Arancini

Emulating x86 binaries on the RISC–V architecture

Simon Kammermeier
Advisor: Dr. Redha Gouicem
Chair of Distributed Systems and Operating Systems
https://dse.in.tum.de/

15.11.2022 – 15.04.2023

Reason for binary translation

- Program compatibility key requirement for adoption of new CPU architectures
- Recompilation often not possible
- General solution: Translate binary to new CPU architecture

Existing solutions

- QEMU: Open source, support for many ISAs, translates at execution time
- Lasagne, mctoll: Open source, x86-64 via LLVM to ARM, translates ahead of time
- Apple Rosetta2: Proprietary, x86-64 to ARM, exploits special hardware, combines translation ahead of time and at execution time

Approaches of Binary Translation

Static binary translation	**Dynamic binary translation**
Ahead-of-time	At Runtime
Slow, complex translation possible	Fast, simple translations
Optimized code	Unoptimized code
Problem: Code Discovery	

⇒ Combine both approaches to get fast runtime and full coverage

Arancini

Hybrid binary translation

System design goals:

- Modularity
- General hardware support
- Multithreaded application support

Problem statement

Implementation of the dynamic path for x86 to RISC-V

Outline

- ~~Motivation~~
- Background
  - RISC-V
- Design
- Implementation
- Evaluation

Background: RISC-V

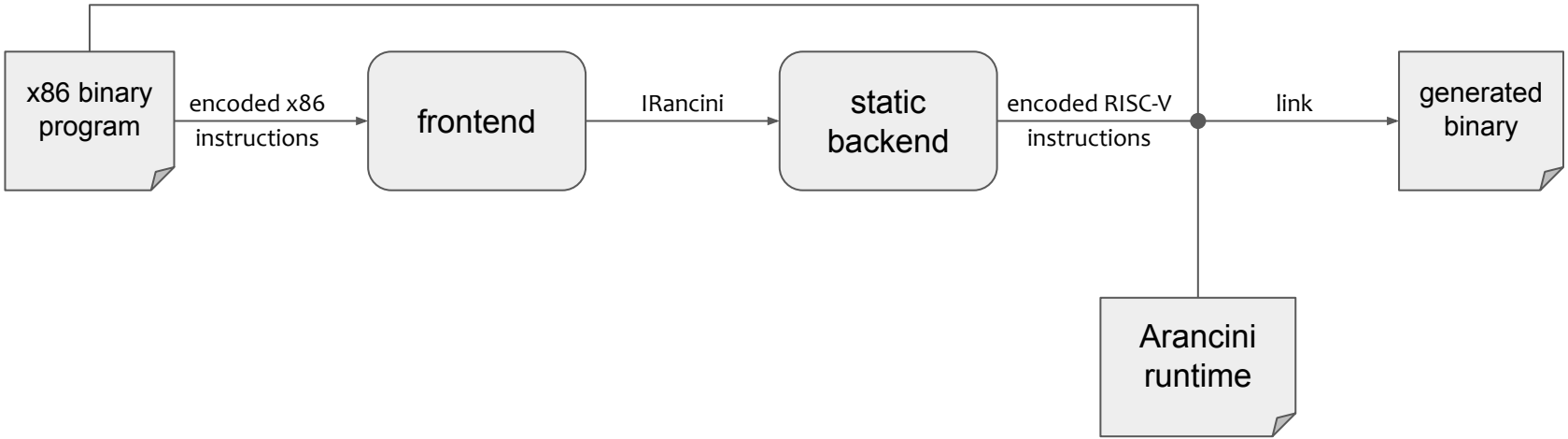
- Initially research project at UC Berkeley
- Goal: free, open source ISA for real world implementations

- Key design differences to x86:
  - RISC philosophy
  - Load-Store Architecture
  - No condition flags

