Processor-Tracing Guided Region Formation in Dynamic Binary Translation

Ding-Yong Hong, Jan-Jan Wu, Yu-Ping Liu, Sheng-Yu Fu, and Wei-Chung Hsu

Academia Sinica
National Taiwan University

Outline

- Background and Motivation
  - Region formation
  - Processor tracing
- Design
  - Region formation guided by processor tracing
- Evaluation Results
- Conclusion

Motivation

- Region formation is an important step in dynamic binary translation (DBT)
  - Find hot execution codes for translation/optimization
- Two reasons to select hot regions
  - Avoid compiling cold code so as to reduce system overhead
  - Gain the most benefits from the optimization

Motivation

- Region formation has two steps:
  - Profiling code segments to verify the “hotness” of regions
  - Selecting code segments to form regions

- A dual issue in designing region formation algorithm
  - How to effectively detect hot regions and remain low overhead?

Region Formation with Instrumentation

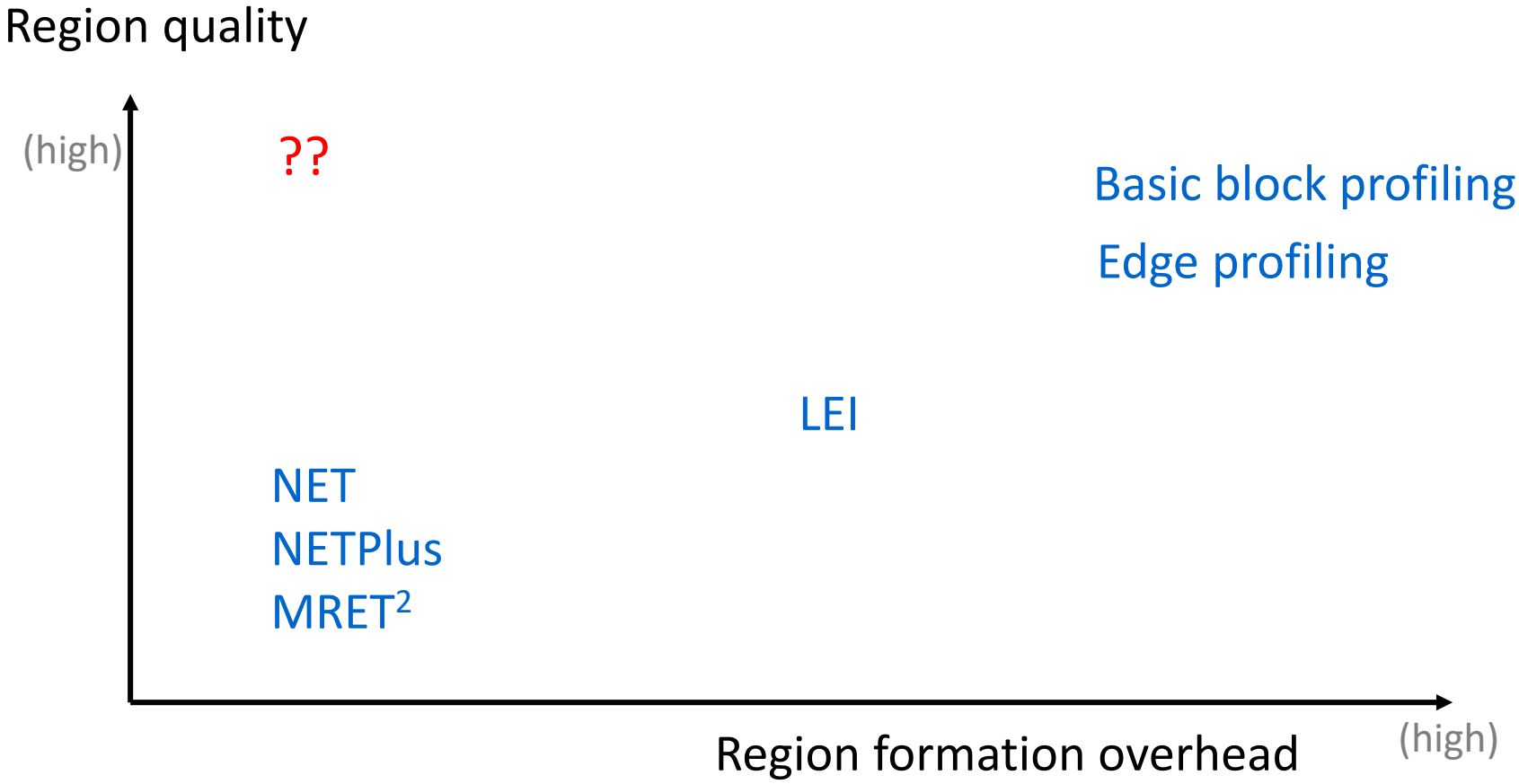
- Instrumentation is a common way to find hot regions
  - Basic block profiling
  - Edge profiling

  } Insert profiling code in all blocks or edges

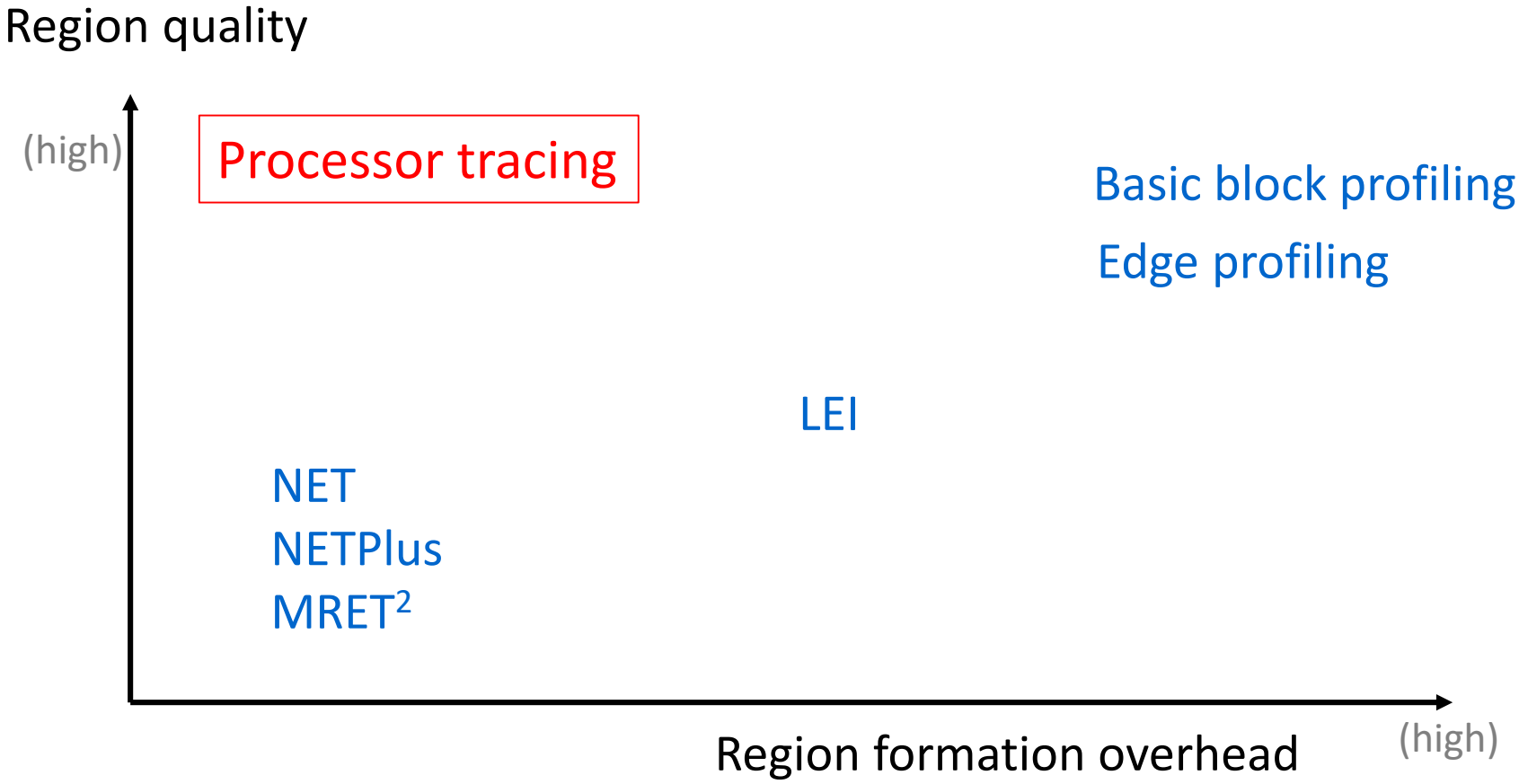
  - Path profiling
  - NET (next executing tail)
  - NETPlus
  - MRET^2
  - LEI (last execution iteration)

