从移植到深耕：
Chromium V8 for RISC-V 的 2025 技术进展

PLCT Lab 邱吉
qiuji@iscas.ac.cn

创新求实 · 永竞一流

Overview

- About me
- Background: why V8 is important
- Retrospect: porting V8 for RISC-V
- Key point: TurboFan Backend
- Daily maintaining
- Current status

About me: Ji Qiu

- Ph.D. of Computer Architecture from ICT, CAS (2007-2013)
- Technical director of PLCT Lab, ISCAS (2019-): Compilers, Runtimes, and Simulators (https://github.com/plctlab)
- RISC-V ambassador , Chair of HelloLLVM （https://hellollvm.org/ ） （2023-）
- 18 years experience in CPU architecture and HW/SW co-design, system software development
- maintainer of V8 for RISC-V

My ISA journey: MIPS, LoongArch, ARM, RISC-V

2002

DLX

Designer	John L. Hennessy and David A. Patterson
Bits	32-bit
Introduced	1994
Version	1.0
Design	RISC
Type	Load–store
Encoding	Fixed
Branching	Condition register
Endianness	Bi-endian
Extensions	None, but MDMX & MIPS-3D could be used
Open	Yes
	Registers
General-purpose	32 (R0=0)
Floating point	32 (paired DP for 32-bit)

2006

MIPS

Designer	MIPS Technologies, Imagination Technologies
Bits	64-bit (32 → 64)
Introduced	1985; 39 years ago
Version	MIPS32/64 Release 6 (2014)
Design	RISC
Type	Load–store
Encoding	Fixed
Branching	Compare and branch, with a 1 instruction delay after the branching condition check
Endianness	Bi
Page size	4 KB
Extensions	MDMX, MIPS-3D
Open	Partly. The R16000 processor has been on the market for more than 20 years and as such cannot be subject to patent claims. Therefore, the R16000 and older processors are fully open.
	Registers
General-purpose	32
Floating point	32

2007

2010

2015

RISC-V®

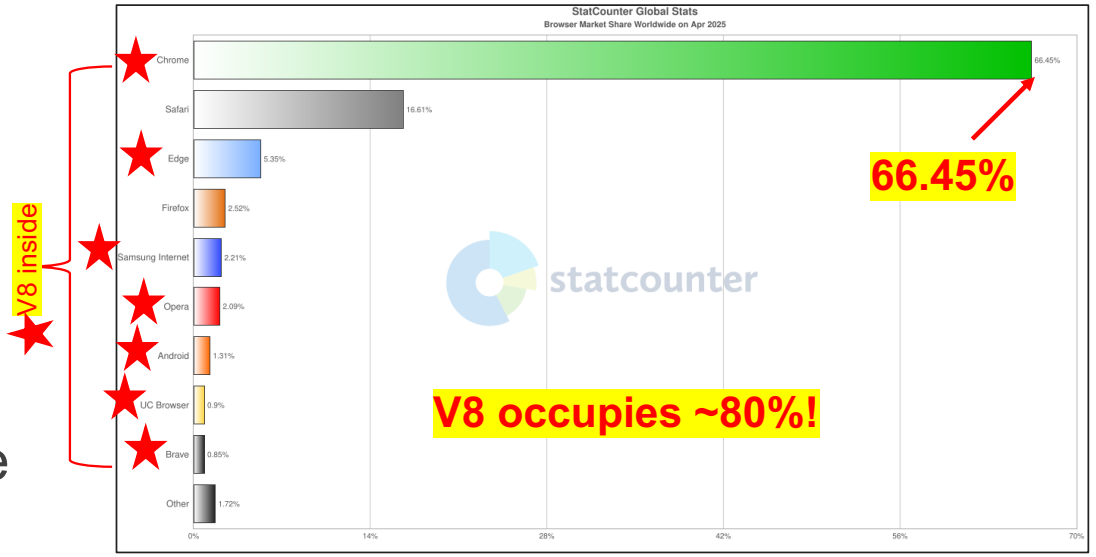
2018

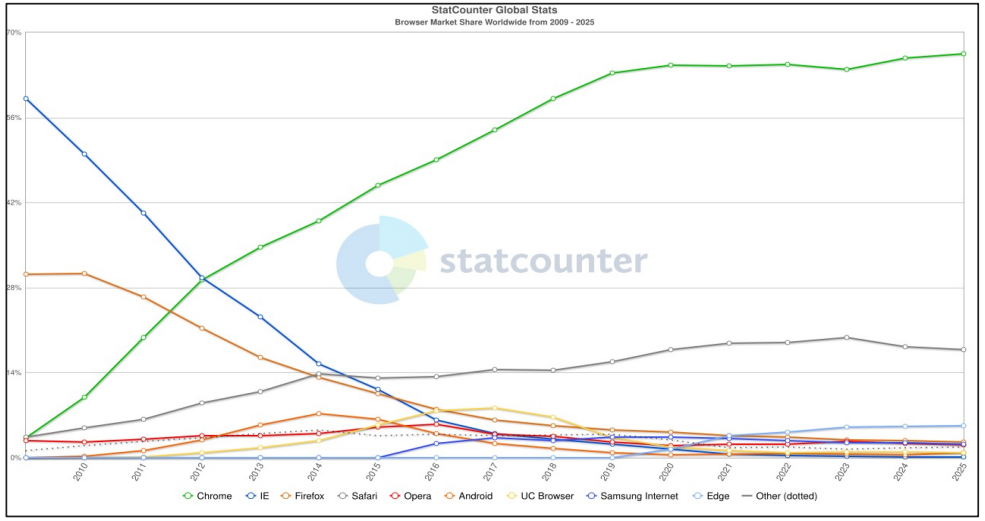
Overview

- About me
- Background: why V8 is important
- Retrospect: porting V8 for RISC-V
- Key point: TurboFan Backend
- Daily maintaining
- Current status

