - Don't use across threads by caching
- Threading Models should match

Learning JNI

JNI Java2 tips

- **verbose:jni option**
- **-Xcheck:jni**
  - **Very expensive**
- **EnsureLocalCapacity, PushLocalFrame, PopLocalFrame**
- **GetStringCritical, GetPrimitiveArrayCritical**
  - **Direct pointer to JVM possible**

JNI Performance Issues

Calls to Native Methods

- **Function/method calls in general are costly**
  - **Creation of new frame on execution stack**
  - **Save/restore of current function's state**
- **Conversion of *calling convention***
  - **Method implementation may not be known at compile time**
  - **Interpreter and JIT calling conventions traditionally designed for fast method calling within their respective worlds**
  - **Newer systems must optimize paths between these worlds**

JNI Performance Issues

Accessing Arrays in Native Code

- **Directly in Java object space**
  - **Garbage collector cannot move object until native code has finished**
    - **Object can be pinned**
    - **GC can be disabled**
- **Copying array into native memory space**
  - **Object must be copied twice to access and modify**

JNI Performance Issues

JNI Array Access Routines

- Get<Type>ArrayRegion
  - Obtain pointer to copy of specified region of array
  - Any changes can be committed to Java object with Set<Type>ArrayRegion call if desired
- Get<Type>ArrayElements
  - Gets entire array, changes committed to Java object at accompanying Release<Type>ArrayElements call
  - Allows VM to choose implementation
    - Pinning: most Sun-based VM's
    - Copying: Sun Production VM, HotSpot

JNI Performance Issues

JNI Array Access Routines

- **GetPrimitiveArrayCritical**
  - **Gets entire array, changes committed to Java object at accompanying SetPrimitiveArrayCritical call**
  - **Instructs VM to avoid copying if at all possible**
    - **Pinning**
    - **Disabling garbage collector**
  - **Restrictions while array is held**
    - **No JNI calls**
    - **No blocking operations**