  } Selectively insert profiling code, e.g. loop headers

Motivation

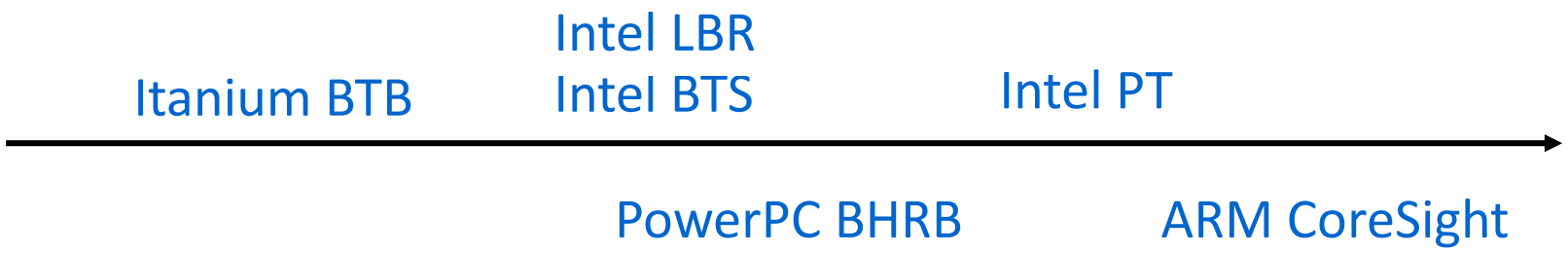

Region quality
(high)
??
Basic block profiling
Edge profiling
LEI
NET
NETPlus
MRET2
Region formation overhead
(high)

Motivation


Region quality
(high)
Processor tracing
Basic block profiling
Edge profiling
LEI
NET
NETPlus
MRET2
Region formation overhead
(high)

Processor Tracing

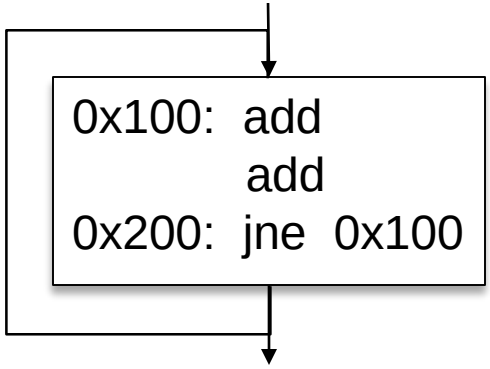
- A hardware function that can track program execution flows, e.g.
  - Source and target addresses of branches
  - Execution mode transition
  - Processor power state changes
  - Transactional memory commit/abort path



* BTB (branch target buffer), LBR (last branch record), BTS (branch trace store)
* BHRB (branch history rolling buffer)

Example: Intel PT

a loop runs 4 iterations

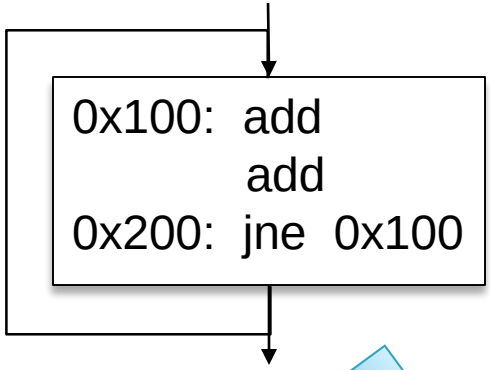


IntelPT packets:

```
PSB (0x100)   # start from 0x100
TNT (1)       # taken
TNT (1)       # taken
TNT (1)       # taken
TNT (0)       # not taken
```

Example: Intel PT

a loop runs 4 iterations



IntelPT packets:

```
PSB (0x100)   # start from 0x100
TNT (1)       # taken
TNT (1)       # taken
TNT (1)       # taken
TNT (0)       # not taken
```

binary disassembling

packet decoding

jne: 0x200 → 0x100
jne: 0x200 → 0x100
jne: 0x200 → 0x100
jne: 0x200 → 0x205

branch history

Our Solution

- Processor-tracing guided region formation
  - Leverage branch history to help form regions
  - No instrumentation
  - High-quality region formation

