Beyond the Coffee Cup:
Leveraging Java Runtime Technologies for Polyglot

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

IBM Runtime Technologies

About me…

- IBM Canada Lab in Toronto (-ish)
- Member of IBM Runtime Technologies team
- Compiler and runtime optimizations for 20 years
- Leading a not-so-secret project to open-source IBM compilation technology

Trademarks, Copyrights, Disclaimers

Additional Important Disclaimers

THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY.

WHILST EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED.

ALL PERFORMANCE DATA INCLUDED IN THIS PRESENTATION HAVE BEEN GATHERED IN A CONTROLLED ENVIRONMENT. YOUR OWN TEST RESULTS MAY VARY BASED ON HARDWARE, SOFTWARE OR INFRASTRUCTURE DIFFERENCES.

ALL DATA INCLUDED IN THIS PRESENTATION ARE MEANT TO BE USED ONLY AS A GUIDE.

IN ADDITION, THE INFORMATION CONTAINED IN THIS PRESENTATION IS BASED ON IBM’S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM, WITHOUT NOTICE.

IBM AND ITS AFFILIATED COMPANIES SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION.

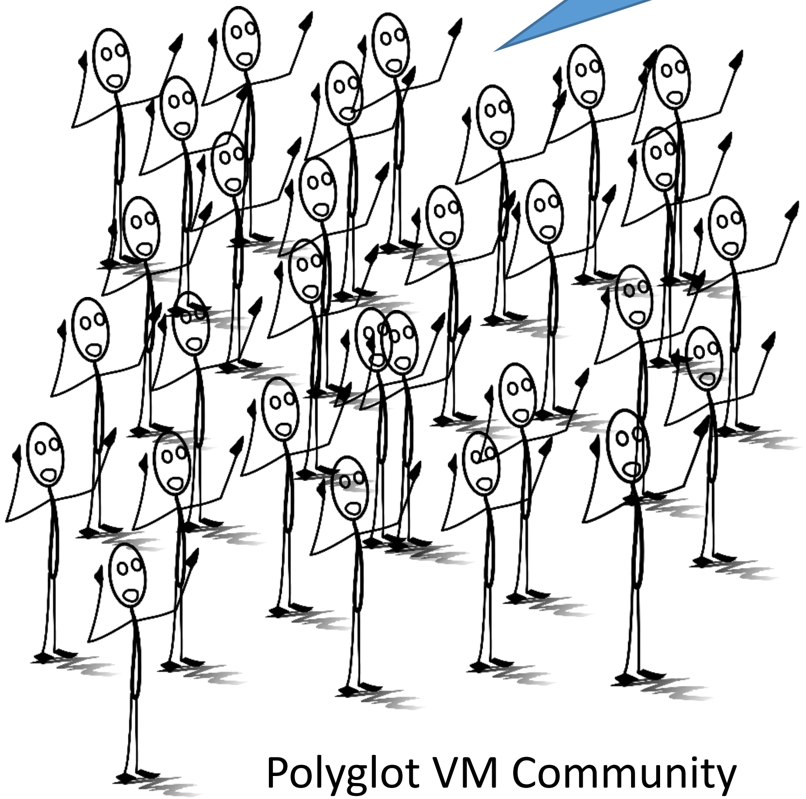
NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, OR SHALL HAVE THE EFFECT OF:

- CREATING ANY WARRANT OR REPRESENTATION FROM IBM, ITS AFFILIATED COMPANIES OR ITS OR THEIR SUPPLIERS AND/OR LICENSORS

IBM to open-source a runtime technology toolkit!

- Announced by Mark Stoodley at JVMLS 2015
- See the complete talk here.

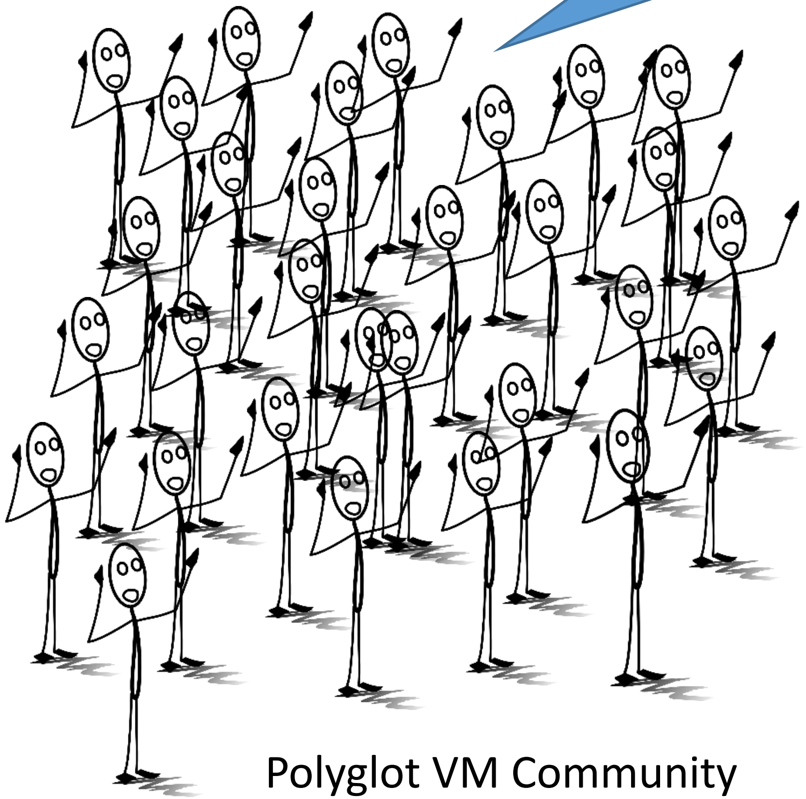
{groan}
Oh, great – *another* common runtime environment!



Polyglot VM Community

{cheer}

Woo hoo – a runtime toolkit that integrates with *my* VM!



