Developing a competitive JVM in Open Source

Pavel Ozhdikhin, Pavel Pervov

Contributors: Xiao-Feng Li, Vladimir Beliaev

Agenda

- About the Harmony project
- Harmony DRLVM
- Garbage Collectors in DRLVM
- DRLVM Execution Engines
- Summary

Apache Harmony

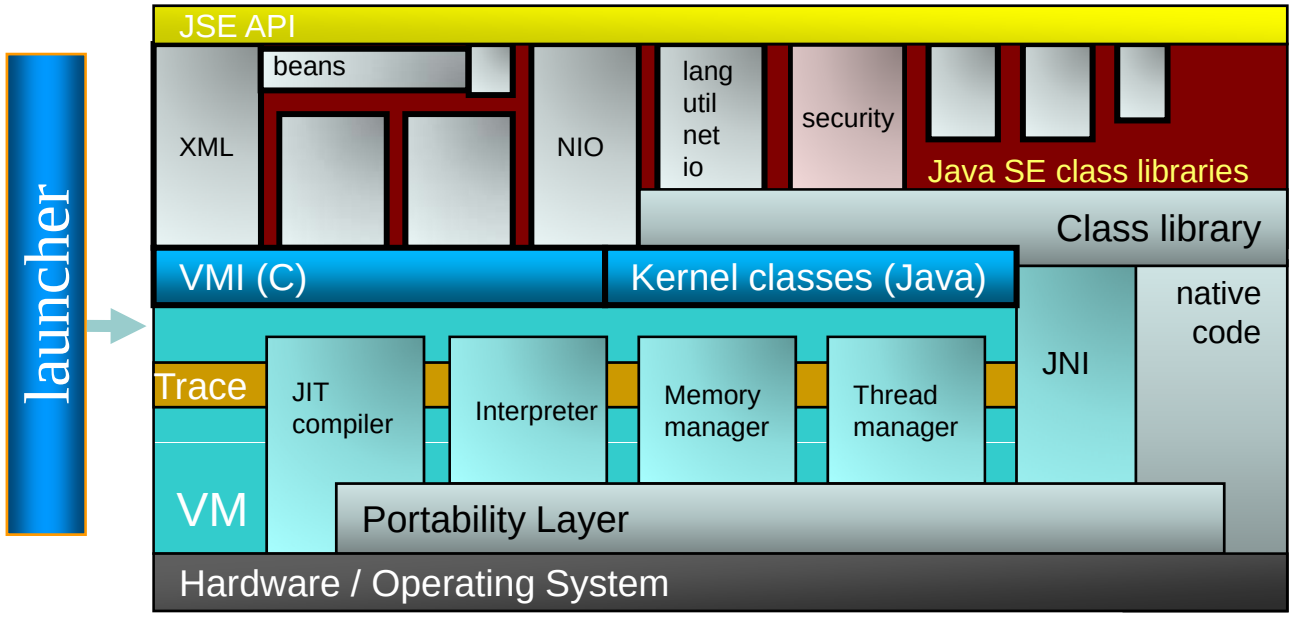
- Primary goal – full implementation of Java SE
  - Compatible class library
  - Competitive virtual machine
  - Full JDK toolset
- Founded in Apache Incubator, May 2005
- Became the Apache Project, Oct 2006
- Facts today
  - 27 committers at the moment, 50 commits weekly
  - 250 messages weekly in mailing list
  - 150 downloads weekly

Peculiarities of Open Source Development

- Problems
  - No initial credibility
  - Highly depends on the community
  - Strong competition
- Solutions
  - Openness
  - Strong modular design
  - Competitive performance

Design Modularity

- Modularity makes it easier for developers, researchers, and testers



Harmony Status

- ~2.3 million LOC (Java 1.6m, C/C++ 0.7m)
- Components
  - API: 98% JDK5, 90% JDK6
  - VMs: JCHEVM, BootJVM, SableVM, DRLVM, IBM J9, BEA JRockit
  - Tools: javac, javah, jarsigner, keytool
- Platforms
  - Windows/Linux, x86/x86-64/ipf
- Continuous integration infrastructure

