A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT)

Wenwen Wang
Pen-Chung Yew, Antonia Zhai, Stephen McCamant
wenwang@cs.umn.edu



UNIVERSITY OF MINNESOTA

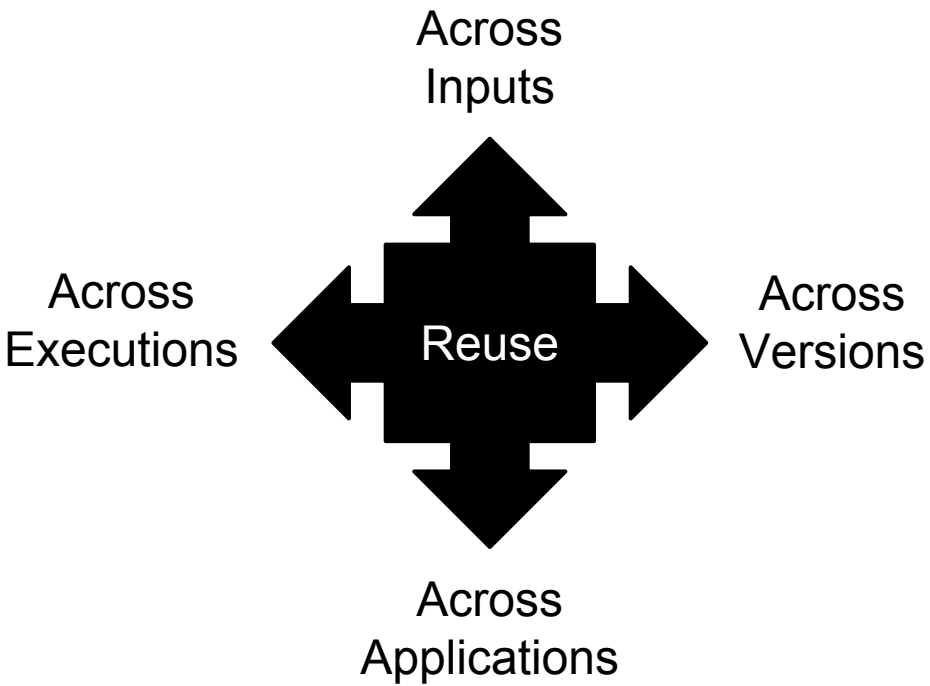
Motivation

Dynamic binary translation (DBT) is a key enabling technology

- ✓ Enable many applications
  - System virtualization
  - Whole program analysis
  - System security
- High translation overhead
  - For long-running applications: amortize translation overhead

☞ **Short-running applications are common in web applications**

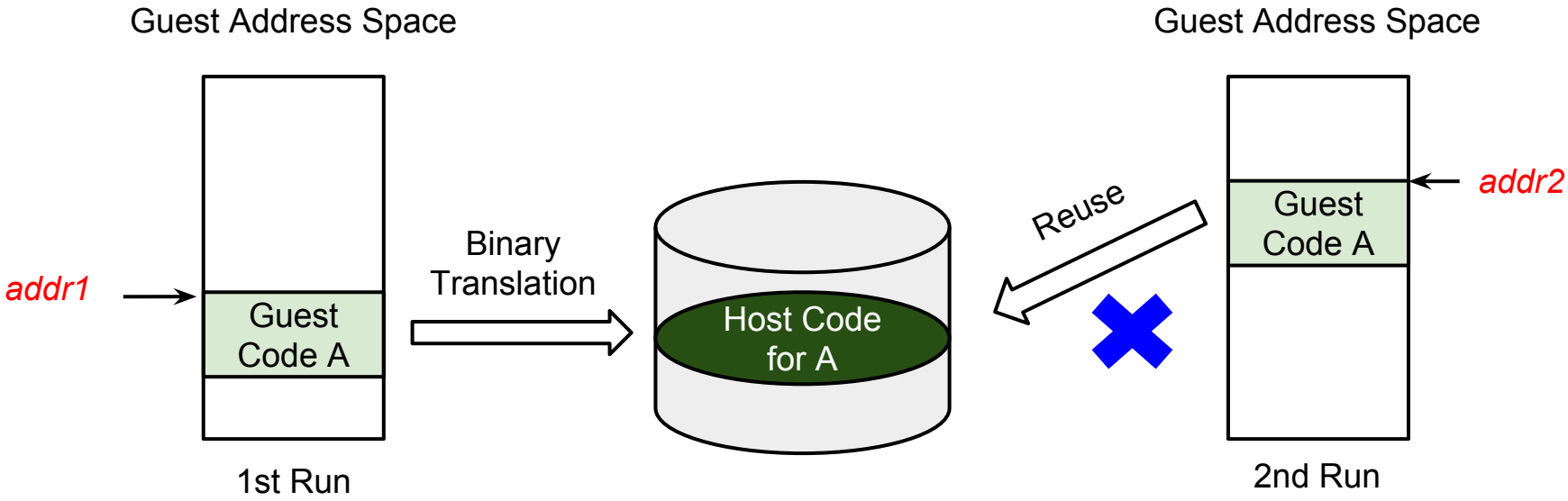
Code Reuse: Effective in Reducing Translation Overhead



