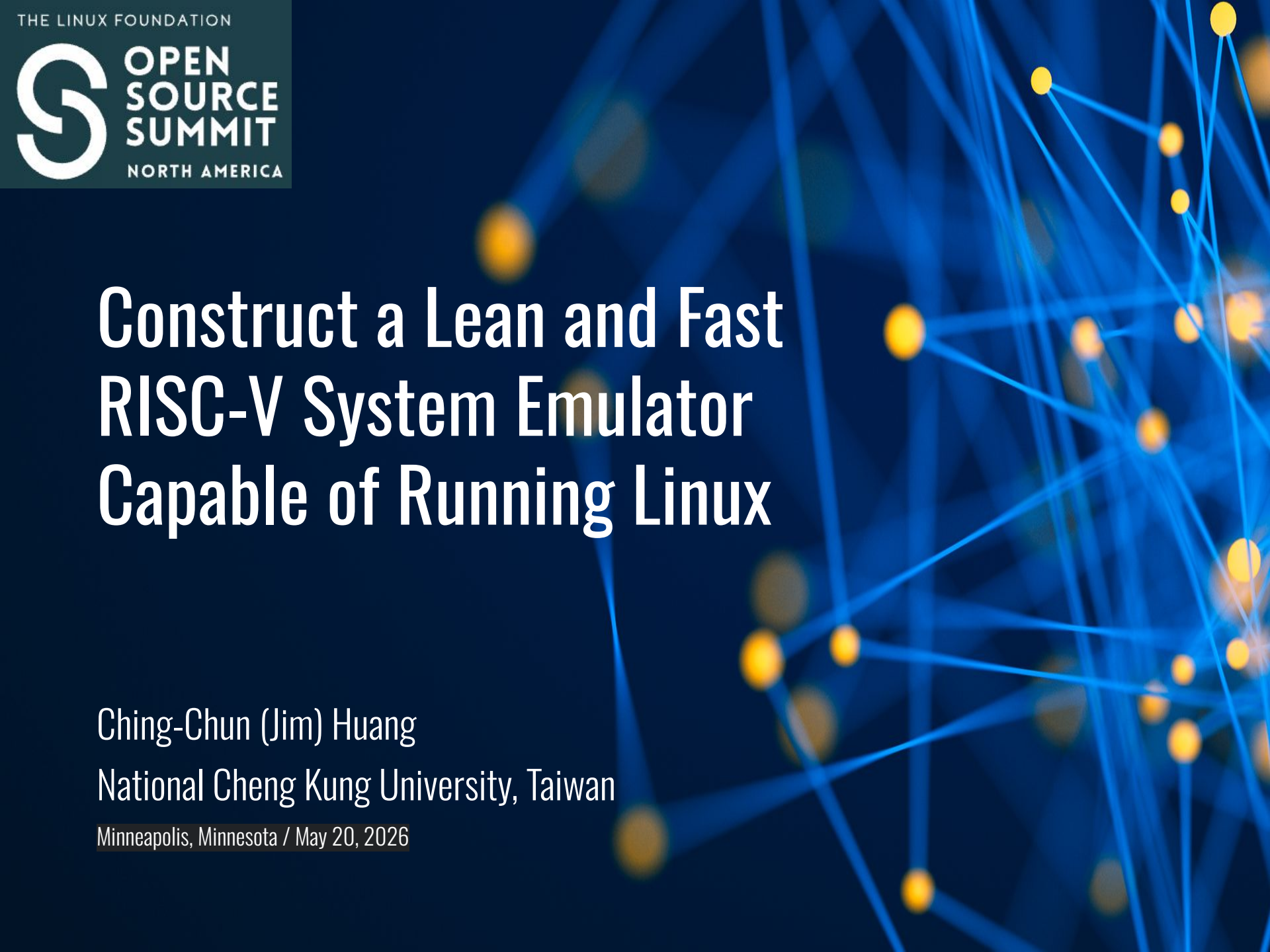



# Construct a Lean and Fast RISC-V System Emulator Capable of Running Linux

Ching-Chun (Jim) Huang

National Cheng Kung University, Taiwan

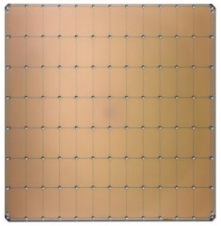
Minneapolis, Minnesota / May 20, 2026

# Why another RISC-V emulator?

- Existing emulators force tradeoffs:
  - *Spike*: correctness-oriented but slow
  - *QEMU*: fast but complex and heavyweight
  - *gem5*: accurate but expensive
- RISC-V research increasingly needs:
  - scalable multicore simulation
  - rapid experimentation
  - lightweight infrastructure
- We explore whether:
  - Design a compact and fast RISC-V emulator to boot Linux
  - lightweight runtime, facilitating VirtIO
  - Permissive license

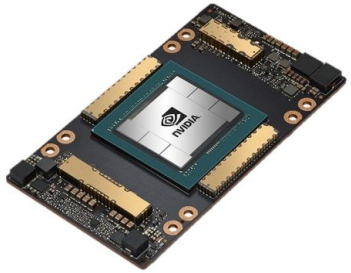
# Motivation

- Performance of Instruction Set Simulator (ISS) is a significant concern
  - In IC design flow, ISS is used to evaluate system feasibility before building electronic-system level (ESL) and register-transfer level (RTL) models.
  - ISS directly affects the speed and efficiency of software development and testing.
- Multi-core and many-core systems are crucial for modern workloads such as machine learning and computational photography.



Cerebras WSE-2
850'000 Cores

https://cerebras.net/chip/



NVIDIA A100: 6912 Cores

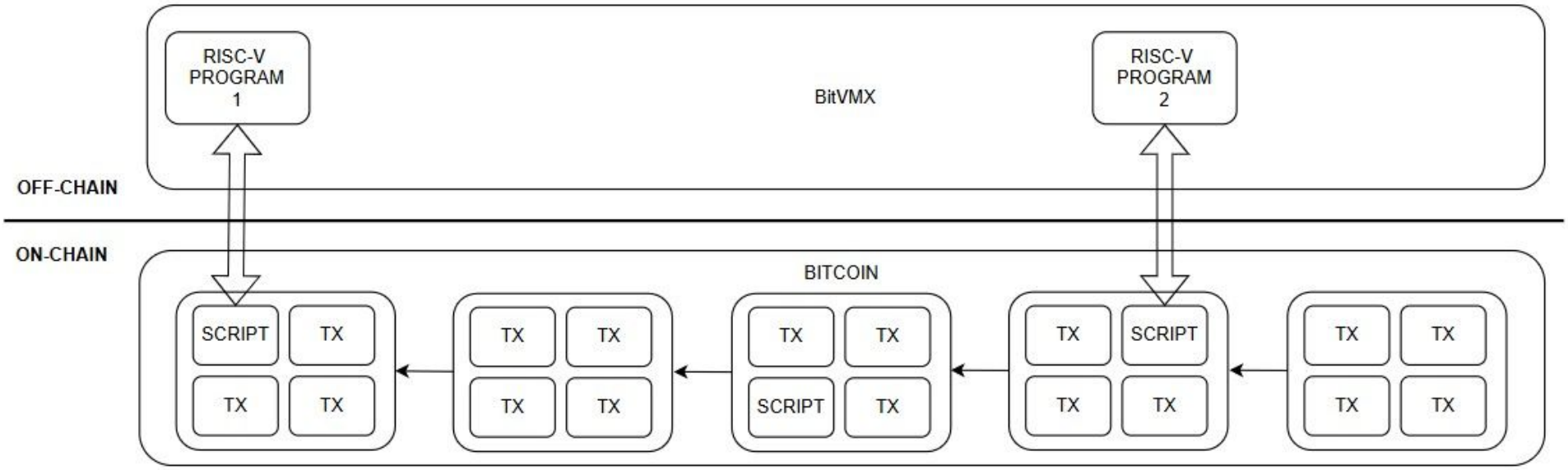
https://www.nvidia.com/de-de/data-center/a100/


Manticore
4096 Cores

https://arxiv.org/pdf/2008.06502.pdf

# Motivation(cont.)

- Cryptocurrency implementations use RISC-V as well.
  - BitVMX extends Bitcoin with off-chain smart contract execution and on-chain dispute resolution, without changing the Bitcoin protocol. Built on the BitVM concept of disputable computation, it generalizes the model to support multi-party computation and full RISC-V program execution.

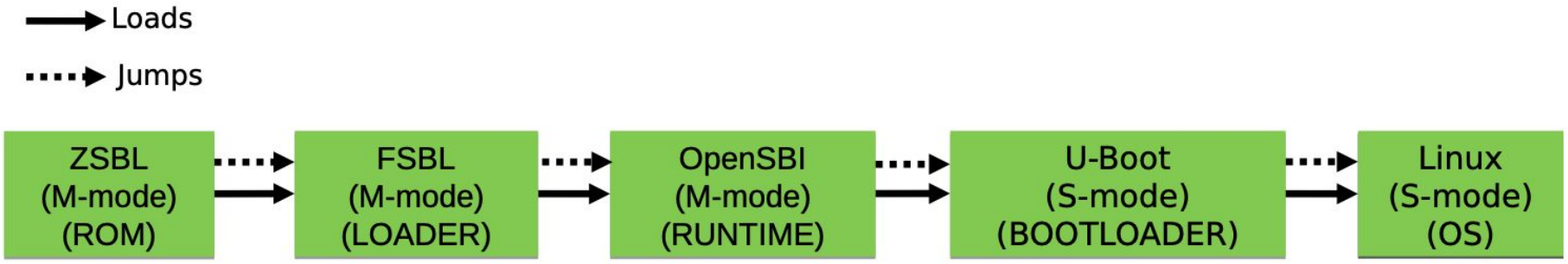



Source:
https://cexplorer.io/article/bitvmx-cardano-smart-contracts-and-the-future-of-bitcoin-defi

# Motivation(cont.)

- RISC-V is being adopted as a domain-specific ISA
  - Efficient and portable execution engines are increasingly important.
  - Modern Java and JavaScript runtimes use advanced techniques like tiered JIT compilation, but rarely explored in DBT-based RISC-V ISS implementations.
- Explore a new balance between: performance, scalability, and implementation simplicity.
- Address performance issue, with a focus on dynamic binary translation (DBT).
  - Refined tiered JIT compilation, targeting low-level VM.
  - A infrastructure for system simulator, eliminating execution latency.
  - Reduced resource consumption per RISC-V hart makes scalable multicore simulation more practical.
  - Small and clean codebase for evaluation and research purposes.

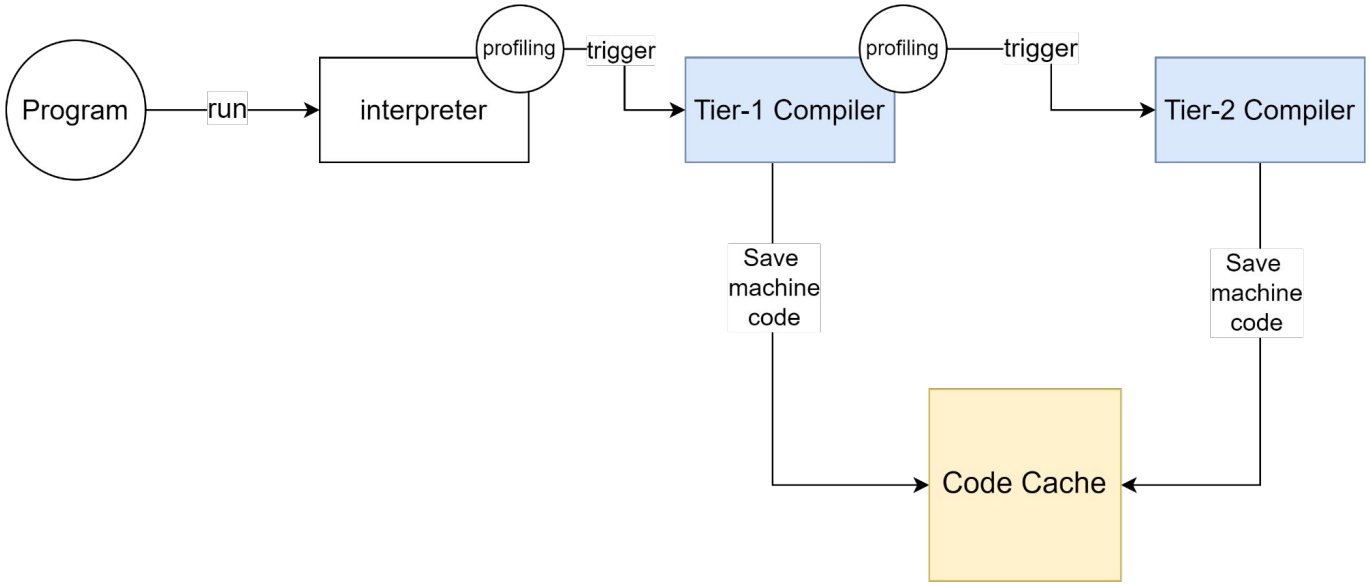
# Boot sequence of RISC-V/Linux




- For faster Linux booting, we can bypass OpenSBI and U-Boot entirely by loading the Linux kernel image directly into memory.
- To reduce instruction execution overhead, efficient instruction dispatch is essential. We therefore introduce tiered JIT compilation to accelerate frequently executed paths while keeping translation overhead low.
- To keep the emulator as compact and lightweight as possible, only essential devices are implemented, including UART and VirtIO.

