ORACLE

GraalVM: State of AArch64

Tom Shull
Senior Researcher
Oracle Labs
April 2, 2022

About Me

- Joined GraalVM Team in June 2020
  - Work in Zürich Office
  - Previously worked at Arm Ltd. on Graal Support

- Lead AArch64 development for GraalVM Compiler and Native Image
  - Also work on general-purpose Native Image improvements

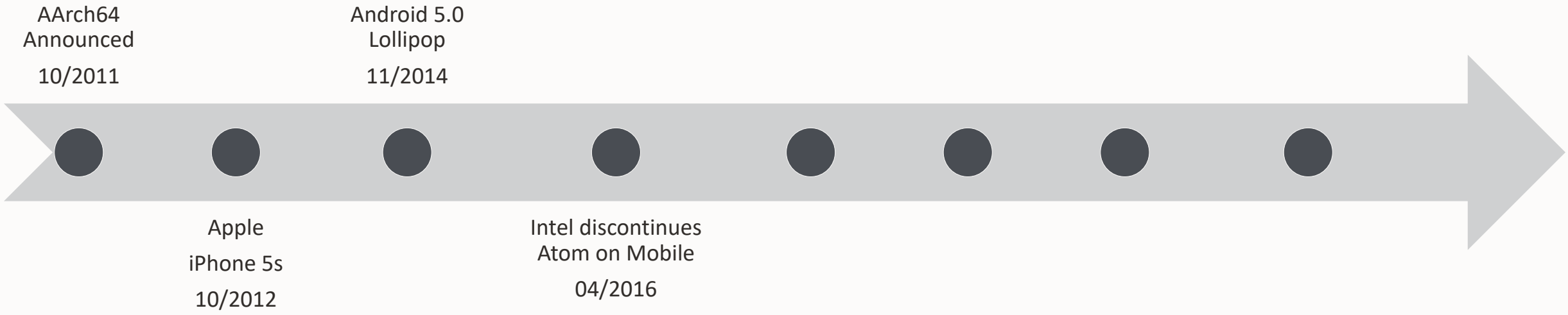
Talk Outline

- Quick Description of AArch64 ISA
- High-Level Overview of Graal's AArch64 Support
- Demo: Setting up Development Environment
- Walkthrough: Navigating AArch64 Codebase
- Graal AArch64 Performance Numbers
- Getting rid of Graal's AMD64isms
- Future Improvements

State of AArch64 Ecosystem

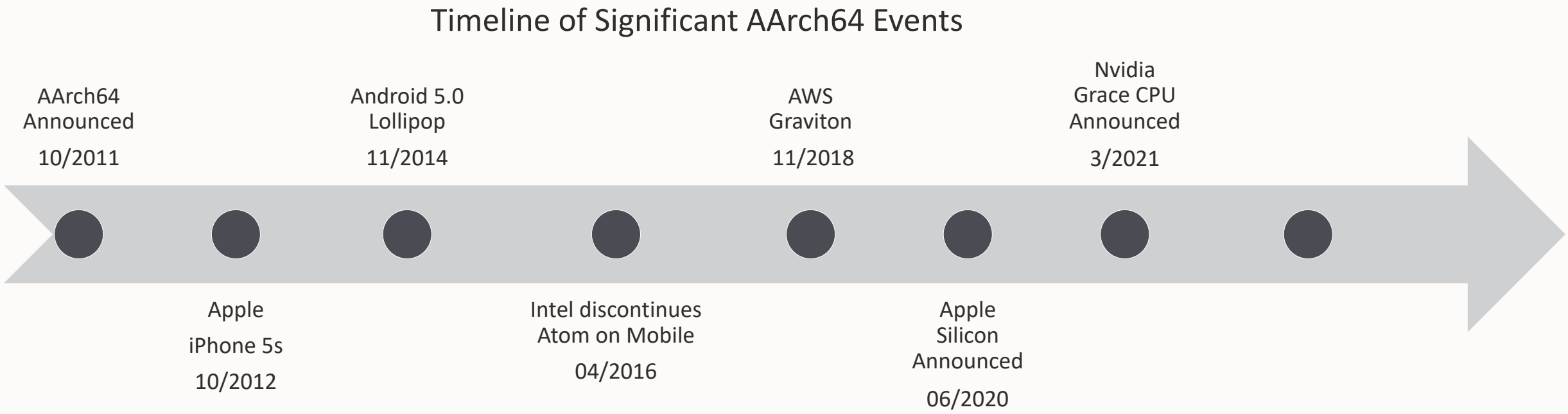
- Note: I am only talking about AArch64 (ARM64), Arm's 64-bit instruction set
  - NOT referring to AArch32, Arm's 32-bit architecture (ARM/A32 & Thumb/T32)

