Multi-tasking Java virtual machines

Laurent Daynès
Sun Microsystems Laboratories
laurent.daynes@sun.com

Sun microsystems

Outline

- Why a multi-tasking virtual machine
- Programming Model
- Key Implementation Aspects
  - > Making the JVM multi-task
  - > Sharing the runtime representation of classes
  - > GC performance isolation
- Related Efforts

Why a Multi-Tasking VM ?

Make the Java Platform a portable, self-sufficient execution environment that runs multiple programs efficiently

- Leverage Java's safety properties to implement software-based protection
  - Promise faster IPC, cheaper cross-applications context-switches
- Increase OS independence
  - Provide platform-independent program life-cycle management
  - Self-sufficient on bare-metal VM port / simple OS
- Fast startup, leaner footprint
- Amortization of managed runtime costs
  - Transformation of architecture-neutral to main-memory platform-dependent format, dynamic compilation, verification, etc.

Why is class loader-based isolation insufficient ?

- Strict isolation required
  - > Confinement of errors to one application
  - > Simplifies security, resource accounting, application termination
  - > Want legacy applications to run as is and without interference with other applications
- Code sharing should be transparent
  - > Developers shouldn't be concerned with details related to sharing class footprint across application boundaries

Multi-tasking aware Heap Management

- Performance isolation
  - GC activity of one task should not impact performance of other tasks
    - GC should not prevent progress of other tasks
    - allocation should not increase pressure on other heap's tasks
- Instantaneous, GC-less reclamation of heap space on task termination
- Independent setting of heap parameters
  - heap size, tuning parameters, algorithm
- Can't do any of the above with a shared heap
  - and therefore, with class loader based isolation...

Isolate

- Abstraction for a fully isolated execution container
  - Analogous to OS process
  - Asynchronous termination, Resource reclamation
- No implementation details exposed
  - Enable various implementation strategies
    - One process/JVM pair per-isolate, multi-tasking JVM *à la* MVM
- Isolation identical to executing with a standalone JVM
  - Superior to class loader-based isolation
- Ability to monitor and control the execution container

Isolation API

- Embodied by JSR 121
  - > Does not mandate any specific implementation
  - > MVM is the RI
- Launching of isolates similar to threads'

```
Isolate i = new Isolate("bsh.Interpreter", "myScript.bsh");
i.start();
i.halt();  // Can safely terminate the Isolate asynchronously
```

- Additional attributes specified using properties
  - > Classpath, current directory, heap sizing, ...
  - > Can specify standard I/O stream (e.g., for redirection)
- Isolate status monitored via messages

Inter-Isolate Communications

- Existing mechanism continue to work
  - > Sockets, Files, NIO, ...
- Links
  - > Isolate-specific mechanism
  - > One-way synchronous communication between two isolates
  - > Implements a rendez-vous semantics
  - > Built-in support for communicating
    - > Byte arrays, Strings, Serializable objects,
    - > Isolate and Link instances,
    - > Composites of the above

Links

```
...
Isolate receiver = new Isolate(Receiver.class.getName());
Link l = Link.newLink(Isolate.currentIsolate(), receiver);
receiver.start(l);  // Pass link to receiver at launch-time
l.send(LinkMessage.newStringMessage(“Hello”); // block till message delivered
...
```

```
class Receiver {
  public static void main(String [] args) {
    Link l = Isolate.getLinks()[0];  // obtain link passed at launch time
    LinkMessage message = l.receive();  // block till message arrive
    String s = message.extractString();
    ...
```

Monitoring Life-cycle events

- Isolate status communicated using ***status*** links

```
Isolate i = new Isolate("bsh.Interpreter", "myScript.bsh");
Link statusLink = i.newStatusLink(Isolate.currentIsolate());
i.start();     // Launch program
do {           // Wait till execution completes
   LinkMessage message = statusLink.receive();
   IsolateStatus.State state = message.extractStatus().getState();
   if (state == IsolateStatus.State.EXITED) break;
} while(true);
```