Three Key Challenges:

- Relocatable guest binaries
- Absolute addresses in translated host binaries
- Dynamically generated guest binaries
  - Code generated by JIT compilers

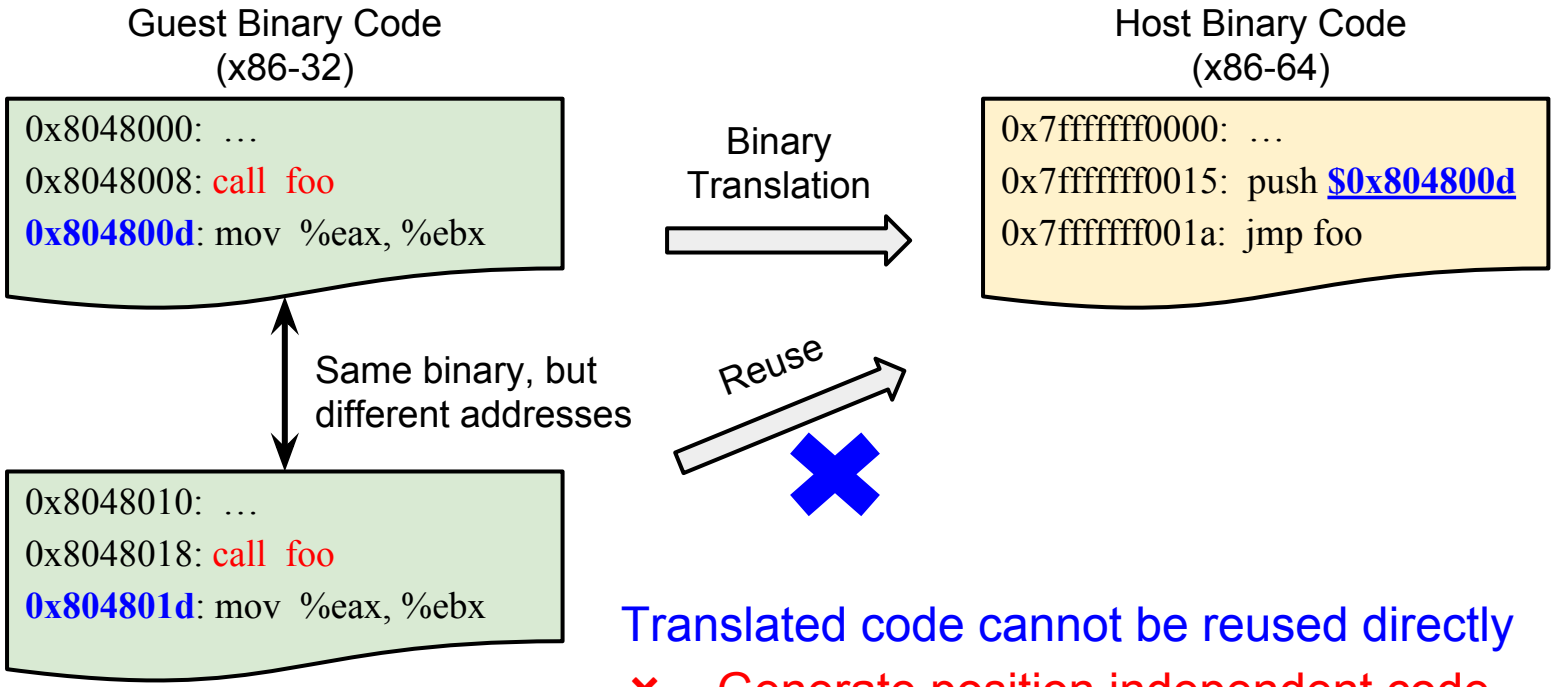
Challenge due to Relocatable Guest Binaries