Why V8 is important

- Browsers are important for RISC-V software ecosystems
- Currently, Google Chrome dominates the browser market
- V8 is the JavaScript and Wasm engine for Chrome
- without architecture support in V8, Chrome is completely unable to run on RISC-V

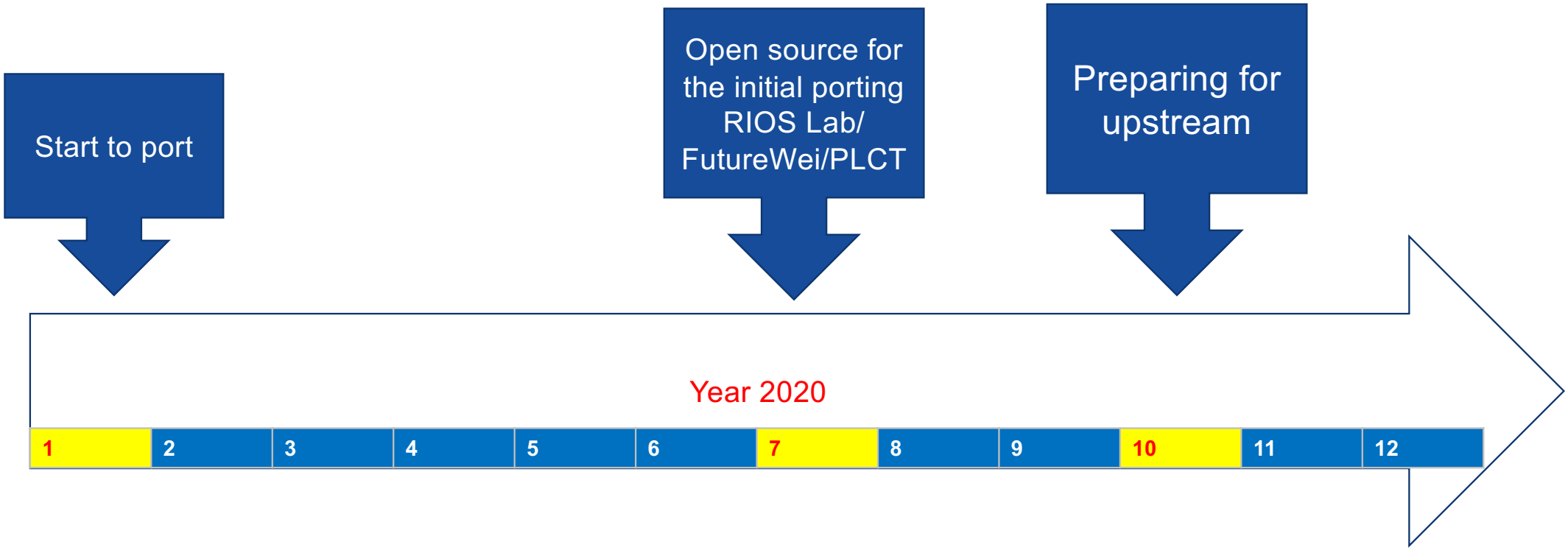




Overview

- About me
- Background: why V8 is important
- Retrospect: porting V8 for RISC-V
- Key point: TurboFan Backend
- Daily maintaining
- Current status