Polyglot VM Community

Motivation for a runtime toolkit

- Experiment with leveraging investment in J9 Java VM technology in a way that facilitates integration of this technology into other VMs

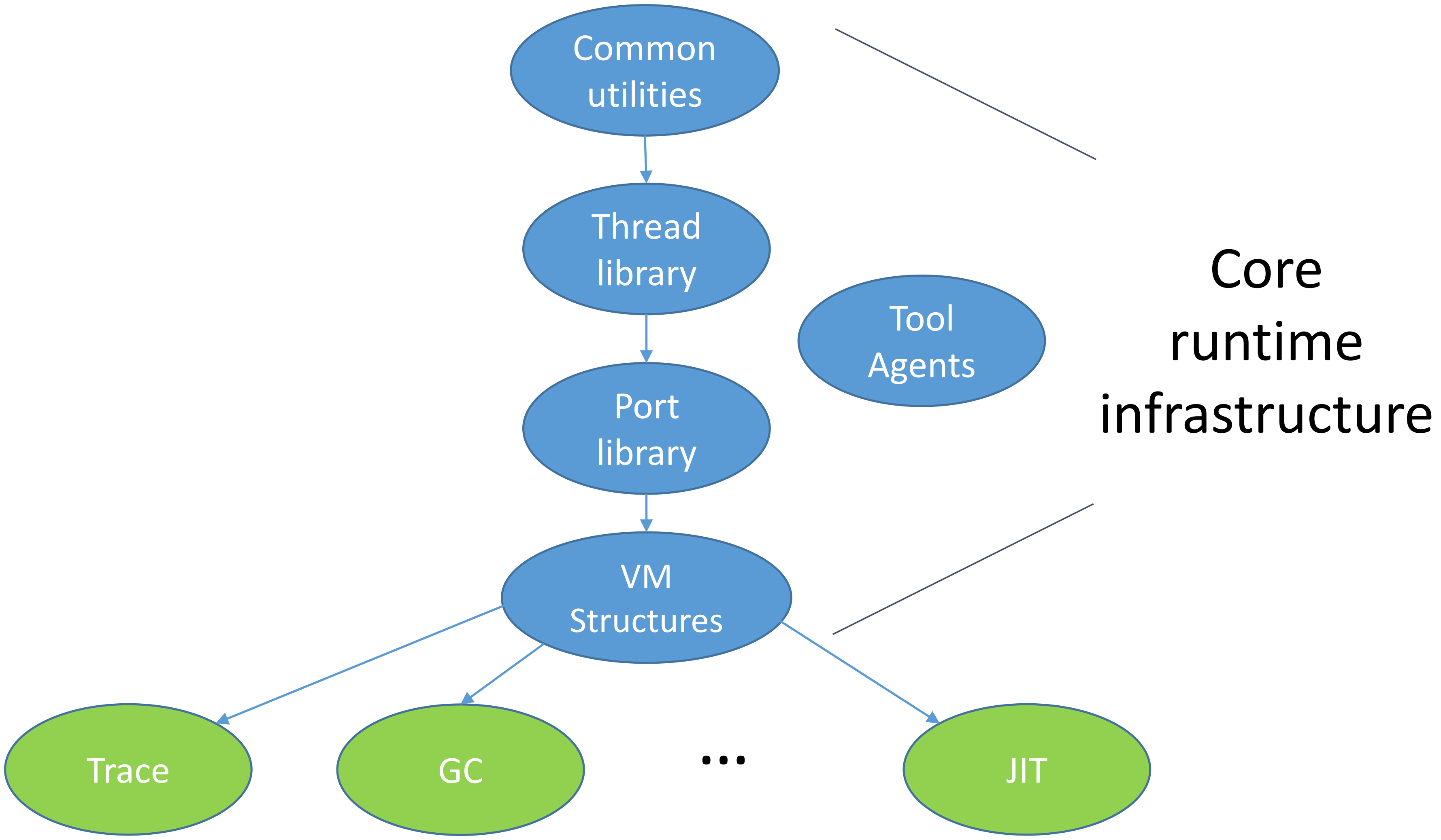
- Compatibility with existing runtimes and their communities
  - Lots of vibrant language communities can't tolerate disruptive technologies
  - Technology should be flexible enough to bend to the community rather than the other way around
  - Work with all the features that make that language great

