



AIM Compilation Products and Technology

IBM Java Technology

Kevin Stoodley
IBM Distinguished Engineer
Toronto Laboratory
stoodley@ca.ibm.com

April 1, 2003

© 2002 IBM Corporation

Outline

- J9 VM Technology Overview
- Testarossa JIT Technology Overview
- Virtual Call-Site Optimizations
- Escape Analysis
- Open Forum

J9 Java Virtual Machine

- Sun IP-free, but Java 2 (1.3) compliant/"Java Powered" (J2ME) and J2SE
- Highly configurable class library implementation
- Multi-platform
- Currently shipping in IBM Websphere Studio Device Developer (WSDD) product v2.0
- Will ship in 2003 as part of IBM JDK 1.3.1 and 1.4.2
- Flexible and sophisticated technology

Sun IP-free, but compliant

- Developed jointly between IBM Ottawa Laboratory (previously Object Technology International--an IBM subsidiary) and IBM Toronto Laboratory
- J2ME™ configurations are free of Sun IP
 - ▶ **CLDC™** Connected Limited Device Configuration
 - ▶ **CDC™** Connected Device Configuration
 - ▶ **MIDP™** Mobile Information Device Profile (ex cell phone)
 - ▶ **PDAP** Personal Digital Assistant Profile
- J2SE™ configuration requires Sun IP dlls (AWT, Swing, etc)



Highly configurable class library implementation--Example configurations

- Choose the size/functionality tradeoff appropriate to the application
 - ▶ Extreme (jclXtr)--100K JVM + 100K jcl
 - io, lang, net, util subsets; no security support
 - ▶ Core (jclCore)--300K JVM + 300K jcl
 - io, lang, net, util, util.zip subsets; no security support
 - ▶ Max (jclMax)--500K JVM + 2M jcl
 - java.io java.lang java.lang.ref java.lang.reflect java.math
java.net java.security java.security.acl java.security.cert
java.security.interfaces java.security.spec java.text
java.util java.util.jar java.util.zip

Multi-platform

- PowerPC (AIX, Linux, OSE, QNX, WinCE, Mac OS X)
- IA32 (Windows, Linux, QNX, Neutrino)
- Sparc (Solaris)
- ARM (Linux, QNX, PocketPC)
- SH4 (WinCE, Itron, QNX)
- SH3 (WinCE)
- MIPS (PocketPC, QNX)
- 68K (PalmOS)
- 390 (Linux)
- Others--supported by VM Builder (Alpha, PA-RISC, X86-64, IA64, ...)

Flexible and sophisticated technology

- Port library
- Multi-VM
- Garbage Collection
- Sharing, eXecute In Place (XIP), Ahead Of Time (AOT) compile
- Debug, profile and runtime JVM interfaces
- Testarossa JIT

Port Library--Designed for Embedding and Retargetting

- Thin layer to isolate use of OS services and resources
 - ▶ memory, files, threads, sockets, locks, interrupt management
 - ▶ equally appropriate for embedding within middleware--J9 does not have to be "on top"
- Multiple independent port libraries can be simultaneously supported

Multi-VM

- Multiple JVMs in single address space
- No MMU support required
- Flexible deployment story
 - ▶ Multiple jcls on same target, eg. RT - non-RT
 - will share "stuff" if the same
 - ▶ Each JVM instance may have different port library
 - ▶ Each JVM instance may have different invocation parameters (GC, code cache, increments, etc)
- JVM-local storage
 - ▶ Used for globals
 - ▶ JNI code made re-entrant per VM (no statics)

Garbage Collection (GC)

- Four (accurate) GC implementations with Thread Local Heap (TLH)
 - ▶ Generational scavenger (write barriers req'd)
 - ▶ Mark & sweep - full stop and optional compact
 - ▶ 3-colour concurrent incremental (write barriers req'd)
 - Predictable, low latency, user callable (with bounds)
 - ▶ Large Heap/Parallel Prototype
- Highly configurable through the invocation API or command line
 - ▶ Nursery size
 - ▶ Tenure size + increment
 - ▶ Memory maximum
 - ▶ Remembered set size
 - ▶ TLH parameters

Sharing, eXecute In Place (XIP), Ahead Of Time (AOT) compile

- Classes divided into ROM and RAM sides
 - ▶ ROM side can be shared across JVM instances
- "JXE" == ROMable jar
 - ▶ Created with JXELink tool
 - ▶ Various flavours of static and feedback-directed optimizations available
 - ▶ Bytecode modified for reentrancy--no quick variants
- AOT compiled code
 - ▶ Leverages JIT infrastructure in a static mode
 - ▶ Code is fully compliant
 - ▶ Used for "instant on", core libraries
 - ▶ Brings compiled code performance to smaller platforms

Debug, profile and runtime JVM interfaces

- Debugging with JDWP
 - ▶ Uses JDI only, JVMDI is **not** supported
 - ▶ Remote proxy to allow for smaller targets
 - ▶ Added hot code replace to WP
- Profiling with JVMPI
- 1.2 Support
 - ▶ JNI 1.2 and class loaders

IBM's approach to J2SE

- Support the diversity of platforms - both **IBM** and **others** - focussing on ..
 - ▶ **Performance** and **scalability**
 - ▶ **Reliability**, **Availability**, **Serviceability**
 - ▶ Service and Support
 - ▶ **Write-Once-Run-Anywhere**
- Provide ..
 - ▶ .. a high-quality base for IBM's broad set of **middleware solutions**
 - ▶ .. competitive leadership for IBM's **hardware solutions**
 - ▶ .. industry **leadership** in web-services and enterprise Java solutions
- Principally deliver Java SDKs and JREs as **part of** IBM hardware and software solutions