# Multi-tier JIT Compilers

- primary tiers commonly found in multi-tier JIT compilation:
  - **Interpreter**: Execute source code directly, without prior compilation to machine code.
  - **Tier-1 JIT Compiler**: Specialize in producing moderately optimized machine code from the intermediate representation (IR) generated by the interpreter.
  - **Tier-2 JIT Compiler**: Apply more aggressive optimizations to produce highly optimized code.
  - **Tier-3 and Beyond**: Some environments implement additional tiers, where each subsequent tier applies increasingly sophisticated optimizations.
- Balance compilation overhead and generated code performance.
- Can tiered JIT improve DBT-based RISC-V system emulation?



# Related Work: VM with Tiered JIT Compilation

- The tiered JIT compilation technique has been widely adopted in many high-performance virtual machines for various high-level programming languages.

| Virtual Machine | Traget Language |
|---|---|
| HotSpot JVM | Java |
| OpenJ9 | Java |
| V8 | JavaScript |
| JavaScriptCore | JavaScript & WebAssembly |
| PyPy | Python |

# Why Tiered JIT for ISS?

- Traditional DBT systems often:
  - optimize too little
  - or optimize too late
- Linux workloads exhibit:
  - stable hotspots
  - long-running loops
  - recurring execution paths
- Therefore:
  - fast initial translation matters
  - aggressive optimization should target only persistent hotspots

# Design Goals

- low warmup latency
- scalable memory usage
- hotspot-aware optimization
- Linux-capable system emulation

# Objectives

- Realize tiered JIT compilation for performance improvements.
  - Interpreter-first strategy.
  - Precisely identify and categorize hotspots.
  - Integrate two-tier JIT compiler with minimal effort.
- Reduce system resource usage for commonly executed paths.
- Offload the clearly identified performance hotspots to LLVM for aggressive compiler optimizations
  - Additionally, the final tiered compiler can be made pluggable.
- Integrate system emulation (e.g., booting the Linux kernel and validating various Linux user programs)
  - consolidate the usability of the proposed tiered JIT compilation.

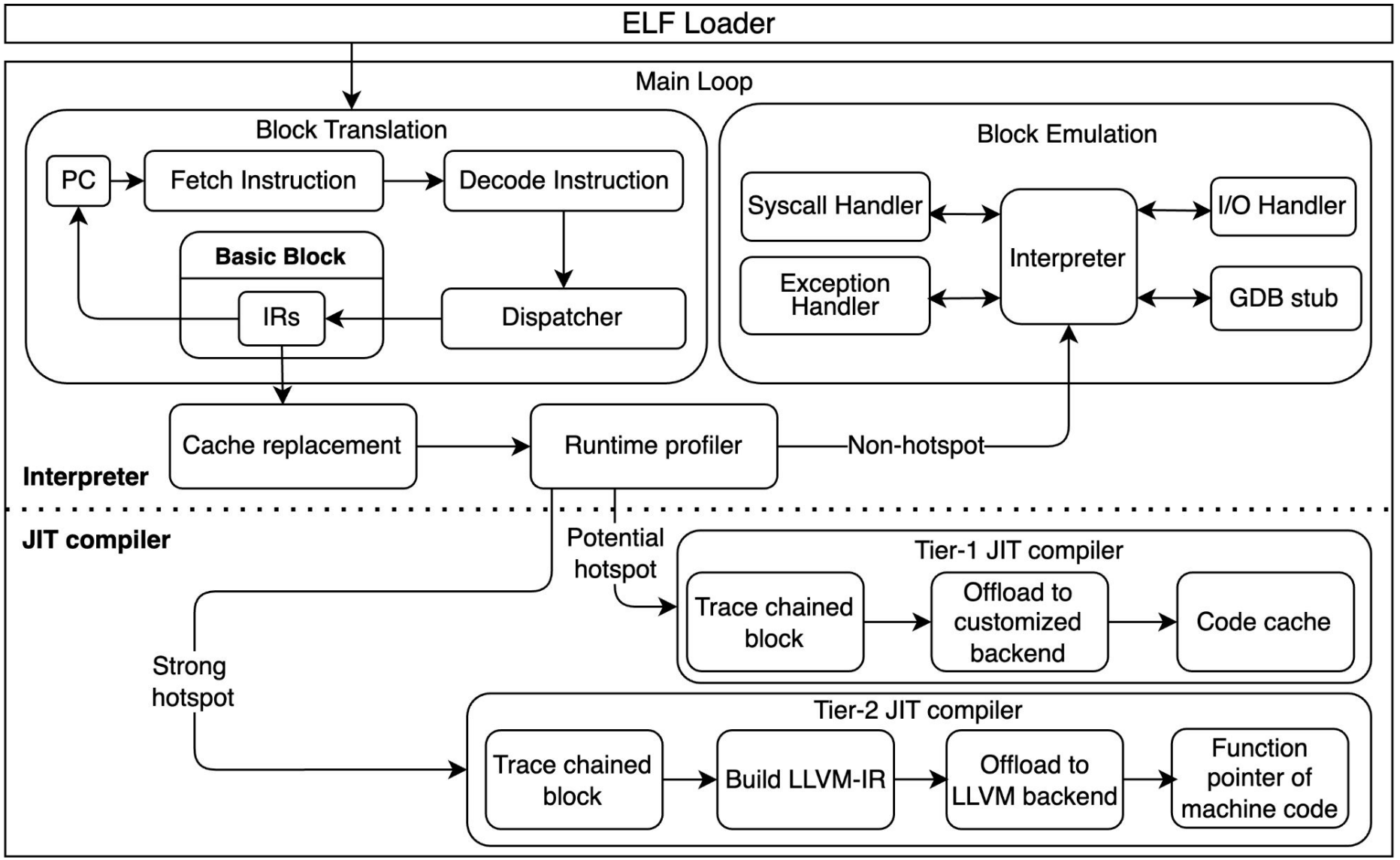
# Key Components: https://github.com/sysprog21/rv32emu

- Interpreter
  - Collect profiling data
- Runtime profiler
  - is the core coordinator
  - Analyze profiling data
  - Precise hotspot identification
  - Trigger JIT compilation process
- Two-tier JIT compilation
  - Tier-1 JIT compiler (T1C)
    - optimization with **low overhead**
    - The quality of generated code is **acceptable**.
  - Tier-2 JIT compiler (T2C)
    - Aggressive optimization with **relatively large overhead**
    - The quality of generated code is **better**.
  - Precise hotspot detection is the key to balancing compilation overhead and execution speed.

"rv32emu" is hard to pronounce, so let's just call it "*emu*" for short.

The emu (/ˈiːmjuː/; Dromaius novaehollandiae) is a species of flightless bird endemic to Australia, where it is the tallest native bird.

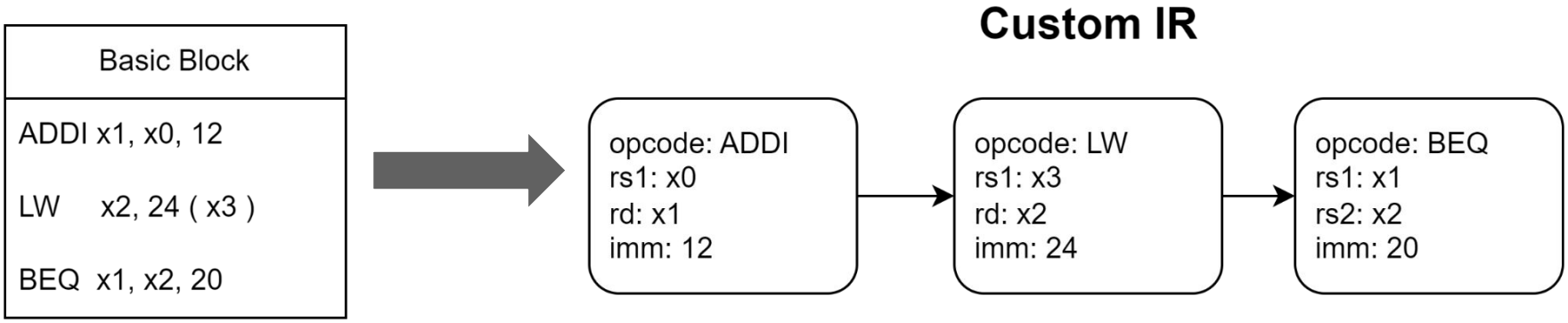
# System Architecture



# Interpreter

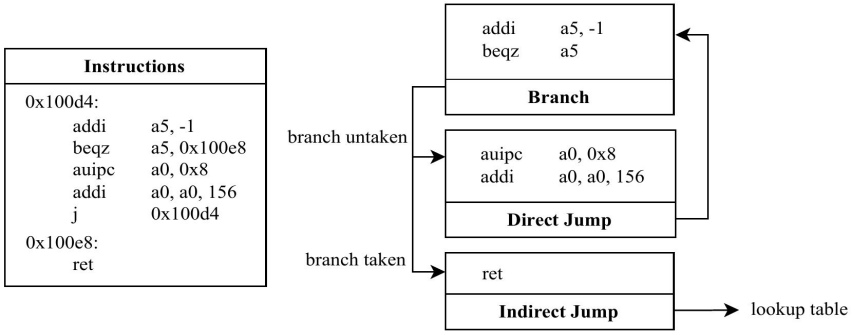
# Custom Intermediate Representation (IR)

- Considering the potential translation of our custom IR into other forms such as **LLVM IR**, the simplified linear IR design enhances compatibility and ease of mapping into other IR forms.
- The structure of linear IR is a singly-linked list, facilitating easier tracing of the linear sequence and optimization by deleting or replacing some nodes.



# Block Chaining

- Block chaining **connects** these basic blocks along an execution path, significantly improving the efficiency of locating subsequent basic blocks during emulation.
- Direct jump and branch
  - Direct jump target and branch target is determined by constant value.
  - Chain the pervious block to the current block.
- Indirect jump
  - Jump target is determined by register value.
  - Lookup **branch history table** and jump.
- Chained blocks represents the **execution path** for T1C and T2C.



# Macro-Operation Fusion

- An optimization strategy that consolidates multiple adjacent instructions into a single fused instruction.
- Based on previous research and an analysis of prevalent instruction patterns among the benchmarks, we chose a set of potential candidates.

| Instruction sequence (X means don't care) |
|---|
| AUIPC r1, imm; ADDI r1, r1; |
| SW X, imm(X); SW X, imm(X); ... |
| LW X, imm(X); LW X, imm(X); ... |
| LUI X, imm; LUI X, imm; ... |
| SLLI/SRLI/SRAI X, imm; SLLI/SRLI/SRAI X, imm; ... |

# Macro-Operation Fusion (cont.)

- Macro-operation fusion effectively diminishes instruction dispatching overhead by reducing the number of instructions.

| Benchmark | original | optimized | instruction reduction |
|---|---|---|---|
| numeric sort | 1,280,007,416 | 1,101,243,105 | 13.97% |
| dhrystone | 1,410,009,176 | 920,006,341 | 34.75% |
| qsort | 2,936,164,776 | 2,598,217,533 | 11.51% |
| miniz | 4,745,983,821 | 4,468,532,798 | 5.85% |
| primes | 7,114,988,162 | 6,446,379,470 | 9.40% |
| sha512 | 7,617,545,069 | 6,131,536,325 | 19.51% |
| FP emulation | 10,473,559,475 | 9,928,114,210 | 5.21% |

# Constant Propagation and Constant Folding

- In the context of the RISC-V instruction set, specific instructions are dedicated to generating or loading constant values, such as **AUIPC** and **ADDI** instructions.
- Replace registers with known constants and subsequently optimizing arithmetic operations when both source operands are identified as constants.

```
0x100f0: auipc gp, 0x18
0x100f4: addi  gp, gp, 0x360
0x100f8: addi  a0, gp, 0x7cc
0x100fc: auipc a2, 0x1a
0x10100: addi  a2, a2, 0x568
```

```
0x100f0: li gp, 0x280f0
0x100f4: li gp, 0x28450
0x100f8: li a0, 0x28c1c
0x100fc: li a2, 0x2a0fc
0x10100: li a2, 0x2a334
```

# Reduce Memory Usage and Footprint

- Manage basic blocks via explicit cache replacement
  - Least *Recently* Used (LRU)
    Low overhead, ignore frequency and does not consider recent history.
  - Least *Frequently* Used (LFU)
    Record using frequency, accumulates data you are no longer using.
  - Adaptive Replacement Cache (ARC)
    Combines the best of LRU and LFU, plus some novel tricks.
- Considering JIT compilation, *modified ARC* is chosen for its ability to detect hotspots.
- Employ a memory pool for deterministic allocations on basic blocks and IR singly-linked list.