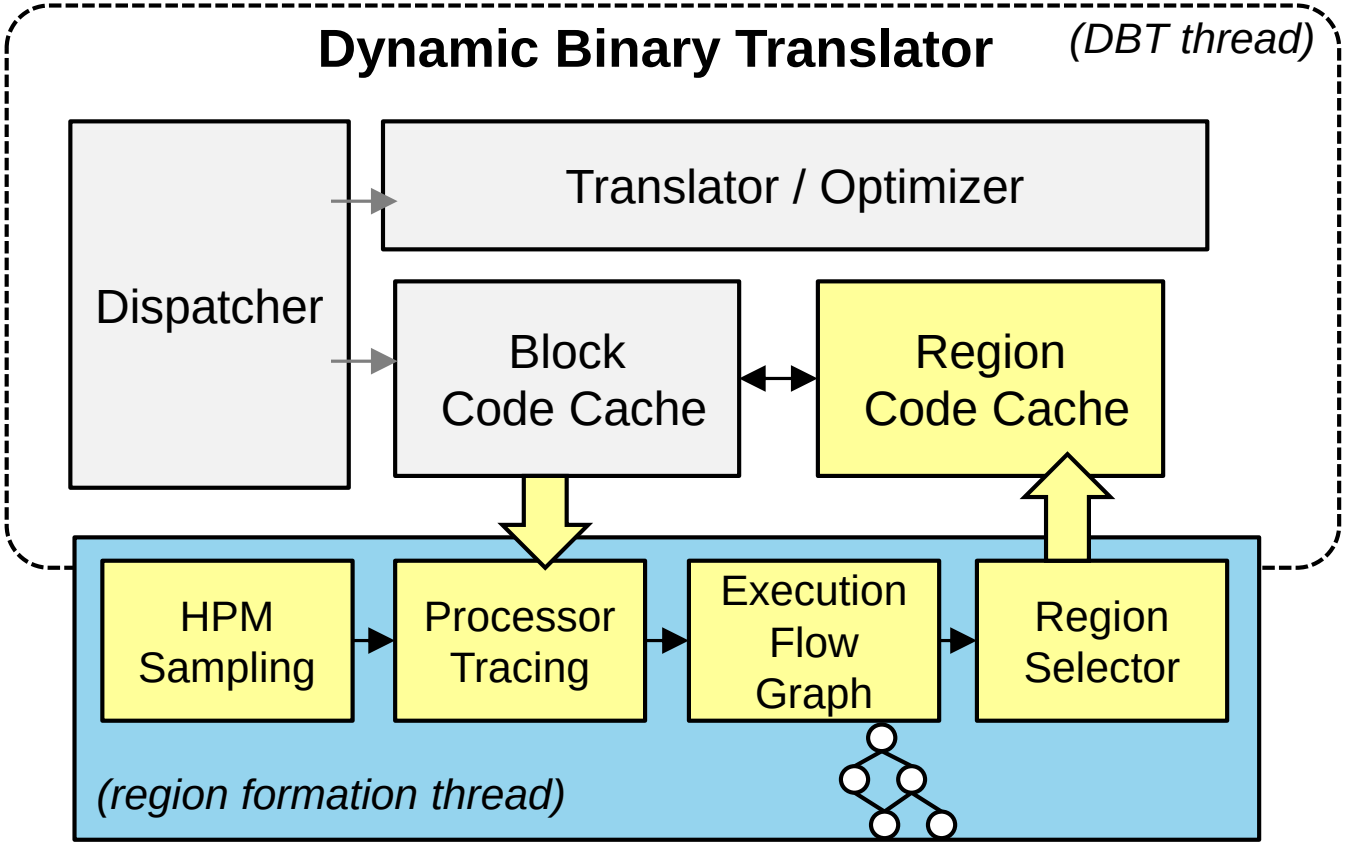
Our Solution

- Processor-tracing guided region formation
  - Leverage branch history to help form regions
  - No instrumentation
  - High-quality region formation

- Challenges
  - How to decide which blocks to be selected in a region
  - When to start processor tracing
  - Overhead of binary disassembling and packet decoding

Processor-Tracing Guided Region Formation

Overall Architecture

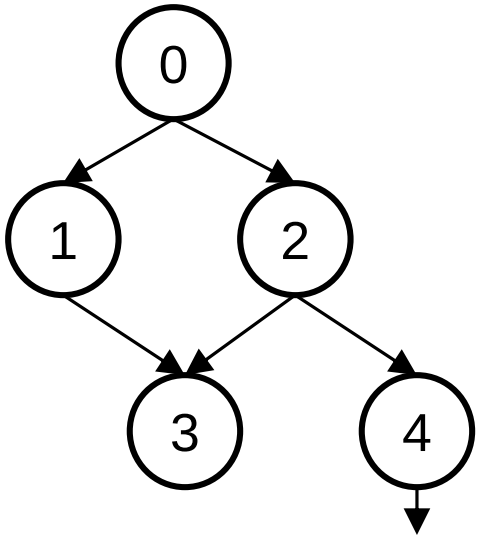


- Region formation conducted by a helper thread

Region Selection Algorithm

- Step 1: build CFG according to the branch history

execution flow graph

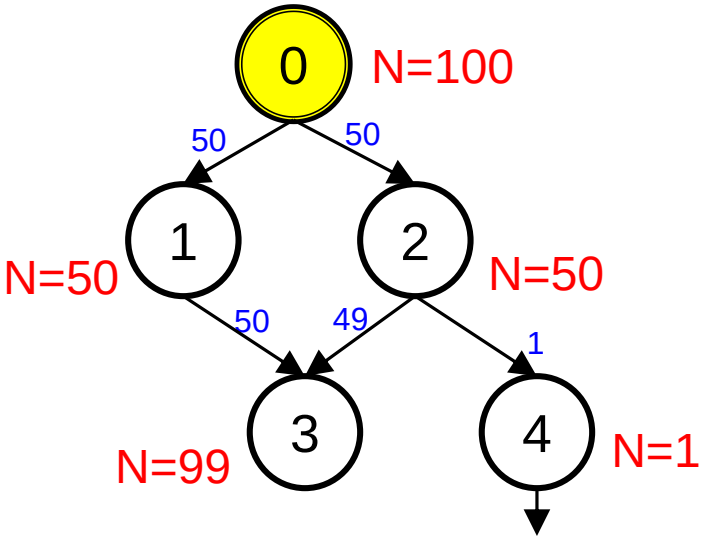


branch history

jne: 0x200 → 0x100
jne: 0x200 → 0x100
jne: 0x200 → 0x100
…
jmp: 0x800 → 0x900

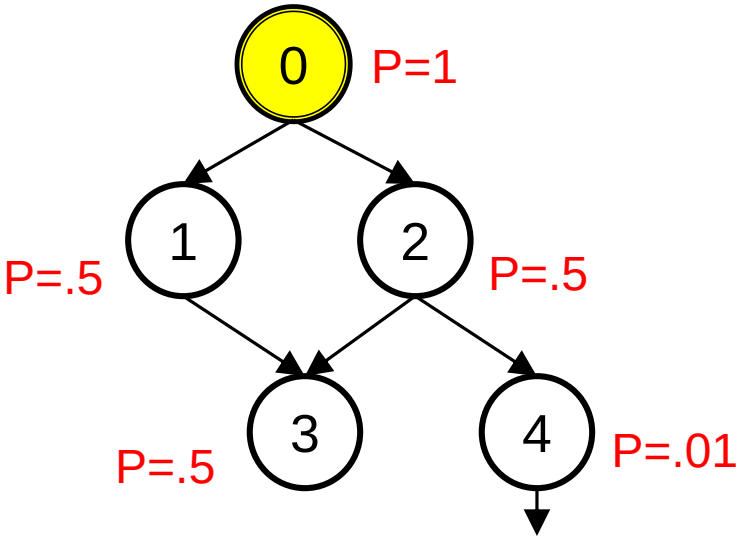
Region Selection Algorithm

- Step 1: build CFG according to the branch history
- Step 2: compute node/edge frequency and set loop headers



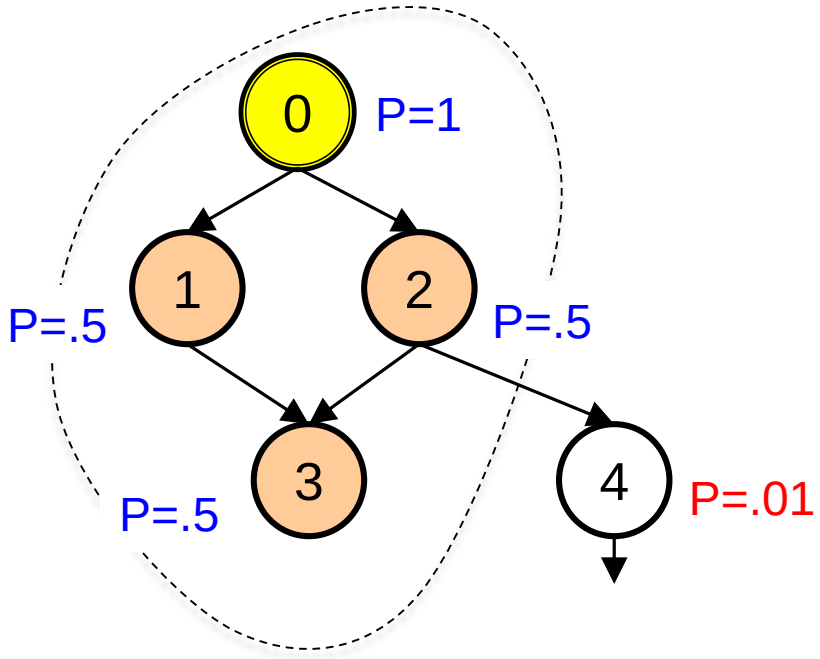
Region Selection Algorithm

- Step 1: build CFG according to the branch history
- Step 2: compute node/edge frequency and set loop headers
- Step 3: select nodes with high reaching probability



Region Selection Algorithm

- Step 1: build CFG according to the branch history
- Step 2: compute node/edge frequency and set loop headers
- Step 3: select nodes with high reaching probability