Retrospect: V8 for RISC-V in 2020

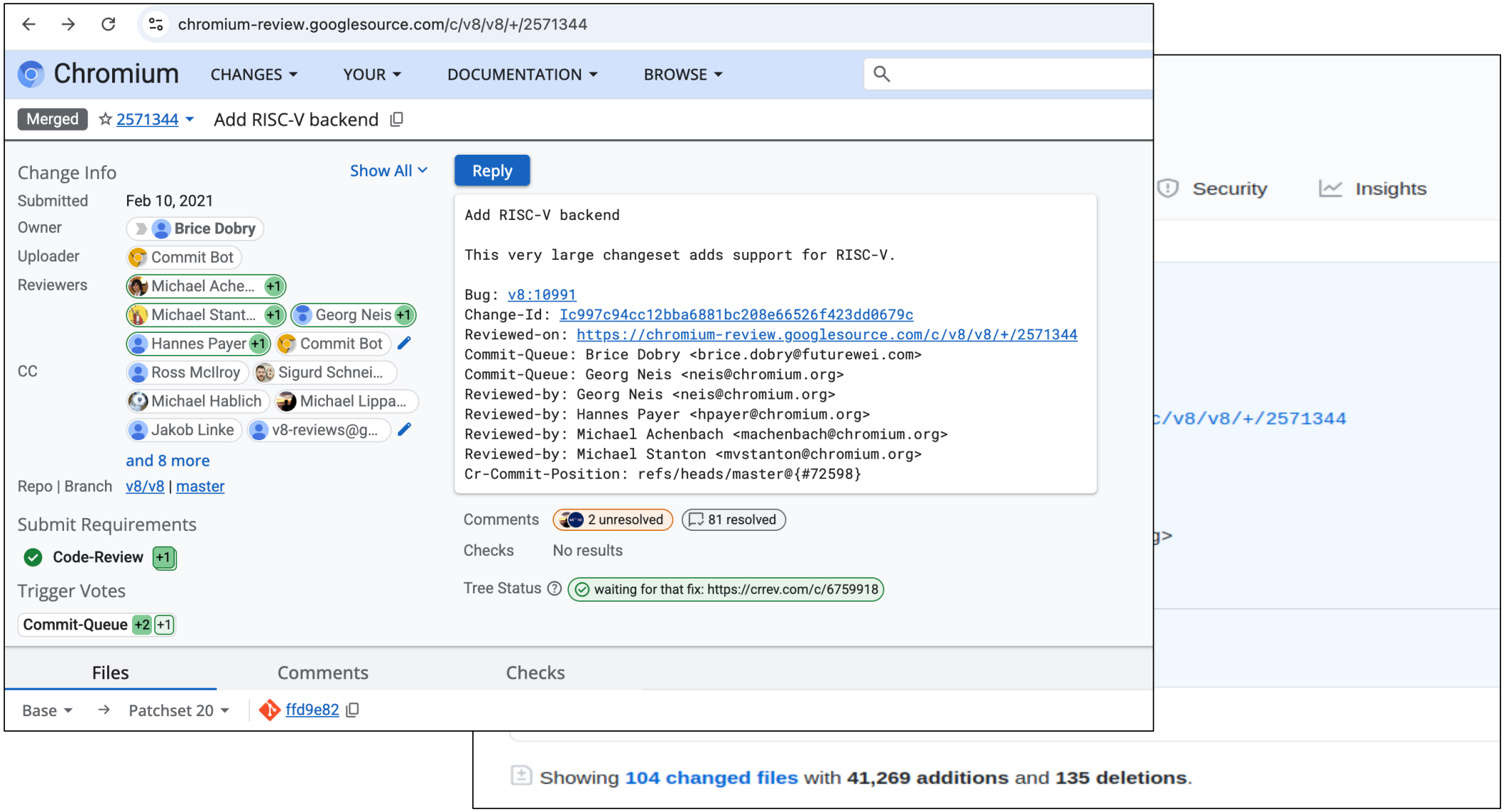


The porting process:

1. Unlocking JavaScript V8 on RISC-V by Peng Wu, https://www.youtube.com/watch?v=MdHL9yGQer8
2. V8 for RISCV, where to com and where to go by PLCT lab, https://github.com/isrc-cas/PLCT-OpenDay-2020/blob/main/V8forRV-PLCTOD2020.pdf

The upstream process

- at first: downstream maintain on GitHub: https://github.com/riscv/v8/
- launch upstream at 2020 October
- many rounds of rebase and bug fix
- skipped several test cases for the regression routine
- finished on Feb 10

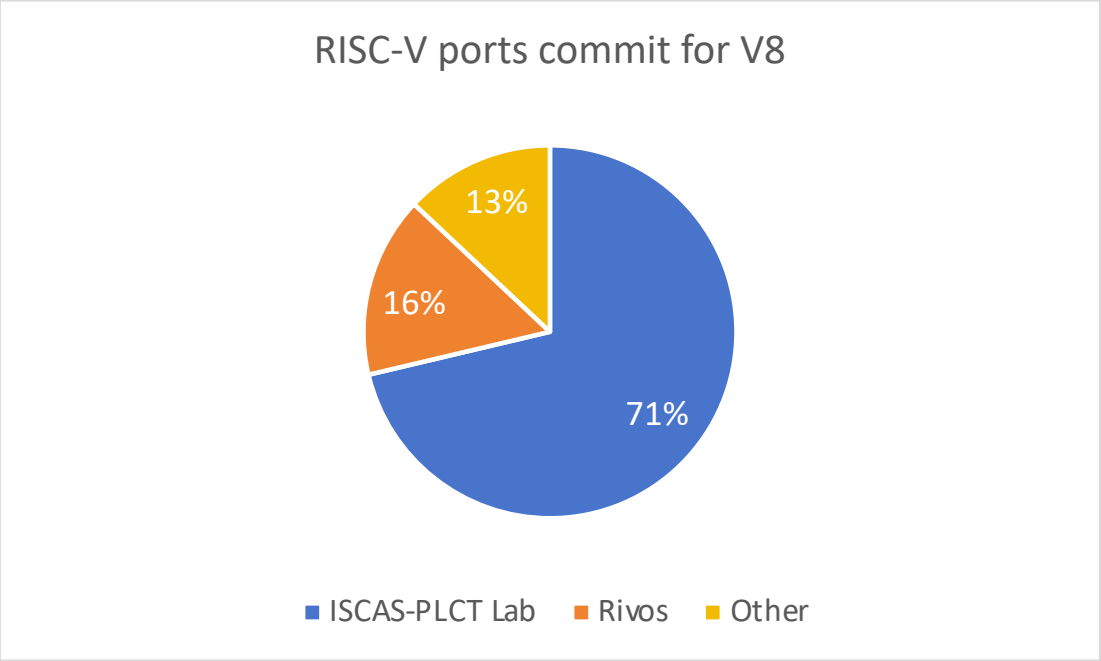


2571344: Add RISC-V backend | https://chromium-review.googlesource.com/c/v8/v8/+/2571344

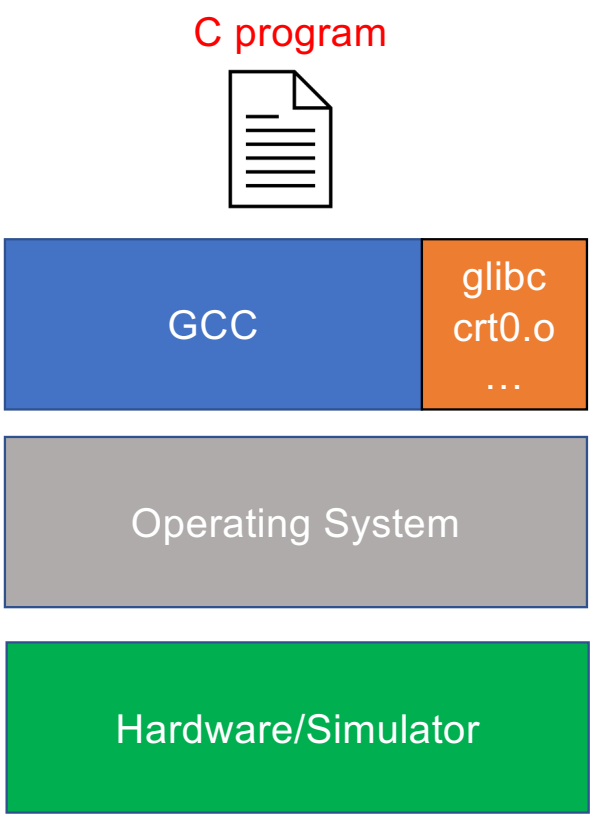
Summary of contributions

Since upstreamed in 2021:

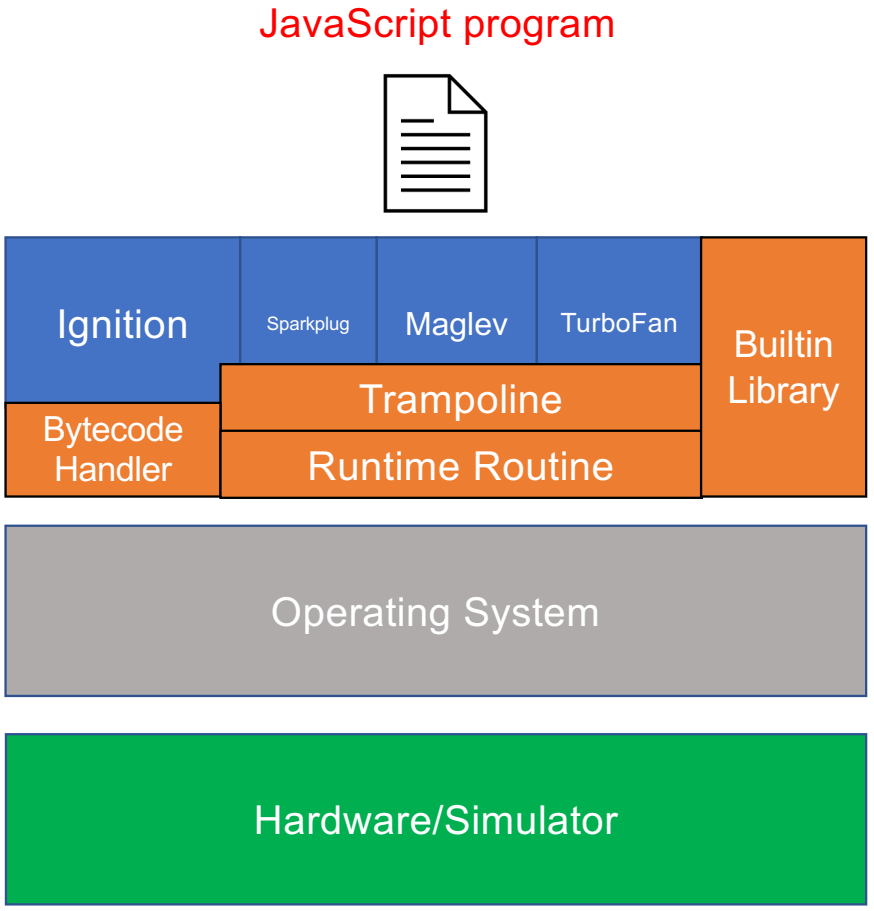
- submitted and merged 631 commits
- accounted for 71% of the total RISC-V port commits
- reviewed and merged 200+ commits



Hierarchy of V8 in Computer Systems
A Comparison with C Compilers



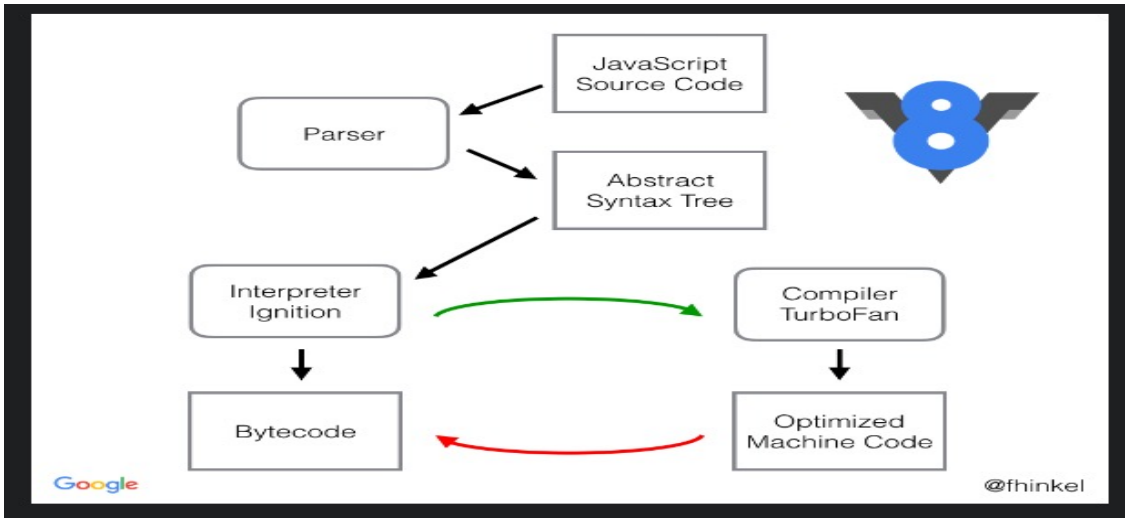
(a) Traditional C compilers and runtime systems



(b) JavaScript engine

V8 modules and ISA-specific parts

- Parser
- JavaScript Engine
  - Ignition：Bytecode generator and Interpreter
  - Sparkplug：baseline JIT
  - Maglev： tier-2 JIT
  - TurboFan： tier-1 JIT
- Wasm Engine：
  - Interpreter: Wasm bytecode interpreter
  - Liftoff： baseline JIT
  - TurboFan： optimized-JIT
- Garbage Collector：Orinoco
- Runtime Support：
  - embedded simulator
  - context/isolate/memory allocator/OS interface
- Builtins
  - CodeStubAssembler
  - Torque
- Build system
- Testsuite and framework

