Biotite: A High-Performance Static Binary Translator using Source-Level Information

Changbin Chen, Shu Sugita, Yotaro Nada,
Hidetsugu Irie, Shuichi Sakai, Ryota Shioya

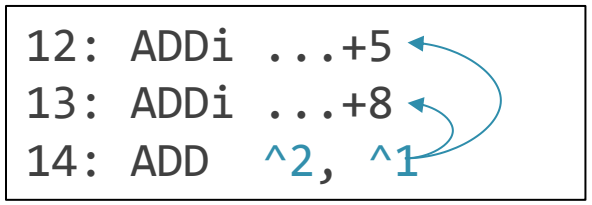
The University of Tokyo

Background: New ISA Research

Traditional RISC

```
ADDi x1 ← ...+5
ADDi x10← ...+8
ADD  x2 ← x1, x10
```

Distance-based ISA



- New Instruction Set Architectures (ISAs) continue to be an active area of research.
  - We are working on a new type of ISA called Distance-based ISA:
    - Operands are specified by their instruction distance instead of register numbers.
    - This approach eliminates false dependencies, removing the need for register renaming.
  - STRAIGHT [MICRO2018], Clockhands [MICRO2023], TURBULENCE [CAL2024].

- Challenge: Implementing a complete toolchain (compiler, linker, libraries) for various languages is extremely difficult.

Background: Challenges in Research ISAs

- Research ISAs typically provide only simple toolchains:
  - Minimal compiler backend
  - Custom libc with limited C language support
- Compiling programs written in various languages is challenging.

Examples of Effort Required to Build a Toolchain for New ISA

- Assembly code porting:
  - Manually porting ISA-specific assembly in libraries.
- Handling platform-specific preprocessor macros,
  - such as rewriting #ifdef RISCV for the new target ISA
- Language-specific backend support:
  - e.g., Clang generates complex LLVM IR for C++ exceptions.
- Providing full tool support:
  - e.g., linking libstdc++ requires many linker options.
- ISA-specific constraints include the following:
  - STRAIGHT and WebAssembly do not allow direct call stack operations.
  - This makes implementing setjmp/longjmp support challenging.

Effort Required to Build a Toolchain for New ISA

- In addition to these challenges, there are many other small but essential tasks.
- While each task may seem simple,
  implementing everything correctly and consistently is extremely difficult.

Biotite: Static Binary Translator for New ISAs

- Our approach: Binary Translation
  - By translating a statically linked binary, It avoids most of the efforts required for toolchain implementation.
  - In ISA research, we can use source code to achieve accurate translation and optimization.

- We propose Biotite, new static translator:
  - Correctly translates a wide range of programs.
    - SPEC CPU 2017 (C, C++, Fortran) and Haskell.
  - Performance in CoreMark:
    - 87.2% of native performance (16.8sec vs 14.7sec)
    - 280% faster than QEMU (16.8sec vs 64.0sec)

Outline

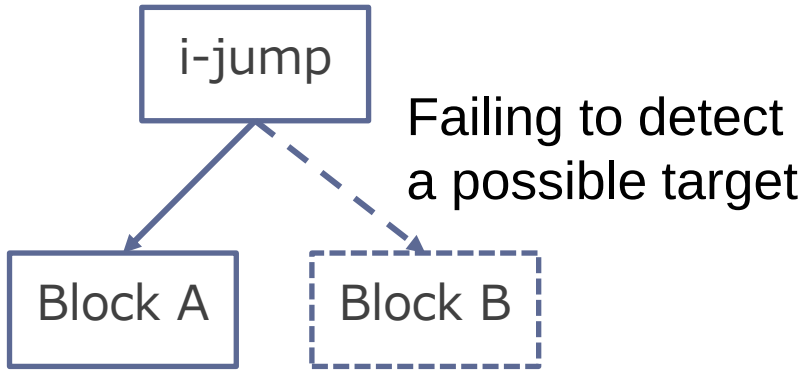
1. Binary Translation
    1. Static Binary Translation
    2. Dynamic Binary Translation
2. Biotite: A Static Binary Translator for Research ISAs
3. Evaluation

Static Binary Translator

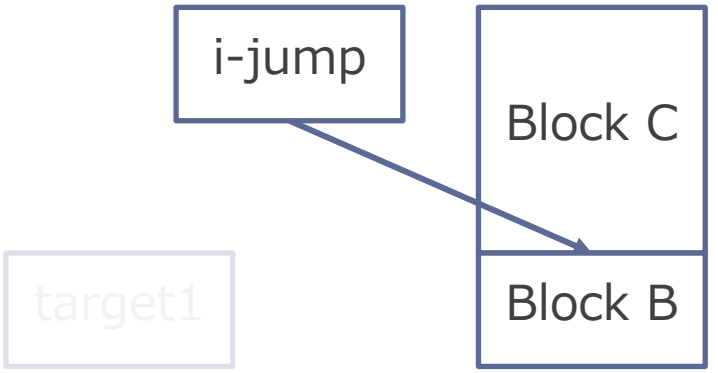
- Static translators translate the entire input program before it is executed.
- Merit:
  - No overhead from runtime translation.
- Challenge: Hard to handle indirect jumps.
  1. Detecting jump targets
  2. High-level intermediate representation (IR) specific problem

Handling Indirect Jumps

- Existing static translators typically reconstruct high-level control flow.
  - e.g., functions, switch-cases
- Challenge: Hard to handle indirect jumps.
  - It is difficult to detect all possible jump targets.
- If jump targets are not detected correctly, the execution may become incorrect.
  - Unexpected control flow transitions in the translated binary lead to incorrect results.



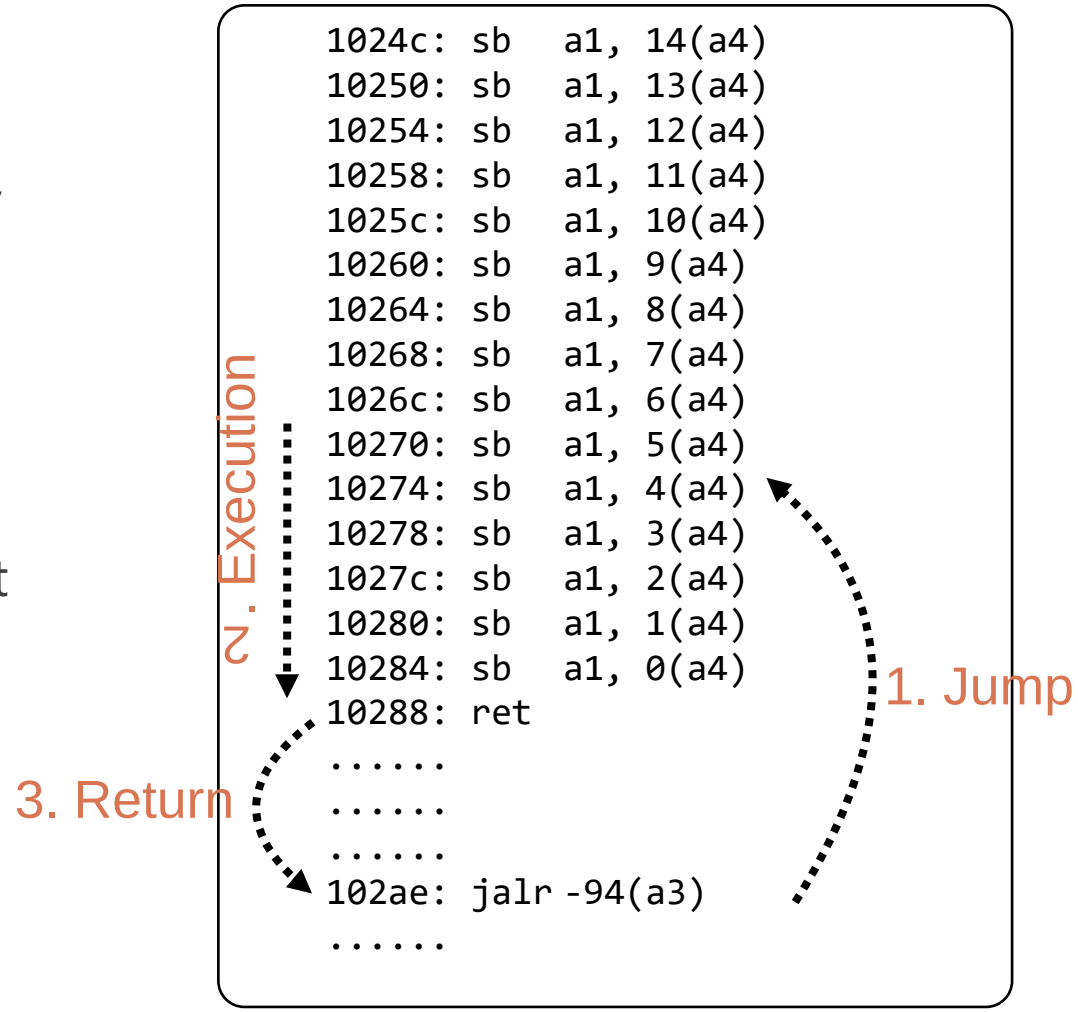
Failing to detect a possible target



C and B are optimized together without assuming a jump into B.