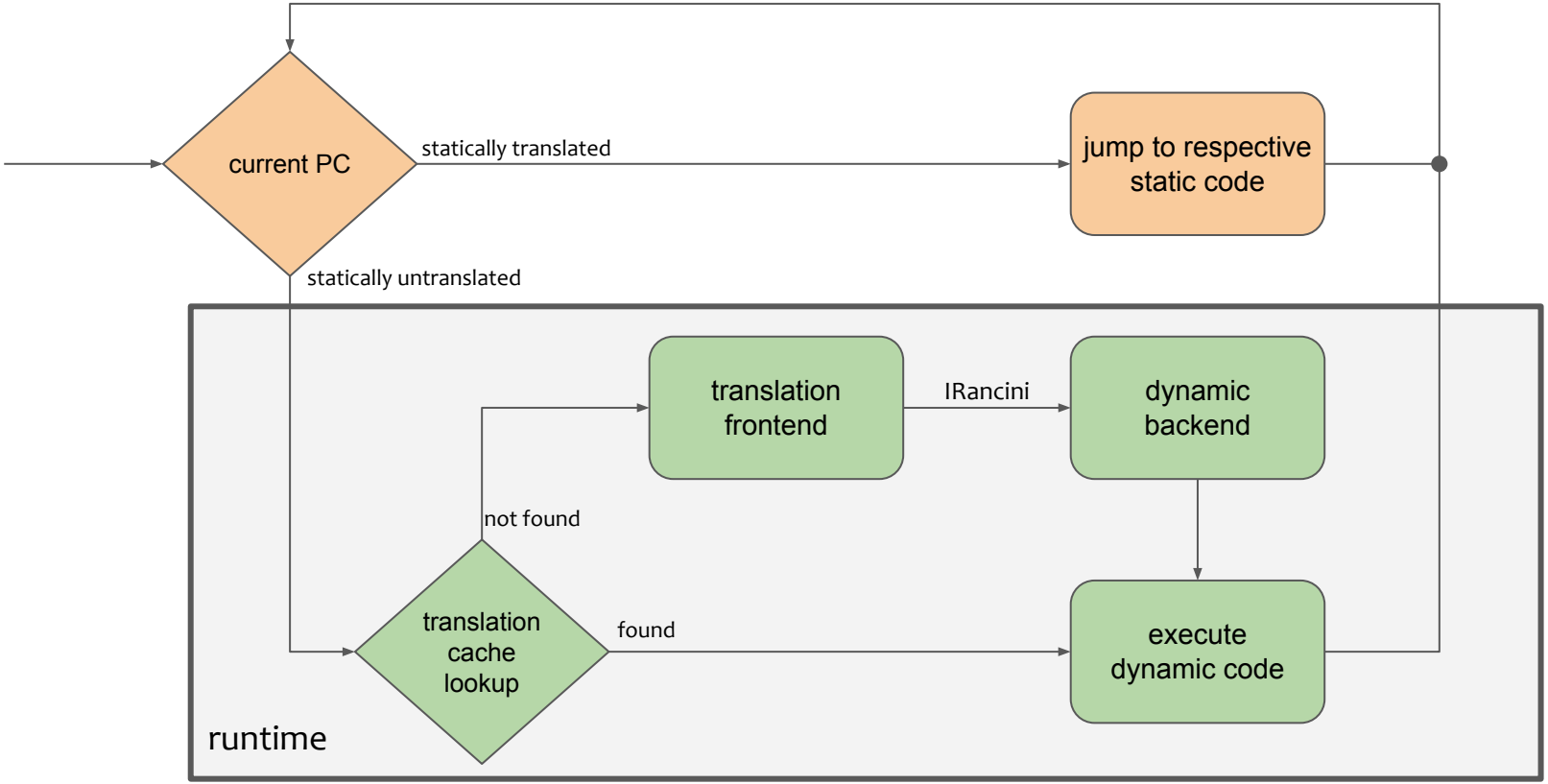
Outline

- ~~Motivation~~
- ~~Background~~
- Design
  - System overview
  - IRancini
- Implementation
- Evaluation