Desktop and Server Technology Features

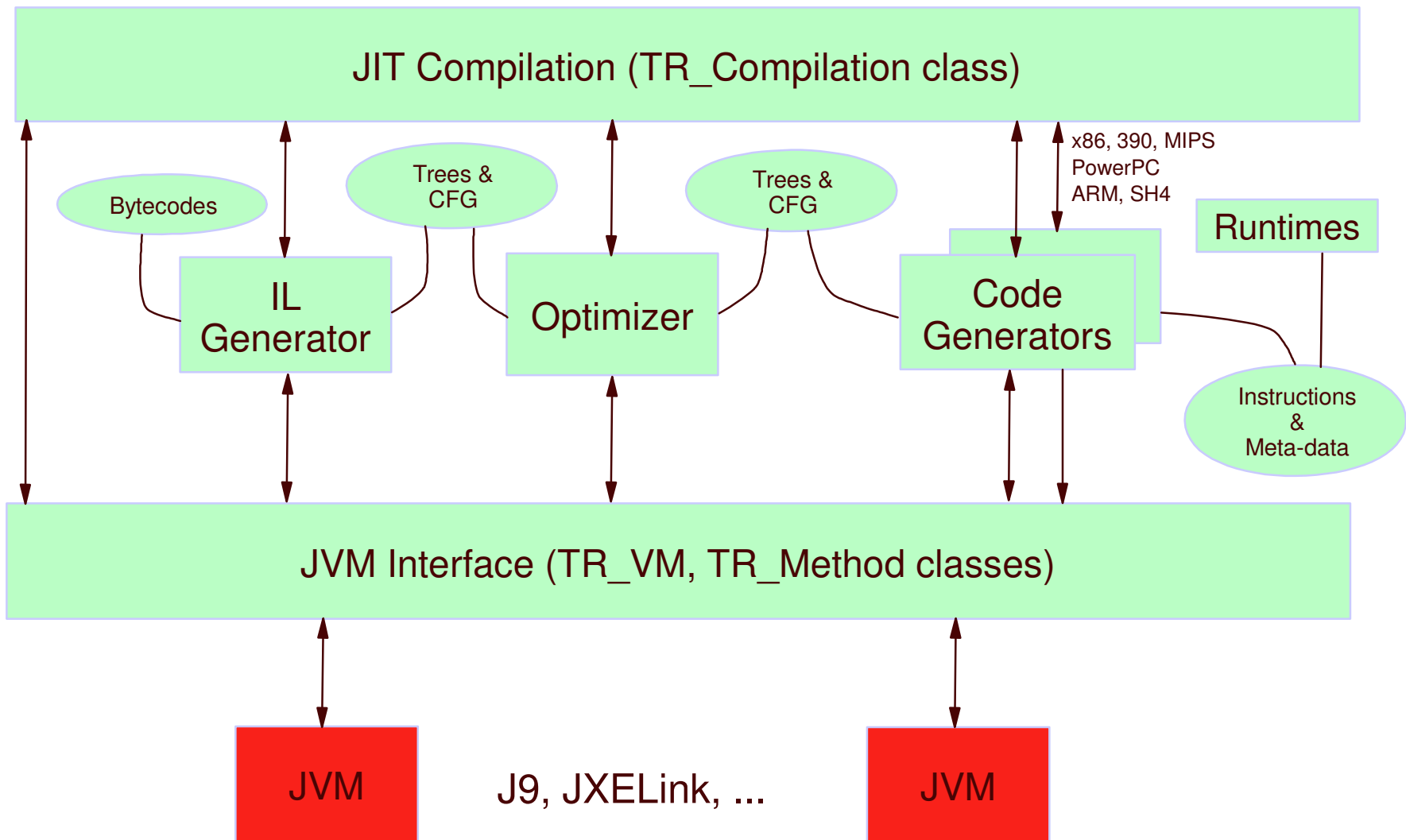
- High performance locking implementation "Tasuki Locks"
- type accurate GC Implementations with TLH
 - ▶ concurrent, incremental, parallel, large heap...
- Efficient object model, esp. 64 bit
- Fine-grained locking of VM data structures (scalability)
- Shared classes
- Profile driven dynamic recompilation (more later)

Testarossa JIT Technology

Testarossa Design Goals

- Clean separation of concerns along 3 major axes
 - ▶ JVM implementation (VM and OS services)
 - ▶ Java implementation (object model, runtime specializations, GC, threads, runtime interfaces)
 - ▶ Hardware targets
- Java centric design
- Portable and maintainable C++ implementation with some special purpose assembler
- Fast compile time
- Small footprint
- Configurable optimization framework
 - ▶ extremely complete suite of classical & Java-specific optimizations
- High performance code with deep platform exploitation
- Hot Code Replace (HCR) and Full-speed Debug (FSD)
- Complete solution: optimizing transformations fully operational in the presence of exception handling, security manager, stack trace, unresolved or volatile entities, etc
- Dynamic recompilation with profile directed optimizations
- Aggressive specialization and speculative optimizations

A peek under the hood



Diverse set of ISAs supported

- Fully supported targets
 - ▶ x86
 - ▶ 32-bit PowerPC (bi-endian)
 - ▶ ARM
 - ▶ SH4
- Under development
 - ▶ 64-bit PowerPC
 - ▶ 32-bit 390
 - ▶ X86-64
 - ▶ MIPS (bi-endian)
- In plan for 2003
 - ▶ 64-bit 390

Testarossa IL representation

- TR_IL is a list of expression trees
- Close to bytecode level of operators
- Side-effects are separated out into different trees
 - ▶ ensures correct order of execution
 - ▶ separation of operation and exception aliasing
- Cross-tree references allowed within a basic block
 - ▶ local common sub-expression elimination and register assignment rely on this capability
 - ▶ simplifies many other local analyses and transformations