- Simple consumption into existing runtimes

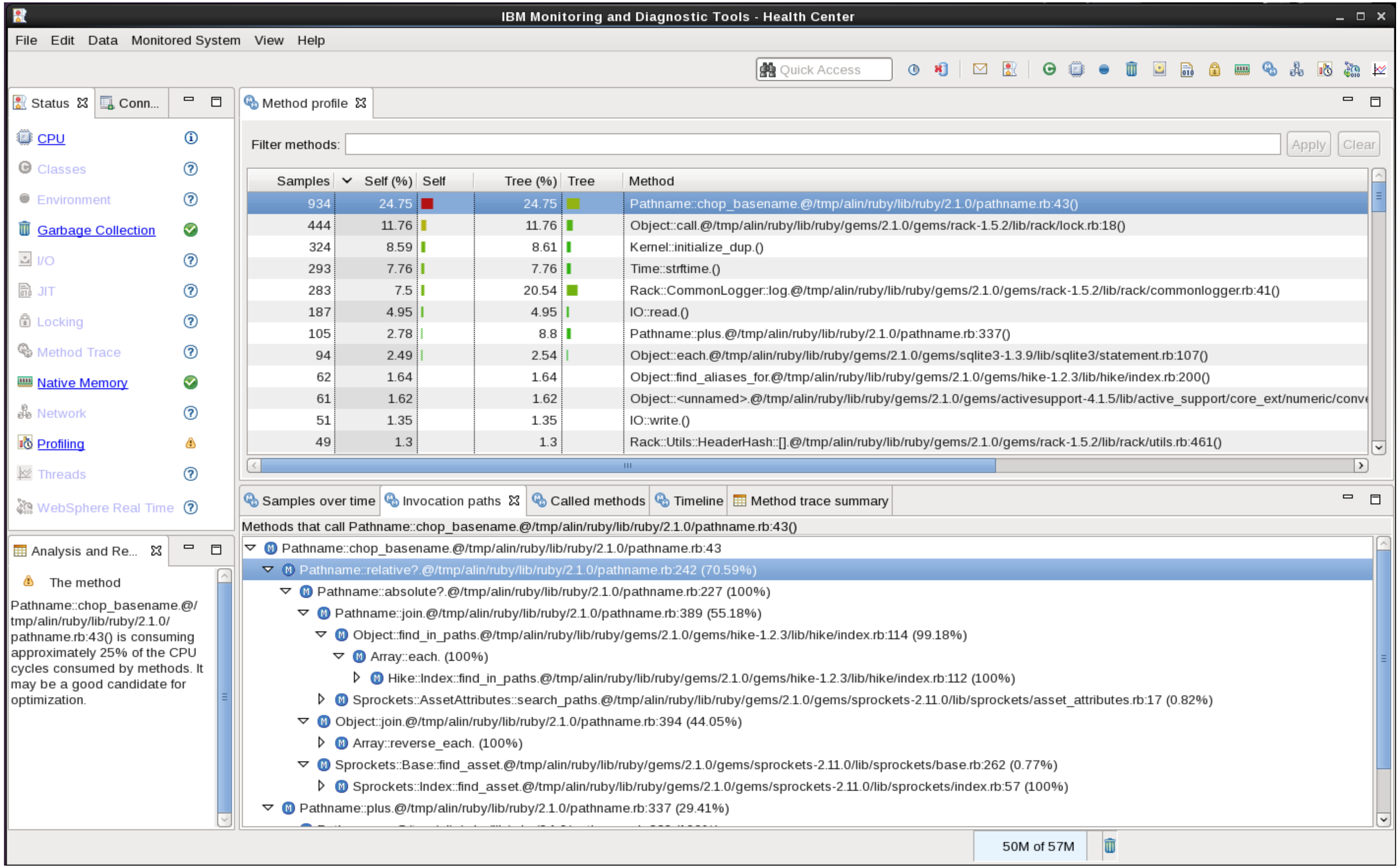
Unlock the “VM” from the J9 Java VM for polyglot

- Refactor several J9 components to create a language-agnostic toolkit designed for integration into language runtimes (including J9 JVM)
  - Memory allocator, thread library, platform port library, event hook framework, VM and application level trace engine, garbage collector, JIT compiler
- Experiments to bring capabilities from J9 to Ruby MRI, CPython, and CSOM
  - Integration by specializing toolkit components with runtime details from MRI, CPython, CSOM
  - Gauge promise of this approach
- Not a research project: our JDK product development team aggressively refactoring our VM, GC, and JIT technology
  - Shipped IBM JDK8 from snapshot of refactored code base
  - JDK9 development ongoing as we continue to experiment

Transplanting J9 capabilities to other runtimes



Method profiling for Ruby MRI: introduce tracepoint to feed call stacks samples to Health Center agent

