Open-source, language-agnostic compiler technology for Eclipse OMR

Daryl Maier
Senior Software Developer at IBM Canada
IBM Runtimes
maier@ca.ibm.com

IBM Runtime Technologies

Eclipse OMR
@EclipseOMR

Follow

#Eclipse OMR now has a Just-In-Time compiler!

Initial contribution of compiler technology by 0xdaryl · P…

high-level optimization technology featuring classic compiler optimizations, loop optimizations, control and data flow analyses, and support data structures code generation technology with...

github.com

http:://twitter.com/eclipseomr

Testarossa compiler technology

- Created in 1998 as an IBM closed source project
- Heritage is a dynamic just-in-time (JIT) compiler for embedded Java
- Clean room implementation
  - Mix of C++, C, native assembler
- Design goals
  - Fast startup time
  - Miserly memory management
  - Flexible to meet different footprint configurations
- Optimizations
  - Configurable high-level optimization framework
  - High performance code generation with deep platform exploitation
- Dynamic recompilation with profile-directed feedback
- Speculative optimizations and supporting runtime framework
- Proven adaptable technology
  - Leveraged in static compiler backends and binary translators

Testarossa technology highlights: 1998-today

- Languages:
  - Production: Java ME and SE, COBOL, PL/I, Z binary emulator, binary (re)optimizer
  - Prototypes: Ruby, Python, SOM++, and more…