Detecting Indirect Jump Targets is Essential but Challenging

- Indirect jumps can jump to any address.
- One reason is that libraries often employ tricky assembly techniques.
  - e.g., **`memset`** in Newlib writes a variable number of bytes.
- Heuristics-based algorithms often fail to detect such targets



High level IR Specific Problem

- Static translators often use LLVM IR as translation output to leverage existing optimizers in LLVM.

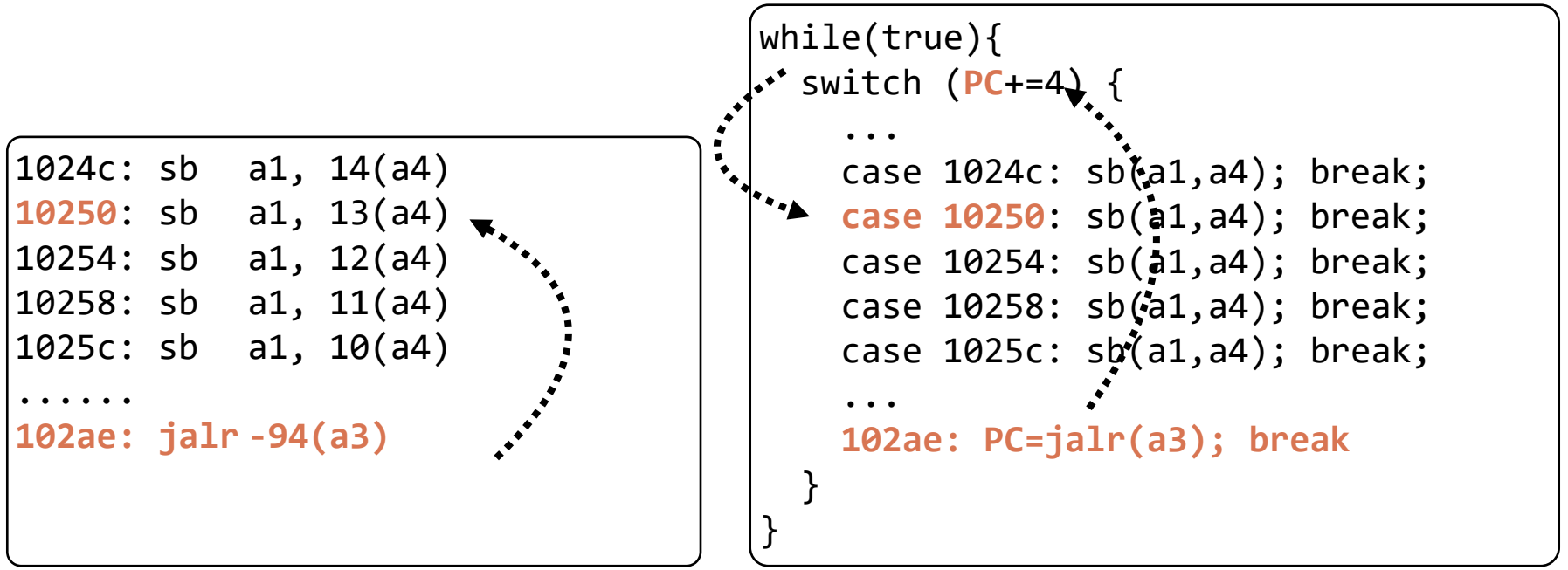
- Challenge:
  - LLVM IR supports only structured control flow (e.g., if, switch, call, return).
  - It cannot directly represent global jumps across functions.
    - Indirect jump can jump to any addresses.

```
function_a() {
  goto X;
}

function_b() {
  X: ...
}
```

Dispatching in High-Level IR

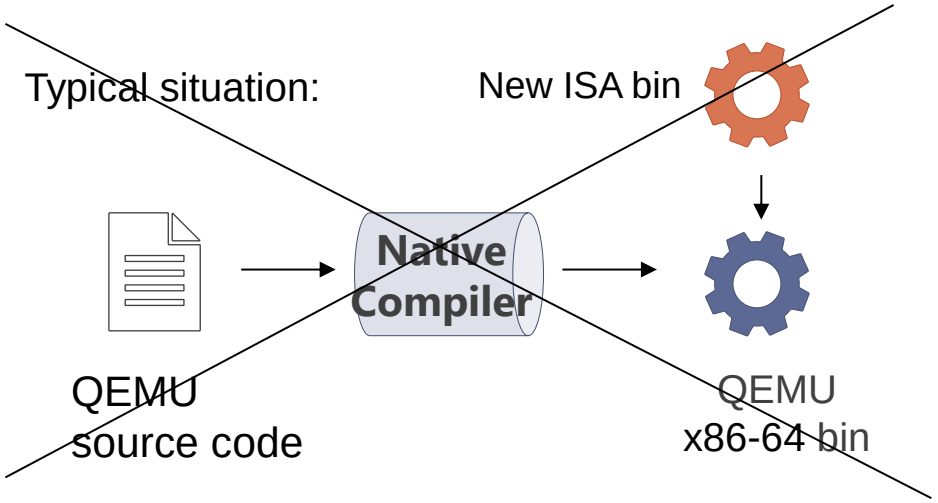
- A naive approach is to implement a giant switch-case structure.
  - Each instruction has its own address in the case, and each case contains the processing for each instruction.
  - This approach is also used in Emscripten's code generation for JavaScript targets.
    - JavaScript only supports structured control flow.

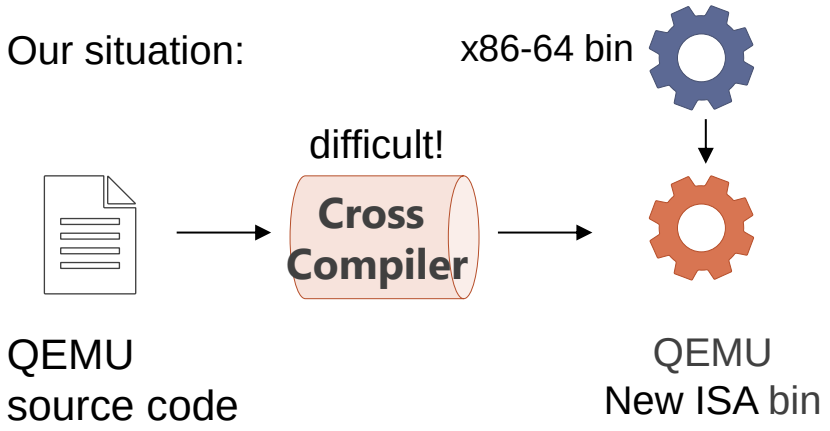
- Optimizing a giant switch-case structure is difficult because the control flow becomes highly complex.



Dynamic Binary Translation

- Instructions are translated dynamically at runtime.
- Merit:
  - Can handle indirect jumps correctly.
- Challenges:
  - High runtime overhead.
  - Difficult to implement for research ISAs without a full toolchain.





