OpenJDK on Morello
Port Status and Initial Lessons

Andy Nisbet, Tim Hartley and Mikel Luján

with input from many other team members

Soteria & MOJO projects funded by UK DSbD programme
Spoiler Alert

- Most memory safe languages (e.g. Java, Javascript, Ruby, ...) execute on managed runtime environments
- Managed runtime environments tend to be written in C/C++
  - JVMs are a key part of the Morello software ecosystem
- We have managed to port interpreted OpenJDK to Morello
  - Next steps are JIT & garbage collection
- Unlike other managed languages & runtimes (e.g. Javascript), the Java APIs expose longs as pointers
- Porting to Morello requires modifications to core Java classes as well JVM internals
Java Virtual Machine (JVM)

Java program → javac → Java bytecode (JVM instructions)

*.java → source code → *.class → binary files

JVM Internals: >1.2M LOC

Compilation System
- Bytecode Interpreter
- Assembler
- Just-in-time Compiler

Interpreted or Compiled method
- Machine code
- Object map

Stack

Heap

Garbage Collector

JNI native code

Graph showing performance over time with time points labeled as Startup, Compilation, Completion.

Legend:
- C1 Compiled and Profilerd
- C2 Compiled and Profilerd

Interpreted and Profilerd

Performance
Roadmap for the talk

- Overview of attacks & exploits on Java/JVMs
- JVM porting strategy to Morello
- Preliminary Performance Results
- Status/development plans for JVM ports
- Future Work/Questions
Threat Model Guided By CVEs

Compilation System

- Bytecode Interpreter
- Assembler
- Just-in-time Compiler

Interpreted or Compiled method

- Machine code
- Object map

Stack, Heap, Garbage Collector

JNI native code

JNI & Java code libraries misuses and flaws
Threat Model Guided By CVEs

JIT Compilation:
Code injection & IR manipulation

Compilation System
- Bytecode Interpreter
- Assembler
- Just-in-time Compiler

Interpreted or Compiled method
- Machine code
- Object map
- Stack
- Heap

Garbage Collector

JIT processes a lowered intermediate representation (IR)

JNI native code

Java bytecode

Machine code

Performance

Start-up
Compilation
Completion
Time

C1 Compiled and Non-Profiled
C2 Compiled and Profiled
Threat Model Guided By CVEs

Malformed inputs *.class and program inputs target

JVM internals
- Class Loading
- Type checking
- Object de/serialisation

Compilation System
- Bytecode Interpreter
- Assembler
- Just-in-time Compiler

Interpreted or Compiled method
- Machine code
- Object map

Stack

Heap

JNI native code

Garbage Collector
Threat Model Guided By CVEs

- JNI/Java code libraries misuse/flaws
  - Especially that related to XML/JSON processing
- JVM internals
  - JIT compilation
  - ClassLoading/type verification
  - Object serialization/deserialization
Outline of JIT Compilation Threats

- Long history of attacks on JavaScript
  - JIT is disabled: Microsoft Edge security & iOS16 Lockdown modes
  - Code is injected via JIT/heap spraying
  - Control flow is directed into JIT-ted code at an altered PC
    - Altering the PC delivers a different instruction sequence
    - One that can be used to construct malicious actions
    - Typically involves taking control of the execution stack
  - Data only attacks corrupting a JIT’s intermediate representation
    - Cause malicious code to be “legally” generated
Protecting the JVM with Capabilities

Options & stages in protecting a JVM

- Morello pure capabilities - referential & spatial memory safety for free
- Temporal safety - requires revoke/invalidate capabilities
- Compartments

Compilation System
- Bytecode Interpreter
- Assembler
- Just-in-time Compiler

Interpreted or Compiled method
- Machine code
- Object map

Stack
Heap
JNI native code
Garbage Collection
Porting a JVM to (Morello) a new CPU

Target spatial memory safety using pure capability C64 mode

- Interpreter, then GC, then JIT
- Morello has A64 and C64 purecap execution modes
- C64: object layout changes, longs cannot represent addresses

Compilation System

Bytecode Interpreter
Assembler
Just-in-time Compiler

Zero assembler: libffi

A64: sizeof(long) == sizeof(void*)
C64: sizeof(long) != sizeof(void*)
C64 modifies the A64 ISA

TemplateTable interpreter machine code is generated at JVM startup. Needs an assembler
Zero Assembler Interpreter

Entire JVM runs in purecap C64
- Fixed JVM assumptions
- Java API issues with longs
- Spatial memory protection

Zero Assembler Bytecode Interpreter

Broke object layout

Machine code address fields/arguments in Java API must use capabilities

EpsilonGC no GC
TemplateInterpreter

Faster and enables profiling to trigger JIT compilation

JVM code runs in A64
- Generates interpreter’s instructions
- Tests interpreter usage of C64 ISA
- Manages A64/C64 transitions

Morello Assembler (A64)

TemplateTable Bytecode Interpreter machine code (C64)