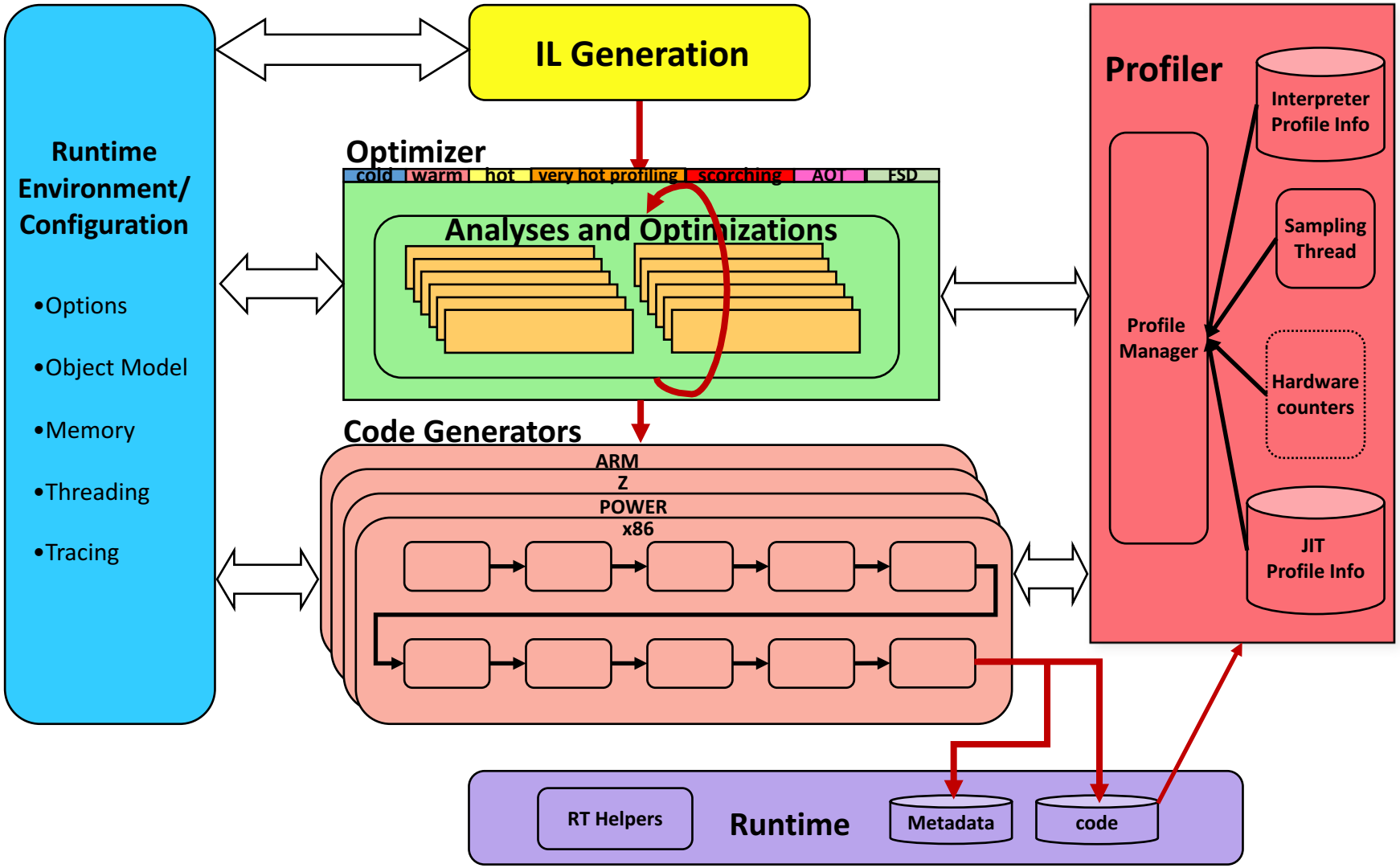
- Some technology highlights implemented by the Java JIT :
  - Cooperative suspend (1999)
  - Diagnostic abilities: e.g. limit files, per method options (1999)
  - Full optimization while supporting type accurate GC (1999)
  - AOT (rom-able) compilation for Java (1999)
  - Aggressive runtime native code patching (2000)
  - Invocation and time-based compilation triggers (2000)
  - Adaptive compilation (cold, warm, hot, very hot, scorching) (1999)
  - JIT profiling infrastructure and optimizations (2001)
  - Speculative class hierarchy based inlining and optimization (2001)
  - Fairly complete set of classical compiler optimizations and dataflow analyses (2001)
  - Java-specific optimizations like ”check” removal (2001)
  - Java debug support (2001)
  - Escape analysis and stack allocation (2001)
  - Automatic lock coarsening (2002)
  - Multiple code caches (2005)
  - Asynchronous compilation (2006)
  - Interpreter profiling (2006)
  - Real-time Specification for Java (AOT and JIT) (2005)
  - Dynamic AOT compilation for Java (2006)
  - Hot Code Replacement support (2007)
  - Compressed references (2007)
  - Multiple compilation threads (2010)
  - On stack replacement (2013)
  - Transactional Memory (2013)
  - Packed objects (2013)
  - Multitenancy (2013)
  - Auto SIMD (2014)
  - Auto GPU (2014)
  - Heuristic tuning and retuning (1999– ongoing)

- Platforms that are or have been supported :
  - ME: ARM32, X86(IA32), MIPS, POWER, SH4
  - 32-bit SE: ARM, POWER, X86, Z
  - 64-bit SE: POWER, X86, Z
  - Hard real-time (RTSJ compliant): IA32
  - COBOL, PL/I, COBOL Automatic Binary Optimizer: Z
  - Z binary emulator: X86, P

- Performance metrics that have been or are actively tracked :
  - Latency (elapsed time)
  - Throughput (operations / sec)
  - Start-up time
  - Ramp-up time
  - CPU consumption
  - Resource consumption at idle
  - Compilation time
  - Memory footprint
  - JIT library size
  - Incremental pauses