IL Trees Example

```

■ int tstBar(int x) {
■   int tot=0;
■   int[] arr = new int[x];
■   for (int i=0;i<x;++i) {
■     arr[i] = i;
■     tot += arr[i];
■   }
■   return tot;
■ }

```

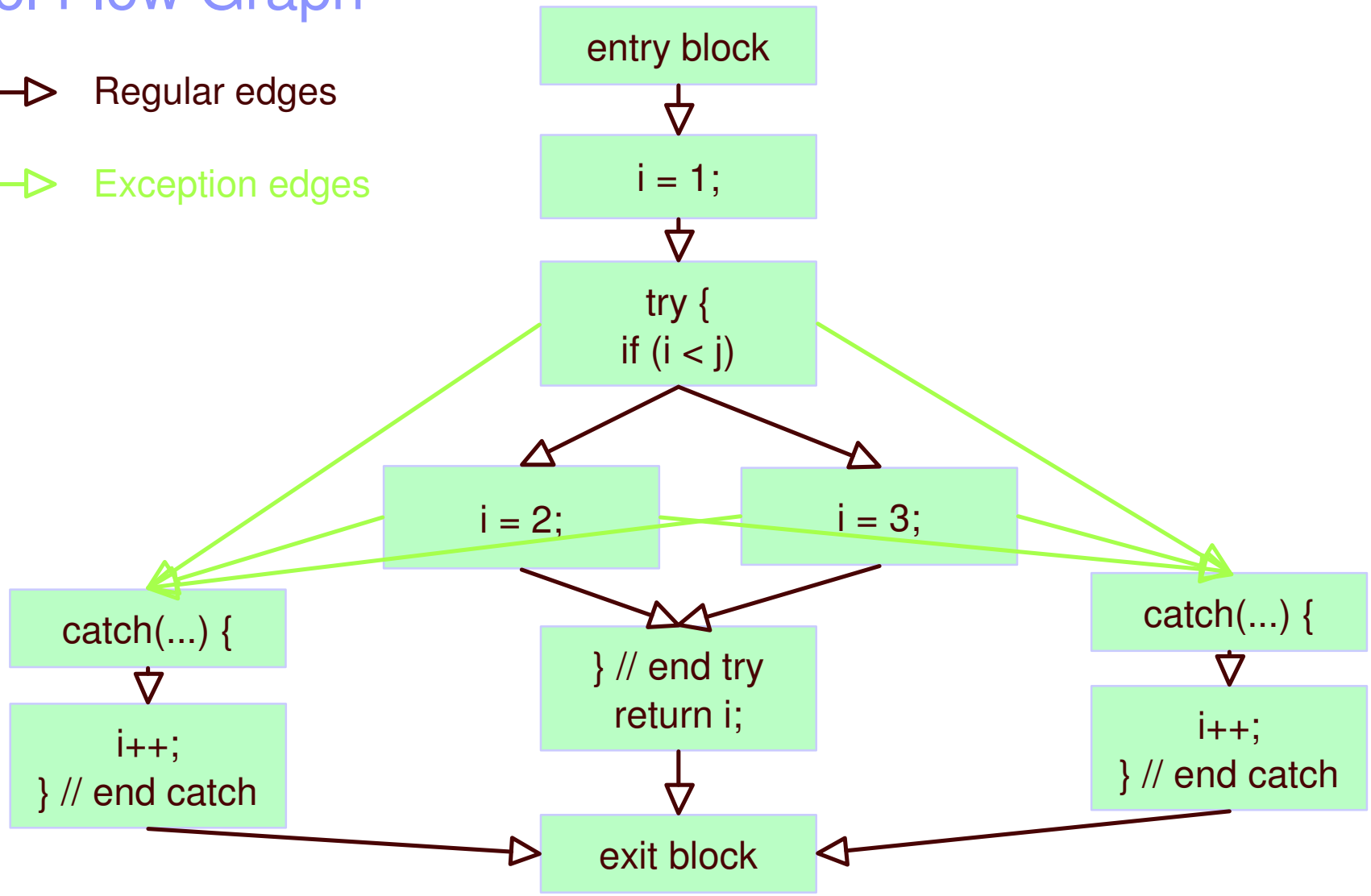
```

■ NULLCHK #12[0x224ce0] Shadow[0x224ccc]
■   iiload #13[0x224d68] Shadow[0x224d78]+12
■     aload #9[0x2245a4] Auto[0x22457c]
■   dbgFence
■   BNDCHK #14[0x224e1c] Shadow[0x224e08]
■     ishr
■       ==>iiload
■         iconst 2
■       iload #10[0x224680] Auto[0x224658]
■     dbgFence
■     iistore #15[0x224f58] Shadow[0x224f44]
■       aiadd
■         ==>aload
■         iadd
■           ishl
■             ==>iiload
■             ==>iconst 2
■           iconst 16
■         ==>iiload

```

Control Flow Graph

- Regular edges
- Exception edges



Complete suite of classical and Java optimizations

- Platform neutral optimizer performs IL-IL transformations
 - ▶ parameterized by platform specific code to handle different cpu capabilities (eg. # regs)
- Multiple optimization strategies for different code quality/compile time tradeoffs
 - ▶ used to compose optimizations into a collection of transformations
 - ▶ spend compile time where it makes biggest difference
- Extremely generalized solutions and infrastructure
 - ▶ Eg. Inliner capable of functioning effectively in presence of exception handling, security manager, stack trace, etc.

Fixed Optimization Strategy

- Method compiled only once.
- Compile triggered by invocation count.
 - ▶ Separate counts for methods with backward branches and methods without
- Same optimization level for all compilations
- Optimization Levels
 - ▶ Level 0: NoOpt
 - Tree Simplification
 - ▶ Level 1: LowOpt
 - Local Optimizations
 - ▶ Level 2: BestAvailOpt
 - Local Optimizations
 - Most Global Optimizations
 - Some short cuts to reduce compile time
 - ▶ Level 3: HighOpt
 - Full Global Optimizations
 - ▶ Level 4: Ahead-Of-Time
 - Full Global Optimizations with multiple passes.

