ISCAS 中国科学院软件研究所 Institute of Software Chinese Academy of Sciences | 智能软件研究中心 Intelligent Software Research Center

V8引擎TurboFan后端代码浅析

PLCT实验室 邱吉

qiuji@iscas.ac.cn

2020/06/07

目 录

目 录

背景介绍-1

V8 是什么？

- V8 是 Google 浏览器 Chrome的 JavaScript 引擎，开源
- JavaScript 是一种脚本语言，JavaScript 的引擎负责将这种语言编写的程序进行“解释执行”(interpreter) 或者“即时编译成本地代码然后执行”（Just-In-Time compile to native code）
- Google Chrome 在当前浏览器市场中占据了绝对分额优势
  （ https://en.wikipedia.org/wiki/Usage_share_of_web_browser ）
- V8 is one of the best JavaScript Engine in the world，V8的代码在Chromium项目中
- Chromium 是 Chrome 浏览器的开源项目名称， Chrome 浏览器基于 Chromium 的代码

背景介绍-2

- 为什么要讲 & 做V8
  - 我们来自PLCT（编程语言和编译技术实验室）

PLCT致力于成为编译技术领域的开源领导者，
推进开源工具链及运行时系统等软件基础设施的技术革新，
具备主导开发和维护重要基础设施的技术及管理能力。
与此同时，努力成为编译领域培养尖端人才的黄埔军校，推动先进编译技术在国内的普及和发展。

  - 所在的团队正在对Google的V8 JS引擎进行RISCV移植
    - RISCV是一个“新兴的”指令集体系结构（ “刚满”10岁
    - 目前V8还没有RISCV的porting（可见一个ISA的生态建设是多么困难的事情，十年树木，二十年树ISA

https://riscv.org/2020/05/happy-10th-birthday-risc-v/risc-v-rollout-at-hotchips-26/
https://riscv.org/software-status/

移植工作的基础、状态

基础

- 移植Firefox JaegerMonkey到龙芯MIPS架构的经验
- 具备GCC/LLVM编译器开发经验
- 2018开始接触和学习RISCV ISA
- 2020年初开始学习和准备V8移植
- 目前是6人小队co work

状态，初步完成TurboFan后端：

- 指令选择和寄存器分配阶段体系结构相关的IR操作码设计和相关代码实现
- 指令选择单元测试
- TurboFan后端RISCV64汇编器
- 正在逐个进行 mksnapshot 阶段BUILTINS的代码生成和宏汇编实现

感悟：Hard but not impossible!

难，极具挑战

Jakob Kummerow

Re: [v8-dev] Re: Contributing to V8
其他收件人： um...@codingblocks.com

将帖子翻译为中文

Hi Umair,

V8 is pretty big; is there anything in particular that you're interested in or any part of it that you're already familiar with?

Issue 2051 looks like it can just be closed (which I've just done) -- as the comments mentioned, the fix was already landed, and only the Chromium-side test still needed to be un-skipped.

Getting started with V8 has a rather steep learning curve, and we don't have a list of easy starter projects. It really depends on what your interests are.

Cheers,
Jakob

v8-dev ›

How can I test my own port v8 engine with small example?

2 名作者发布了 2 个帖子

dima...@gmail.com Hello guys. I'm writing my own port v8 engine with exotic architecture. As I can see in code v8 load a lot of built-in js files: array-iterator.js array.js arraybuffer.js collection-iterator.js collection.js generator.js 16/4/20