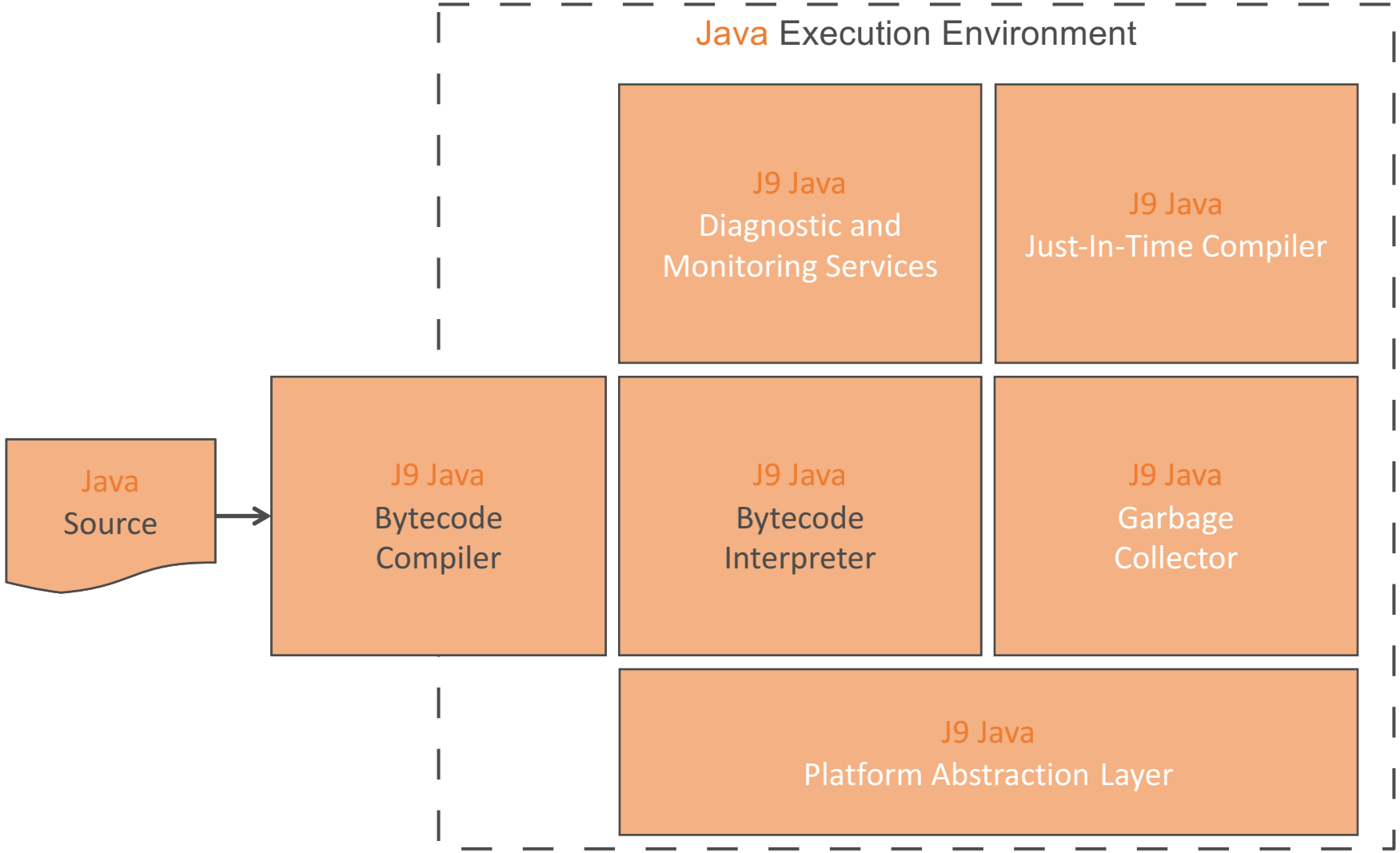
- Hardware exploitation highlights:
  - Efficient CPU instruction sequences
  - Managing different kinds of hardware registers
  - Exploiting hardware data type support
  - Cryptographic, compression acceleration
  - Character conversion loop recognition and acceleration
  - Atomic locking and other synchronization optimization
  - Simultaneous Multi Threading
  - Transactional Memory
  - SIMD (Single instruction multiple data)
  - GPU (Graphics processing unit)