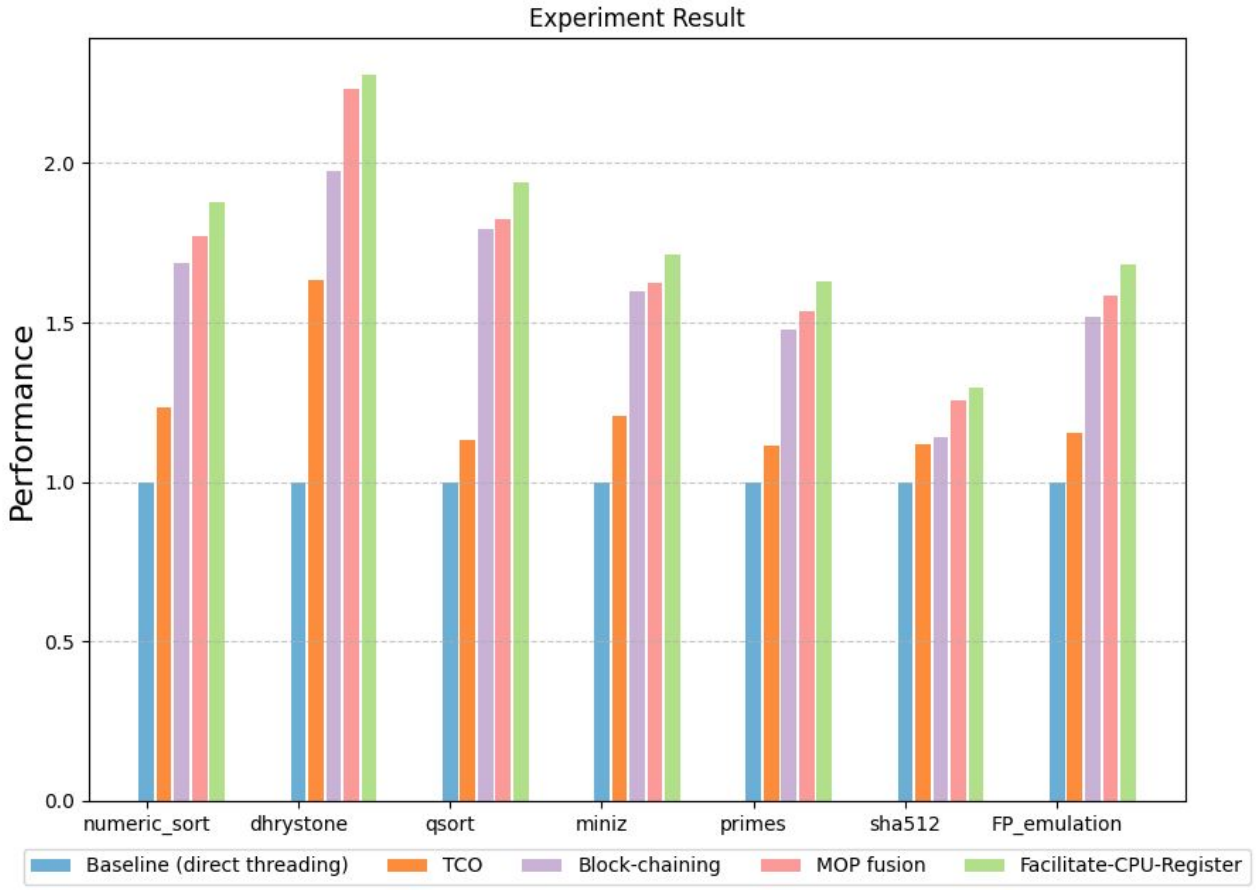
# Interpreter Evaluation – Benchmarks

- **numeric sort:**
  Integer arithmetic and memory bandwidth by sorting arrays of random integers.
- **dhrystone:**
  String processing, integer arithmetic, and bitwise operations designed to simulate a typical system's workload.
- **qsort:**
  Recursive function calls, integer arithmetic, and memory access by sorting large arrays of data.
- **miniz:**
  Integer arithmetic, memory access, and bitwise operations by compressing and decompressing data.
- **primes:**
  Integer arithmetic and algorithm efficiency by finding and verifying prime numbers within a specified range.
- **sha512:**
  Integer arithmetic, bitwise operations, and memory access by computing SHA-512 hash values for large data sets.
- **FP emulation:**
  Floating-point arithmetic by emulating floating-point calculations using integer arithmetic, which is useful for systems without hardware floating-point support.

| Benchmark | Dynamic Instruction Counts |
|---|---|
| numeric sort | 1,280,007,416 |
| dhrystone | 1,410,009,176 |
| qsort | 2,936,164,776 |
| miniz | 4,745,983,821 |
| primes | 7,114,988,162 |
| sha512 | 7,617,545,069 |
| FP emulation | 10,473,559,475 |

# Interpreter Evaluation

- Each mechanism in interpreter contributes to enhancing the interpreter's performance.



# Tiered JIT Compilation

# Overview of Tiered JIT Compilation

- Translate IR into host binary to improve efficiency.
- Compile hotspots only
  - The overhead of JIT compilation is significant.
  - The benefit from JIT compilation might be canceled by its overhead.
- More optimization makes more time delayed.
  1. Trigger the simple compiler which does less optimization but emits code quickly.
  2. Trigger the powerful ones but take more time.

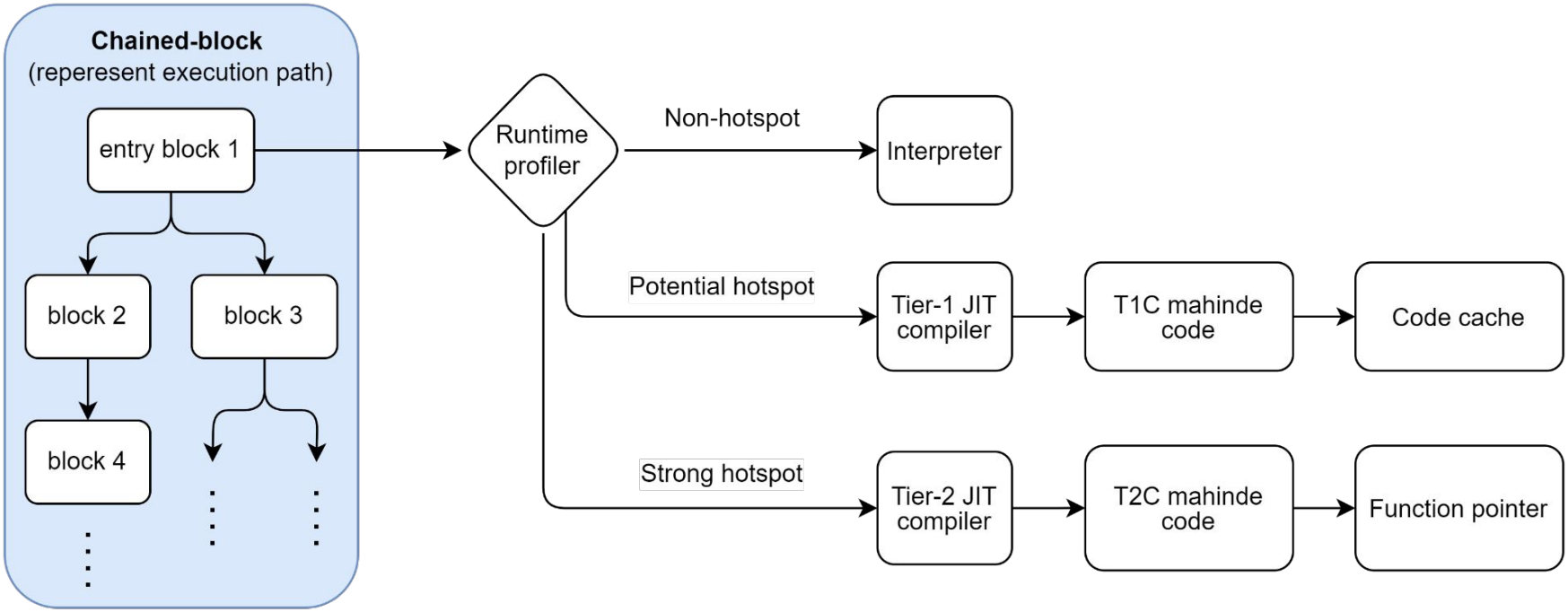
- How to categorize the hotspot to appropriate tier of JIT compilation?

# Runtime Profiler

- Potential hotspot
  - The block which execution count exceeds the threshold.
  - The block is a part of loop.

- Strong hotspot
  - The execution count of T1C-generated code surpasses the threshold.

# Runtime Profiler (cont.)

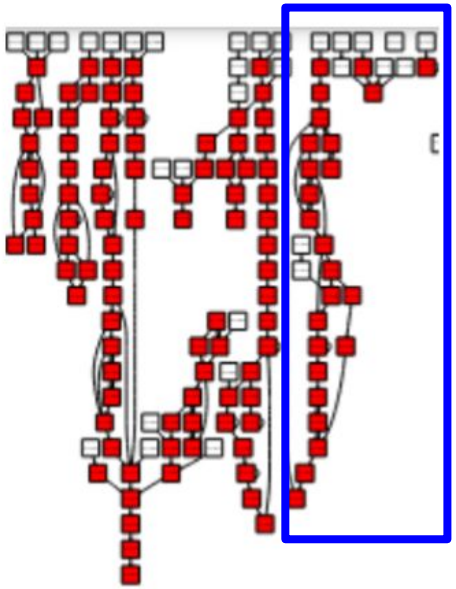


# Runtime Profiler – Loop Detection

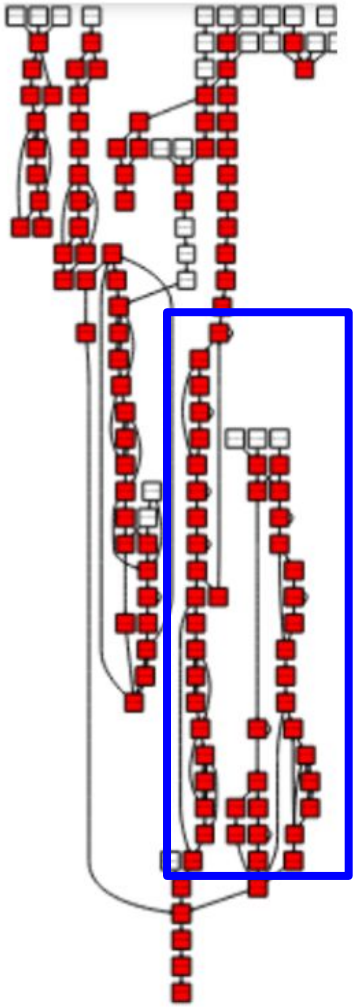
- have evaluated different methods for loop detection:
  - Based on backward jump detection (deprecated)
  - Trace interpreter execution path
- The former sometimes breaks down the inner nested loop.

# Runtime Profiler – Loop Detection (cont.)

Backward jump detection

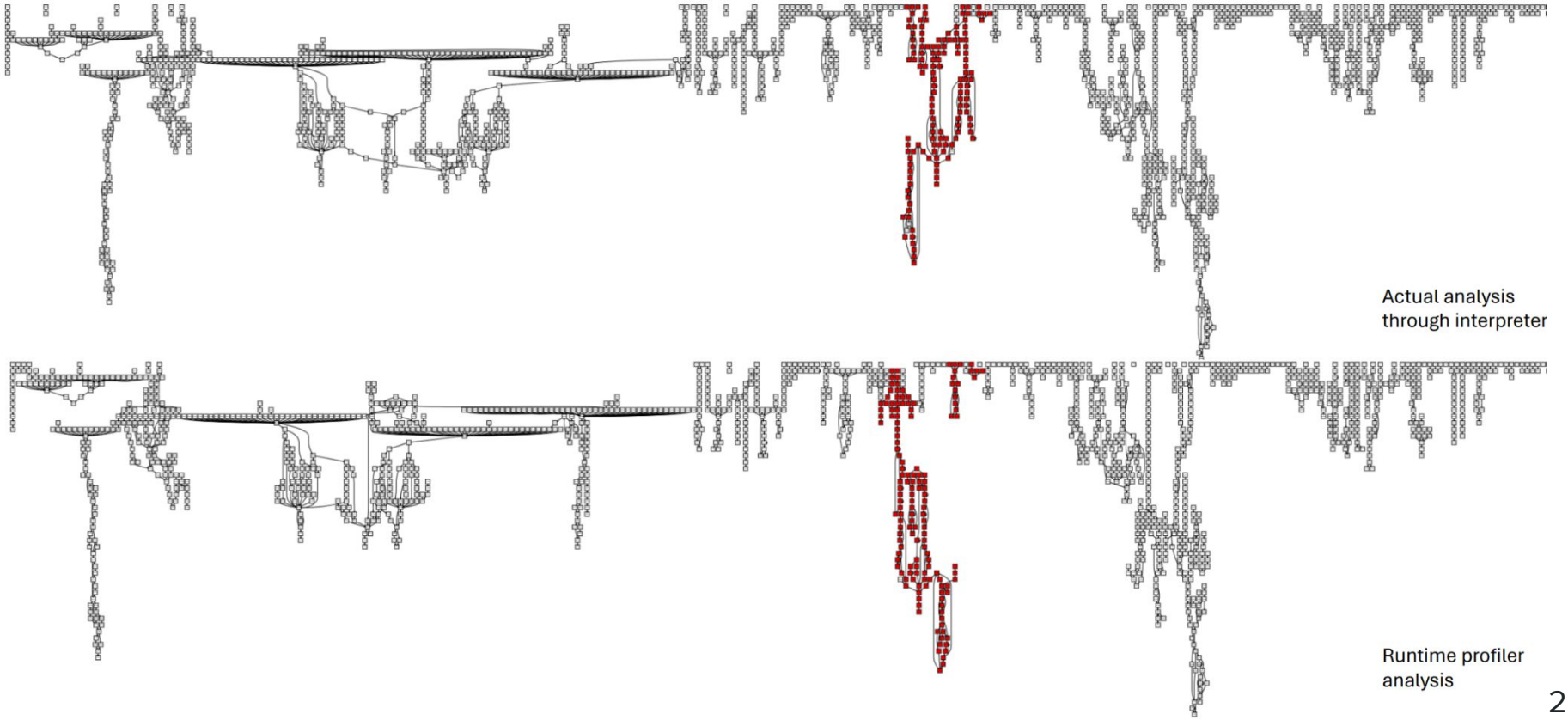


Purposed loop detection



# The Effectiveness of Runtime Profiler

- In **emfloat** benchmark, the hotspots derived from the interpreter and the one derived from the runtime profiler are identical. (**138** basic blocks as strong hotspots within **1,601** basic blocks)

