Hyperchaining Optimizations for an LLVM-Based Binary Translator on x86-64 and RISC-V Platforms

Jyun-Kai Lai and Wuu Yang
Department of Computer Science,
National Yang Ming Chiao Tung University, Taiwan

Outline

- Background
- Hyperchaining
- Experiments
- Conclusion

Outline

- **Background**
- Hyperchaining
- Experiments
- Conclusion

Binary translation

- Translate from guest (input) binary to host (output) binary and the translated program runs on a host machine
- **Rabbit** - an LLVM-based binary translator developed by our lab
  - Our work is based on Rabbit and translate from RISC-V binary to x86-64

Architecture

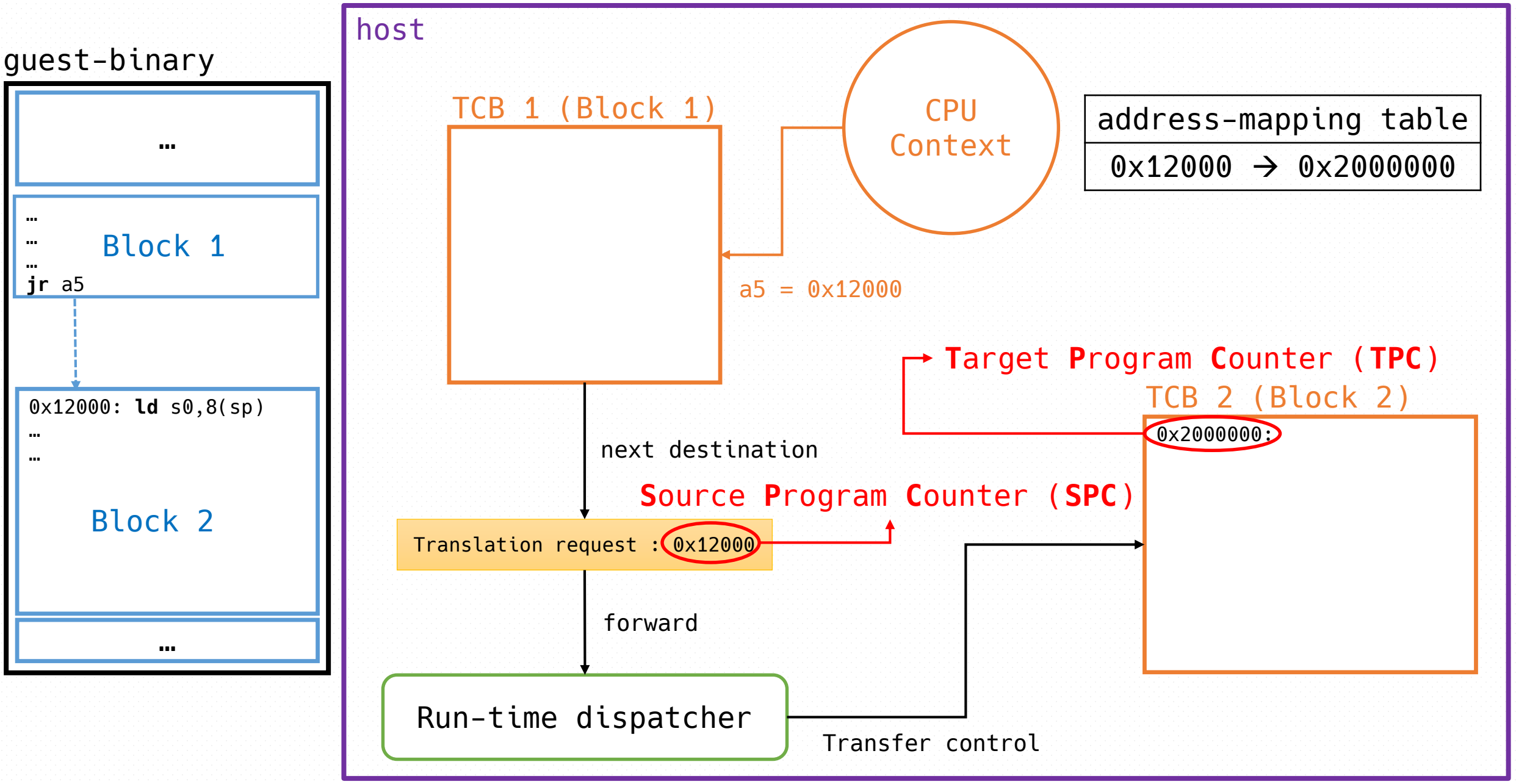
- Frontend (Guest → LLVM IR)
  - Translate from guest binary to LLVM IR basic blocks
  - Incorporate one or more instructions into a LLVM IR basic block depend on SBT or DBT
- Backend (LLVM IR → Host)
  - Translate from LLVM IR basic blocks to binary chunks

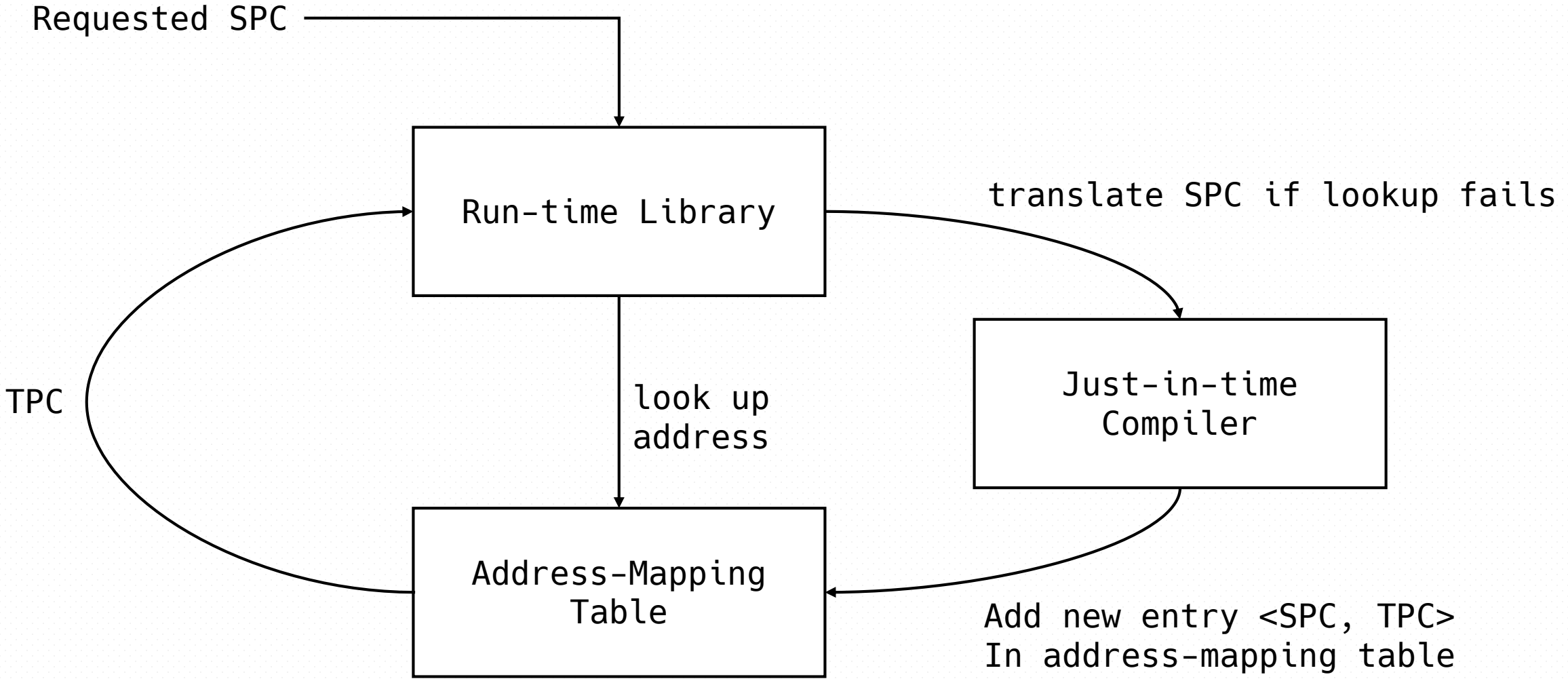
Translation

- Static Binary Translation (SBT)
  - Translate code blocks before run-time
- Dynamic Binary Translation (DBT)
  - Translate code blocks in run-time
- Hybrid Binary Translation (HBT; adopted by Rabbit)
  - SBT + DBT

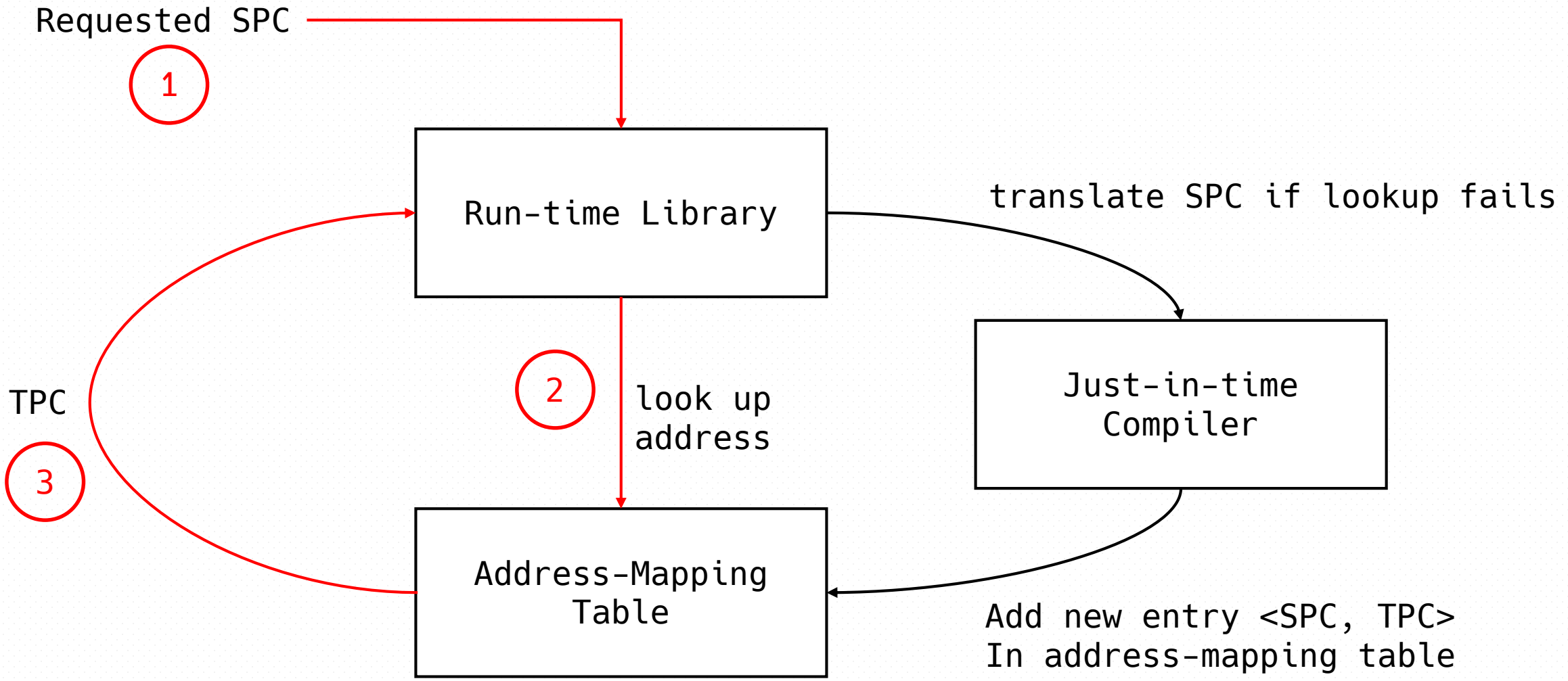
Translated code block

- Translation phase processes the client's instruction and translated into many code blocks, which we call **T**ranslated **C**ode **B**locks (**TCB**s)
- You may get many guest-to-host mapping relationship in the translation phase

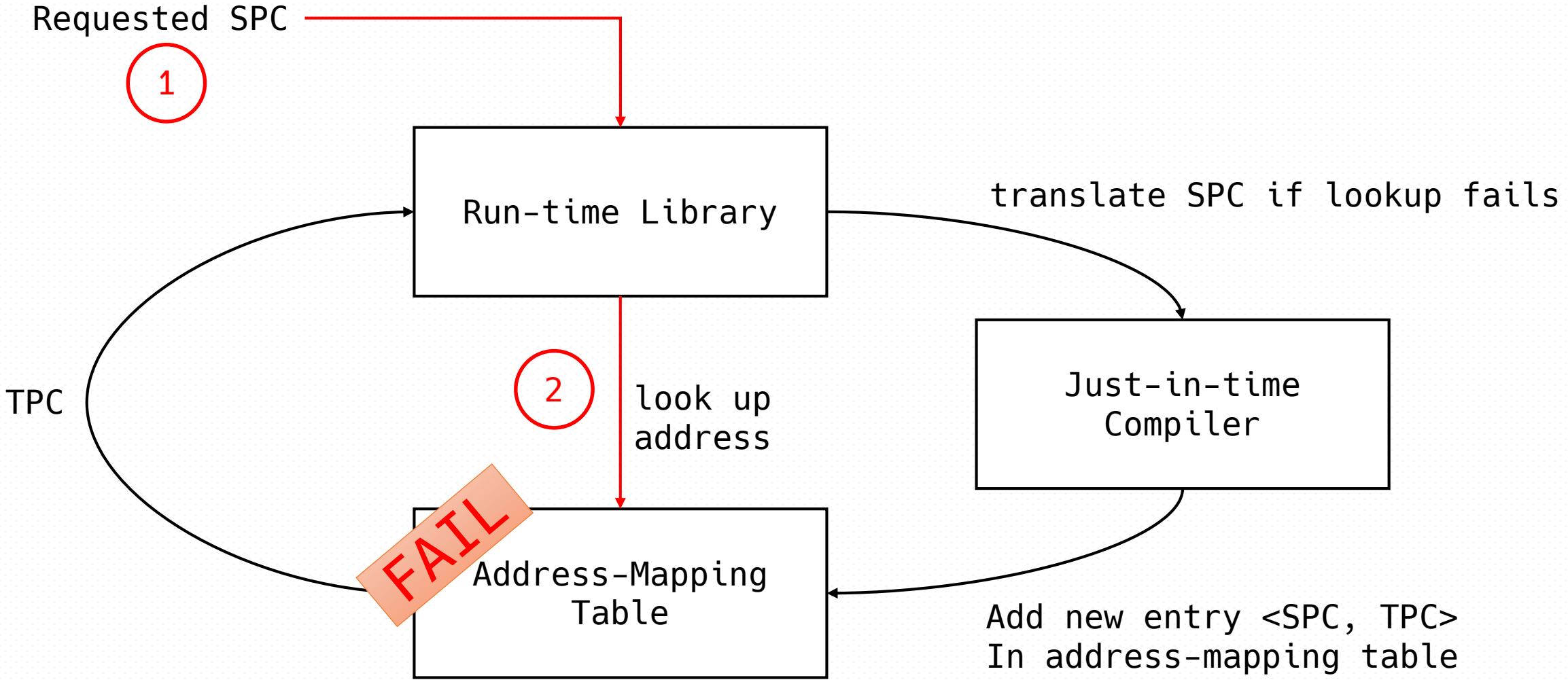

guest-binary
…
…
…
…
jr a5
Block 1
0x12000: ld s0,8(sp)
…
…
Block 2
…
host
TCB 1 (Block 1)
CPU Context
address-mapping table
0x12000 → 0x2000000
a5 = 0x12000
Target Program Counter (TPC)
TCB 2 (Block 2)
0x2000000:
next destination
Source Program Counter (SPC)
Translation request : 0x12000
forward
Run-time dispatcher
Transfer control



