Instruction Cost Modelling: Can we do better?

Sushant Gokhale (Nvidia), Madhur Amilkanthwar (Nvidia)

Presented by: Neil Hickey (Nvidia)

Agenda

- Introduction to Cost modelling
- What is it good at?
- What challenges does its implementation present?
- How could we extend it?

Introduction to cost modelling

Philosophy:

- Simple/ Quick approximation of LLVM-IR cost from the perspective of code gen
- High-level tunable heuristic or relative instruction costs mainly from the perspective of reciprocal throughput
- Used to guide a number of optimisations in the LLVM backend

```
for (BasicBlock &B : F) {
    InstructionCost Cost(0);
    for (Instruction &Inst : B) {
        // Estimate intrinsic cost if the instruction is an intrinsic
        if (auto *II = dyn_cast<IntrinsicInst>(&Inst)) {
            IntrinsicCostAttributes ICA(II->getIntrinsicID(), *II);
            Cost += TTI.getIntrinsicInstrCost(ICA, CostKind);
        } else {
            Cost += TTI.getInstructionCost(&Inst, CostKind);
        }
    }
    OS << "Instruction Cost for BB: " << *Cost.getValue() << std::endl;
}
```

Backend defined

- Implemented in the backend TargetTransformInfo class

```
switch (ISD) {
default:
    return BaseT::getArithmeticInstrCost(Opcode, Ty, CostKind, Op1Info, Op2Info);
case ISD::SDIV:
    if (Op2Info.isConstant() && Op2Info.isUniform() && Op2Info.isPowerOf2()) {
        // On AArch64, scalar signed division by constants power-of-two are
        // normally expanded to the sequence ADD + CMP + SELECT + SRA.
        // The OperandValue properties many not be same as that of previous
        // operation; conservatively assume OP_None.
        InstructionCost Cost = getArithmeticInstrCost(
            Instruction::Add, Ty, CostKind,
            Op1Info.getNoProps(), Op2Info.getNoProps());
        Cost += getArithmeticInstrCost(Instruction::Sub, Ty, CostKind,
                                       Op1Info.getNoProps(), Op2Info.getNoProps());
        Cost += getArithmeticInstrCost(
                    Instruction::Select, Ty, CostKind,
                    Op1Info.getNoProps(), Op2Info.getNoProps());
        Cost += getArithmeticInstrCost(Instruction::AShr, Ty, CostKind,
                                       Op1Info.getNoProps(), Op2Info.getNoProps());
        return Cost;
}
```