Details: NoOpt

- Tree Simplification
- Inline and fast-path some method calls
 - ▶ getClass, currentThread, currentTimeMillis
- Inline and fast-path some helpers
 - ▶ instanceof, checkcast, monitor enter, monitor exit
- Fast path JNI calls (Direct2JNI)
- Local Register Allocation

Tree Simplification

- A local optimization, operates on a block at a time
- Constant folding
- Strength reduction
- Algebraic simplification
- Tree normalization (move constant child to RHS)
- Constant switch folding
- Remove branch to following block
- Remove constant conditional branch
- Coalesce adjacent blocks

Details: LowOpt

All of NoOpt, plus

- Inlining
 - ▶ guarded inlining of virtuals
- CFG Simplification
- Simple loop unrolling
- Return block hoisting
- Local re-ordering
- Block re-ordering
- Basic block extension
- Local constant/type propagation
- Local Async Check Removal
- Local Common Subexpression Elimination
- Catch block removal
- Local dead store elimination
- Isolated store elimination
- Simple global register allocation
- Local live variables for GC

Details: BestAvailOpt

All of LowOpt, plus

- More Inlining
- Virtual guard tail splitting
- Global constant/type propagation
- Late Inlining
- Loop canonicalization
- Loop versioning
- Loop unrolling
- Global copy propagation
- Global dead store elimination
- Global Async Check Removal
- Global live variables for GC
- Fast-path ArrayCopy
- IA32 Floating Point Precision Conversion

Some optimizations are repeated for loopy methods

Details: HighOpt

All of BestAvailOpt,
plus

- Iterative Global constant/type propagation
- Aggressive thresholds for Inlining
- Better Global Register Allocation
- Ahead-Of-Time Strategy:
 - ▶ Similar to HighOpt
 - ▶ Compile time is assumed to be less of an issue.

Some optimizations are repeated
conditionally

Recompilation Strategy

- Compile methods cheaply initially
- Recompile hot methods more aggressively later
- Counting Recompilation
 - ▶ First compilation triggered after a said number of invocations
 - ▶ Compiled code decrements counter at yield points (method entry, and async checks)
 - ▶ Code at method entry decides if recompilation is to be performed
- Sampling Recompilation
 - ▶ Regularly (~10ms) interrupt application threads
 - ▶ Analyze the context of the application thread to find method being executed
 - ▶ Previous compilation sets up counters and hotness for next compilation
 - ▶ A counter is decremented each time a method is sampled
 - ▶ Recompilation is triggered when
 - The counter goes to zero, or
 - Every 3rd time the method is sampled, check if it is particularly hot
 - If this method has shown up 3 or more times in the last 20 global samples, it is scorching hot.
 - If this method has shown up 3 or more times in the last 300 global samples, it is hot.
 - ▶ Still Evolving

Recompilation Levels

- Cold Method Strategy: early compile for a loopy method
 - ▶ Local optimizations
- Warm Method Strategy: first compile of an initially loopless method, triggered after an initial number of invokations
 - ▶ Most Global Optimizations
 - ▶ Some short cuts to reduce compile time
- Hot Methods Strategy: A loopy method that has been compiled before, and is discovered to be hot
 - ▶ Full Global Optimizations
- Profiling Strategy: en route to Scorching Strategy
 - ▶ Similar to Hot Method Strategy
 - ▶ Code instrumented for profiling
- Scorching Strategy: Last compile for a method
 - ▶ Aggressive full global optimizations, utilizing profiling information.

Profile directed sampling recompilation

- Sampling thread drives compilation based on perceived hotness
- Initial compilation is no or low opt
- Hot methods are recompiled at increasingly higher optimization levels
- "Scorching hot" methods
 - ▶ recompiled with profiling instrumentation
 - edge counts with inferences, value profiling, virtual call sites, etc
 - ▶ run long enough to gather representative data
 - ▶ recompiled at highest opt level directed by profile information
 - basic block scheduling
 - inlining
 - loop versioning and unrolling
 - devirtualization
 - aggressive replication
 - speculative opts