Testarossa compilation process

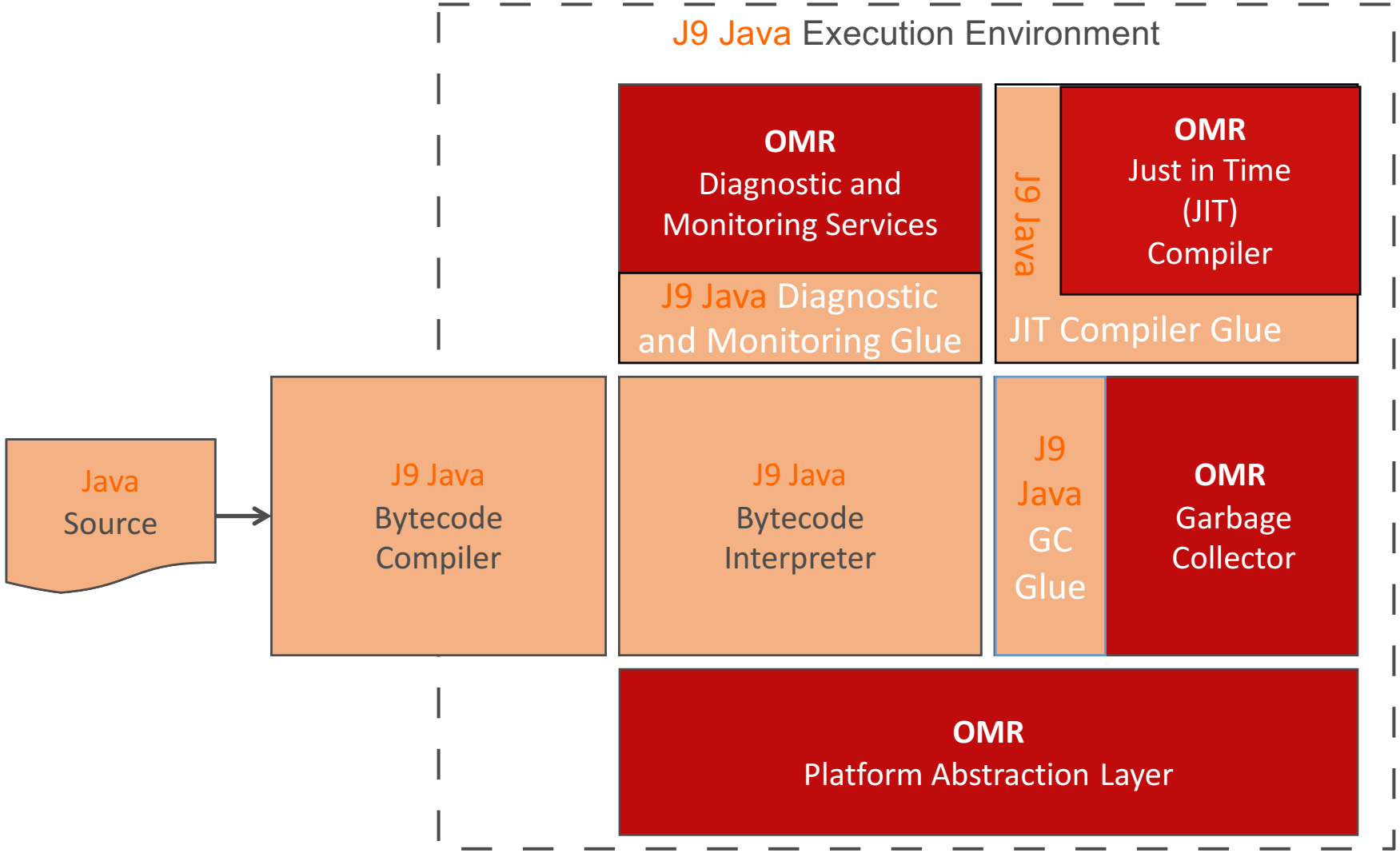


How did we open source this as a generic compilation framework?