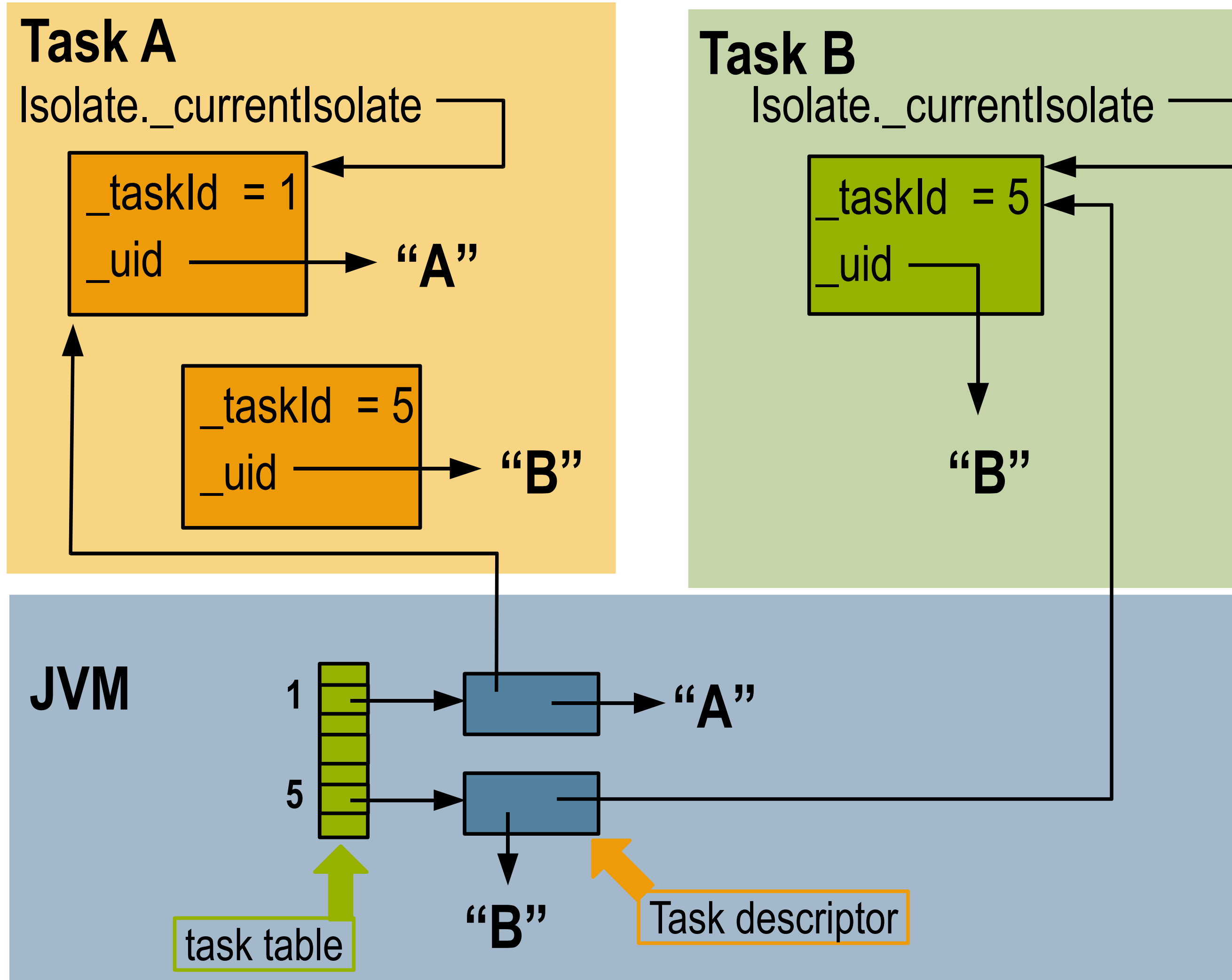
Task Management

- Additional VM data structures
  - > Array mappings task identifier to task data structure
    - > also maps task identifier to global unique identifier
  - > task identifier used to access all table-mediated per-task resources
- Recycling of task identifier
  - > Requires clearing of all roots to task-specific data
  - > Done opportunistically at GC times
  - > Force GC if no task ID left

Task Identifiers

- 3 Identifiers
  - Global uid, visible across all tasks
    - a string encoding a unique ID
  - JVM-defined identifier for fast access to per-task data
    - an int index that identifies slot in task-tables maintained by the JVM
    - Allocated by the JVM when a task start
  - An object reference
    - Reference to the task object, unique within a task
    - Implemented by javax.isolate.Isolate