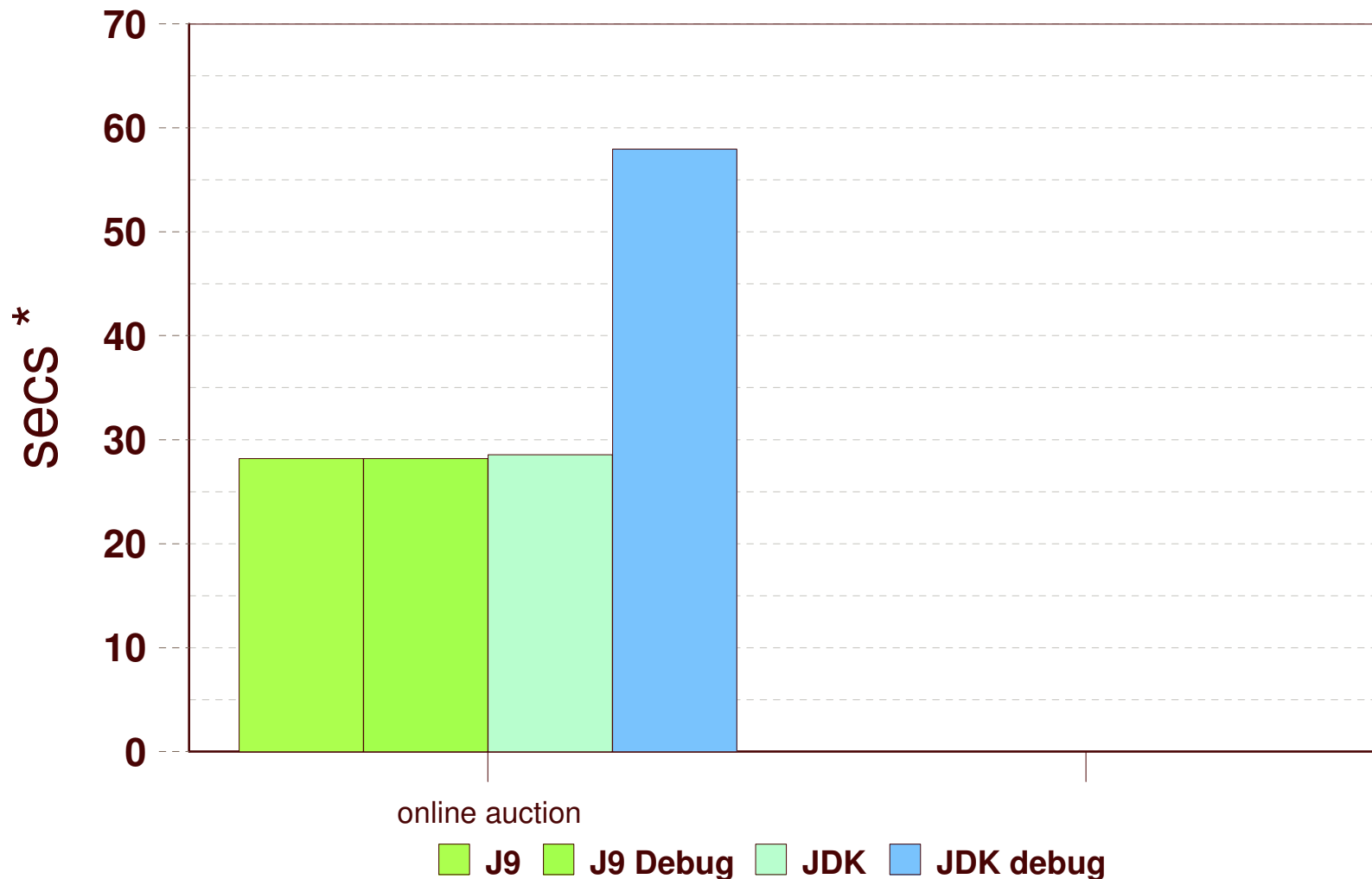
Recompilation infrastructure is basis for

- Aggressive speculative optimizations
 - ▶ pre-existence based devirtualization and inlining
 - ▶ other class hierarchy based optimizations
 - ▶ single threaded optimizations
- Hot Code Replace (HCR)
 - ▶ Fix/Enhance code while running and without restarting
- Advanced problem determination and performance monitoring features
- Phase change adaptations

Full Speed Debug (FSD)

- JIT is active while running program under debugger
 - Most (fairly hot) application code being run is compiled
 - Current level of optimization is quite low
 - ▶ initial release, no technical barrier precluding higher opt levels
 - Debug events may cause decompilation using on-stack replacement or may set a limit to preclude compiling that method
 - ▶ setting a breakpoint
 - ▶ single step
 - Supports Hot Code Replace (HCR)
 - Supports Data Change Breakpoints
-
- Gives combination of rich interpreted debug environment with performance of a dynamically compiled runtime

Full Speed Debug (FSD): WSAD online auction example



HW: P4 2.0GHz machine with 1GB of RAM under WSAD5.0.

* time taken from the "Operation in progress" dialogue disappearing to the Auction web page coming up with the "Done" status.

Virtual Call-Site Optimizations Overview

- Motivation
- Guarded Devirtualization
- Virtual Guard NOPing
- Type Propagation
- Pre-Existence
- Class-Lookahead
- Miscellaneous
 - ▶ Escape Analysis
 - ▶ Splitting
- Future Work and Ideas

Motivation: Method Inlining

- Important for Java JIT Compilers
 - ▶ Java methods are typically small
 - ▶ Interprocedural analysis usually too expensive
- Relatively simple for non virtual calls
 - ▶ Dispatches on methods in final classes
 - ▶ Constructors, private methods, and super()
 - ▶ Static methods
- Tricky for virtual calls
- Dynamic Class Loading
 - ▶ Allows demand driven loading of classes
 - ▶ The target of a virtual call may change at run time
 - ▶ Not all possible targets are known at compile time.
- Devirtualization
 - ▶ Guarded Devirtualization
 - ▶ Direct Devirtualization

```
class A {  
    int foo() { return 1; }  
}
```

```
class Other {  
    int x;  
    int bar(A a) {  
        x = 0;  
        x += a.foo();  
        return x;  
    }  
}
```

Guarded Devirtualization

Generally, all needed classes are already loaded before a method is invoked

- ▶ Make assumptions about the current state of the class hierarchy
- ▶ While providing mechanism for correctness once the assumption is invalidated

```
int bar(A a) {  
    x = 0;  
    x = x + a.foo();  
    return x;  
}
```

Guarded Devirtualization

Generally, all needed classes are already loaded before a method is invoked

- ▶ Make assumptions about the current state of the class hierarchy
- ▶ While providing mechanism for correctness once the assumption is invalidated

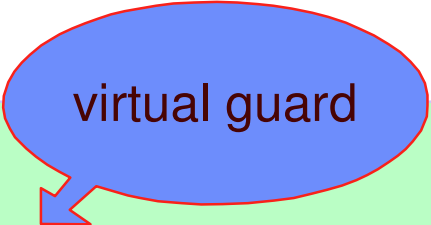
Inline currently monomorphic sites, surrounded by an if-statement (guard)