Jochen Eisinger 16/4/20

将帖子翻译为中文

Hey,

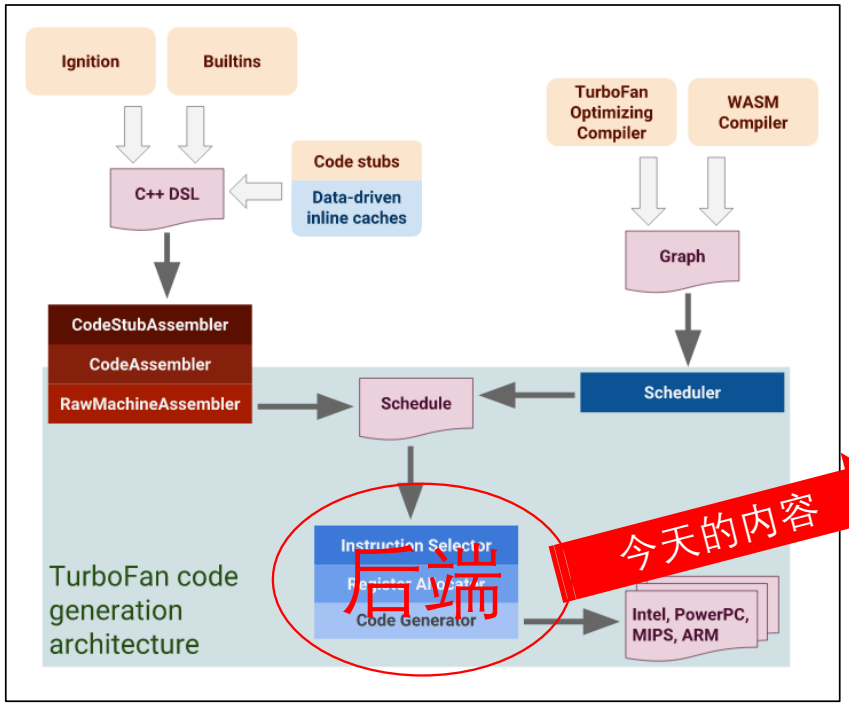
I don't think dropping the JS dependencies will significantly drop the required operations for the assembly backend. You could disable the optimizing compilers (--noopt) and compile without a snapshot and see how far you can get there, but in general, porting v8 to another architecture is a significant piece of work, and I don't know of any short cuts.

best
-jochen

目录

Quick Glance: TurboFan 是 V8的“多才多艺”编译器

Ignition：V8的bytecode解释器，每一个bytecode对应一段预先编译成为unoptimized target binary的bytecode handler
Builtins：Javascript内置函数，提供标准库操作，如RegExp、Array等，使用V8特有的Torque DSL编写
TurboFan Optimizing Compiler：在解释执行阶段，如果发现热点代码，会再次进行优化编译
Wasm Compiler：wasm的编译器
Code Stubs： V8特有的一种DSL，low-level的平台无关汇编语言，用于编写InlineCache/Trampoline等底层操作



C++DSL: Torque和Code Stubs
Graph：编译生成的IR图，Sea-of-Node IR
Schedule：将不带控制流的 SoN Graph变成的IR图

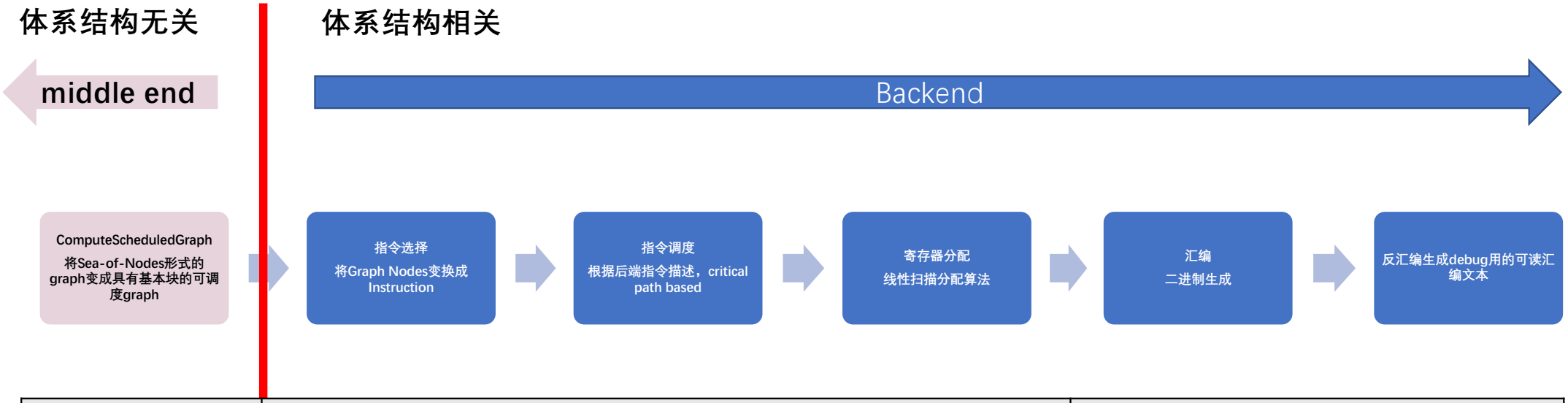
CodeStubAssembler，CodeAssembler，RawMachineAssembler：
用于将CodeStub编译成Graph IR