Timeline of Significant AArch64 Events



State of AArch64 Ecosystem

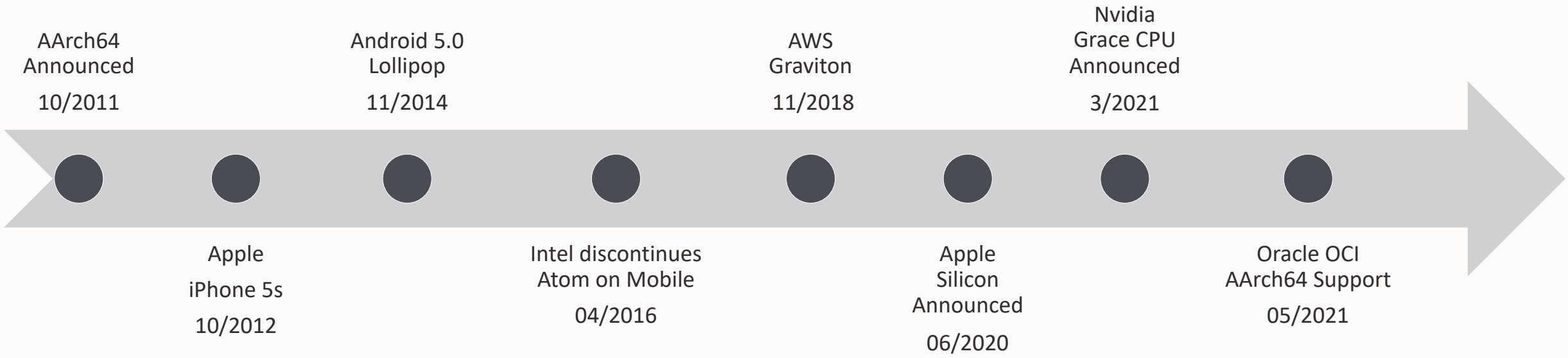
- Note: I am only talking about AArch64 (ARM64), Arm's 64-bit instruction set
  - NOT referring to AArch32, Arm's 32-bit architecture (ARM/A32 & Thumb/T32)



State of AArch64 Ecosystem

- Note: I am only talking about AArch64 (ARM64), Arm's 64-bit instruction set
  - NOT referring to AArch32, Arm's 32-bit architecture (ARM/A32 & Thumb/T32)

Timeline of Significant AArch64 Events



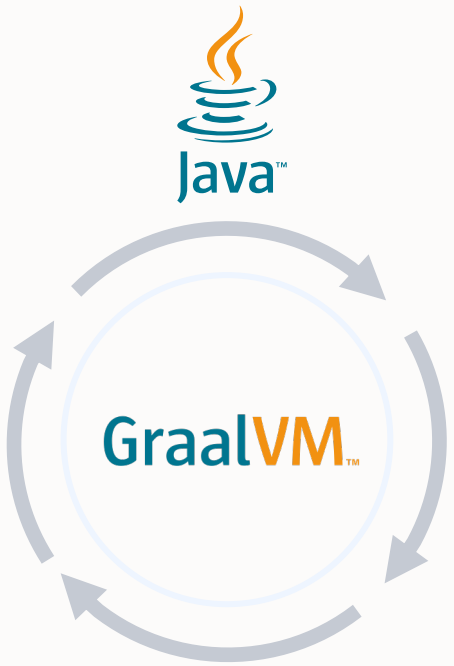
AArch64 ISA Overview

- 64-bit architecture with floating point & vector support

- 31 general purpose registers
  - Also has zero (zr) and stack pointer (sp) registers
- 32 floating-point / vector registers
  - ASIMD (NEON) vector length is 128-bits