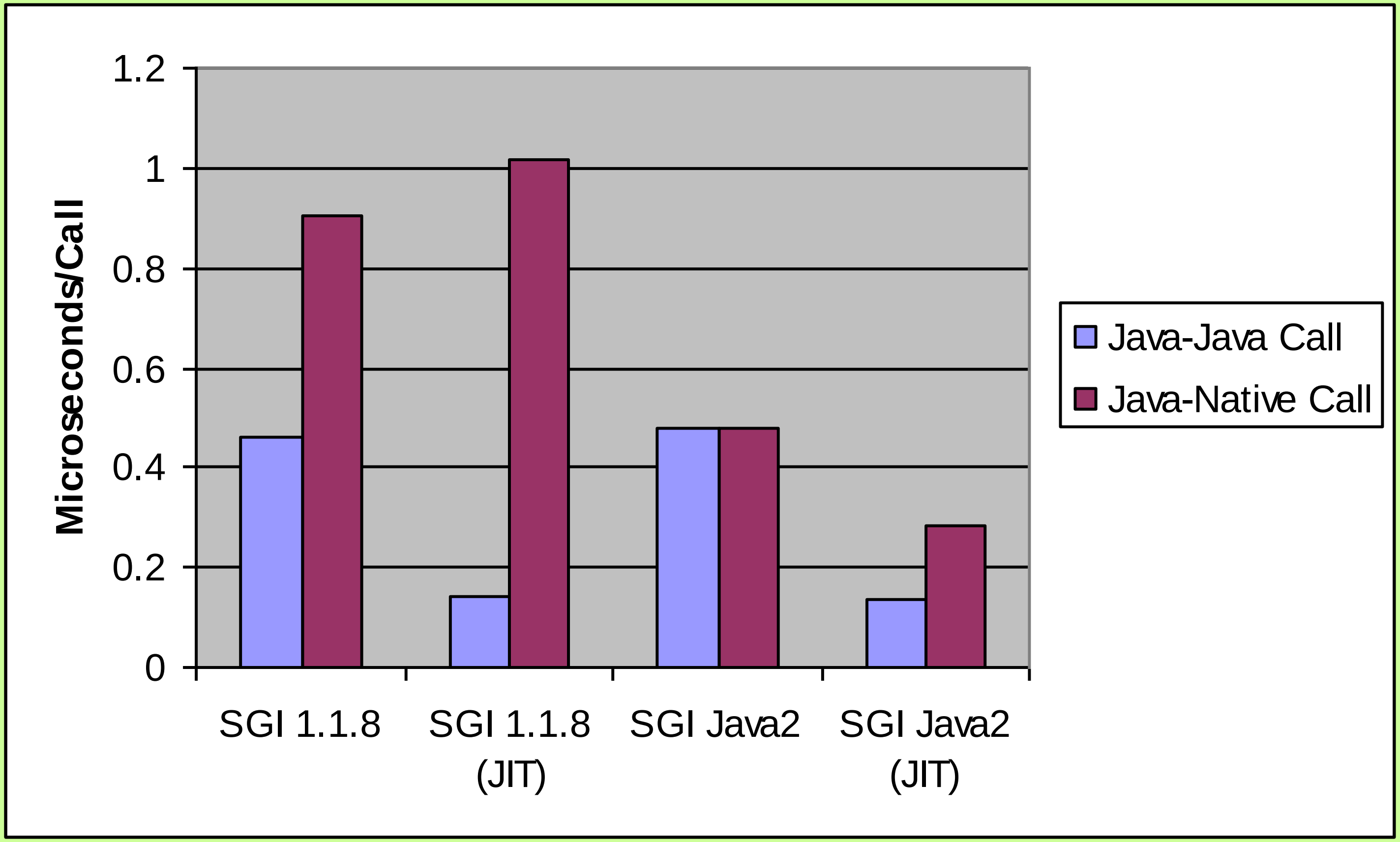
JNI Benchmarking

Goals

- **Measurement of all important native functionality**
  - **Calls into native code, parameter passing**
  - **Native code access to Java fields and methods**
  - **Native code access to Java arrays**
  - **Native code management of Java references (global, local, weak, …)**
- ***Microbenchmarks* to measure µs necessary for each operation, and comparison of different implementations**