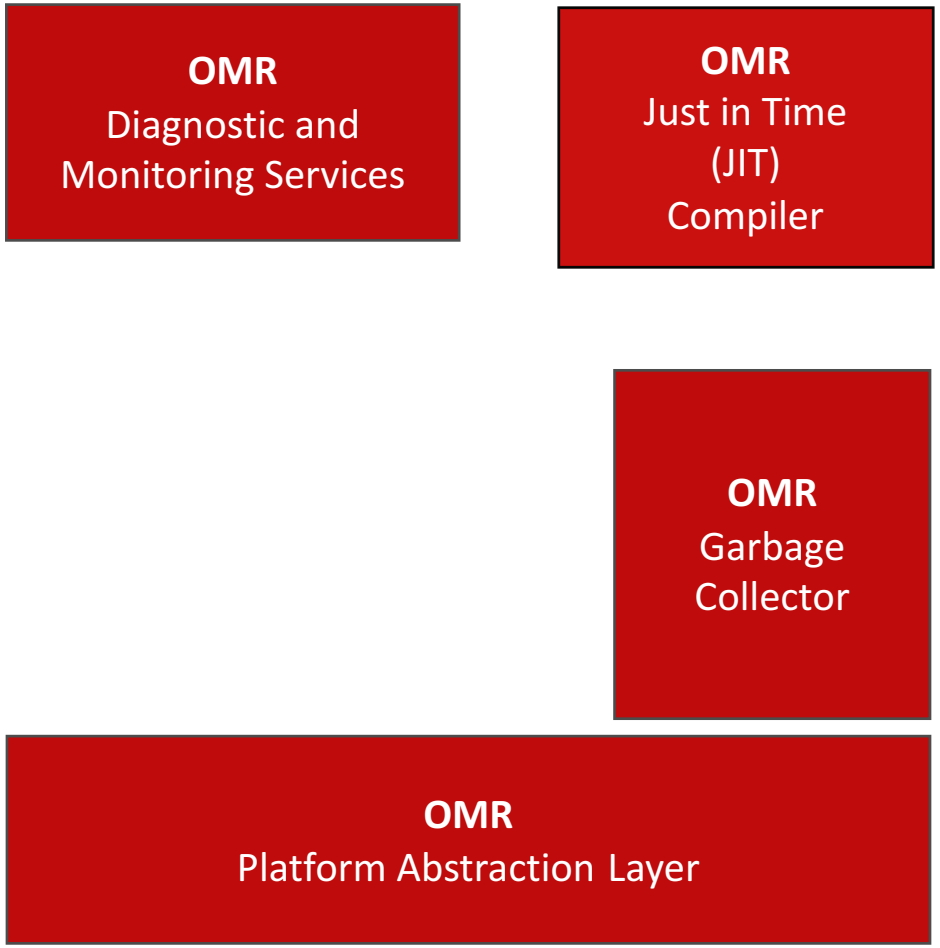
Starting with an enterprise-calibre IBM J9 Java runtime



Refactor “Java”-ness into a ***Glue*** layer that adds language specifics to each core component



Form OMR around core components



Key Goals for OMR

1. OMR has ***no*** language semantics

2. OMR is ***not*** a language runtime

3. OMR components can be integrated into ***any*** language runtime, new or existing, ***without*** influencing language semantics

Eclipse OMR

Created March 2016

http://www.eclipse.org/omr
https://github.com/eclipse/omr
https://developer.ibm.com/open/omr/

Dual License:
Eclipse Public License V1.0
Apache 2.0

Users and contributors very welcome!