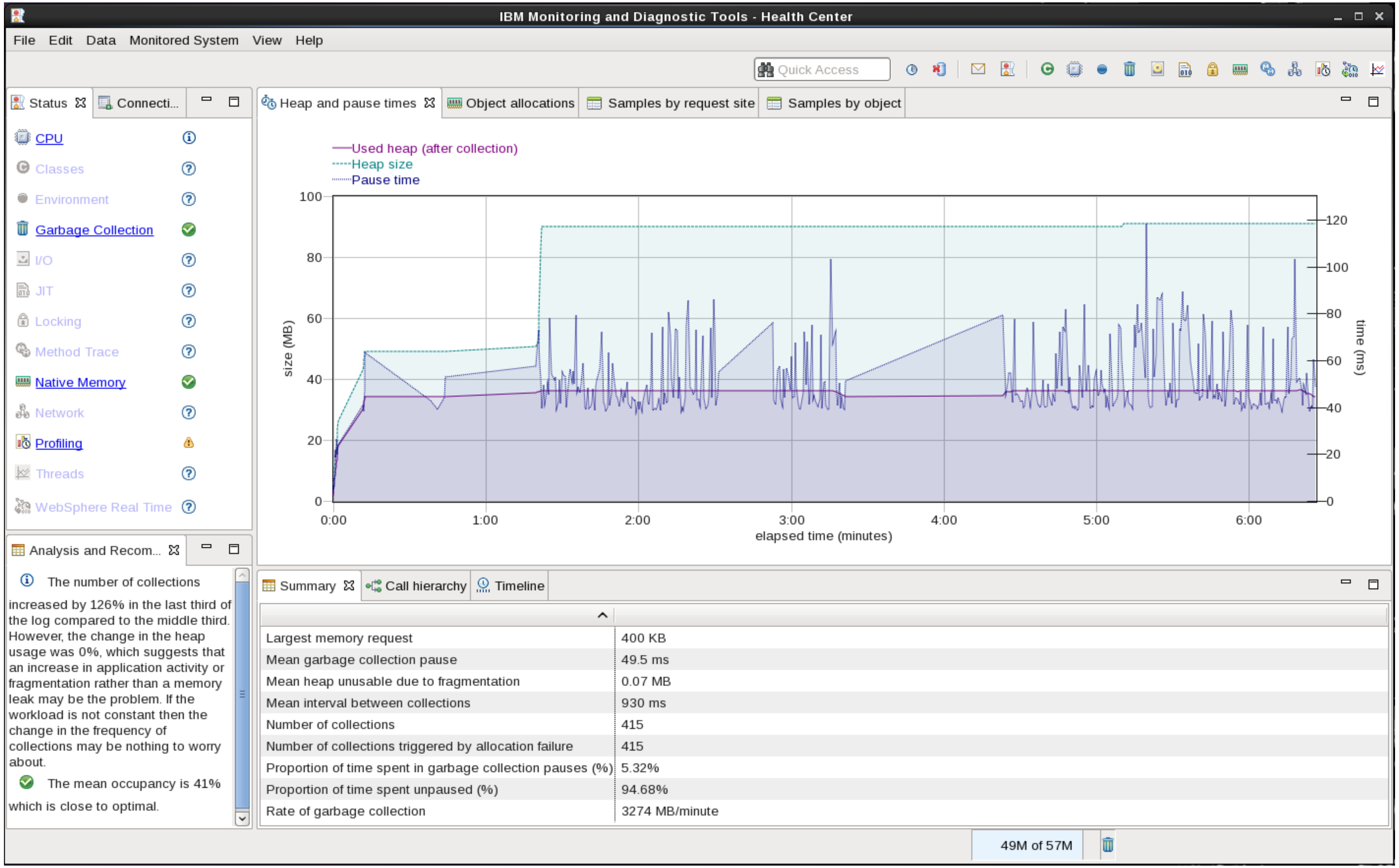


Scalable garbage collector integration

- Integrated GC into Ruby MRI
  - Type accurate, but used conservatively so extensions work as-is
  - MRI: can move off-heap native memory into manageable heap
- Proof point: verbose GC

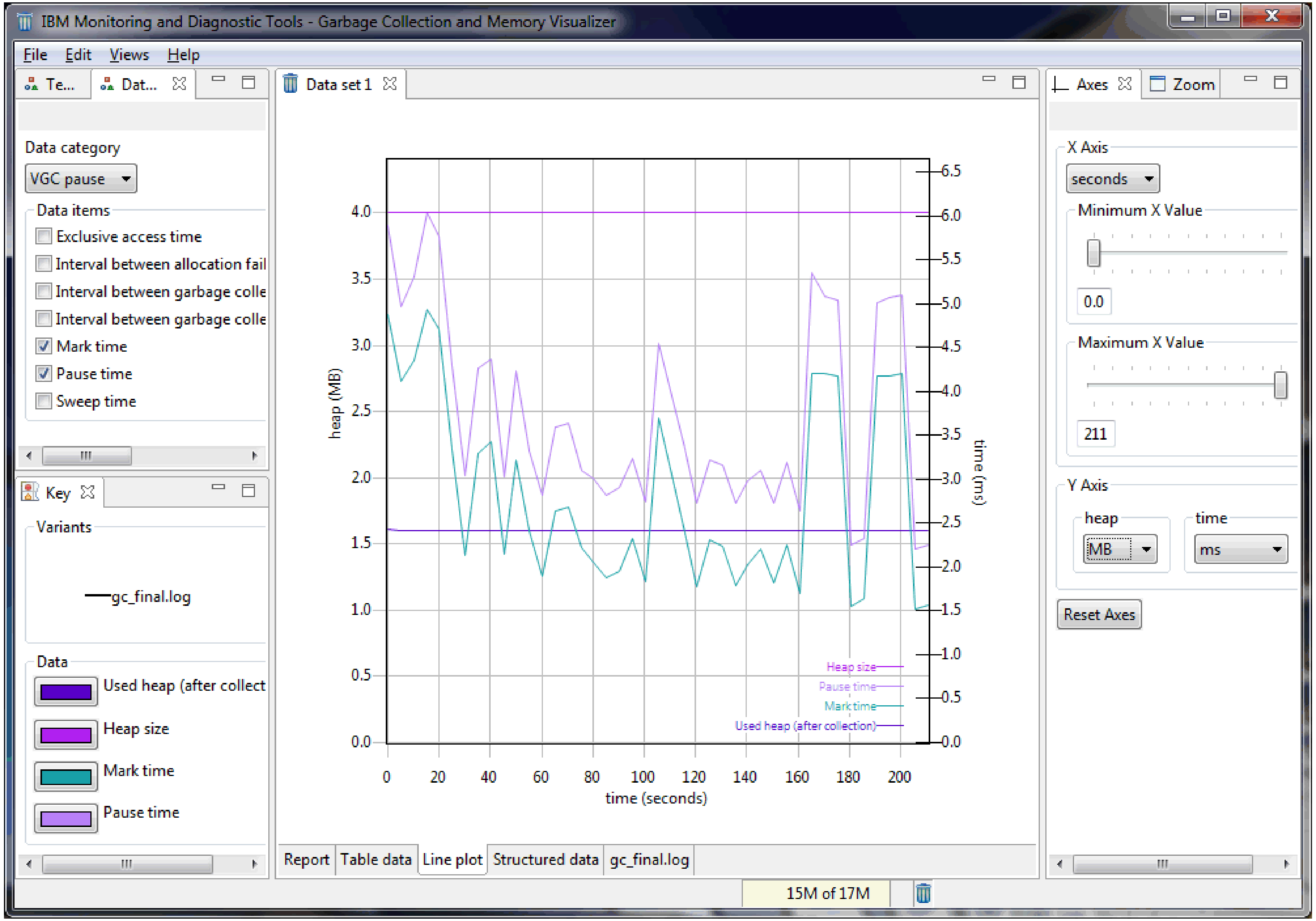

```
<cycle-start id="2" type="global" contextid="0" timestamp="2015-08-05T17:21:58.105" intervalms="5066.731" />
<gc-start id="3" type="global" contextid="2" timestamp="2015-08-05T17:21:58.105">
  <mem-info id="4" free="596848" total="4194304" percent="14">
    <mem type="tenure" free="596848" total="4194304" percent="14" />
  </mem-info>
</gc-start>
<allocation-stats totalBytes="3596216" >
  <allocated-bytes non-tlh="720016" tlh="2876200" />
</allocation-stats>
<gc-op id="5" type="mark" timems="4.881" contextid="2" timestamp="2015-08-05T17:21:58.110">
 <trace-info objectcount="8914" scancount="7208" scanbytes="288320" />
</gc-op>
<gc-op id="8" type="sweep" timems="0.688" contextid="2" timestamp="2015-08-05T17:21:58.111" />
<gc-end id="9" type="global" contextid="2" durationms="5.896" usertimems="7.999" systemtimems="1.999" timestamp="2015-08-05T17:21:58.111" activeThreads="2">
 <mem-info id="10" free="2508160" total="4194304" percent="59">
  <mem type="tenure" free="2508160" total="4194304" percent="59" micro-fragmented="297048" macro-fragmented="723458" />
 </mem-info>
</gc-end>
<cycle-end id="11" type="global" contextid="2" timestamp="2015-08-05T17:21:58.111" />
```