Cost modelling example

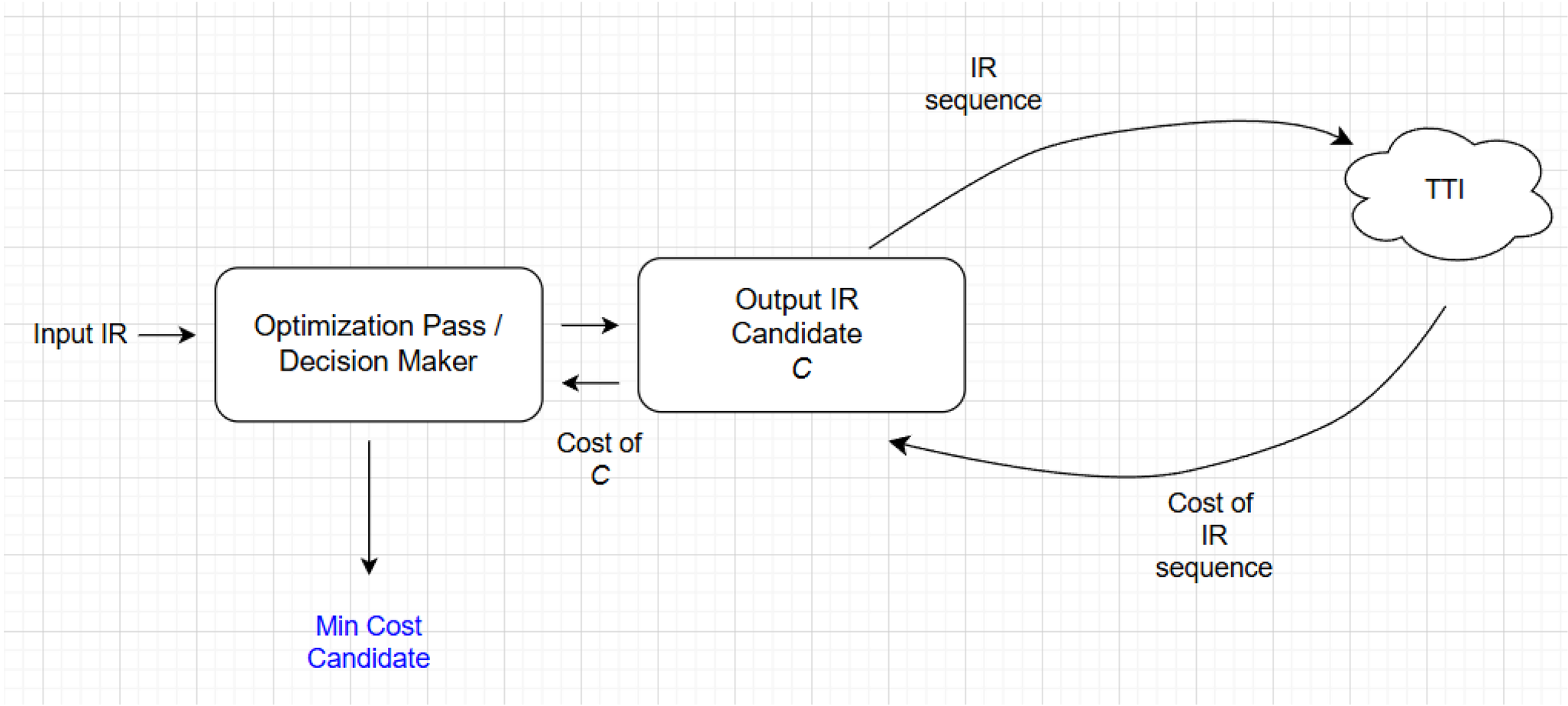
LoopVectoriser:

- Tries to decide, among different vector factors, what is the most work for minimal cost, comparing different strategies

```
// Load: Scalar load + broadcast
// Store: Scalar store + isLoopInvariantStoreValue ? 0 : extract
// FIXME: This cost is a significant under-estimate for tail folded
// memory ops.
const InstructionCost ScalarizationCost =
IsLegalToScalarize() ? getUniformMemOpCost(&I, VF)
: InstructionCost::getInvalid();
```

```
// Choose better solution for the current VF, Note that Invalid
// costs compare as maximumal large. If both are invalid, we get
// scalable invalid which signals a failure and a vectorization abort.
if (GatherScatterCost < ScalarizationCost)
    setWideningDecision(&I, VF, CM_GatherScatter, GatherScatterCost);
else
    setWideningDecision(&I, VF, CM_Scalarize, ScalarizationCost);
continue;
```

Pictorial depiction of cost calculation process



Huh, what is it good for?

Thankfully, not absolutely nothing:

- ~O(1) queries, not reliant on code gen
- Composable and overridable by different backends
- Common infrastructure shared across backends
- Used as a decision tool, doesn’t have to be 100% accurate

What are the challenges

- Maintainability
- Heuristic based
- Low-level of granularity (low numbers, and integers)
- Used as a sum of IR costs.
- Happens at the IR level not at what the hardware actually executes

Maintainability

- Cost model spans 100,000+ lines across LLVM
  - LoopVectorize: 143 InstructionCost uses, 15+ cost functions
  - SLPVectorizer: 195 InstructionCost uses (26,691 lines)
  - 563 TTI method calls across all transforms
- Dual cost models in LoopVectorizer (lines 7079-7102)
  - Legacy cost model + VPlan cost model
  - Must be kept in sync with assertions
  - Comment: "TODO: Switch to more accurate costing based on VPlan"
- Vectorizer cost computations are universal/generic across a backend not specified per core
- LoopUnroll has its own summation logic, as does inliner, SLPVectorise and others
- Not a simple case of updating a single TTI hook value for a backend, but tracing usage and calculations through many lines of code