- All instructions are 32-bits (4-bytes) long
  - Different from AMD64 (AMD64 is variable length)
  - Impact: less space for immediate encodings in AArch64 instructions

GraalVM Ecosystem

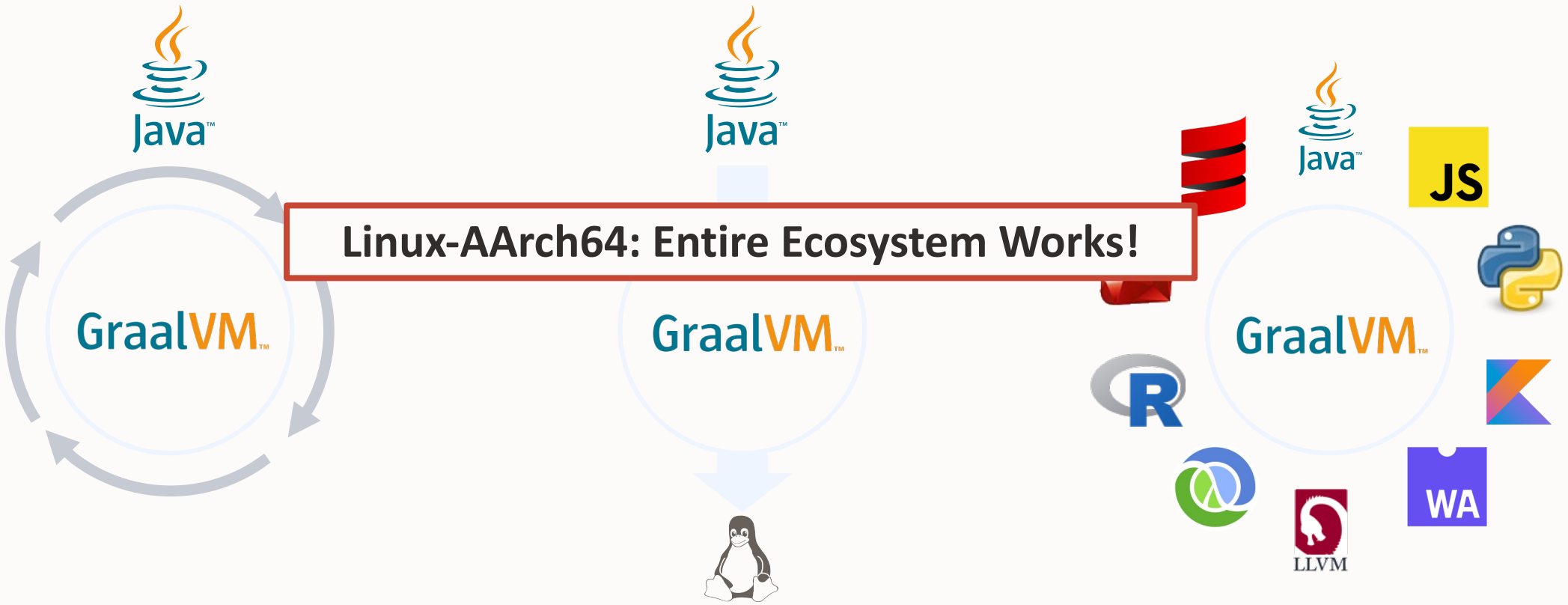


High-performance optimizing Just-in-Time (JIT) compiler

Ahead-of-Time (AOT) “Native Image” generator

Multi-language support

GraalVM Ecosystem



High-performance optimizing Just-in-Time (JIT) compiler

Ahead-of-Time (AOT) “Native Image” generator

Multi-language support

GraalVM Ecosystem

High-performance optimizing Just-in-Time (JIT) compiler

Ahead-of-Time (AOT) “Native Image” generator

Multi-language support

Building & Developing Graal on AArch64