https://github.com/eclipse/omr/blob/master/CONTRIBUTING.md

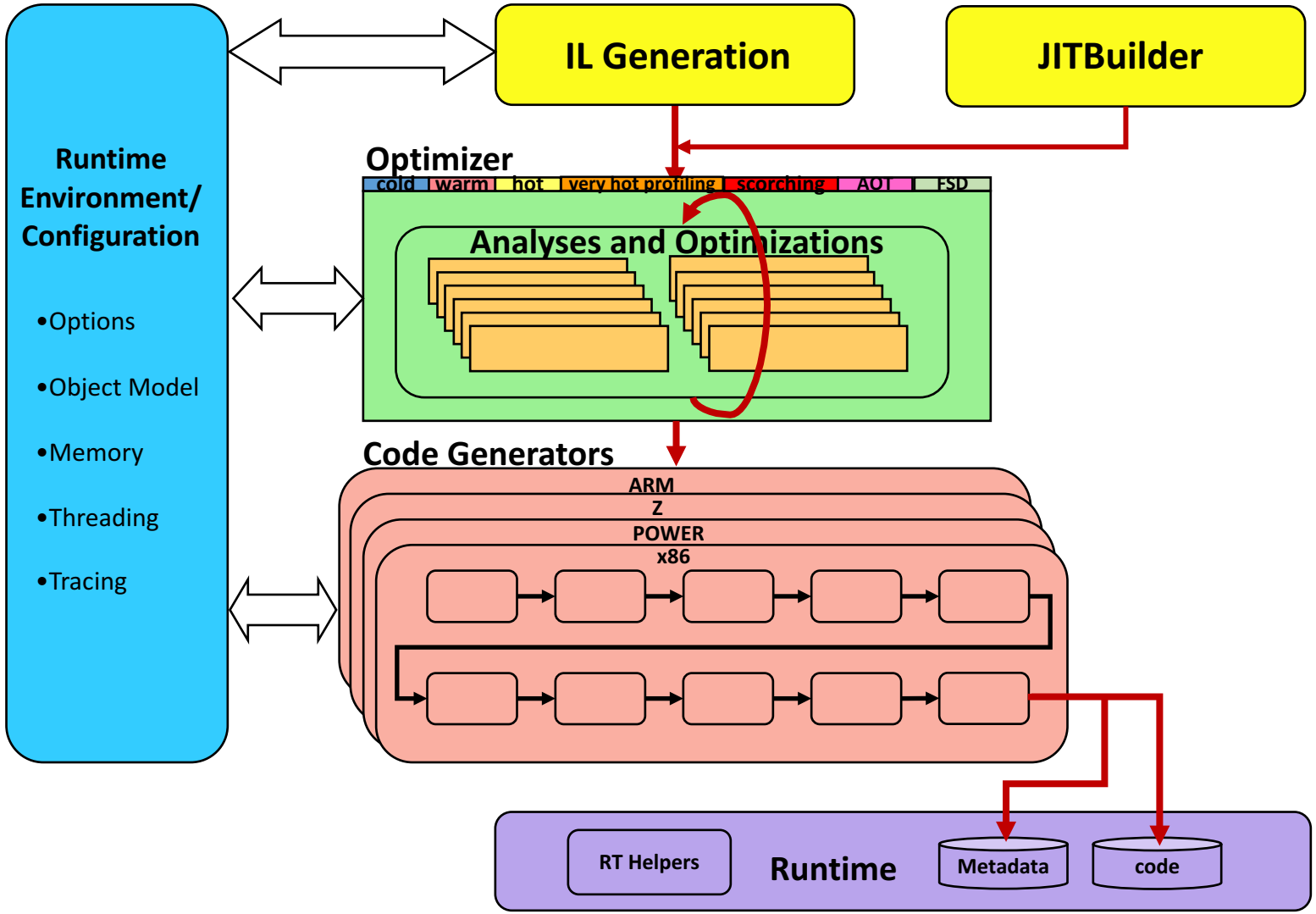
Eclipse OMR unboxing

port	platform abstraction (porting) library
thread	cross platform pthread-like threading library
vm	APIs to manage per-interpreter and per-thread contexts
gc	garbage collection framework for managed heaps
compiler	extensible compiler framework
jitbuilder	WIP project to simplify bring up for a new JIT compiler
omrtrace	library for publishing trace events for monitoring/diagnostics
omrsigcompat	signal handling compatibility library
example	demonstration code to show how a language runtime might consume OMR components, also used for testing
fvtest	language independent test framework built on the example glue so that components can be tested outside of a language runtime, uses Google Test 1.7 framework

+ a few others

~800KLOC at this point, more components coming!

Eclipse OMR compiler unboxing



- high-level optimization technology featuring classic compiler optimizations, loop optimizations, control and data flow analyses, and support data structures
- code generation technology with deep platform exploitation for x86 (i386 and x86-64), Power, System Z, and ARM (32-bit)
- expressive tracing and logging infrastructure for problem determination
- JitBuilder technology to simplify the effort to integrate a JIT compiler into an existing language interpreter
- a framework for constructing language-agnostic unit tests for compiler technology

Runtime integration: native IL generator

- Produce OMR IL directly from interpreted method “bytecodes”
- Deepest exploitation with the Eclipse OMR compiler technology

☺ Maximize performance and functionality

☺ Permits greatest specialization for a host runtime

☹ Steeper learning curve, esp. OMR compiler intermediate language

☹ You assume ownership of complexity

Runtime integration: JITBuilder

- Prototype interface to compiler technology
- Designed to simplify work to bootstrap a JIT compiler to generate native instructions for interpreted methods
- https://developer.ibm.com/open/2016/07/19/jitbuilder-library-and-eclipse-omr-just-in-time-compilers-made-easy