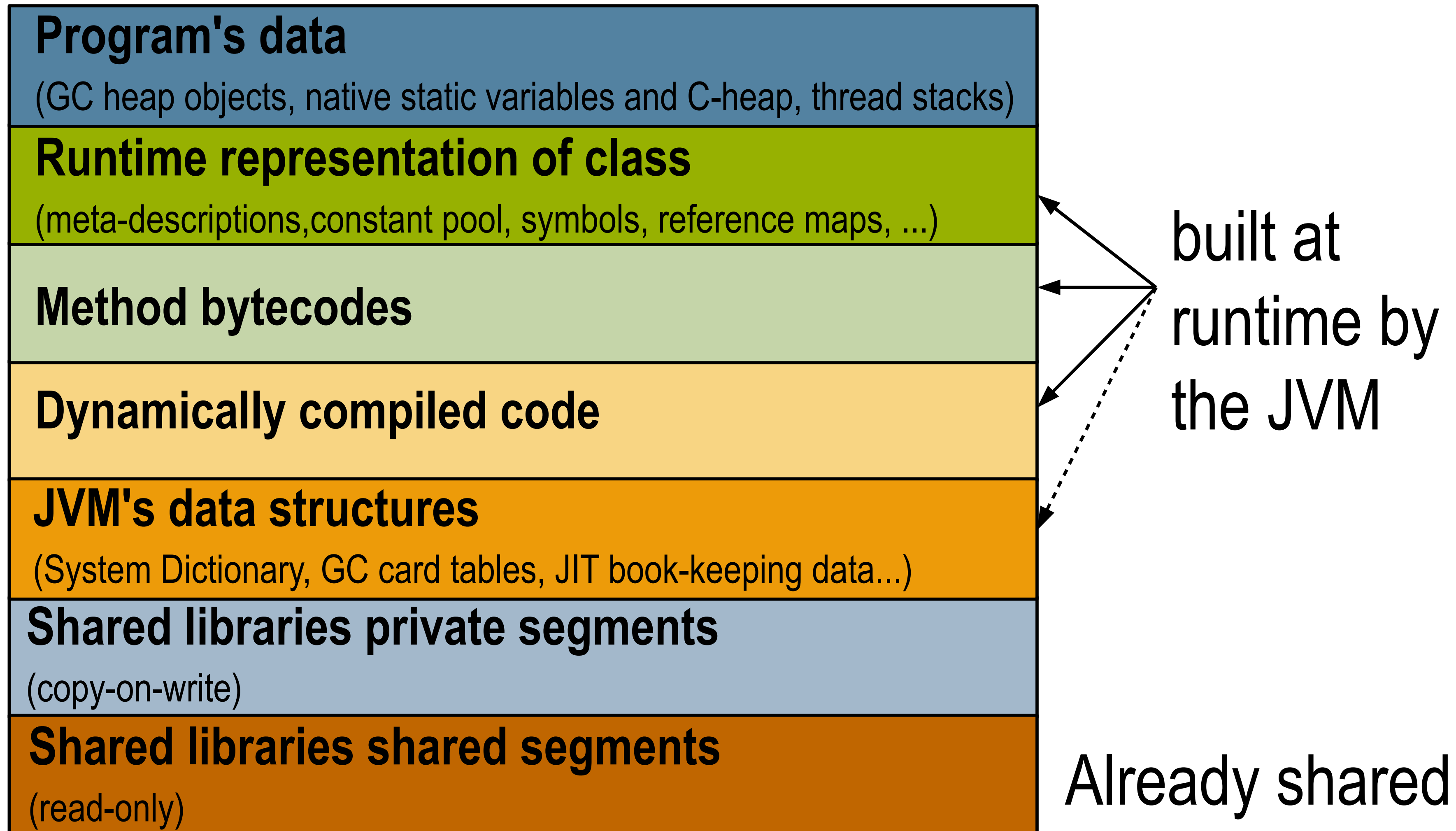
JDK / JVM Task Interface



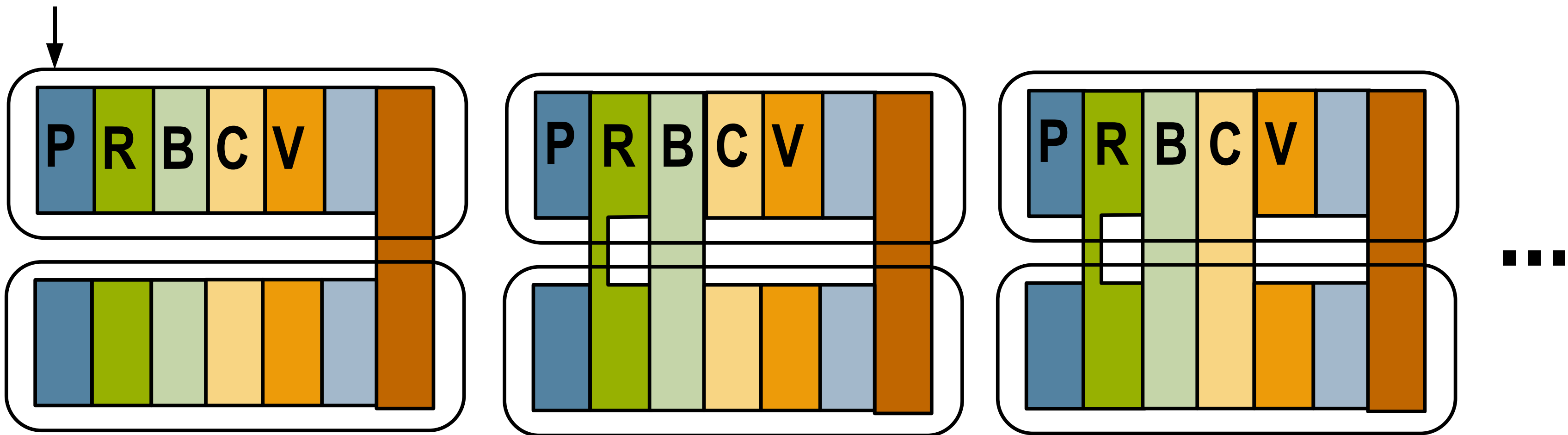
The Root of all ills

- Lack of sharing across running programs
  - Bloat aggregate footprint
    - Impact scalability / number of programs run simultaneously
    - Obstacle to deployment on small client with demand for rich interface
  - Slow down startup
    - JVM loads ~ 300 classes to start
      - 244 for JDK 1.4.2, 292 for JDK 1.5.0, 309 for JDK 1.6
    - Makes the Java platform unfit for small programs
  - Impact performance
    - Overhead to verify and construct an optimized program image is repeated at each program execution