GC visualization for Ruby MRI via existing GC trace points feeding GC events to Health Center agent



IBM Runtime Technologies

Garbage Collection Memory Visualizer for Ruby MRI with zero changes to the tool



IBM Runtime Technologies

JIT integration

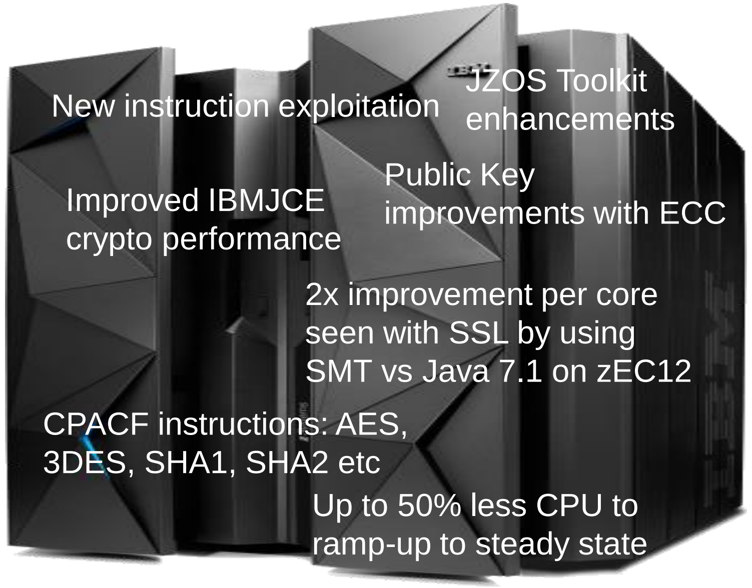
- Ruby MRI and CPython do not have JIT compilers
- Both environments are challenging for JIT compilers
  - Highly dynamic
  - Unmanaged direct use of internal data structures by extensions
  - Design choices in the runtimes themselves (e.g., setjmp/longjmp)
- Our effort to date has particular emphases
  - Compile native instructions for methods and blocks
  - Avoid big changes to how MRI/CPython works (to ease adoption)
  - Consistent behavior for compiled code vs. interpreted code
  - No restrictions on native code used by extension modules
  - No benchmark tuning or specials
- Compatibility success story: We can run Rails!
- Performance success story: 1.2x + on many Bench9k kernels on 3 architectures without tuning

Proof point: IBM JDK 8

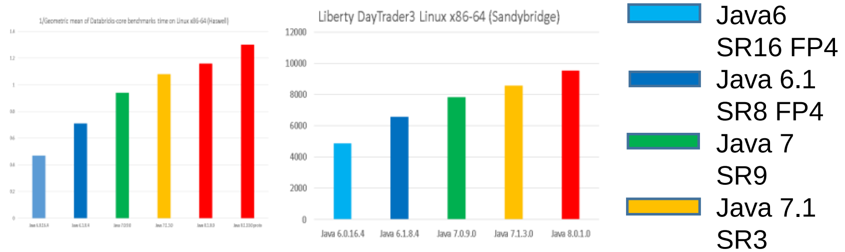
Liberty and tWAS Daytrader3 workload increases (15% on Linux Intel)

ORB optimizations can show 3x Improvements in real-World banking scenarios

Health Center enables remote Bluemix performance analysis



PERFORMANCE !



Apache Spark 1.4   Daytrader3 Linux Intel

Power

Up to 25% faster Liberty workload deployment

20% better ramp-up using Runtime Instrumentation

NVIDIA GPU support

IBMJCE crypto improvements on both AES and ECC encryption using hardware exploitation

Open community

- Create an open community of contribution based around a toolkit of components that can be used to build VMs for any language
- Efficient place for individuals, communities, and companies to safely collaborate on core VM infrastructure
- Enable everyone to focus more energy on innovation, not on building more wheels
- Build more robust core technology
  - Fix bugs once
  - Tested in many different scenarios
- Collection of best practices and shared learning
- Lower entry barrier for new languages and VM ideas
  - Test ideas faster and more reliably

What this talk is about…

- Mark's JVMLS talk focused on the "whys" and "whats", this talk will focus on the "hows"
- GC experience deep dive
  - CON7863: What's in an Object? Java Garbage Collection for the Polyglot (Charlie Gracie)
- JIT experience with refactoring the "VM" from the "JVM"

Are there enough re-usable components in a Java JIT to build a polyglot toolkit?

Sampling thread determines which methods spend the most time executing

Add “hot” methods to compilation queue for asynchronous compilation

Choose a method from the queue for compilation

Translate Java bytecode into compiler IR

Choose a tiered optimization strategy

Perform high-level classical, speculative, and Java-specific optimizations

Generate code (instructions from IR, register assignment, binary encoding, relocations)

Publish method metadata

Bind method into VM

Direct interpreter and JIT call sites to newly compiled body

Recompile at higher optimization level

Patch runtime assumption guards, polymorphic inline caches, monitor speculative optimizations