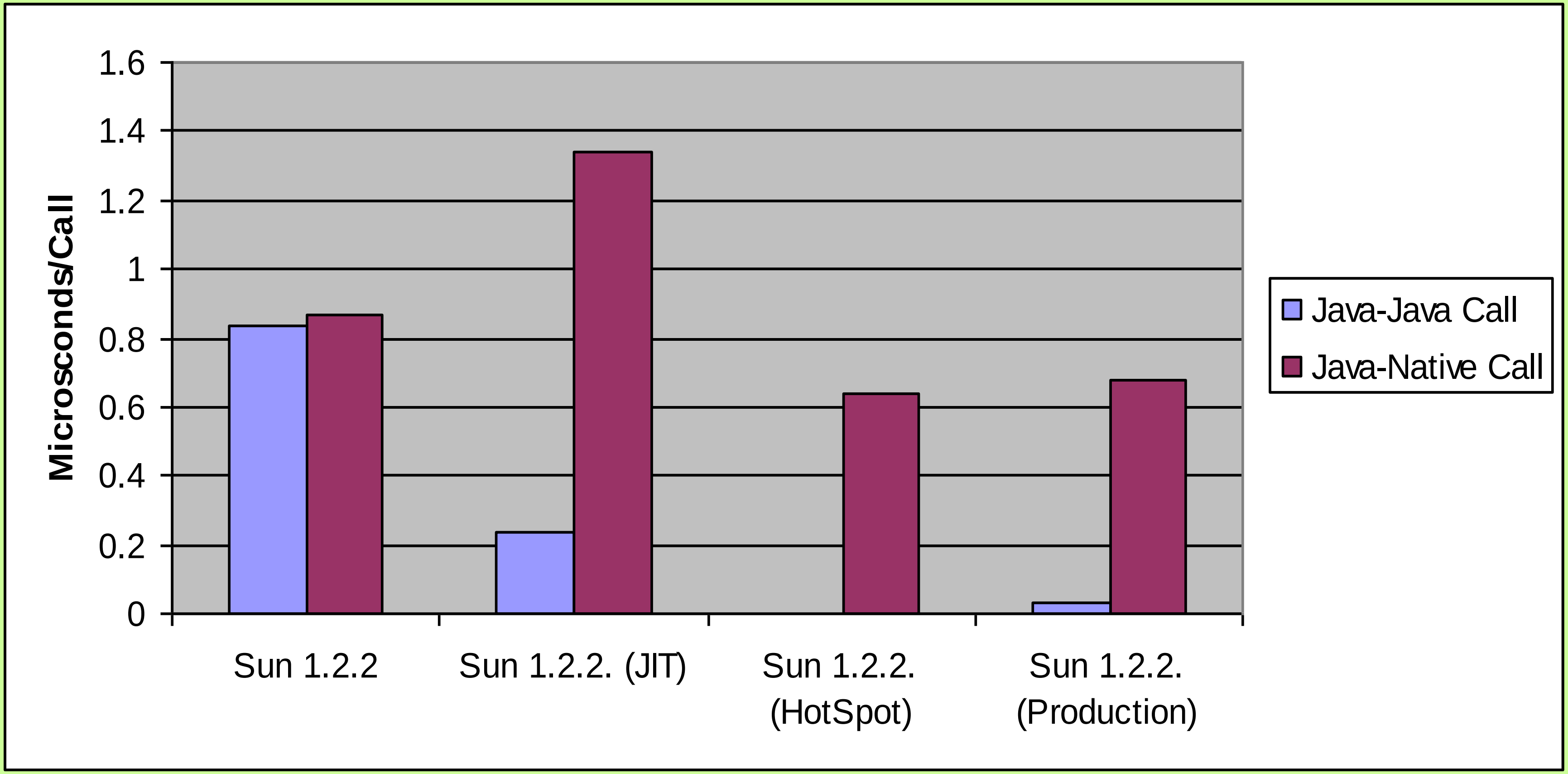
JNI Performance

Native Call Cost: SGI Results



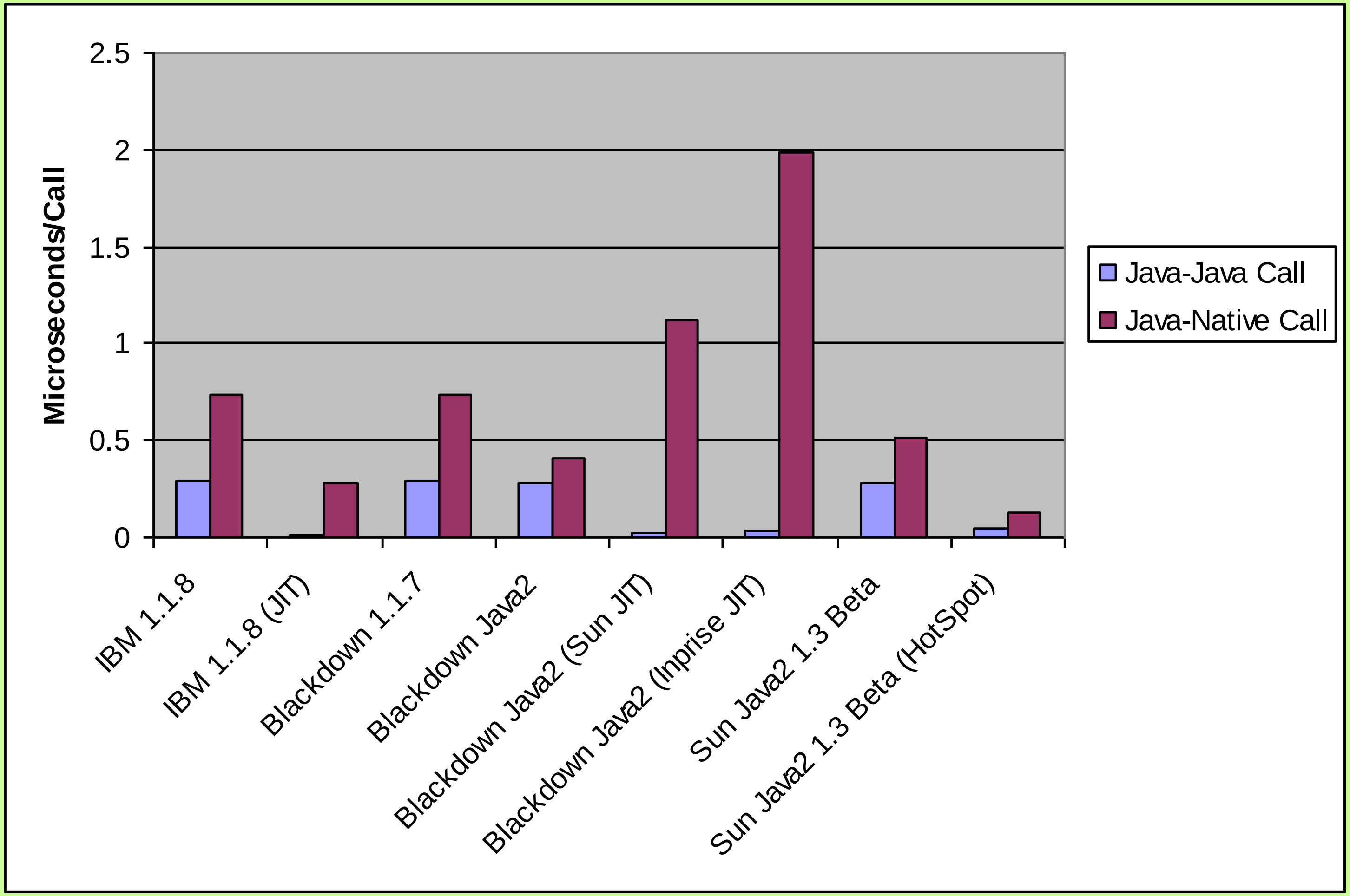
JNI Performance

Native Call Cost: Sun Results



JNI Performance

Native Call Cost: Linux Results



JNI Performance

Native Call Cost: Conclusions

- **Native method overhead is significant**
- **Overhead has decreased from JDK 1.1.x to Java2**
- **JIT compilation can make JNI calls…**
  - **significantly faster than under the interpreter (SGI Java2, IBM 1.1.8, HotSpot)**
  - **significantly slower (Blackdown Java2)**

More Benchmark references

For you observations

JNI Benchmarking

Java Timer Class from UCSD Benchmark

```
public class Timer
{
    /* various field and method definitions ... */
    public void mark() {
        base_time = System.currentTimeMillis();
    }
    public void record() {
        elapsed_time += (System.currentTimeMillis() -
                         base_time);
    }
    public float elapsed() {
        return ((float) elapsed_time) / 1000;
    }
}
```

JNI Benchmarking

Use of Timer Class

```
t.mark();
longRunningTest();
t.record();
System.err.println(name + " " +
                   t.elapsed());
```

```
for (int i=0; i<iterations; i++) {
    t.mark();
    longRunningTest();
    t.record();
}
System.err.println(name + " " +
           (t.elapsed() / iterations));
```

If the test duration is much larger than the granularity of the timer...