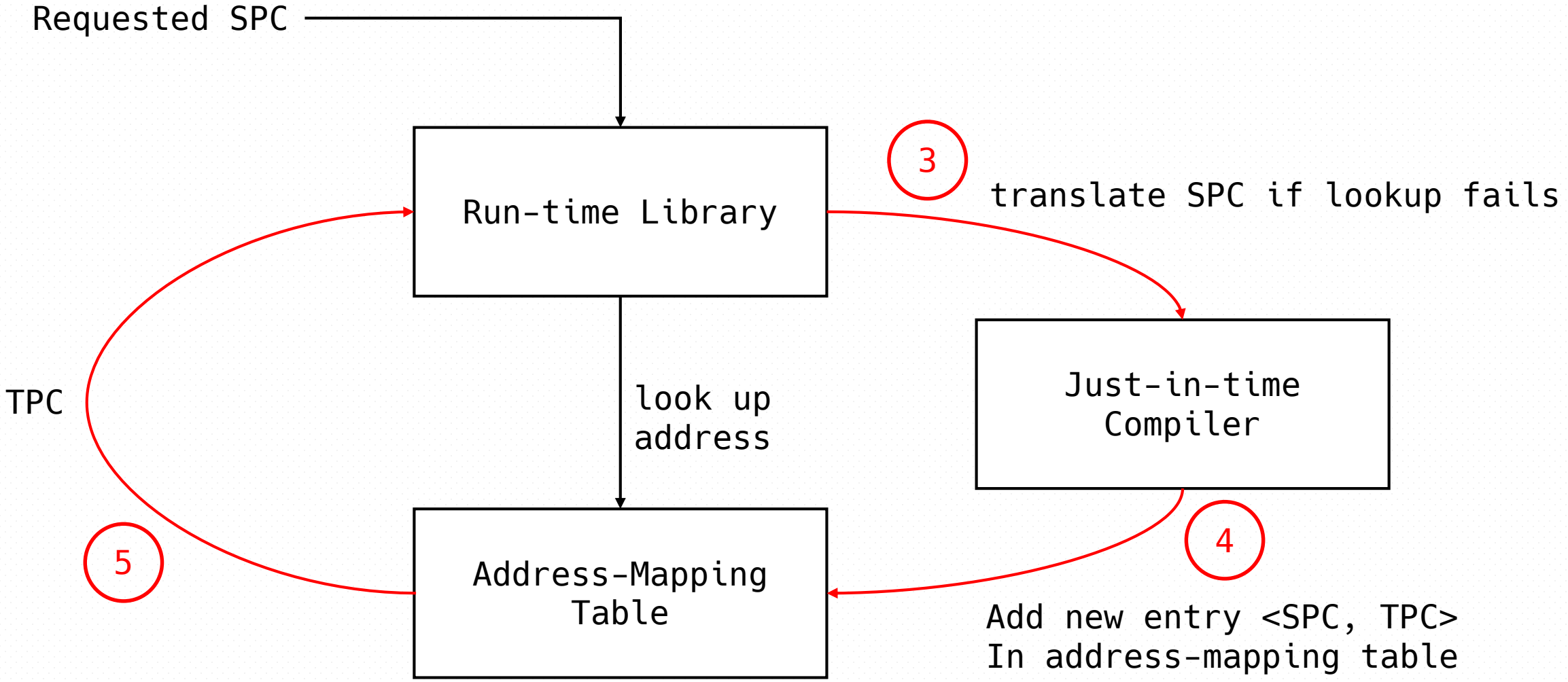
Possible Case 1 : the source block has been translated (from SBT or DBT)
Possible Case 2 : the source block has never been translated



Possible Case 1 : the source block has been translated (from SBT or DBT)
Possible Case 2 : the source block has never been translated



Possible Case 1 : the source block has been translated (from SBT or DBT)
Possible Case 2 : the source block has never been translated



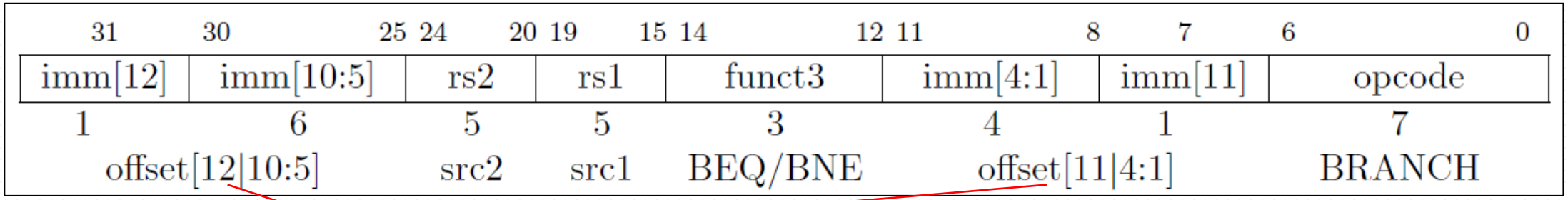
Possible Case 1 : the source block has been translated (from SBT or DBT)
Possible Case 2 : the source block has never been translated

Instruction Terminator

- Every basic block ends with an instruction terminator
- Can be classified into two groups
  - Direct Branches
  - Indirect Branches

Direct Branches

- Conditional branches, unconditional branches, direct function calls
- Single destination
- It is possible to know the destination in compilation time

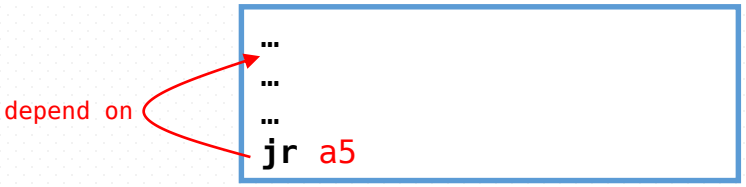


From https://github.com/riscv/riscv-isa-manual/releases/download/draft-20210612-98f3349/riscv-spec.pdf

Target ← $PC + offset (PC-relative addressing)

Indirect Branches

- May have more than one destination
- The destination may be stored in a register or memory
- It is difficult to predict the destinations without context information



The value of a5 ?

Optimizations in Rabbit

- Direct Branch Link Optimization
  - In compilation time
- Direct Function Return Optimization
  - In compilation time
- Block Chaining Optimization
  - In run-time

Direct Branch Link

- Steps
  1. Analyze LLVM basic blocks which end with direct branches
  2. Try to find the SPC of successor block as much as possible and modify the former block to directly jump to the next one in compilation time
- Mitigate run-time overhead and improve performance

Direct Function Return

- In the normal situation, the return address of a function call should be the address of next instruction
- This optimization exploits that fact to tell the translator the TPC of return address in compilation time
  - No need to lookup return address in run-time

Guest-binary

```
<foo>:
...
ld   s0,8(sp)
addi sp,sp,16
ret
```
Block 1

...

```
...
α-4 : addi s0,sp,16
α+0 : jal  ra,<foo>
```
Block 2

```
α+4 : li   a5,0
...
```
Block 3

...

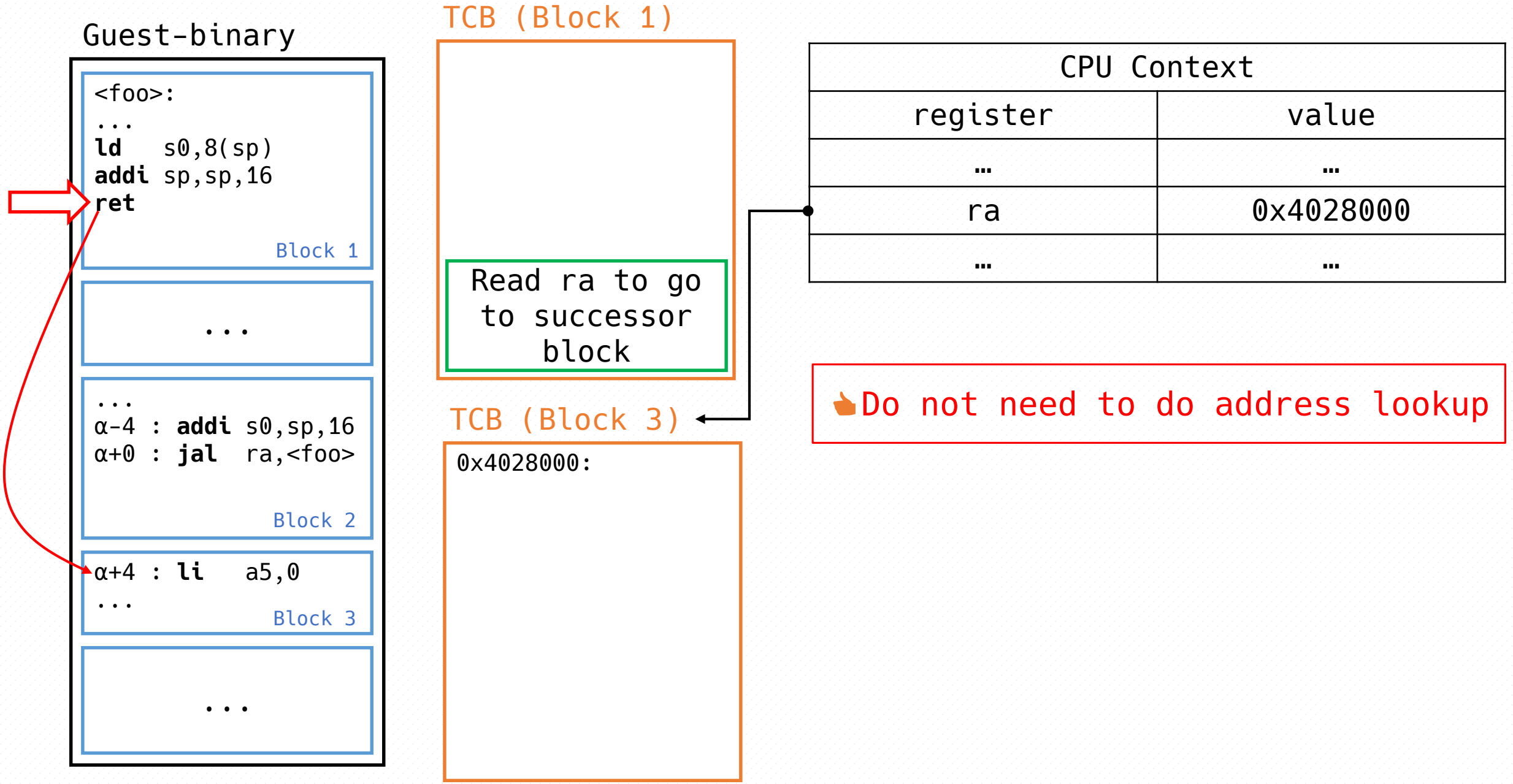
TCB (Block 2)

Code to write TCP of return address to CPU Context

CPU Context	
register	value
…	…
ra	~~α+4~~ ➔ 0x4028000
…	…

Replaced by TPC

Address-Mapping Table	
SPC	TPC
…	…
α+4	0x4028000
…	…


Guest-binary
<foo>:
...
ld s0,8(sp)
addi sp,sp,16
ret
Block 1
...
...
α-4 : addi s0,sp,16
α+0 : jal ra,<foo>
Block 2
α+4 : li a5,0
...
Block 3
...
TCB (Block 1)
Read ra to go to successor block
TCB (Block 3)
0x4028000:
CPU Context
register
value
…
…
ra
0x4028000
…
…
Do not need to do address lookup

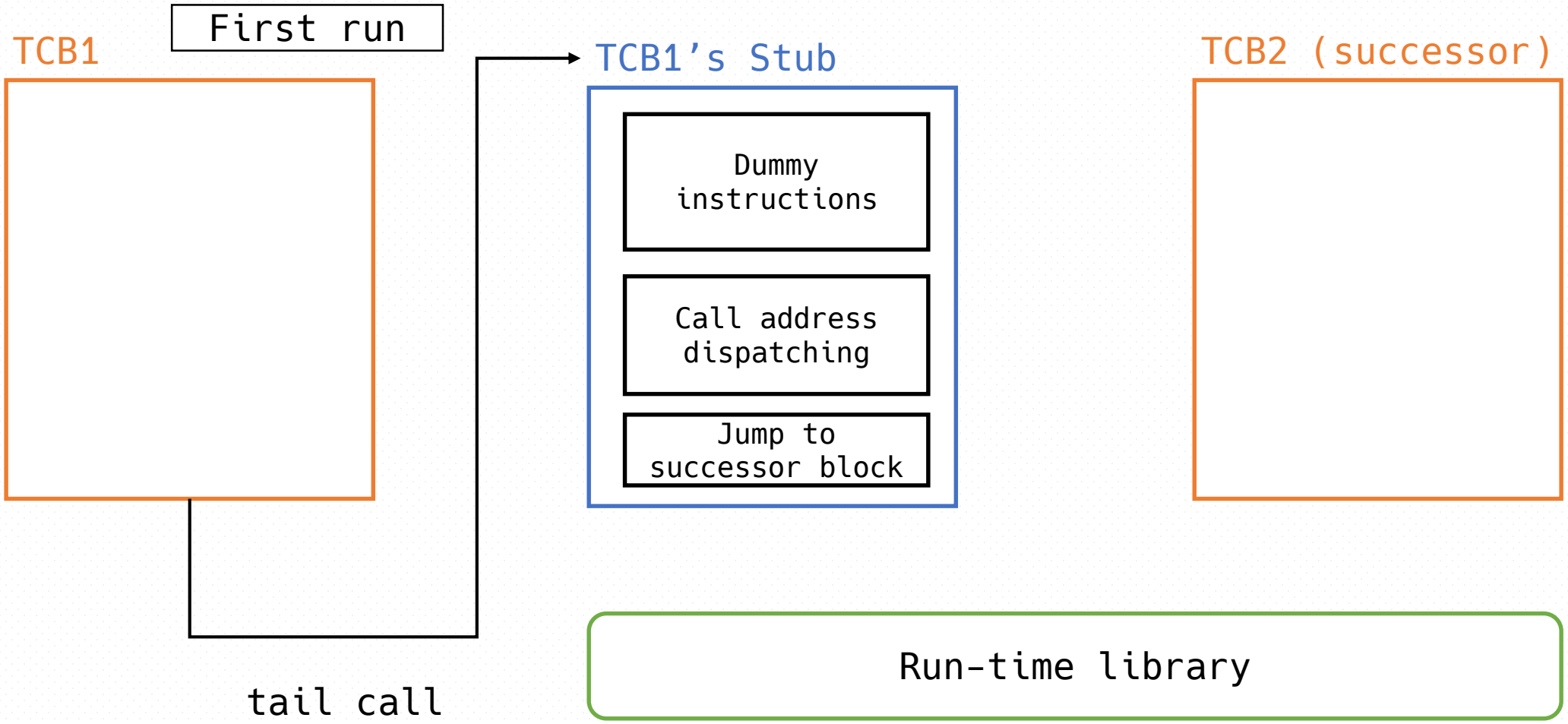
Direct Function Return

- Caveat : We assume the link register will not be modified in the subroutine
- Otherwise, the program will crash because of corrupt link register

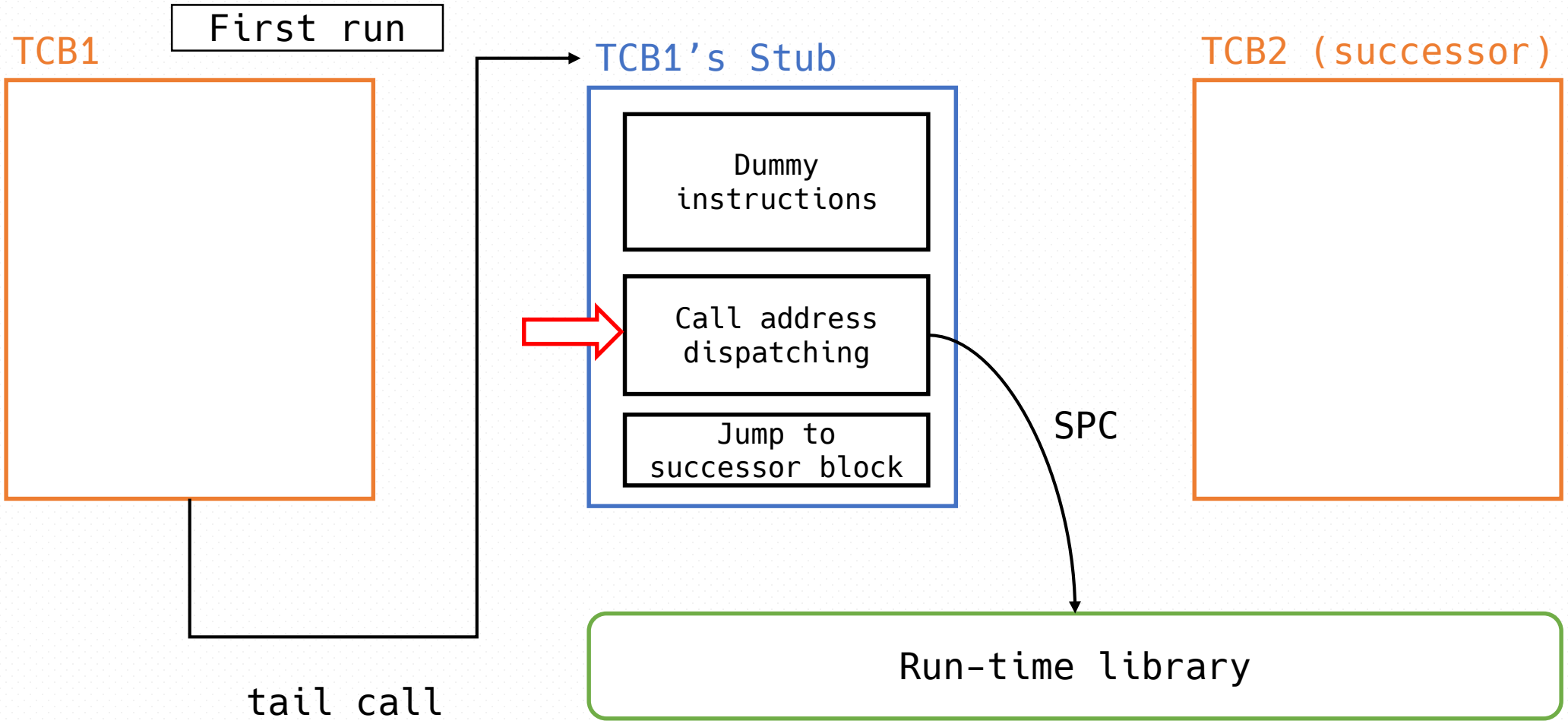
Block Chaining Optimization

- Basically, block chaining optimization do the same thing as direct branch link optimization does but in run-time
- Adopt **self-modifying code**
  - Modify its own instructions in run-time
  - Patch the destination into the code