Agenda

- About the Harmony project
- Harmony DRLVM
- Garbage Collectors in DRLVM
- DRLVM Execution Engines
- Summary

Harmony DRLVM

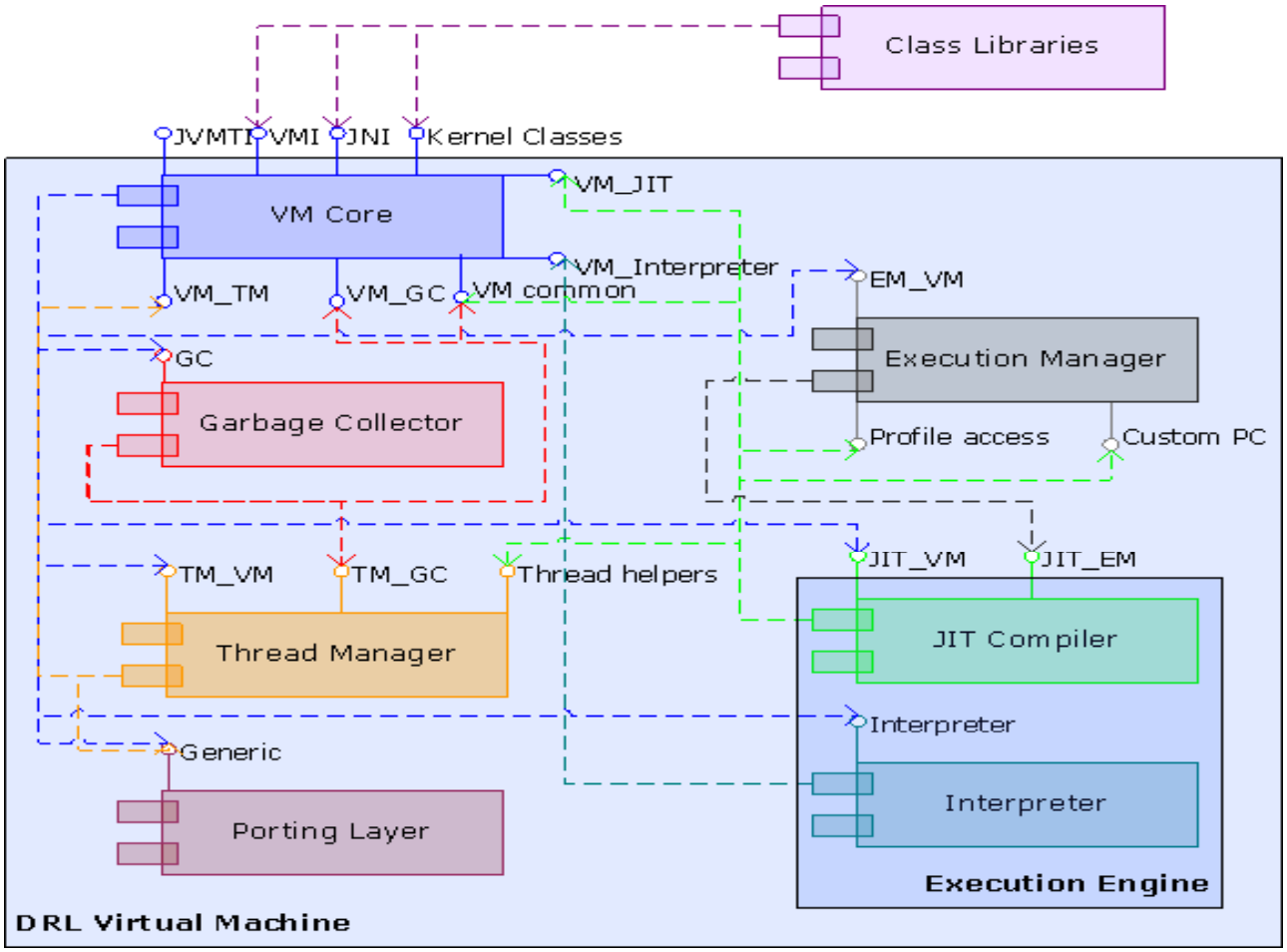
- The default VM for Apache Harmony

- Components
  - Two JIT compilers: fast and optimizing
  - Three GCs: copying, parallel, and concurrent
- Features
  - optimized monitors, JVMTI, class unloading, interpreter
- Targets
  - Robustness, performance, and modularity
  - Server and desktop

DRLVM Modularity Principles

- *Modularity*: Functionality is grouped into a limited number of coarse-grained modules with well defined interfaces.
- *Pluggability*: Module implementations can be replaced at compile time or run time. Multiple implementations of a given module are possible.
- *Consistency*: Interfaces are consistent across platforms.
- *Performance*: Interfaces fully enable implementation of modules optimized for specific target platforms.

DRLVM Modules



Agenda

- About the Harmony project
- Harmony DRLVM
- Garbage Collectors in DRLVM
- DRLVM Execution Engines
- Summary

DRLVM GC Design Goals

- Robust, performing, and flexible GC
  - Robustness: modularity and code quality