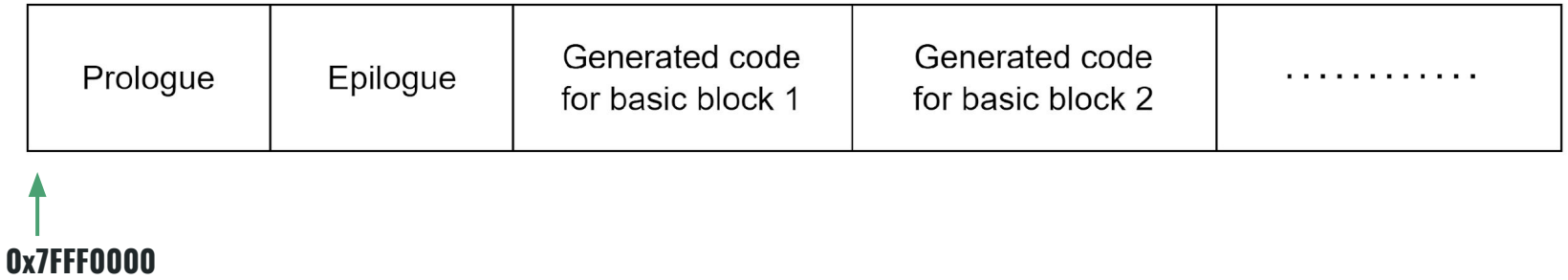


# Tier-1 JIT Compiler - Overview

- Quick, low cost translator
  - Translate IR into host binary with **low overhead**.
  - Apply **basic optimization** to generate machine code with **acceptable quality**.
    - Linear-scan register allocation
    - Branch prediction (indirect jump improvement)
- Target to x86-64 and Arm64 architectures.

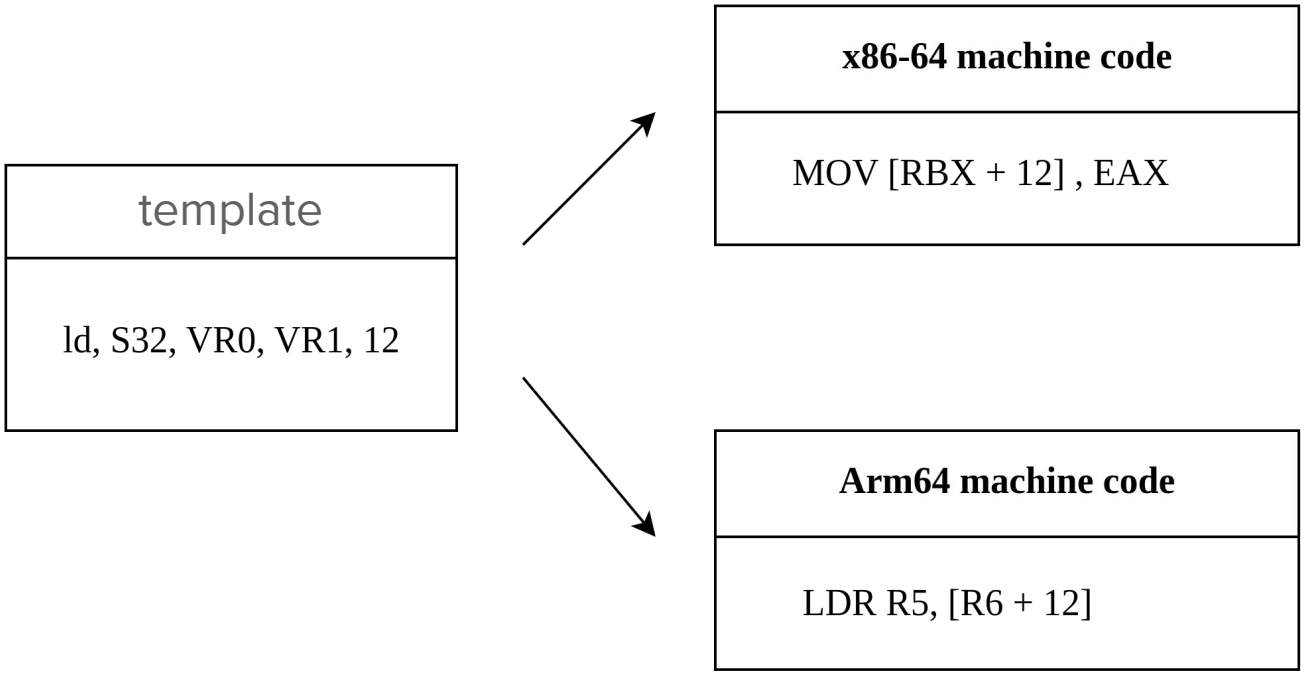
# Tier-1 JIT Compiler - Implementation

- Prepare the code cache
  - Create a readable, writable, and executable memory with **mmap** syscall.
  - Provide function prologue and epilogue to preserve host register value.
  - Flush (least-frequently-used) the code cache if the code cache is full.
- Execution flow: Prologue > entry block > next chained block > … > Epilogue



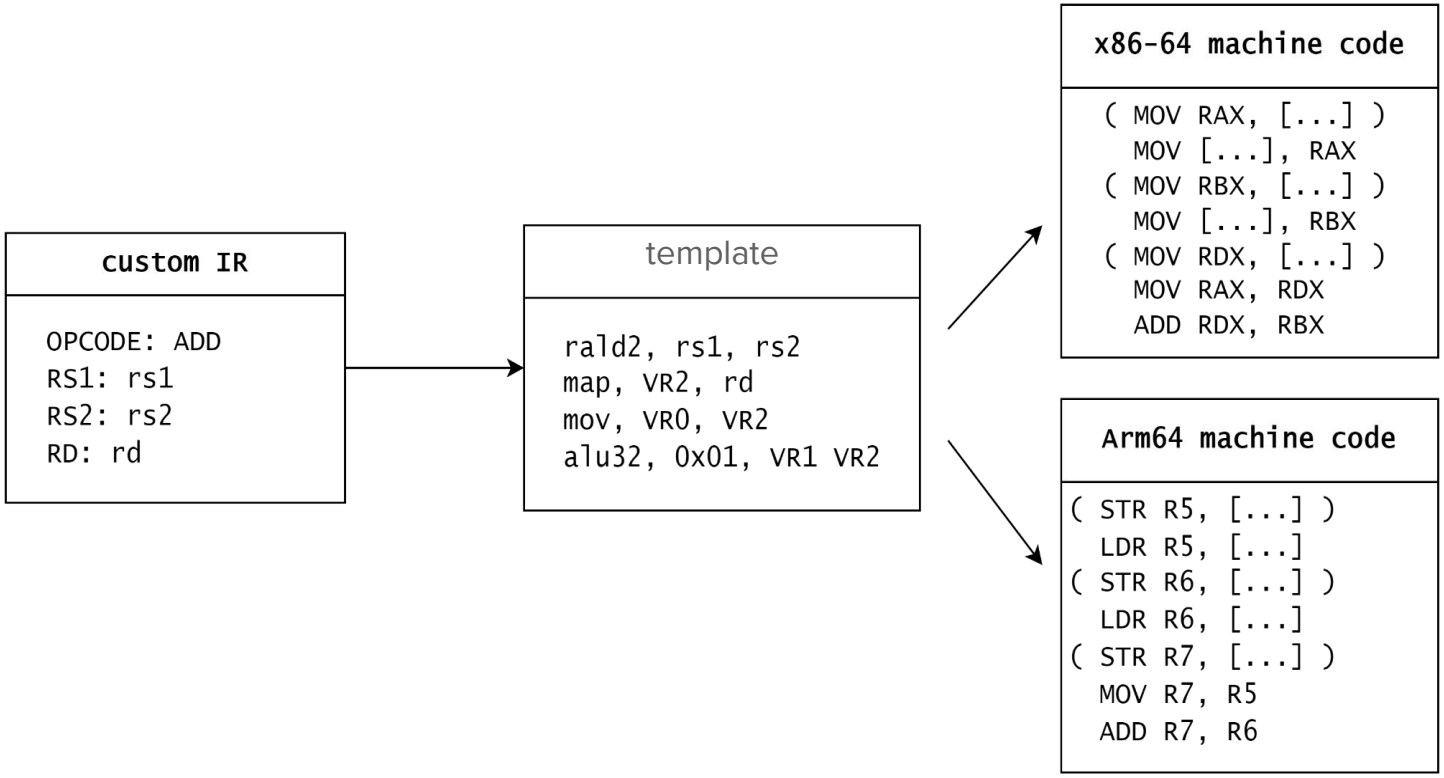
# Tier-1 JIT Compiler - Architecture-Independant Template

- API-like template to emit different binary of multiple architectures.



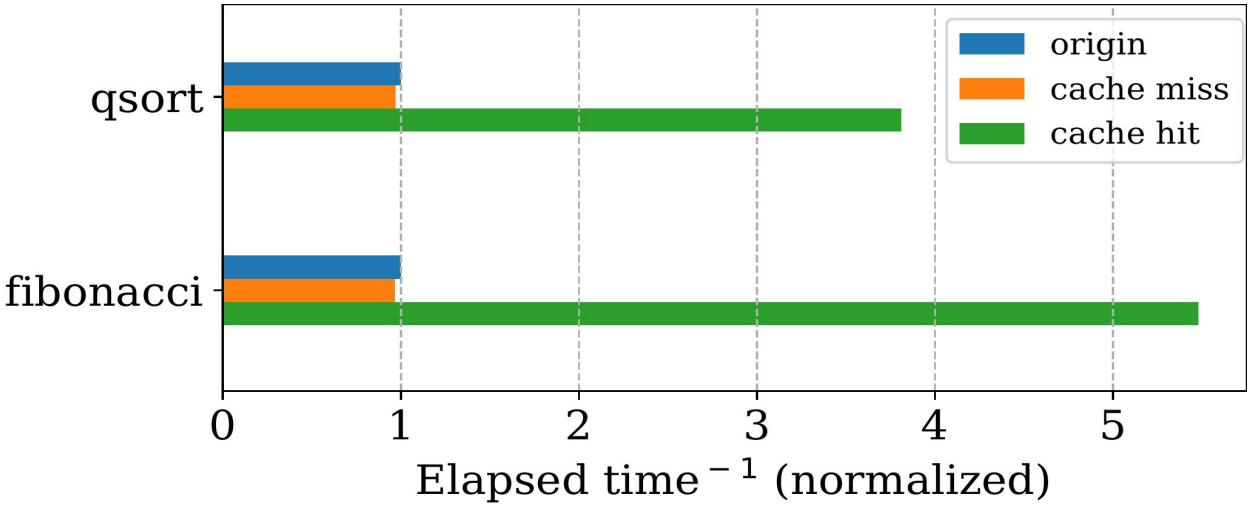
# Tier-1 JIT Compiler - Register Allocation

- To find the appropriate register to be spilled.
- The effectiveness is significant due to the limited number of available registers.



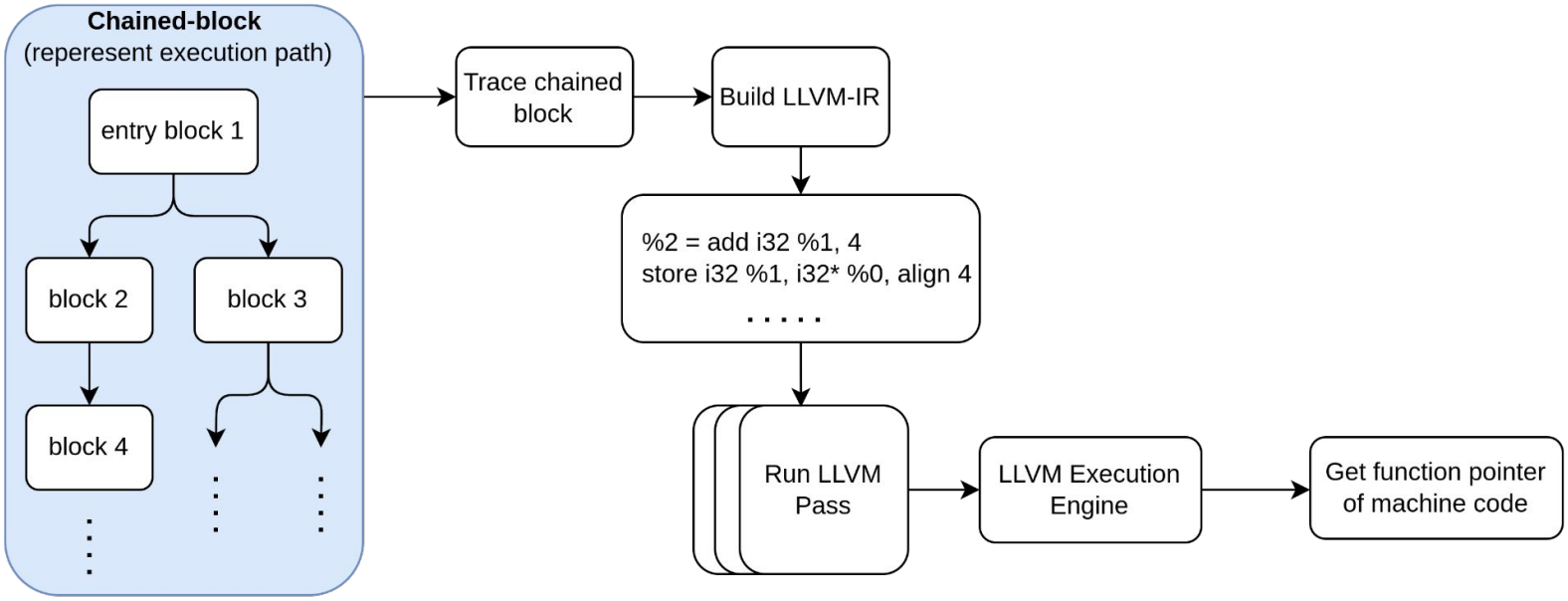
# Tier-1 JIT Compiler - Branch Prediction (Indirect Jump)

- Indirect jump is a critical issue for many DBT researches.
  - The target address of indirect jump is determined by the register value in run time, so the target block cannot be chained directly.
  - The execution of JIT compilation should terminate when encountering indirect jump and convert back to interpreter.
- To address this problem, we record the history when interpreter executing. Then, insert comparison function with the most frequently jump target.



