Integration of LLVM-JIT Compiler with Interpreter and manually prepared machine code snippets

Marc Auberer (SAP)
Lukas Rapp (Heidelberg University)

April 16, 2025

Public

Agenda

- Introduction
- Why is our interpreter slow?
- ASM Snippets
- Snippet Management
- Code Generation
- Results
- Conclusion

Introduction

Introduction
Query Execution

- Working on SAP HANA, SAP's flagship in-memory database
- SQL queries processed through complex multi-stage pipeline (ending with *L* program execution)
- Execution uses tiered compiler-interpreter approach

Introduction

Execution Tiers

L Compiler

- Uses LLVM as backbone (MCJIT)
- Default -O0 pipeline
- Custom -O1 pipeline (production)
- Default -O2 pipeline
- Default -O3 pipeline
- IR is constructed in main process
- Middle/backend compilation happens in separate process

L Interpreter

- Custom implementation
- Sequential block-wise execution similar to IR interpretation

Introduction

Product Requirements

Execute query as fast as possible from an E2E perspective.
Start early (low compile time) and *finish quickly* (low runtime).

Balance between latency and throughput is key!

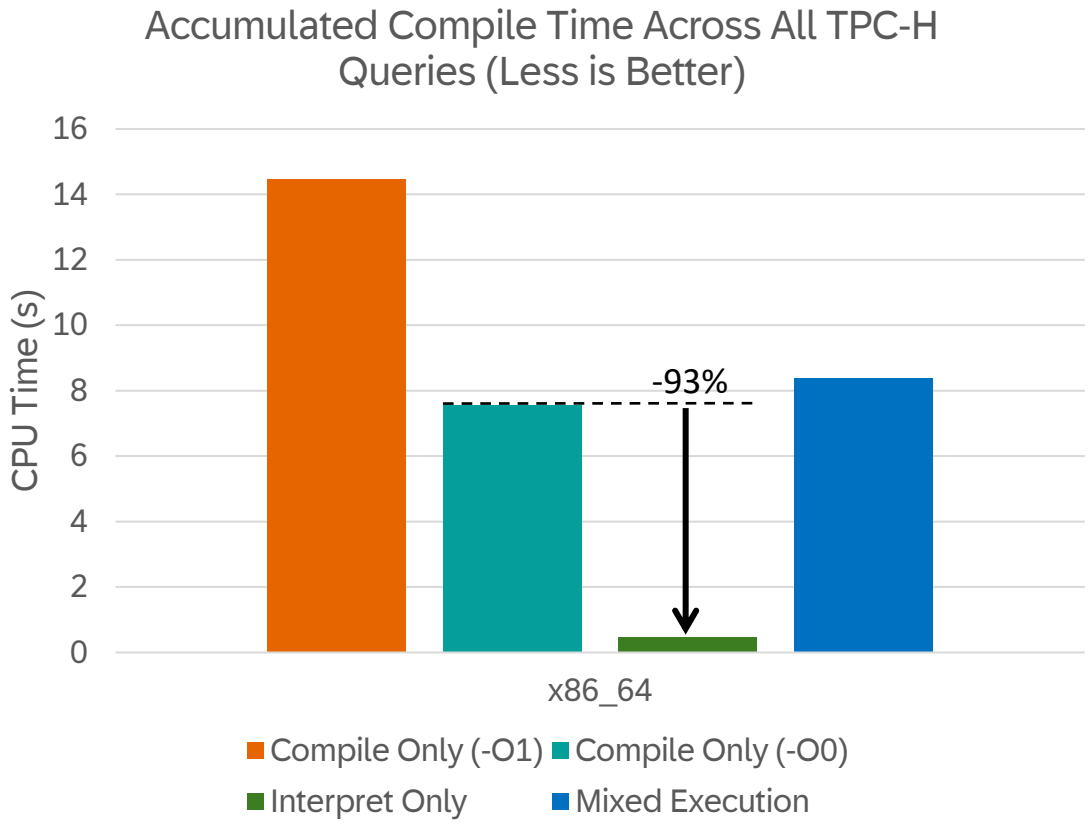
Introduction
Product Requirements

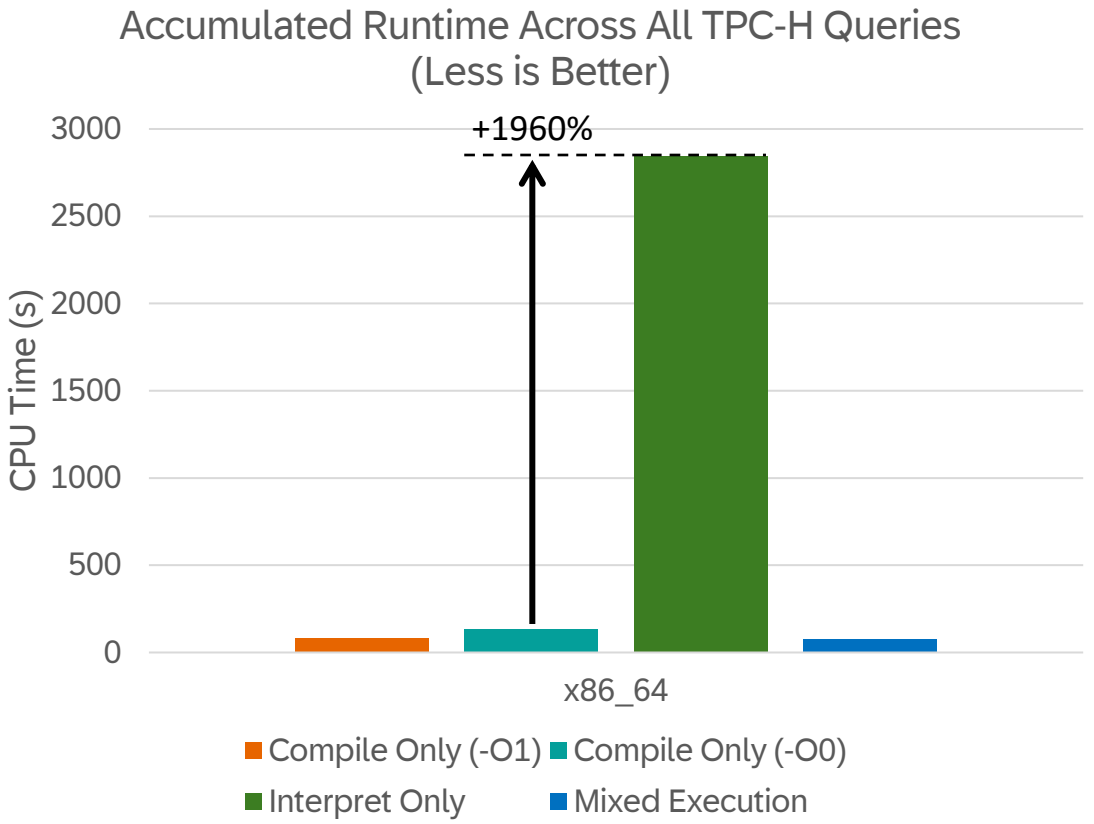
Absolutely essential precondition for our in-memory database context:

CRASH UNDER NO CIRCUMSTANCES!