  - Performance: scalability and throughput
  - Flexibility: configurability and extensibility

DRLVM GC Current Status

- GCv4.1
  - Copying collector with compaction fallback
  - Sequential, non-generational
- GCv5
  - Copying collector with compaction fallback
  - Parallel, generational (optional)
- Tick
  - On-the-fly mark-sweep-compact
  - Concurrent, parallel, generational (optional)

GCv4.1: Characteristics

Pros

- Good performance
- Easy to learn

Cons

- Algorithm is not parallel
  - Cannot leverage multiple cores
- Has no generational support

GCv5: Characteristics

Pros

- Good performance
- Scalable on multiple cores
- Runtime adaptations

Cons

- Pause time in major collection is high
- No support for conservative collection

Tick: Characteristics

Pros

- Short collection pause time
  - Target is at ms level
- Parallel and adaptive collection
- Diverse working models
  - Concurrent or stop-the-world
  - Standalone or generational

Cons

- Collection pause is a tradeoff with GC throughput

Agenda

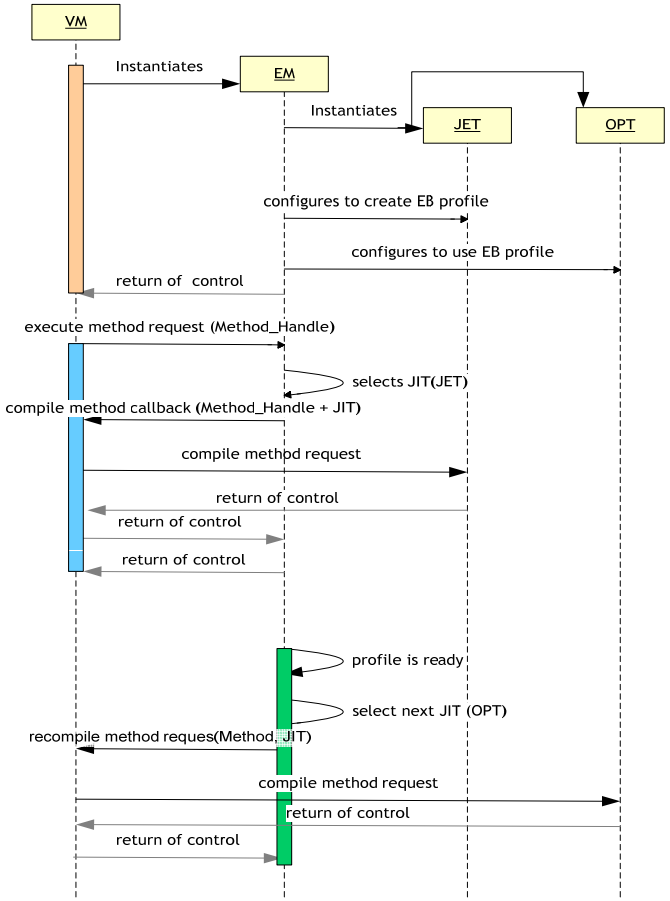
- About the Harmony project
- Harmony DRLVM
- Garbage Collectors in DRLVM
- DRLVM Execution Engines
- Summary