JNI Benchmarking

Use of Timer Class

```
controlTimer.mark();
for (int i=0; i<iterations; i++) {
    // unremovable code ...
}
controlTimer.record();

t.mark();
for (int i=0; i<iterations; i++) {
    // unremovable code ...
    veryQuickTest();
}
t.record();

System.err.println(name + " " +
      ((t.elapsed() - controlTimer.elapsed())
      * (1000000.0 / iterations)));
```

Typical durations of our JNI tests (in µs) are much smaller than the granularity of `currentTimeMillis()` (ms), so we measure many executions inside a loop and then account for the loop overhead

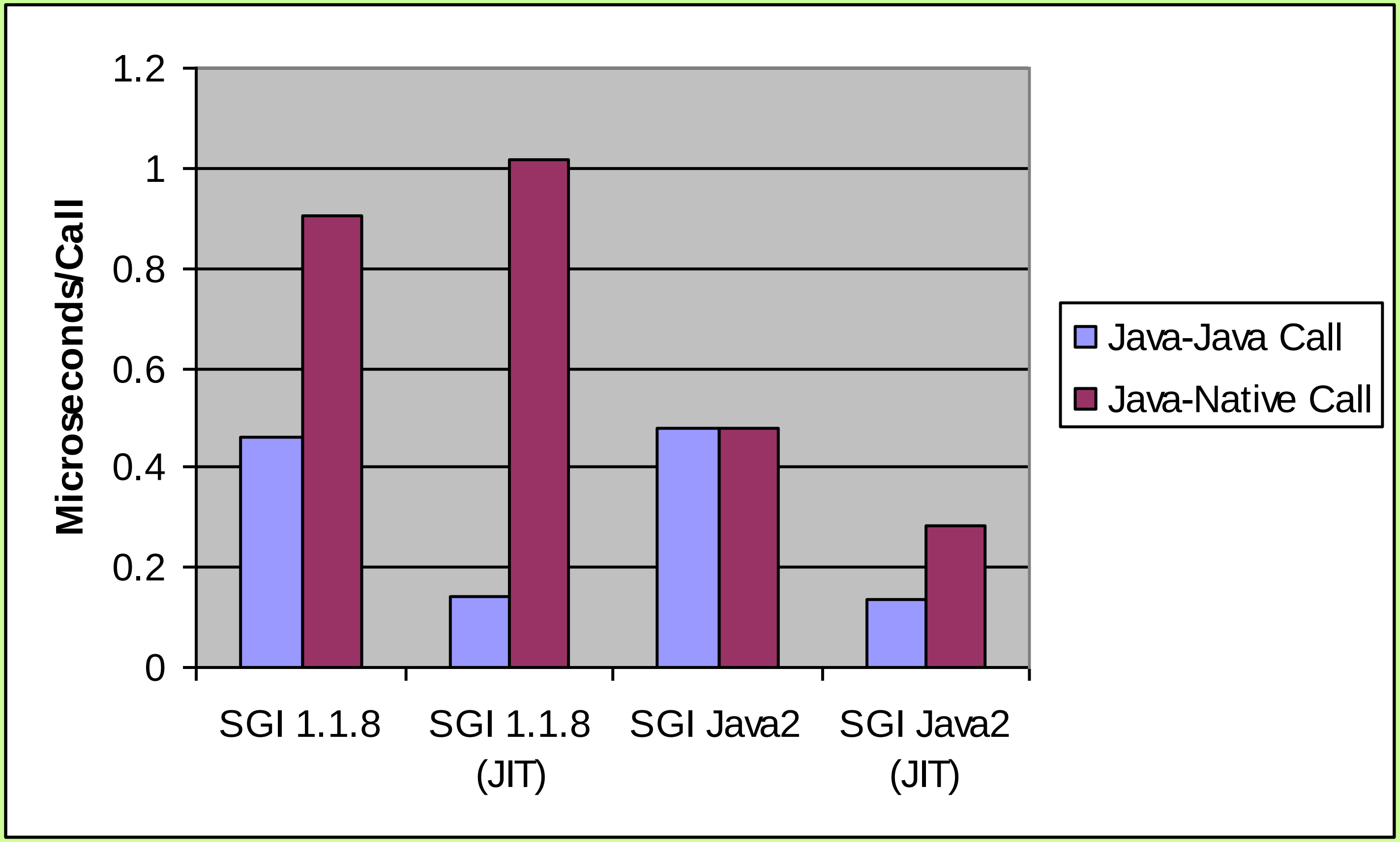
JNI Benchmarking

Platforms Tested

- **SGI 2000 (300 MHz R10000), IRIX 6.5.7**
  - **JDK 1.1.8, Java2 1.2.2**
- **Dell (350 MHz Pentium II), Red Hat Linux**
  - **IBM JDK 1.1.8, Blackdown JDK 1.1.7, Blackdown Java2-prev2, Sun Java2 1.3 Beta (including HotSpot Client VM)**
- **Sun UltraSPARC (143 MHz)**
  - **Sun JDK 1.2.2, HotSpot Server 1.0**