Outline

1. Binary Translation
    1. Static Binary Translation
    2. Dynamic Binary Translation
2. Biotite: A Static Binary Translator for Research ISAs
3. Evaluation

Biotite

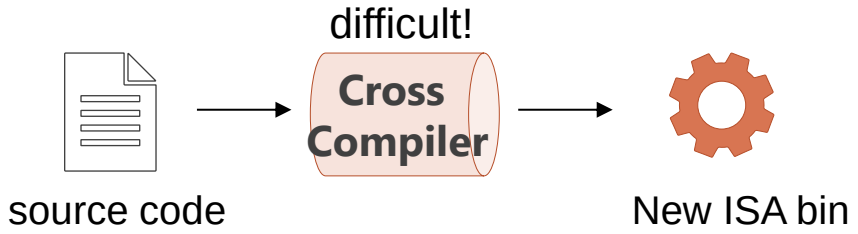
This photo is from Wikipedia
https://en.wikipedia.org/wiki/Biotite

- A novel static binary translator for research ISAs.
  - Translates RISC-V binaries to ISA-independent LLVM IR.
- It is named 'Biotite' after the layered mineral.

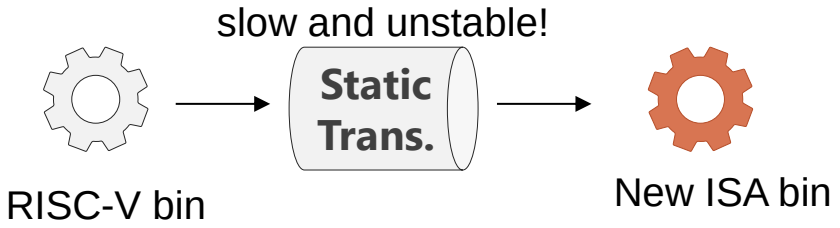
Biotite-Overview

- Our goal is to address the many challenges associated with toolchains.
  - e.g., assembly porting, backend features, linker features…
- Biotite utilizes source-level information for optimization.
  - Accurate and fast, yet simple

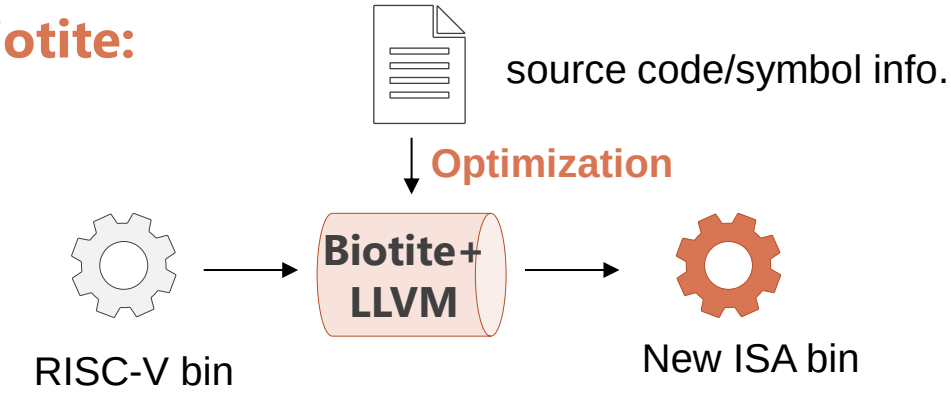
Ideal case:



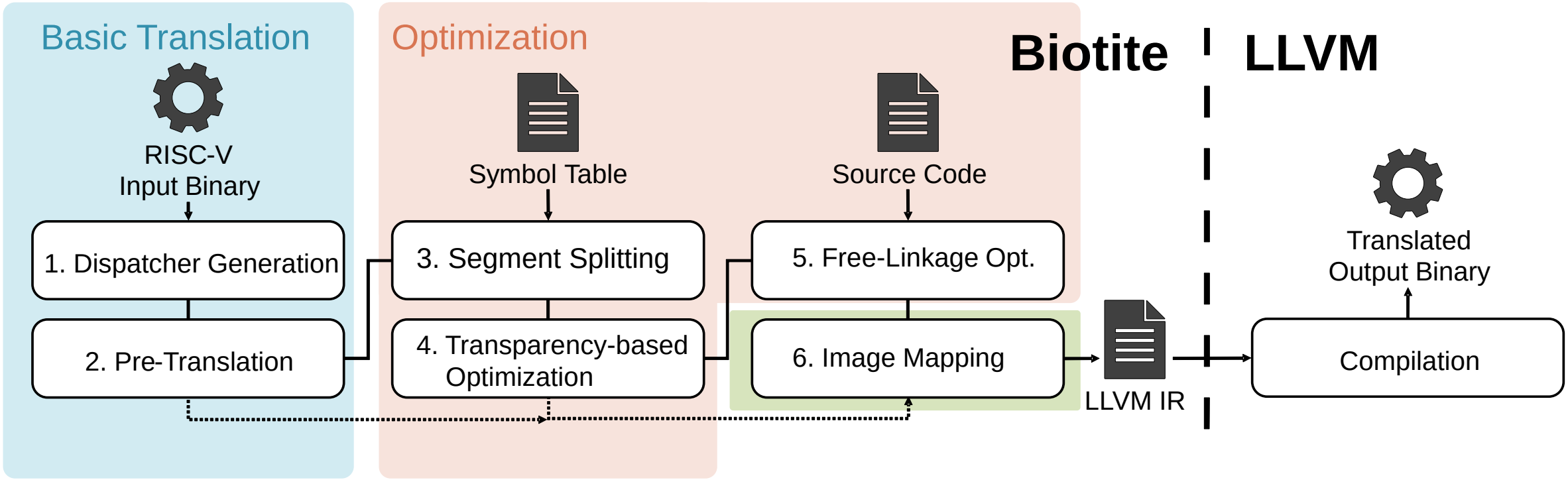
Static:



Our Biotite:

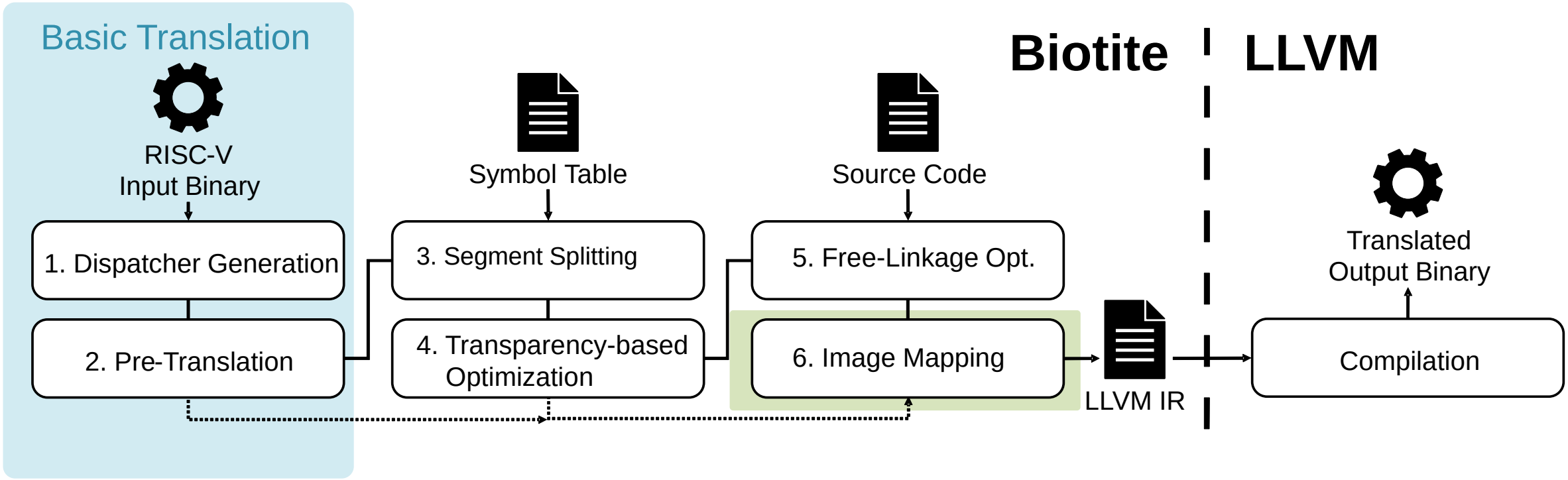


Biotite Translation Flow



- The Biotite translation process consists of the following steps:
  1. Basic Translation
  2. Optimization using source level information
  3. Memory Image Mapping