InstructionSelector: Target相关的指令选择
RegisterAllocator：Target相关的寄存器分配
CodeGenerator：代码（二进制生成）

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

目录

TurboFan Backend的流程



IR形式： SoN graph & node	IR形式： Inst	不再具有IR，变成二进制形式
操作码名称： IrOpcode	操作码名称： ARCH_OPCODE	硬件指令

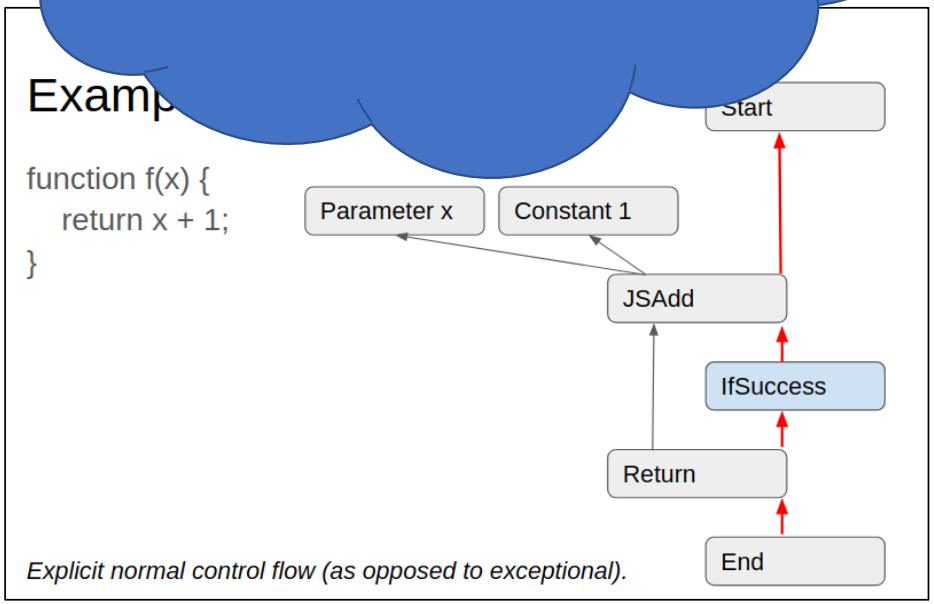
前情提要--后端部分的输入：TurboFan的IR[1]

- 基本概念

- Graph based IR
  - Nodes for operations.
  - Edges for value flow, *control flow* and dependencies.
  - No distinction between basic blocks and statements.
  - Single-static assignment.