Introduction

Tier Performances (x86_64)

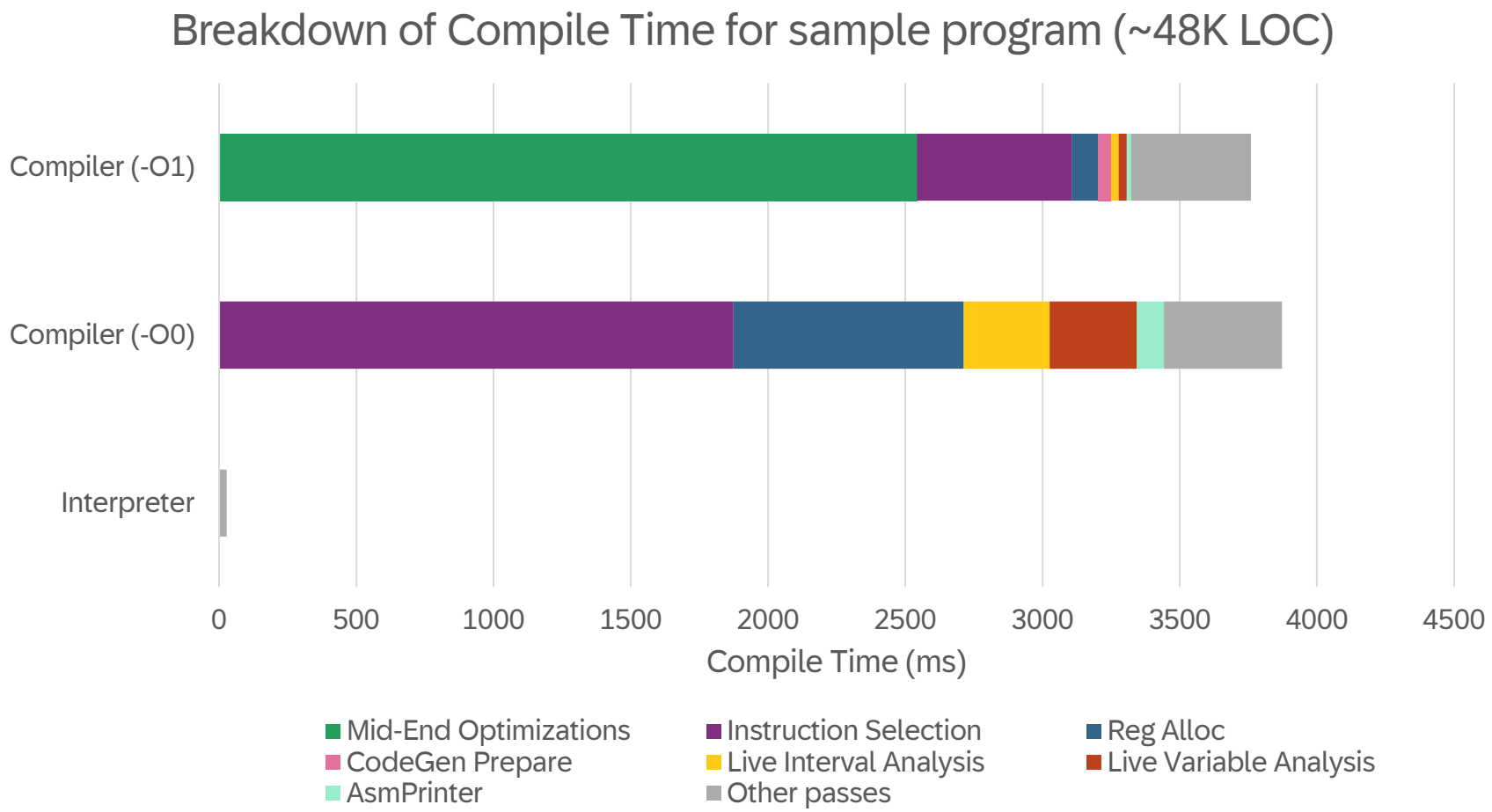




Benchmark Machine: 128 cores / 256 threads

Introduction

Compile Time Breakdown



The program involves many loops, jumps and mathematical expressions.

Why is our interpreter slow?

Why is our interpreter slow?

Virtual Function Calls

```
1  void LocalInterpreter::run() {
2      // > Interpreter Loop
3      while (m_instrPtr ≠ nullptr) {
4          const InterpreterASTNode* nextInstr = *m_instrPtr;
5          try {
6              m_instrPtr++;
7              nextInstr->execute(this);
8          } catch (const LlangReturnCodeException& e) {
9              e.mark_processed();
10             markThrow(e.m_code, nextInstr->m_node);
11         }
12     }
13 }
```

- Interpreter main loop
- Most instructions are simple
- Each instruction has a virtual *execute()* method
- Virtual call overhead dominates runtime

Why is our interpreter slow?

Redundant Loads/Stores

LlangValue

- Generic container for temporary or named value
- Live consecutively in memory
- Either contain buffer with value or point to the heap

- An instruction does not know what it's predecessor already did (node isolation)
- This leads to many redundant loads/stores

Why is our interpreter slow?

Central Questions

(1) How to get rid of the execution overhead?

(2) Can we do this incrementally?

ASM Snippets

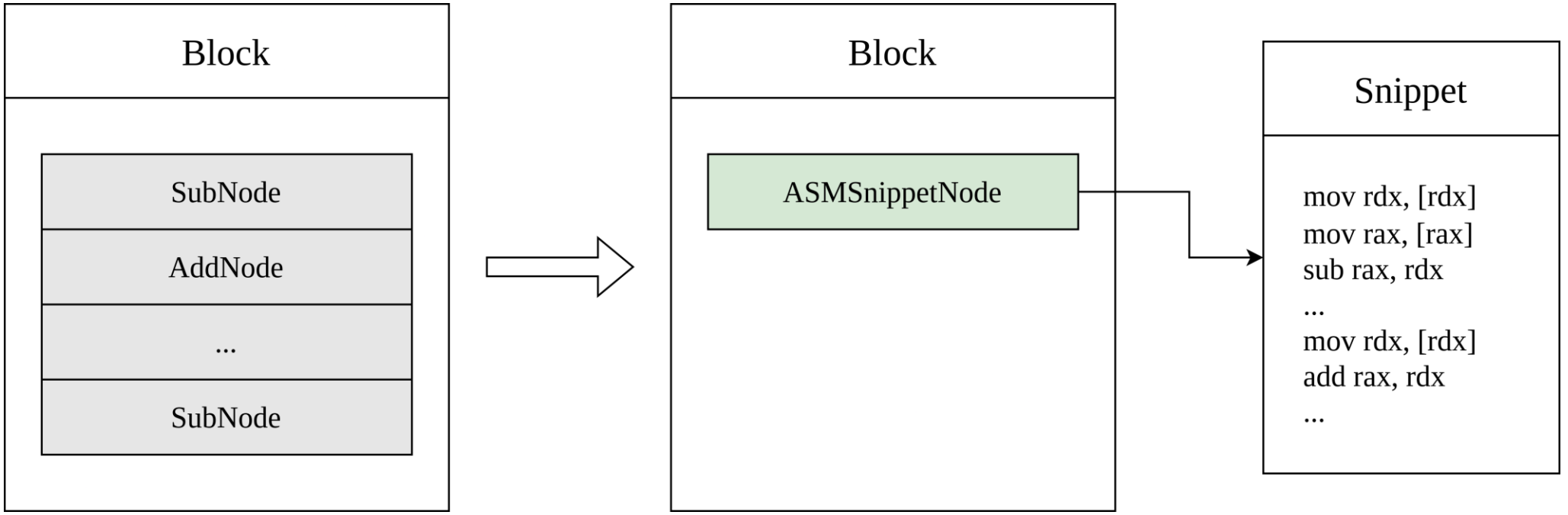
ASM Snippets

Concept

- Simple instructions produce simple machine code
- Generate machine code for those at runtime
- Incremental integration into existing interpreter architecture
- Focus on essential aspects of the language (low complexity and compile time)