# Tier-2 JIT Compiler - Overview

- Optimized dynamic translator
  - Transforms custom IR within chained block into LLVM IR via LLVM-C API
  - Offload built LLVM IR to the LLVM backend, where it undergoes optimization through several selected LLVM passes
  - Pass Optimized LLVM IR to the LLVM execution engine and get a function pointer of the generated machine code
- **Background compilation** to lower compilation overhead.

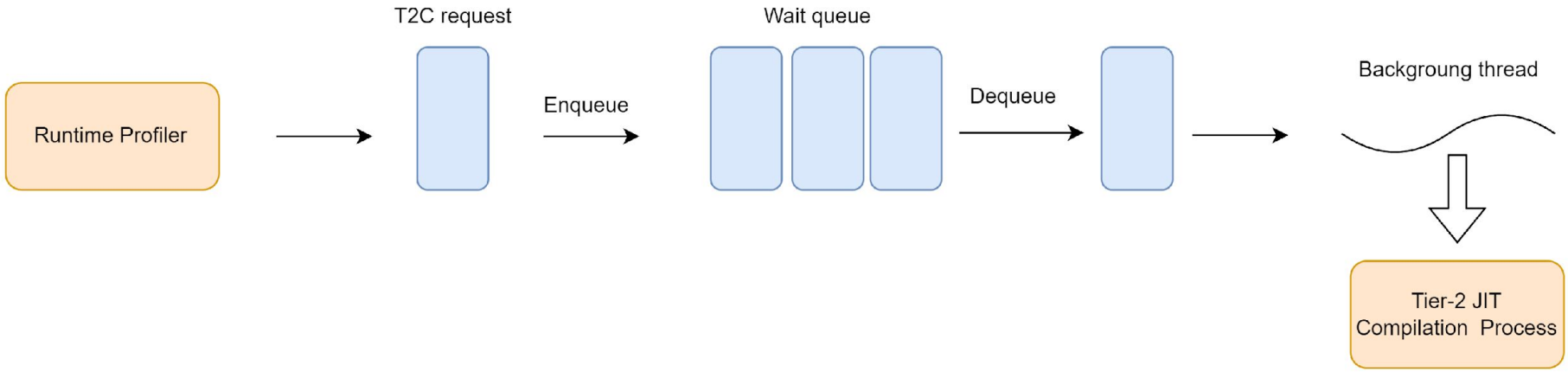


# Tier-2 JIT Compiler - LLVM JIT

- Our IR is close to LLVM IR.
  - Build LLVM IR module by LLVM-IR C API.
- Select LLVM Pass based on other researches and our evaluation:
  - Optimization level (**O3**)
  - Early common subexpression elimination
  - Memory copy optimization

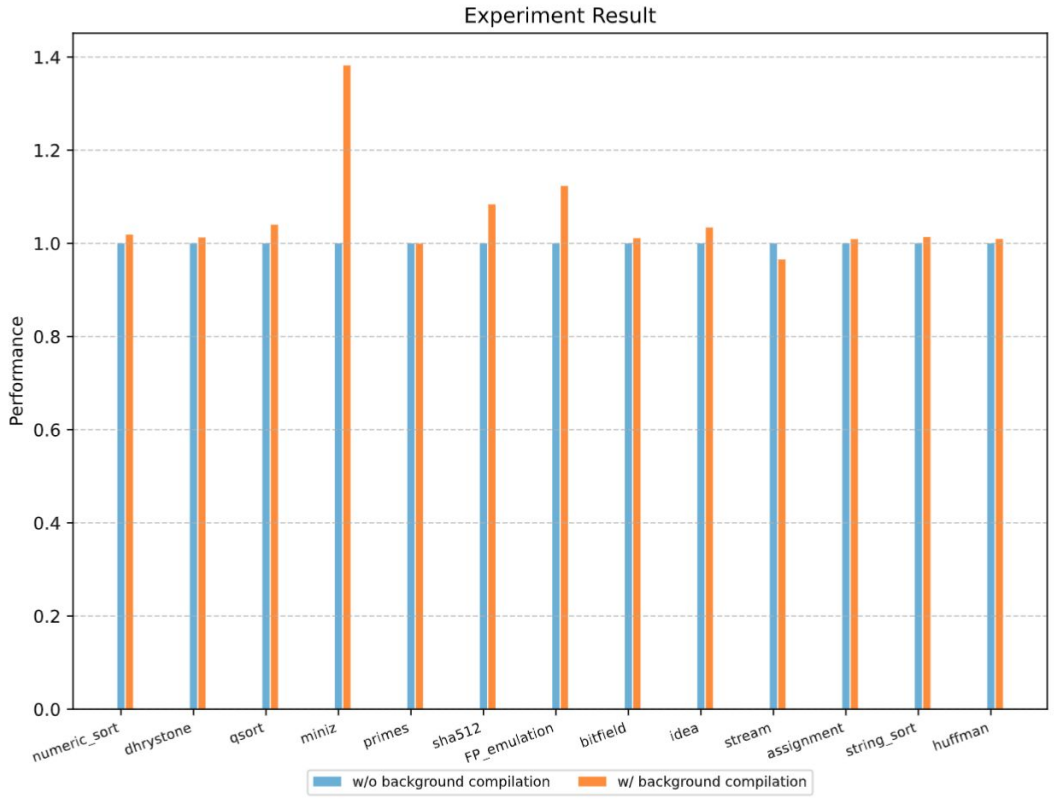
# Tier-2 JIT Compiler - Background Compilation

- Wait queue
  - Runtime profiler adds a T2C compilation request to the wait queue once identifying a strong hotspot.
- Background Thread
  - Continuously monitor the wait queue and trigger T2C to process the compilation request.
  - Notifies the main thread upon request completion by updating a flag.



# The Effectiveness of Background Compilation

- Background compilation substantially benefits cases with short execution times and frequent T2C compilation invocations.
- Lead to more frequent use of T1C-generated code.



# Performance Evaluation & Discussion

# Measurements

- Compilation time, elapsed time: **HRT (high resolution timer), perf**
- Memory consumption: **valgrind (massif)**

- **QEMU** version v9.0.0

# Benchmarks

- **bitfield**
  Bitwise operations and integer arithmetic by performing a series of bitfield manipulations.
- **idea**
  Integer arithmetic, bitwise operations, and memory access by encrypting and decrypting.
- **stream**
  Memory bandwidth and data transfer by performing a series of operations on large arrays.
- **assignment**
  Integer arithmetic and memory access by implementing an assignment problem algorithm.
- **string sort**
  String handling, memory access, and sorting algorithms by sorting arrays of random strings.
- **huffman**
  Integer arithmetic and memory access by compressing and decompressing data using Huffman coding.

| Benchmark | Dynamic Instruction Counts |
|---|---|
| bitfield | 10,680,145,644 |
| idea | 30,461,520,799 |
| stream | 48,075,191,816 |
| assignment | 52,083,333,383 |
| string sort | 61,299,322,263 |
| huffman | 80,402,895,889 |

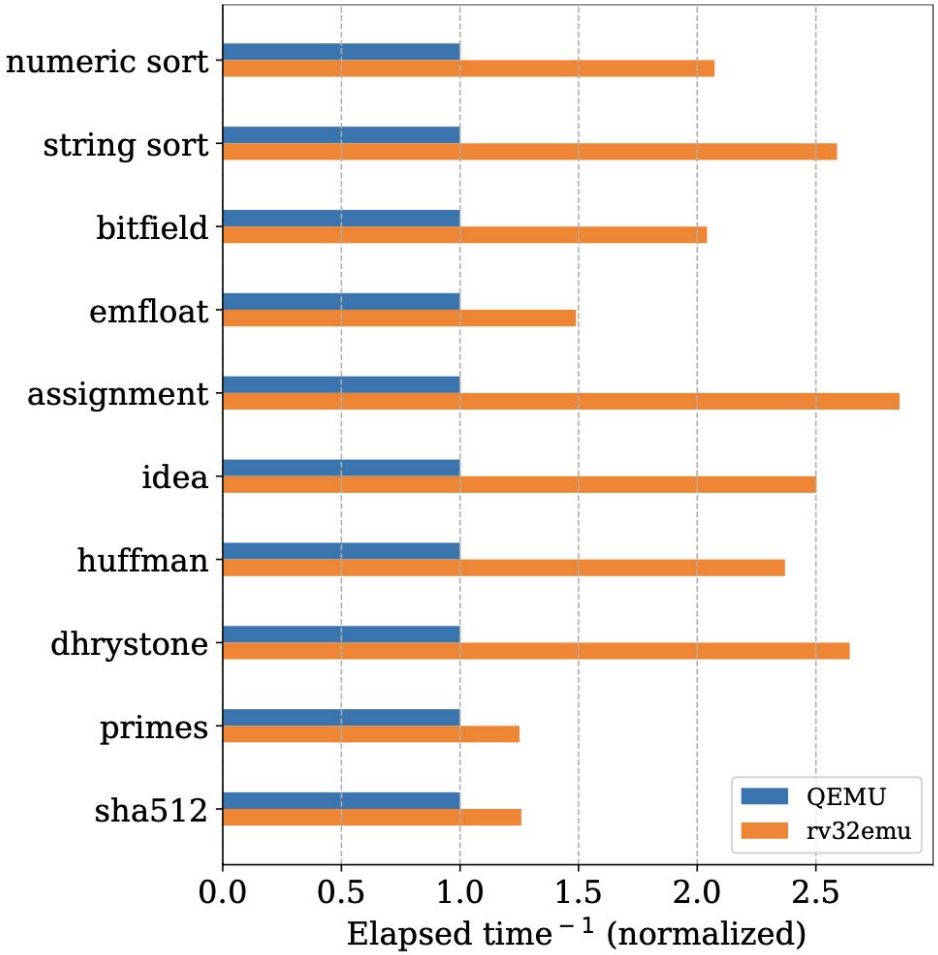
# Performance: Warmup-time Analysis

- In **assignment** benchmark,

| JIT Level | $\Delta_{T1C}$ | $\Delta_{T2C}$ | Elapsed Time |
|---|---|---|---|
| T1C-only | 1.62 ms | - | 62.9 s (+72.8%) |
| T2C-only | - | 1.23 s | 35.4 s (–2.75%) |
| Tiered | 1.14 ms | 0.59 s | 36.4 s (+0.00%) |

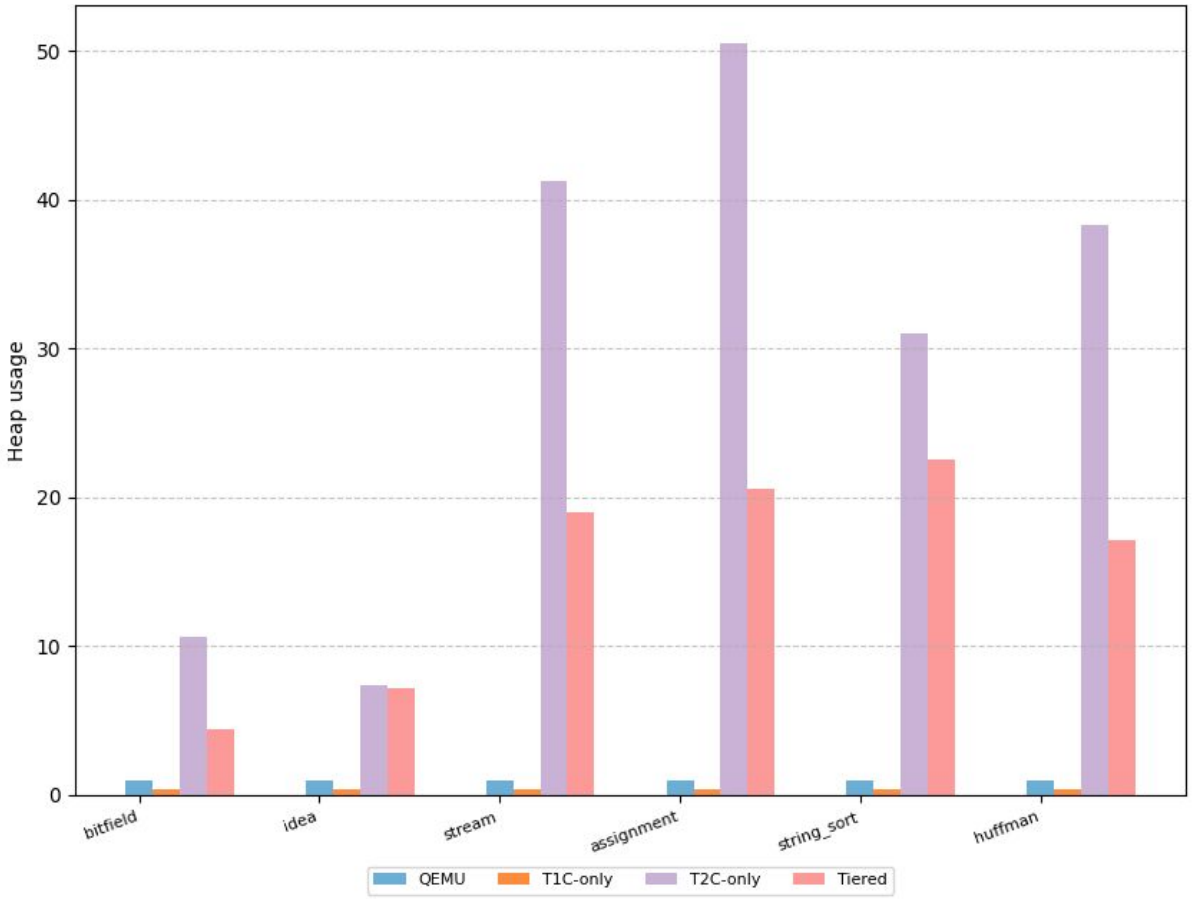
# Performance: QEMU vs rv32emu

- The performance of rv32emu with tiered JIT compilation outperforms **QEMU** among all benchmarks.