Anatomy of a Java Program



Design Space for Sharing

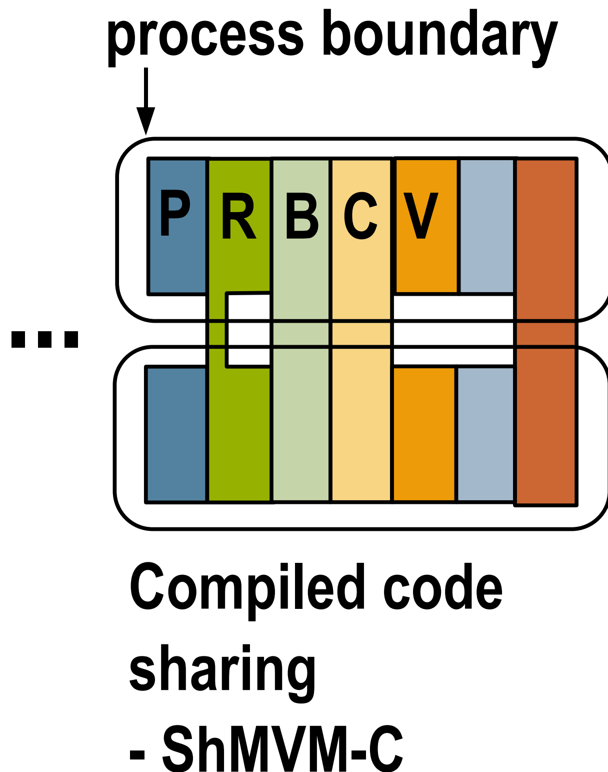


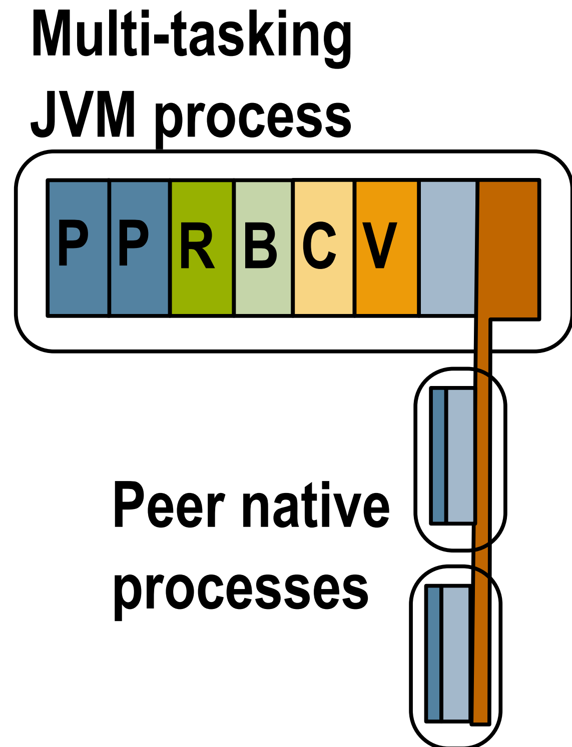
Standard JVM

Bytecodes sharing
- HotSpot w/ CDS
- ShMVM-B
- SLVM

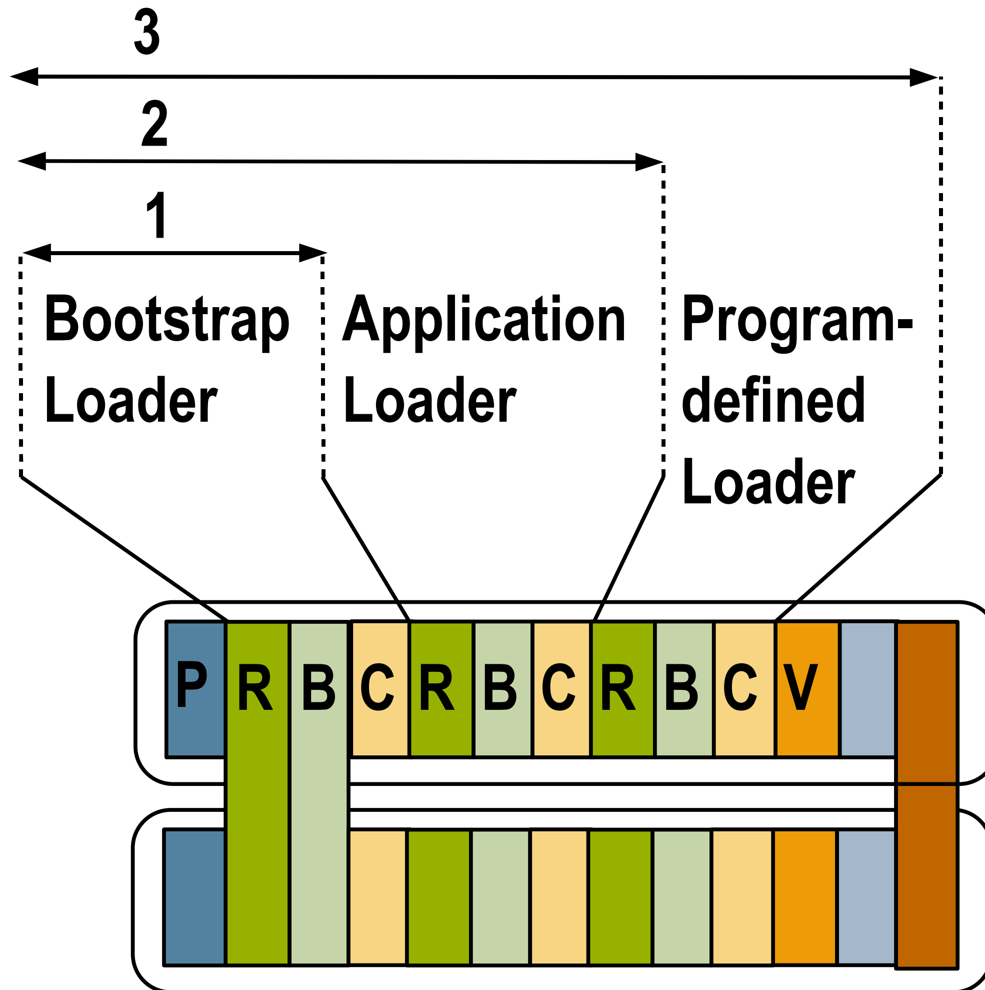
JIT-ed code sharing
- ShMVM-C

Design Space for Sharing





Levels of Class Data Sharing



HotSpot class data sharing
level 1

MVM
level 2

CLSVM
level 3

What makes Sharing hard ?

- Fine grain dynamic linking
  - > Unit of dynamic linking is the class
    - > size of class >> size of shared library
    - > large number of cross-unit symbolic references
- Highly mutable meta-data
  - > bytecodes may change
  - > GC may update references
  - > dynamic optimizations may change data structures or machine instructions
    - > inline caches, fast type-checking, etc...
- Problematic when sharing across address space

Sharing in MVM

- Level 2 sharing of classes
  - > Sharing of the runtime representation of classes defined by the boot, extension, and system loaders
  - > Share the entire runtime representation of classes, method bytecode and JIT-compiled code
  - > Transparently share immutable program-visible objects
    - > e.g., interned strings
    - > use virtualized monitors
- Share most of the JVM runtime data structures
  - > Class dictionary, table of interned strings, GC and compiler data structures
  - > Native state of core native methods

Exploits quasi-static Linking

- Dynamic links often constant across programs
  - For classes defined by class loader under the JVM's control
    - boot loader always resolve symbolic references identically for a given JVM version
    - system loader always resolve symbolic link identically for a given set of jar files.
- Constant symbolic links simplify sharing
    - Pointers to super-classes and interfaces identical
    - Offsets to class fields identical
    - virtual and interface tables can be shared
    - quickened bytecodes can be shared
    - compiled code exploiting resolved links can be shared

Class Sharing in MVM-lite

Class loading "domains"

rt.jar

Boot loaders

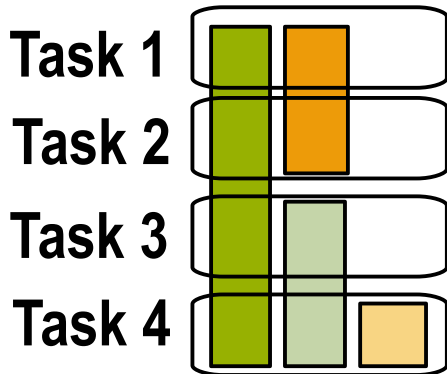
a.jar:b.jar:c.jar

d.jar:e.jar

System loaders

u.jar

User-defined loaders



sharing across tasks

Class Loader Keys

- Identifies a set of resolved class links
  - Identifies an ordered collection of class binary representations
  - Search of a class name over the collection is deterministic
    - linear search for first class name match
- Built-in class loaders assigned to a class loader key
  - Class loaders with the same class loader keys can share the runtime representation of their classes
    - use a loader reentrant class representation
    - only one loader reentrant representation per class loader key
  - Loader reentrant classes keyed on class loader key in the class dictionary