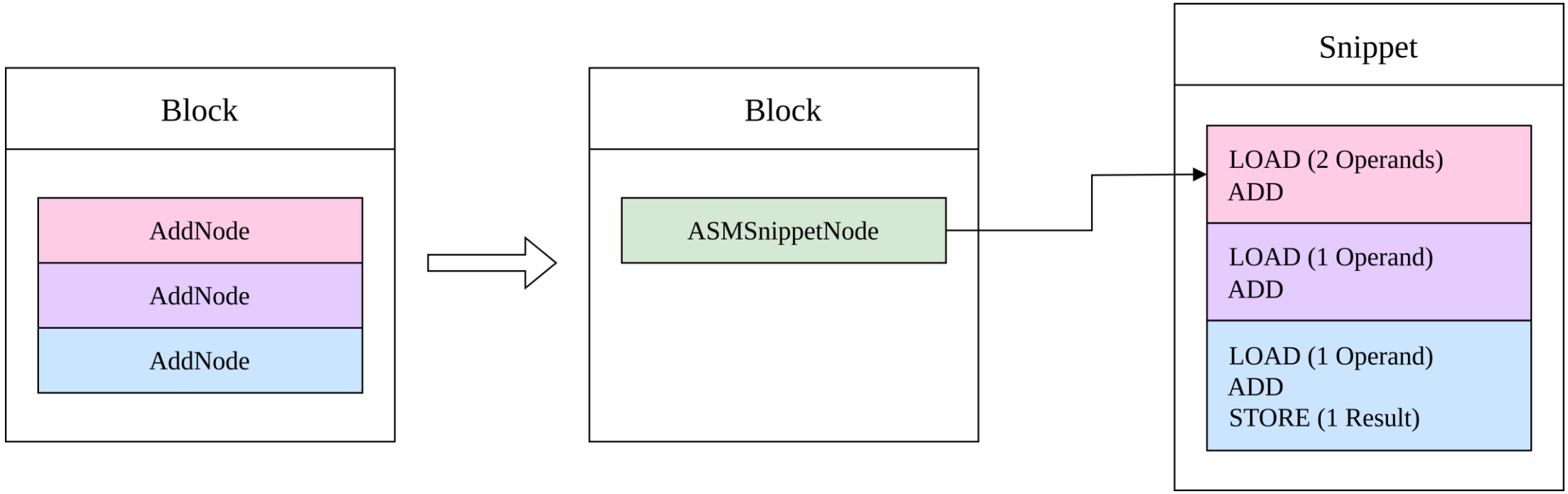
ASM Snippets

Concatenation



ASM Snippets

Compactization



ASM Snippets

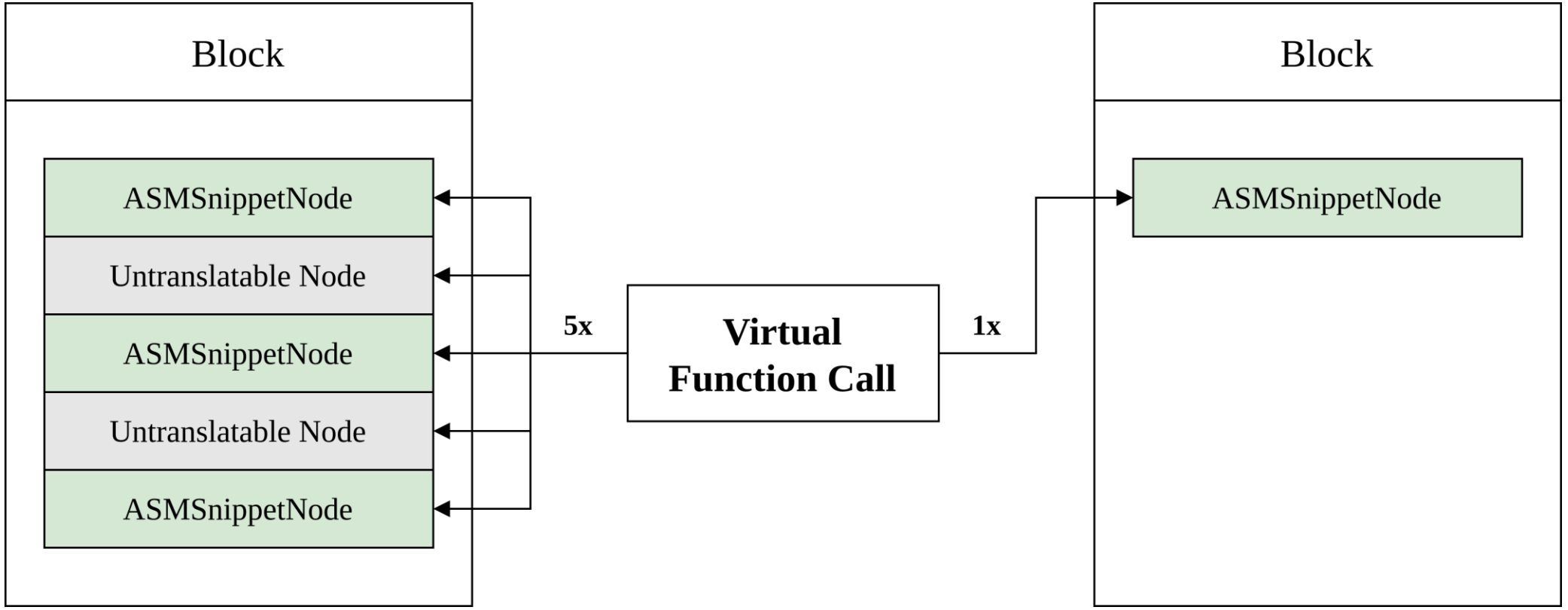
Compactization

There are several reasons as to why a node might not (yet) have an explicit machine code translation:

1) Dependencies
2) Implementation Pending (Priority)
3) Technical Difficulties (Complexity, Exceptions)

ASM Snippets

Devirtualization



ASM Snippets

Devirtualization

Devirtualization is a generic fallback mechanism that removes *Dynamic Dispatch*, enabling most nodes to be included in ASM snippets and unlocking performance gains.