- 一个简单的例子：右图->
  - 灰色边：反向后的数据流
  - 红色边：反向后的控制流

Sea-of-Node IR没有BB的概念，在进入后端之前，要先进行“调度”，产生基本块和控制流，因此，后端的输入可以认为有BB，BB内部是直线代码

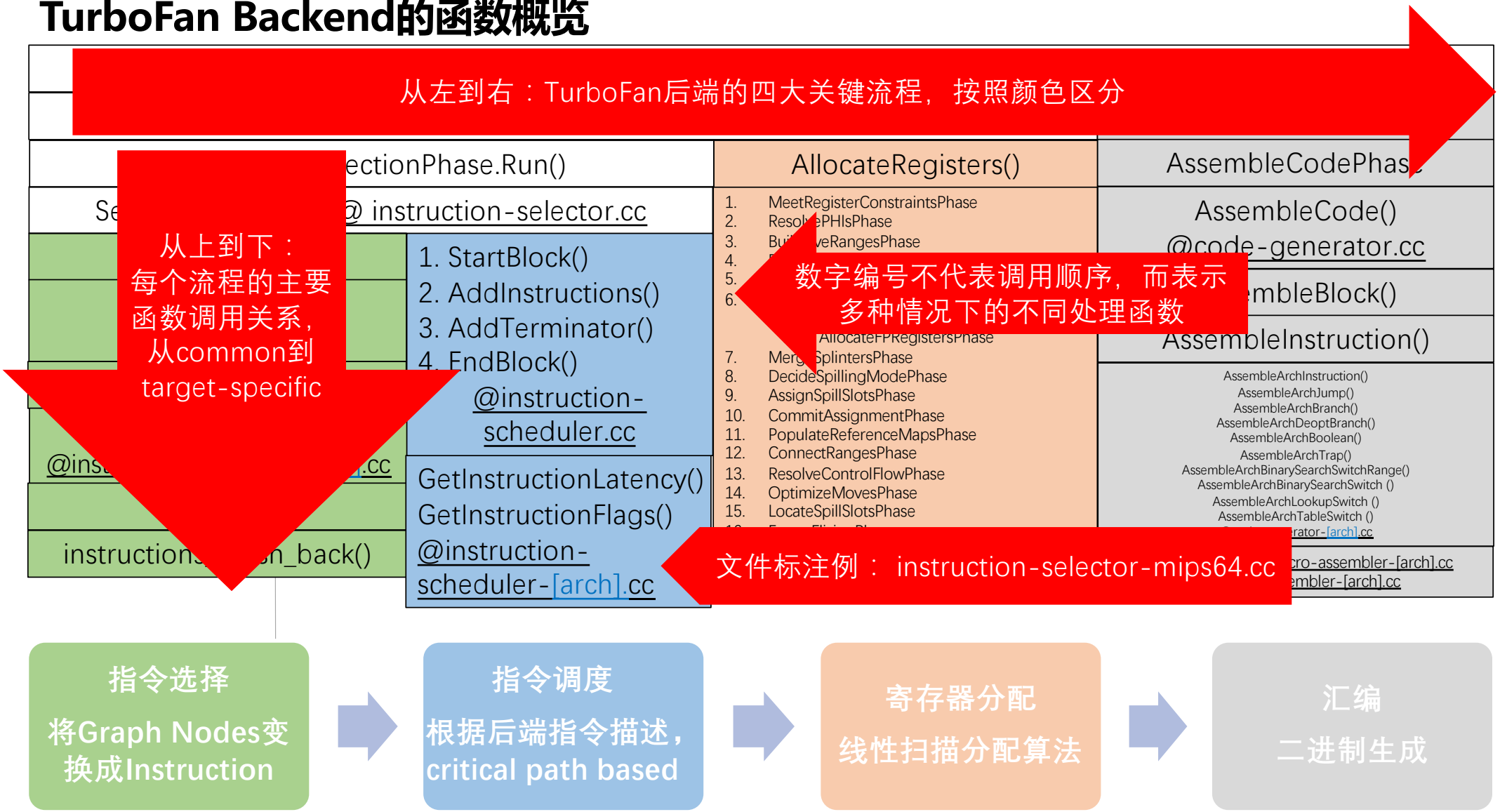


[1] https://docs.google.com/presentation/d/1Z9iIHojKDrXvZ27gRX51UxHD-bKf1QcPzSijntpMJBM/edit#slide=id.p

TurboFan Backend 所包含5个步骤的基本概念

- 指令选择
  - 找出实现一个或一组给定的IR表达式所对应的“恰当的”机器指令序列
  - 完成IR从“机器无关”到“机器相关”的lowering
- 指令调度
  - 找出一个指令序列在某个特定处理器微架构上具有较短执行时间的排列顺序
- 寄存器分配
  - 找出IR中表达式哪些放在寄存器，哪些放在内存，哪些作为硬件指令编码中的常量
  - 指派硬件寄存器，指派内存单元（stack slot）
- 汇编
  - 把IR变成二进制机器码
- 反汇编（non-trivial）
  - 把二进制机器码变成汇编助记符，提高logging和debug info的可读性

TurboFan Backend的函数概览



TurboFan Backend的函数概览

SelectInstructionsAndAssemble() @ pipeline.cc

SelectInstructions()	AllocateRegisters()	AssembleCode()
InstructionSelectionPhase.Run()		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()		__ [MacroInst]() @ macro-assembler-[arch].cc __ inst() @ assembler-[arch].cc
instructions_.push_back()		