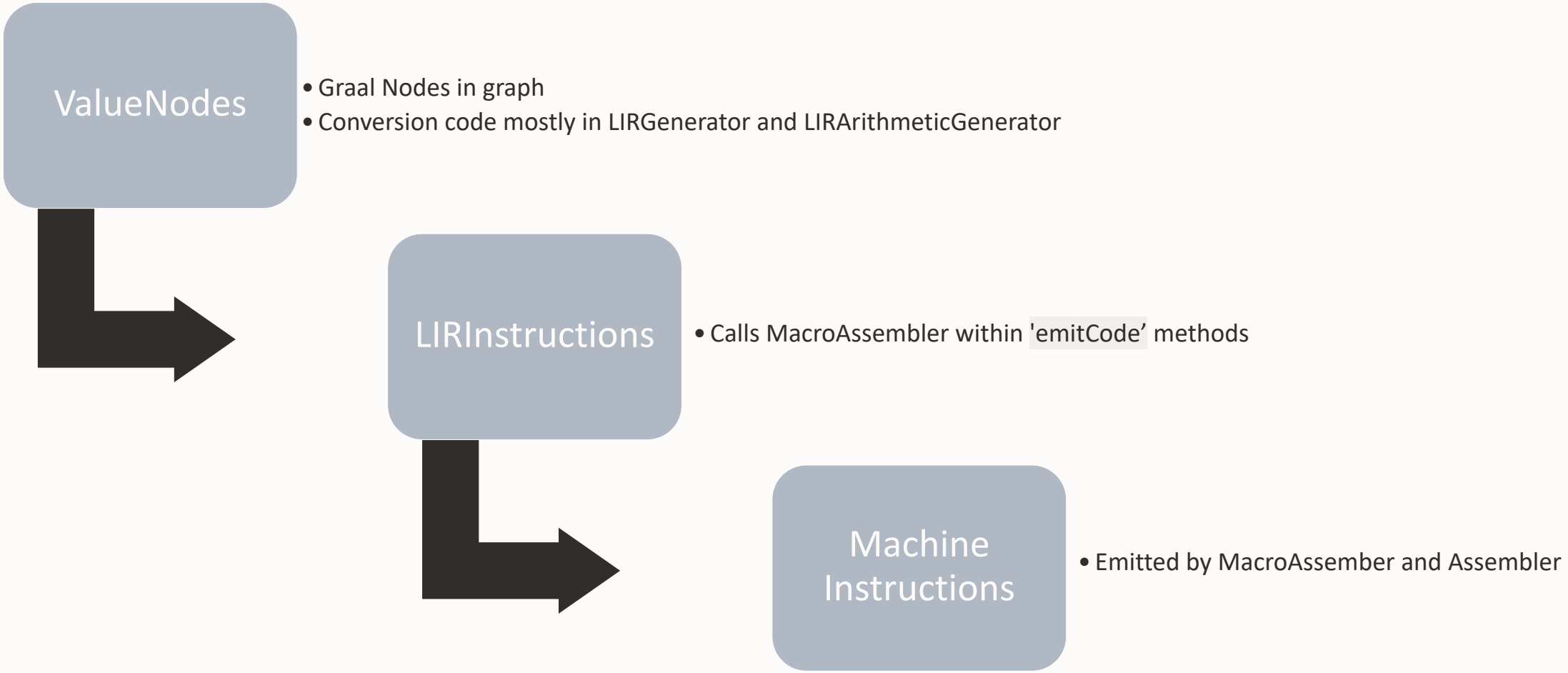
Build process is same as AMD64:

1. Clone graal repo
2. Clone mx repo and add to PATH
3. Get of copy of jvmci-enabled openjdk and set as JAVA_HOME
   - from labs-openjdk releases (labs 11) (labs 17)
   - Alternative: use `mx fetch-jdk --to [download_dir]`
4. Use `mx [--env setup or --dy imports] build`

Developing:

- Intellij, Eclipse work on both Linux and MacOS
  - use `mx ideinit` | `mx intellijinit` | `mx eclipseinit` to setup IDE project files

Code Generation Process



Code Walkthrough References

- AddNode's generate
- AArch64ArithmeticLIRGenerator's emitAdd
- AArch64ArithmeticOp Add Enum
- Calling assembler add instruction
- Assembler Add instruction