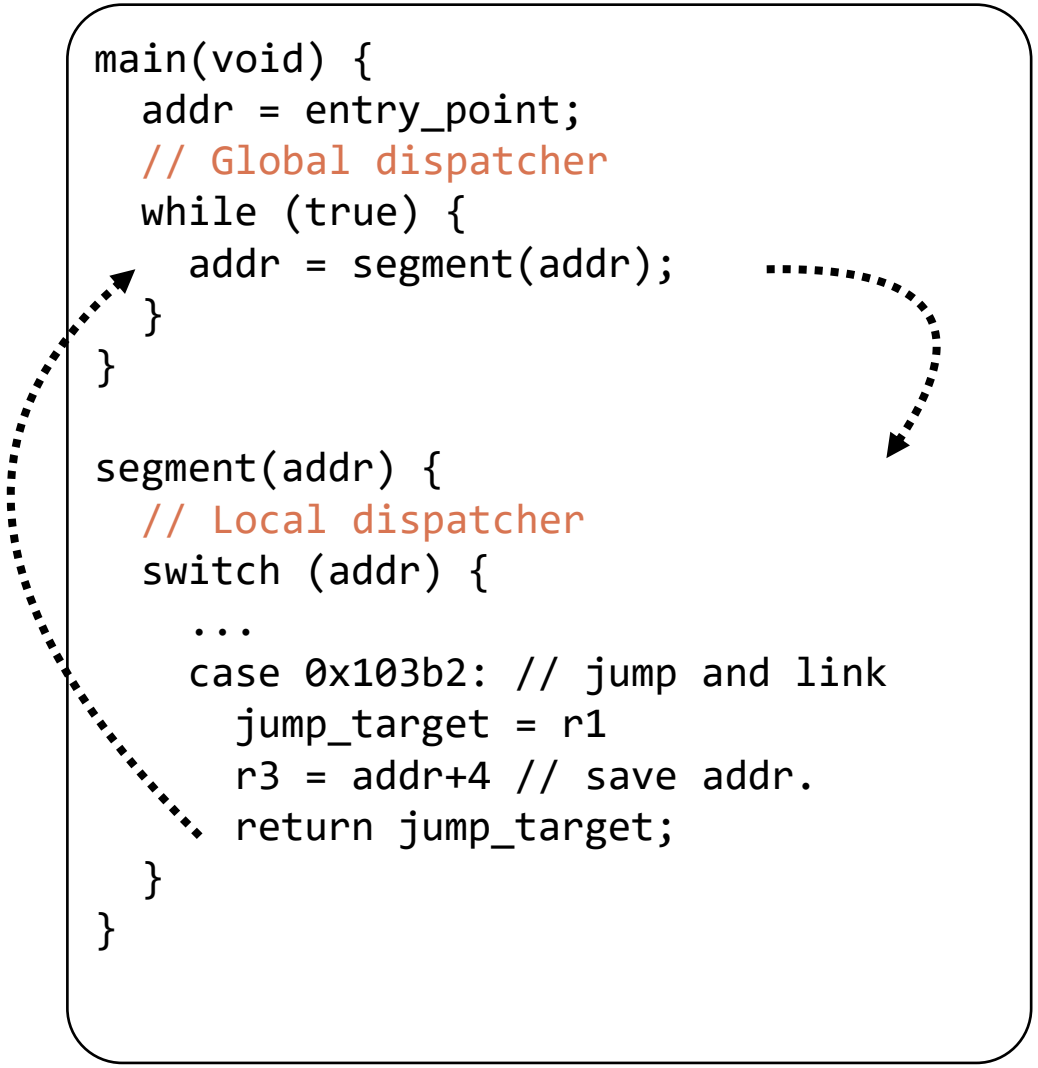
Biotite Translation Flow – Basic Translation



1. Basic translation process
    1. Dispatcher generation
    2. Pre-translation

Stage 1. Dispatcher Generation

- Biotite initially generates the following structure:
  - A global dispatcher inside main()
    - Calls segment functions in a loop, passing the next instruction address as an argument.
  - Segment function:
    - Contains all translated instructions.
    - Returns the address of the next instruction to execute.
  - Local dispatcher:
    - Uses a switch-case to jump to the correct instruction.
- Note that:
  - This example is written in C-like style for simplicity.
  - In practice, Biotite generates an internal data structure and passes it to the next stages.



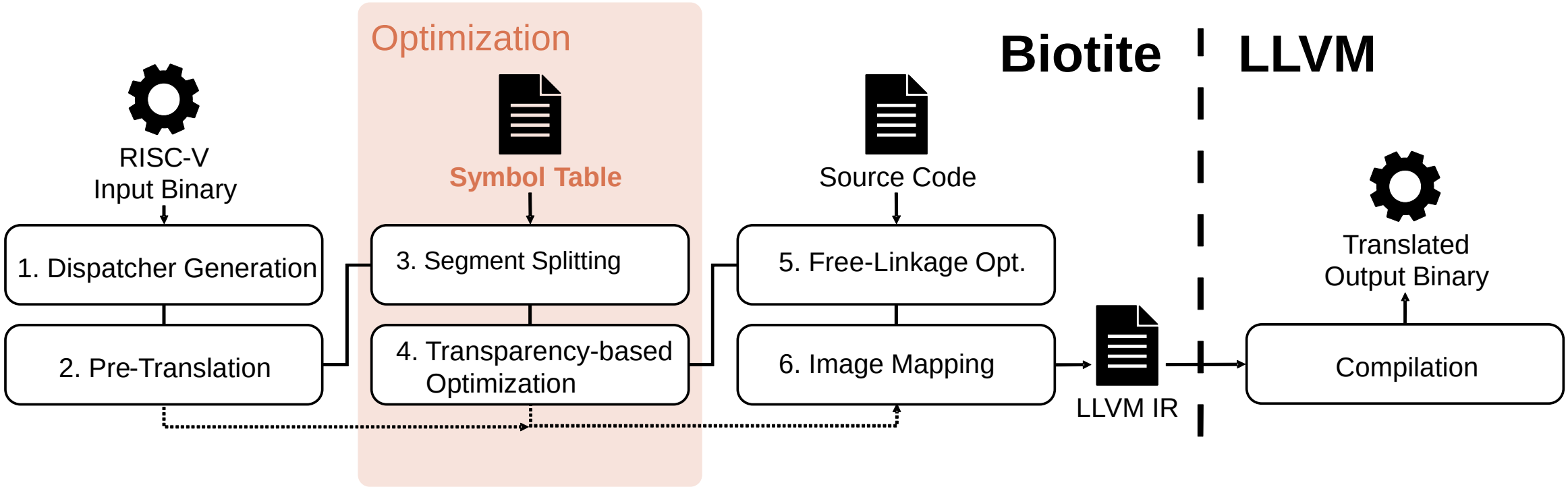
```
main(void) {
  addr = entry_point;
  // Global dispatcher
  while (true) {
    addr = segment(addr);
  }
}

segment(addr) {
  // Local dispatcher
  switch (addr) {
    ...
    case 0x103b2: // jump and link
      jump_target = r1
      r3 = addr+4 // save addr.
      return jump_target;
  }
}
```

Stage 2. Pre-Translation

- This stage generates code inside the local dispatcher:
  - It simulates RISC-V registers, and they are stored as global variables in LLVM IR.
    - Every register access involves a load/store operation.
  - Control instructions are translated into return statements.
    - Local dispatcher returns to the global dispatcher with the jump target address.

```
switch(addr){
// 0103b0: addi r10,r10,1
case 0x103b0:
    op0 = load(&r10)
    tmp = op0 + 1 // addi
    store(&r10, tmp)
    return 0x103b4;

// 0103b4: jr r10
case 0x103b4
    op0 = load(&r10)
    return op0 // return to the global dispatcher
}
```