Select blocks of reaching probability (*P*) > hot threshold (*T*)

➔ execution is likely to stay in the formed region

Processor-Tracing Overhead

- Problem: processor tracing has overhead
  - Hardware tracing overhead (low)
  - Software overhead of decoding binary and packets (high)

- Two solutions
  - Demand-based processor tracing
  - Branch instruction decode cache

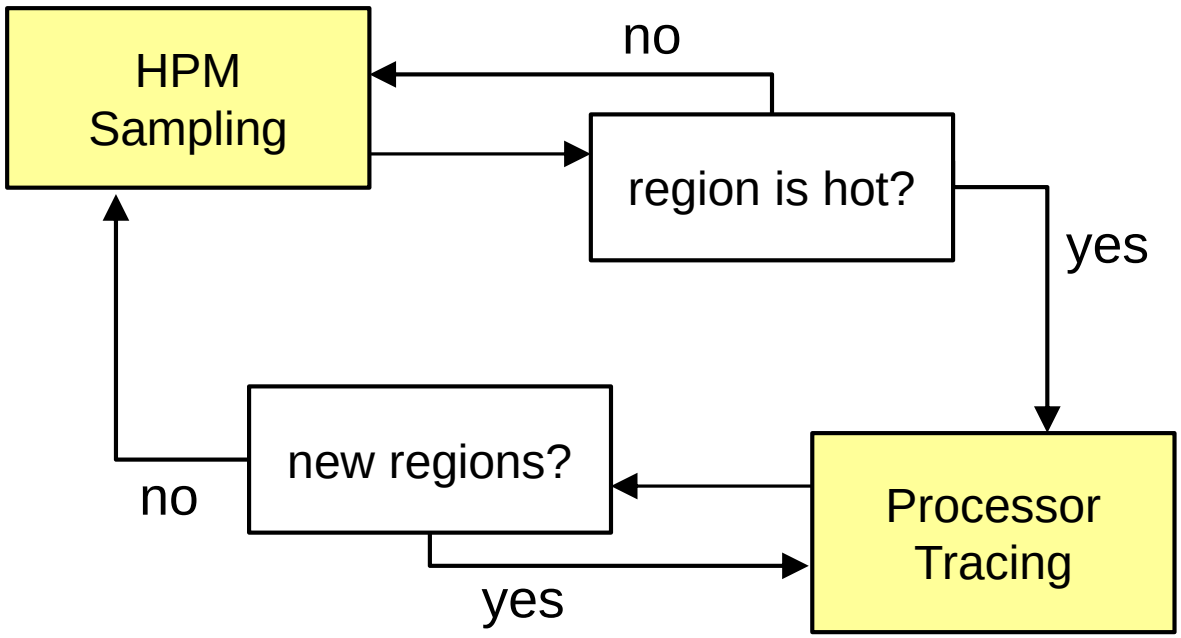
Demand-Based Processor Tracing

- Not enable processor tracing permanently
  - Disable processor tracing when (1) running in cold codes, (2) all hot regions have been built
  - Enable processor tracing only when running in hot regions
- Challenge:
  - How to know the program is running in the hot regions?
- Solution:
  - lightweight HPM sampling

* HPM (hardware performance monitor)

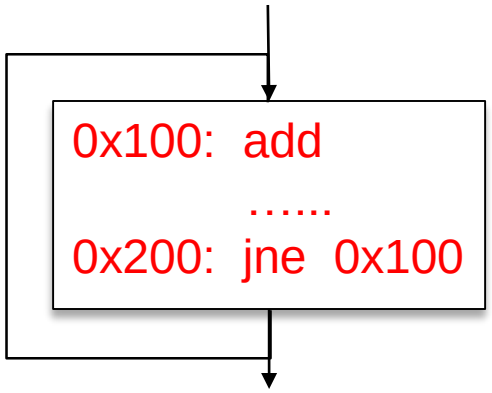
Demand-Based Processor Tracing

- HPM sampling to collect program counters (PCs)
  - Idea: most PCs are in a few basic blocks ➔ hot regions



Software Decoding Overhead

- Significant overhead from decoding the binary
  - Produce one branch may require tens of instruction execution



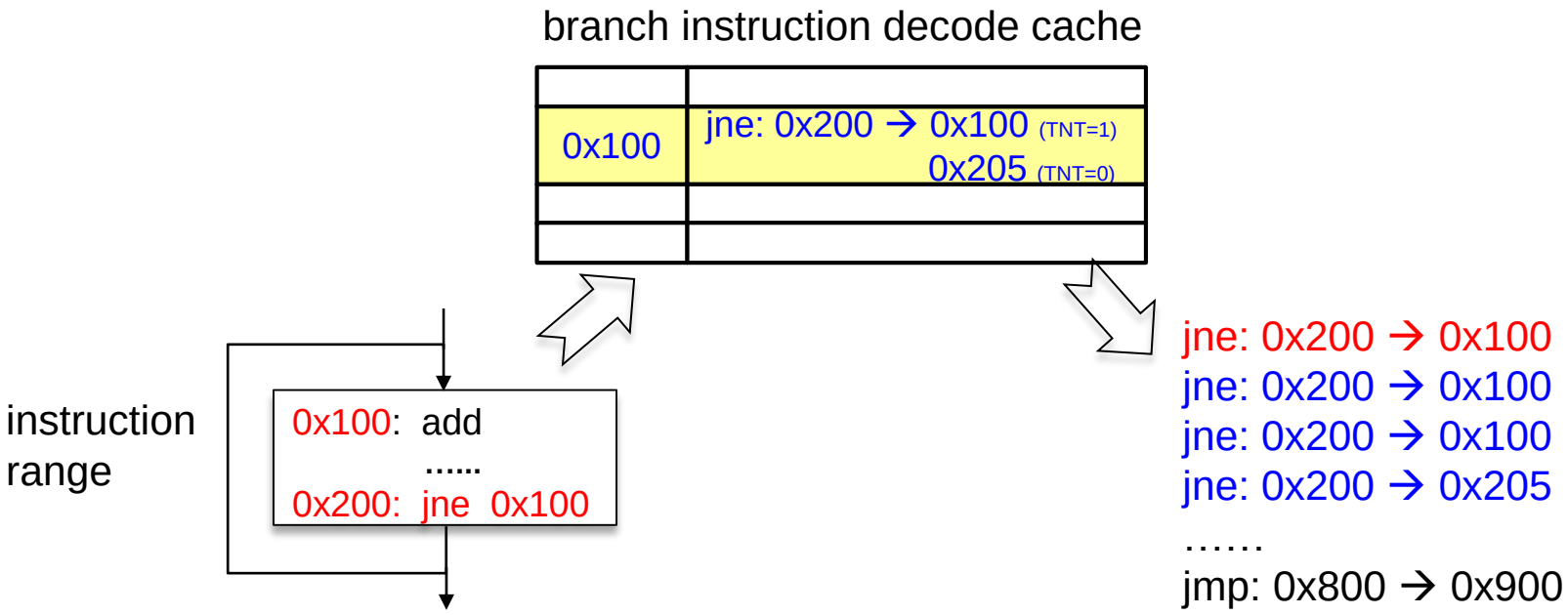
PSB (0x100)
TNT (1)
TNT (1)
TNT (1)
TNT (0)
……

branch history:

jne: 0x200 → 0x100
jne: 0x200 → 0x100
jne: 0x200 → 0x100
jne: 0x200 → 0x205
……
jmp: 0x800 → 0x900

Branch Instruction Decode Cache

- Solution: branch instruction decode cache
  - Cache walked instruction ranges with <start address, branch_info>
  - Upon cache hit, fast generating branch instructions