✖ Cannot use instruction addresses to locate translated code

☞ Directly use the guest binary code to search for translated code

Challenge due to Absolute Addresses in Host Binaries



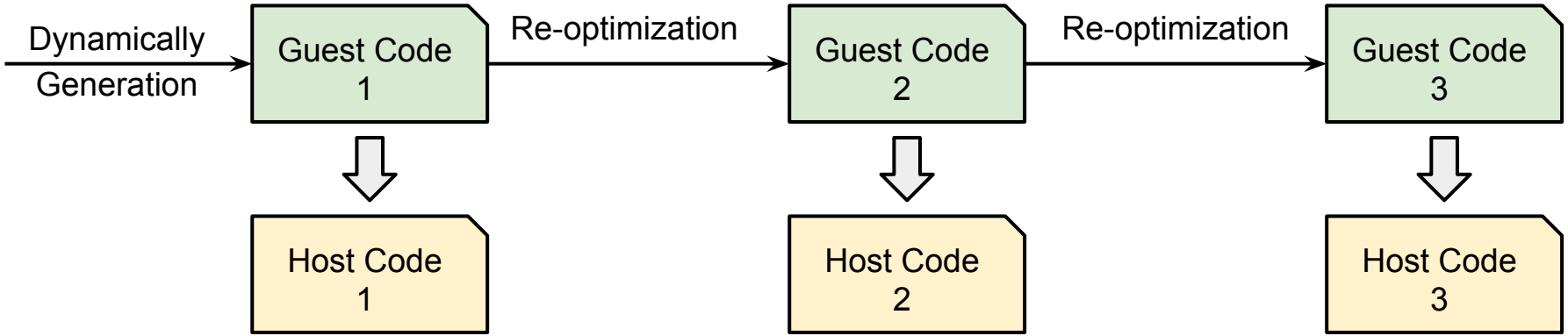
Translated code cannot be reused directly