ASM Snippets

Current State

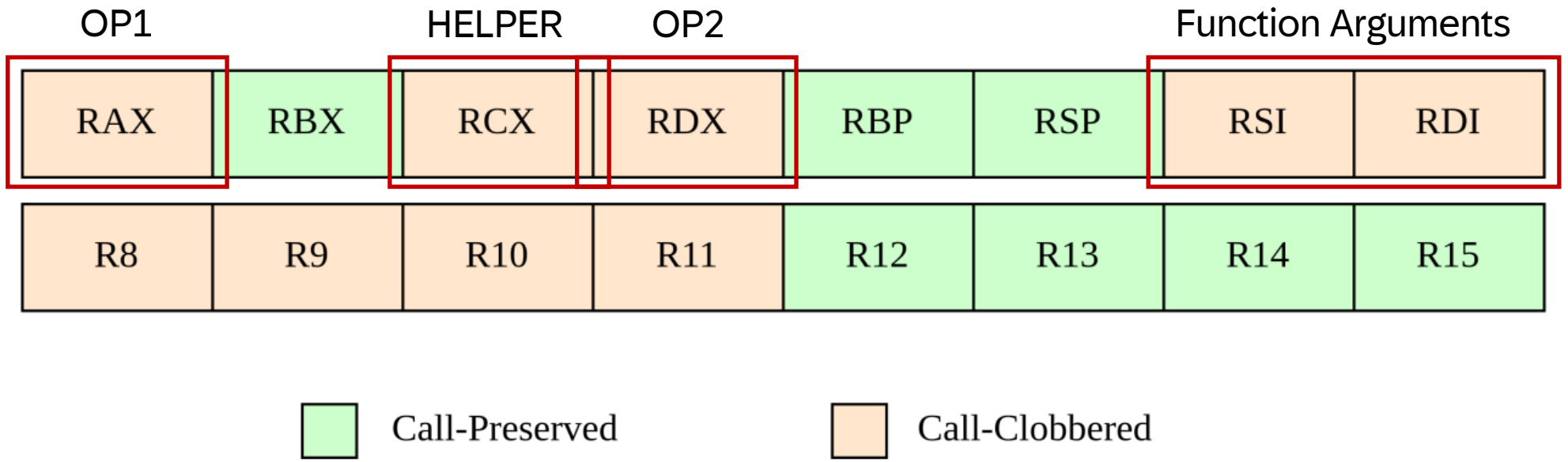
As of today, the implementation explicitly supports:

1) 30-ish x86 / AArch64 Instructions
2) Unary/Binary Operations (e.g. Arithmetic)
3) Member Selection
4) Assignments
5) Logical Jumps
6) Exception Handling
7) Devirtualization
8) Intrinsics (e.g. Strings)

Code Generation

Code Generation

Compactization



Code Generation

Compactization

We define a small set of simple rules to drastically simplify register management and code generation:

- Only call-clobbered registers
- LHS always in OP1
- RHS always in OP2
- Results/intermediary in RAX/OP1
- Lazy data movement (load/store)

Code Generation

Devirtualization

```
1 // Load "this" (PARAMETER ONE) and "LocalInterpreter*" (PARAMETER TWO)
2 uint64_t thisPtr = reinterpret_cast<uint64_t>(astNode);
3 m_target->writeMOV(PARAMETER_ONE, ASMImmediate::Imm64(thisPtr));
4 m_target->writeMOV(PARAMETER_TWO, m_target->STACK_FRAME_BASE() + LOCAL_INTERP_STACK_OFFSET);
```

(a) Prepare *execute()* function call arguments

```
1 // Load devirtualized function address
2 uint64_t funcPtr = static_cast<uint64_t>(astNode->getFctPtrToExecuteMethod());
3 m_target->writeMOV(HELPER, ASMImmediate::Imm64(funcPtr));
4
5 // Call devirtualized function
6 m_target->writeCALL(HELPER);
```

(b) Resolve function address and invoke *execute()* function

```
1 mov rdi, -4548986510646525730
2 mov rsi, qword ptr [rbp-8]
3 mov rcx, -4548986510646525730
4 call rcx
5 cmp al, 1
6 je 18
```

(c) Resulting x86 machine code

Code Generation

Intrinsic Support

As a side node,* we identified common data types with a noticeable impact on performance and simple logic, which can easily be implemented:

1) Strings
2) NullBool / NullInt
3) Simplified C++ function calls

* pun intended

Snippet Management

Snippet Management

Explicit Handling

Since the machine code functions are generated at runtime, we have to explicitely provide and register information that the compiler would otherwise have prepared:

1) Function Metadata (Name, Location)
2) Unwind Table (C++ Exception Handling)

Snippet Management

Stack Traces

```
[CRASH_STACK]  Stacktrace of crash: (2025-04-07 12:44:10 579 Local)
----> Symbolic stack backtrace <----
  0: ljit::dynamic/foo + 0x30
        SFrame: IP: 0x00007f2706a3d070 (0x00007f2706a3d040+0x30) FP: 0x00007f26f8a123a0 SP: 0x00007f26f8a12380 RP: 0x00007f27adc2f41d
        Params: 0x7f2706993390, 0x7f2706a35f68, 0x141, 0xface1fff, 0x0, 0x7f27b286ea10
        Regs: rax=0x7b, rdx=0x141, rcx=0xface1fff, rbx=0x7fff405a0404, rsi=0x7f2706a35f68, rdi=0x7f2706993390, rbp=0x7f26f8a12390, r8=0x8
        Source: foo:1
    ----------------------------------------
  1: ljit::llang::LocalInterpreter::process(ljit::llang::InterpreterASTAsmSnippetNode const*) + 0x7d
        Symbol: _ZN4ljit5llang16LocalInterpreter7processEPKNS0_28InterpreterASTAsmSnippetNodeE
        SFrame: IP: 0x00007f27adc2f41d (0x00007f27adc2f3a0+0x7d) FP: 0x00007f26f8a123d0 SP: 0x00007f26f8a123a0 RP: 0x00007f27adc356e4
        Regs: rbp=0x7f26f8a123c0
        Source: llvm/llang/impl/ljit_llang_LocalInterpreter.cpp:290
        Module: /data/i516467/src/build/DebugMold/gen/libhdbljitbase.so
    ----------------------------------------
  2: ljit::llang::InterpreterASTAsmSnippetNode::execute(ljit::llang::LocalInterpreter*) const + 0x24
        Symbol: _ZNK4ljit5llang28InterpreterASTAsmSnippetNode7executeEPNS0_16LocalInterpreterE
        SFrame: IP: 0x00007f27adc356e4 (0x00007f27adc356c0+0x24) FP: 0x00007f26f8a123f0 SP: 0x00007f26f8a123d0 RP: 0x00007f27adc2ee17
        Regs: rbx=0x7fff405a0404, rbp=0x7f26f8a123e0, r12=0x0, r13=0x7fff4059f77f, r14=0x7fff4059fce0, r15=0x0
        Source: llvm/llang/impl/ljit_llang_LocalInterpreter.cpp:1887
        Module: /data/i516467/src/build/DebugMold/gen/libhdbljitbase.so
    ----------------------------------------
  3: ljit::llang::LocalInterpreter::run() + 0xb7
        Symbol: _ZN4ljit5llang16LocalInterpreter3runEv
        SFrame: IP: 0x00007f27adc2ee17 (0x00007f27adc2ed60+0xb7) FP: 0x00007f26f8a12490 SP: 0x00007f26f8a123f0 RP: 0x00007f27adb1e9c6
        Regs: rbx=0x7fff405a0404, rbp=0x7f26f8a12480, r12=0x0, r13=0x7fff4059f77f, r14=0x7fff4059fce0, r15=0x0
        Source: llvm/llang/impl/ljit_llang_LocalInterpreter.cpp:196
        Module: /data/i516467/src/build/DebugMold/gen/libhdbljitbase.so
```