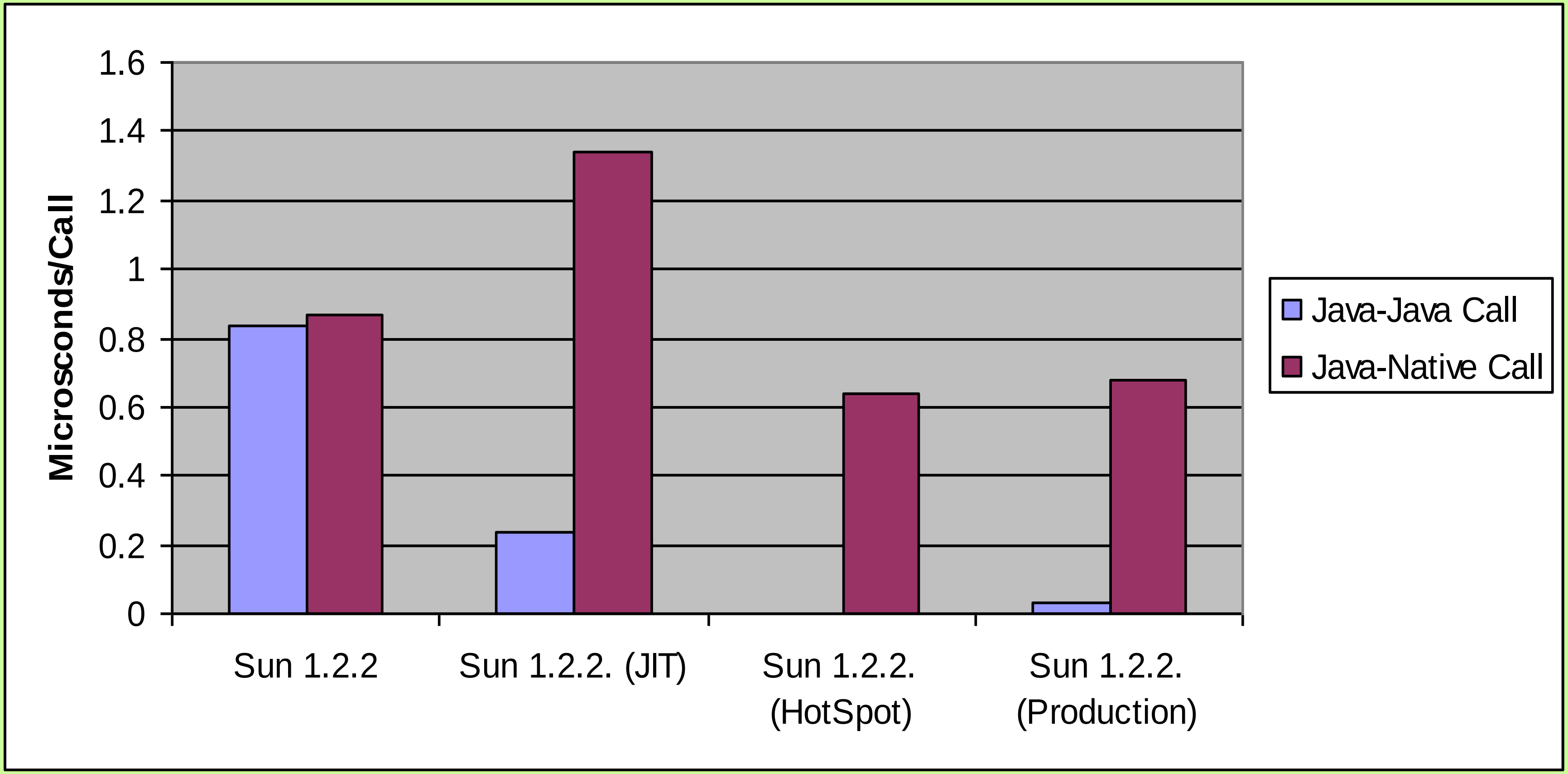
JNI Performance

Native Call Cost: SGI Results



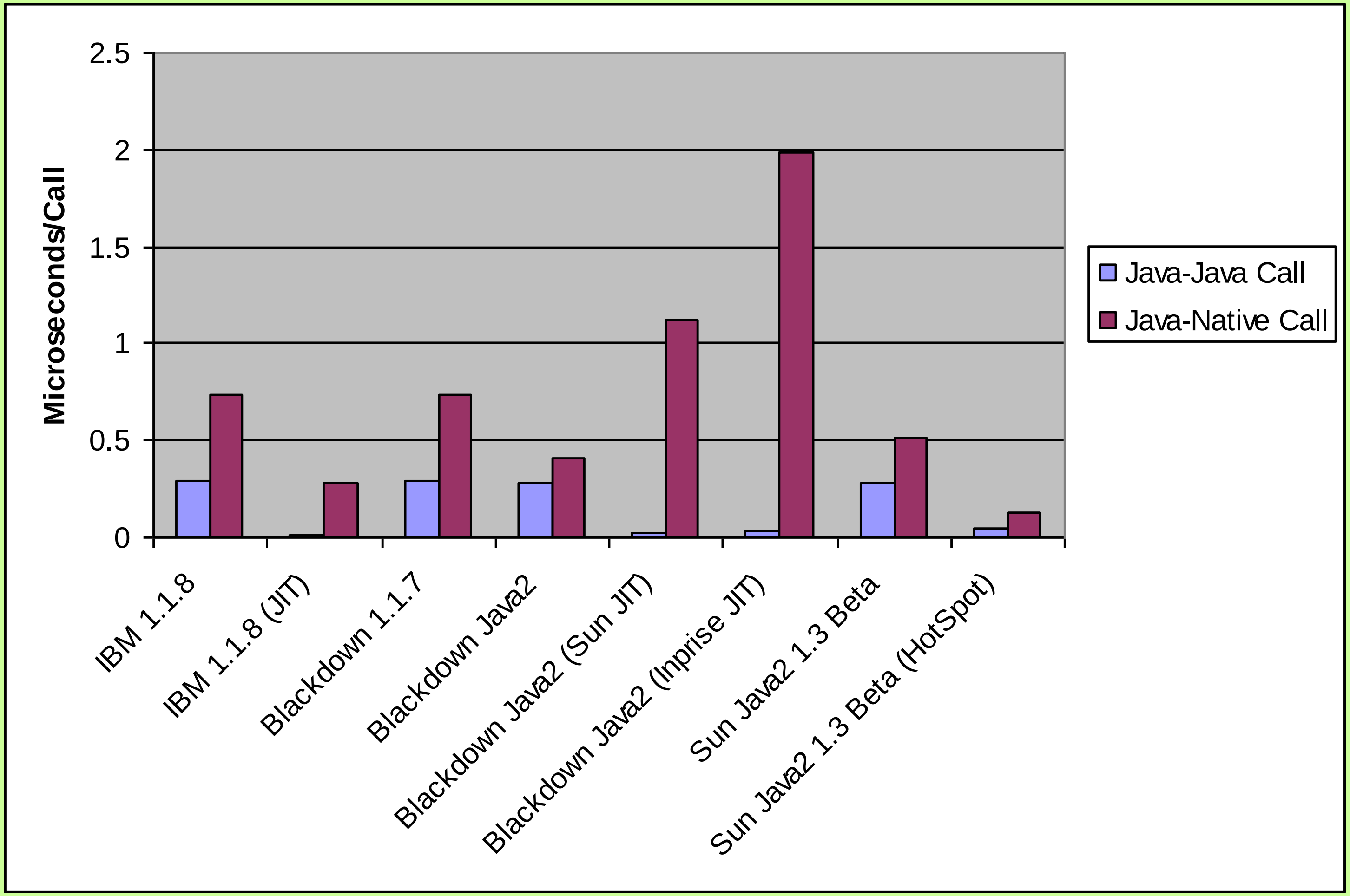
JNI Performance

Native Call Cost: Sun Results



JNI Performance

Native Call Cost: Linux Results



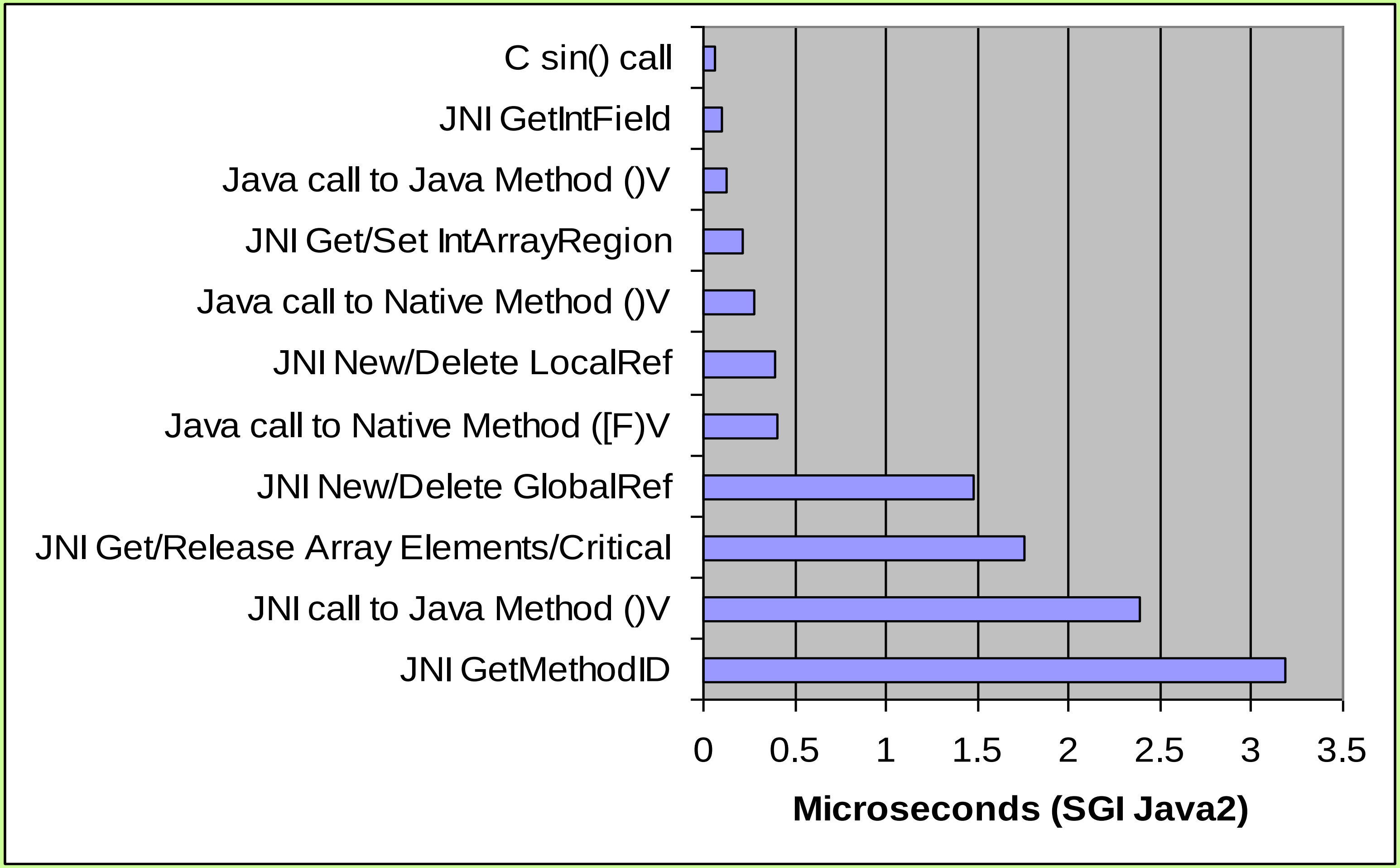
JNI Performance

Native Call Cost: Conclusions

- **Native method overhead is significant**
- **Overhead has decreased from JDK 1.1.x to Java2**
- **JIT compilation can make JNI calls…**
  - **significantly faster than under the interpreter (SGI Java2, IBM 1.1.8, HotSpot)**
  - **significantly slower (Blackdown Java2)**