Runtime test to ensure correct dispatch

- ▶ need to pay the cost of performing the test for each invoke
- ▶ will pay the extra penalty of the test/jump if the target changes at runtime because of dynamic class loading
- ▶ Can reduce this cost by using NOPing.

```
int bar(A a) {
  x = 0;
  x = x + a.foo();
  return x;
}
```

```
int bar(A a) {
  x = 0;
  if (A.foo is still the correct target) {
    t = 1;
  } else {
    t = a.foo()
  }
  x = x + t;
  return x;
}
```



Virtual Guards

- Conventional Types of Virtual Guards
 - ▶ Class Test
 - At runtime, check the type of the receiver object
 - Relatively low runtime cost
 - Implementable with a single dereference operation
 - Can guard multiple class sites dispatching different methods on the same object
 - Have good type information in the protected region of code
 - ▶ Method Test
 - At runtime, check the address of the target method
 - Higher runtime cost
 - At least 2 dereference operations
 - Can cover call sites where receiver type is polymorphic, but target is monomorphic
- Testarossa: 'Not-Overridden' Guard
 - ▶ Default virtual guard used by Testarossa
 - ▶ Keep a bit on each method indicating if it has been overridden
 - ▶ Does not need Type Analysis or Class Hierarchy Analysis (CHA)
 - Devirtualization can be done even at "NoOpt" optimization level
 - ▶ Low runtime cost, single dereference

Virtual Guards

- Several Kinds of Guards used by Testarossa
 - ▶ Not-Overridden-Guard
 - ▶ VFT Guard (Class Test)
 - ▶ Not-Overridden-Guard for InterfaceCall
 - checks the is-overridden bit on the method in the implementing class
 - Used for interface classes that currently have exactly one implementation
 - ▶ VFT Guard for Interface Call
 - checks the class pointer against that of the current unique concrete implementation
 - ▶ VFT Guard for Profiled Call
 - used for dispatches on methods that are already overridden
 - profile the class-object of the call
 - inline the common case with a class test
 - ▶ Method Guard for methods not overridden in a particular part of the hierarchy
 - ▶ VFT Guard for methods not overridden in a particular part of hierarchy

Virtual Guard NOPing

Guard tests incur runtime penalty

```
test  [is-overridden-bit], 4
jnz   LVCALL
; inlined code
...

LVCALL:
```

Virtual Guard NOPing

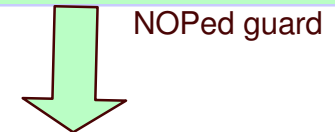
Guard tests incur runtime penalty

Emit a NOP instead of the comparison

- ▶ create RuntimeAssumption
 - on the overridden status of the method (not-overridden/method guards)
 - or on the Currently-Final status of a class (vft-guards)

```
test  [is-overridden-bit], 4
jnz   LVCALL
; inlined code
...
```

LVCALL:



```
nop
; inlined code
...
```

LVCALL:

Virtual Guard NOPing

Guard tests incur runtime penalty

Emit a NOP instead of the comparison

- ▶ create RuntimeAssumption
 - on the overridden status of the method (not-overridden/method guards)
 - or on the Currently-Final status of a class (vft-guards)

When the RuntimeAssumption is violated

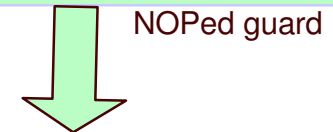
- ▶ VM informs JIT of every class load and method-override event
- ▶ change method code to always execute the virtual dispatch

NOPing Issues:

- ▶ Atomicity
- ▶ Cache Coherence
- ▶ NOPs are not always free

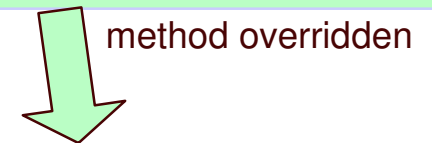
```
test  [is-overridden-bit], 4
jnz   LVCALL
; inlined code
...

LVCALL:
```



```
nop
; inlined code
...

LVCALL:
```



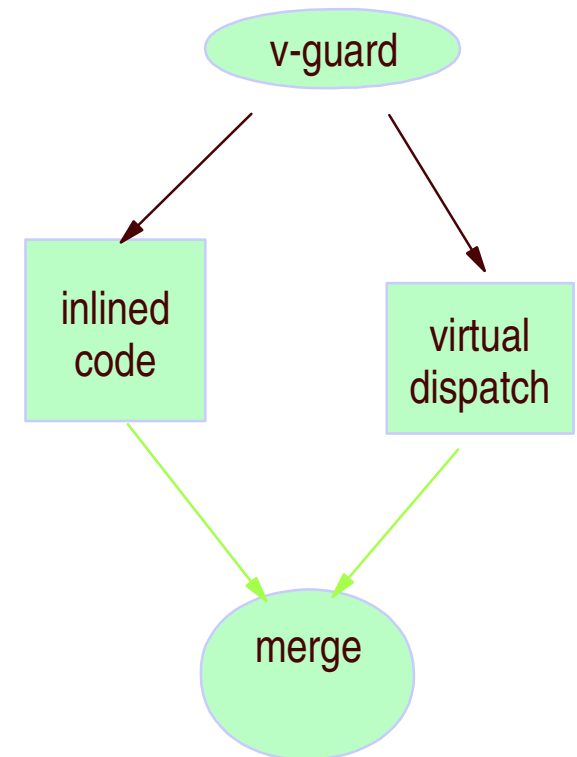
```
jmp   LVCALL
; ...
...

LVCALL:
```

Effects of Guarded Devirtualization

- Virtual Calls are kill points for most JIT optimizations
- Optimizations have to be pessimistic about the side effects
 - ▶ the method being called may not even exist yet
- Aliasing
 - ▶ A call may write to all global and indirect symbols (fields)
- The call node still exists in the control flow graph
- Optimizations still account for the presence of the call

```
int bar(A a) {  
    x = 0;  
    if (A.foo not overridden) {  
        t = 1;  
    } else {  
        t = a.foo()  
    }  
    x = x + t;  
    return x;  
}
```