DRLVM execution engines

- The Execution Manager
- Interpreter
- Jitrino compilers:
  - Jitrino.JET
  - Jitrino.OPT
    - Optimizations
    - Pipeline Management Framework
    - Internal profiler

The Execution Manager

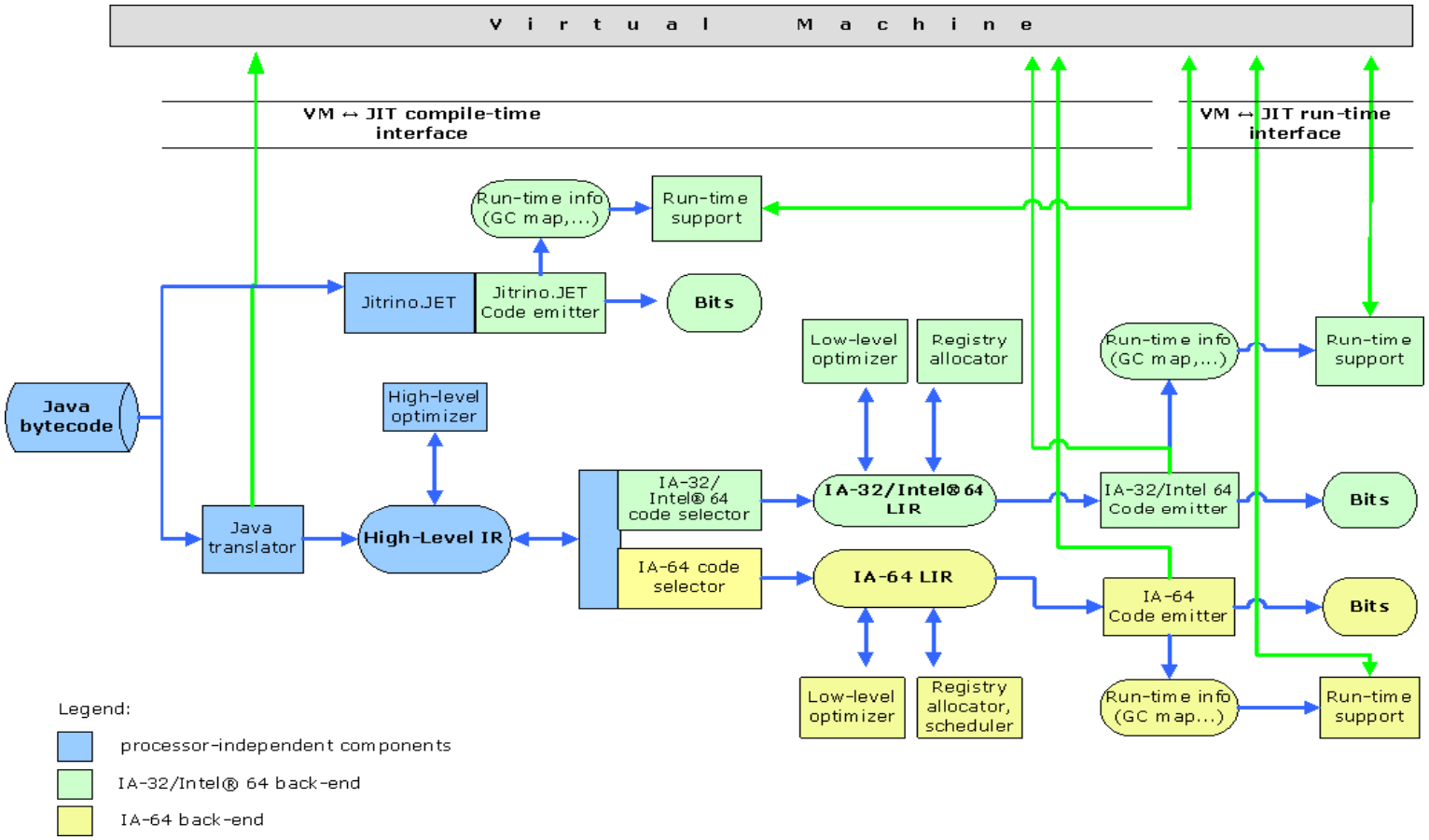
- Keeps a registry for all execution engines and profile collectors available at run time
- Selects an execution engine to compile a method by a VM request according to the configuration file
- Coordinates profile collection and use between various execution engines
- Supports asynchronous recompilation in a separate thread to utilize multi-core