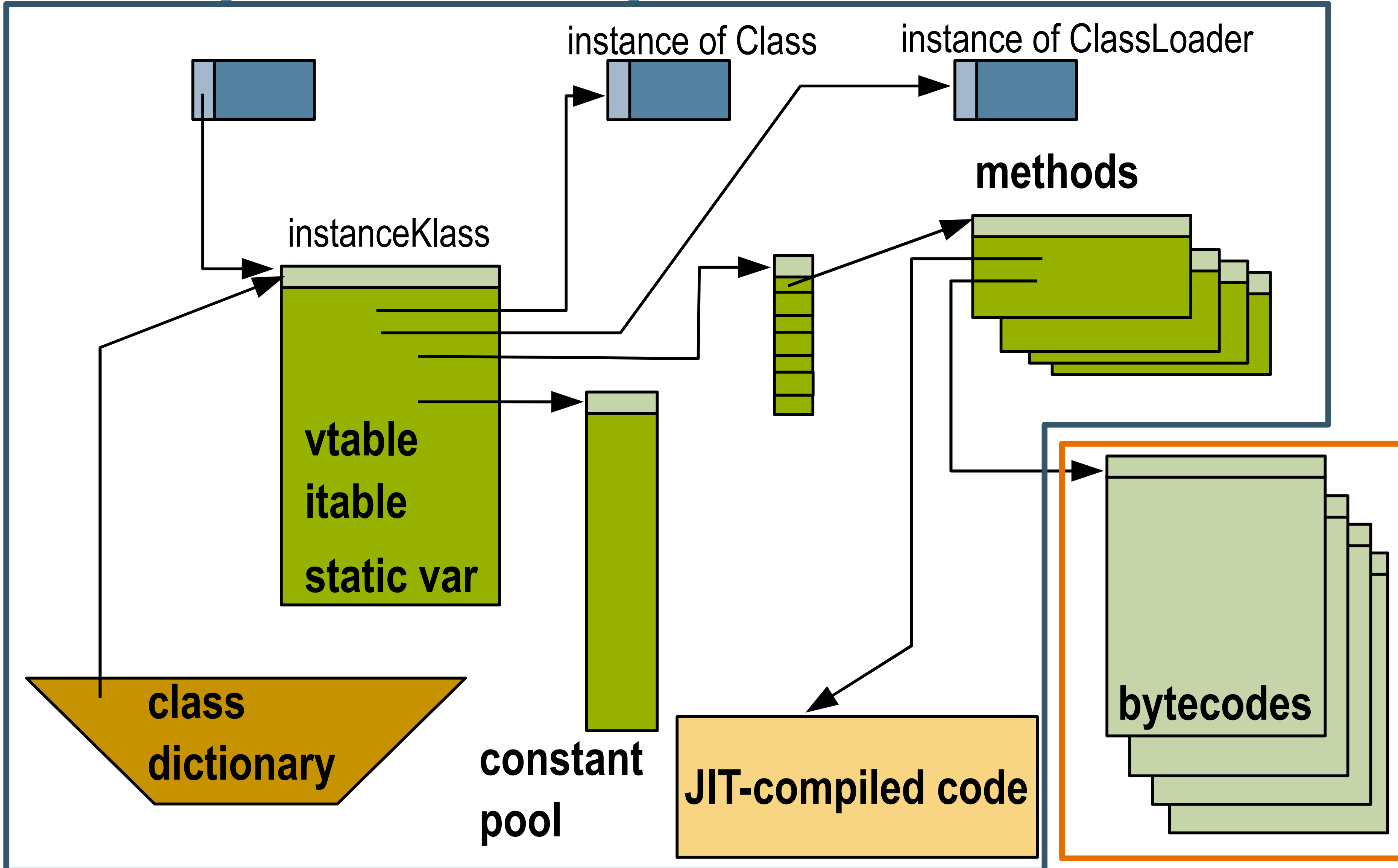
Class Loader Key Representation

- All boot loaders assigned to the same key
  - use null as class loader key
- Use interned strings as class loader keys for extension and system loaders
  - can compare class loader keys using their reference
  - can built system loader class loader keys by concatenation

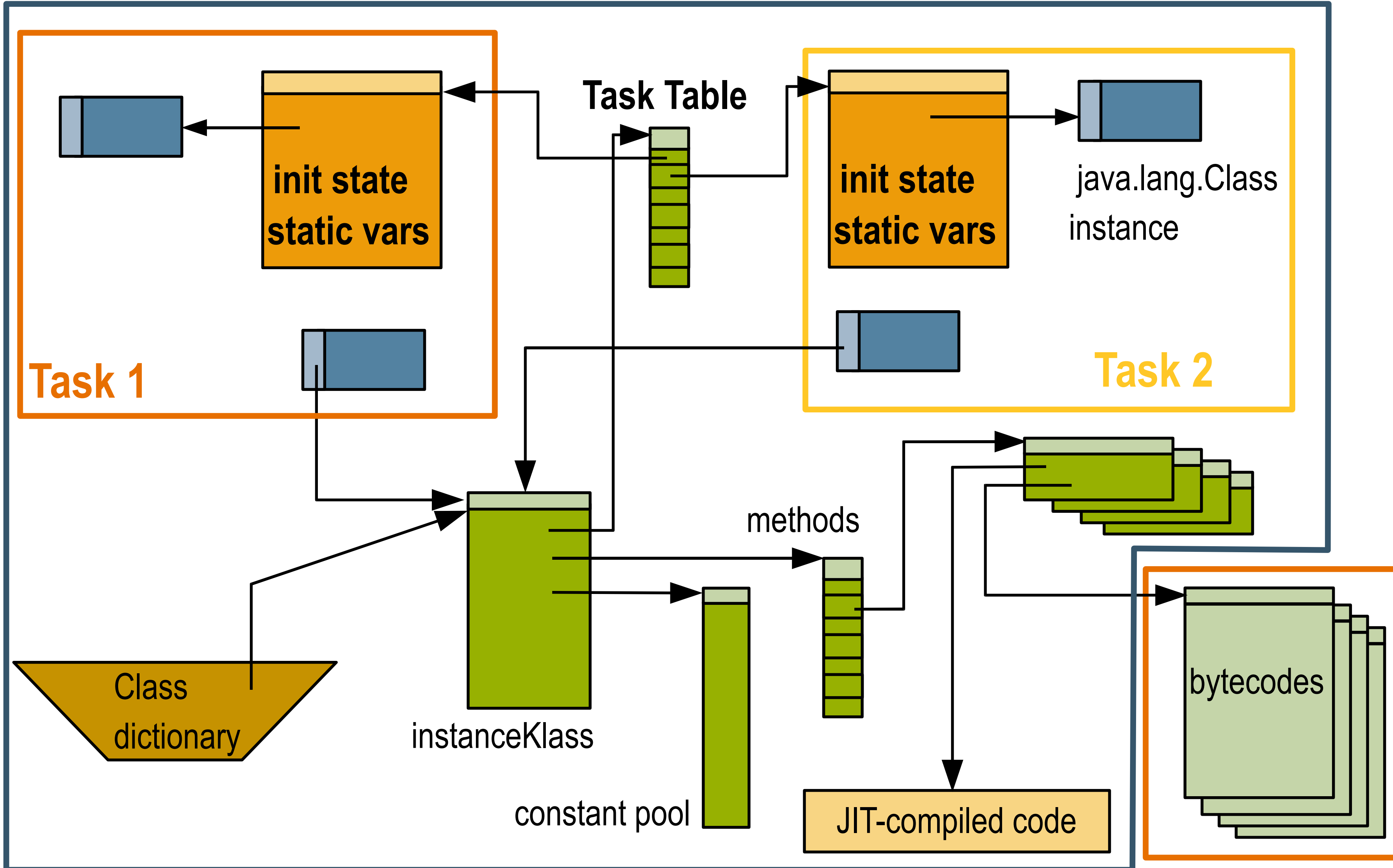
Class Loader Key Assignment

- Built-in loader already associated to an ordered collection of class representation
  - > Boot loaders associated to rt.jar
  - > Extension loaders associated to jars in the extension directory
  - > System loaders associated to a user-defined classpath
- Must take delegation relationship into account
- MVM uses a canonicalized classpath as keys
  - > Too conservative !
    - > relies on hand-crafted classpath
  - > Loader keys should be crafted by some Application Management Systems