× Generate position independent code

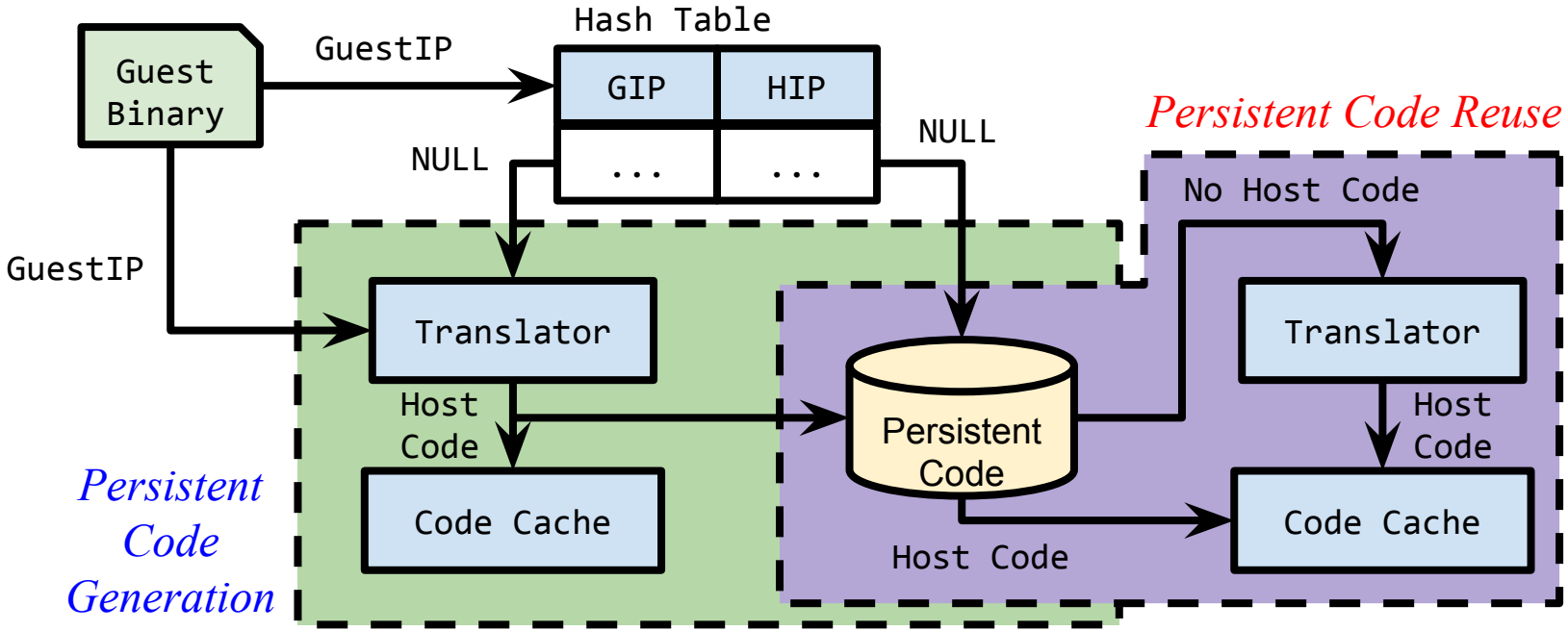
✓ Generate relocation records in translated code

Challenge due to Dynamically Generated Binaries

- Dynamically generated code have different addresses across runs
  - Search translated code with guest binary
- Unable to locate the original guest code after re-optimization
  - Save guest code after the translation