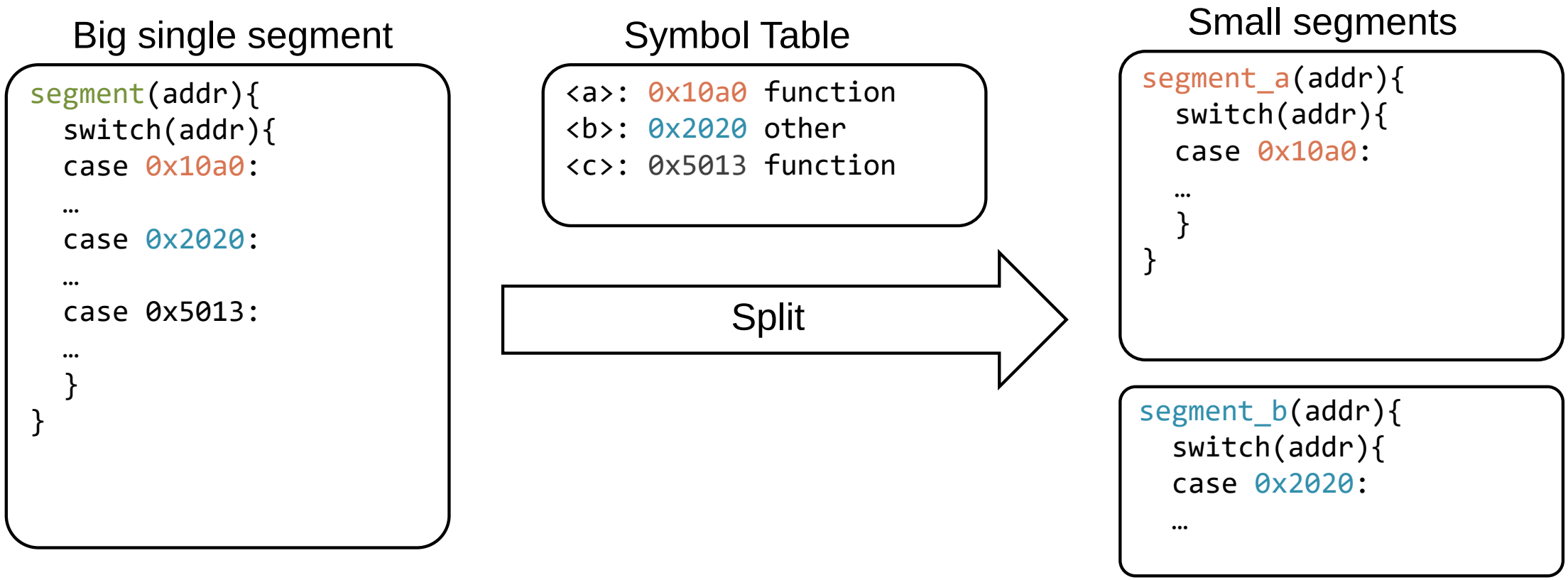
Biotite Translation Flow – Optimization



- It leverages a symbol table for optimization.
  1. Segment Splitting
  2. Transparency-based Optimization

3. Segment Splitting

- The big single segment is split into multiple segments using a symbol table as hints.
  - Labels in the symbol table define segment boundaries.
  - Segments do not necessarily align with function boundaries; however, aligning them improves performance.



3. Segment Splitting

- Global dispatcher retrieves and calls a next segment.
- In a small segment, CFG is easily determined.
  - Further optimization can be applied.

```
// Global dispatcher
while (true) {
  addr = segment(addr);
}
...
segment(addr){
  switch(addr){ // big local dispatcher
  case 0x10a0:  …
  case 0x10a4:
  …
  case 0x2020:
  …
  }
}
```

```
// Global dispatcher
while (true) {
  s_ptr = get_segement(addr);
  addr = (*s_ptr)(addr);
}
...
segment_a(addr) {
  switch(addr) // local dispatcher
  case 0x10a0 :
    r10=load(&r10);…
  case 0x10a4 :
    return 0x2020; // j 0x2020
  }
}

segment_b(addr) { // local dispatcher
  switch(addr)
  case 0x2020:
    …
  }
}
```

4. Transparency-based optimization

- Split segments are classified based on the concept of “Transparency”:
  - Transparent segments: contain no indirect jumps.
  - Opaque segments: include indirect jumps.
- Biotite optimizes segments depending on its transparency.
- （This is referred to as "jump localization" in the paper.)

Transparent Segment

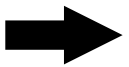
- It includes direct jumps/branches only.
  - That is, all returns to the global dispatcher have an immediate value.

```
// global dispatcher
while (true) {
  func = get_containing_func(addr);
  addr = (*func)(addr);
}


segment_1(addr) {
  // Local dispatcher
  case 0x103b4 // j 0x30044
    return 0x30044 // return to global
  ...
  case 0x30044 // bne 0x103b4
    if (...=!0)
      return 0x103b4 // return to global
  }
}
```

Jump Conversion for Transparent Segments

- Each return to the global dispatcher can be replaced with segment-local **goto**.
  - It eliminates the local dispatcher.
- Original control flow in each function can be well reconstructed.

```
segment_1(addr) {
  // Local dispatcher
  case 0x103b4 // j 0x30044
    return 0x30044 // return to the global
  ...
  case 0x30044 // bne 0x103b4
    if (...=!0)
      return 0x103b4 // return to the global
  }
}
```

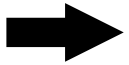


```
segment_1(addr) {
0x103b4: // j 0x30044
  goto 0x30044;
0x30044: // bne 0x103b4
  if (...!=0)
     goto 0x103b4;

0x50018:
    addr = segment_3(...); // function call
    if (addr!=0x5001C)
      return addr; // special care
}
```

Local Optimizations in Transparent Segments

- Several local optimizations can be effectively applied after the conversion.
  - The control flow in this segment is now significantly simple.
- e.g., SSA conversion
  - Register accesses to global memory are converted to SSAs in LLVM IR.

```
segment_1(addr) {
0x103b0: // addi r10+1
  op0 = load(&r10)
  tmp = op0 + 1 // addi
  store(&r10, tmp)

0x103b4: // bne 0x103b0
  op0 = load(&r10)
  if (op=!0)
    goto 0x103b0
```



```
segment_1(addr) {
%r10_1 = load(&r10); //sync

0x103b0: // addi
%r10_2 = %r10_1 + 1

0x103b4 // bne 0x103b0
  if (%r10_2 != 0)
        goto 0x103b0
    ...
  store(&r10, %r10_2) // sync
  return next_addr;
}
```

Optimization for Opaque Segments

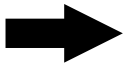
- These segments include indirect jumps.
- We cannot remove a dispatcher.
  - For each indirect jump, it should return to the global dispatcher.

```
segment_1(addr) {
  // Local dispatcher
  switch (addr) {
    case 0x103b2: // jr
      addr = load(&r1);
      return addr; // to the global
    case 0x203b2:
      ...
  }
}
```

Optimization for Opaque Segment

- Loops are introduced within local dispatchers.
- This optimization reduces unnecessary returns to the global dispatcher.