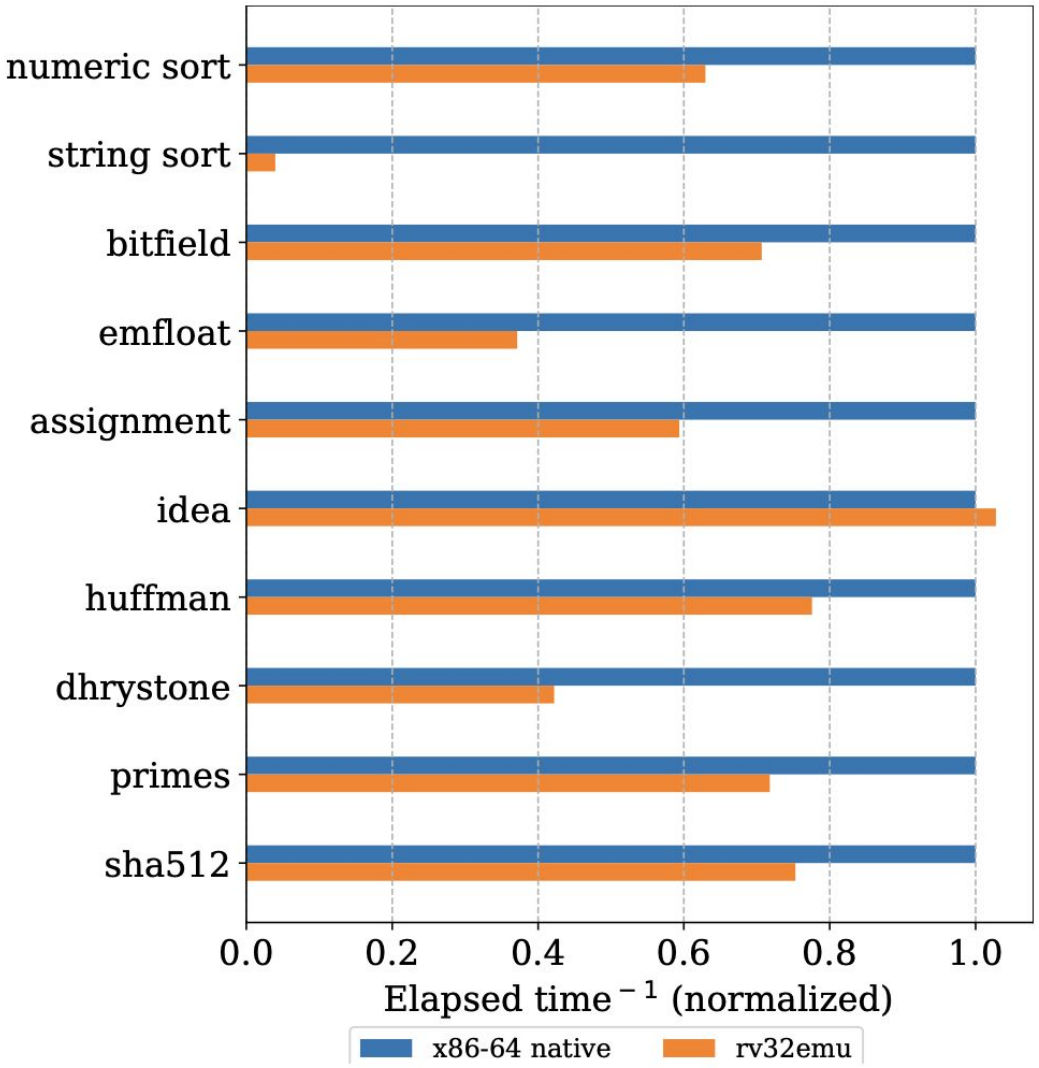
# Memory Consumption: QEMU vs different level of JIT

- A significant increase associated with the integration of LLVM for T2C, correlating with the frequency of T2C invocations.



# Performance: x86-64 native vs rv32emu

- sting sort:
  - x86 **REP** instruction on string copying.
- IDEA encryption:
  - Benefit from profiler and LLVM (O3)



# Discussion: Tier-1 JIT compilation

- T1C lacks of proper profiling, affecting T2C's activation, which depends on T1C invocation times. In scenarios like **loops** with numerous iterations, this design can block T2C activation, reducing its effectiveness.
- There are more powerful instructions for the host architecture, but T1C cannot detect the pattern for selecting these instructions.
- Only linear-scan register allocation and indirect jump improvement are applied.

# Discussion: Tier-2 JIT compilation

- Runtime compilation overhead of LLVM JIT compilation is relatively large, the improvements from the LLVM JIT compiler become substantial only when hotspot execution paths are executed numerous times.
- Based on our experimental results, LLVM has proven to be resource **inefficient**
  - not be ideal for low-latency DBT
- This observation suggests potential shortcomings for choosing LLVM as JIT compiler, we still need more researches to search alternatives.

# Let's run user mode emulation

- Simply open https://sysprog21.github.io/rv32emu-demo/user/
- You can run precompiled RISC-V programs directly in your browser, including Doom and Quake.

# System Emulation

# System Emulation

- Key components:
  - SV32 MMU
  - 8250 UART
  - PLIC
  - Interrupt
    - Software
    - Timer
    - External
  - Trap/Interrupt handling
  - Supervisor Binary Interface (SBI)
  - privileged instruction
    - fence
    - sfencevma
    - wfi
- boot Linux v6.1.y smoothly.

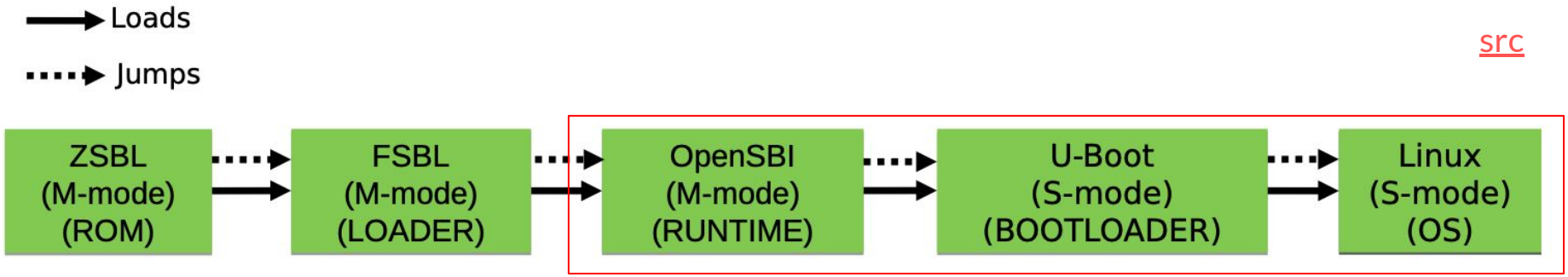
```
[    0.000000] Linux version 6.1.119 (mirv32@node11) (riscv32-buildroot-linux-gnu-gcc.br_real
[    0.000000] Machine model: rv32emu
[    0.000000] earlycon: ns16550 at MMIO 0xf4000000 (options '')
[    0.000000] printk: bootconsole [ns16550] enabled
[    0.000000] Zone ranges:
[    0.000000]   Normal   [mem 0x0000000000000000-0x00000001ffffffff]
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x0000000000000000-0x00000001ffffffff]
[    0.000000] Initmem setup node 0 [mem 0x0000000000000000-0x00000001ffffffff]
[    0.000000] SBI specification v0.3 detected
[    0.000000] SBI implementation ID=0x999 Version=0x1
[    0.000000] SBI TIME extension detected
[    0.000000] SBI SRST extension detected
[    0.000000] riscv: base ISA extensions aim
[    0.000000] riscv: ELF capabilities aim
```

```
[    1.652274] Run /init as init process
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Starting network: OK

Welcome to Buildroot
buildroot login: root
# 
```

# RISC-V/Linux Boot Sequence

- Privilege mode (the lowest the higher the privilege):
  - User
  - Supervisor
  - Machine
- BootLoader loads three key components and put them into desired memory address:
  - Kernel Image
  - Root File System
  - Device Tree Blob




src

```
build/rv32emu -k build/Image -i build/rootfs.cpio -b build/minimal.dtb
```

# Trap Handling

- The trap entry PC is decided by the stvec and scause CSR depending on the mode.
- The scause and stval CSR should be set according to the trap / interrupt.
- The sepc CSR stores the stop point PC, allowing the execution to resume by returning to the stored PC via sret instruction.
- The current privilege mode and interrupt status should also be stored, as mode changes are triggered by trapping.
- What about the regular register?

```
const uint32_t sstatus_sie =
    (rv->csr_sstatus & SSTATUS_SIE) >> SSTATUS_SIE_SHIFT;
rv->csr_sstatus |= (sstatus_sie << SSTATUS_SPIE_SHIFT);
rv->csr_sstatus &= ~(SSTATUS_SIE);
rv->csr_sstatus |= (rv->priv_mode << SSTATUS_SPP_SHIFT);
rv->priv_mode = RV_PRIV_S_MODE;
base = rv->csr_stvec & ~0x3;
mode = rv->csr_stvec & 0x3;
cause = rv->csr_scause;
rv->csr_sepc = rv->PC;
```

```
switch (mode) {
/* DIRECT: All traps set PC to base */
case 0:
    rv->PC = base;
    break;
/* VECTORED: Asynchronous traps set PC to base + 4 * code */
case 1:
    /* MSB of code is used to indicate whether the trap is interrupt
     * or exception, so it is not considered as the 'real' code */
    rv->PC = base + 4 * (cause & MASK(31));
    break;
}
IIF(RV32_HAS(SYSTEM))(if (rv->is_trapped) __trap_handler(rv);, )
```

# Trap Handling(cont)

- The trap or interrupt handler is not translated into a basic block because it isn't always needed. e.g.,, the trap path is followed only in case of a page fault, otherwise, the trap block becomes invalid. To address this, designed it as a step-by-step path within a while loop until an ***sret*** instruction is reached.
- The ***sret*** instruction is the final instruction after handling the trap or interrupt, used to return to the stop point. It restores all CSR values before trapping.

```c
/* SRET: return from traps in S-mode */
#if RV32_HAS(SYSTEM)
RVOP(
    sret,
    {
        rv->is_trapped = false;
        rv->priv_mode = (rv->csr_sstatus & SSTATUS_SPP) >> SSTATUS_SPP_SHIFT;
        rv->csr_sstatus &= ~(SSTATUS_SPP);

        const uint32_t sstatus_spie =
            (rv->csr_sstatus & SSTATUS_SPIE) >> SSTATUS_SPIE_SHIFT;
        rv->csr_sstatus |= (sstatus_spie << SSTATUS_SIE_SHIFT);
        rv->csr_sstatus |= SSTATUS_SPIE;

        rv->PC = rv->csr_sepc;

        return true;
    },
```

```c
#if RV32_HAS(SYSTEM)
static void __trap_handler(riscv_t *rv)
{
    rv_insn_t *ir = mpool_calloc(rv->block_ir_mp);
    assert(ir);

    /* set to false by sret implementation */
    while (rv->is_trapped && !rv_has_halted(rv)) {
        uint32_t insn = rv->io.mem_ifetch(rv, rv->PC);
        assert(insn);

        rv_decode(ir, insn);
        reloc_enable_mmu_jalr_addr = rv->PC;

        ir->impl = dispatch_table[ir->opcode];
        rv->compressed = is_compressed(insn);
        ir->impl(rv, ir, rv->csr_cycle, rv->PC);
    }

    prev = NULL;
}
#endif /* RV32_HAS(SYSTEM) */
```

# 8250 UART / RISC-V Platform Level Interrupt Controller

- 8250 UART is emulated for the guestOS console (tty).
  - The Linux kernel expects a console for serial communication, so emulating the 8250 UART gives the guest OS illusion such that a device exists.
  - Leveraging hostOS standard-{in, out} for such emulation.
- PLIC
  - The interrupt gateways convert global interrupt signals into a standard format and manage the flow of interrupt requests to the PLIC core.