Navigating AArch64 Codebase – Important Packages

Main AArch64 packages:

- org.graalvm.compiler.asm.aarch64: Calls for emitting AArch64 instructions
- org.graalvm.compiler.lir.aarch64: AArch64 LIRInstructions
- org.graalvm.compiler.core.aarch64: Logic for lowering Graal Nodes to LIRInstructions
- org.graalvm.compiler.hotspot.aarch64: HotSpot specific LIRInstruction generation hooks
- com.oracle.svm.core.graal.aarch64: Native Image specific LIRInstruction generation hooks

Navigating AArch64 Codebase – Important Files

Emitting AArch64 Instructions

- AArch64Assembler
- AArch64MacroAssembler
- AArch64ASIMDAssembler
- AArch64ASIMDMacroAssembler

AArch64 LIR Instructions

- AArch64ArithmeticOp
- AArch64Move
- AArch64AtomicMove
- AArch64Call
- AArch64 Compare
- AArch64ControlFlow

Node → LIRInstruction Lowering

- AArch64LIRGenerator
- AArch64ArithmeticLIRGenerator
- AArch64HotSpotBackend
- AArch64HotSpotLIRGenerator
- SubstrateAArch64Backend

Comparing AArch64 vs AMD64 Optimizations

- Optimizations: equivalent*
  - Same passes run within HighTier, MidTier, and LowTier phases
  - AArch64 (EE) also runs same Loop and Linear Vectorization passes
    - AMD64 can (potentially) benefit more due to bigger vector widths
  - *AMD64 has more peephole optimizations (via LoweringProviders)
  - *Optimization heuristics are currently tuned for AMD64

- Intrinsics
  - (Custom Nodes for efficient execution of select Java Methods)
  - AArch64 is similar to AMD64
    - See UnimplementedGraalIntrinsics
    - Also compare AArch64GraphBuilderPlugins and AMD64GraphBuilderPlugins

Performance Comparisons

- Two sets of comparisons:
  - Intel Ice Lake vs Ampere Altra
  - Ampere Altra vs Apple M1 Pro

Company	Intel	Ampere	Apple
Core Name	Ice Lake	Altra	M1 Pro
Base Frequency	3 GHz	3 GHz	~2-3 GHz
# cores (threads)	18 (36)	80 (80)	10 (10)

Comparing Intel Ice Lake and Ampere Altra

Raw times [ms] running Renaissance 0.11 on 36 cores for Ice Lake and Altra

	Ice Lake: OracleJDK 11.0.11	Ice Lake: GraalVM EE 21.1 (JDK11)	Altra: OracleJDK 11.0.11	Altra: GraalVM EE 21.1 (JDK 11)
Geomean	**1665 ms**	**1023 ms**	**2201 ms**	**1466 ms**

GraalVM 21.1 (JDK 11) Speedup over OracleJDK 11.0.11

Ice Lake	Altra
1.628x	1.501x

Raw Numbers

- benchmark suite: renaissase 0.11
- values are execution time [ms]
- both executions are limited to 36 logical cores
  - Ice Lake has 36 logical cores, 18 physical cores (2-way SMT)

Benchmark	Ice Lake: OracleJDK 11.0.11	Ice Lake: GraalVM EE 21.1 (JDK11)	Altra: OracleJDK 11.0.11	Altra: GraalVM EE 21.1 (JDK11)
akka-uct	9891	7903	14543	14815
als	2251	1257	3273	2290
chi-square	890	627	1237	754
dec-tree	1285	789	1639	1160
dotty	1849	1540	2216	1892
fj-kmeans	847	847	806	970
future-genetic	1611	1527	2262	1872
gauss-mix	851	517	1026	798
log-regression	1477	1424	1553	1511
mnemonics	4208	2020	4444	2567
movie-lens	5317	4271	27307	26107
naive-bayes	214	95	484	122
neo4j-analytics	6919	2185	4556	2766
page-rank	1125	984	1819	1801
par-mnemonics	3627	1611	4041	2171
philosophers	4693	4133	6426	5844
reactors	13636	10815	15792	15323
rx-scrabble	231	196	261	233
scala-doku	7593	2002	11050	3343
scala-kmeans	469	164	477	198
scala-stm-bench7	1077	934	1279	1262
scrabble	120	30	158	24
geomean	1665	1023	2201	1466