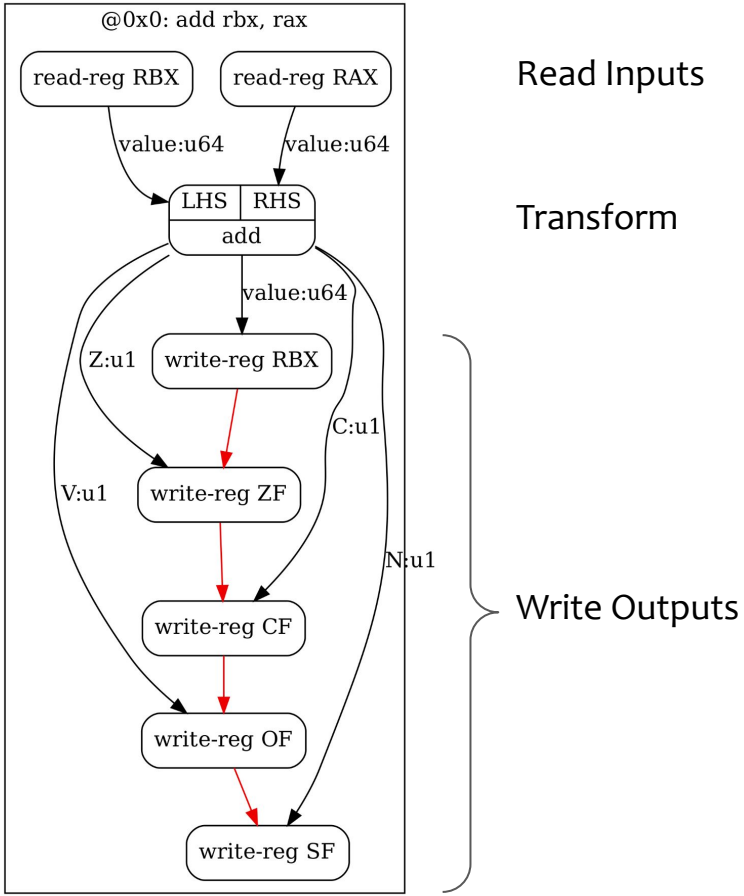
System overview: Static Phase



System overview: Runtime Phase



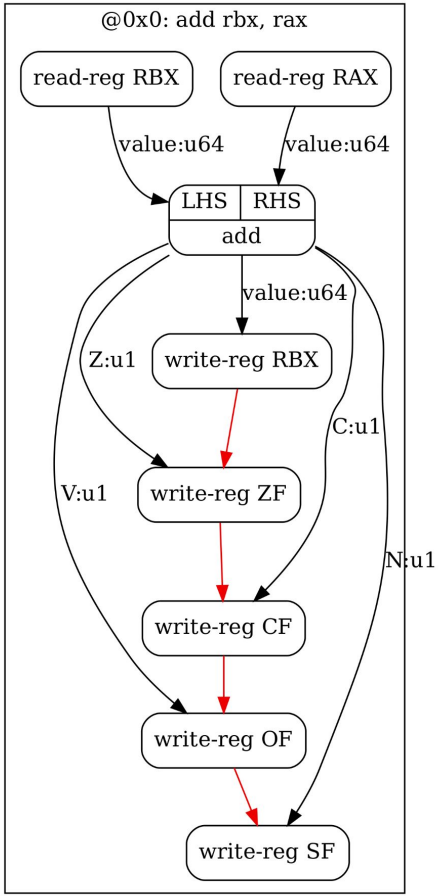
Design overview: IRancini



Outline

- ~~Motivation~~
- ~~Background~~
- ~~Design~~
- Implementation
- Evaluation

Implementation




```
ld     a0, 32(fp)
ld     a1,  8(fp)

add    s1,  a0, a1

sltz   s9,  a0
slt    s10, s1, a1
xor    s10, s10, s9
sltu   s9,  s1, a1
seqz   s8,  s1
sltz   s11, s1

sd     s1,  32(fp)
sb     s8, 392(fp)
sb     s9, 393(fp)
sb    s10, 394(fp)
sb    s11, 395(fp)
```

Overflow flag

Outline

- ~~Motivation~~
- ~~Background~~
- ~~Design~~
- ~~Implementation~~
- Evaluation

Evaluation

- Are the programs emulated correctly?