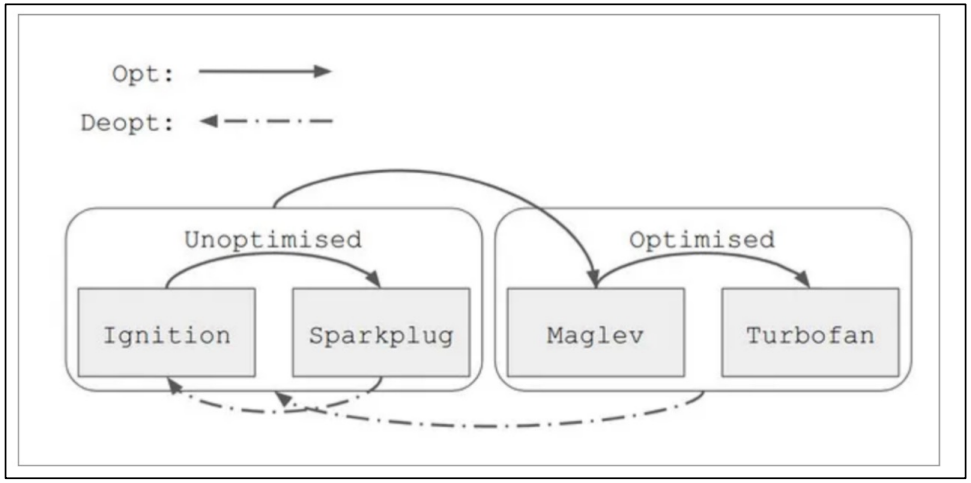


https://fhinkel.rocks/2017/08/16/Understanding-V8-s-Bytecode/

V8 modules and ISA-specific parts

- Parser
- JavaScript Engine
  - Ignition：Bytecode generator and Interpreter
  - Sparkplug：baseline JIT
  - Maglev： tier-2 JIT
  - TurboFan： tier-1 JIT
- Wasm Engine：
  - Interpreter: Wasm bytecode interpreter
  - Liftoff： baseline JIT
  - TurboFan： optimized-JIT
- Garbage Collector：Orinoco
- Runtime Support：
  - embedded simulator
  - context/isolate/memory allocator/OS interface
- Builtins
  - CodeStubAssembler
  - Torque
- Build system
- Testsuite and framework

heavily depend on ISA part
slightly depend on ISA part

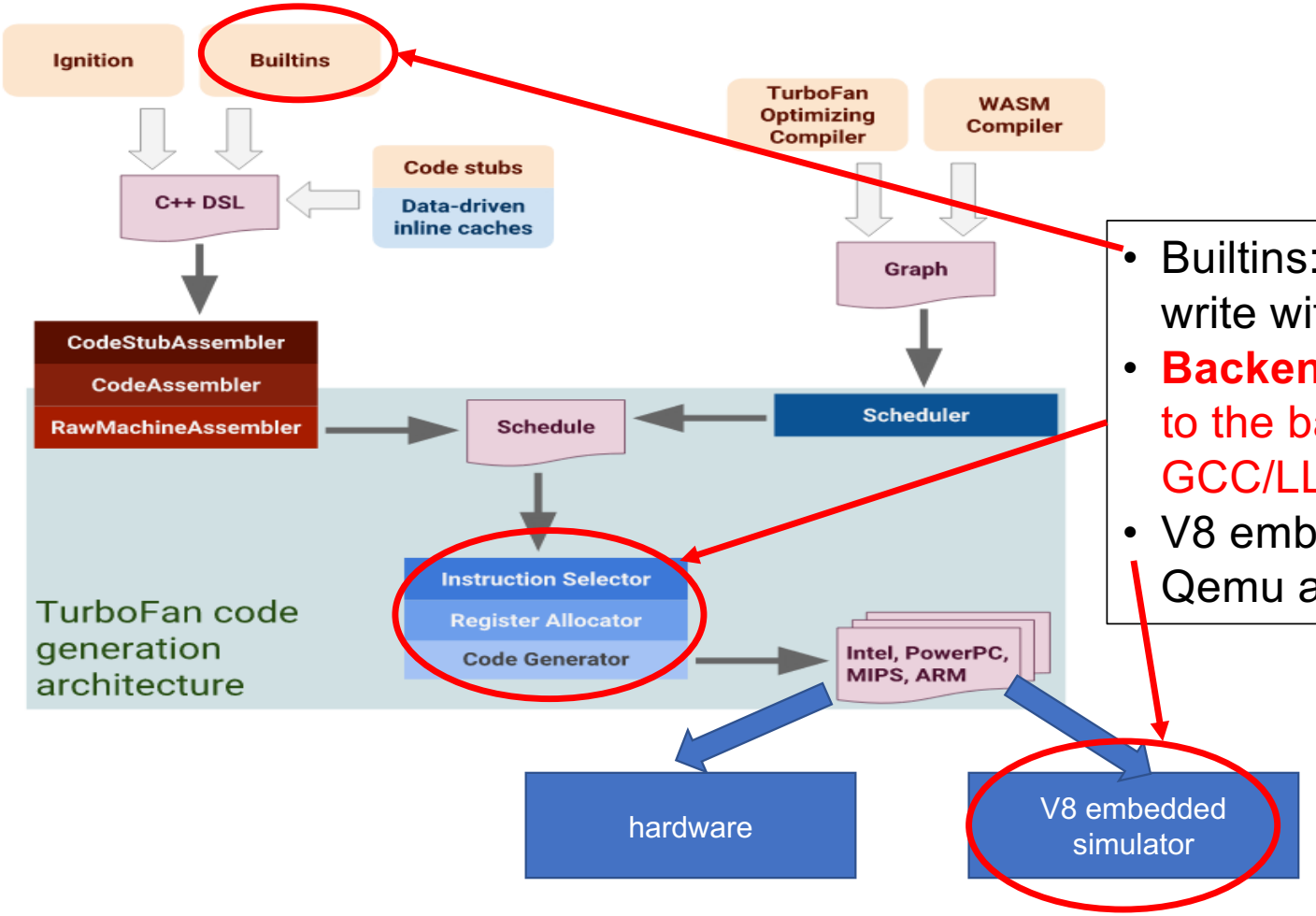


https://www.phoronix.com/news/Google-Chrome-Maglev-Compiler

Overview

- About me
- Background: why V8 is important
- Retrospect: porting V8 for RISC-V
- Key point: TurboFan Backend
- Daily maintaining
- Current status