Snippet Management

Disassembly

```
   5    return a + b;
          asm snippet call
          disassembled snippet:
000000: 55                          pushq     %rbp
000001: 48 89 e5                    movq %rsp, %rbp
000004: 57                          pushq     %rdi
000005: 56                          pushq     %rsi
scope begin block statement
000006: 48 bf de c0 de c0 de c0 de c0   movabsq   $-4548986510646525730, %rdi
000010: 48 8b 75 f8                 movq -8(%rbp), %rsi
000014: 48 b9 de c0 de c0 de c0 de c0   movabsq   $-4548986510646525730, %rcx
00001e: ff d1                       callq     *%rcx
builtin method call: _operator_add [vid=2]
000020: 48 8b 7d f0                 movq -16(%rbp), %rdi
000024: 48 8b 57 18                 movq 24(%rdi), %rdx
000028: 8b 12                       movl (%rdx), %edx
00002a: 48 8b 47 48                 movq 72(%rdi), %rax
00002e: 8b 00                       movl (%rax), %eax
000030: 48 8b 08                    movq (%rax), %rcx   Here is the bug!
000033: 89 87 88 00 00 00           movl %eax, 136(%rdi)
000039: c9                          leave
00003a: c3                          retq
          return statement
   6  }
```

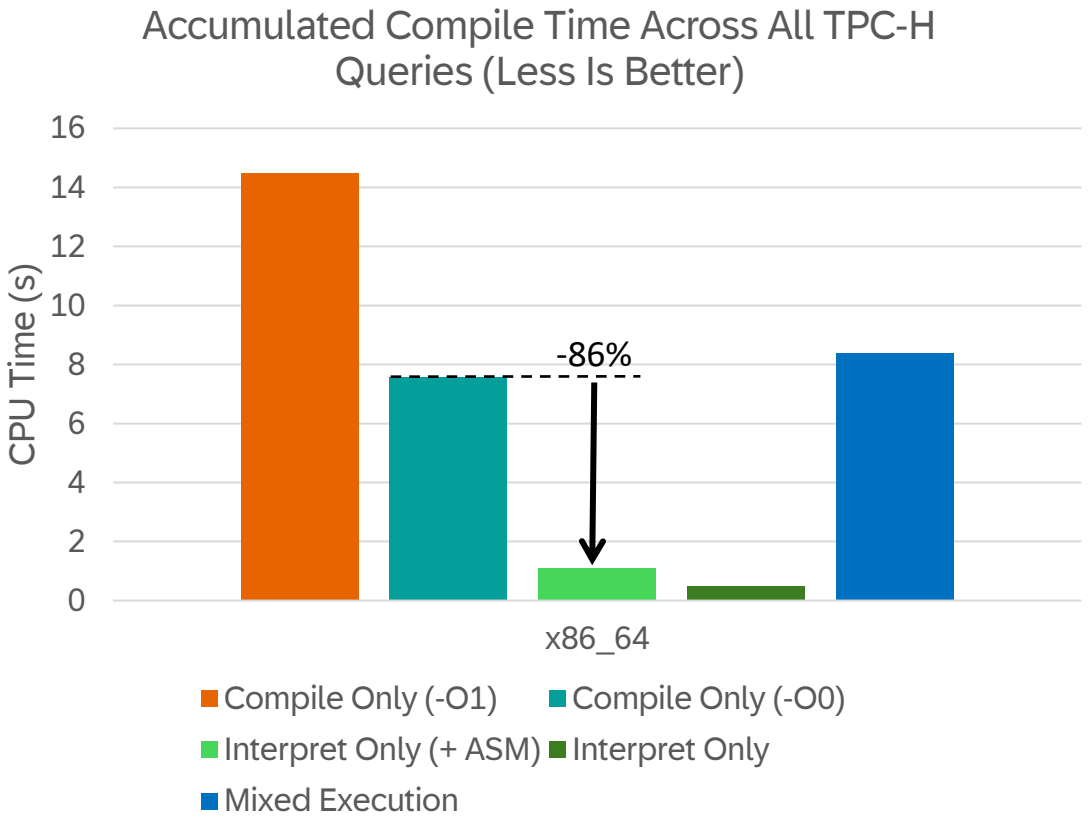
Further diagnostics:

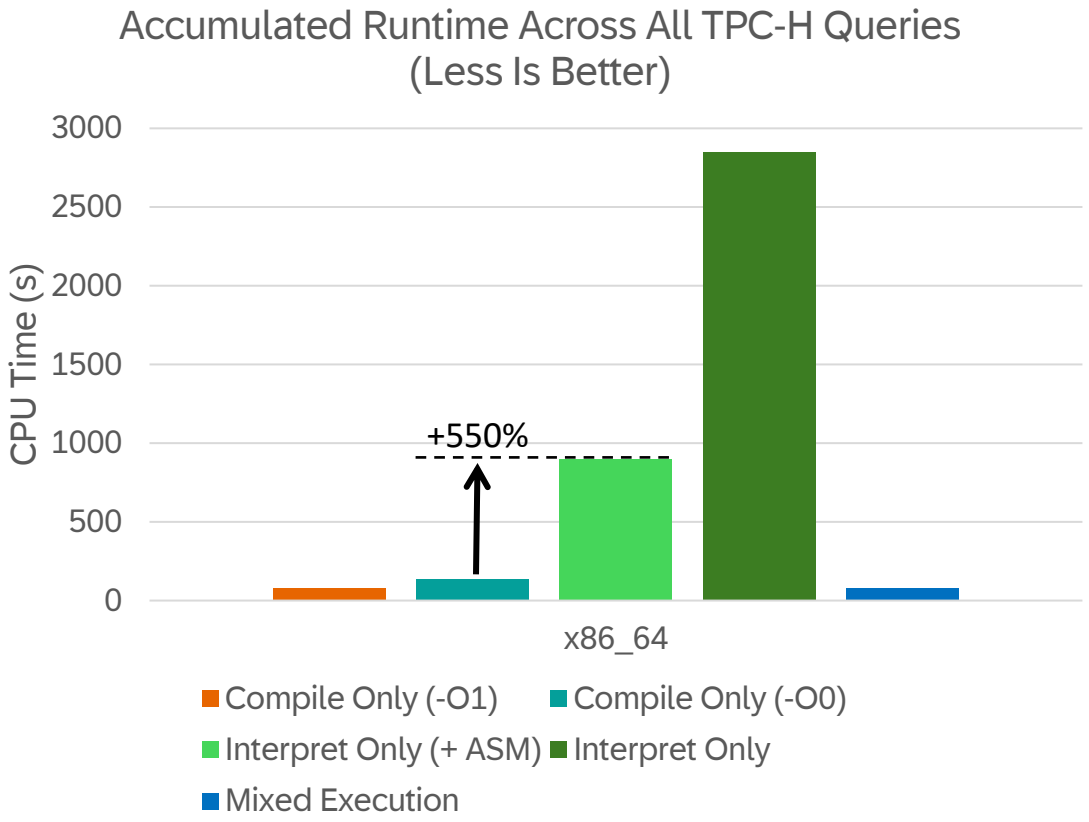
1) Disassembly in crash dumps is easily obtainable
2) Disassembly dump option for development

Results

Results

Tier Performances (x86_64)