HotSpot class representation



MVM Class Virtualization



Changes to instanceKlass

klass
instanceKlass
java mirror
task table
class loader
klassTaskTableOop
taskKlassOop
java mirror
java.lang.Class instance
in perm gen
in old gen
class loader key
Class defined by reentrant loader

java mirror
task table
class loader
taskKlassOop
java mirror
in perm gen
java.lang.Class instance
instance of subclass of java.lang.ClassLoader
Class defined by non-reentrant loader

Interpreter Changes

- Enables Bytecode Rewriting
  - added fast version of getstatic and putstatic
- Support for Class Initialization Barriers
  - changed get/put static, new, invokestatic
  - changed constant pool cache entries
    - store a klassTaskTableOop or a taskKlassOop in f1 for static field entries
    - store a klassTaskTableOop in f2 for static method entries of task-reentrant classes
    - modified GC iterators accordingly
- Extended Interpreter Runtime
  - fast class initialization, task termination

Class state transition barriers

- Link resolution barriers
  - > ldc, checkcast, instanceof, getfield, putfield, invokevirtual, invokeinterface, invokespecial
- Initialization barriers
  - > new, getstatic, putstatic, invokestatic
- Dynamically removable barriers
  - > Interpreter uses bytecode quickening techniques
  - > Compiled code uses code patching techniques
- Impact on task re-entrance
  - > initialization barriers cannot be eliminated
  - > link resolution barrier that cannot change the initialization state of a class can be eliminated
    - > e.g., ldc, getfield, putfield, invokevirtual, invokeinterface, invokespecial

Barrier Implementation & Elimination

- Why 2 entries ?
  - > Avoids extra load and test
    - – don't test for taskKlassOop presence, then for its initialization state
  - > Inlining of slow path for initializer thread
  - > Single entry possible for static initializer-free classes
- Simple static analysis + additional tricks
  - > Keep track of boot classes initialized at JVM startup
    - > "timestamp" instanceKlass built at JVM startup
      - – a non-startup class can never trigger the initialization of a startup class
    - > keep track of the "initializer" of the startup class
      - – for barrier elimination when accessor is startup class
  - > Other classic "redundancy" elimination possible
    - > e.g., hoisting off loop, etc.

MVM Class Virtualization

- Table-driven access to task-specific class data
  - Class initialization state, static variables, Class and ClassLoader objects
- Each thread carries a task identifier
  - Maps to index in the task table of every class
- Add class initialization and link resolution barriers
  - In the interpreter and in JIT-compiled code
  - Optimized away by the JIT
    - eliminate redundant or unnecessary barriers
- Direct access to task-specific class data for non-reentrant classes

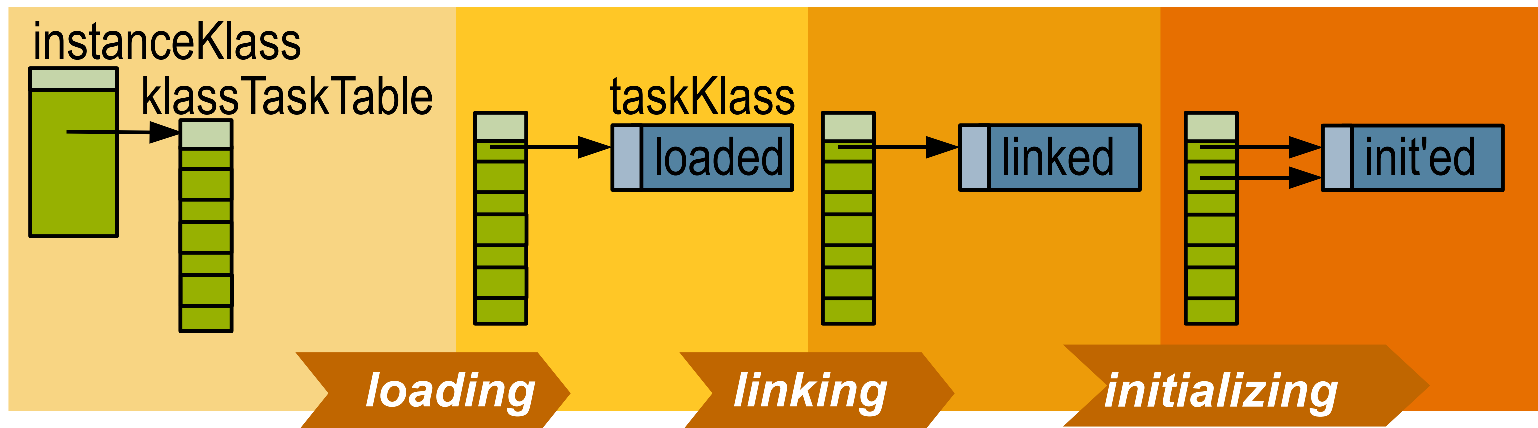
Barrier implementation

```
// HotSpot
sethi hi(klass), k
add k, lo(klass), k



ld [k+offset_reg], tos
```

```
// MVM
sethi hi(tasktable), k
add k, lo(tasktable), k
ld [thread + task_id], tmp      // table mediated
ld [tmp + k], k                 // access
brz, k slow_path                // initialization barrier
ld [k+offset_reg], tos
```