Java	
Sampling thread determines which methods spend the most time executing	Choose methods or blocks to compile based on invocation count
Add “hot” methods to compilation queue for asynchronous compilation	
Choose a method from the queue for compilation	Chosen method is compiled synchronously on application thread
Translate Java bytecode into compiler IR	Translate CRuby iseq to compiler IR
Choose a tiered optimization strategy	Choose a fixed optimization strategy
Perform high-level classical, speculative, and Java-specific optimizations	Perform high-level classical, speculative, and Ruby-specific optimizations
Generate code (instructions from IR, register assignment, binary encoding, relocations)	Generate code (instructions from IR, register assignment, binary encoding, relocations)
Publish method metadata	Publish method metadata
Bind method into VM	Bind method into VM
Direct interpreter and JIT call sites to newly compiled body	Direct interpreter and JIT call sites to newly compiled body
Recompile at higher optimization level	Patch runtime assumption guards, polymorphic inline caches, monitor speculative optimizations
Patch runtime assumption guards, polymorphic inline caches, monitor speculative optimizations	

	Java	
Select a method to compile	Sampling thread determines which methods spend the most time executing	Choose methods or blocks to compile based on invocation count
	Add “hot” methods to compilation queue for asynchronous compilation	
Compile a method with appropriate optimizations	Choose a method from the queue for compilation	Chosen method is compiled synchronously on application thread
	Translate Java bytecode into compiler IR	Translate CRuby iseq to compiler IR
	Choose a tiered optimization strategy	Choose a fixed optimization strategy
	Perform high-level classical, speculative, and Java-specific optimizations	Perform high-level classical, speculative, and Ruby-specific optimizations
	Generate code (instructions from IR, register assignment, binary encoding, relocations)	Generate code (instructions from IR, register assignment, binary encoding, relocations)
	Publish method metadata	Publish method metadata
	Bind method into VM	Bind method into VM
Dispatch to compiled body	Direct interpreter and JIT call sites to newly compiled body	Direct interpreter and JIT call sites to newly compiled body
Adapt compiled method to changing environment	Recompile at higher optimization level	Patch runtime assumption guards, polymorphic inline caches, monitor speculative optimizations
	Patch runtime assumption guards, polymorphic inline caches, monitor speculative optimizations	

Components of a compiler toolkit

Life Cycle	JIT startup and shutdown; initialization and destruction of resources (compilation threads, options processing, memory management, …)
Compilation Trigger	Logic for determining when a method should be compiled: runtime method sampling at execution, sample processing, count-and-send targets
Infrastructure	Supporting infrastructure for compilation: data structures (CFGs, blocks, trees, instructions, symbol reference tables, aliasing, code caches), tracing/logging, enabled feature processing
VM/JIT Interface	API between VM and JIT. Ask/answer questions about environment (e.g., lookup class or method info), language semantics (e.g., float association), available capabilities (e.g., GPU present), object model (e.g., array header size), configuration (e.g., GC policy), runtime helpers
Compilation	Intermediate representation; data types; optimization frameworks; classical, dynamic, and speculative optimizations; code generation; register assignment; instruction schedulers; binary encoders; code cache management; relocation processing; stack mapping
Method Dispatch	Dispatch to compiled method from interpreted and JITed call sites; call site fixup
Runtime Adaptation	Method meta data, runtime assumption managers, profiling, recompilation framework, code patching

IBM creating a toolkit of extensible compiler components

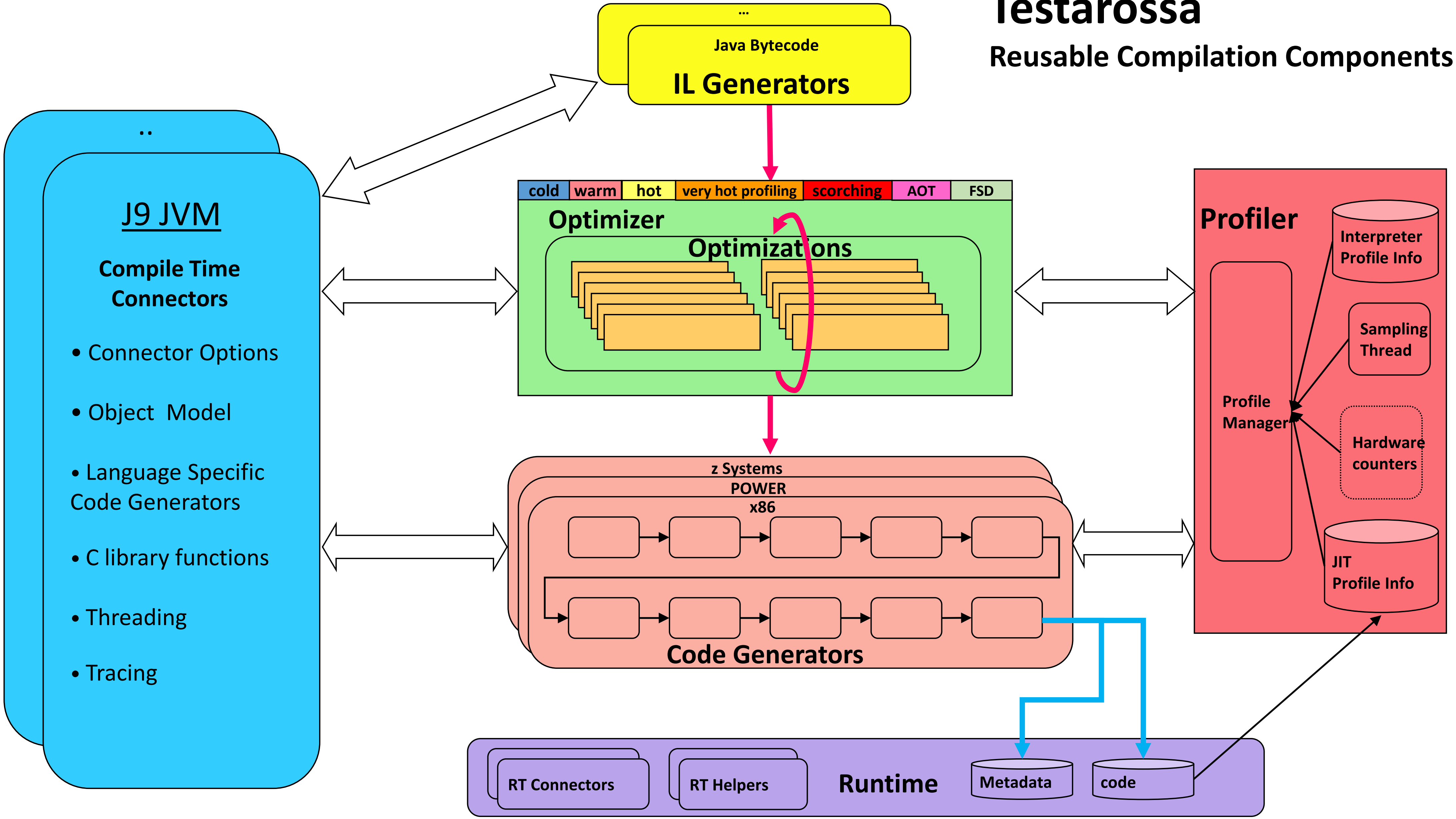
- Start with mature J9 Java just-in-time compiler (aka Testarossa, or TR)
- Isolate the Java parts from the generic parts
- Re-engineer source code to allow specialization
- TR technology has already proven to be highly-adaptable to different compilation uses
  - 8 different compiler technology products or use-cases
- Consumption model: clone VM; clone O/S JIT; make