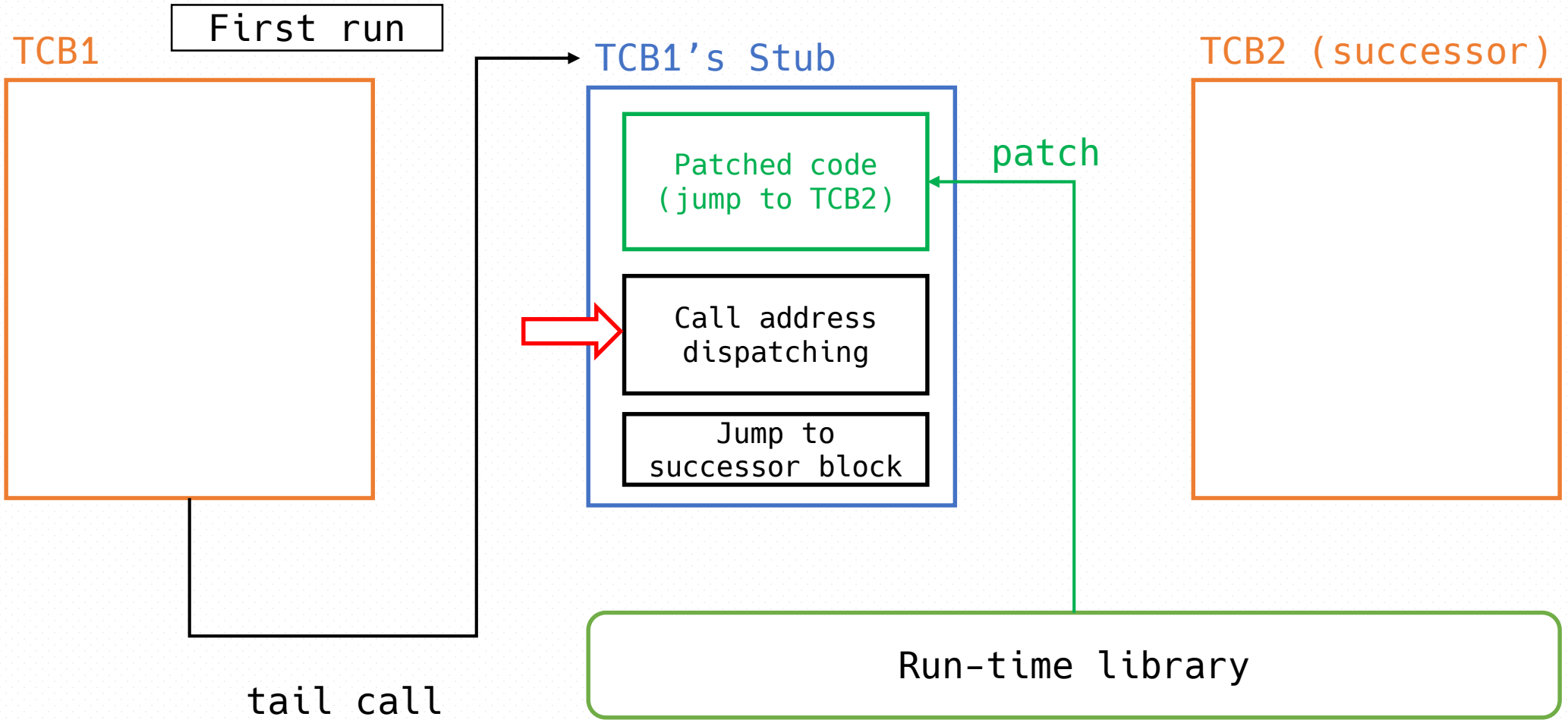
Block Chaining Optimization



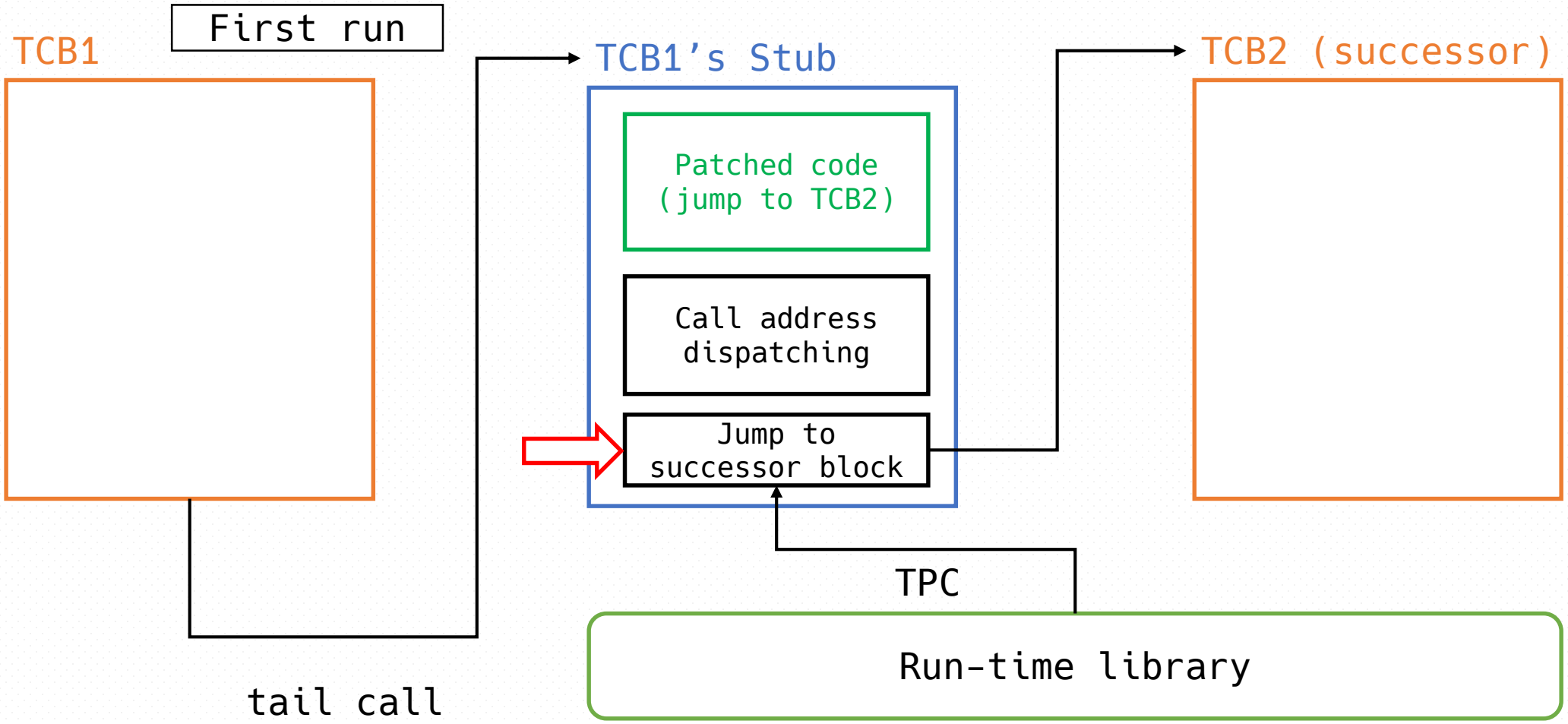
Block Chaining Optimization



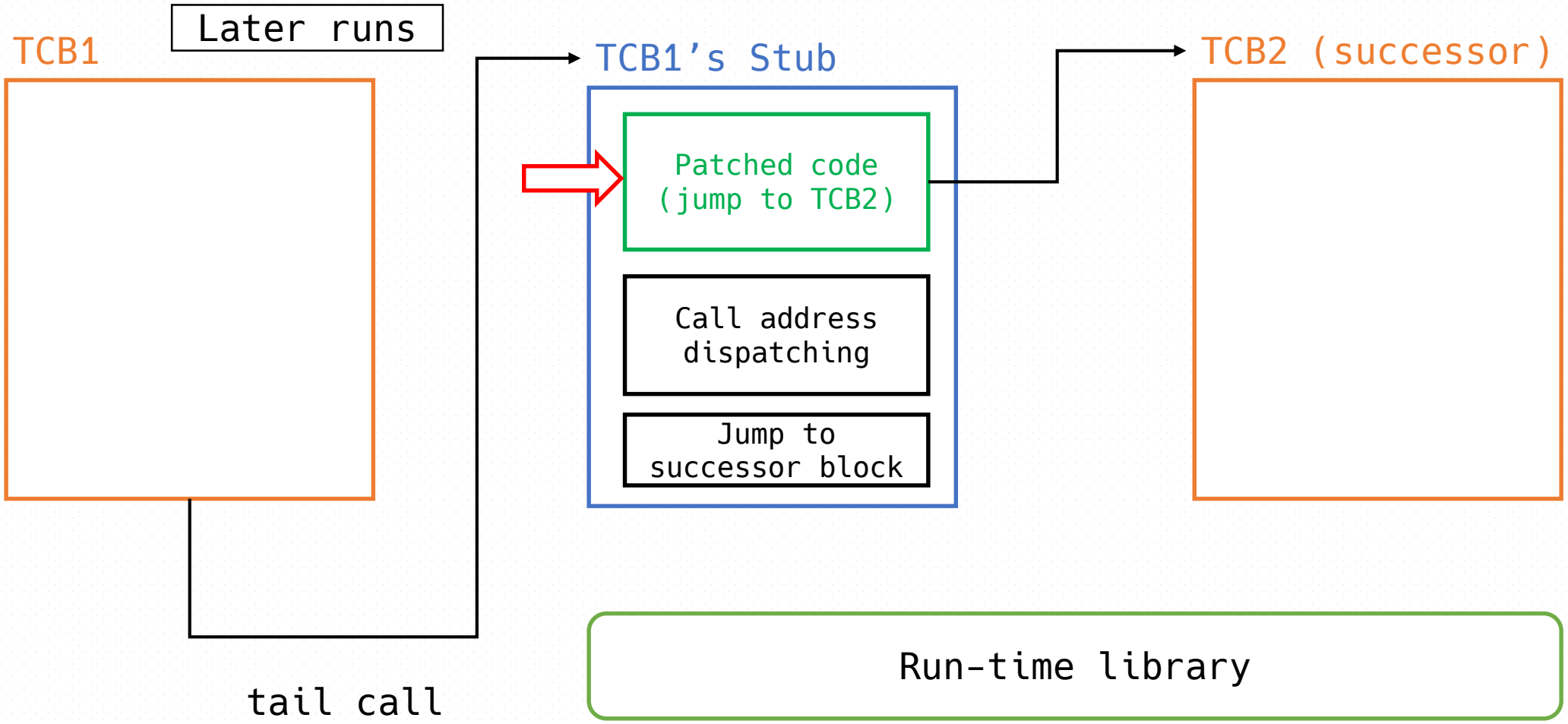
Block Chaining Optimization



Block Chaining Optimization



Block Chaining Optimization



Block Chaining Optimization

- Limitations
  - Only support chaining dtblocks to dtblocks
    - That means this optimization can’t handle (stblocks → stblocks; stblocks → dtblocks and dtblocks → stblocks) chaining
  - Tail call optimization is involved that means we have an extra jump instruction. That is not efficient
  - Cannot handle indirect branch chaining
  - This optimization only supports old translator we had made before. Rabbit cannot use this optimization

Outline

- Background
- **Hyperchaining**
- Experiments
- Conclusion

Hyperchaining

- We develop **Hyperchaining** optimization, an improved version of block chaining optimization
- Be able to chain both stblocks and dtblocks
- Be able to handle indirect branches
- Design to work well with HBT and Rabbit

Hyperchaining

- Depend on the implementation, hyperchaining optimization can be classified into two groups
  - **Platform-independent** (indep) hyperchaining
  - **Platform-dependent** (dep) hyperchaining

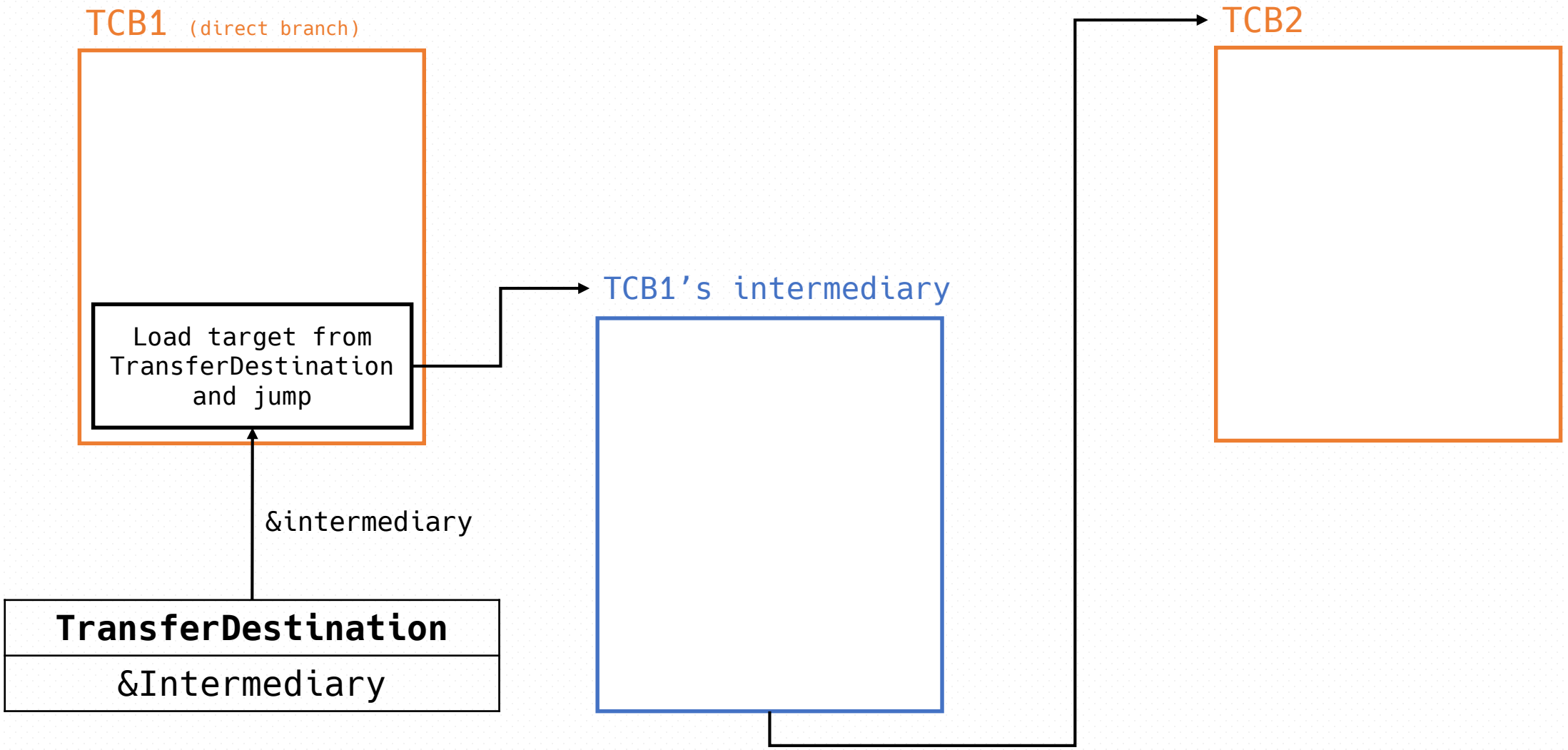
Indep-Hyperchaining

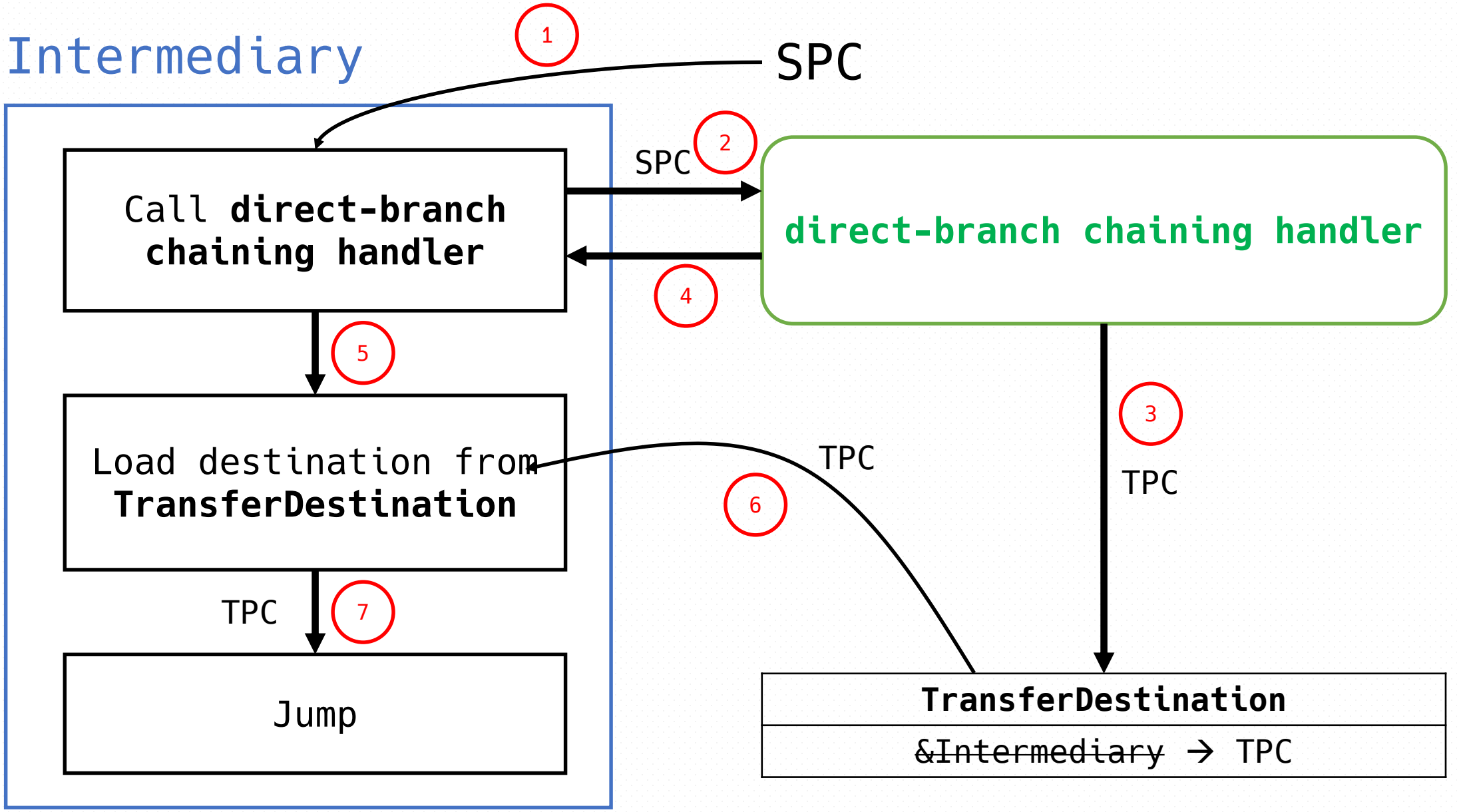
- In indep-hyperchaining， we design a subroutine, **direct-branch chaining handler**, to chain blocks for direct branches
- Implemented in LLVM IR level
- This subroutine supports some functionalities of run-time dispatcher
  - Dynamic binary translation
  - Address lookup