指令选择

将Graph Nodes变换成Instruction

→

指令调度

根据后端指令描述，critical path based

→

寄存器分配

线性扫描分配算法

→

汇编

二进制生成

目录

V8 TurboFan backend之指令选择简介

- TF指令选择代码（src/compiler/backend/instruction-selector.cc）
  - 依次遍历Graph中的Block和Node，根据Node的arch无关的IrOpcode，在一个大的switch-case语句中，调用VisitfooInst函数，例如VisitWord32And、VisitWord64And
  - VisitfooInst有两种形式
    - 对于大部分简单可以直接映射到机器指令的IrOpcode，直接调用到arch相关的代码，如src/compiler/backend/arm64/instruction-selector-arm64.cc中的Visit***函数；在这些函数中，要么是直接Emit inst，要么继续根据operand类型，分发到不同的更底层Visit函数中，最后Emit inst
    - 还有一类具有JavaScript语义特性的IrOpcode，例如，具有Speculative特性、超越函数计算等，还会在arch无关中有更进一步的公共处理，继续lower，然后再分发到arch相关的Visit函数，最后Emit inst
- TF指令选择的形式表现（Node转化成了Inst）
  - Node的Context、IrOpcode、Input和Output信息被层层剥离，最后转化成Instruction数据结构表示的instr，然后存储在了InstructionSeletcor对象的_instructions数组中

例子梳理和重点解析

- 一个生成Word32And指令的调用栈（略去进入InstructionSelector之前的trace）

//三层Emit函数，最后一层就是在instructions_ 数组中加入New得到的Instruction

#0 v8::internal::compiler::InstructionSelector::Emit (this=0x7fffffffeca28, instr=0x55555a0a2900) at ../../src/compiler/backend/instruction-selector.cc:257

#1 0x00005555587c7168 in v8::internal::compiler::InstructionSelector::Emit (this=0x7fffffffeca28, opcode=157, output_count=1, outputs=0x7fffffffec460, input_count=3, inputs=0x7fffffffec410, temp_count=0, temps=0x0) at ../../src/compiler/backend/instruction-selector.cc:253

#2 0x00005555587c7428 in v8::internal::compiler::InstructionSelector::Emit (this=0x7fffffffeca28, opcode=157, output=..., a=..., b=..., c=..., temp_count=0, temps=0x0) at ../../src/compiler/backend/instruction-selector.cc:201

//Node在#3到#2时消失，此时，通过直接解析Node得到Input信息（具体方法参考Node数据结构slides）

//Output的获得比较tricky：output就是Node自身，调用OperandGenerator，为Node alloc一个Operand（Vreg or Slot）

#3 0x000055555850a4b1 in v8::internal::compiler::InstructionSelector::VisitWord32And (this=0x7fffffffeca28, node=0x55555a01fe48) at ../../src/compiler/backend/arm64/instruction-selector-arm64.cc:973

#4 0x00005555587cdd29 in v8::internal::compiler::InstructionSelector::VisitNode (this=0x7fffffffeca28, node=0x55555a01fe48) at ../../src/compiler/backend/instruction-selector.cc:1411

#5 0x00005555587c6a1b in v8::internal::compiler::InstructionSelector::VisitBlock (this=0x7fffffffeca28, block=0x55555a0857c0) at ../../src/compiler/backend/instruction-selector.cc:1164

#6 0x00005555587c5e59 in v8::internal::compiler::InstructionSelector::SelectInstructions (this=0x7fffffffeca28) at ../../src/compiler/backend/instruction-selector.cc:93

V8 TurboFan backend之指令调度简介

- TF指令调度代码（src/compiler/backend/instruction-scheduler.cc），两个调度算法：
  - Schedule<CriticalPathFirstQueue>(): 对每一个Block，从Block的end到start一次遍历每一条inst，根据arch相关的指令类型、指令延迟，以及inst的def-use信息，计算其start_cycle，将就绪（前序依赖已经完成的）指令放入ready_list,并对DFG建立critical path，然后基于critical path所定义的优先级，从ready_list找出下一条发射的指令
  - Schedule<StressSchedulerQueue>():不进行critical patch的计算，而只是从ready list中随机找一条指令发射，用于调度模块的压力测试
- TF指令调度的形式表现
  - inst转化成了ScheduleGraphNode
  - 转而存储在了InstructionScheduler的InstructionSequenece sequence_数组中