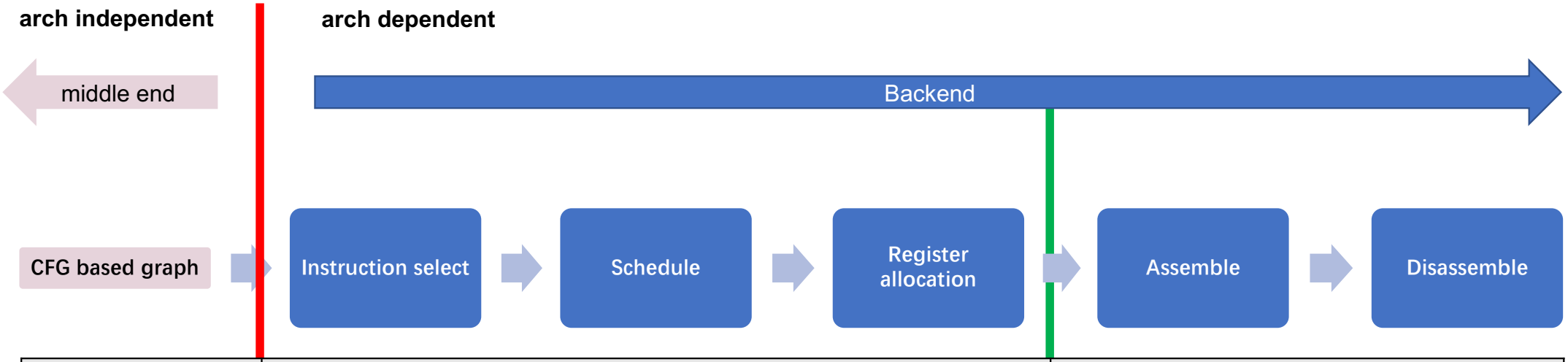
Porting of the TurboFan JIT



- Builtins: similar to the C libraries, write with macro assemble APIs.
- **Backend, i.e., IS/RA/CG：** similar to the backend components of GCC/LLVM, key point to port
- V8 embedded simulator：similar to Qemu and Gdb, written with C++

https://benediktmeurer.de/2017/03/01/v8-behind-the-scenes-february-edition

V8's JIT Backend (TurboFan backend)



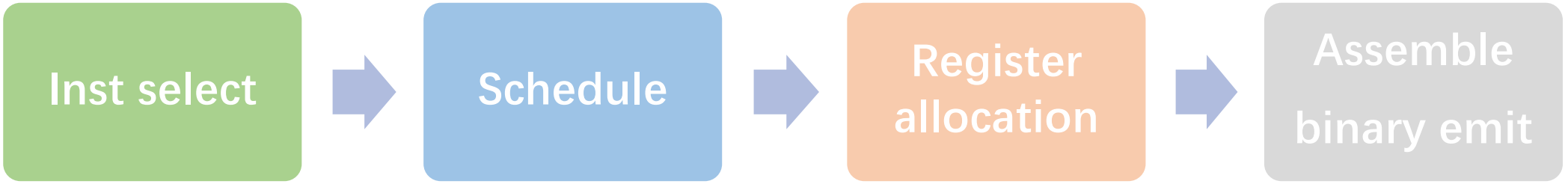
IR type: CFG-based SSA	IR type: Inst	No IR, just binary and meta data
IrOpcode	ARCH_OPCODE	Target binary

The backend of TurboFan is shared by all the AOT and JIT's in V8!

TurboFan Backend Overview

SelectInstructionsAndAssemble() @ pipeline.cc

SelectInstructions()		AssembleCode()
InstructionSelectionPhase.Run()	AllocateRegisters()	AssembleCodePhase
SelectInstructions() @ instruction-selector.cc	1. MeetRegisterConstraintsPhase 2. ResolvePHIsPhase 3. BuildLiveRangesPhase 4. BuildBundlePhase 5. SplinterLiveRangesPhase 6. RegisterAllocation<LinearScanAlloc> a) AllocateGeneralRegistersPhase b) AllocateFPRegistersPhase 7. MergeSplintersPhase 8. DecideSpillingModePhase 9. AssignSpillSlotsPhase 10. CommitAssignmentPhase 11. PopulateReferenceMapsPhase 12. ConnectRangesPhase 13. ResolveControlFlowPhase 14. OptimizeMovesPhase 15. LocateSpillSlotsPhase 16. FrameElisionPhase 17. JumpThreadingPhase	AssembleCode() @code-generator.cc
VisitBlock()	1. StartBlock() 2. AddInstructions() 3. AddTerminator() 4. EndBlock() @instructionscheduler.cc	AssembleBlock()
VisitControl() VisitNode()		AssembleInstruction()
VisitFooNonArchInst		AssembleArchInstruction() AssembleArchJump() AssembleArchBranch() AssembleArchDeoptBranch() AssembleArchBoolean() AssembleArchTrap() AssembleArchBinarySearchSwitchRange() AssembleArchBinarySearchSwitch () AssembleArchLookupSwitch () AssembleArchTableSwitch () @code-generator-[arch].cc
VisitFooArchInst @instruction-selector-[arch].cc	GetInstructionLatency() GetInstructionFlags() @instruction-scheduler-[arch].cc	
Emit()		
instructions_.push_back()		__ [MacroInst]() @ macro-assembler-[arch].cc __ inst() @ assembler-[arch].cc



Overview

- About me
- Background: why V8 is important
- Retrospect: porting V8 for RISC-V
- Key point: TurboFan Backend
- Daily maintaining
- Current status