Limited spatial protection until it becomes fully purecap

JIT compilation can be added
OpenJDK17 Initial Port Steps & Status

Compilation System
- Bytecode Interpreter
- Assembler
- Just-in-time Compiler

Garbage Collector

JVM Overview
- Interpreted or Compiled method
  - Machine code
  - Object map
- Heap
- Stack

JNI

Morello assembler A64
OpenJDK17 Initial Port Steps & Status

Compilation System
- Bytecode Interpreter
- Assembler
- Just-in-time Compiler

JVM Overview
- Interpreted or Compiled method
  - Machine code
  - Object map
- Garbage Collector
- Heap
- Stack

Morello assembler A64
TemplateTable interpreter mixed A64/C64
OpenJDK17 Initial Port Steps & Status

Compilation
System
Bytecode
Interpreter
Assembler
Just-in-time
Compiler
Garbage Collector

Interpreted or
Compiled method
Machine code
Object map
Heap
Stack

JVM Overview

Zero assembler interpreter
fully C64 purecap

TemplateTable interpreter mixed A64/C64

Morello assembler A64

EpsilonGC C64
OpenJDK17 Initial Port Steps & Status

Compilation System
- Bytecode Interpreter
- Assembler
- Just-in-time Compiler
- Garbage Collector

JVM Overview
- Interpreted or Compiled method
  - Machine code
  - Object map

Garbage Collector

Zero assembler interpreter fully C64 purecap

Morello assembler A64

TemplateTable interpreter mixed A64/C64

EpsilonGC C64

SerialGC C64

JNI
Preliminary JDK17 SciMark Composite Results

Preliminary means performance has not been optimised, and thus results are expected to be worst case

- Zero purecap assembler interpreter performance is 50% of the equivalent AArch64 JVM
- Template interpreter hybrid A64/C64 is 13x faster than AArch64 Zero assembler interpreter
- Template interpreter AArch64 is 20x faster than the AArch64 Zero assembler interpreter
- Template interpreter hybrid A64/C64 performance is 66% of the equivalent AArch64 JVM
Recap: OpenJDK Port

- Significant effort to get here
- Preliminary relative performance of AArch64 vs. Morello
- Demonstrated benefits of the templateInterpreter
- SciMark benchmark - subset of SpecJVM
OpenJDK17 Next Steps

- Compilation System
  - Bytecode Interpreter
  - Assembler
  - Just-in-time Compiler
  - Garbage Collector

- Interpreted or Compiled method
  - Machine code
  - Object map
  - Heap
  - Stack

- JVM Overview
  - Morello assembler C64
  - TemplateTable interpreter fully C64 purecap execution
  - C1 JIT compiler
  - Serial & Epsilon GC
  - Zero assembler interpreter fully C64 purecap
MOJO: OpenJDK17 Next Steps

Compilation System
- Bytecode Interpreter
- Assembler
- Just-in-time Compiler
- Garbage Collector

Interpreted or Compiled method:
- Machine code
- Object map
- Heap
- Stack

G1 JIT compiler
C1 JIT compiler
Graal JIT compiler
Morello assembler C64
Serial & Epsilon GC
TemplateTable interpreter fully C64 purecap execution
Zero assembler interpreter fully C64 purecap
Takeaways for Porting Managed Languages

- Problems if managed language does not encapsulate machine code addresses (Java longs in API core classes)
- Hybrid A64/C64 execution needs detailed knowledge of codebase
- Moving to C64 execution can “break everything”
  - Object layout changes, field offset calculations
  - C64 code pointers have LSB set (problems in assembly stubs)
    - Usage of LSBs for VM housekeeping potentially problematic
- Necessary to port in incremental steps
  - Make individual VM components C64 aware
  - Use capabilities derived from the A64 default-data capability
### Ongoing/Future Work

- Improving OpenJDK port functionality/usage of capabilities
  - Supporting Guest languages JavaScript/Python on Java
- Improving security
  - Fine-grained constraints for base/limit of capabilities
  - Temporal safety
  - Compartmentalization models/APIs JNI/JIT compilers ...
- Evaluate threat weaknesses in JVMs
- Exploit attack injection techniques for specific classes using modified JVMs
Soteria & MOJO team

Soteria & MOJO projects: much more than just OpenJDK

Andy Nisbet, Tim Hartley, Kunjian Song, David Jackson, Nikos Foutris, John Mawer, Guillermo Callaghan, Cosmic Gorgovan, Igor Wodiany, Lucas Cordeiro, Christos Kotselidis, Pierre Olivier, Giles Reger, Konstantin Korovin, Mikel Lujan: University of Manchester

Hannah Cushworth, Adam Dad, Eloise Slater, Philip Wilson, Hui Ling Wong, Christopher Woodham, James Mercer: The Hut Group

Avi Shaked, Thomas Melham: Oxford University

Thanks to Andrew Dinn (RedHat) for advice on OpenJDK internals and attack injections

Soteria & MOJO projects are funded via Innovate UK, as part of the DSbD programme
Questions?