Indep-Hyperchaining

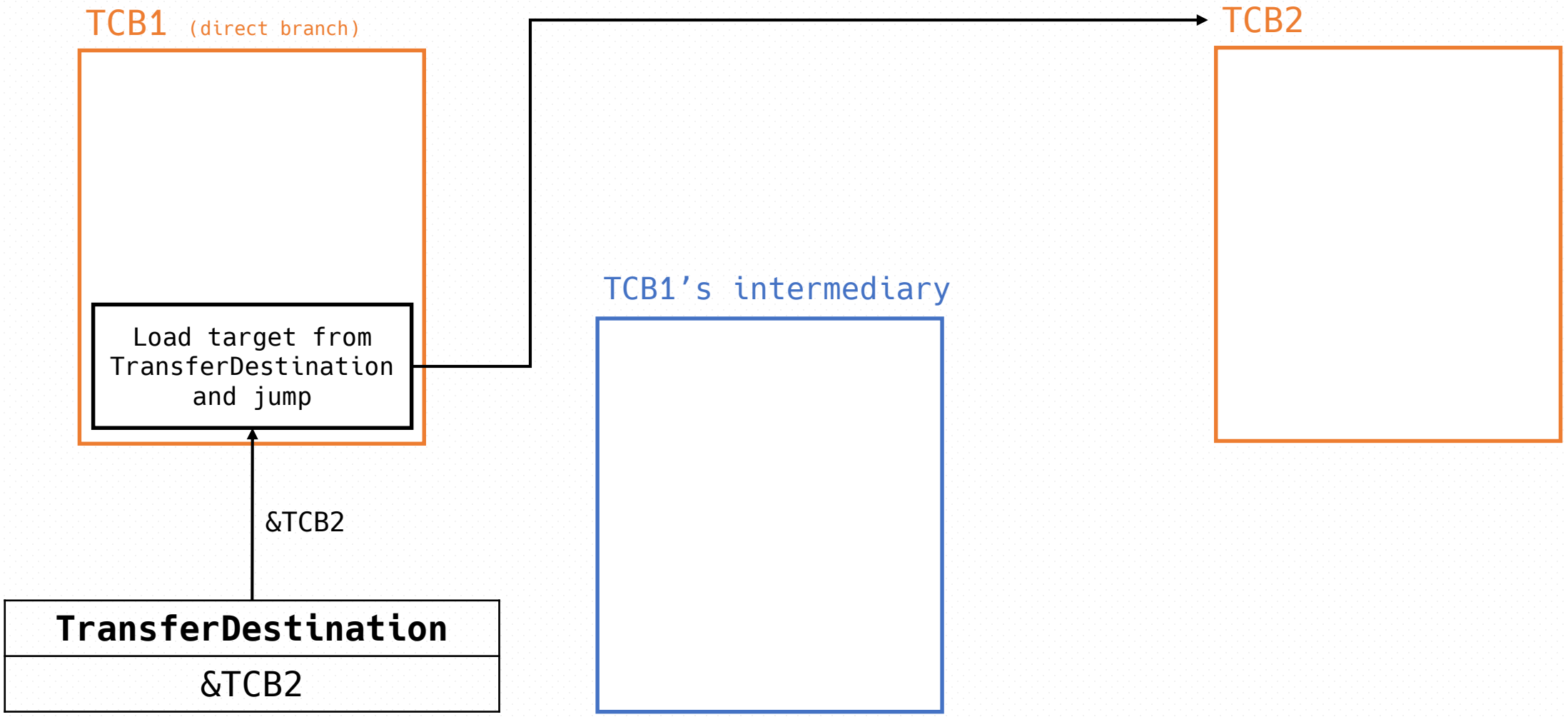
- Steps
  1. TCB loads address from a dedicated memory, **TransferDestination**, for every TCB
     - TransferDestination is initialized with the address of **Intermediary**
     - Intermediary calls direct-branch chaining handler to chain blocks and update TransferDestination with TPC

First run


TCB1 (direct branch)
Load target from TransferDestination and jump
TCB1's intermediary
TCB2
&intermediary
TransferDestination
&Intermediary


Intermediary
1
SPC
SPC
2
Call direct-branch chaining handler
direct-branch chaining handler
4
5
3
TPC
Load destination from TransferDestination
TPC
6
TPC
7
Jump
TransferDestination
&Intermediary → TPC

Second run

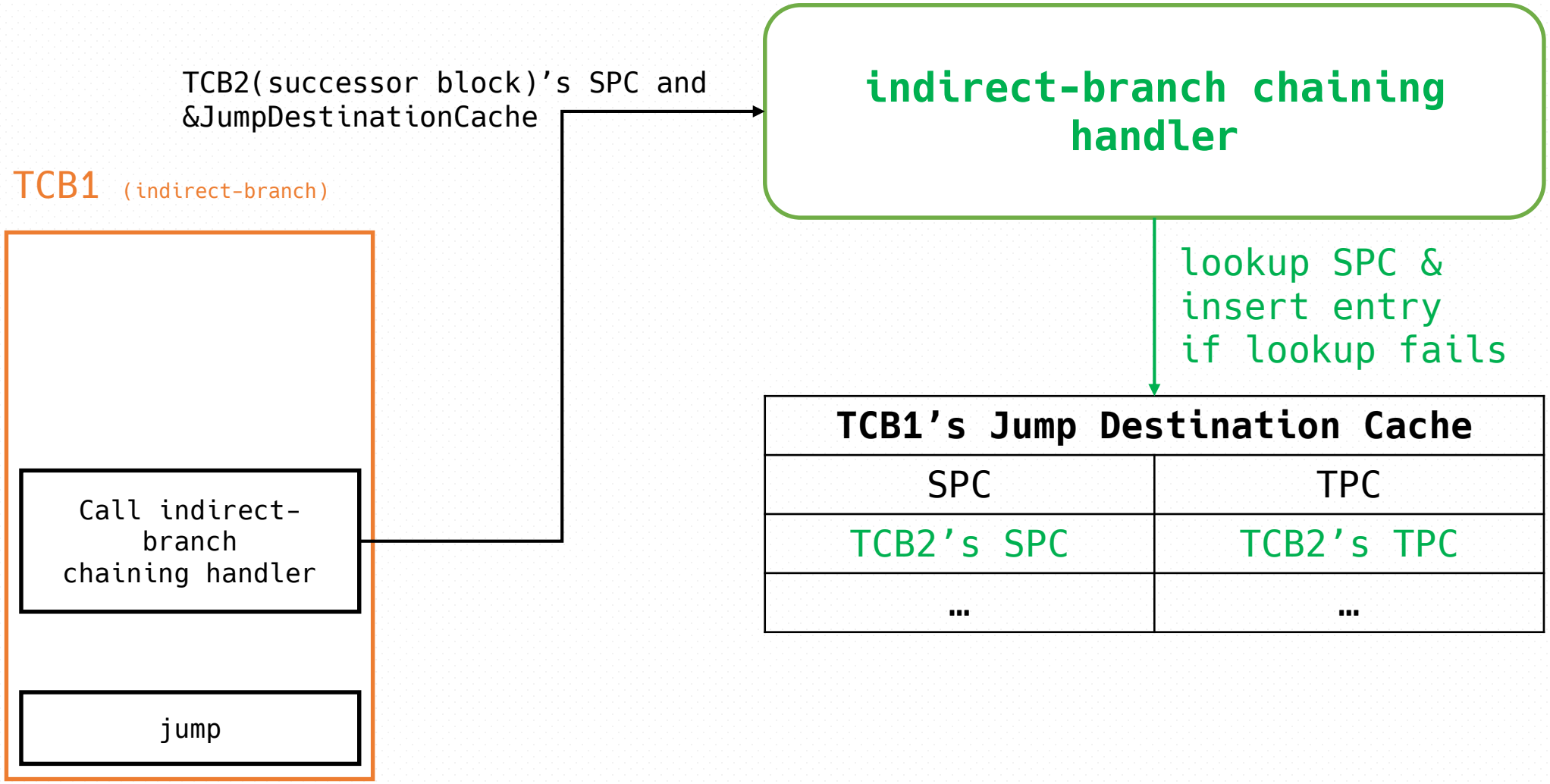

TCB1 (direct branch)
TCB2
TCB1's intermediary
Load target from TransferDestination and jump
&TCB2
TransferDestination
&TCB2

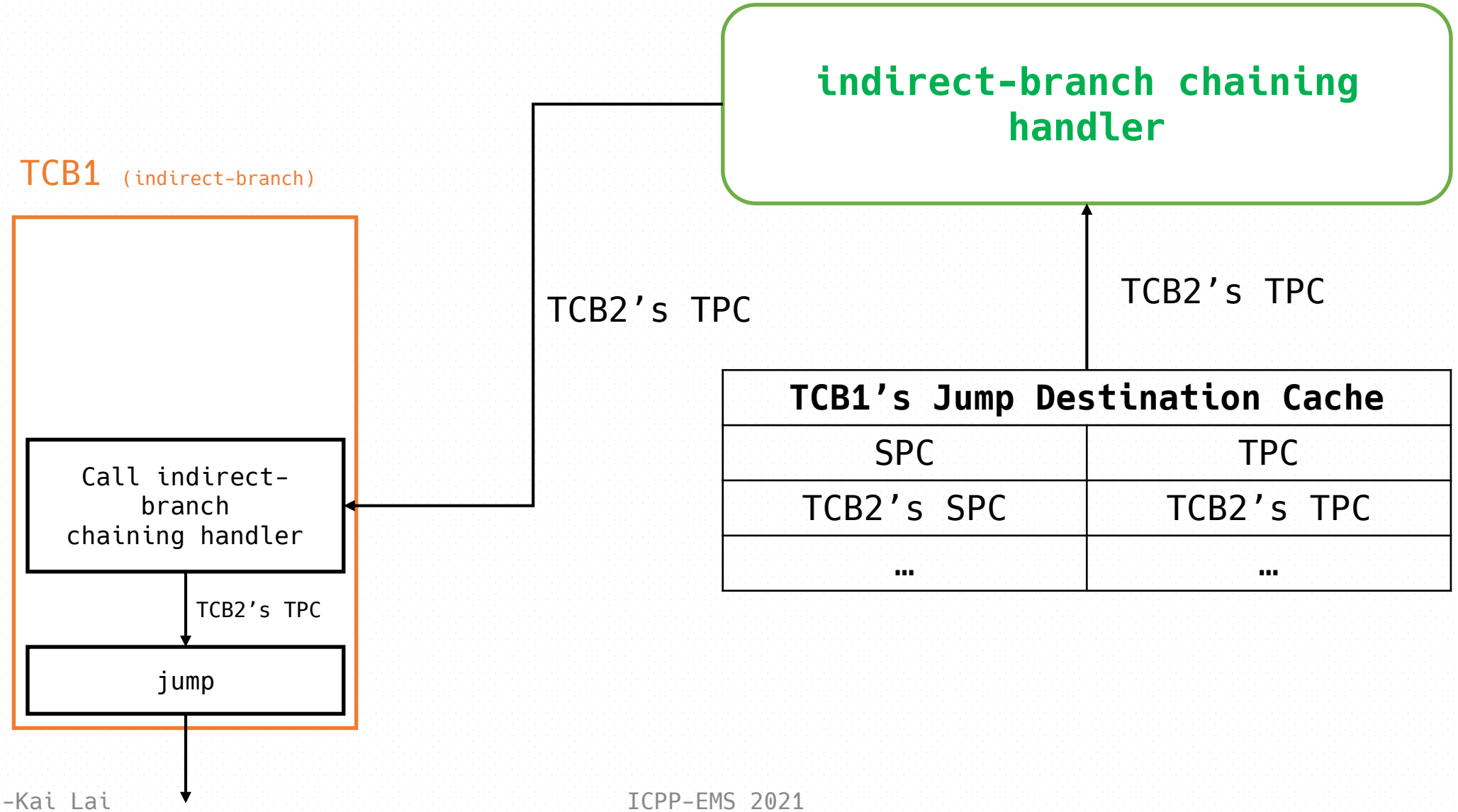
Indep-Hyperchaining For Indirect Branches

- For indirect branch, we have similar subroutine, **indirect-branch chaining handler**, to chain blocks with indirect branches
- Implemented in LLVM IR level

Indep-Hyperchaining For Indirect Branches

- Steps
  - TCB calls indirect-branch chaining handler and pass SPC to handler
  - Chaining handler lookups SPC in **Jump Destination Cache** first. If nothing is found, do address lookup to acquire corresponding TPC and store address translation pair entry <SPC, TPC> to that cache

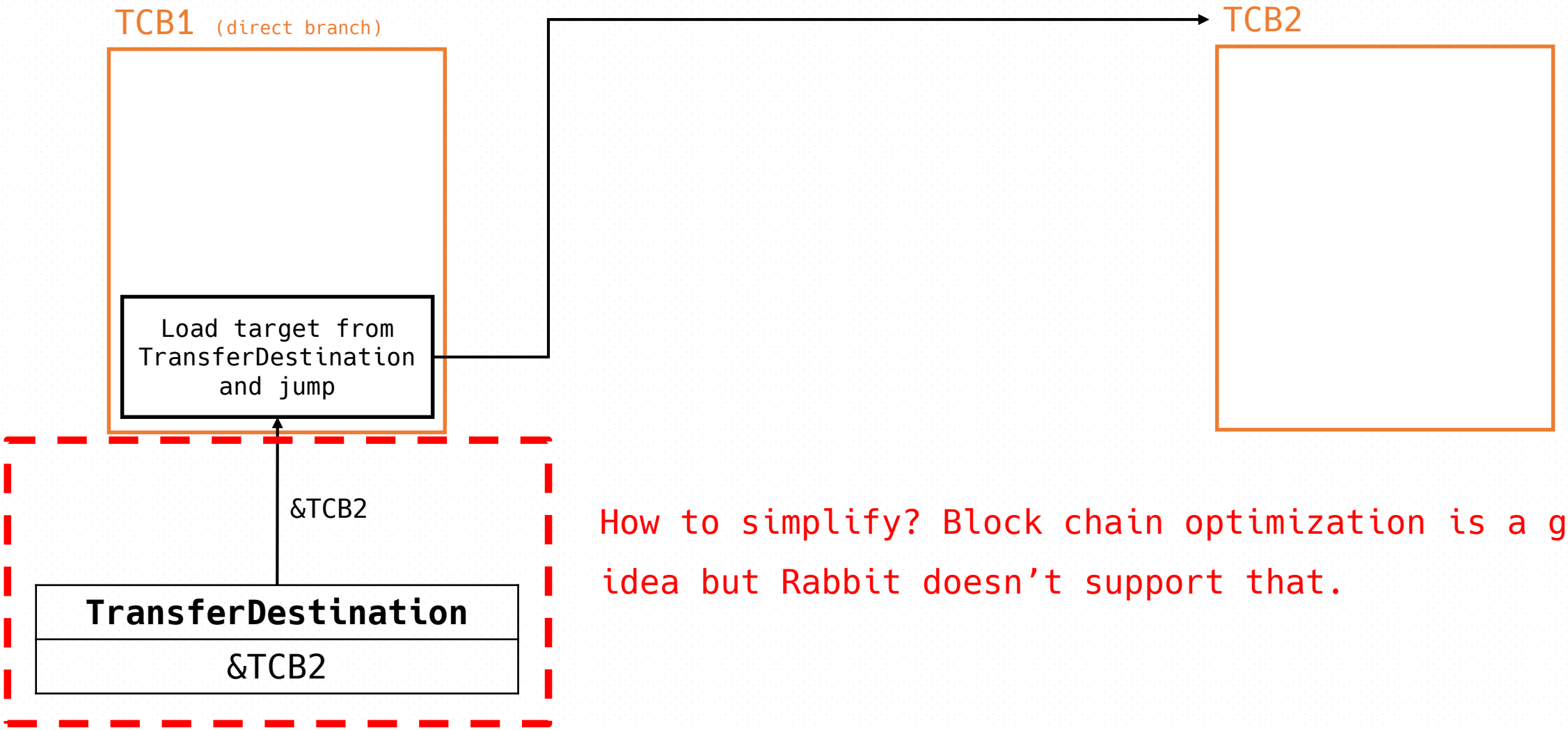

TCB2(successor block)'s SPC and
&JumpDestinationCache
indirect-branch chaining handler
TCB1 (indirect-branch)
lookup SPC &
insert entry
if lookup fails
Call indirect-branch chaining handler
jump
TCB1's Jump Destination Cache
SPC
TPC
TCB2's SPC
TCB2's TPC
…
…


indirect-branch chaining handler
TCB1 (indirect-branch)
TCB2's TPC
TCB2's TPC
Call indirect-branch chaining handler
TCB2's TPC
jump
TCB1's Jump Destination Cache
SPC
TPC
TCB2's SPC
TCB2's TPC
…
…