Comparing Altra and M1 Pro

Raw times [ms] running Renaissance 0.14 on 10 cores for Altra and M1 Pro

	Altra GraalVM EE 22.0 (JDK 17)	M1 Pro GraalVM EE 22.0 (JDK 17)
Geomean	1624 ms	958 ms

M1 Pro Speedup over Altra	1.695x

Raw Numbers

- benchmark suite: renaissase 0.14
- values are execution time [ms]
- both executions are limited to 10 logical cores
  - M1 Pro has 10 logical cores
    - 8 high-performance & 2 efficiency

benchmark	Altra GraalVM EE 22.0 (JDK 17)	M1 Pro GraalVM EE 22.0 (JDK 17)
akka-uct	6809	5415
als	1919	1137
chi-square	522	269
dec-tree	825	479
dotty	1161	455
finagle-chirper	4098	6942
finagle-http	7191	2647
fj-kmeans	2577	1367
future-genetic	2008	1567
gauss-mix	1010	611
log-regression	1140	560
mnemonics	2104	1096
movie-lens	11620	7219
naive-bayes	1336	621
page-rank	3441	2113
par-mnemonics	1936	949
philosophers	2135	1249
reactors	13195	6104
rx-scrabble	227	106
scala-doku	1508	1487
scala-kmeans	197	116
scala-stm-bench7	1182	938
scrabble	124	59
geomean	1624	958

Considering Perf/$

- Prior comparisons are using equal core counts
- Altra has 80 cores per node vs 18 (36 threads) for Intel
- AArch64 machines are generally cheaper in the cloud
  - 1 Altra vCPU = $0.01/hr vs 1 Ice Lake vCPU = $0.027/hr (OCI Bare Metal Instance)
- **Takeaway:** Today AArch64 has very high Perf/$

For more details: blog post

Deep Dive: handling AMD64isms in Codebase

Main AArch64 differences from AMD64:

1. All instructions are 32-bits long
   - Limits # of immediates which can be encoded in instructions
2. AArch64 only requires a “relaxed” memory consistency model
   - Allows instructions to execute differently than program order

Dangerous AMD64isms: PC relative branching & Patching

- AMD64 PC-relative instructions can access ± 2GB
  - address displacement immediate is 4bytes

- AArch64 instructions cannot access ± 2GB
  - branches: ± 128 MB
  - loads: ± 1 MB

- Workaround: use a sequence of instructions to reach ± 4GB
  - branches: adrp, add, br [reg]
  - loads: adrp, ldr [reg + #imm]

Dangerous AMD64isms: PC relative branching & Patching

- Problem: AArch64 native image code was directly ported from AMD64
  - Didn’t consider access ranges

- Impact:
  - accidentally jump to wrong spot
  - access wrong data
  - overwrite instructions incorrectly

- Fix: Replace with correct `adrp (add br|ldr)` sequences
  - Add patching logic to correctly update this code at link time
  - Add guaranteed to code to ensure impossible patch fails at code generation time

Dangerous AMD64isms: Memory Model Differences

- AMD64 defines a TSO (total store order) memory model
  - Load ⇒ Load, Load ⇒ Store, Store ⇒ Store must be in order
  - Store ⇏ Load: load can execute before store

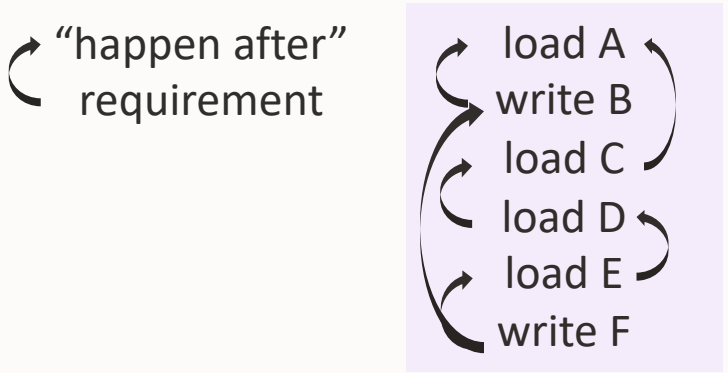
- AArch64 defines a relaxed memory model
  - accesses to same address must be in order
  - all other ordering requirements must be enforced special instructions