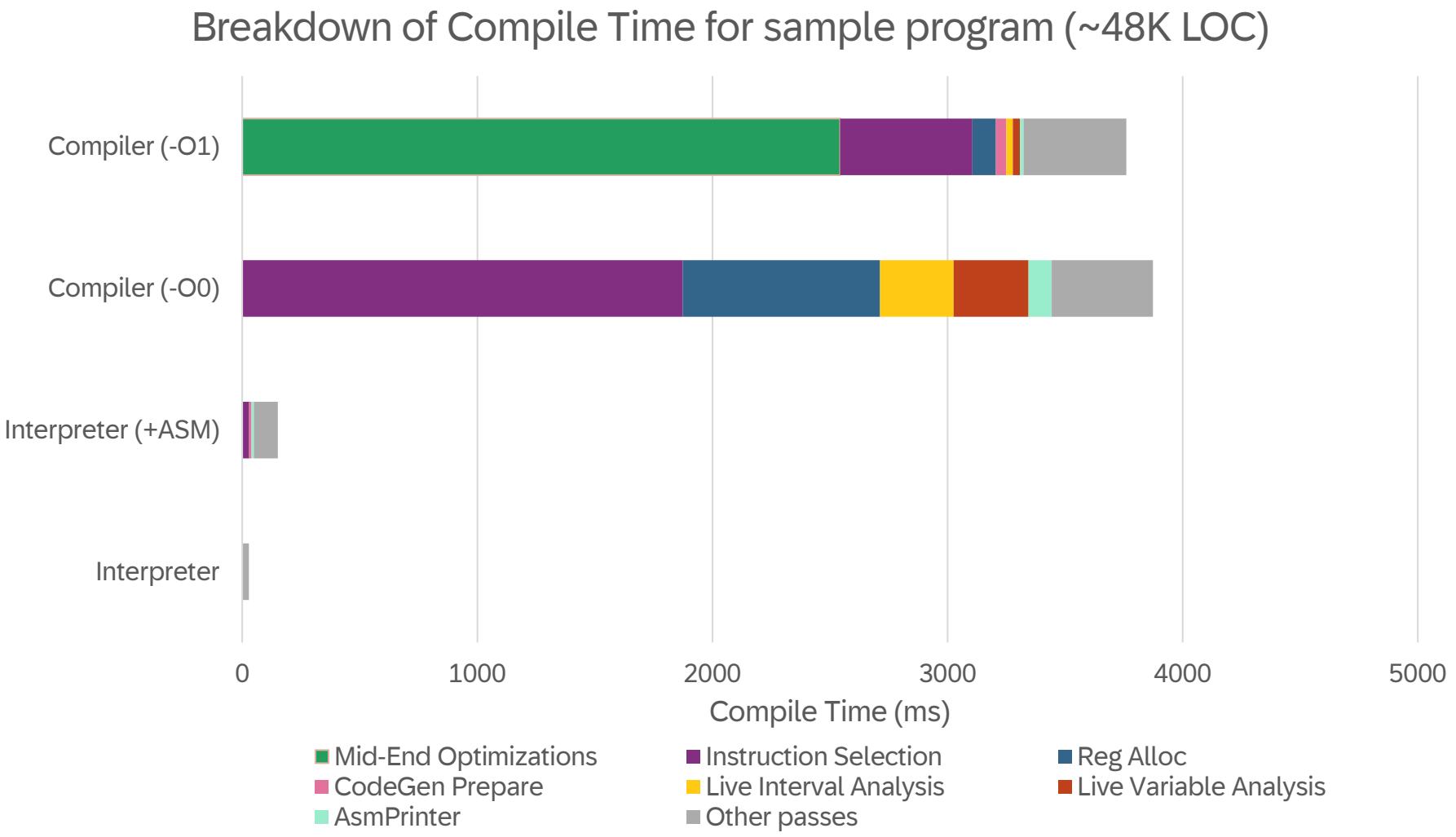




Benchmark Machine: 128 cores / 256 threads

Results

Compile Time Breakdown



Conclusion

Conclusion

Summary

- Bridge LLVM latency gap with ASM Snippets at runtime
- Little effort for a robust, tailored solution
- Slight compile-time increase, big performance gains
- Order-of-magnitude difference in interpreter vs compiler runtime performance (~7x)
- Supports an incremental transition

Conclusion

Learnings (for LLVM)

- LLVM -O0 compile time performance not sufficient for latency-sensitive applications (at least LLVM 20)
- -O0 mostly spends time RegAlloc, ISel and LiveIntervalAnalysis
- Ultra-low latency compilation not available
- More backend options desireable:
  - Custom register allocation stategy (RegAlloc)
  - Limit instruction selection (ISel)

Thank you.

Contact information:

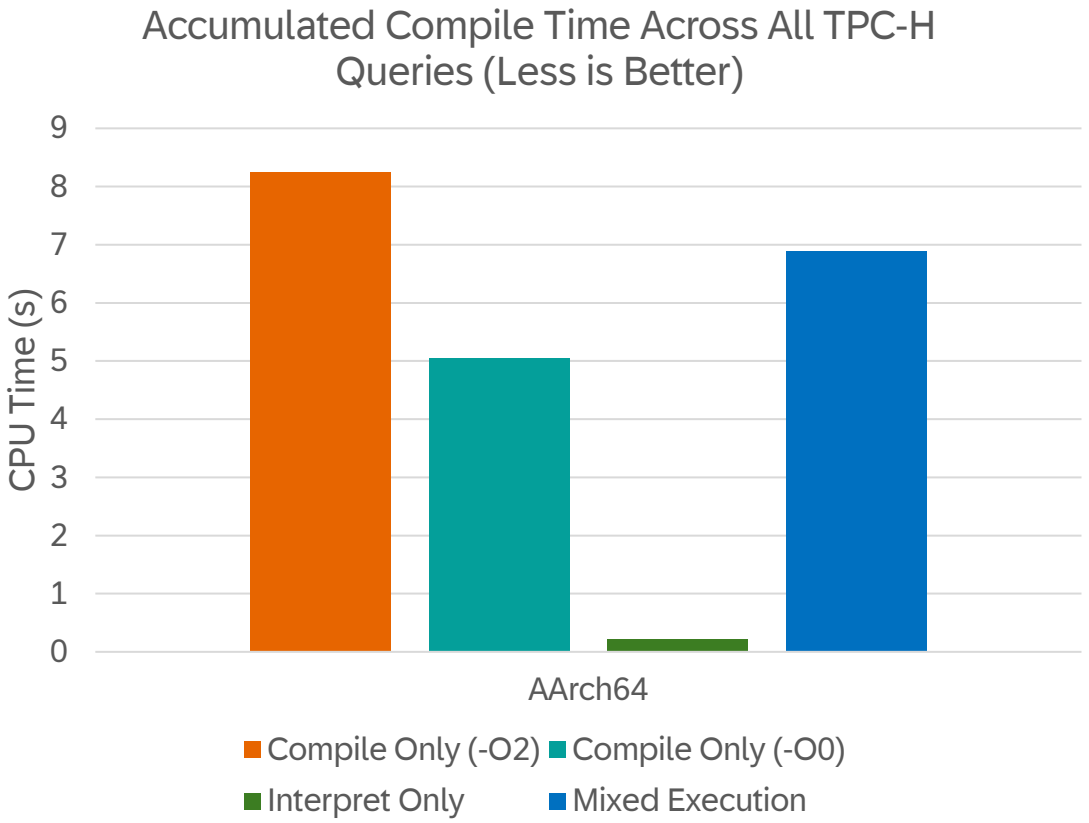
Marc Auberer
marc.auberer@sap.com

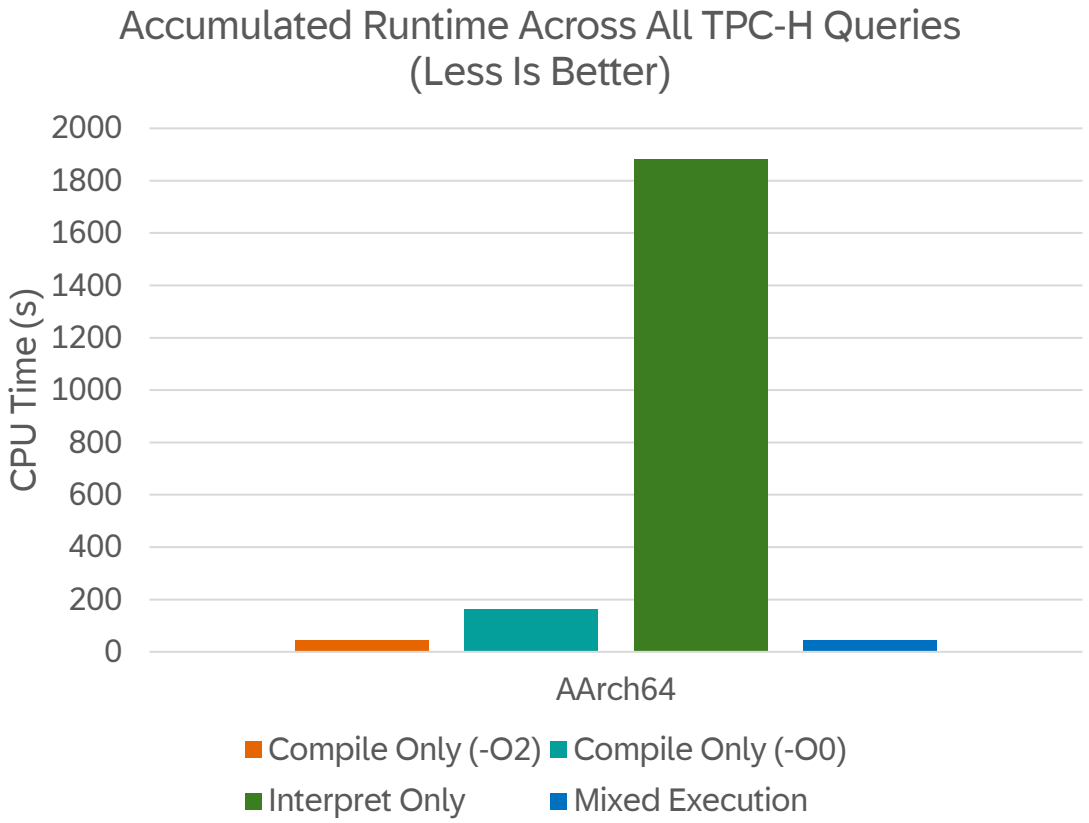
Lukas Rapp
lukas.rapp@stud.uni-heidelberg.de

SAP Bring out your best.

Introduction

Tier Performances (AArch64)