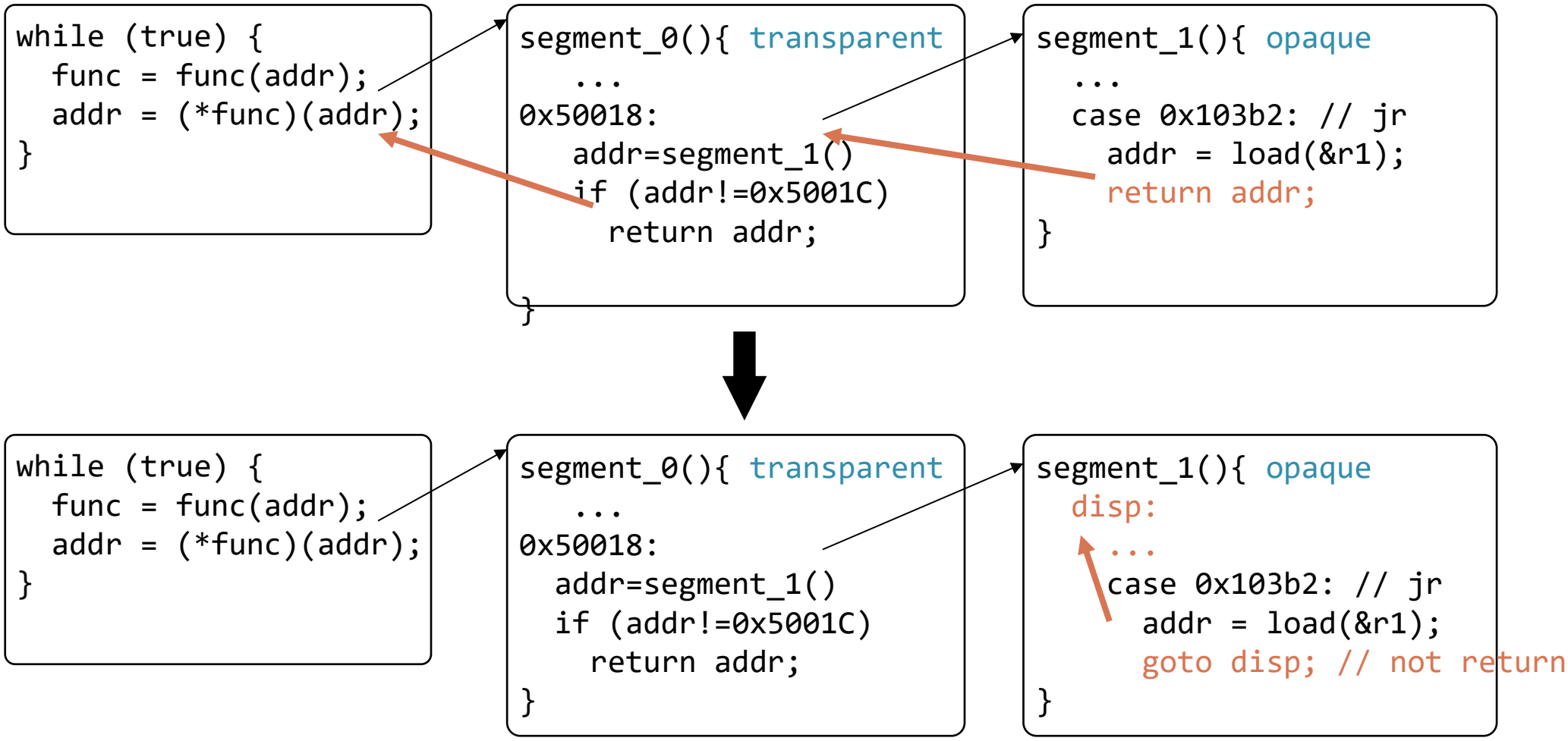
```
segment_1(addr) {
  // Local dispatcher
  switch (addr) {
    case 0x103b2: // jr
      addr = load(&r1);
      return addr; // to the global
    case 0x203b2:
      ...
  }
}
```



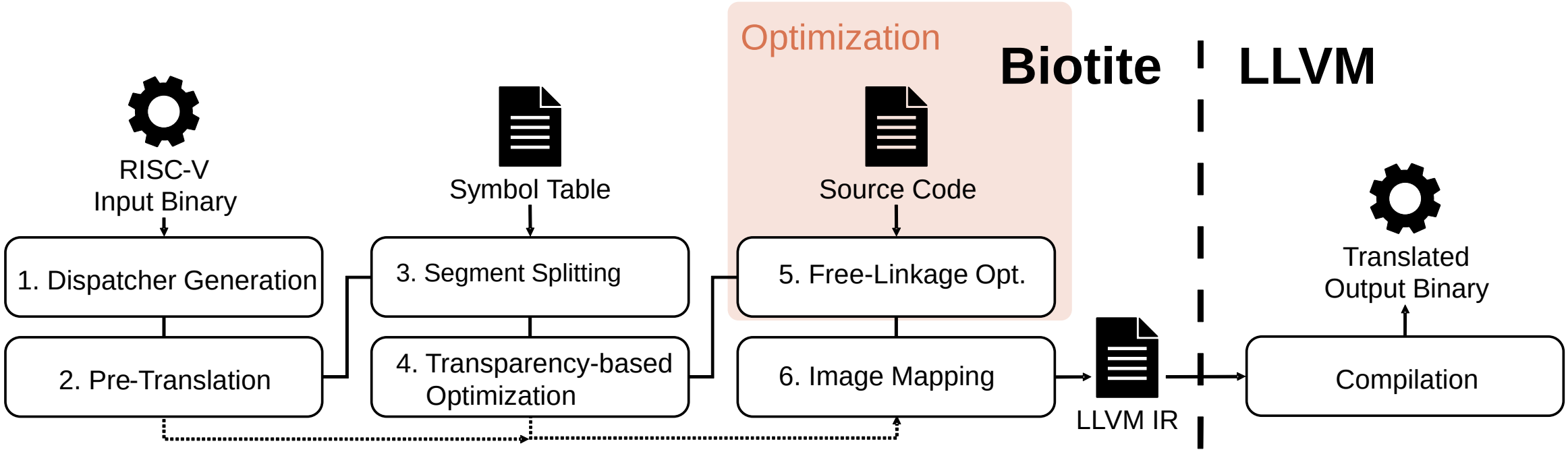
```
segment_1(addr) {
// Local dispatcher
disp:
  switch (addr) {
    case 0x103b2: // jr
      addr = load(&r1);
      goto disp;   // goto the head
    case 0x203b2:
      ...

    default:
      // fallback,
      // return to the global...
      return addr;
  }
}
```

Optimization for Opaque Segment

- This avoids unnecessary returns to the global dispatcher when called from a transparent segment.
- The optimized execution flow maintains local control whenever possible.



Optimization using Source Code



- Source code is leveraged for optimizations.
  - Free-linkage optimization technique

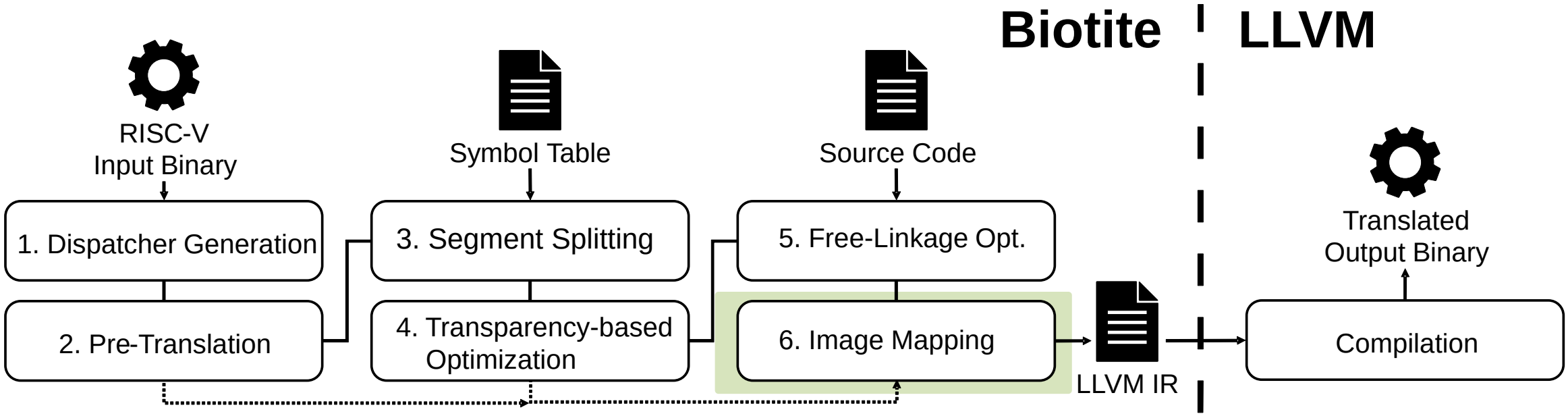
Free Linkage

- Translated functions are replaced with directly compiled LLVM IR whenever possible.
  - This applies only to functions that can be compiled from the source code.