Dynamic profilers

- **EB_PROFILER**
  Entry/backedge profile. Collects 2 values for each method:
  - number of times a method has been called (entry counter)
  - number of loop interactions (backedge counter) performed in a method
- **EDGE_PROFILER**
  Edge profile. Collects 2 types of values for each method:
  - number of times a method has been called
  - number of times every branch in a method has been taken
- **VALUE_PROFILER**
  Value profile. Collects up to N the most frequent values for each registered profiling site in a method. Uses advanced Top-N-Value algorithm.

Jitrino Architecture



Jitrino.JET – baseline compiler

- Simple: no internal representation, just 2 passes over bytecode
- Small: ~500K code, ~14K NSLOCs
- Fast: Compilation speed ~ 10-20K methods per second (1.5Ghz laptop)
- Supports JVMTI, VMMagic and can easily be modified to support new features
- Produces more then 10 times faster code than the interpreter (and ~2 times slower than the code made by Jitrino.OPT)

Jitrino.JET: log sample

Java method:

```
public static int max(int x, int y) {
  return x > y ? x : y;
}
```

Prologue: <store all callee-save registers in use>

```
;;   0) ILOAD_0
;;   1) ILOAD_1
;;   2) IF_ICMPLE       ->9<-
0x03EB00B6  cmp ebx, esi
0x03EB00B8  jle dword 0x11
;;   5) ILOAD_0
;;   6) GOTO            ->10<-
0x03EB00BE  mov [ebp+0xffffff14], ebx
0x03EB00C4  jmp 0xb
;;   9) ILOAD_1
0x03EB00C9  mov [ebp+0xffffff14], esi
;;  10) IRETURN
0x03EB00CF  mov eax, [ebp+0xffffff14]
```

Epilogue: <restore all callee-save register in use>

Jitrino.OPT – optimizing compiler

- The fast, aggressively optimizing compiler
- Features:
  - High- and low-level intermediate representations
    - Most optimizations run at the platform-independent high level
  - Supports edge and value profiles
  - Pipeline Management Framework
  - A flexible logging system enables tracing of major Jitrino activities, including detailed IR dumps during compilation

Jitrino.OPT optimizations

- Guarded devirtualization
- Global Code Motion
- Escape Analysis based optimizations:
  - Synchronization elimination
  - Scalar replacement
- Array initialization/copying optimizations
- Array bounds check elimination

...and many other most known optimizations

Advanced optimizations

- VM Magics and helper inlining
  - Allow developers to write performance critical code in Java using address arithmetic and low-level compiler intrinsics.
- Value profile guided devirtualization
  - Effectively de-virtualize not only virtual but also interface and abstract calls
- Lazy exceptions
  - Create exception objects on demand, i.e. only if it’s actually used in the exception handler