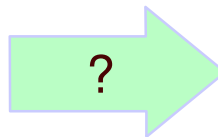
Type Propagation

- Type Propagation can directly devirtualize when the type of the receiver is more accurately known.
- Type Information originates from
 - ▶ Method and Field Signatures
 - ▶ Freshly new'd allocations
 - ▶ instanceof / checkcast
- Performs unguarded devirtualization
 - ▶ Backup paths are not generated
 - ▶ Kills backup paths that already exist
- May decide to change to type of the guard to better exploit type information in the inlined code.
 - ▶ Mark the receiver as of fixed type in a region guarded by a VFT guard
 - ▶ Try to inline more in a VFT guarded region

Pre-Existence

- A.foo() has only one current target
- Idea: Specialize the method bar
 - ▶ directly devirtualize all currently monomorphic invokes
 - ▶ recompile the specialized method when A gets extended.
- Needs on-stack-replacement, in general
- Special cases in which this can be done without needing on stack replacement

```
class Other {  
  int x;  
  int bar(A a) {  
    x = 0;  
    x = x + a.foo();  
    return x;  
  }  
}
```



```
int bar(A a) {  
  return x = 1;  
}
```

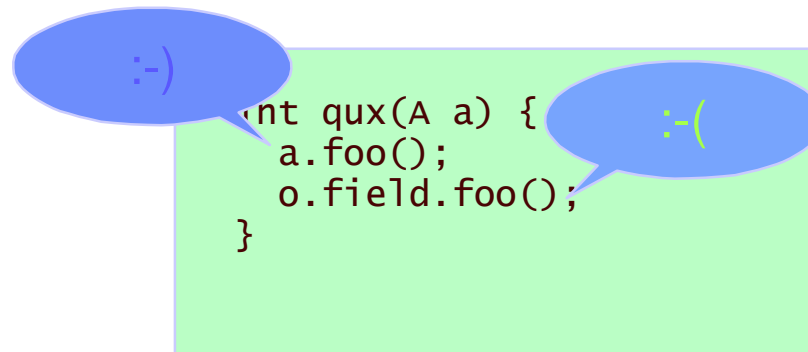
Invariant Argument Pre-Existence

- Arguments to a method must be allocated before the invoke
 - ▶ i.e. they "Pre-Exist"
- Autos and Params are thread-local
- If a class is loaded during the execution of qux
 - ▶ Argument 'a' cannot become anything other than type A, if it is of type A at method entry.
- Recompile qux on next invocation.

```
int qux(A a) {  
    a.foo();  
    o.field.foo();  
}
```

Invariant Argument Pre-Existence

- Arguments to a method must be allocated before the invoke
 - ▶ i.e. they "Pre-Exist"
- Autos and Params are thread-local
- If a class is loaded during the execution of qux
 - ▶ Argument 'a' cannot become anything other than type A, if it is of type A at method entry.
- Recompile qux on next invocation.
- Cannot, generally, devirtualize invoke on o.field
 - ▶ Need more powerful opts: eg. Single Threaded Opts.

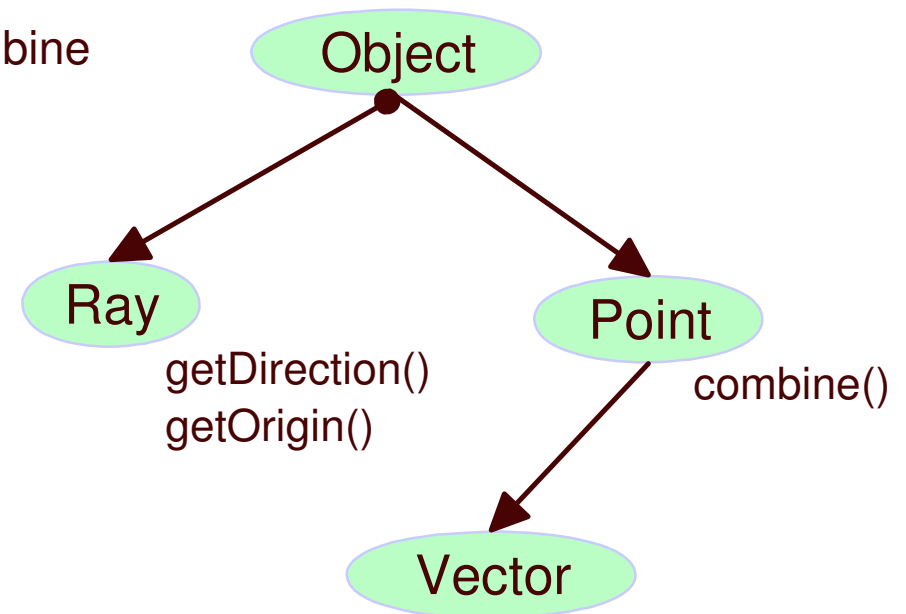


```
int qux(A a) {  
    a.foo();  
    o.field.foo();  
}
```


Invariant Argument Pre-Existence (Example)

```
OctNode Intersect(Ray ray, Point intersect, float Threshold) {  
    ...  
    intersect.Combine(ray.GetOrigin(), ray.GetDirection(), 1.0f,  
t);  
    ...  
}
```

- Ray is currently final
- Point has been extended but Combine is not overridden.
- Can directly devirtualize all calls listed above
- Need to recompile Intersect if
 - ▶ Ray is extended
 - ▶ Combine is overridden



Class Lookahead

- Peek into the classes of hot methods
- Prove that particular fields are always initialized to the same types
- Analyze all methods in the class to see how the field is accessed
 - ▶ Find private fields that are always initialized and are immutable in the rest of the class
 - ▶ Also analyze inner classes if they exist
- Can directly devirtualize invokes on these fields
 - ▶ Also, remove checkcasts, boundchecks, nullchecks, etc.