Heuristic Based

- Initially presented as a simple instruction cost metric to represent reciprocal throughput
- Has evolved through tuning to be purely a heuristic to get the desired output
- Often retroactively applied as lags behind codegen improvements
- Instructions given different costing based on optimization goals
- AArch64 reduction costs (Commit 828a867ee010, July 2025):
  - OR/XOR/AND v8i8: 15 → 5 (reduced by 10!)
  - OR/XOR/AND v16i8: 17 → 7 (reduced by 10!)
  - Reason: "Since the costs were added the codegen improved"
- AArch64 i1 reductions (Commit e79fac2968dc, June 2023):
  - v32i1: 91 → 3 (30x reduction!)
  - v64i1: 181 → 5 (36x reduction!)
  - v128i1: 362 → 9 (40x reduction!)
- AArch64 trip count threshold (Commit d945a2c9efda, Aug 2022):
  - Added getMinTripCountTailFoldingThreshold() to TTI interface
  - Reason: "I noticed regressions... as a solution I propose threshold”
  - Regression-driven, not hardware-based!

Add is 1

- All instructions represented as a multiple of an add instruction
- Doesn’t allow finer-grained management of dispatch constraints and resource utilization
- RISCV percentage cost system (Commit 81efb825703c, July 2021):
  - Problem: RVC compressed = 0.7×, need fractional costs
  - Solution: Scale by 100 (RVI=100, RVC=70, QCEXT=150)
  - Reason: "RVC instructions... can take longer to execute... we consider that two RVC instruction are slightly more costly”
- Each target invents own scaling when integers insufficient

Sum of IR costs

- Just trivially sums up basic blocks by querying each IR instruction
- Doesn’t model modern complex hardware

Can’t capture complex backend fuses

Consider the output IR sequence as anticipated by the optimizer.

```
define i64 @fuse(i64 %x1, i64 %x2){
entry:
    %1 = mul i64 %x2, 8    ; x2 * 8
    %2 = add i64 %x1, %1   ; x1 + (x2 * 8)
    %addr = inttoptr i64 %2 to ptr
    %4 = load i64, ptr %addr
    ret i64 %4
}
```

```
Fuse
        ldr     x0, [x0, x1, lsl #3]
        ret
```

Cost model overestimates cost of highlighted instructions that don’t get materialized in final codegen !

```
Printing analysis 'Cost Model Analysis' for function 'fuse':
Cost Model: Found an estimated cost of 1 for instruction:   %0 = mul i64 %x2, 8
Cost Model: Found an estimated cost of 1 for instruction:   %1 = add i64 %x1, %0
Cost Model: Found an estimated cost of 0 for instruction:   %addr = inttoptr i64 %1 to ptr
Cost Model: Found an estimated cost of 1 for instruction:   %2 = load i64, ptr %addr, align 8
```

How can we improve things?

- Simplify the specification of instruction costs
- JSON configuration file simplifies experiments
- Type-aware overrides (i32, f32, v4i32, v2f64)
- Categories: arithmetic, memory, vector, compare_select, intrinsics
- Graceful fallback to default costs

Example JSON :