- How much overhead is created by the translation?

Evaluation

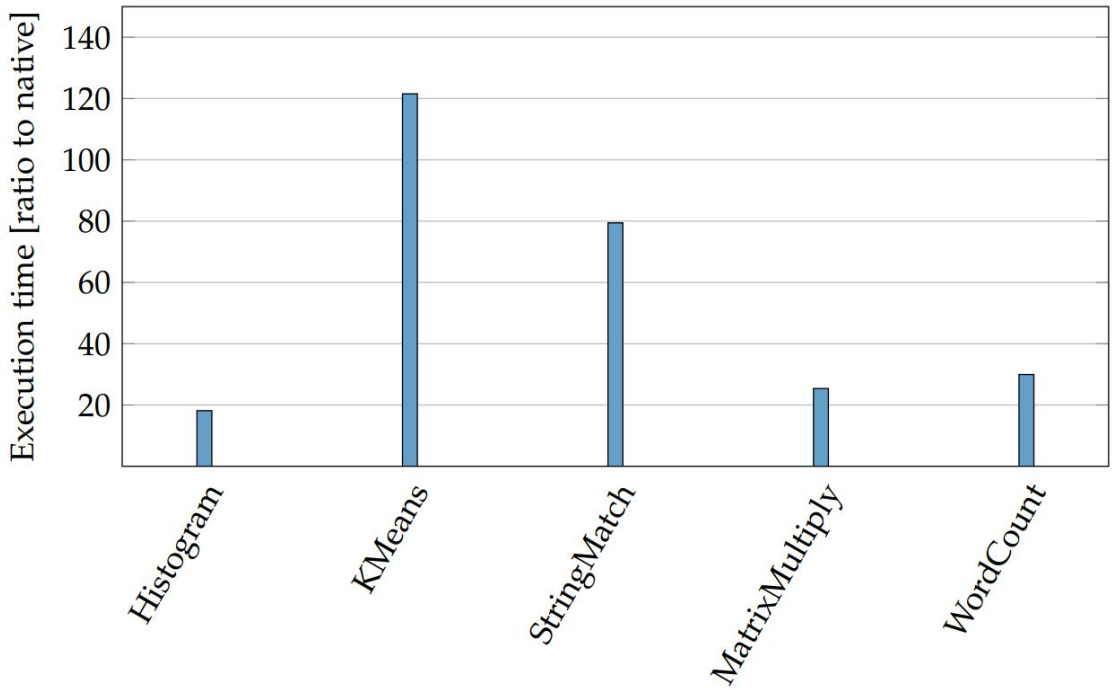
- Experimental setup:
  - HiFive Unmatched RISC–V Development Kit by SiFive
  - U74-MC CPU:
    - 4 U74 cores: RV64GC

- Test programs:
  - Various single threaded integer benchmarks provided as part of phoenix
  - Comparison to benchmarks natively compiled

Correctness

- No crashes in any of the test programs
- Comparison of benchmark outputs showed no differences
- No strict proof of correctness but good evidence

Runtime overhead



Optimization opportunities

- Register mapping to avoid memory accesses
- Elimination of redundant instructions
  - unused flag computations
- Block chaining

Summary

Translation approach of split front and backend very feasible for x86 to RISC–V

Correct translation of single threaded programs using integer scalar instructions

Performance is currently not good

Some easy optimizations could improve performance

Backup

Design overview: IRancini