Dep-Hyperchaining For Direct Branches

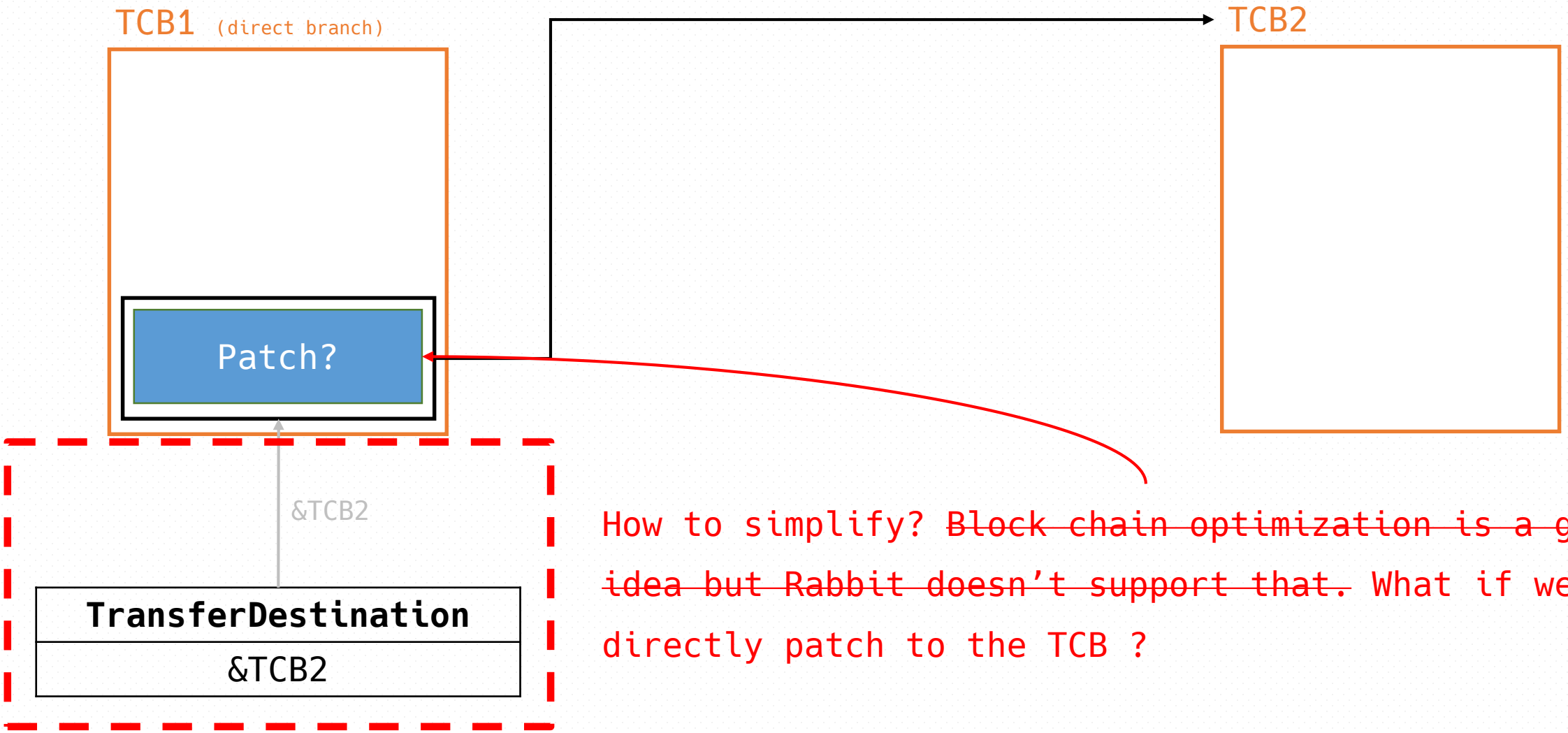
- The reasons why we develop **dep-hyperchaining** (**platform-dependent hyperchaining**)
  - We observe some redundant operations in Indep-hyperchaining
    - For direct branches, TCBs need to read the same TPC of successor's block every time the block executes
    - For indirect branches, one possible case is single destination indirect branch which makes jump destination cache inefficient
  - The design of **dep-hyperchaining** is based on the **indep-hyperchaining**

Indep-hyperchaining for direct branches



How to simplify? Block chain optimization is a good idea but Rabbit doesn't support that.

Indep-hyperchaining for direct branches



How to simplify? ~~Block chain optimization is a good idea but Rabbit doesn't support that.~~ What if we directly patch to the TCB ?

Challenges & Solutions

- Where is the instructions of load memory operations in TCBs
  - We need a anchor point to tell the run-time library the address of that instructions in TCBs
- How to patch the code. Is it practicable ?
  - We need to analyze possible generated code patterns

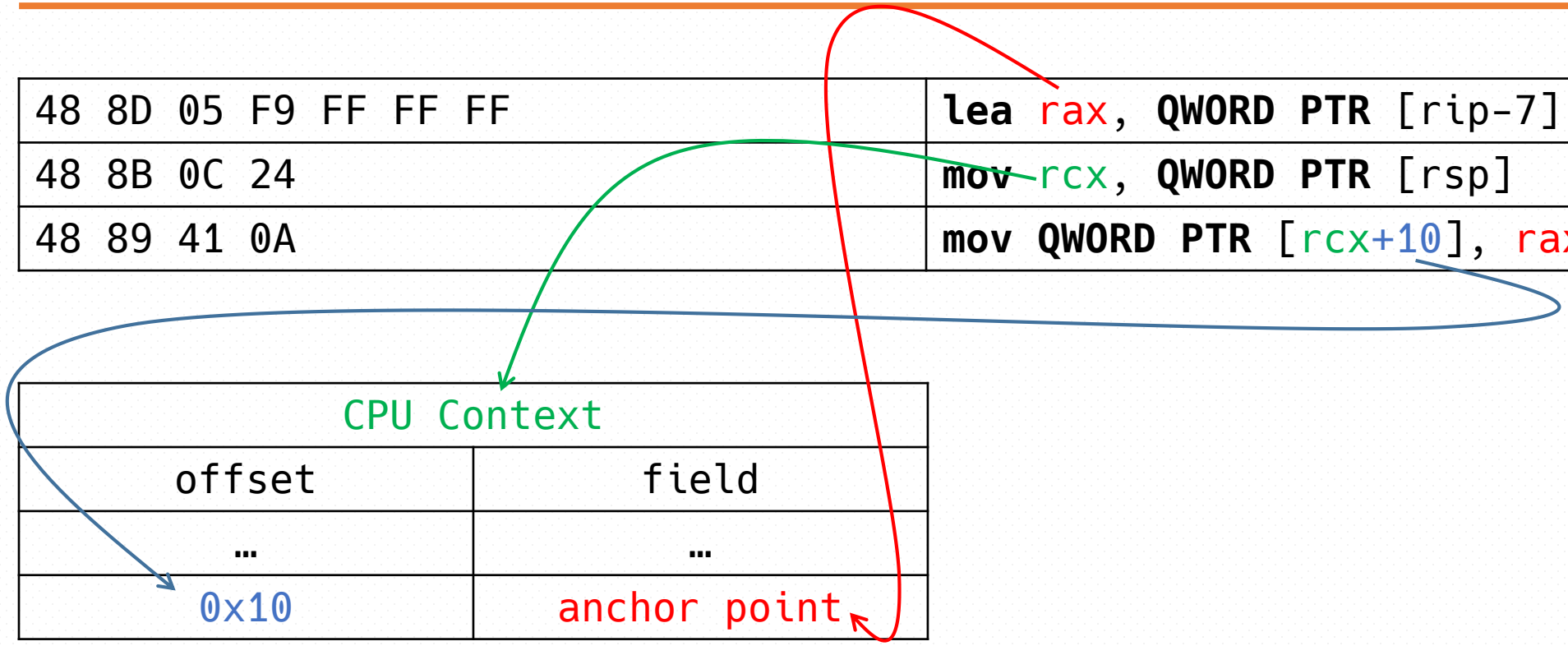
Load Effective Address

- LEA
  - **lea** permits the program counter as its source operand
  - rip minus the offset 7 (i.e. the length of **lea**) to get the address of lea instruction itself
- Store the address into the globally accessible **CPU context**

REX.W 8D ModRM F9 FF FF FF	**lea** r64, **QWORD PTR** [rip - 0x7]

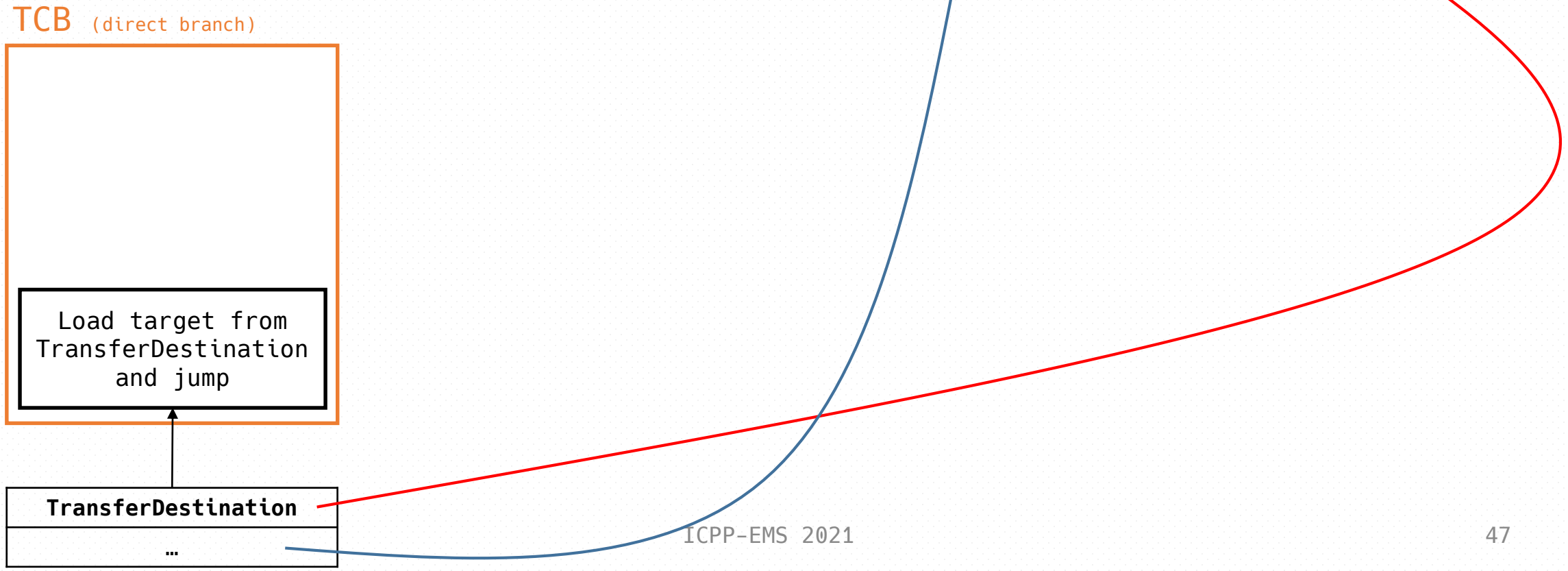
Store Address Into CPU Context

48 8D 05 F9 FF FF FF	**lea** rax, **QWORD PTR** [rip-7]
48 8B 0C 24	**mov** rcx, **QWORD PTR** [rsp]
48 89 41 0A	**mov QWORD PTR** [rcx+10], rax



Load TransferDestination's Address

48 B8 08 60 EA F7 FF 7F 00 00	**movabs** rax, 0x7ffff7ea6008
48 8B 00	**mov** rax, **QWORD PTR** [rax]



Jump To Target

48 89 CF	**mov** rdi, rcx
FF E0	**jmp** rax

Every block contains only CPU context as a parameter to make code block be able to read and write simulated registers through CPU context.

Code Pattern 1

- The aforementioned instruction sequences constitute code pattern 1 in our definition

48 8D 05 F9 FF FF FF	**lea** rax, **QWORD PTR** [rip-7]
48 8B 0C 24	**mov** rcx, **QWORD PTR** [rsp]
48 89 41 0A	**mov QWORD PTR** [rcx+10], rax
48 B8 08 60 EA F7 FF 7F 00 00	**movabs** rax, 0x7ffff7ea6008
48 8B 00	**mov** rax, **QWORD PTR** [rax]
48 89 CF	**mov** rdi, rcx
FF E0	**jmp** rax

Then, How To Patch

- The intuitive way is patching the last instruction “**jmp** r64”
- Unfortunately, it is impracticable
  - Patch out-of-bound
  - Conditional branch block

Jump Patch

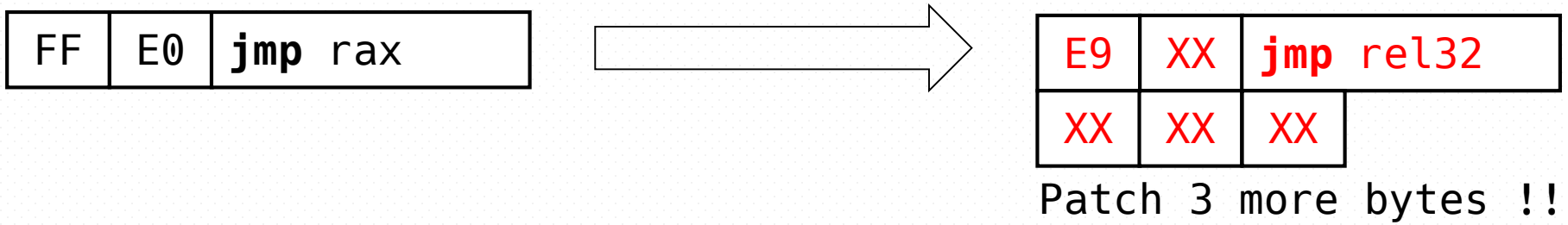
- A minimal jump patch is a direct-jump patch
  - “JMP rel32” takes 5 bytes
  - “JMP rel8” is useless here because tiny displacement is rare to appear. We don’t consider such case

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
EB *cb*	JMP *rel8*	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits
E9 *cw*	JMP *rel16*	D	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 *cd*	JMP *rel32*	D	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits

From Intel® 64 and IA-32 Architectures Software Developer’s Manual

Patch Out-of-bound

- A minimal patch, “**jmp** rel32”, takes 5 bytes
- However, last instruction “**jmp** r64” only provides 2 bytes space to patch



Conditional Branch Blocks

- “taken” and “not taken” blocks are contiguous in the run-time memory
  - There are no reserved bytes between two TCBs
  - Patch out-of-bound problem implies “not taken” block get overwritten at the first few bytes and becomes corrupt

Patch

- For such concerns, we don’t patch last instruction
- We pick “**movabs** rax, 0x7ffff7ea6008” as a proper instruction to patch
  - **movabs** offers 10 bytes space that is quite enough

Patched TCB for code pattern 1

Before

48 8D 05 F9 FF FF FF	**lea** rax, **QWORD PTR** [rip-7]
48 8B 0C 24	**mov** rcx, **QWORD PTR** [rsp]
48 89 41 0A	**mov QWORD PTR** [rcx+10], rax
48 B8 08 60 EA F7 FF 7F 00 00	**movabs** rax, 0x7ffff7ea6008
48 8B 00	**mov** rax, **QWORD PTR** [rax]
48 89 CF	**mov** rdi, rcx
FF E0	**jmp** rax

After

48 8D 05 F9 FF FF FF	**lea** rax, **QWORD PTR** [rip-7]
48 8B 0C 24	**mov** rcx, **QWORD PTR** [rsp]
48 89 41 0A	**mov QWORD PTR** [rcx+10], rax
48 89 CF	**mov** rdi, rcx
E9 XX XX XX XX	**jmp** <TPC> (successor block) →
00 00	??? (don't care; unreachable)
48 8B 00	**mov** rax, **QWORD PTR** [rax]
48 89 CF	**mov** rdi, rcx
FF E0	**jmp** rax

Indirect-Branch Patch

- If the displacement is too large to be encoded as a direct-jump patch, we use **indirect-branch patch**

movabs rax, <TPC>
jmp rax

Patched TCB For Code Pattern 1

Before

48 8D 05 F9 FF FF FF	**lea** rax, **QWORD PTR** [rip-7]
48 8B 0C 24	**mov** rcx, **QWORD PTR** [rsp]
48 89 41 0A	**mov QWORD PTR** [rcx+10], rax
48 B8 08 60 EA F7 FF 7F 00 00	**movabs** rax, 0x7ffff7ea6008
48 8B 00	**mov** rax, **QWORD PTR** [rax]
48 89 CF	**mov** rdi, rcx
FF E0	**jmp** rax