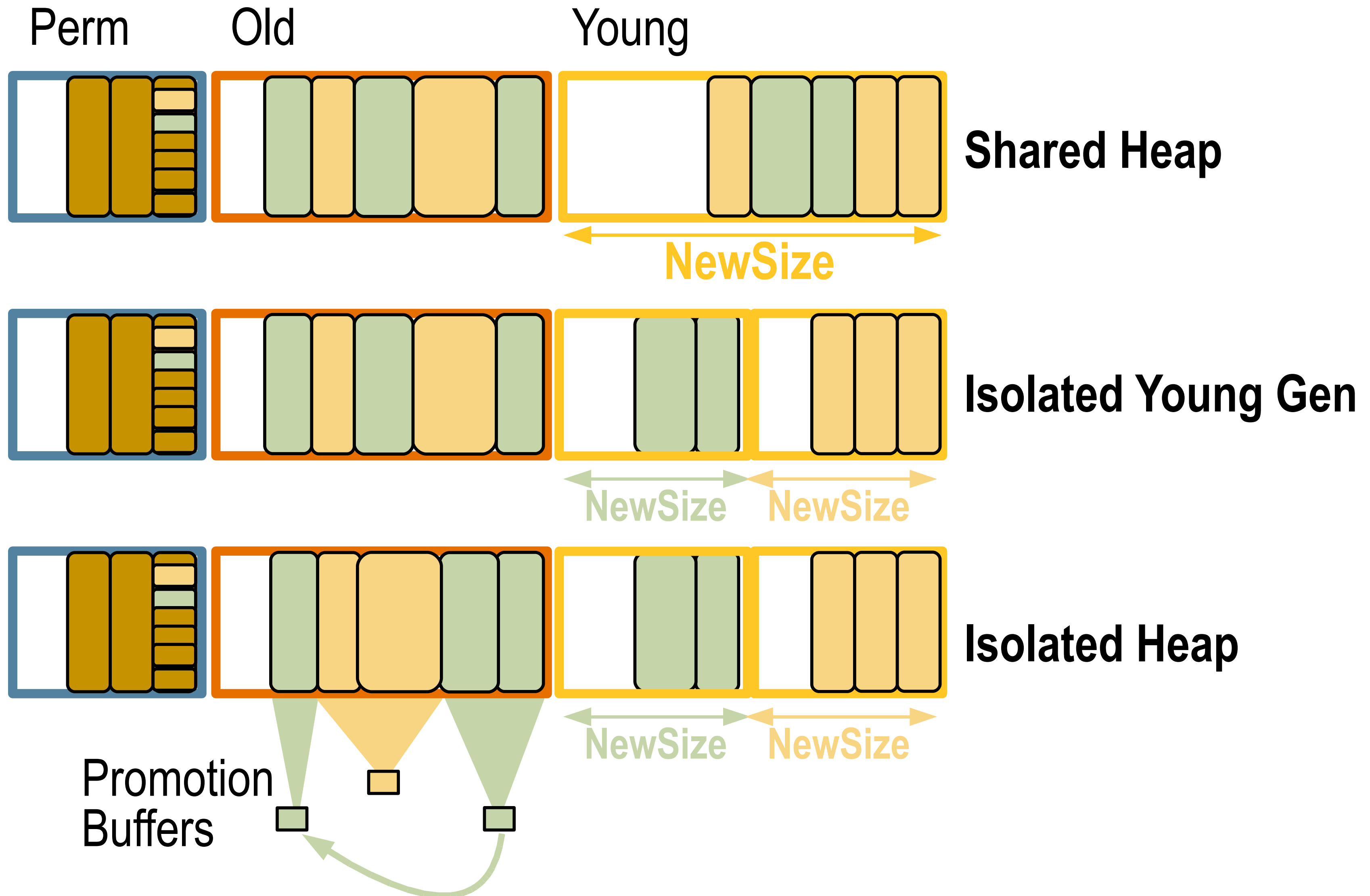
Impact of Compiled Code Sharing

- May defeat some optimization
  - may increase # polymorphic call sites
    - cleanup inline cache when task terminates
    - done on next GC
  - may reduce # CHA-based inlining
- Removal of compilation costs offset the above

Monitor Virtualization

- Preserve Isolation when transparently sharing objects across tasks
  - mostly, interned strings
- Pay cost of virtualization only when conflict becomes visible
  - virtualization kicks in when monitor is inflated
  - inflated monitor turns into a list of inflated monitor, with one inflated monitor per task
  - One thread per task at most can own the monitor
  - Fast path stay unchanged

MVM Garbage Collection Schemes



Shared Heap

- Stop-the-world (all tasks)
  - > Zeroing of root from terminated tasks at each GC
    - – Task table entries, dictionary entry
    - – Performed in GC prologues
- Per Task ReferenceQueues
  - > transparently virtualized
  - > Per task reference handler thread
  - > GC needs to identify what queue a reference should go to

Issues with Shared Heap

- Space not immediately reclaimed
  - > young gen space reclaimed at full GC time
  - > promote garbage to old gen, pollution from terminated tasks
- Hard to tune
  - > especially for generational heaps
- Poor performance isolation
  - > Stop all tasks, allocation intensive applications impose frequent GC to all tasks and may create nepotism, etc...
- What reference queue to process at GC time ?
  - > Task that triggered the GC ?
- Task may trigger GC while starting up