☺ Simple API to describe method prototypes and behaviours, types, and compile and dispatch functionality

☹ Sacrifice some performance and specialization for rapid integration of compiler technology

☹ Technology is a WIP, but continually improving

Eclipse OMR technology in action!

- IBM J9 Java 8, IBM J9 Java “Next”
  - Production consumer of Eclipse OMR technology
- Open J9
  - Open source implementation of Java virtual machine technology derived from the IBM J9 Java virtual machine
  - http://openj9.org
- Ruby
  - Integration of Eclipse OMR GC and compiler technology into CRuby VM
  - Ruby Core: http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/77461
  - https://github.com/rubyomr-preview/ruby.git
- Swift
  - Research project in conjunction with Matt Medland and Matt Zaleski from U of Toronto
  - Translate and compile Swift IL into native code using Eclipse OMR compiler and JitBuilder
- SOM++
  - Proof point of simple integration with JitBuilder technology (1 week of work yielded benchmark speedups of 2x-15x)
- Lua
  - Experimental compiler integration as proof point for technology and driving JitBuilder innovation

What's next?

- Improve onboarding experience
  - Documentation, designs, sample code, how-to blogs
- IBM actively contributing to the technology
- Community building
  - Grow the base of the Eclipse OMR project
  - Work with language runtime communities to foster broader adoption and drive improvements into the technology
- Engage with research communities
  - Research projects in runtime/compiler technologies based on Eclipse OMR
  - Eclipse OMR as an undergrad/graduate teaching platform for runtime technologies

Eclipse OMR

http://www.eclipse.org/omr
https://github.com/eclipse/omr
https://developer.ibm.com/open/omr/

Follow us @EclipseOMR

Users and contributors very welcome!

https://github.com/eclipse/omr/blob/master/CONTRIBUTING.md

Mark Stoodley
IBM OMR Project Lead
mstoodle@ca.ibm.com

Daryl Maier
IBM OMR Compiler Project Lead
maier@ca.ibm.com

Charlie Gracie
IBM OMR GC/VM Project Lead
cgracie@ca.ibm.com