Daily maintain after upstream

- latest changes port
  - for passing build
  - for new features
- ISA and ABI related configurations
- bug fix
- arch-related optimization

Daily maintain after upstream- part1

latest changes port

- for passing build
- for new features

1. Port the latest changes to pass build
   ① 2690182[*]
   ② 2914246
   ③ 2894036
2. Port the latest changes to introduce new feature
   ① OSR shadow stack (2831171)
   ② add static interface descriptors (2834632)
   ③ The newly introduced compiler Sparkplug port (2763963)
   ④ use EnterFrame and LeaveFrame with StackFrame (2848735)
   ⑤ wasm atomic implementation (2881494)

[*] This is the change number in the Chromium's Gerrit system.
i.e., https://chromium-review.googlesource.com/c/v8/v8/+/2690182

Daily maintain after upstream- part2

latest changes port

- for passing build
- for new features

ISA and ABI related configurations

1. add toolchain configurations for RISC-V (2725484, 2754588)
2. make t6 as a call target register (2814726)
3. rename riscv flags (2878150)
4. C-ext configuration (2876854)
5. fix disassemble of illegal instruction (2874399)

Daily maintain after upstream- part3

latest changes port

- for passing build
- for new fe

ISA and AB

bug fix

arch-relatec

1. NaN processing issue (2814725)
2. Record correct offset in StoreTaggedPointer (2814723)
3. CPU profiler related bug (also help MIPS change the bug) (2902341)
4. typo fix (2917597)
5. unaligned mem access (2887023)
6. fix constant pool (2839546)
7. skip atomic test (2853282)
8. skip cctests (2881505)
9. skip inspector test (2894025)

Daily maintain after upstream- part4

latest changes port

- for passing build
-

1. Port PC-relative bultin-to-builtin calls (2814722)
2. enable constant pool (2814724)
3. remove of unnecessary not (2847673)
4. add/sub with immediate (2848732)
5. add RISCVCmpZero to gen bnez/beqz with zero_reg (2814562)
6. fold imm to the load/store offset (2814563)

arch-related optimization

Overview

- About me
- Background: why V8 is important
- Retrospect: porting V8 for RISC-V
- Key point: TurboFan Backend
- Daily maintaining
- Current status

What's new in 2025

New Specifications and Features of the JavaScript/TC39 Language

- Improvement of the Temporal API
- Enhancement of JSON Parsing
- Security Enhancement of BigInt
- Support for the TC39 Base64 Proposal
- Support for the TC39 Float16Array Proposal

What’s new in 2025

New Features of Wasm

- Growable Stacks
- Compatibility and Security Optimization of Atomics
- Support for Code Coverage Tools
- Implementation of the WebAssembly Shared-Everything Thread
- Implementation of the Wasm Custom Descriptors Proposal
- Implementation of Wasm JSPI and Stack Switching Features

What's new in 2025

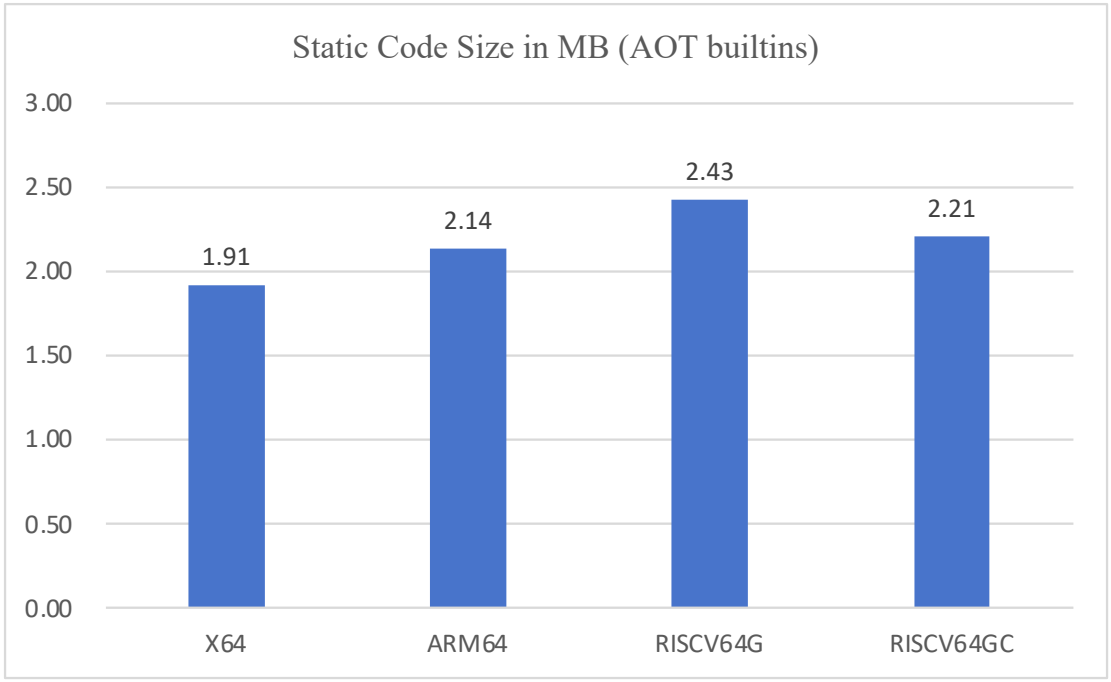
Major Improvements and Optimizations of the V8 Architecture

- Support for Conservative Stack Scanning in the Scavenger Garbage Collector
- Introduction of the New CFG IR and Removal of the Outdated SON IR
- Support for Maglev
- Support for High-Performance Sandbox
- Wasm Interpreter

What's new in 2025
ISA Features of RISC-V

- B extension
- ZiCond extension
- ZicFiss extension　(WIP)
- C extension code-gen optimization

Static code size (AOT Builtins, like libc)