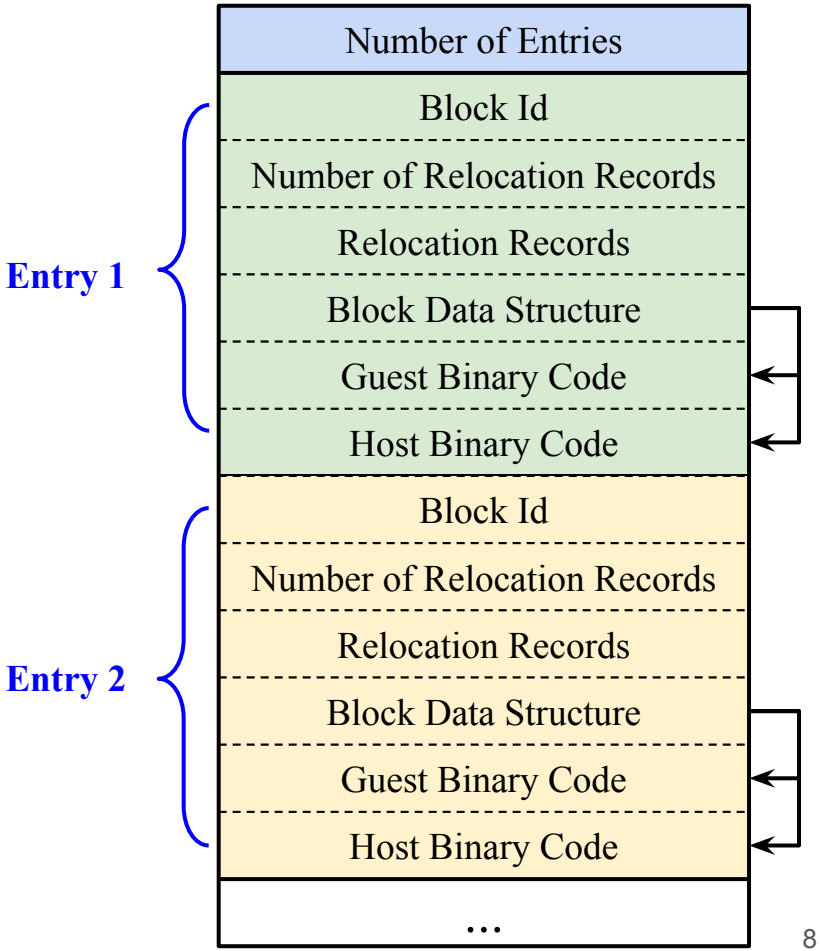


A General Framework - Two Phases



Persistent Code Structure

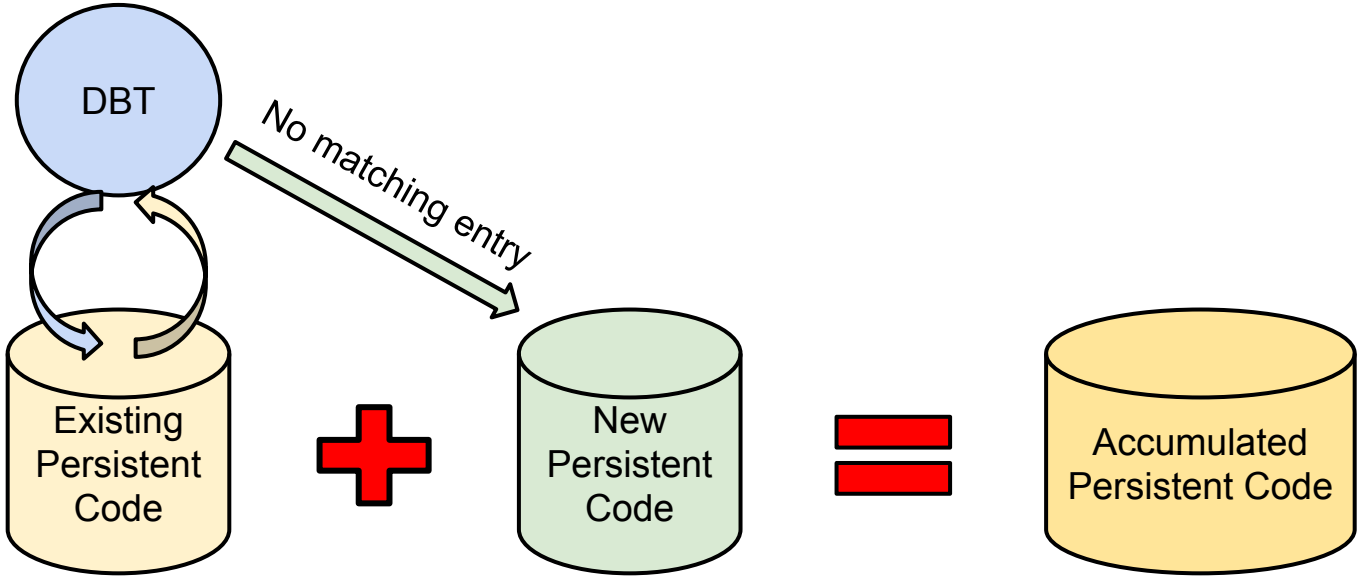
- Each persistent code entry corresponds to the translation unit
  - Basic block
  - Trace
- Each entry contains
  - Block/Trace ID
  - Guest binary code
  - Host binary code
  - Relocation records for host binary code
  - Meta data for the translation unit



Persistent Code Searching

- Create a hash table
  - Using the size and content of guest binary code instead of guest instruction addresses
- Handle hash table collision
  - Two level hash key to reduce hash table collision
  - Byte-level comparison using guest binary code
- Global optimizations across translation units
  - Compare guest binary code of connected translation units (Details in Section 2.2)