Miscellaneous

- Escape Analysis
 - ▶ Remove monitor enter/monitor exit from invokes on objects that do not escape
 - ▶ .. or haven't escaped yet.
 - ▶ Ask Desynchronizing Inliner to inline the call if possible
 - ▶ Remove existing locks, if already inlined
- Block Splitting
 - ▶ Works with profiling statistics gathered
 - ▶ Split control flow at merge points
 - ▶ Better type information results in more devirtualization
- Tail Splitting
 - ▶ Perform tail duplication of control flow to avoid merge points
 - ▶ Optimize common code sequences of the form:
 - `OctFaces[3].GetVert().GetDirection().GetX()`

Escape Analysis Overview

- Purpose
- Overview of the analysis
- Some details
- Results
- Future work

Purpose

- Find allocations in this method that
 - ▶ Cannot escape from this method to other threads, and
 - ▶ Cannot escape from this method to its caller
- These are "local allocations", with the same lifetime as the method.

Purpose

- We may be able to put local allocations on the stack instead of the heap
- We can de-synchronize calls to the local allocation object's methods

Finding local allocations

- Look for TR_new, TR_newarray, TR_anewarray
 - ▶ Discard new of class that implements "Runnable" (or that may, i.e unresolved class)
- Use def/use and value number info to collect all value numbers that can possibly represent the use of a candidate
- Look for each use of each value number to see if the candidate can escape the method
- Sniff into called methods as much as possible, mapping arguments to parameters

Handling local allocations

- They are all candidates for de-synchronization
- They are NOT candidates for stack allocation if
 - ▶ The object is too big
 - ▶ Allocation is inside a loop
 - ▶ JVMPI is enabled
 - ▶ Class is not completely initialized
 - ▶ Class is finalizable, abstract or interface
 - ▶ Array with variable bound
 - ▶ Array of doubles or longs (alignment problems)

Allocating on the stack

- If there is code that needs the object to keep its original shape, allocate space for the complete object on the stack, including header (contiguous allocation)
- Otherwise allocate each referenced field as if it were a separate local and don't allocate a header (non-contiguous allocation)
- Initialize fields and header (if needed) at the original allocation point

When must an allocation be contiguous?

- A use can be reached from some other def

```
C o;  
if (x)  
    o = new C;  
else  
    o = p;  
y = o.f;
```

- The allocation is passed as a call argument

```
o = new C;  
o.foo();
```

When must an allocation be contiguous?

- An array index is non-constant

```
C o[] = new C[10];  
o[5] = c1; // This reference is
```

OK

```
o[i] = c1; // but this one isn't
```

- Access to an unresolved field (the access must stay an indirect load or store)

```
o = new C;  
o.f = 2; // f is unresolved
```

When must an allocation be contiguous?

- Access to a slot within the header
- Checkcast or instanceof to an unresolved class (will be a call to a helper)
- Also other calls to helpers, e.g. arraycopy

When must an allocation be contiguous?

- Sometimes we can allocate the fields as separate locals and allocate a separate `java/lang/Object` representing the header:

```
o = new C;  
if (x)  
    o1 = o;  
else  
    o1 = p;  
y = o.f;  
if (o1 == o)  
    ...
```

- (also used for failing checkcast)

After the dust settles

- Fix up all references to candidates that are to be local allocations
- Fix up the allocation nodes themselves, add initializations
- Perform useful inlining

Set up contiguous stack allocation

- For each field or array element reference
 - ▶ If the local allocation is the only def to reach the access, remove write barrier store
- Replace the allocation node with a loadaddr for its local and insert explicit header initialization
- Insert explicit zero initialization, using existing field symbol references where possible

Set up non-contiguous stack allocation

- For each field or array element reference
 - ▶ Find or create the local that is to represent it
 - ▶ Change the load or store into a direct load or store of the local
 - ▶ Convert an `ArrayStoreCHK` on a candidate array into a (special) checkcast, if needed
- Replace the allocation node with explicit zero initialization of the locals representing the fields
 - ▶ Note that only referenced fields are turned into locals, unreferenced fields are ignored
- If necessary, create and initialize a local object containing just the object header

De-synchronizing

- Desynchronizing calls
 - ▶ Currently can only be done by inlining the call
 - ▶ Special inliner code does not generate monent and monexit for the inlined method

- Monitor enter and monitor exit
 - ▶ Mark the monitor enter and exit as being for a local object
 - ▶ Leave it to Redundant Monitor Elimination to get rid of them if it can

Multiple passes

- If inlining will cause a contiguous allocation to become a non-contiguous allocation
 - ▶ Leave the allocation node alone
 - ▶ Inline the appropriate methods
 - ▶ Do another pass of Escape Analysis once this one is completed

- Maximum number of passes depends on method hotness

Splitting local allocations

- Sometimes it is advantageous to split an allocation so that on one path it is local

```
o = new C;  
if (virtual-guard-for-x)  
{  
    ... // inlined x.foo(o);  
}  
else  
    x.foo(o);
```

becomes

```
if (virtual-guard-for-x)  
{  
    o = new C;  
    ... // inlined x.foo(o);  
}  
else  
{  
    o = new C;  
    x.foo(o);  
}
```

Special handling for exception objects

- If "printStackTrace" is never called, then ...
- No need to call "fillInStackTrace"

Future work

- Handle monitor enter and exit properly for local non-contiguous and contiguous objects
- Allow indirect store of one candidate into another candidate's field or array element
- Handle allocations inside loops
- Track values returned from sniffed methods
- Desynchronize by method versioning
- Don't map contiguous local allocations into the GC stack map

Open Forum