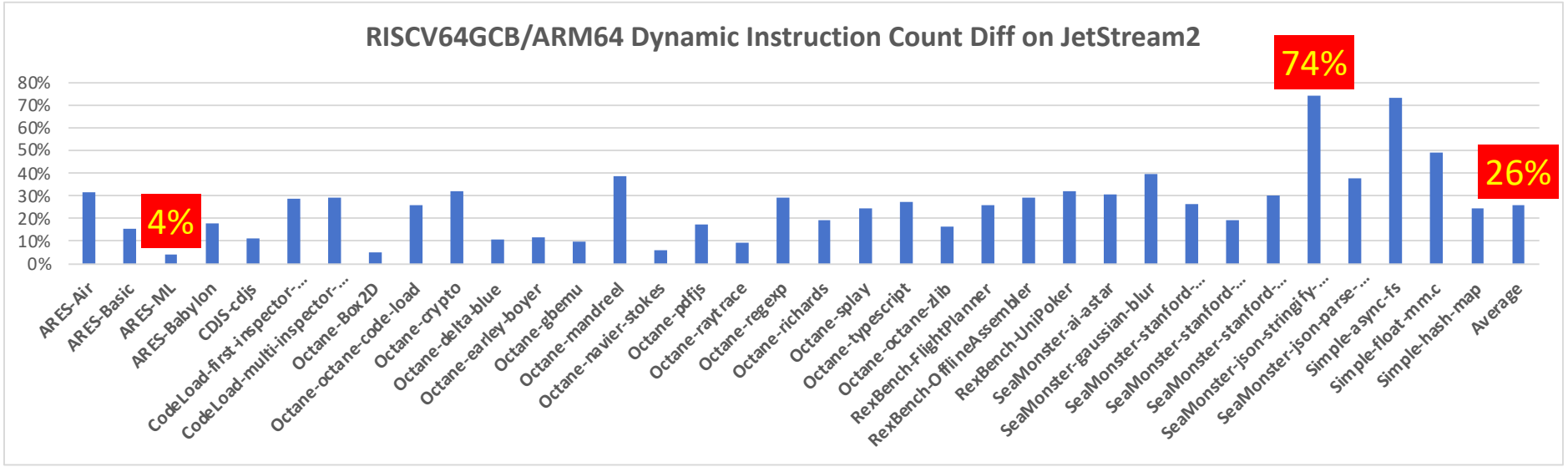


RISCV64GC is 3.2% larger than ARM64

Dynamic code size

- Using qemu-user and qemu-plugins to get user-space dynamic instruction count
- statistics include V8's main body (by C++ compiler), V8's AOT and V8's JIT
- without dynamic library and privileged level instructions

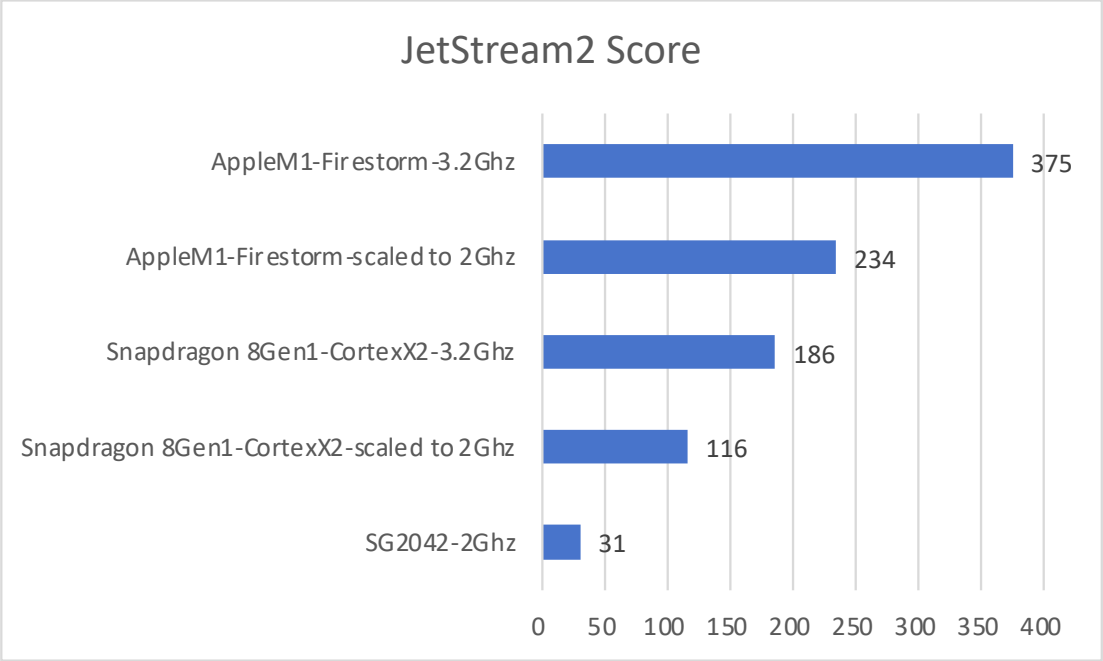
avg 26% more than ARM64
in range from 4% to 74%



https://www.qemu.org/docs/master/devel/tcg-plugins.html#instructions

JetStream2 Score

SG2042 (64core, C920, 2GHZ) with 32GB DDR
Debian GNU/Linux13



!!!Still need in-depth profiling and hw/sw co-tuning

Summary

- V8 is a JavaScript and WebAssembly engine that holds an absolutely dominant market share
- It relies heavily on architecture-specific just-in-time (JIT) compilers
- For daily maintenance, in addition to adding support for new ISA features oriented to hardware, it is even more important to follow upstream updates to port and adapt new language and JIT features
- Compared with Tier 1 platforms, there is still considerable room for optimizing

Welcome to join us!

- https://github.com/plctlab/weloveinterns/blob/master/open-internships.md BJ17
- Or just reach to me

ISCAS
中国科学院软件研究所
Institute of Software Chinese Academy of Sciences

Thank you!