- Specifically,
  - Rewrites function signature/calls and memory accesses in the directly compiled IRs.
  - Synchronization code is inserted for the arguments and simulated register values as in transparent segments.

```
; Adaptor function for `i64 @func(i64 %arg)`
; Assume the guest address of `@func` is 0x13fcc
define i64 @.u0x13fcc(i64 %entry) {
  %arg = load i64, ptr @.a0
  %rslt = call i64 @func(i64 %arg)
  store i64 %rslt, ptr @.a0
  %ra = load i64, ptr @.ra
  ret i64 %ra
}

; Some example instructions after rewriting
define i64 @func(i64 %arg) {
  ; Inst 1: %rslt = call i64 @g(i64 0)
  ; Assume the guest address of `@g` is 0x8e48f
  store i64 0, ptr @.a0
  call i64 @.u0x8e48f(i64 u0x8e48f)
  %rslt = load i64, ptr @.a0

  ; Inst 2: call void %func_ptr()
  %func_addr = ptrtoint ptr %func_ptr to i64
  call i64 @.find_func(i64 %func_addr)

  ; Inst 3: %value = load i64, ptr %var_ptr
  %var_addr = ptrtoint ptr %var_ptr to i64
  %host_ptr = call ptr @.get_mem_ptr(i64 %var_addr)
  %value = load i64, ptr %host_ptr
}
```

Memory Image Mapping



- Memory Image Mapping
  - Baseline mapping involves some dynamic address translation.
  - If possible, the translated memory image is placed at its original address using linker scripts.
  - It can removes address conversion overhead completely.

Outline

1. Binary Translation
    1. Static Binary Translation
    2. Dynamic Binary Translation
2. Biotite: A Static Binary Translator for Research ISAs
3. Evaluation

Evaluation Settings

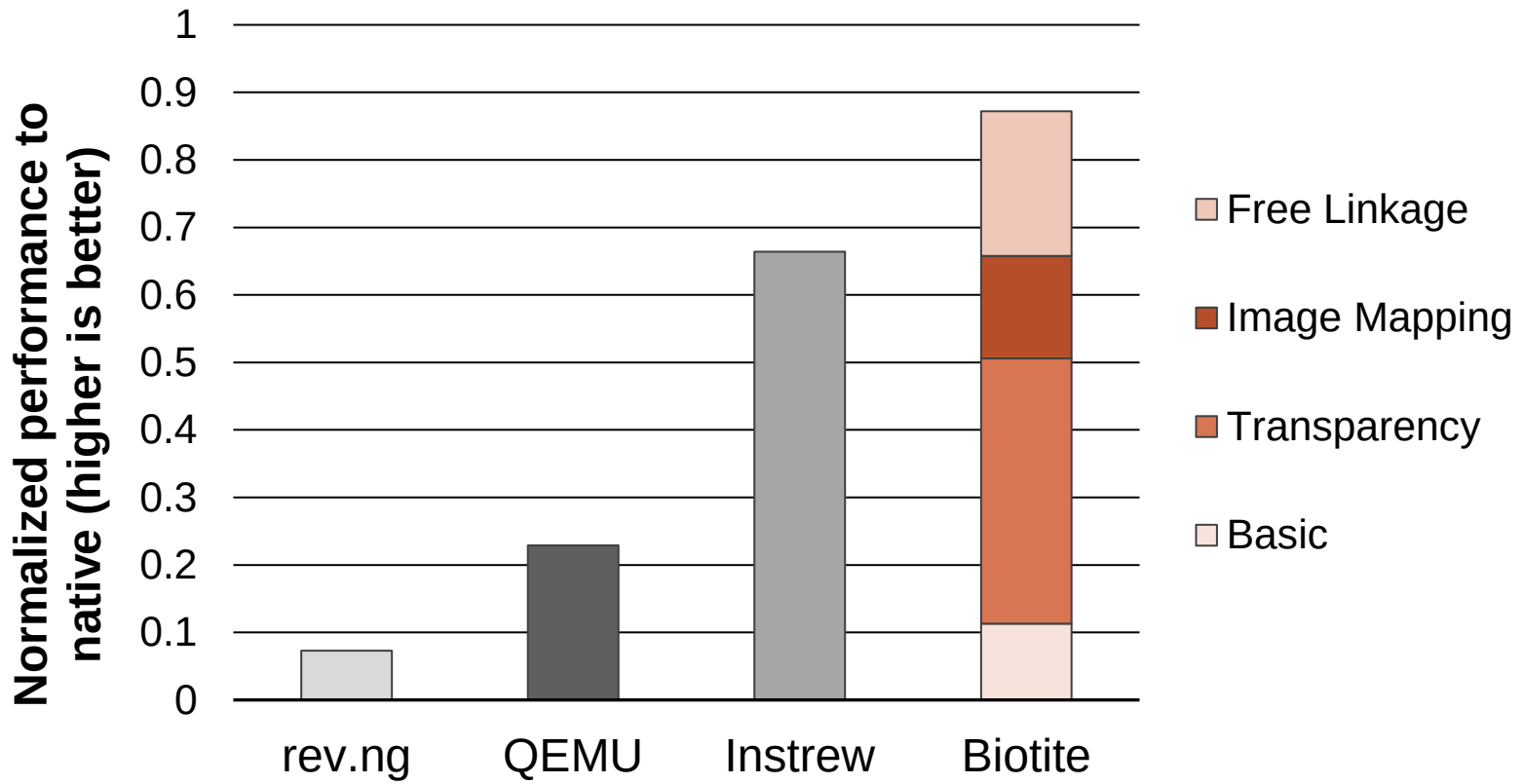
- Biotite implementation details:
  - 7,000 lines of Rust + 400 lines of LLVM IR
  - Compiled using Rust 1.85.0
- Performance evaluation:
  - Compared against various existing binary translators:
    - rev.ng – The only static translator that can translate SPEC CPU correctly
    - Instrew – A state-of-the-art dynamic translator
    - QEMU – A widely used dynamic translator
- Note: rev.ng does not support RISC-V, so ARM binaries were used as input.
- We measure Biotite's performance when translating RISC-V binaries to x86-64.

Evaluation Settings

- Benchmark suites:
  - Coremark: A simple integer benchmark used in embedded environments.
  - SPEC CPU 2017 (Integer): A widely used CPU benchmark suite.
  - NoFib: A Haskell benchmark suite included in GHC.
- Compilers used for input binary compilation :
  - GCC 13 – for Coremark
  - GCC 8 – used for SPEC CPU due to compatibility issues in RISC-V GCC
  - GHC 8.8 – for NoFib

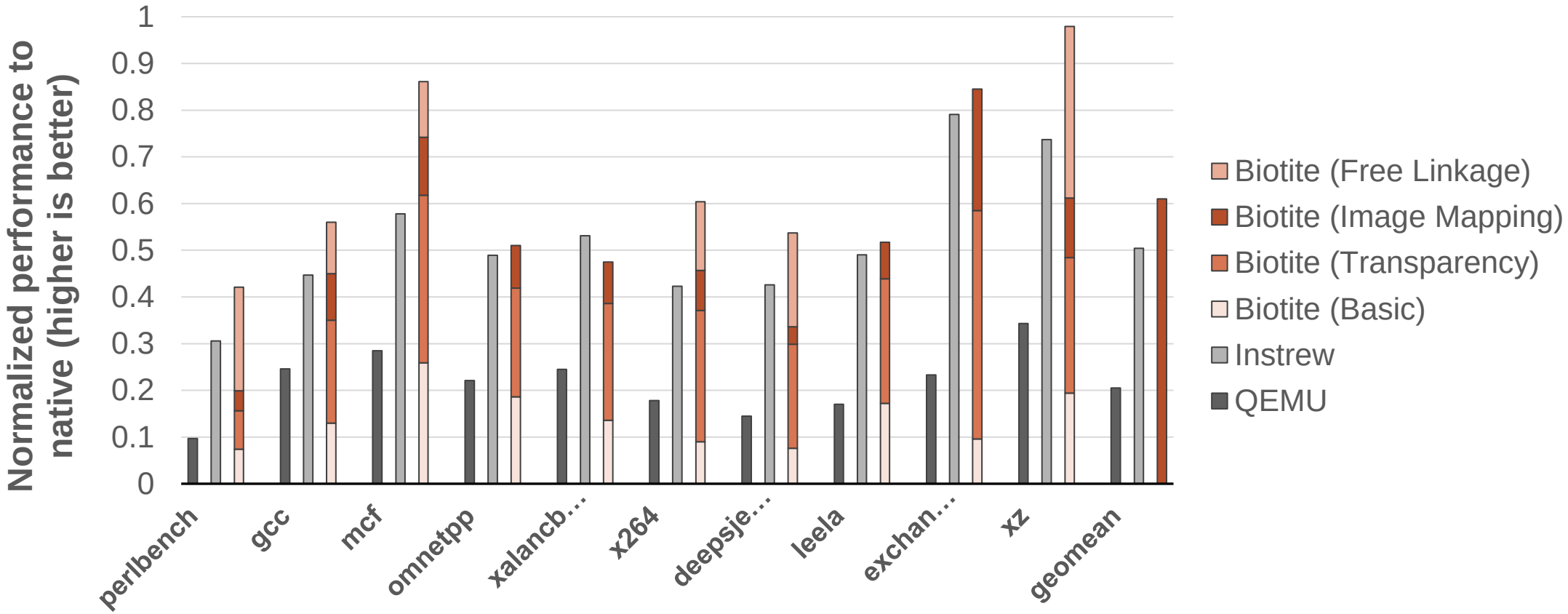
Coremark Results

- Biotite achieves 87% of native execution performance.
  - 24% faster than Instrew and 280% faster than QEMU
- rev.ng significantly slower due to its complex indirect jump handling.
  - Excluded from further evaluations