After

48 8D 05 F9 FF FF FF	**lea** rax, **QWORD PTR** [rip-7]
48 8B 0C 24	**mov** rcx, **QWORD PTR** [rsp]
48 89 41 0A	**mov QWORD PTR** [rcx+10], rax
48 B8 XX XX XX XX XX XX XX XX	**movabs** rax, <TPC> (replaced by successor block)
0F 1F 00	**nop DWORD PTR** [rax](disable dereference)
48 89 CF	**mov** rdi, rcx
FF E0	**jmp** rax

Code Pattern 2

- Similar to code pattern 1 but has less instructions
- **rdi** still holds &**CPUContext**

48 8D 05 F9 FF FF FF	**lea** rax, **QWORD PTR** [rip-7]
48 89 41 0A	**mov QWORD PTR** [rcx+10], rax
48 B8 08 60 EA F7 FF 7F 00 00	**movabs** rax, 0x7ffff7ea6008
48 8B 00	**mov** rax, **QWORD PTR** [rax]
FF E0	**jmp** rax

Patched TCB For Code Pattern 2

Before

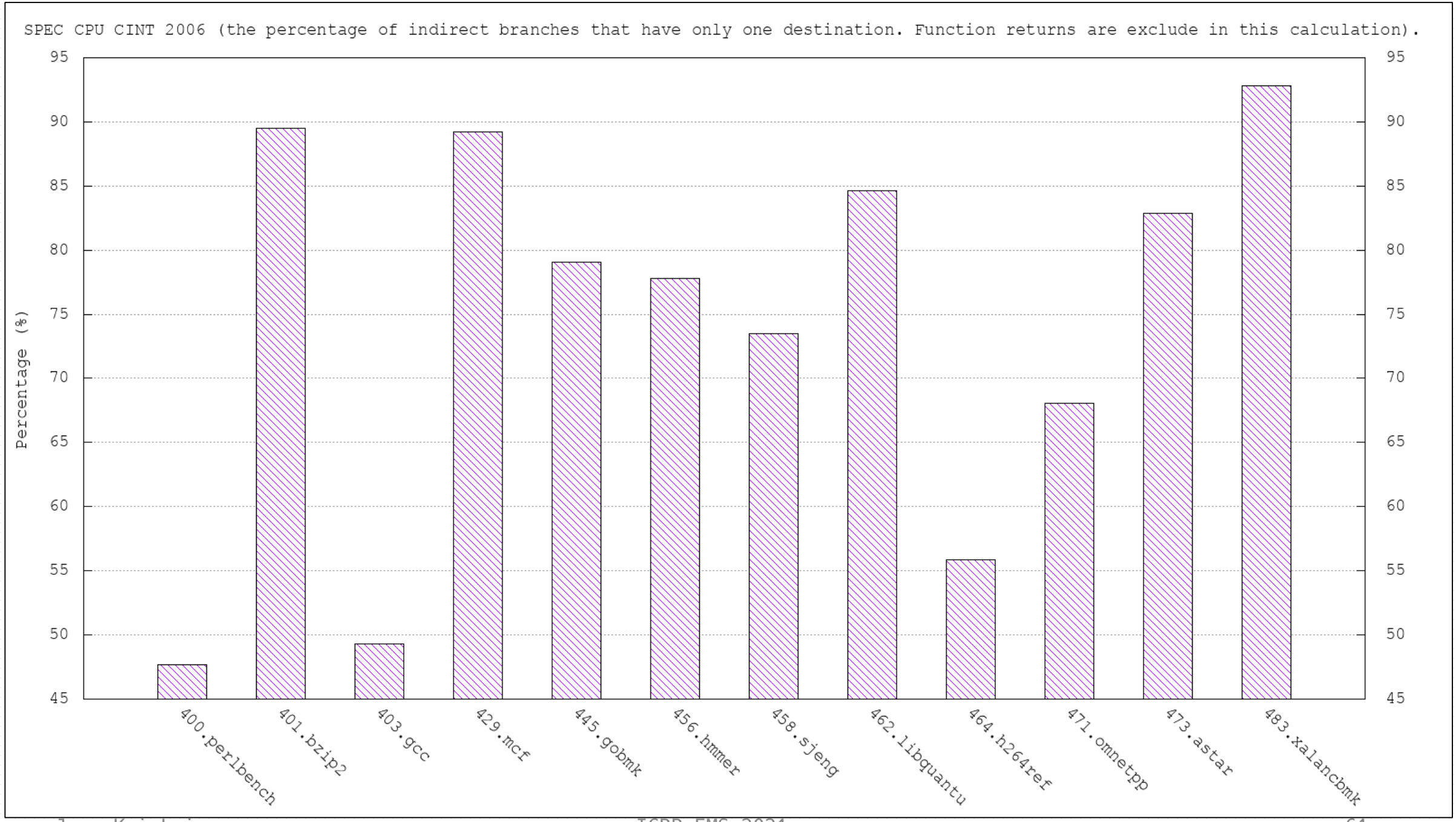
48 8D 05 F9 FF FF FF	**lea** rax, **QWORD PTR** [rip-7]
48 89 41 0A	**mov QWORD PTR** [rcx+10], rax
48 B8 08 60 EA F7 FF 7F 00 00	**movabs** rax, 0x7ffff7ea6008
48 8B 00	**mov** rax, **QWORD PTR** [rax]
FF E0	**jmp** rax

After

E9 XX XX XX XX	**jmp** <TPC> (successor block) →
FF FF	??
48 89 41 0A	**mov QWORD PTR** [rcx+10], rax
48 B8 08 60 EA F7 FF 7F 00 00	**movabs** rax, 0x7ffff7ea6008
48 8B 00	**mov** rax, **QWORD PTR** [rax]
FF E0	**jmp** rax

Dep-Hyperchaining For Indirect Branches

- According to our study of SPEC CPU 2006 CINT benchmarks
  - 56% (avg.) of the indirect branches in the source binary are function-return instructions
  - 72.88% (avg.) of the indirect branches which are not function return have only a single jump destination
    - jump-destination cache is unnecessary in single destination case


SPEC CPU CINT 2006 (the percentage of indirect branches that have only one destination. Function returns are exclude in this calculation).
Percentage (%)
95
90
85
80
75
70
65
60
55
50
45
400.perlbench
401.bzip2
403.gcc
429.mcf
445.gobmk
456.hmmer
458.sjeng
462.libquantu
464.h264ref
471.omnetpp
473.astar
483.xalancbmk

Design

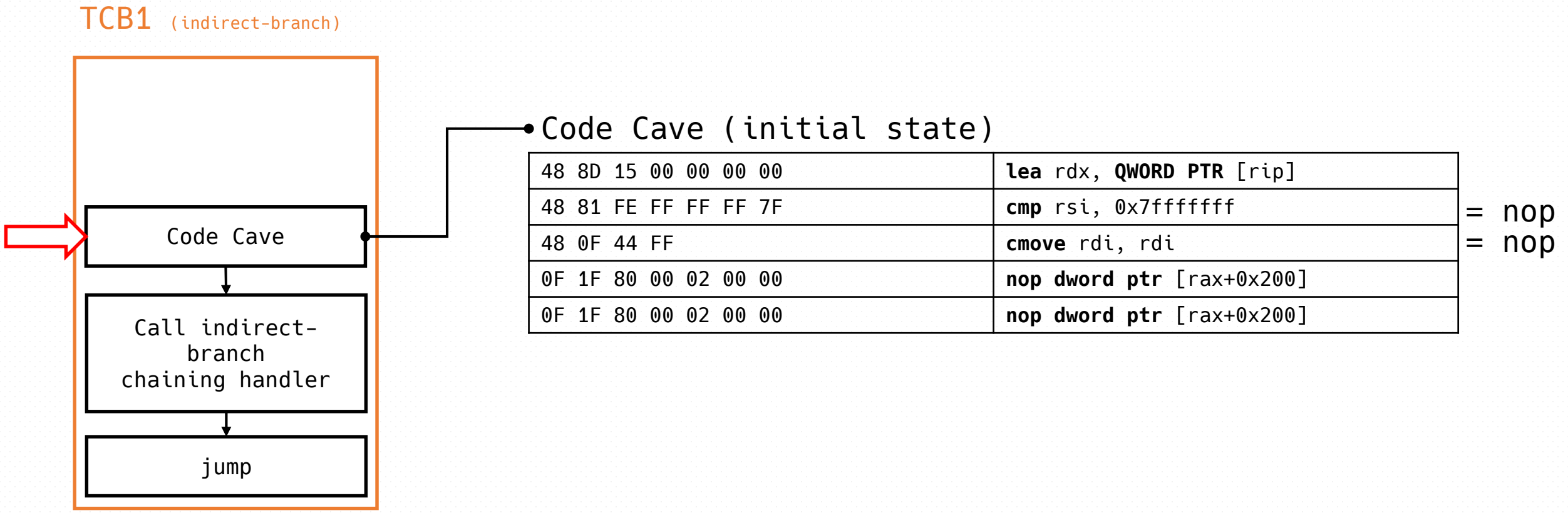
- The code design need to consider situations for both single destination and multiple destinations
- Change the code behavior dynamically
  - Transform single destination → multiple destinations by patching code

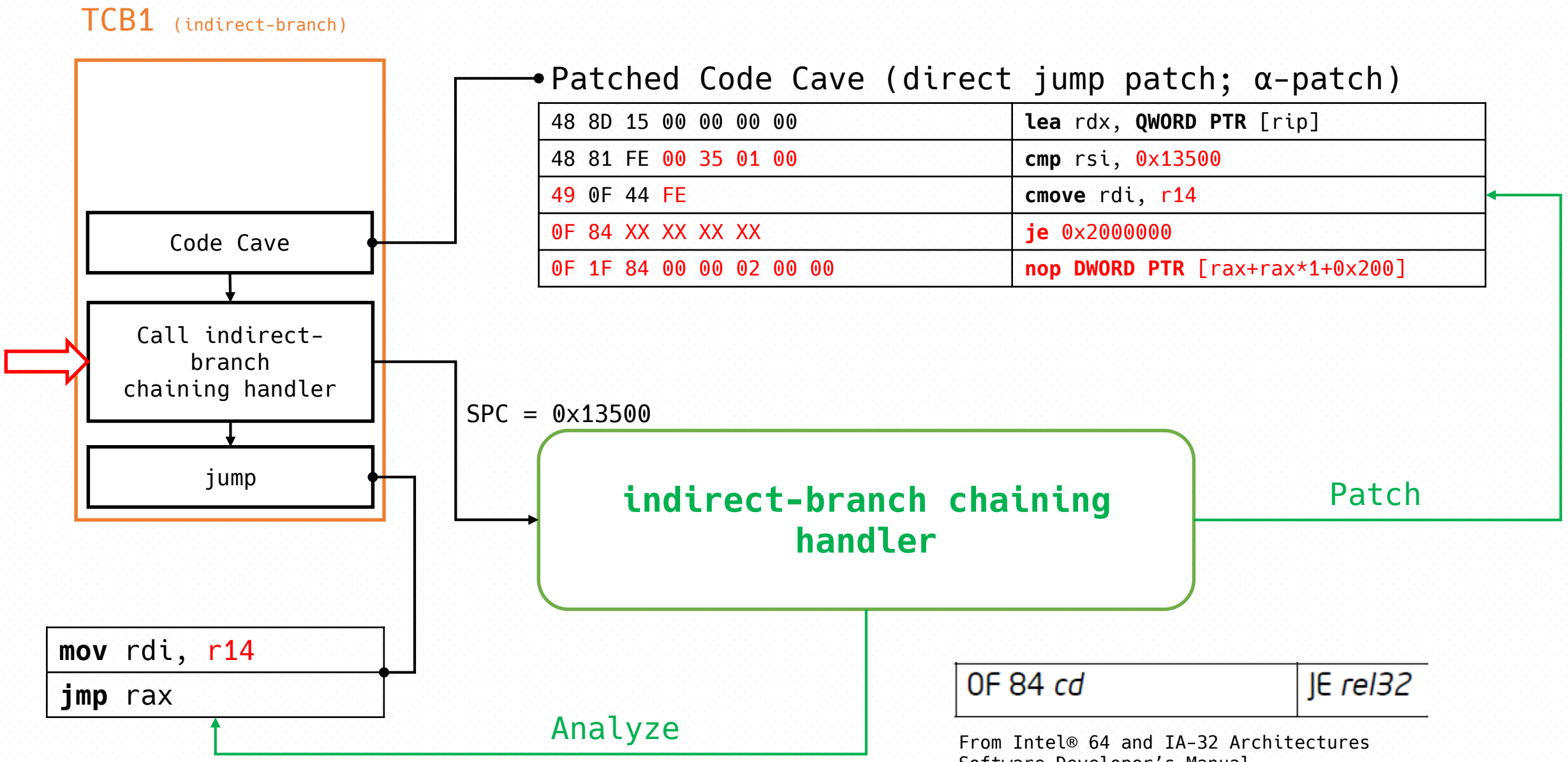
Solutions

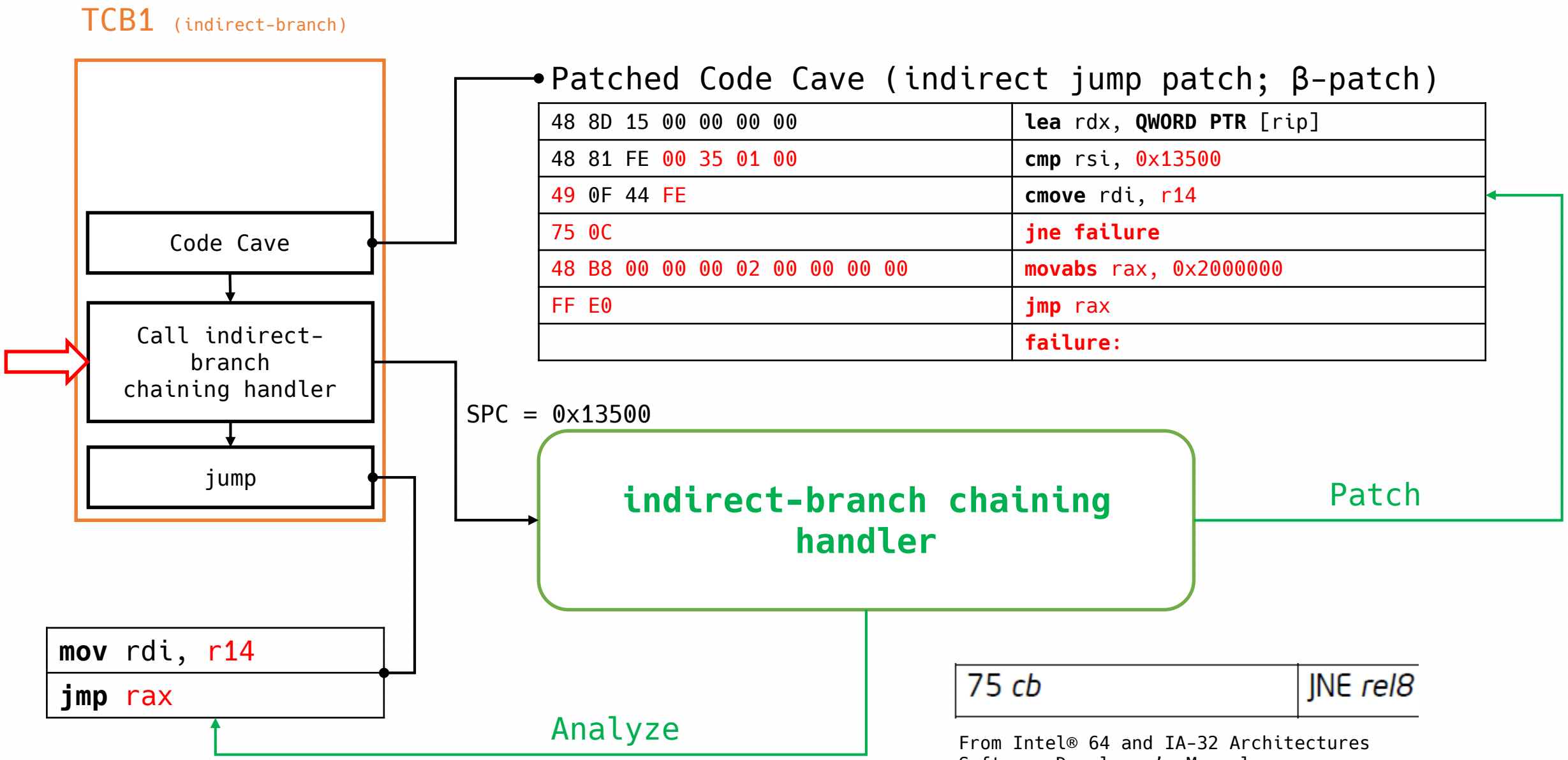
- The code design need to consider situations for both single destination and multiple destinations
  - Single destination : **Code Cave**

Code Cave

- A **code cave** is one or several consecutive instructions in which some fields are intentionally filled with dummy values in the beginning
- Later these fields are patched with data related to the indirect branch, such as SPC, TPC, etc.