Persistent Code Accumulation



Translated codes can be shared across applications

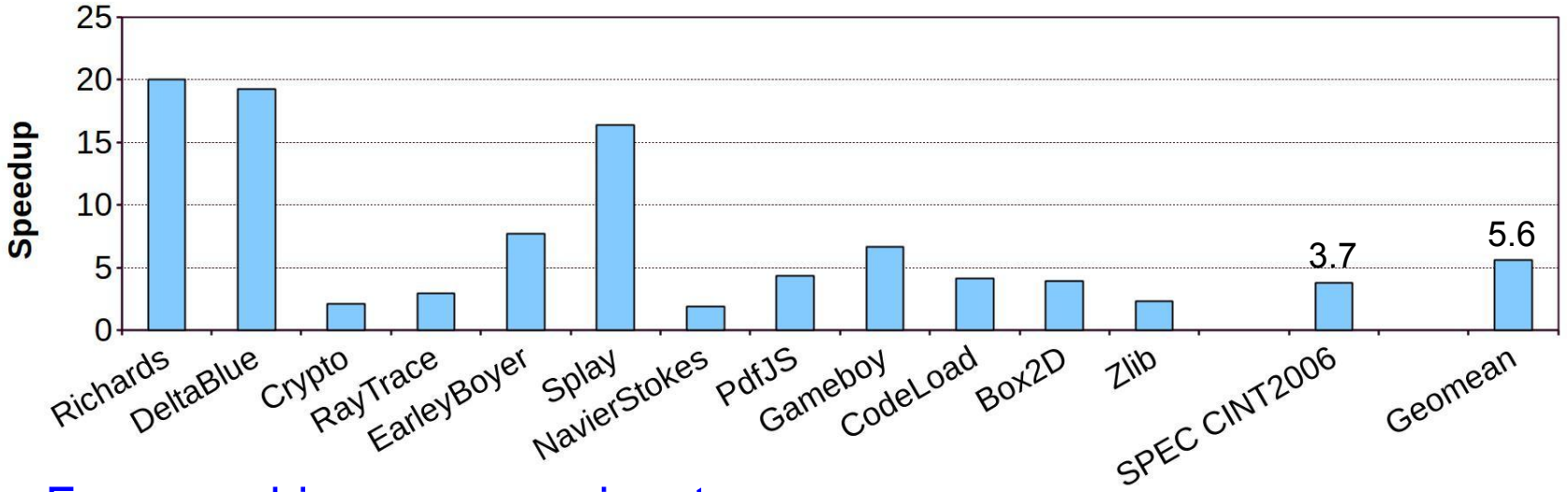
Implementation and Experiments

- Leverage an existing DBT system, HQEMU [13]
  - Retargetable DBT system based on QEMU
  - Use QEMU for basic block translation
  - Use LLVM for trace optimization

- Benchmarks
  - SPEC CINT2006 with *test* input set
  - SpiderMonkey from Mozilla with Google Octane



How much translation overhead can be reduced?