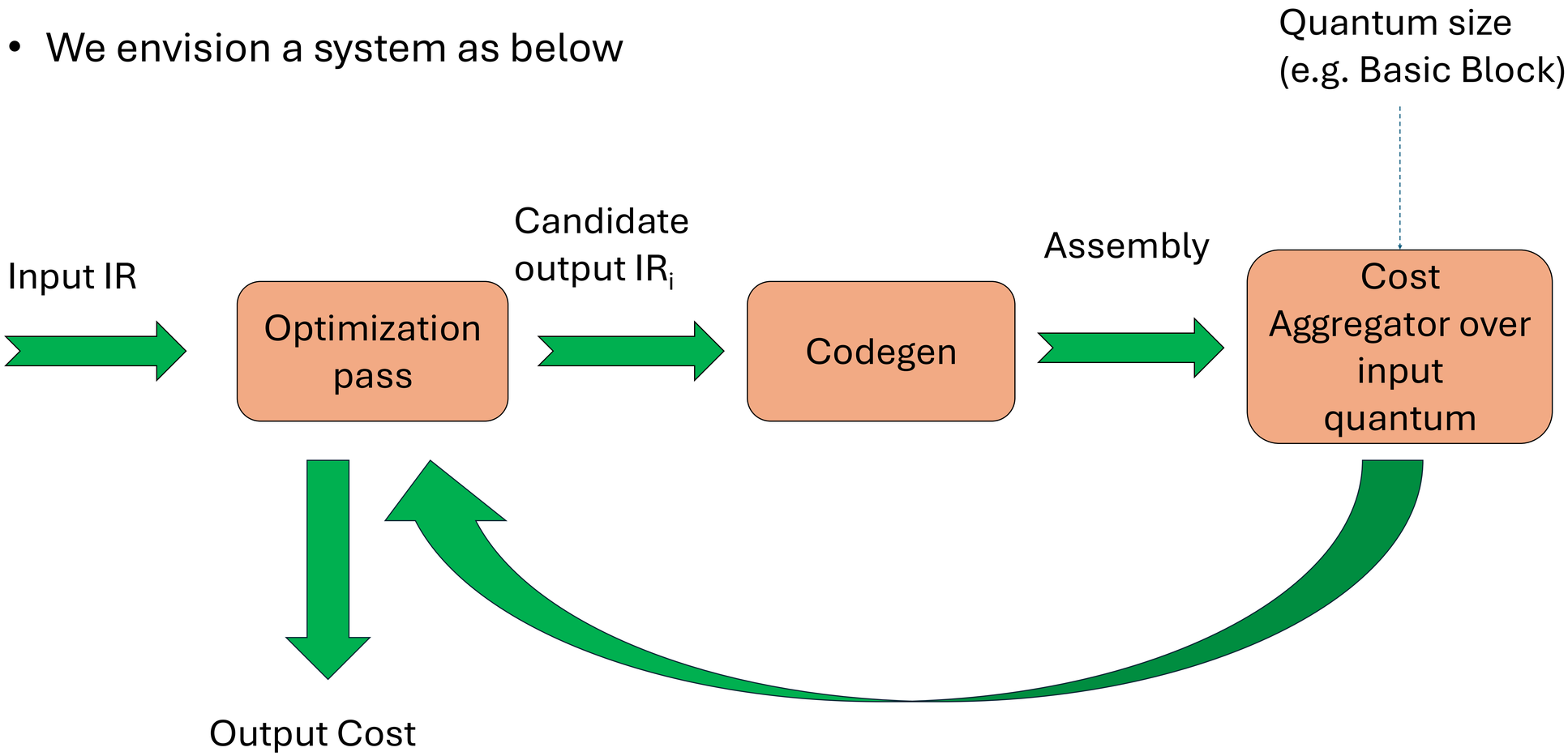
```
{
  "instruction_costs": {
    "arithmetic": {
      "add": { "i32": 1, "i64": 1, "v4i32": 2 },
      "mul": { "i32": 3, "i64": 4, "v4i32": 6 }
    }
  }
}
```

Critical path counting

1. For a given IR sequence, lower to target assembly (virtual)
   a) Use existing codegen infrastructure
   b) Don't emit, just analyze
2. Apply target scheduling model to the basic block
   a) Use existing MachineScheduler infrastructure
   b) Already has: instruction latencies, resource usage, dependencies
3. Extract critical path through the BB
   a) Not sum of all instructions (avoids ILP overcounting)
   b) Critical path = longest dependency chain
   c) Accounts for: instruction fusion, resource conflicts, parallelism
4. Return critical path length as cost
   a) Maps naturally to reciprocal throughput
   b) Already target-specific via scheduling model

Derive costs from actual assembly

- We envision a system as below



- Vision: Automatic cost derivation
- Input IR → Codegen → Measure assembly → Extract costs
- Quantum size: Basic Block
  - Large enough to capture instruction fusion, selection
  - Small enough to be reusable across programs
- Challenges to address:
- 1. Assembly analysis complexity (fusion, macro-ops, uops)
- 2. Mapping assembly back to IR constructs
- 3. Handling context-dependent costs (register pressure, dependencies)
- 4. Build time overhead
- 5. Cache/storage for cost databases