Evaluation and Conclusion

Evaluation Setup

- HQEMU, a cross-ISA DBT
  - Integrate QEMU and LLVM
  - Regions are compiled by LLVM with O2 optimization

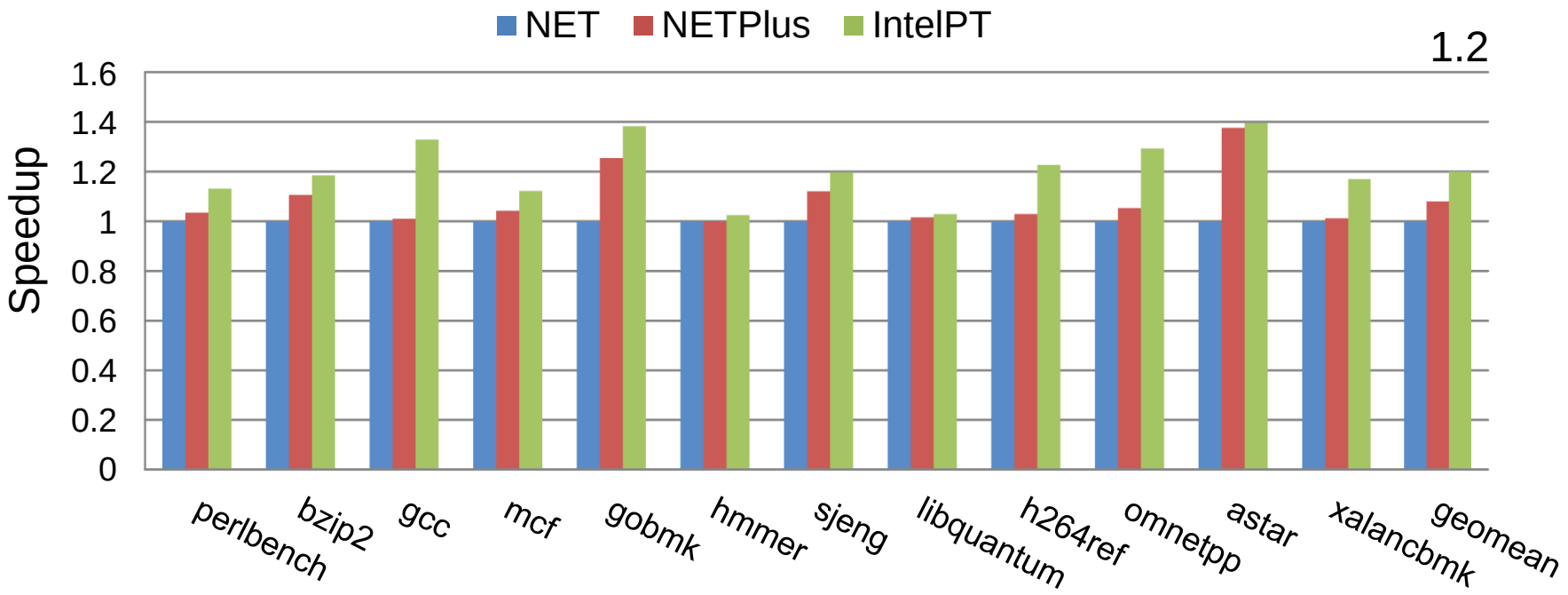
- Cross-ISA binary translation

Type	Host CPU	Processor-tracing
ARMv8 to x64	Intel Core i7 (Skylake)	Intel PT

Evaluation Setup

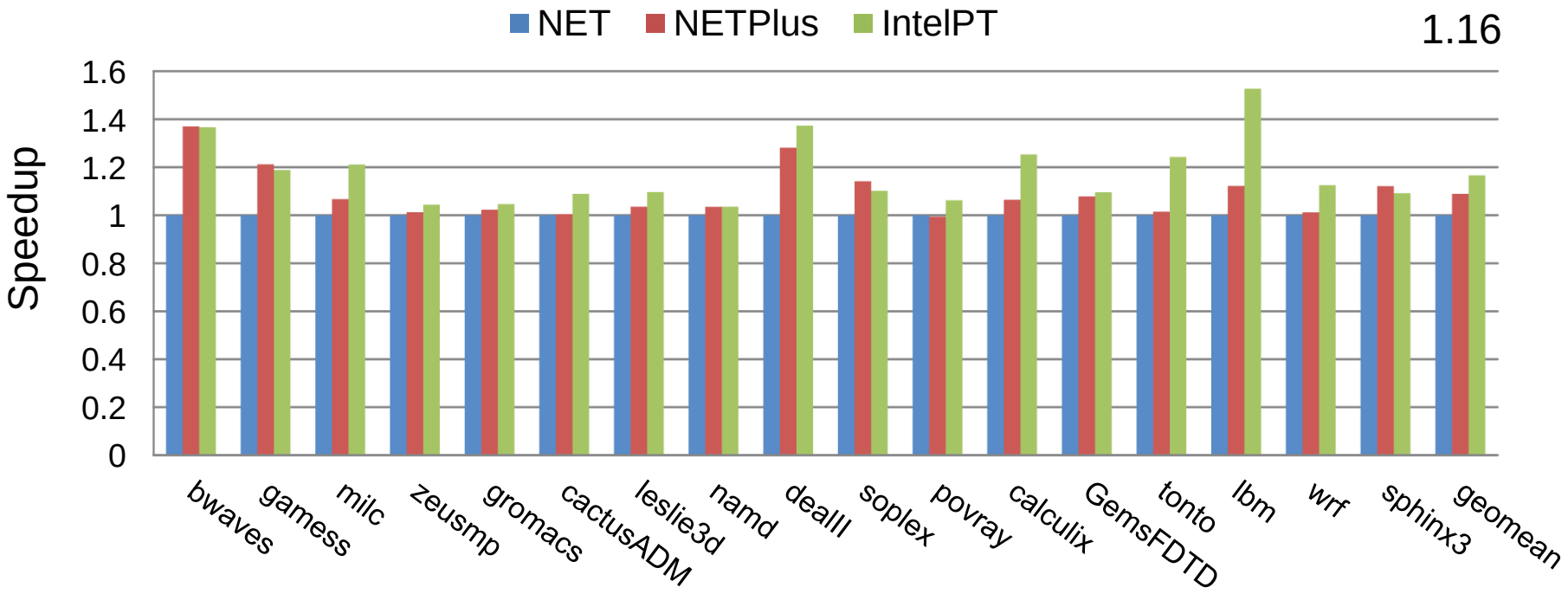
- Benchmarks
  - SPEC CPU2006 benchmarks with reference inputs
- Comparison
  - NET (baseline)
  - NETPlus