Testarossa Java compiler technology

- Heritage is a dynamic JIT 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

Testarossa

Reusable Compilation Components

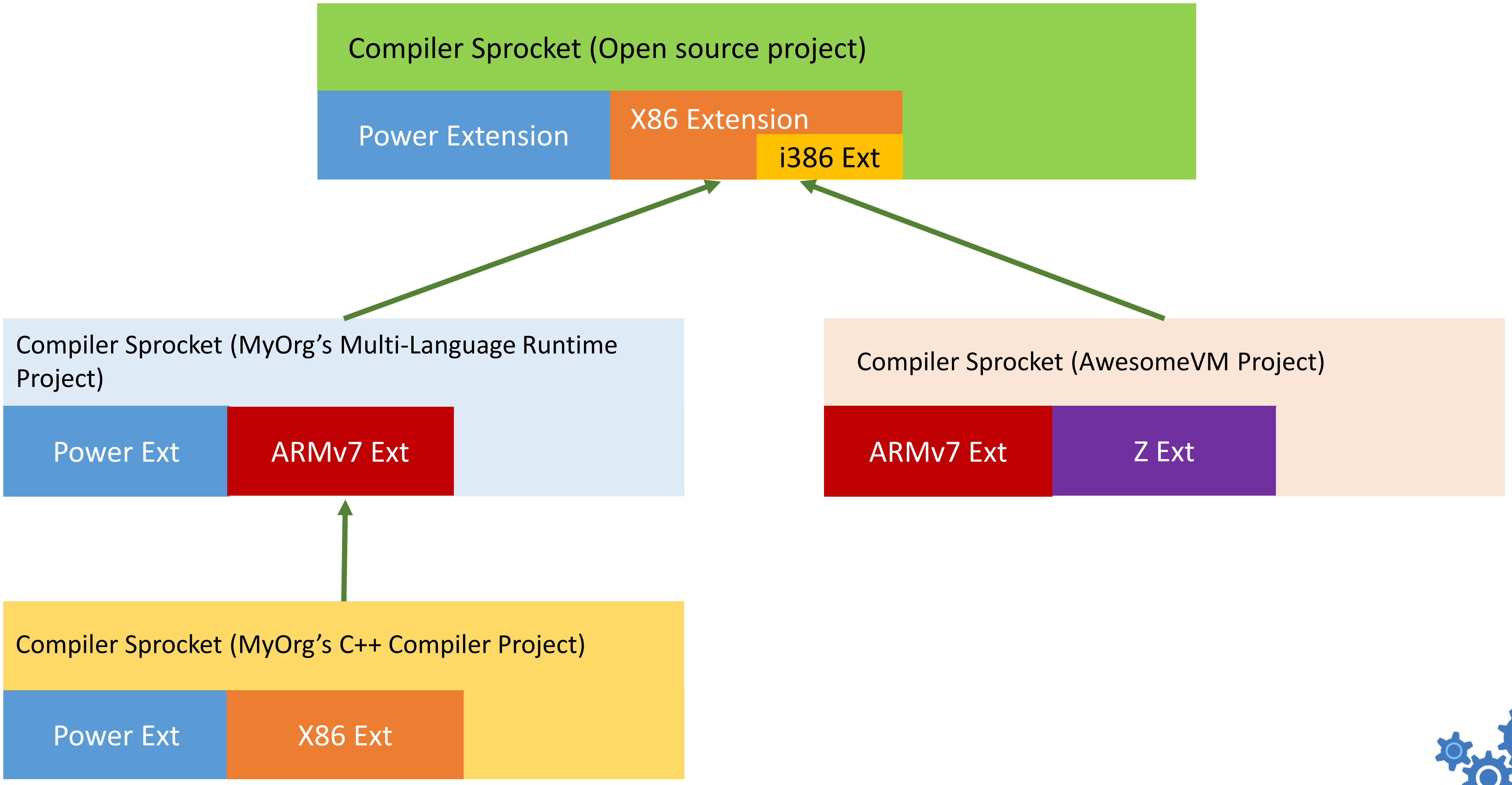


Specialization of compiler components

- Not straightforward how to distill generic functionality from core TR components and allow specialization for polyglot
- Two main axes of specialization
  - the kind of compiler you're trying to build
  - the processor architecture you're targeting
- Goals
  - Isolation of compiler features
  - Minimize impact to key compilation metrics: startup, compile-time, and footprint
  - Minimize future merge and integration costs of specializations (easy consumption)
  - Permit future extensibility