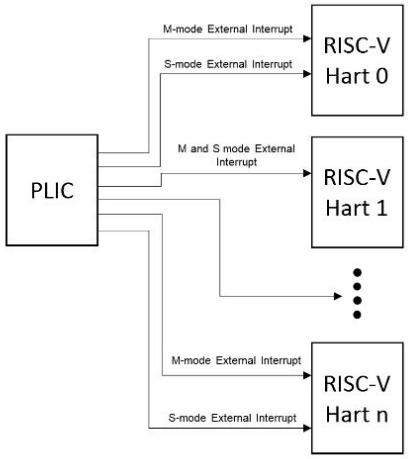



*Figure 1. RISC-V PLIC Interrupt Architecture Block Diagram*

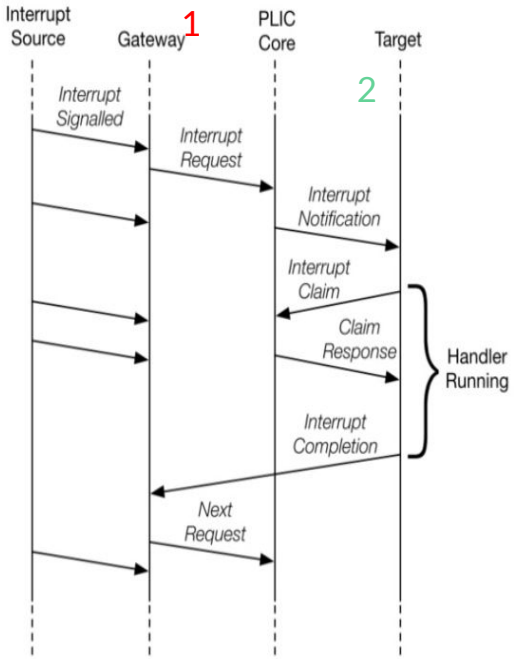



*Figure 2. PLIC Interrupt Flow*

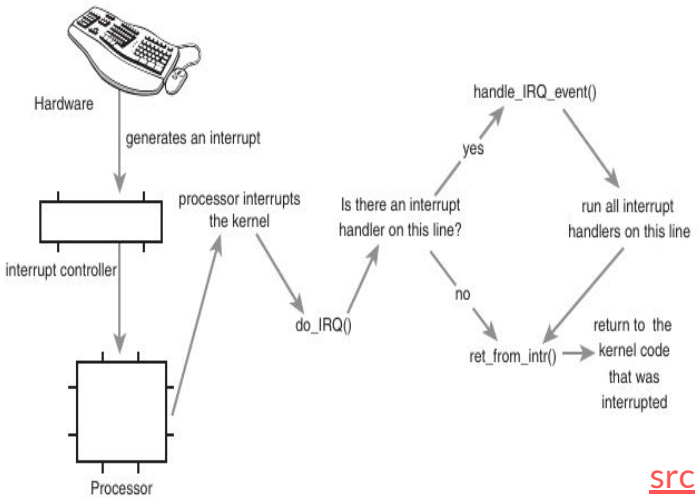
```c
void emu_update_uart_interrupts(riscv_t *rv)
{
    vm_attr_t *attr = PRIV(rv);
    u8250_update_interrupts(attr->uart);
    if (attr->uart->pending_intrs)
        attr->plic->active |= IRQ_UART_BIT;
    else
        attr->plic->active &= ~IRQ_UART_BIT;
    plic_update_interrupts(attr->plic);
}
```

```c
void plic_update_interrupts(plic_t *plic)
{
    riscv_t *rv = (riscv_t *) plic->rv;

    /* Update pending interrupts */
    plic->ip |= plic->active & ~plic->masked;
    plic->masked |= plic->active;
    /* Send interrupt to target */
    if (plic->ip & plic->ie)
        rv->csr_sip |= SIP_SEIP;
    else
        rv->csr_sip &= ~SIP_SEIP;
}
```

# Interrupts

- Make the CPU jump to a piece of code and then return.
- Combine the SIP and SIE CSR can determine the status of the interrupt.
- Check the interrupts after every block emulation.
- Three type of interrupts:
  - Software
  - Timer
  - External




src

```c
if (rv_has_plic_trap(rv)) {
    uint32_t intr_applicable = rv->csr_sip & rv->csr_sie;
    uint8_t intr_idx = ilog2(intr_applicable);
    switch (intr_idx) {
    case (SUPERVISOR_SW_INTR & 0xf):
        SET_CAUSE_AND_TVAL_THEN_TRAP(rv, SUPERVISOR_SW_INTR, 0);
        break;
    case (SUPERVISOR_TIMER_INTR & 0xf):
        SET_CAUSE_AND_TVAL_THEN_TRAP(rv, SUPERVISOR_TIMER_INTR, 0);
        break;
    case (SUPERVISOR_EXTERNAL_INTR & 0xf):
        SET_CAUSE_AND_TVAL_THEN_TRAP(rv, SUPERVISOR_EXTERNAL_INTR, 0);
        break;
    default:
        break;
    }
}
```

# Supervisor Virtual Memory (SV32 MMU)

- The MMU scheme for 32-bit RISC-V CPU.
- Default using 4K page size, but also support 4MB superpage.
- The first stage translation  in two-stage translation of guestOS since we are performing a system emulation.
- Three types of page fault must be checked and handled (if needed):
  - Instruction fetch page fault.
  - Load page fault.
  - Store page fault.

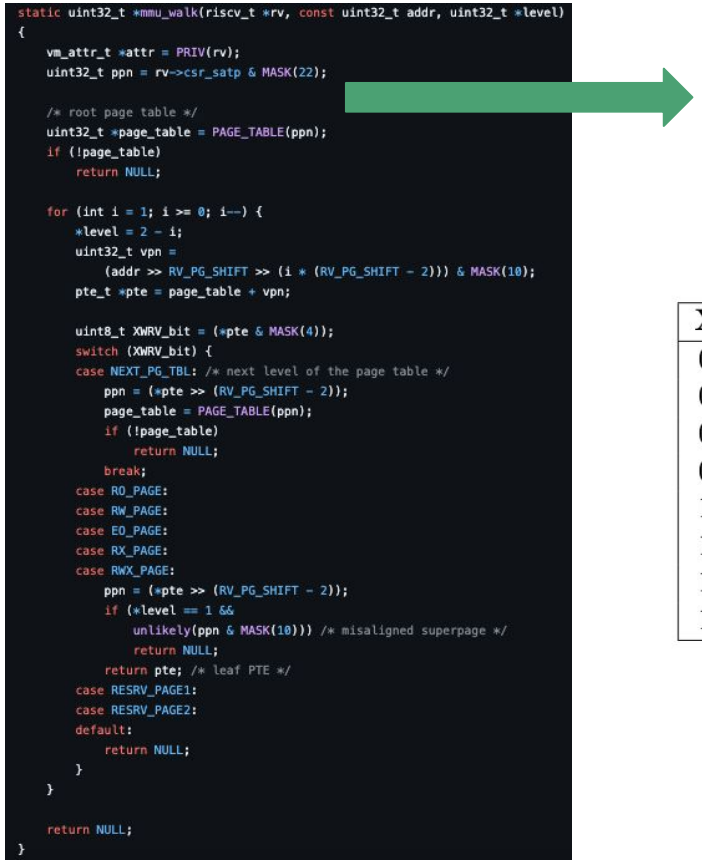
| | |
|---|---|
| 12 | Instruction page fault |
| 13 | Load page fault |
| 14 | *Reserved* |
| 15 | Store/AMO page fault |

- The page faults typically occur when:
  - the PTE is not found.
  - the PTE is invalid.
  - the permission is not granted.
- The scause CSR holds the page fault code, while the stval CSR contains the fault value, which represents the fault address.
- Trap to the kernel page fault handler to handle the fault and resume execution from the stopping point (PC).

```
if (!(pte && (*pte & access_bits))) {
    SET_CAUSE_AND_TVAL_THEN_TRAP(rv, scause, stval);
    return false;
}
```

```
/* PTE not found, map it in handler */
if (!pte) {
    SET_CAUSE_AND_TVAL_THEN_TRAP(rv, scause, stval);
    return false;
}
```

# Supervisor Virtual Memory (SV32 MMU)



```c
static uint32_t *mmu_walk(riscv_t *rv, const uint32_t addr, uint32_t *level)
{
    vm_attr_t *attr = PRIV(rv);
    uint32_t ppn = rv->csr_satp & MASK(22);

    /* root page table */
    uint32_t *page_table = PAGE_TABLE(ppn);
    if (!page_table)
        return NULL;

    for (int i = 1; i >= 0; i--) {
        *level = 2 - i;
        uint32_t vpn =
            (addr >> RV_PG_SHIFT >> (i * (RV_PG_SHIFT - 2))) & MASK(10);
        pte_t *pte = page_table + vpn;

        uint8_t XWRV_bit = (*pte & MASK(4));
        switch (XWRV_bit) {
        case NEXT_PG_TBL: /* next level of the page table */
            ppn = (*pte >> (RV_PG_SHIFT - 2));
            page_table = PAGE_TABLE(ppn);
            if (!page_table)
                return NULL;
            break;
        case RO_PAGE:
        case RW_PAGE:
        case EO_PAGE:
        case RX_PAGE:
        case RWX_PAGE:
            ppn = (*pte >> (RV_PG_SHIFT - 2));
            if (*level == 1 &&
                unlikely(ppn & MASK(10))) /* misaligned superpage */
                return NULL;
            return pte; /* leaf PTE */
        case RESRV_PAGE1:
        case RESRV_PAGE2:
        default:
            return NULL;
        }
    }

    return NULL;
}
```

| X | W | R | Meaning |
|---|---|---|---|
| 0 | 0 | 0 | Pointer to next level of page table. |
| 0 | 0 | 1 | Read-only page. |
| 0 | 1 | 0 | *Reserved for future use.* |
| 0 | 1 | 1 | Read-write page. |
| 1 | 0 | 0 | Execute-only page. |
| 1 | 0 | 1 | Read-execute page. |
| 1 | 1 | 0 | *Reserved for future use.* |
| 1 | 1 | 1 | Read-write-execute page. |

Table 4.5: Encoding of PTE R/W/X fields.

# Supervisor Binary Interface (SBI)

- SBI enables S-mode software (kernel) to communicate with low-level M-mode software (firmware).

- Three main type of SBI are necessary:
  - Base (called by *sbi_base_ecall*)
    - All functions in the base extension must be supported by all SBI implementations.
    - For probing SBI version and vendor information purpose.
  - Timer (called by *sbi_set_timer*)
    - Used to set the next timer event for timer interrupt.
  - Reset (called by *sbi_srst_reset*)
    - Used to poweroff gracefully.

```
_(sbi_base,          0x10)
_(sbi_timer,         0x54494D45)
_(sbi_rst,           0x53525354)
```

- To support SMP emulation, inter-process interrupt (IPI), remote fence, and hart state management (HSM) extension should be implemented.

# Supervisor Binary Interface (SBI)

```c
long __sbi_base_ecall(int fid)
{
        struct sbiret ret;

        ret = sbi_ecall(SBI_EXT_BASE, fid, 0, 0, 0, 0, 0, 0);
        if (!ret.error)
                return ret.value;
        else
                return sbi_err_map_linux_errno(ret.error);
}

/**
 * sbi_set_timer() - Program the timer for next timer event.
 * @stime_value: The value after which next timer event should fire.
 *
 * Return: None.
 */
void sbi_set_timer(uint64_t stime_value)
{
       __sbi_set_timer(stime_value);
}
```

```c
static void sbi_srst_reset(unsigned long type, unsigned long reason)
{
        sbi_ecall(SBI_EXT_SRST, SBI_EXT_SRST_RESET, type, reason,
                  0, 0, 0, 0);
        pr_warn("%s: type=0x%lx reason=0x%lx failed\n",
                __func__, type, reason);
}
```

# Supervisor Binary Interface (SBI)

- When a userspace reboot command was issued, the hart was incorrectly halted as if it were a shutdown.
- reboot-safe resource management - Add "check for reboot" comments throughout initialization to reuse already-allocated resources (memory, fd_map, PLIC, UART, vblk, block_map, etc) instead of re-allocating
  ⇒ Demo

# Privileged Instructions

- **`fence`**
  - Order device I/O and memory accesses as viewed by other RISC-V harts and external devices or coprocessors.
- **`sfencevma`**
  - Synchronize updates to in-memory memory-management data structures with current execution.
  - Flushing TLB cache.
- **`wfi`**
  - Wait for interrupt
  - The RISC-V hart might execute this instruction and get into sleep state.
- These three instructions are critical in Linux kernel as you can objdump the Linux kernel Image to determine them especially in SMP part.