Speedup of Integer Benchmarks



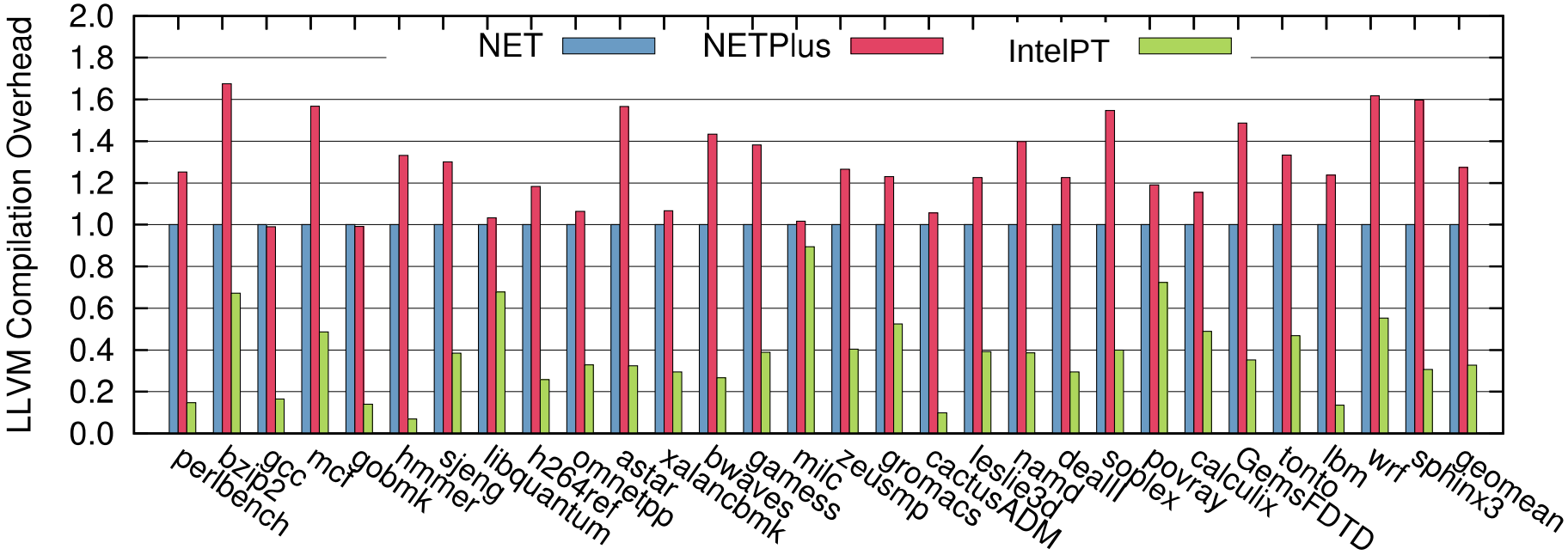
- On average, our approach achieves
  - 1.2x speedup over NET, and 1.1x over NETPlus
  - Significant improvement for benchmarks with complex CFG, e.g., gcc

Speedup of Floating-Point Benchmarks



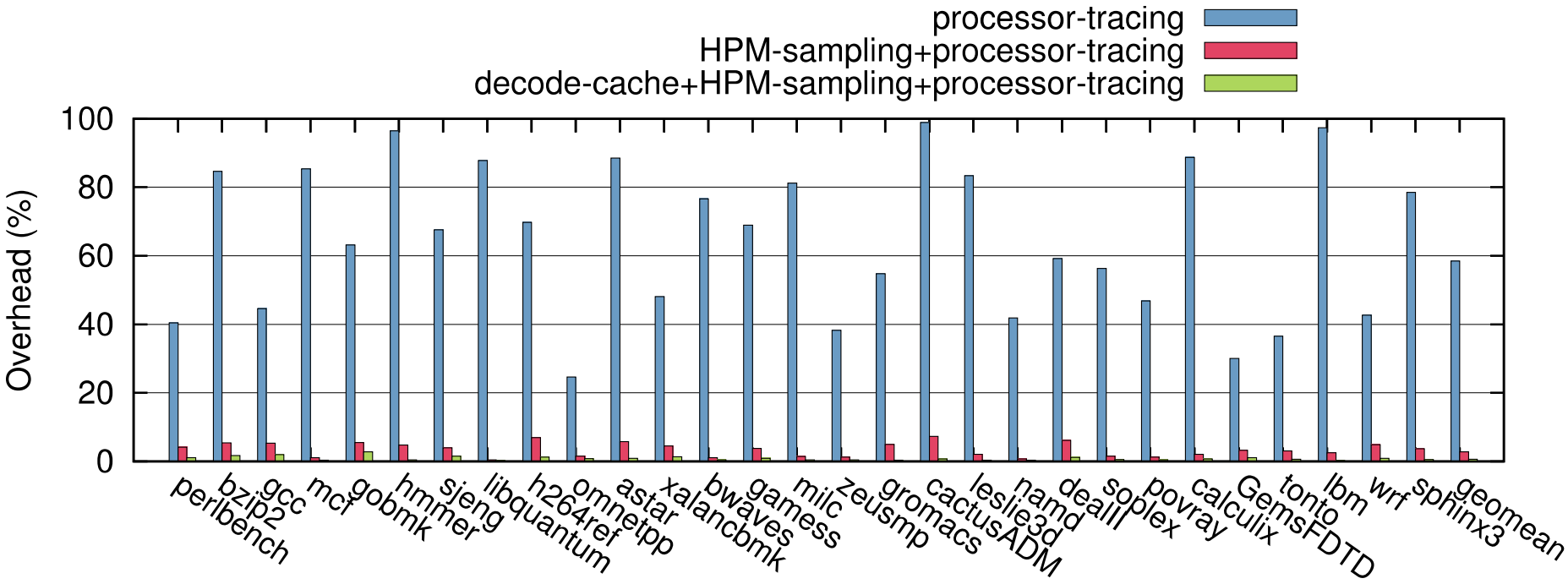
- On average, our approach achieves
  - 1.16x speedup over NET, and 1.07x speedup over NETPlus

LLVM Optimization Overhead



- Our approach achieves the lowest compilation overhead
  - 1/3 NET overhead, and 1/4 NETPlus overhead, on average
  - Generate much fewer number of regions

Region Formation Overhead



- Processor tracing only: 58% on average
- HPM sampling: < 8%
- HPM sampling + branch instruction decode cache: < 3%

Conclusion

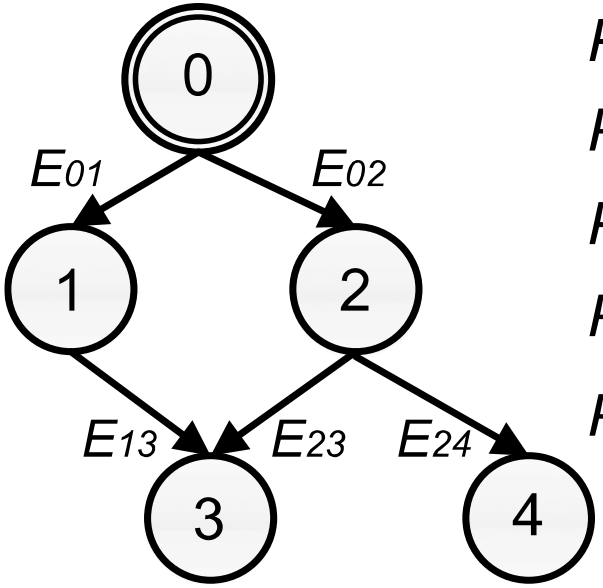
- Processor-tracing guided region formation
  - High-quality region formation
  - No instrumentation is required
  - Region selection algorithm based on reaching probability
- Minimize processor-tracing overhead
  - Lightweight HPM sampling
  - Branch instruction decode cache
- Achieve significant improvements
  - The best performance and lowest compilation overhead, compared to NET and NETPlus

Q & A

Thank you for your attention

Backup

Region Selection Algorithm



$P_0 = 1.0$

$P_1 = P_0 \times E_{01}$

$P_2 = P_0 \times E_{02}$

$P_3 = \max (P_1 \times E_{13}, P_2 \times E_{23})$

$P_4 = P_2 \times E_{24}$

- 1. Compute reaching probabilities (*P*) from node/edge frequencies
- 2. *P* (loop header) = 1
- 3. *P* (child) = *P* (parent) * Edge's Probability
- 4. Select max *P* if multiple incoming edges

Example: Branch Instruction Decode Cache

Assembly:

```
0x100:  add
        add
0x200:  jmp 0x400

0x400:  add
        add
0x600:  jne 0x100
```

IPT packet:

PSB (0x100)
TNT(1)
TNT(1)
TNT(1)
TNT(0)