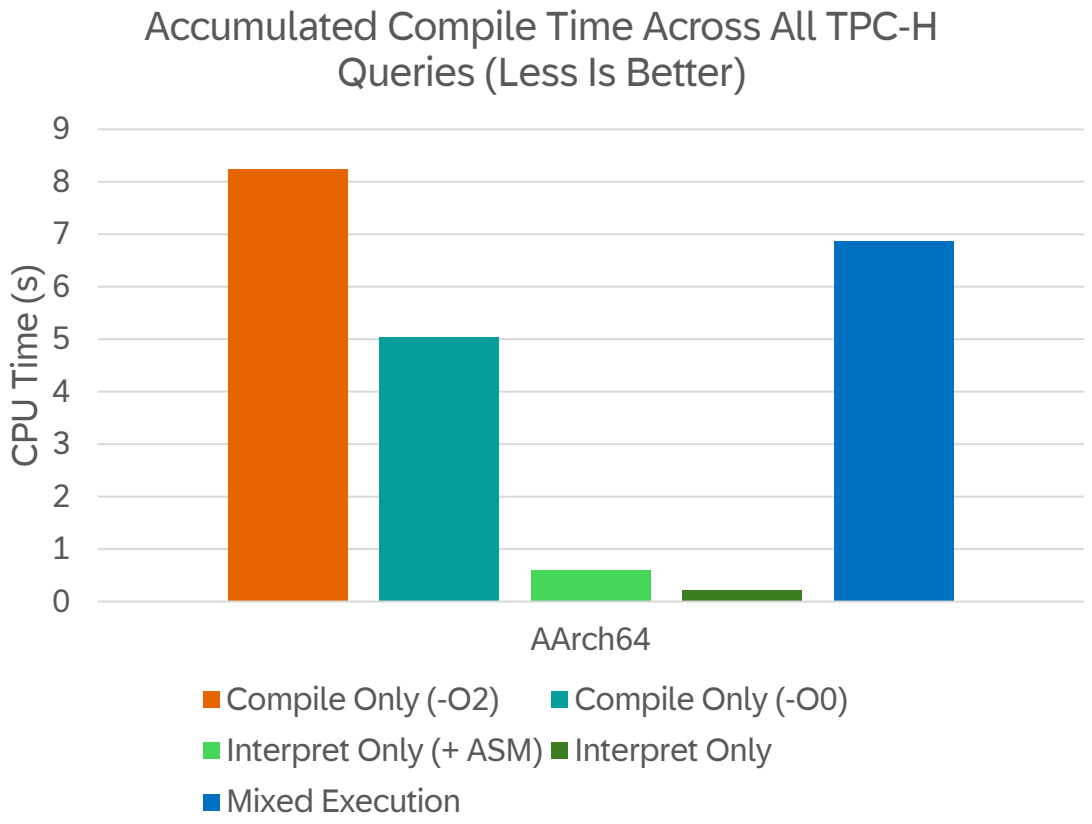


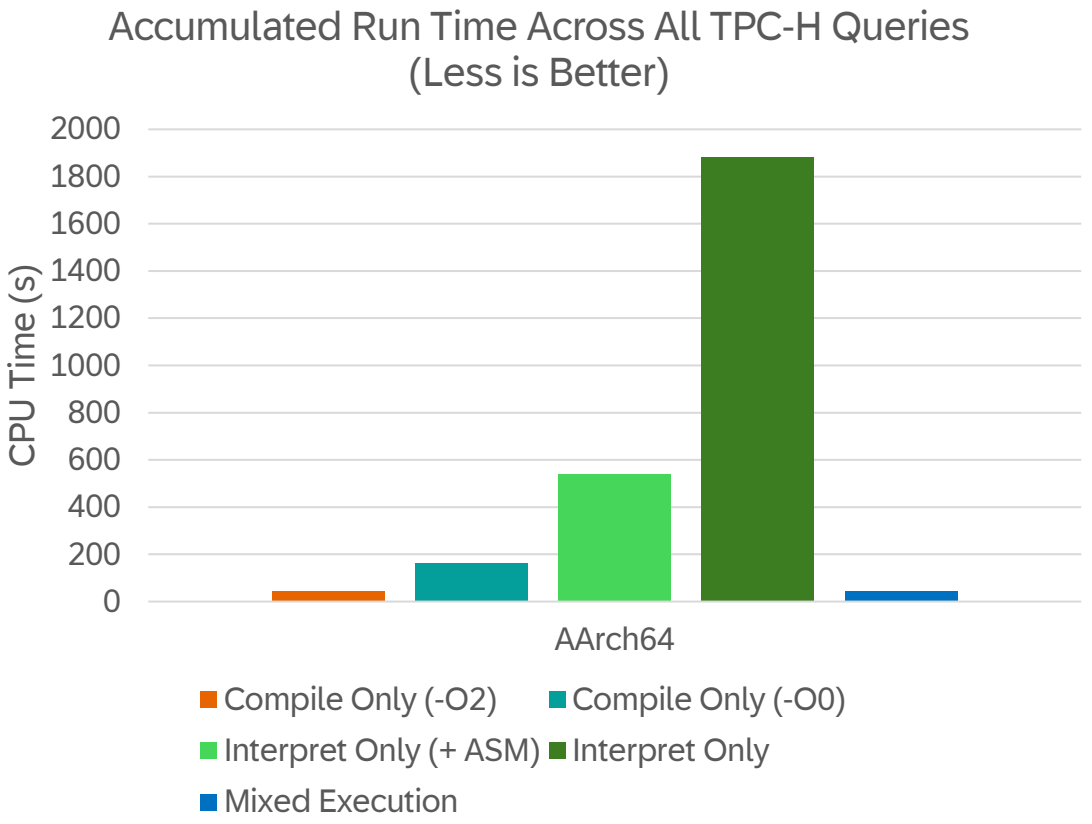


Benchmark Machine: 64 cores / 64 threads

Results

Tier Performances (AArch64)





Benchmark Machine: 64 cores / 64 threads

Snippet Management

Exception Handling

- Devirtualized function calls may produce a C++ exception that must be caught
- Uncaught exception can cause crash
- Unwind table needed for restoration of call-preserved registers (normally in `.eh_frame` in ELF executable)
- We can partially use LLVM for this
- Technical difficulties prevented implementation

Snippet Management
Exception Handling

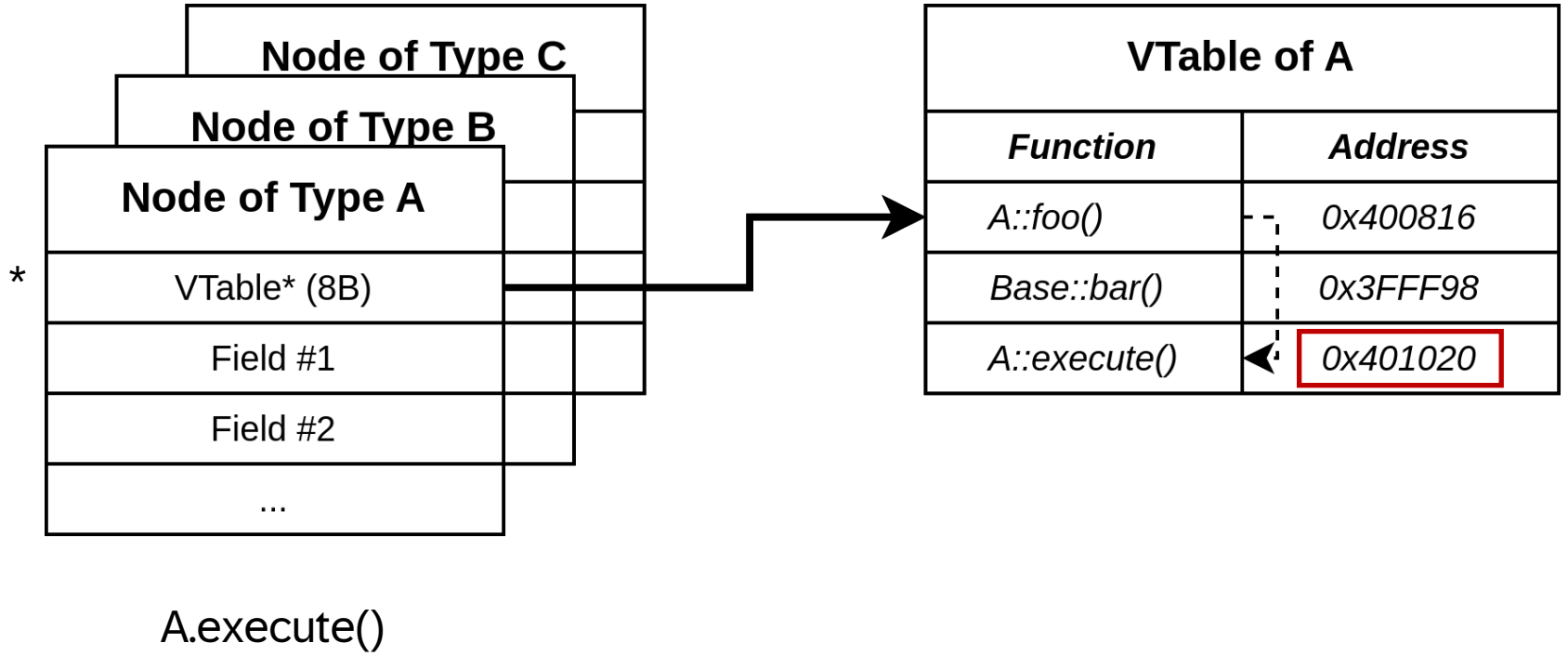
```
0x00000000: CIE
  Length:               0x00000014
  CIE ID:               0x00000000
  Version:              1
  Augmentation:         "zR"
  Code alignment:       1
  Data alignment:      -8
  Return address:       16
  Augmentation data:    pointer encoding = 0x00 (absolute pointer)
  DW_CFA_def_cfa:       r7 (rsp) +8
  DW_CFA_offset:        r16 (rip) at cfa-8

0x0000001c: FDE cie=00000000 pc=0x4f983040..0x4f98b799
  DW_CFA_def_cfa_offset: 68
  DW_CFA_def_cfa_register: r12
  DW_CFA_offset: r6 (rbp) at cfa-128
```

We can use the same DWARF FDE, that spans over all snippet functions, since the used registers, that need to be restored are always the same

Code Generation

Devirtualization



* According to Itanium C++ ABI (GCC, Clang/LLVM)

Code Generation

Devirtualization

```
uintptr_t InterpreterASTNode::getFctPtrToExecuteMethod() const {
  using ExecuteFunction = bool (InterpreterASTNode::*)(LocalInterpreter*) const;
  ExecuteFunction executeFct = &InterpreterASTNode::execute;

  // The "pointer-to-member" offset of a function (in bytes) corresponds to
  // the offset of that function in the vtable, but is offset by one.
  // Determine vtable index of execute method. Note that this is platform-dependent
  // or compiler-dependent
  intptr_t methodPtrRaw = *reinterpret_cast<intptr_t*>(&executeFct);
  methodPtrRaw &= ~1;
  intptr_t* ourVptr = *reinterpret_cast<intptr_t**>(const_cast<InterpreterASTNode*>(this));
  intptr_t methodAddress = ourVptr[methodPtrRaw / sizeof(intptr_t)];
  return static_cast<uintptr_t>(methodAddress);
}
```

Why is our interpreter slow?

Virtual Function Calls

(1) How can we get rid of these virtual calls and redundant loads/stores?

(2) Can we do this incrementally or do we have to go for all-or-nothing?

Code Generation

Compactization

- Lazy loading of values into registers
- If temporary is used immediately in next instruction, skip store and reload