# WebAssembly support

# WebAssembly Translation

- WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications.
- After being ported to WebAssembly, rv32emu can run anywhere as long as a browser is available.
- Enhance the accessibility of the emulator.
  - No need to download
  - No need to compile
  - No package dependency problems (e.g., libsdl for video games)

```
#ifdef __EMSCRIPTEN__
        emscripten_set_main_loop_arg(rv_step, (void *) rv, 0, 1);
#else
        /* default main loop */
        for (; !rv_has_halted(rv);) /* run until the flag is done */
            rv_step(rv);            /* step instructions */
```

- rv32emu leverages Emscripten technology for porting.
- Porting steps:
  - Replace emulator's main while loop with ***emscripten_set_main_loop_arg*** API
  - Export the entry function which is global accessible (Wasm runtime calls it)
    - Normally, this function will be the main function in C language

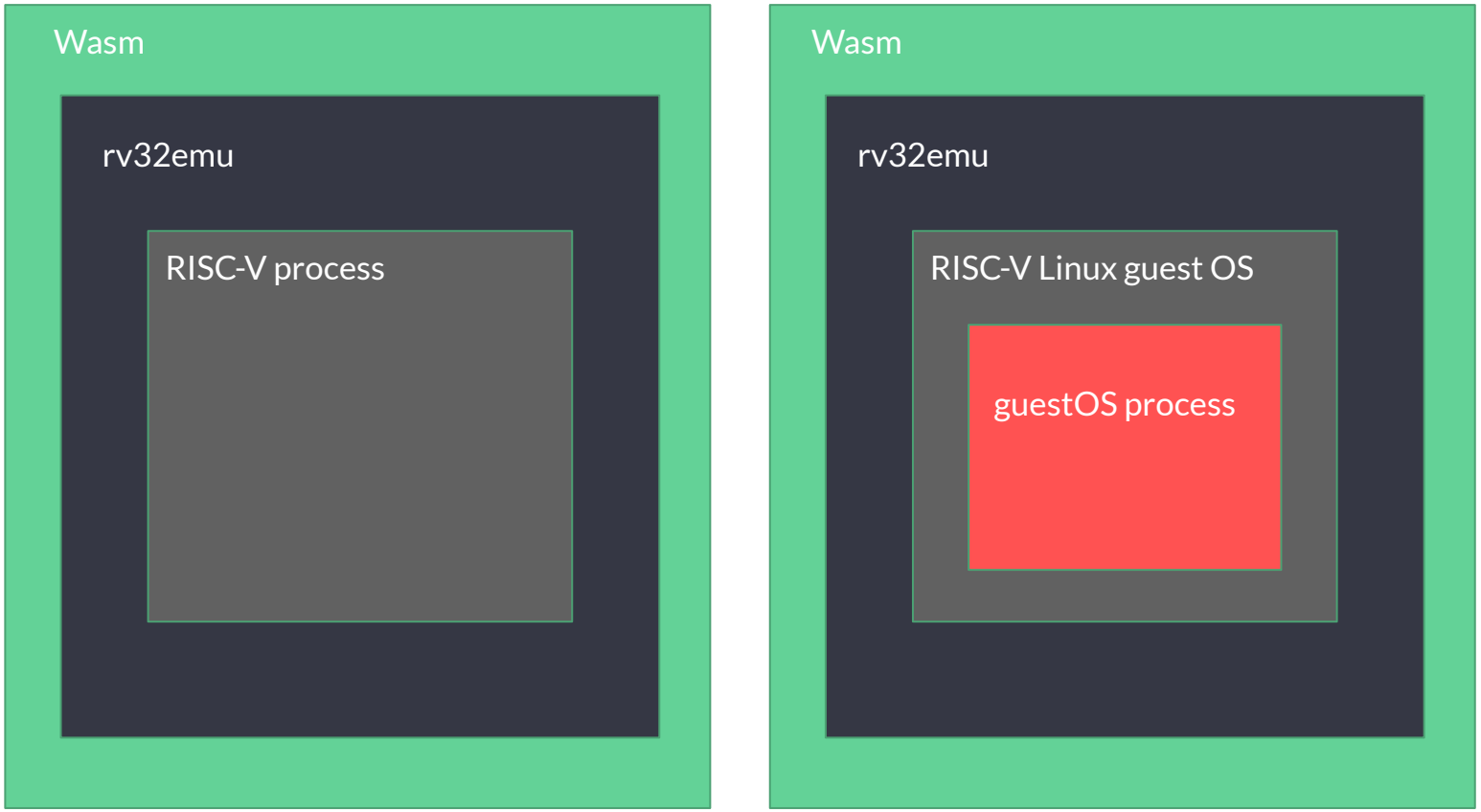
# WebAssembly Translation

- -sINITIAL_MEMORY: The RAM of the Wasm.
- -sALLOW_MEMORY_GROWTH: Allow dynamically expand the memory.
- -sEXPORTED_FUNCTIONS: The exported function from the application and callable in Wasm runtime.
- -sSTACK_SIZE: The stack size of the Wasm.
- -sPTHREAD_POOL_SIZE: The thread pool size in Wasm runtime. *navigator.hardwareConcurrency* is the maximum available web workers in the browser. Useful for multithreaded application.
- -embed-file: Specify a file (with path) to embed inside the generated Wasm.
- -pre-js: Specify a file whose contents are added before the emitted code and optimized together with it.
- -O3: Most aggressive optimization.
- -w: Suppress warning.
- Note: Emscripten generates a Wasm  module and a demo HTML page if the output file ends with ".html", otherwise, it only generates the Wasm module.

```
# More build flags
CFLAGS_emcc += -sINITIAL_MEMORY=2GB \
           -sALLOW_MEMORY_GROWTH \
           -s"EXPORTED_FUNCTIONS=$(EXPORTED_FUNCS)" \
           -sSTACK_SIZE=4MB \
           -sPTHREAD_POOL_SIZE=navigator.hardwareConcurrency \
           --embed-file build@/ \
           --embed-file build/riscv32@/riscv32 \
           --embed-file build/timidity@/etc/timidity \
           -DMEM_SIZE=0x40000000 \
           -DCYCLE_PER_STEP=2000000 \
           --pre-js $(WEB_JS_RESOURCES)/pre.js \
           -O3 \
           -w
```

# WebAssembly Translation

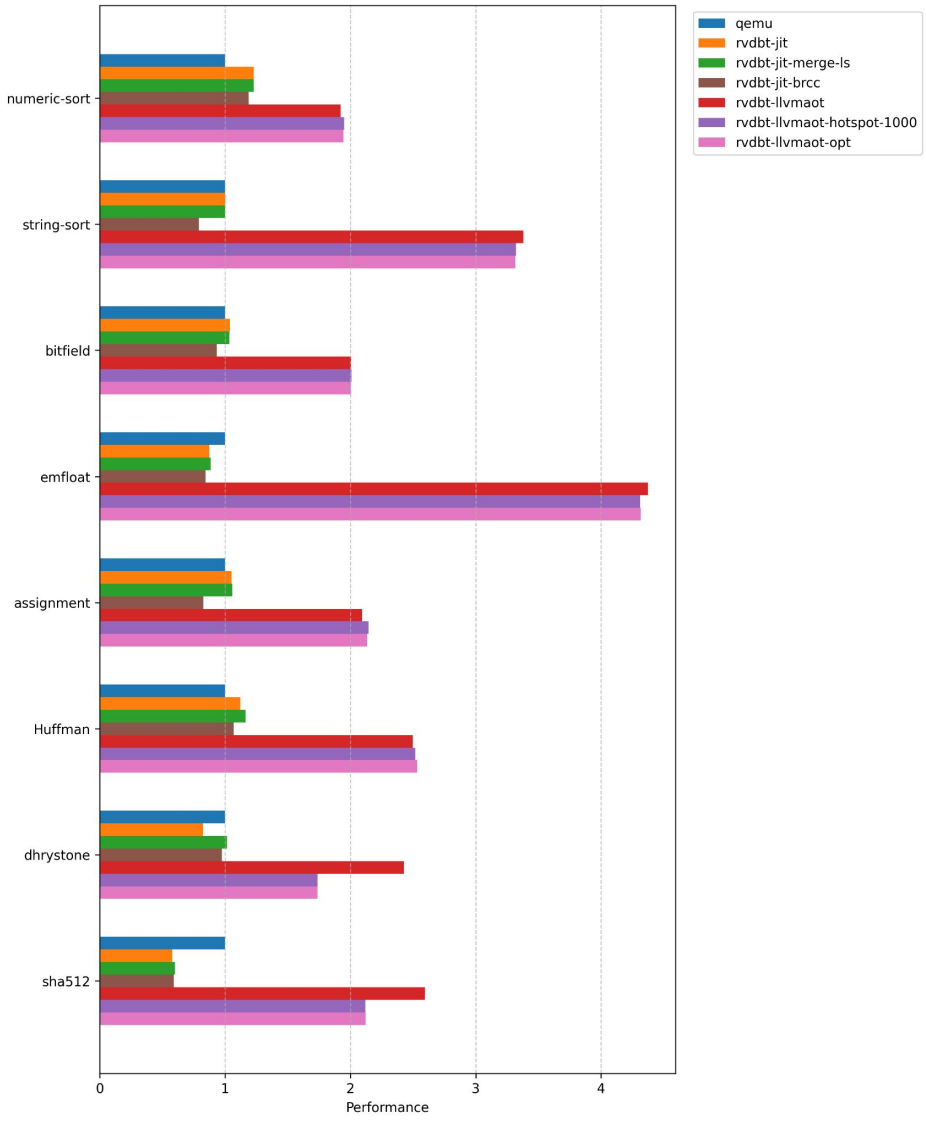
- Running rv32emu inside WebAssembly.
- Open https://sysprog21.github.io/rv32emu-demo/system/

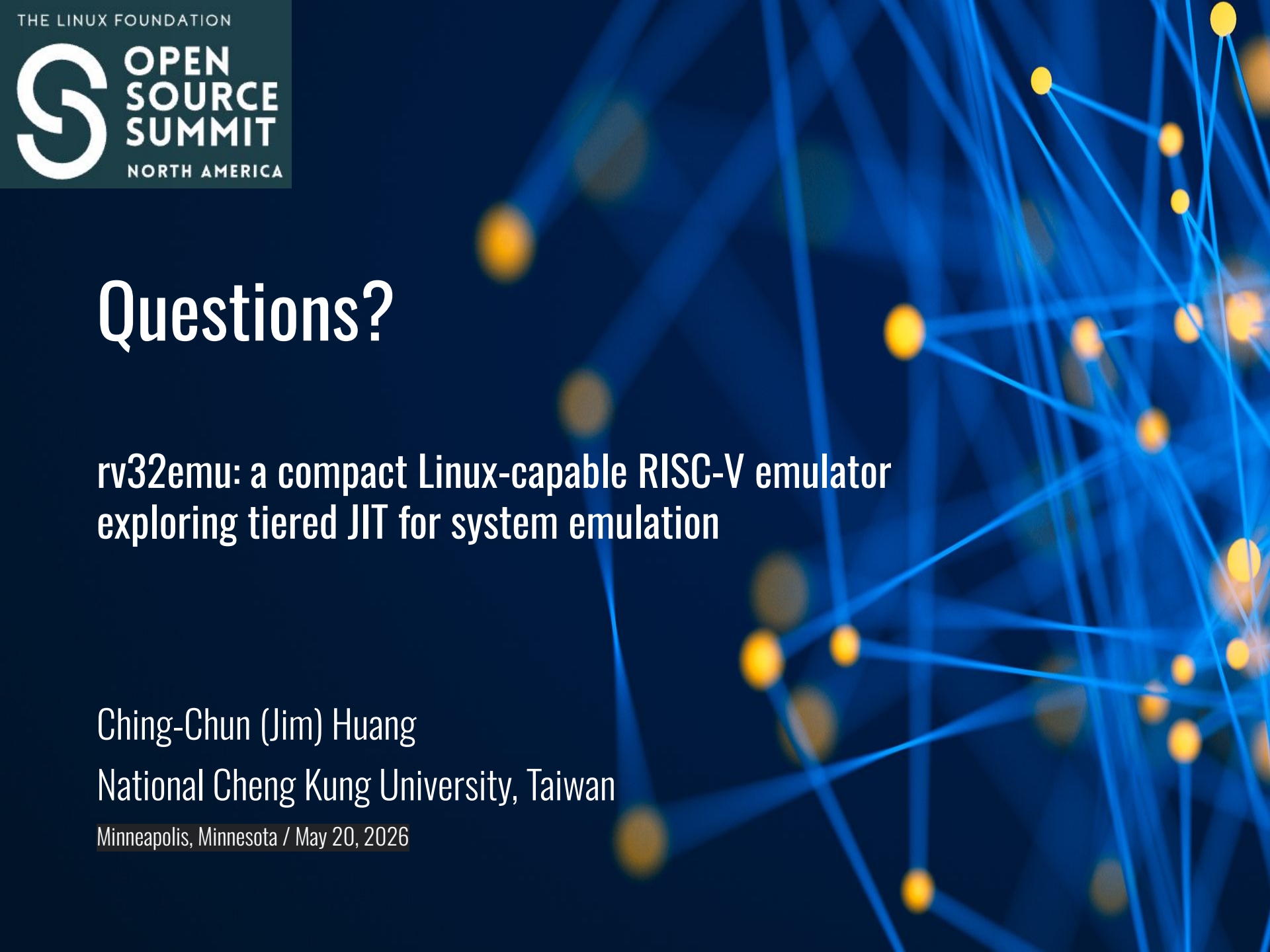


# Ongoing Improvements

- Full Vector extension
- Efficient SMP
- multicore scalability
- Faster Tier-1 JIT compiler
- Hybrid JIT/AOT compilation
- More VirtIO
- Refine APIs for integration
- 64-bit RISC-V

- Follow
  https://github.com/sysprog21/rv32emu






# Questions?

## rv32emu: a compact Linux-capable RISC-V emulator exploring tiered JIT for system emulation

Ching-Chun (Jim) Huang

National Cheng Kung University, Taiwan

Minneapolis, Minnesota / May 20, 2026