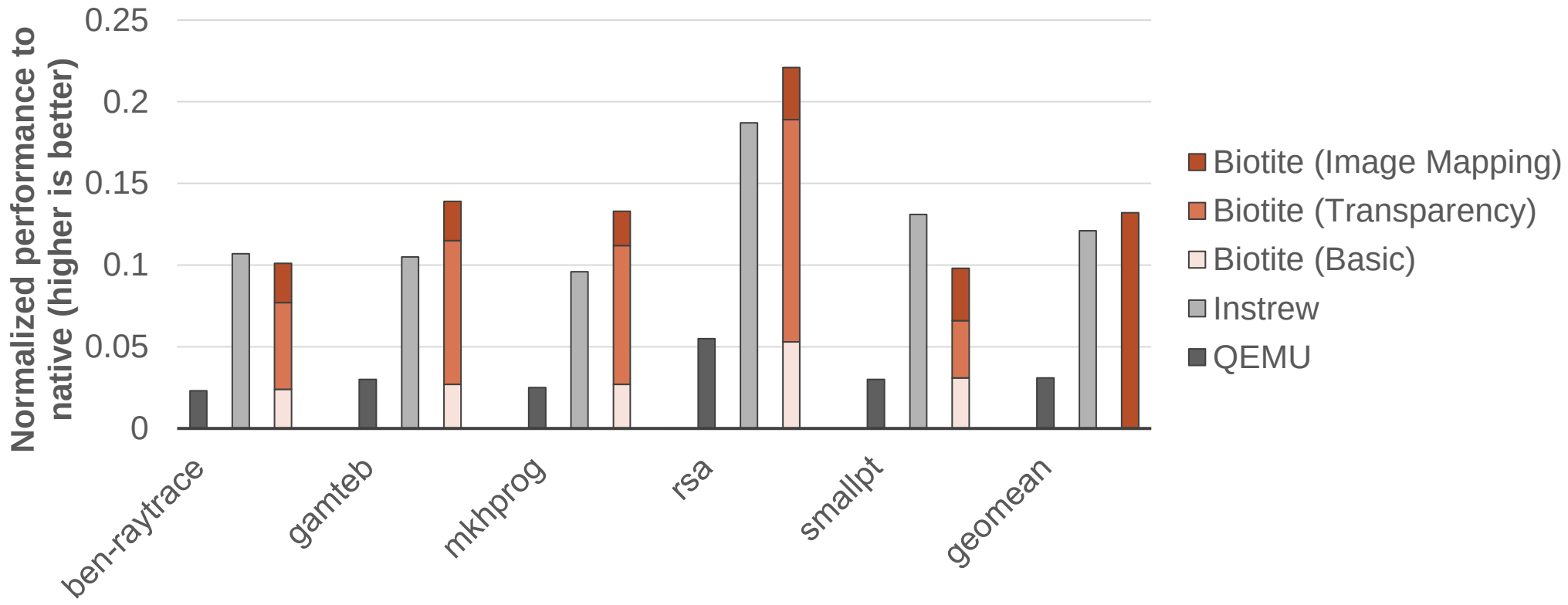
SPEC CPU 2017 (Integer) Results

- All evaluated translators translated SPEC CPU 2017 correctly.
- Biotite achieves 61% of native execution performance.
  - The xz benchmark achieves performance nearly equivalent to native execution.
  - 21% faster than Instrew and 198% faster than QEMU



NoFib (Haskell Benchmarks) Results

- Free linkage is not yet supported for Haskell.
- Biotite achieves 13% of native execution performance.
  - 9.1% faster than Instrew and 326% faster than QEMU



STRAIGHT ISA Support

- STRAIGHT is a new research ISA.
  - The toolchain includes only a basic LLVM backend and a small libc.

- Biotite successfully translates not only complex C programs but also C++ and Fortran.
  - No C++ or Fortran standard libraries are available for STRAIGHT.

SPEC CPU 2017 Integer	Native STRAIGHT	Via Biotite
perlbench_s (C)	✗	✓
gcc_s (C)	✓	✓
mcf_s (C)	✓	✓
omnetpp_s (C++)	✗	✓
xalancbmk_s (C++)	✗	✓
x264_s (C)	✓	✓
deepsjeng_s (C++)	✗	✓
leela_s (C++)	✗	✓
exchange2_s (Fortran)	✗	✓
xz_s (C)	✓	✓

SPEC CPU 2017 Floating Point	Native STRAIGHT	Via Biotite
bwaves_s (Fortran)	✗	✓
cactuBSSN_s (C++, C, Fortran)	✗	✓
lbm_s (C)	✓	✓
wrf_s (Fortran, C)	✗	✓
cam4_s (Fortran, C)	✗	✓
pop2_s (Fortran, C)	✗	✓
imagick_s (C)	✗	✓
nab_s (C)	✓	✓
fotonik3d_s (Fortran)	✗	✓
roms_s (Fortran)	✗	✓

✓: Success ✗: Failure

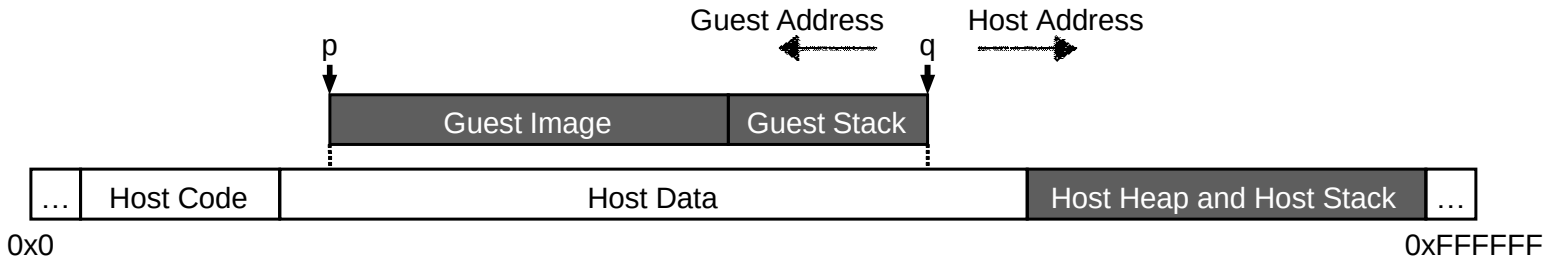
Conclusion

- We propose Biotite, a high-performance static binary translator.
  - Our goal is to address the many challenges associated with toolchains in new ISAs.
  - Biotite utilizes source-level information for optimization.
    - Accurate and fast, yet simple
- Evaluation results:
  - Biotite accurately translates a wide range of programs, including SPEC CPU 2017 (C, C++, Fortran) and Haskell.
  - CoreMark Performance:
    - 87.2% of native performance (16.8 sec vs 14.7 sec)
    - 280% faster than QEMU (16.8 sec vs 64.0 sec)
- Open-source Release: https://github.com/shioya-lab-public/biotite

backup

Memory Layout

- Guest and host memory are separated using a threshold value q.
- This enables single-pointer representation without requiring additional metadata.




```
get_mem_ptr(addr) {
    if (addr < q) {
        // RISC-V guest address
        return p + addr;
    }
    else {
        // host heap
        return addr;
    }
}
```

Memory Address Optimization

- Removes address conversion overhead completely.
  - The guest image is placed at the original address using linker scripts.