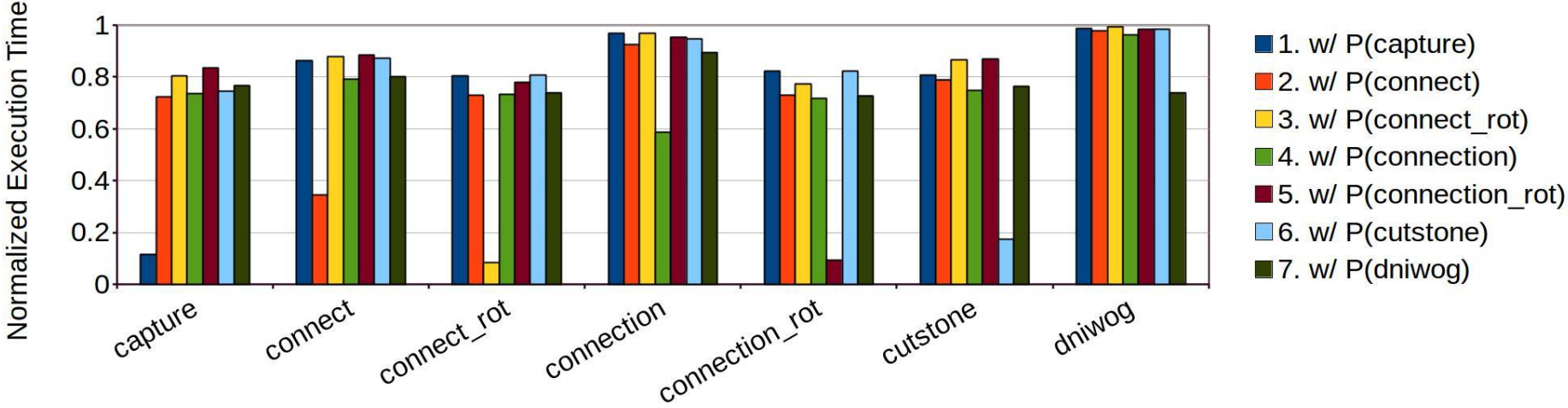
For same binary + same input

- ❏ Speedup > 5X
- ❏ Size < 20MB
- ❏ Overhead < 1%

☞ **Is translated code reuse effective across inputs + applications?**

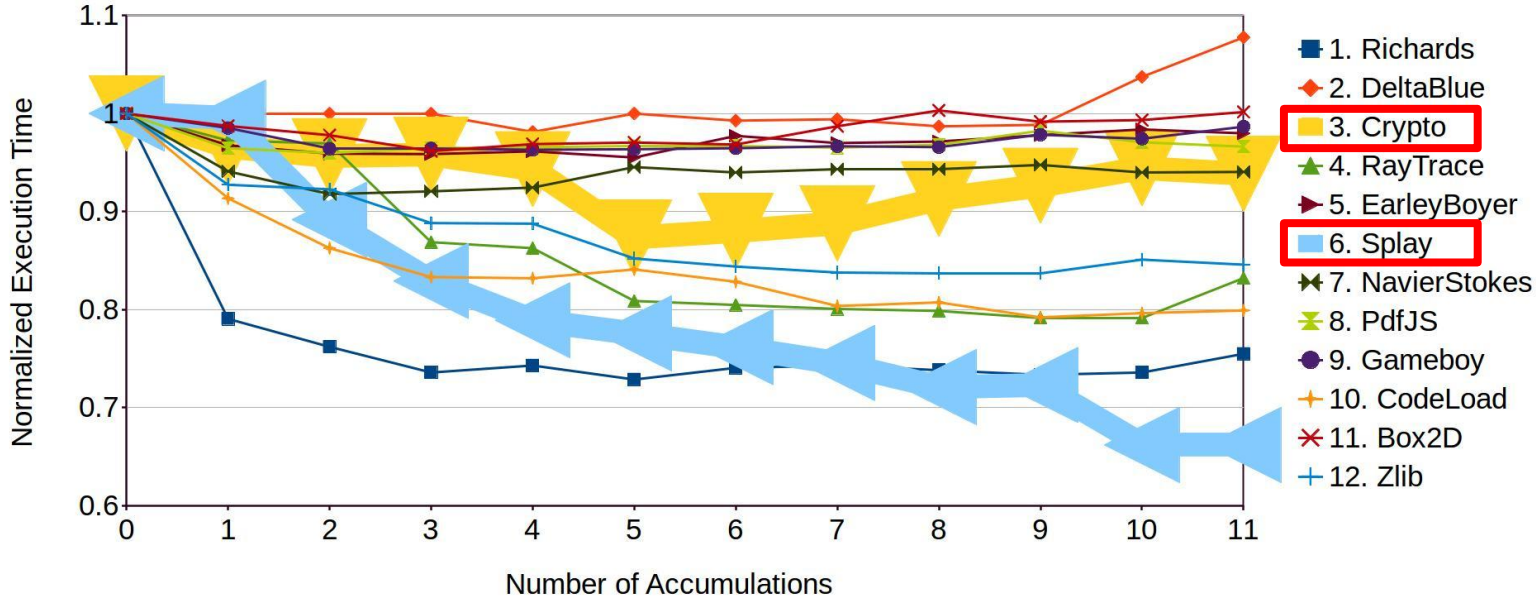
Code Reuse across Inputs

Gobmk from SPEC CINT2006: 7 inputs in test input set



☞ **Persistent code caching is effective across different inputs**

Code Reuse across Applications



☞ **Persistent code can be shared across applications**

- **Code size management is necessary**

Conclusion

- Using persistent code caching is an effective way to mitigate translation overhead for short running applications
- Need to overcome several challenges for effective persistent code caching
  - Relocatable guest binaries
  - Absolute addresses in host binaries
  - Dynamically generated guest binaries
- Our proposed framework can address those challenges effectively to reuse persistent code across different executions, different inputs, or different applications

Thank you!

wenwang@cs.umn.edu