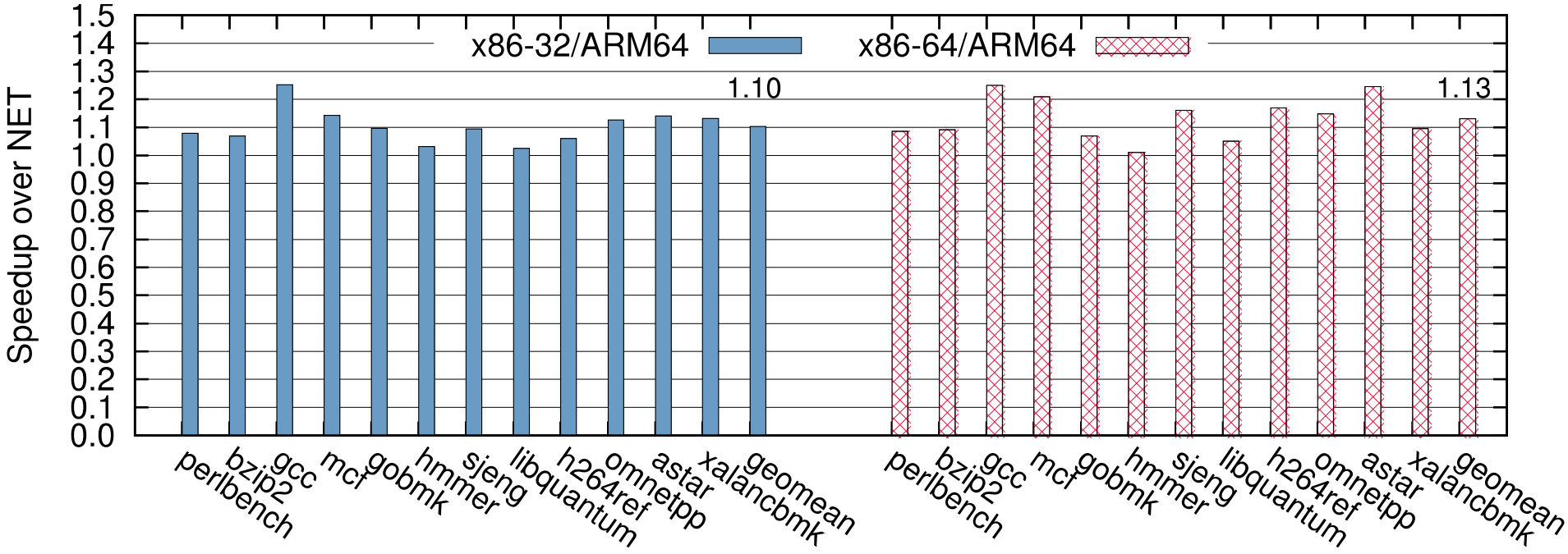
Branch record:

<jmp,0x200,0x400>
<jne, 0x600,0x100>
<jmp,0x200,0x400>
<jne, 0x600,0x100>
<jmp,0x200,0x400>
<jne, 0x600,0x100>
<jmp,0x200,0x400>
<jne, 0x600,0x606>

Key	Value
0x100	<jmp,0x200,0x400>
0x400	<jne, 0x600,X>

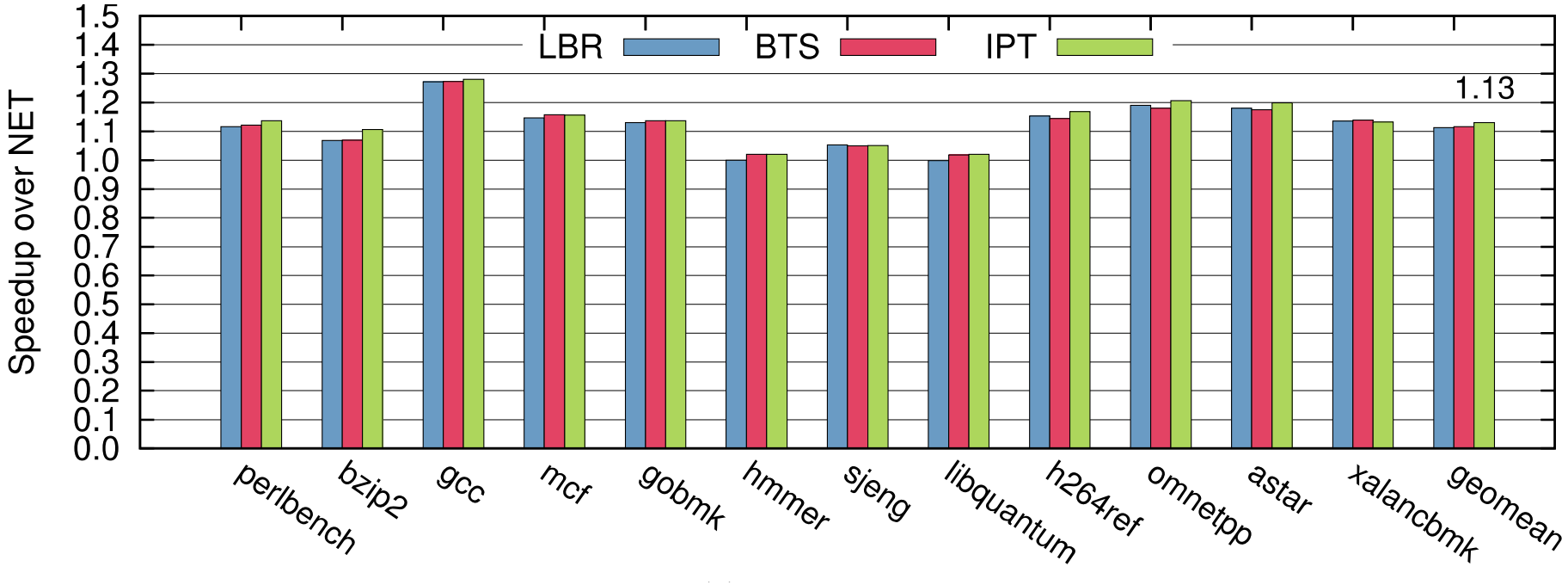
- Instruction range: the range between the last branch target and next branch instruction

Speedup with x86-to-ARM64 Translation



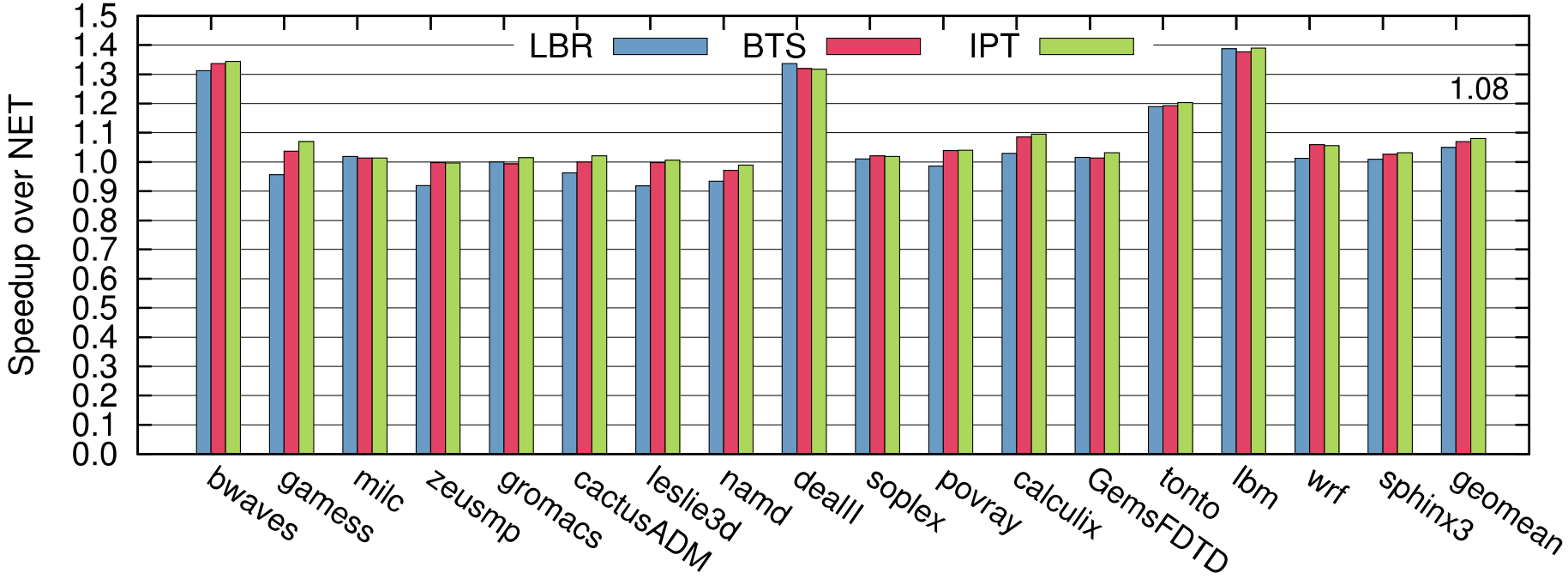
- x86/x64 to ARMv8 translation on Cortex A57, with ARM CoreSight
- On average, our approach achieves 1.1x speedup over NET