V8 TurboFan backend指令调度的机器描述方法

- 需要实现两类描述
  - arch相关的指令的latency
    - 在对RISCV64移植过程中，我们暂时没有某款特定微架构的Itin信息，所以全部指令都刻画为相同周期
  - arch相关的指令的Flags属性设置，设置如下的属性flag
    - HasSideEffect：比如store、call等指令
    - LoadOperation：Load指令（不能跟有SideEffect的指令换顺序）
    - MayNeedDeoptOrTrapCheck（不能跟有SideEffect的指令换顺序）
    - Barrier：能引发GC或前后指令顺序不能调整的指令

V8 TurboFan backend之寄存器分配简介

- V8寄存器分配的任务
  - 将inst的操作数，分配到硬件寄存器、或者分配到stack slot上，或者assign为指令中的常量编码
- 主要子phase的功能介绍[1]
  - MeetRegisterConstraintsPhase：数据流合法化，插入合适的move指令
  - ResolvePHIsPhase：插入move指令，lower PHI节点
  - BuildLiveRangePhase：用于给vr赋予生存期
  - SplinterLiveRangePhase：将同一个vr的“冷路径” 和“热路径” 生存期分开，避免“冷路径”的spill在“热路径”中频繁reload
  - RegisterAllocationPhase：经典线性扫描分配算法
  - MergeSplinterPhase：将“冷路径”和“热路径”的分配结果合并
  - AssignSpillSlotsPhase：合并spill区间不重合的不同live interval，给它们分配同一个SpillSlots以减小内存开销
- 移植所需工作：在register-[arch].h文件中对target的寄存器特征和使用进行定义

[1] https://docs.google.com/document/d/1aeUugkWCF1biPB4tTZ2KT3mmRSDV785yWZhwzlJe5xY/edit

V8 TurboFan backend之汇编简介

- 宏汇编器MacroAssembler：src/codegen/mips64/macro-assembler-*
  - 与汇编器的最显著区别是指令可以有一个操作数不区分register或imm，能够简化上层codegen的调用
  - 包含若干具有“高级语义”的宏汇编操作，如CallCFunction
- 汇编器Assembler：src/codegen/mips64/assembler-mips64*
  - 基本上与ISA-SPEC一一对应
  - 对于RISCV64而言，区分好指令的R/I/S/B/U/J encoding type[1]，分门别类逐条实现
  - 实现伪指令：便于codegen和macro-assembler的使用

[1] https://riscv.org/specifications/isa-spec-pdf/

目 录

参考资料 by PLCT

- PLCT contribute slides： https://github.com/isrc-cas/PLCT-Open-Reports
  - V8移植简介
  - V8测试流程介绍以及指令选择单元测试源码分析
  - V8指令选择过程中的优化
  - Dive-into-Torque
- PLCT contribute report videos：
  - 深入V8引擎-第01课：上手开始看 V8 Ignition 解释器的字节码（Bytecodes）
  - 深入V8引擎-第02~04课：从零开始分析V8的构建系统构成part1/part2/part3
  - 深入V8引擎：V8 Call Interface Descriptors
  - V8 Assembler 学习小结
  - V8源码学习：CSA、Builtins、中断等
  - V8单元测试框架
  - V8指令选择中的优化
  - V8移植简介

参考资料 public

- 视频搬运工：
  - https://space.bilibili.com/296494084/channel/detail?cid=100747
- 其他
  - J. E. Smith and R. Nair, Virtual machines - versatile platforms for systems and processes. Elsevier, 2005.
  - Alfred V. Aho, Monica S. Lam, Ravi Sethi , Jeffrey D. Ullman , Compilers: Principles, Techniques, and Tools, Pearson Education, 1st Edition
    by Alfred V. Aho (Author), Monica S. Lam (Author), Ravi Sethi (Author), Jeffrey D. Ullman (Author)
  - https://v8.dev/docs/

谢 谢

欢迎交流合作

2020/06/07