Engineering for extensibility

- Some reorganization is necessary to facilitate building an extensible model
  - Code and source files organized into an ordered hierarchy of "projects" each of which contains some specialization of compiler functionality
- Refactor core compiler technology classes into "extensible" C++ classes
- Follows a single-inheritance, composition model for specialization
- Static polymorphism for efficiency
- Makefile and include path determine which specializations to use


Compiler Sprocket (Open source project)
Power Extension
X86 Extension
i386 Ext
Compiler Sprocket (MyOrg's Multi-Language Runtime Project)
Power Ext
ARMv7 Ext
Compiler Sprocket (AwesomeVM Project)
ARMv7 Ext
Z Ext
Compiler Sprocket (MyOrg's C++ Compiler Project)
Power Ext
X86 Ext

Effective Sprocket compositions

Compiler Sprocket for i386 MyOrg C++ Build

- Compiler Sprocket (Open Source Project)
- Compiler Sprocket X86 Extension (O/S)
- Compiler Sprocket i386 Extension (O/S)
- Compiler Sprocket (MyOrg MLR Project)
- Compiler Sprocket (MyOrg C++ Project)
- Compiler Sprocket X86 Extension (MyOrg C++)

Compiler Sprocket

Compiler Sprocket for Power MyOrg C++ Build

- Compiler Sprocket (Open Source Project)
- Compiler Sprocket Power Extension (O/S)
- Compiler Sprocket (MyOrg MLR Project)
- Compiler Sprocket Power Ext (MyOrg MLR)
- Compiler Sprocket (MyOrg C++ Project)
- Compiler Sprocket Power Ext (MyOrg C++)

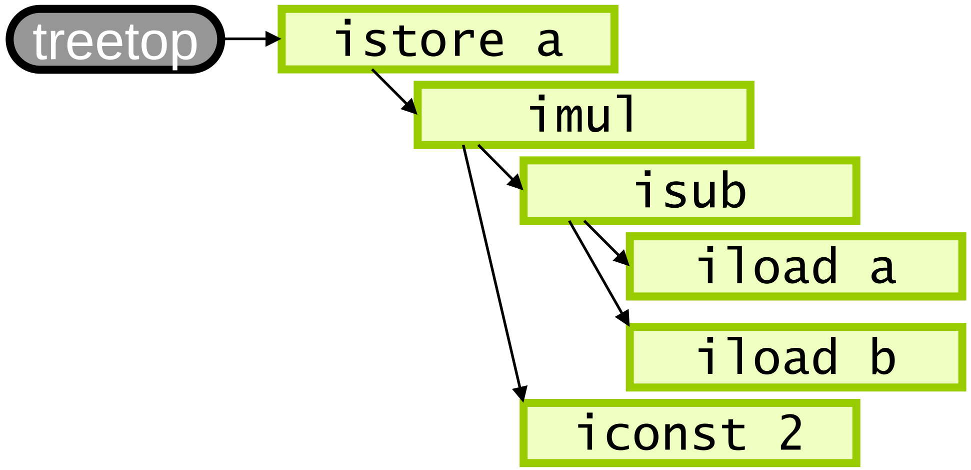
Compiler Sprocket

Testarossa intermediate language

- TR uses a tree-based intermediate representation, where the “tree” represents a single expression or statement

Java Bytecode

```
iload a
iload b
isub
bipush 2
imul
istore a
```



- Internal node opcodes and datatypes can be extended for different projects and architectures

IL generators

- Produce IL that can be consumed by JIT technology
- Highly specialized to the environment VM
  - Depend largely on the input: bytecodes, instructions, etc.
  - Consume and represent symbol information
- Each are generally independent, but shares IL construction utilities for tree, node, and block creation

High-level IL optimizer

- Complete suite of classical and Java-specific optimizations
- Platform-neutral, each optimization consumes and produces IL
- Flexible configuration allows optimization strategies to be constructed
  - Spend compile-time where and when it makes sense for each VM

- Most challenging to adapt for polyglot because some optimizations provide VM- or architecture-specific specializations that are entwined with analysis and transformation phases

- Make it easier to adapt by separating policy from mechanism in each optimization, and then specializing both as necessary

Minimal compiler toolkit consumption model

1. Clone your favorite VM
2. Clone the open-source compiler toolkit
3. Implement a VM-specific extension to the compilation trigger and method dispatch interface
4. Implement an IL generator for your VM, perhaps on a subset of all possible “bytecodes” or “instructions”
5. Implement VM-specific extensions as needed to
   - Core technology (e.g., IL nodes, opcodes, code generation, instructions)
   - VM <-> JIT interface (e.g., Q&A about object model, name lookup, bytecode info)
6. Modify VM makefiles and include paths to integrate JIT technology
7. make

Implement richer support for all inputs and VM features once basic hookup is completed!

Longer term challenges

- Must optimize at a higher semantic level for maximum performance
  - Optimizing for compatibility has a point of diminishing returns
  - Can't just optimize the connective tissue in the interpreter
  - e.g., arithmetic operations
- Increase use of method meta data
  - Don't waste execution time maintaining interpreter state
  - Leverage on-stack replacement to focus on what matters
- Build in fork tolerance
  - Forking is the means by which some VMs achieve parallelism
  - Difficult to manage compilation efficiently across multiple processes

The road to open-source

- Early results are very promising for all our technologies
- Next steps
  - Some components in the toolkit are ready (Port, Thread, Trace, GC)
    - Compiler technology needs more time
  - Balance refactoring work against developing proof points
  - Engage with runtime communities and partners
- I invite your feedback on our open proposal, our compiler toolkit, or your interest in becoming involved

Daryl Maier	Mark Stoodley	John Duimovich
O/S JIT Lead	O/S Project Lead	CTO, IBM Runtime Technologies
maier@ca.ibm.com	mstoodle@ca.ibm.com	John_Duimovich@ca.ibm.com

Acknowledgements

- Clip art on slides 6 and 7 was sourced from
  - https://openclipart.org/image/2400px/svg_to_png/92065/paro-AL-calling.png