Speedup with ARM32-to-x64 Translation



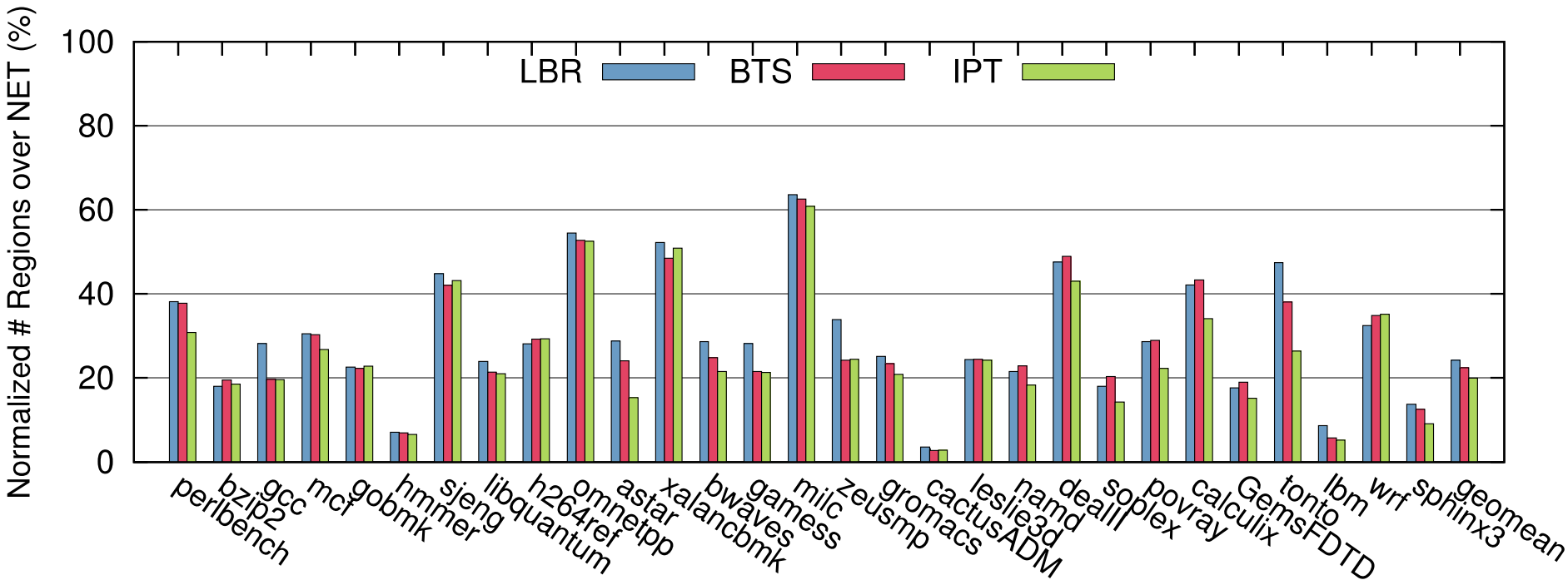
- Comparison with LBR, BTS and IntelPT
- Our approach achieves a speedup of 1.13x over NET with CINT2006

Speedup with ARM32-to-x64 Translation



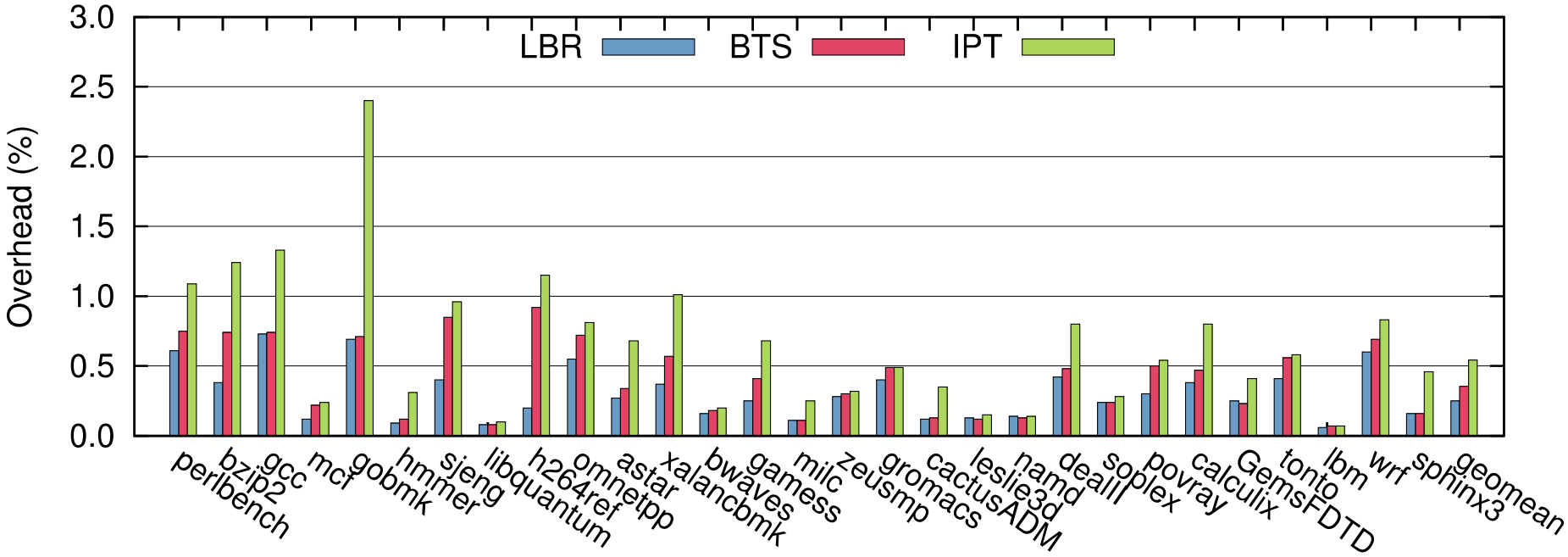
- Comparison with LBR, BTS and IntelPT
- Our approach achieves a speedup of 1.08x over NET with CFP2006

Number of Regions



- 20% the number of NET regions

Region Formation Overhead



- LBR and BTS: < 1%
- Intel PT: < 3%