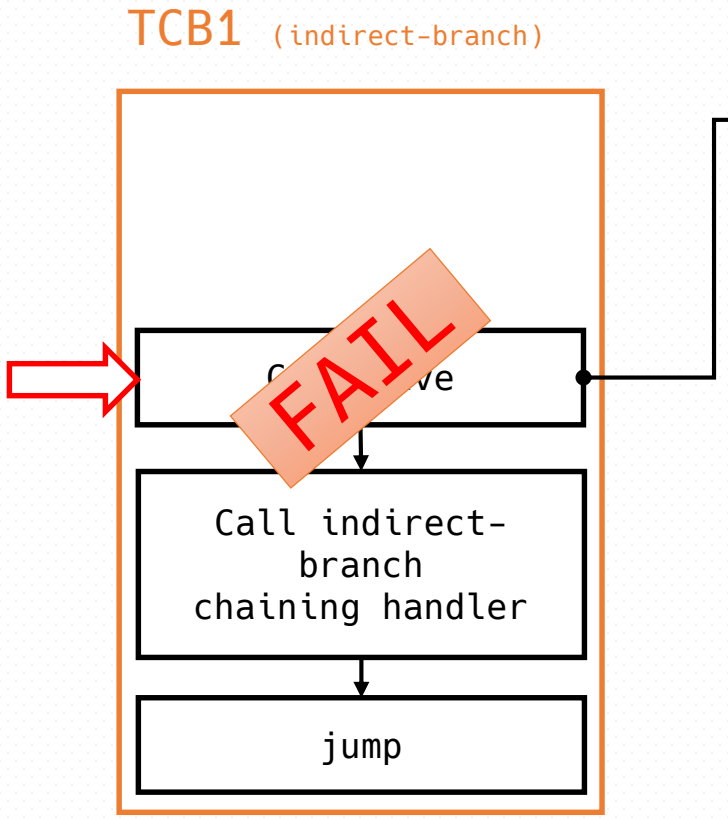

TCB1 (indirect-branch)
Code Cave
Call indirect-branch chaining handler
jump
Code Cave (initial state)
48 8D 15 00 00 00 00
lea rdx, QWORD PTR [rip]
48 81 FE FF FF FF 7F
cmp rsi, 0x7fffffff
= nop
48 0F 44 FF
cmove rdi, rdi
= nop
0F 1F 80 00 02 00 00
nop dword ptr [rax+0x200]
0F 1F 80 00 02 00 00
nop dword ptr [rax+0x200]


TCB1 (indirect-branch)
Code Cave
Call indirect-branch chaining handler
jump
Patched Code Cave (direct jump patch; α-patch)
48 8D 15 00 00 00 00 | lea rdx, QWORD PTR [rip]
48 81 FE 00 35 01 00 | cmp rsi, 0x13500
49 0F 44 FE | cmove rdi, r14
0F 84 XX XX XX XX | je 0x2000000
0F 1F 84 00 00 02 00 00 | nop DWORD PTR [rax+rax*1+0x200]
SPC = 0x13500
indirect-branch chaining handler
Patch
mov rdi, r14
jmp rax
Analyze
0F 84 cd | JE rel32
From Intel® 64 and IA-32 Architectures Software Developer's Manual


TCB1 (indirect-branch)
Code Cave
Call indirect-branch chaining handler
jump
Patched Code Cave (indirect jump patch; β-patch)
48 8D 15 00 00 00 00 | lea rdx, QWORD PTR [rip]
48 81 FE 00 35 01 00 | cmp rsi, 0x13500
49 0F 44 FE | cmove rdi, r14
75 0C | jne failure
48 B8 00 00 00 02 00 00 00 00 | movabs rax, 0x2000000
FF E0 | jmp rax
| failure:
SPC = 0x13500
indirect-branch chaining handler
Patch
mov rdi, r14
jmp rax
Analyze
75 cb | JNE rel8
From Intel® 64 and IA-32 Architectures Software Developer's Manual

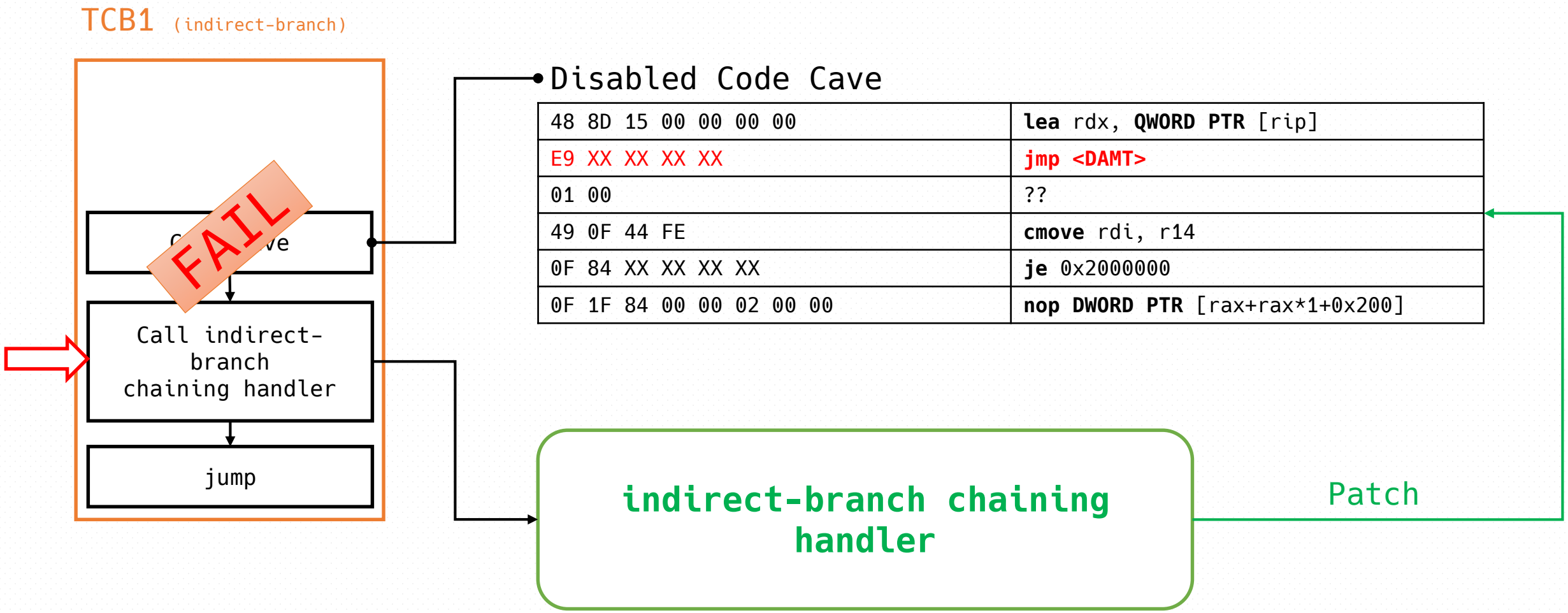
Solutions

- The code design need to consider situations for both single destination and multiple destinations
  - Single destination : **Code Cave**
  - Multiple destinations : **Dynamic Address Mapping Table**
- Change the code behavior dynamically
  - Repatch the code cave

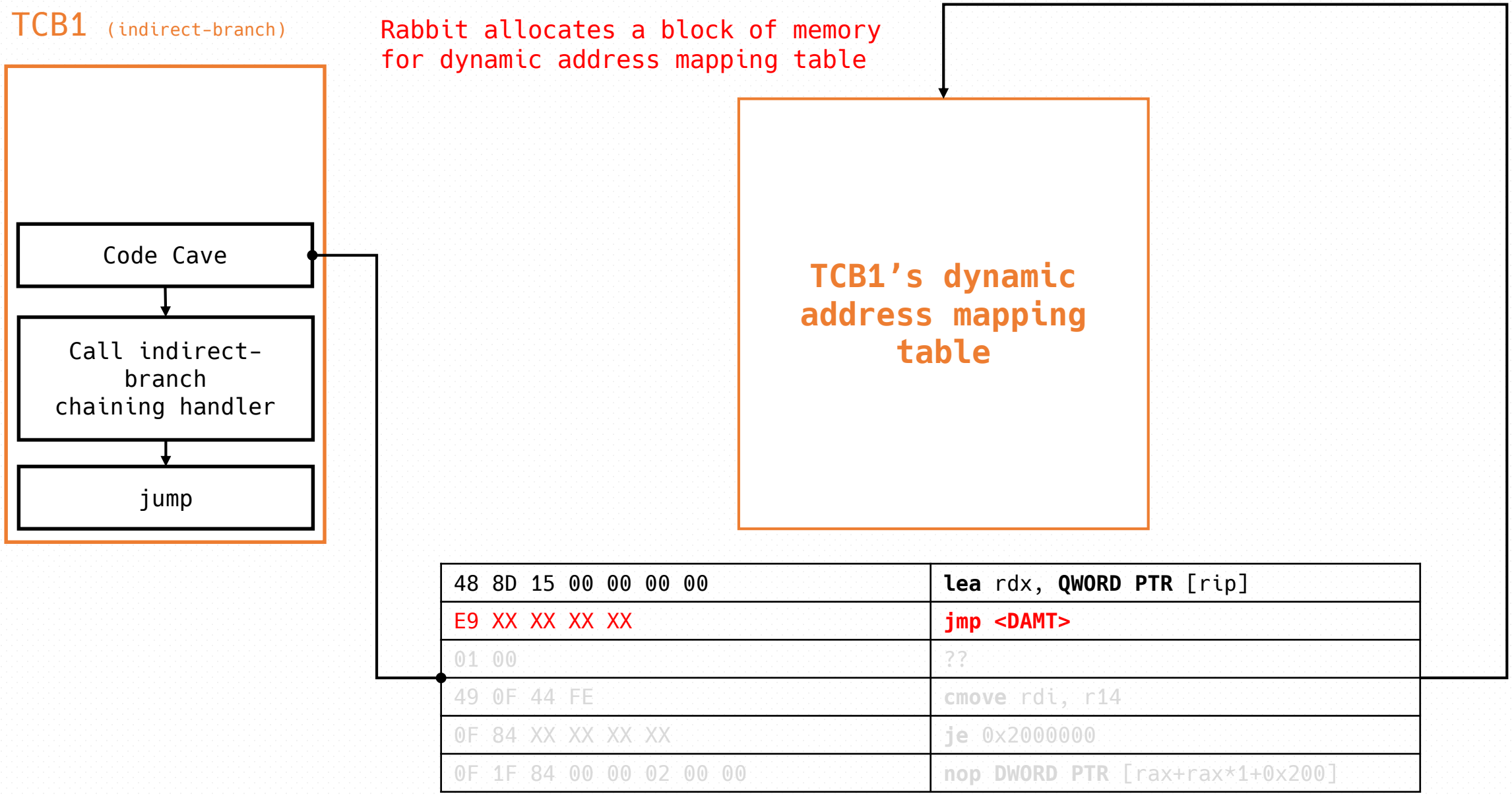


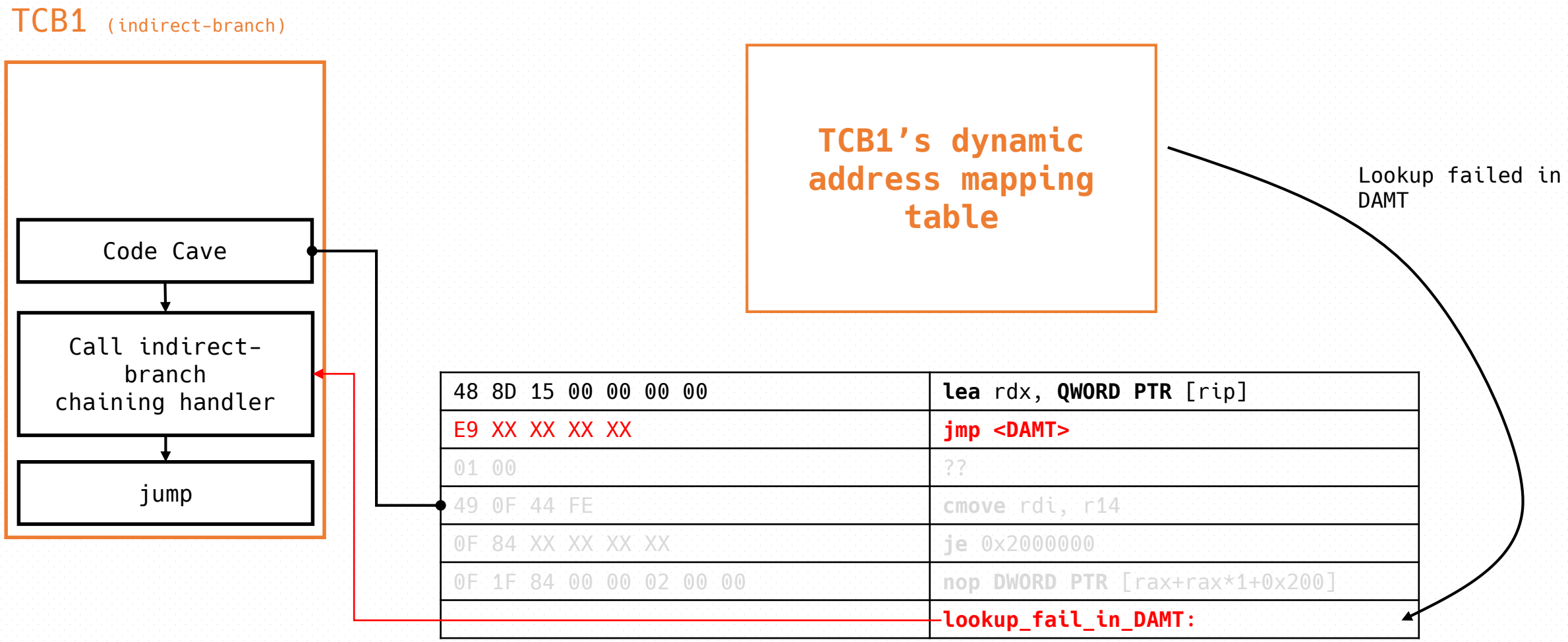
Patched Code Cave (direct jump patch; α-patch)

48 8D 15 00 00 00 00	**lea** rdx, **QWORD PTR** [rip]
48 81 FE 00 35 01 00	**cmp** rsi, 0x13500
49 0F 44 FE	**cmove** rdi, r14
0F 84 XX XX XX XX	**je** 0x2000000
0F 1F 84 00 00 02 00 00	**nop DWORD PTR** [rax+rax*1+0x200]



We find that this indirect branch has more than one destination


TCB1 (indirect-branch)
Code Cave
Call indirect-branch chaining handler
jump
Rabbit allocates a block of memory for dynamic address mapping table
TCB1's dynamic address mapping table
48 8D 15 00 00 00 00
lea rdx, QWORD PTR [rip]
E9 XX XX XX XX
jmp <DAMT>
01 00
??
49 0F 44 FE
cmove rdi, r14
0F 84 XX XX XX XX
je 0x2000000
0F 1F 84 00 00 02 00 00
nop DWORD PTR [rax+rax*1+0x200]


TCB1 (indirect-branch)
Code Cave
Call indirect-branch chaining handler
jump
TCB1's dynamic address mapping table
Lookup failed in DAMT
48 8D 15 00 00 00 00
lea rdx, QWORD PTR [rip]
E9 XX XX XX XX
jmp <DAMT>
01 00
??
49 0F 44 FE
cmove rdi, r14
0F 84 XX XX XX XX
je 0x2000000
0F 1F 84 00 00 02 00 00
nop DWORD PTR [rax+rax*1+0x200]
lookup_fail_in_DAMT:

Dynamic Address Mapping Table

```
DAMT:
cmp rsi, 0x10000
je  tag1
jmp <code cave>
```

```
tag1:
mov rdi, r14
movabs rax, 0x7ffff0000000
jmp rax
```

Visited Destination	
SPC	TPC
0x10000	0x7ffff0000000

Dynamic Address Mapping Table

```
DAMT:
cmp rsi, 0x10000
je  tag1
cmp rsi, 0x11000
je  tag2
jmp <code cave>
```

```
tag1:
mov rdi, r14
movabs rax, 0x7ffff0000000
jmp rax
tag2:
mov rdi, r14
movabs rax, 0x7ffff1000000
jmp rax
```