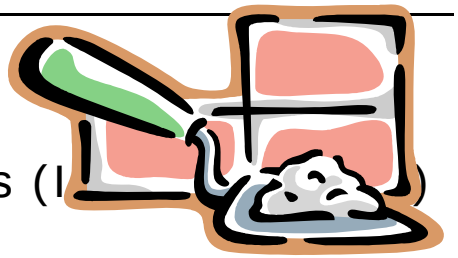
Helper: Bump-Pointer Allocation

```
@Inline
public static Address alloc(int objSize, int allocationHandle) {
    Address TLS_BASE = VMHelper.getTlsBaseAddress();

    Address allocator_addr = TLS_BASE.plus(TLS_GC_OFFSET);
    Address allocator = allocator_addr.loadAddress();
    Address free_addr = allocator.plus(0);
    Address free = free_addr.loadAddress();
    Address ceiling = allocator.plus(4).loadAddress();

    Address new_free = free.plus(objSize);
    if (new_free.LE(ceiling)) {
        free_addr.store(new_free);
        free.store(allocationHandle);
        return free;
    }
    return VMHelper.newResolved (objSize, allocationHandle);
}
```

Pipeline Management Framework



PMF - the JIT pluggability vehicle

- PMF features:
  - Standard interface for the pipeline steps (I[illegible])
  - Nested pipelines
  - Full control over the pipeline steps and their options through the Java property mechanism
  - Rich control over the logging based on JIT instances, pipelines, class and method filters
- PMF details:
  - http://harmony.apache.org/subcomponents/drlvm/JIT_PMF.html

Jitrino.OPT internal profiler

The internal profiler (iprof) in the Jitrino.OPT compiler can instrument the code so that per-method counters of the instructions executed at run time will be dumped.

- To use iprof you need to create the iprof.cfg configuration file with the profiler's configuration and specify the following option: -XX:jit.arg.codegen.iprof=on
- An example of the iprof output:

Method name	Insts	ByteCodeSize	MaxBBExec	HottestBBNum	…
java/lang/Thread.<clinit>	7	13	1	2	…
java/lang/Object.<init>	6445	1	6445	2	…
java/lang/Thread.<init>	2440	257	24	0	…
…	…	…	…	…	…

JIT Resources

- Execution Manager:
  http://harmony.apache.org/subcomponents/drlvm/EM.html
- Jitrino JIT Compiler:
  http://harmony.apache.org/subcomponents/drlvm/JIT.html
- Pipeline Management Framework and Jitrino logging system:
  http://harmony.apache.org/subcomponents/drlvm/JIT_PMF.html
- Jitrino.OPT internal profiler:
  http://harmony.apache.org/subcomponents/drlvm/internal_profiler.html
- Harmony performance reports:
  http://harmony.apache.org/performance.html

Agenda

- About the Harmony project
- Harmony DRLVM
- Garbage Collectors in DRLVM
- DRLVM Execution Engines
- Summary

Summary

- Harmony DRLVM is a product-quality VM being developed in Open Source
- DRLVM benefits from its modularity and pluggability in
  - Development, research and testing
- DRLVM provides a competitive performance with its advanced JIT and GC implementations

What's in this for me

- Build your research projects taking advantage of DRLVM modular design
- Use Harmony DRLVM to run your product
- Reuse parallel Garbage Collector, Classlib or JIT
- Contribute to the Harmony project with your ideas and energies

Resources

- Harmony project:
  http://harmony.apache.org
- Project downloads:
  http://harmony.apache.org/download.cgi
- DRLVM Developer's Guide:
  http://harmony.apache.org/subcomponents/drlvm/developers_guide.html
- Debugging DRLVM
  http://harmony.apache.org/subcomponents/drlvm/debugging_VM_and_JIT.html
- How to write GC for DRLVM:
  http://harmony.apache.org/subcomponents/drlvm/gc-howto.html

Thanks!

Q&A