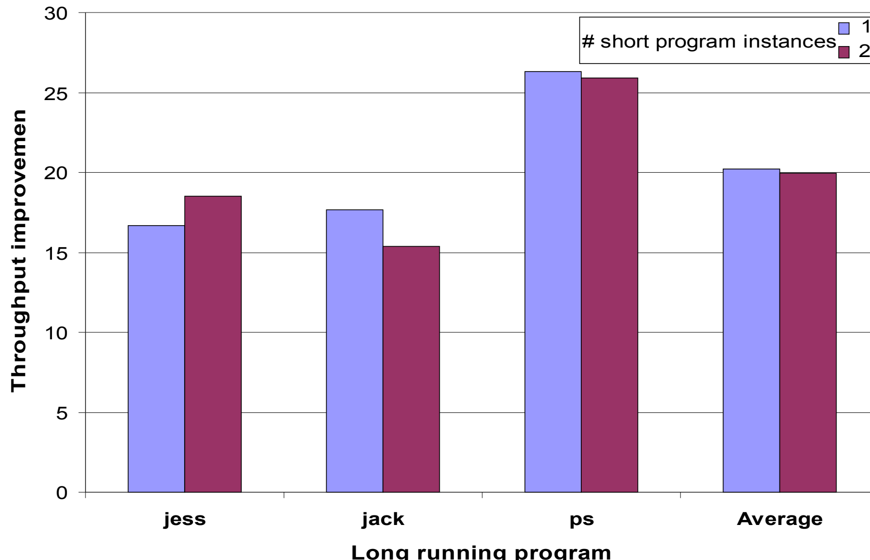
Isolated Young Generation

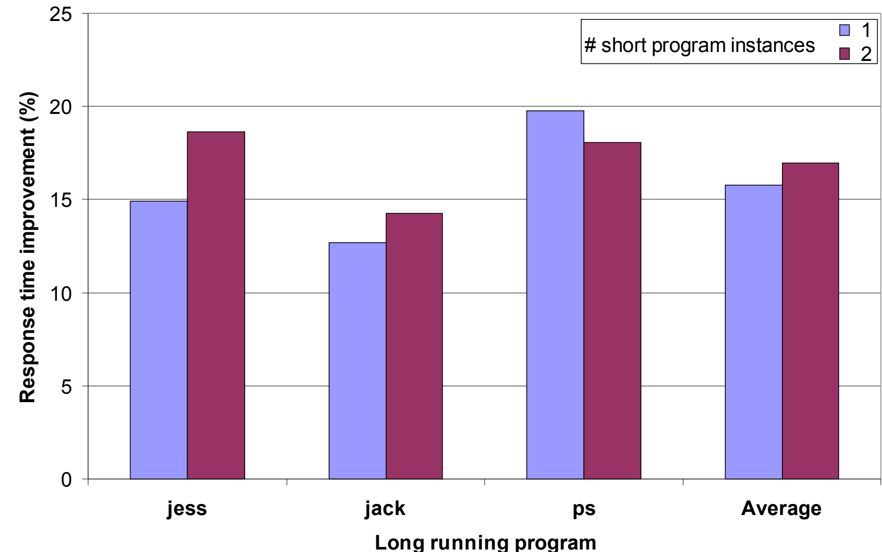
- New gen typically much smaller than old gen
- Most collections takes place in new gen
- Improved performance isolation
  > New gens can be sized independently for each tasks
  > One new gen can be scavenged independently of others
  > Scavenges can be performed concurrently to other tasks
  > Use of TLABs mask indirection overhead

GC Performance Isolation Impact

Load: run concurrently continuous streams of short programs with a long-lived program in the background

Improvement when using promotion buffers and concurrent scavenge





Isolated Young Generation Cons

- Young gen space reclaimed after one GC
  - > Need to clean references from old gen to young gen
  - > Scavenging of a single task iterates over all remembered sets
- Dead tasks old gen space reclaimed with Full GC
  - > Old gen pollution by dead tasks
- Full GC stops all tasks

Sun microsystems

Isolated Heap

- Old generation space sliced in fixed-size regions
  - Allocated on demand at promotion time
    - Used mostly when promoting objects
    - May be used to allocate large objects
  - Improve scavenging of new gen
    - Reduce amount of cards to scan to identify roots
  - Precise accounting of task's heap space usage
  - Enable Task-independent GC
    - GC performed concurrently with other tasks
- GC-less reclamation of terminated task heap space
  - Turn into free space on task termination
  - Enable immediate re-use of old and new gen space

Performance Summary

- Faster warm startup
  - Up to **x 5** for serial execution of headless programs, **x 2** for GUI-based programs loading > 1000 classes at startup, compared to HotSpot CDS
  - Up **70%** for GUI-based programs warmed up by other programs
- Competitive end-to-end performance
  - > Notable performance improvement on SPECjvm benchmarks
    - **~ 20 %** on AMD64, **~10 %** on SPARC improvement over CDS
- Leaner footprint when running multiple programs
  - > Multiple instance of same program **> 70%**

Paper trail

- Multitasking without compromise (OOPSLA 2001)
- Automated and Portable Native Code Isolation (Proceedings of IEEE ISSRE 2001)
- Code Sharing among Virtual Machines (ECOOP 2002)
- Dynamically Loaded Classes as Shared Libraries (IEEE IPDPS, 2003)
- A Multi-User Virtual Machine (Usenix 2003)
- Sharing the Runtime Representation of Classes across Class Loaders (ECOOP 2005)
- Task-aware Garbage collection in a Multi-tasking Virtual Machine (ISMM 2006)
- MTM2: Scalable Memory Management for Multi-tasking Managed Runtime Environments (ECOOP 2008)

Related Work (1)

- Faster startup, leaner footprint, code and data sharing
  - Cross-process sharing
    - HotSpot CDS, CLSVM (ECOOP02), Using ELF format and OS support (IPDPS)
  - Sharing across arbitrary class loaders (ECOOP 2005)
  - IBM's QuickSilver
    - (storing result of dynamic compilation on disk + stitching on re-use)
  - IBM re-usable VM
  - Xmem (PLDI 2008) sharing of objects across process

Related Work (2)s

- Isolation layered on top of the JVMs
  - Class Loader based isolation
    - Java applets
      - Sandbox model becoming insufficient for applets.
    - JKernel, JavaSeal
  - Czajkowski OOPSLA 2000, Application isolation in the virtual machine.
    - Use bytecode editing to virtualize static variables and other shared resource to overcome the issues of class loader based-isolation.

Related Work (3)

- Cloneable VM (VEE 2007)
  - Extend Isolate API with cloning capability for fast startup of fully isolated application. Application must be “cloning” aware to exploit this mechanism. Doesn't help with footprint.
- I-JVM
  - Replicate static application state, but allows reference to cross isolates, trading strong isolation for faster IPC. High risk of error propagation. No heap isolation.
- Luna (OSDI 2002)
  - Extends the Java programming language with special “Remote Pointer” type for shared data, distinct from local pointers. It enables strict isolation within resource control and fast IPC.

Related Work (4)

- Industrial Efforts
  - Sun's CLDC Hi Multi-tasking JVM
    - Industrial-strength implementation of a multi-tasking VM according to MVM principles (Sunlabs tech-transfer :-)
    - Use leaner Isolate API (no links)
    - Does heap accounting, but no isolated heap
  - SavaJE OS
    - AMS + ThreadGroup + ClassLoader + ad-hoc termination
  - .NET's Application Domains
    - Very similar to Isolate
    - Doesn't share assemblies across domain boundaries

Multi-tasking Virtual Machines

Laurent Daynès
Sun Microsystems Laboratories
laurent.daynes@sun.com

Sun microsystems