Dangerous AMD64isms: Memory Model Differences

Generated Code

load A
write B
load C
load D
load E
write F

AMD64 Ordering Requirements



loads C, D & E can happen in any order relative to write B

AArch64 Ordering Requirements

load A
write B
load C
load D
load E
write F

No ordering requirements! Memory accesses can happen in any order

- Because of relaxed memory model, application races are more likely to cause problems on AArch64
- Applications which “work fine” on AMD64 (but are actually buggy) may not work on AArch64

Reducing volatile overhead in AArch64 via acquire and release

- AArch64 also has instructions with **acquire** & **release** semantics
  - In addition to fences/barriers for LOAD_LOAD, LOAD_STORE, etc...

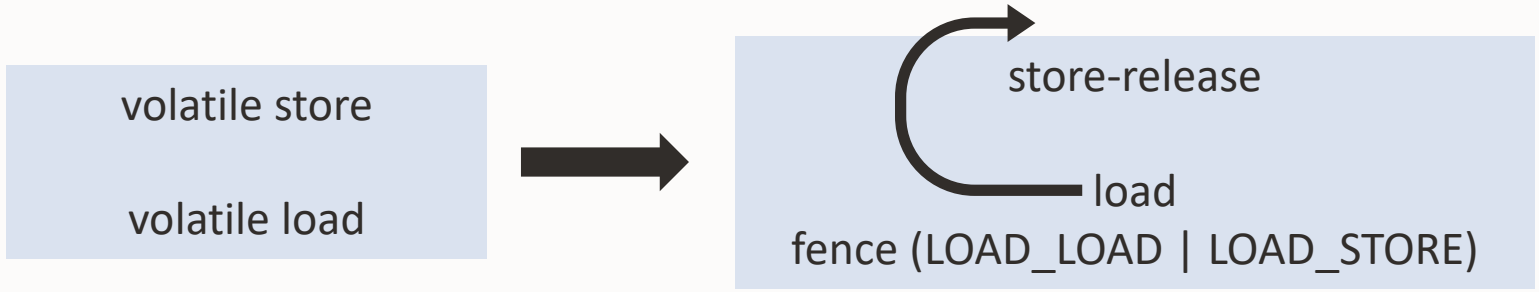
- **acquire**: enforces ordering requirement on all **subsequent** memory accesses
  - memory instructions before acquire can still be reordered
  - acquire also must be ordered with all prior releases

- **release**: enforces ordering requirement on all **prior** memory accesses
  - memory instructions after release can still be reordered
  - release also must be ordered with all subsequent acquires

- Ideal fit for implementing Java's volatile accesses

Oil & Vinegar: Fences & Acquire/Release

- Graal started to partially use Acquire/Release in 2019 for volatile accesses
  - Github PR 1772: rework handling of volatile accesses on aarch64
- Problem: Fences & Acquire/Release don't mix



- Early 2022, switched to using Acquire/Release for all volatile accesses on AArch64

Future Improvements

- Add support for SVE (Scalable Vector Extension)
  - SVE is AArch64's new vector instruction family

- Better Optimize AArch64 Memory Instructions
  - Improve address generation
  - Further minimize need for fences

Other Miscellaneous:

- Tune Compiler Optimization Heuristics for AArch64
- G1GC for native-image EE
- Improved debugging support

Takeaways

- Linux AArch64 runs entire GraalVM ecosystem
  - Darwin AArch64 support will soon follow
- Using Graal on AArch64 offers performance advantages over Hotspot’s C2
  - Geomean 1.5x speedups on renaissance 0.11
- AArch64 is a high priority for GraalVM
  - We are working to make it rival GraalVM on AMD64

Thank you