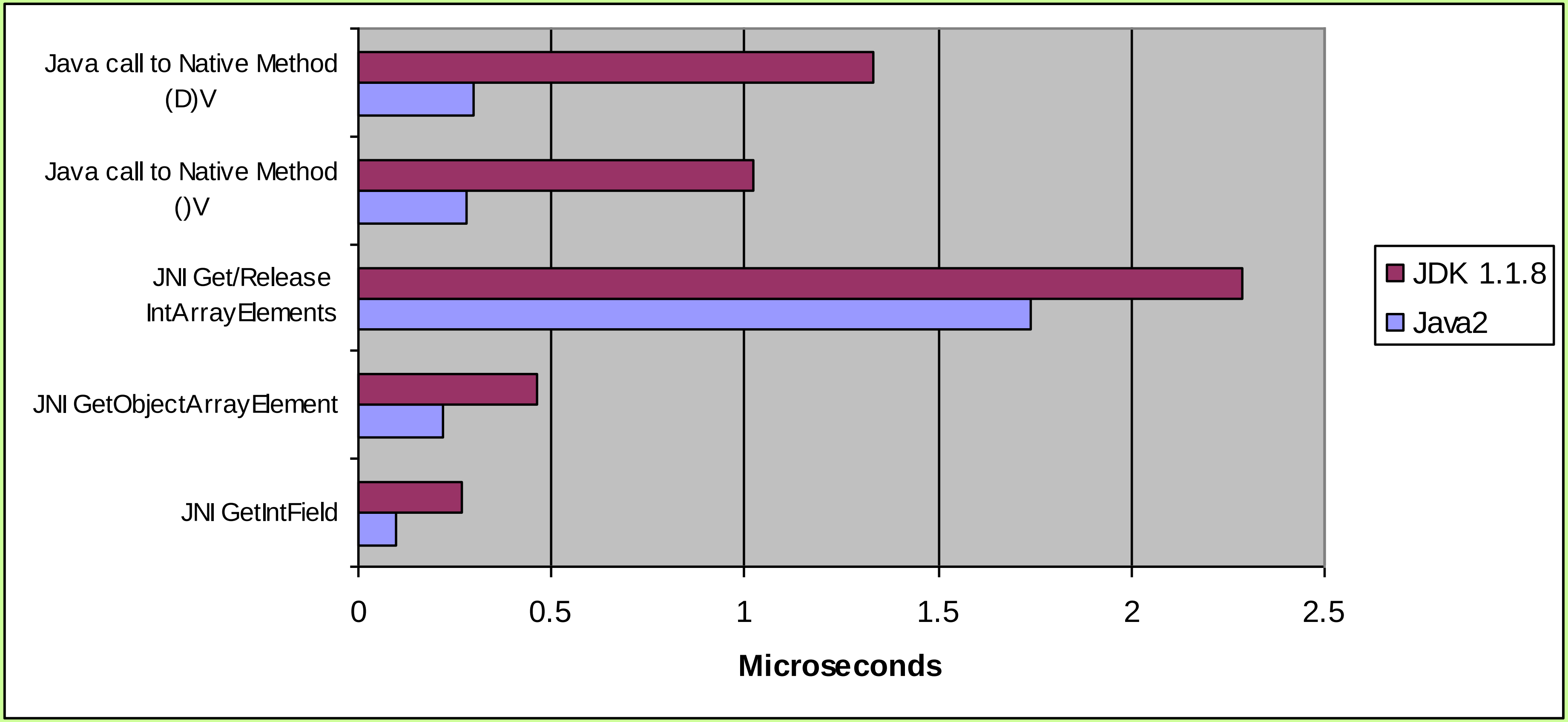
JNI Performance

JNI Performance Landscape



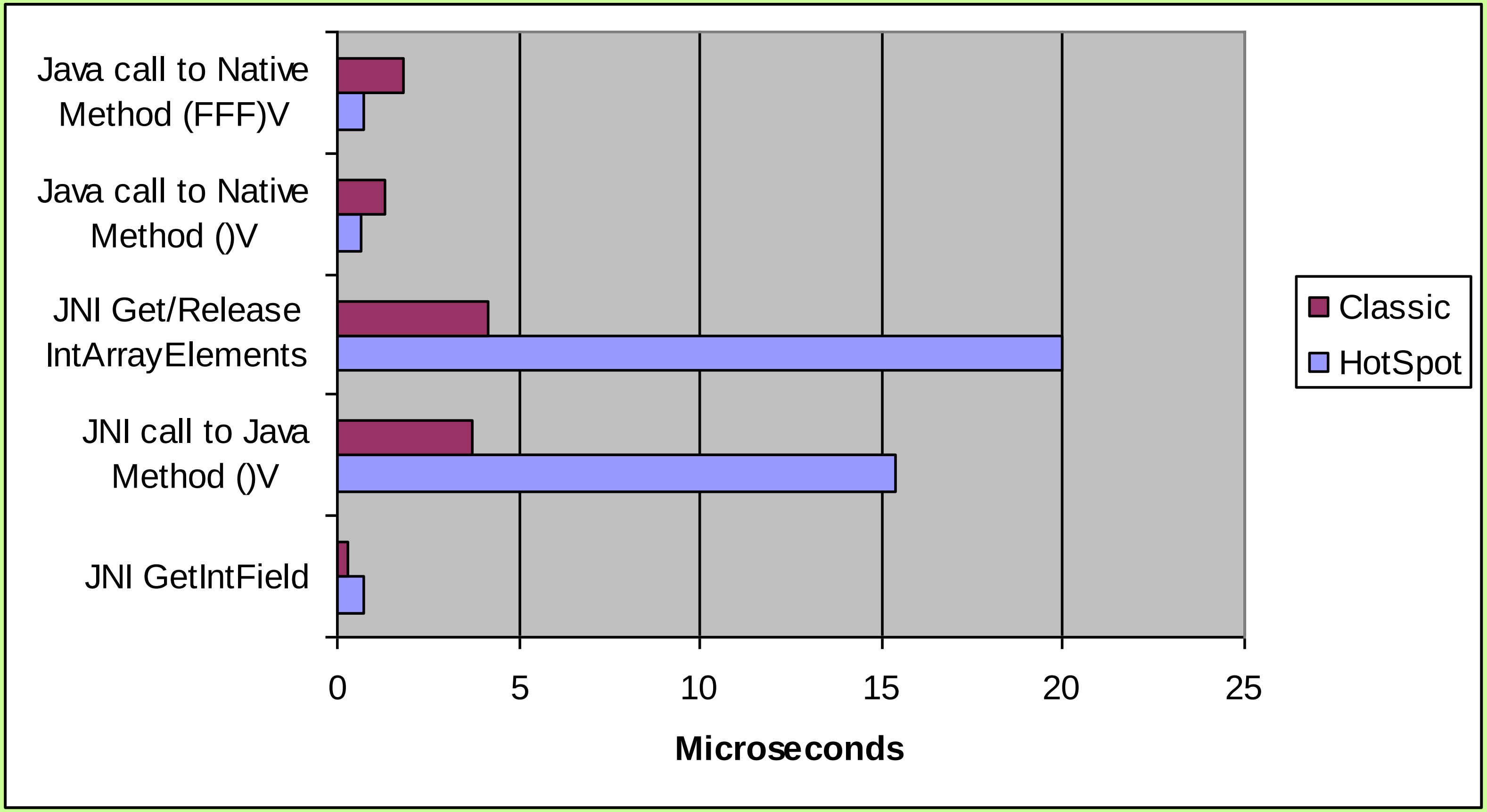
JNI Performance

JDK 1.1.8 v. Java2 (SGI)



JNI Performance

Classic v. HotSpot (Sun Java2)



JNI Performance

JNI Performance: Conclusions

- **Native calls and scalar parameter passing can be relatively efficient with JIT support**
- **Native access to Java arrays can be costly, even if the JVM uses pinning instead of copying**
- **JNI Global References are costly**
- **Calls back into Java are costly**
- **JNI implementations are mostly improving**

JNI Performance

Coding Recommendations

- **Choose the right granularity**
- **Partition application to use native code well**
  - **Good: computation**
  - **Bad: accesses to Java fields and methods**
  - **Necessary: low-level system access**
- **Caching: array data, field and method ID's**

Conclusions

- **Overhead of native method calls and JNI functionality is decreasing, but still deserves careful attention from developers**
- **Our IRIX/MIPS Fast JNI implementation has improved the performance of real-world applications such as Prowess and the Java OpenGL bindings**