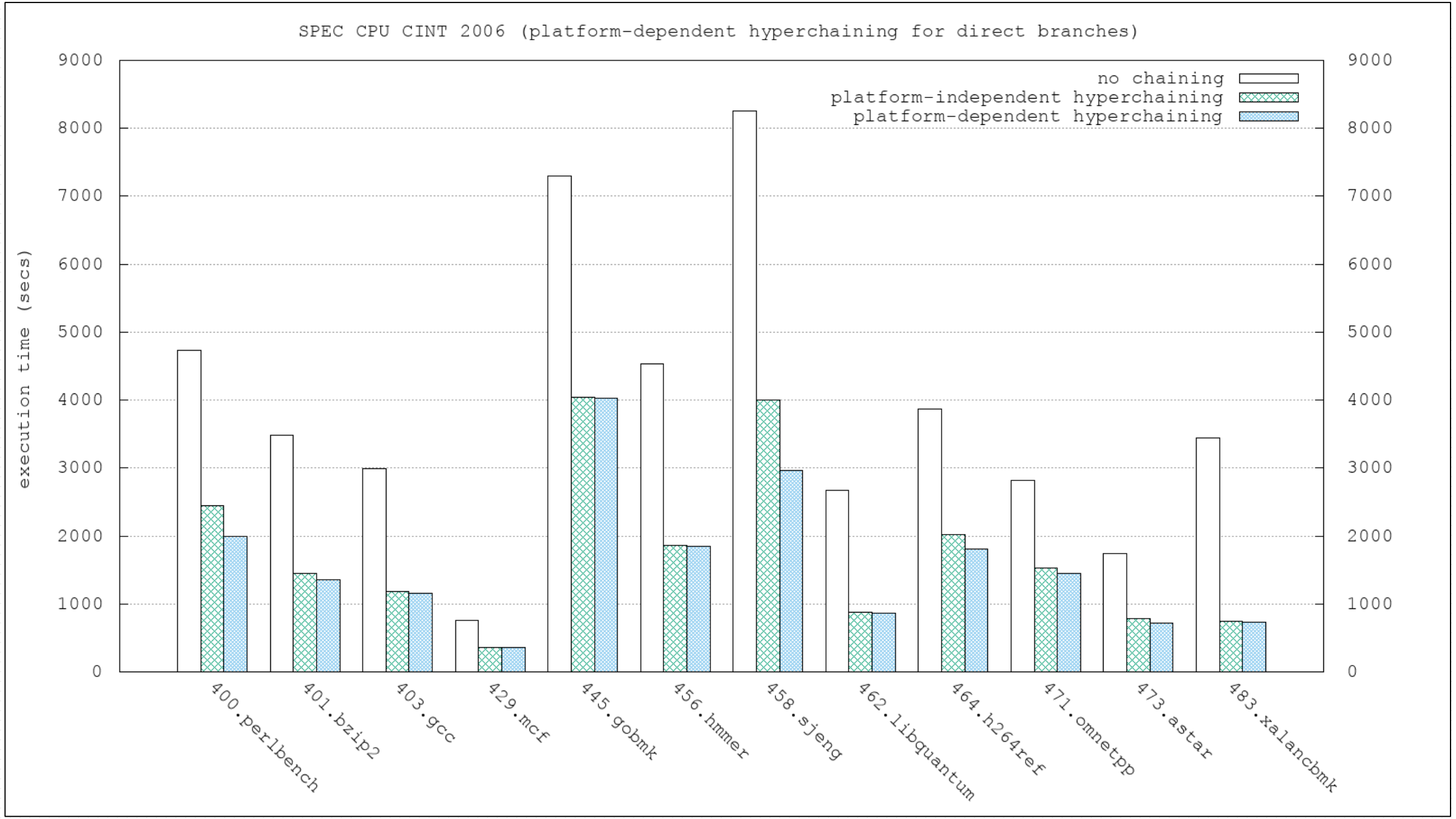
Visited Destination	
SPC	TPC
0x10000	0x7ffff0000000
0x11000	0x7ffff1000000

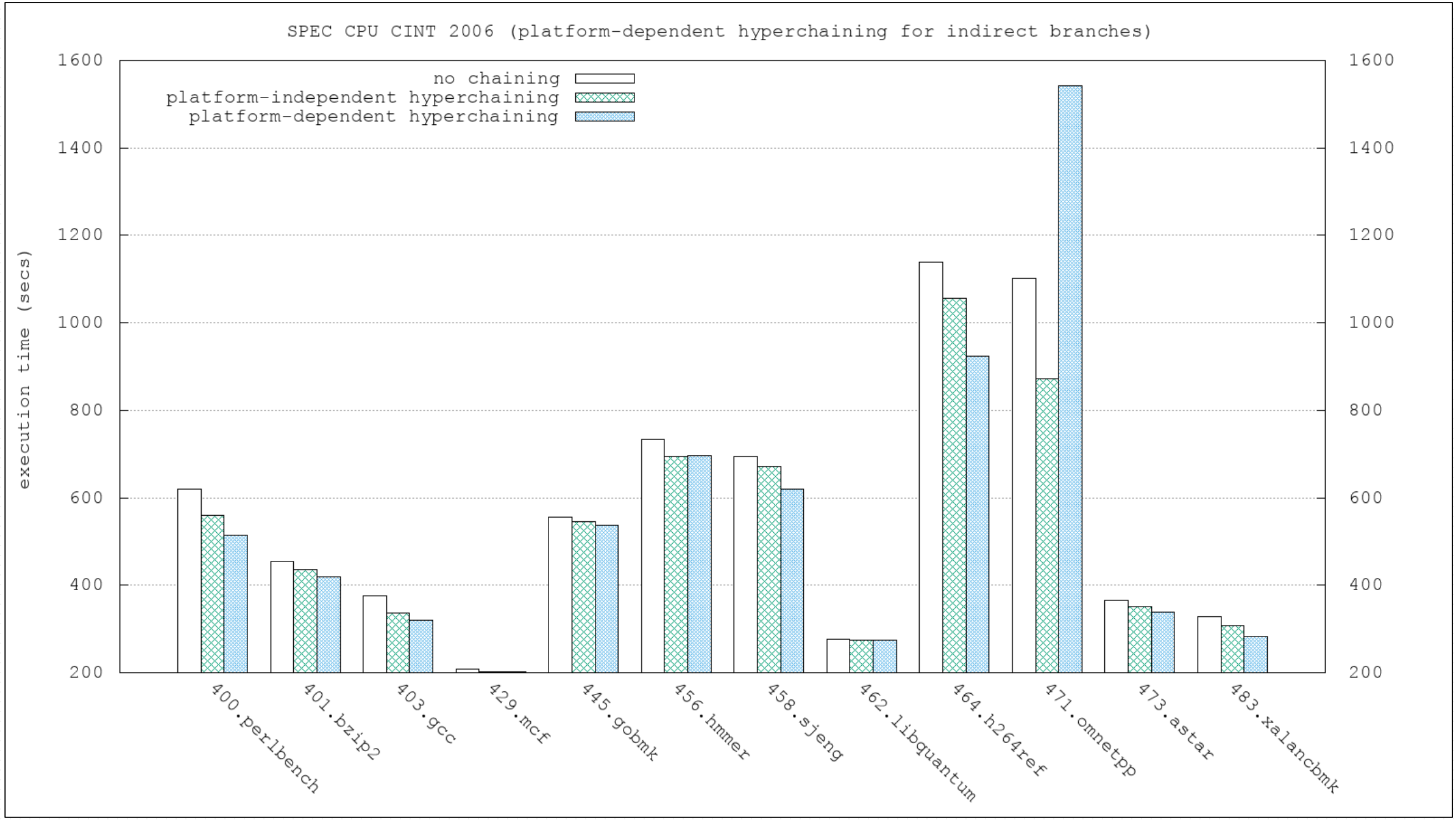
Outline

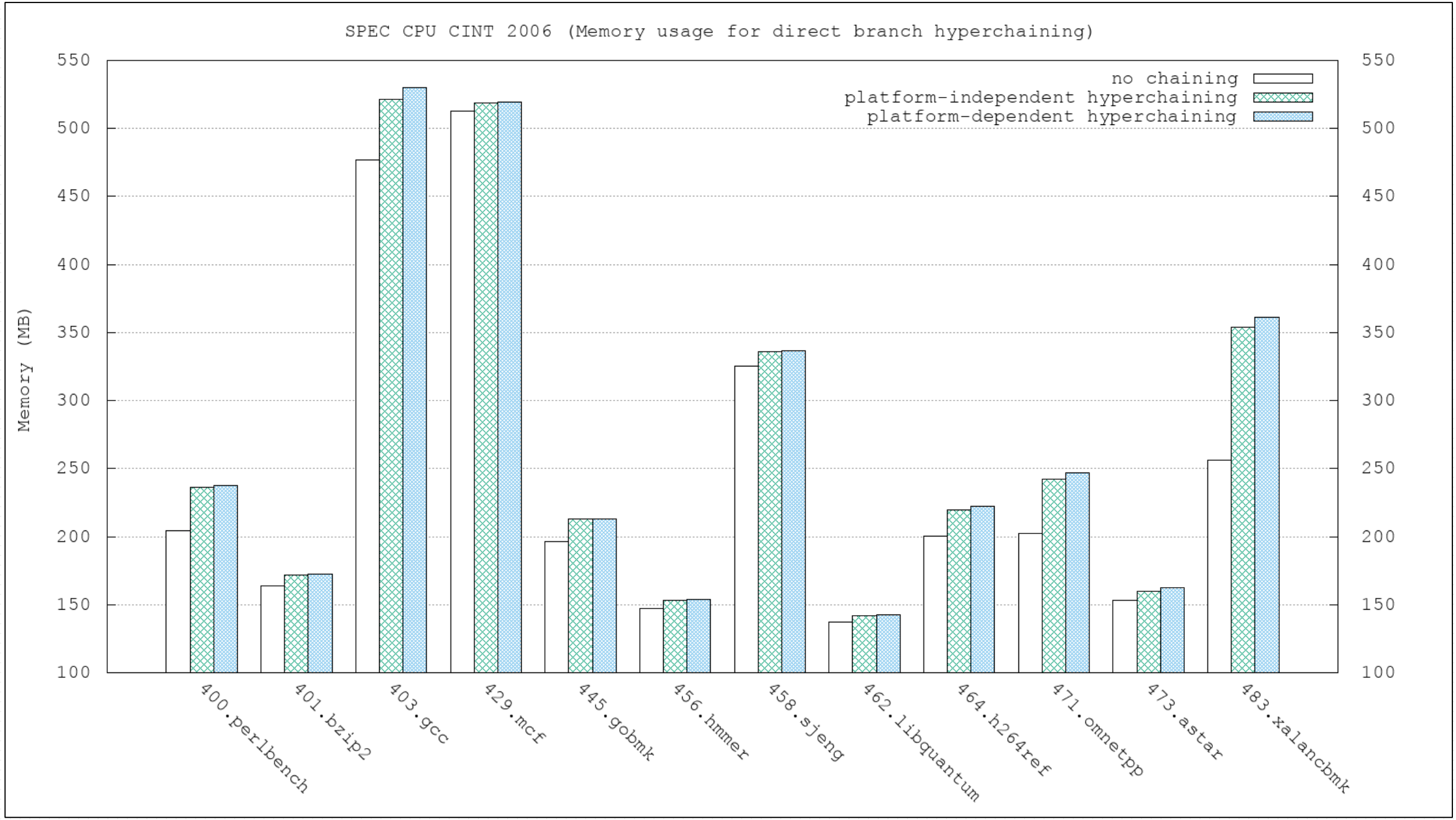
- Background
- Hyperchaining
- **Experiments**
- Conclusion

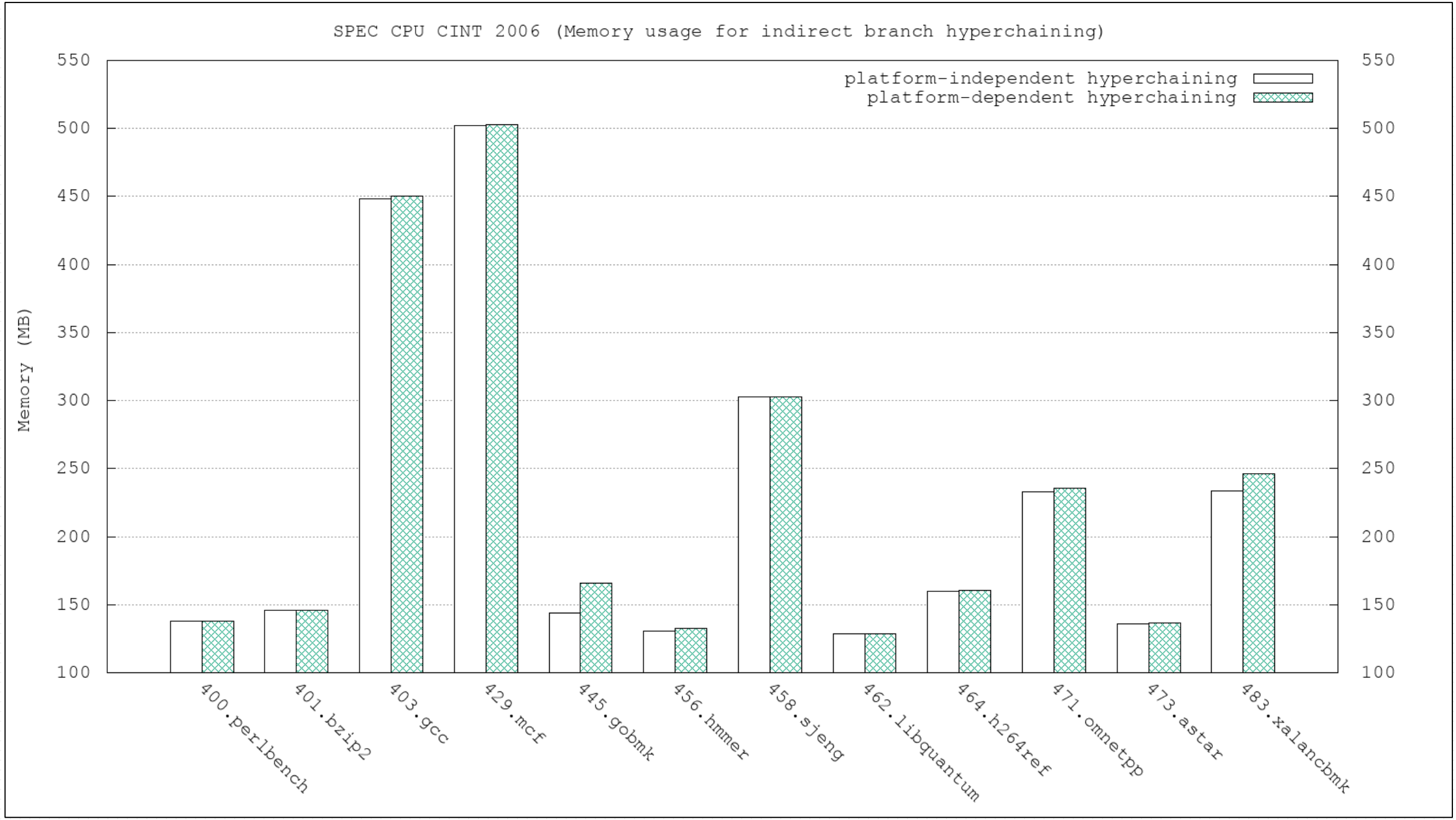
Experimental Setup

- CPU : Intel Core i7-7700 3.60GHz
- OS : Ubuntu 18.04.3 LTS with Linux kernel 4.15.0
- RAM : 16GB
- LLVM : 8.0.1
- Benchmark : SPEC CPU 2006 CINT


SPEC CPU CINT 2006 (platform-dependent hyperchaining for direct branches)
execution time (secs)
no chaining
platform-independent hyperchaining
platform-dependent hyperchaining
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
400.perlbench
401.bzip2
403.gcc
429.mcf
445.gobmk
456.hmmer
458.sjeng
462.libquantum
464.h264ref
471.omnetpp
473.astar
483.xalancbmk


SPEC CPU CINT 2006 (platform-dependent hyperchaining for indirect branches)
no chaining
platform-independent hyperchaining
platform-dependent hyperchaining
execution time (secs)
1600
1400
1200
1000
800
600
400
200
400.perlbench
401.bzip2
403.gcc
429.mcf
445.gobmk
456.hmmer
458.sjeng
462.libquantum
464.h264ref
471.omnetpp
473.astar
483.xalancbmk


SPEC CPU CINT 2006 (Memory usage for direct branch hyperchaining)
Memory (MB)
550
500
450
400
350
300
250
200
150
100
no chaining
platform-independent hyperchaining
platform-dependent hyperchaining
400.perlbench
401.bzip2
403.gcc
429.mcf
445.gobmk
456.hmmer
458.sjeng
462.libquantum
464.h264ref
471.omnetpp
473.astar
483.xalancbmk


SPEC CPU CINT 2006 (Memory usage for indirect branch hyperchaining)
platform-independent hyperchaining
platform-dependent hyperchaining
Memory (MB)
550
500
450
400
350
300
250
200
150
100
400.perlbench
401.bzip2
403.gcc
429.mcf
445.gobmk
456.hmmer
458.sjeng
462.libquantum
464.h264ref
471.omnetpp
473.astar
483.xalancbmk

Outline

- Background
- Hyperchaining
- Experiments
- **Conclusion**

Conclusion

- This paper proposes platform-independent hyperchaining and platformdependent hyperchaining on the x86-64 platform
- The experimental results show that platform-dependent hyperchaining can reach 1.08x and 1.05x speedup compared to platform-independent hyperchaining

Conclusion

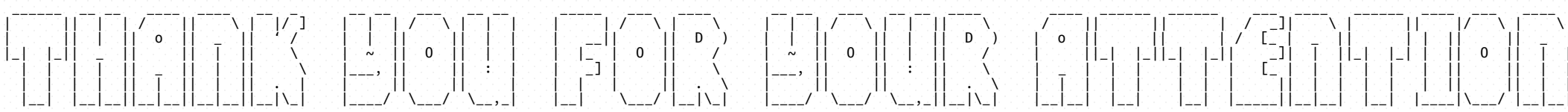
- Memory usage of platform-dependent hyperchaining is 1.0089x and 1.019x times that of platform-independent hyperchaining for direct branches and indirect branches, respectively
- It has small memory overhead to adopt platform-dependent approach

Future Work

- The approach of platform-dependent hyperchaining for indirect branches implies potential opportunities to be improved on performance
  - Schedule the first section of the SPC section to reduce and mitigate the miss penalty in run-time
  - Change the mechanism of searching in DAMT (Hash table ?)
  - Lack of a policy to disable platform-dependent hyperchaining when the dynamic address mapping table is full

Credits

- The hack font - authors of Source Foundry, which is licensed under MIT https://github.com/source-foundry/Hack


THANK YOU FOR ATTENTION