MicroProbe: An Open Source Microbenchmark Generator Ported to the RISC-V ISA

Schuyler Eldridge    Ramon Bertran    Alper Buyuktosunoglu    Pradip Bose

7th RISC-V Workshop, 2017
Motivation

Architectural, Design, and Validation Teams have Questions

For my new architecture/design:
- What is the maximum power consumption?
- Are there performance bugs?
- Is my design reliable?
- ...

A typical strategy involves writing microbenchmarks

However, microbenchmarks are:
- Time consuming, tedious and error prone to design
- Not portable across architectures, microarchitectures, and environment
- The expertise for microbenchmark design is limited to a few designers
# MicroProbe: An *Automated* Microbenchmark Generator

## What is it?

- A utility for people building microprocessors
- A code generation framework that enables *advanced* microbenchmark generation methodologies, e.g.:
  - Microarchitectural-aware test generation
  - Pre-silicon and post-silicon tests for power, resilience, and performance
  - New research innovations
- The internal IBM microbenchmark generator for IBM Power Systems and IBM z Systems [1, 2]
- A project led by **Ramon Bertran** [3, 4]

---


MicroProbe: A User’s View

What about...
- A loop for each instruction?
- A loop with a given instruction distribution?
- A loop with certain memory activity?
- Inductive noise at a given resonant frequency?

Figure: Example use case of a hardware designer wanting to write targeted, microarchitecturally aware benchmarks
MicroProbe decouples all aspects

- Target Definition using YAML
  - Architecture (ISA)
  - Microarchitecture (ISA implementation)
  - Environment (Operating System, Execution context)
- Code Generation using Python
- Generation policies using Python
Generic Microbenchmark Generation is Difficult...

MicroProbe decouples all aspects

- Target Definition using YAML
  - Architecture (ISA)  Necessary work for the RISC-V port
  - Microarchitecture (ISA implementation)
  - Environment (Operating System, Execution context)
- Code Generation using Python
- Generation policies using Python
Figure: MicroProbe generation framework overview: ISA, architecture, and environment definitions describe a target system. User-defined Generation Policies are used to drive microbenchmark generation utilities that may or may not rely on external tools for additional information.
Figure: The composition of a MicroProbe ISA definition: instructions, instruction formats, instruction fields, operands, registers, and register types. This is automatically parsed from the rv8 simulator [1].

Figure: The composition of a MicroProbe ISA definition: instructions, instruction formats, instruction fields, operands, registers, and register types. This is automatically parsed from the rv8 simulator [1].

RISC-V Target ISA Definition

Figure: The composition of a MicroProbe ISA definition: instructions, instruction formats, instruction fields, operands, registers, and register types. This is automatically parsed from the rv8 simulator [1].

Figure: The composition of a MicroProbe ISA definition: instructions, instruction formats, instruction fields, operands, registers, and register types. This is automatically parsed from the rv8 simulator [1].

**Figure:** The composition of a MicroProbe ISA definition: instructions, instruction formats, instruction fields, operands, registers, and register types. This is automatically parsed from the rv8 simulator [1].

**Figure:** The composition of a MicroProbe ISA definition: instructions, instruction formats, instruction fields, operands, registers, and register types. This is automatically parsed from the rv8 simulator [1].

Figure: The composition of a MicroProbe ISA definition: instructions, instruction formats, instruction fields, operands, registers, and register types. This is automatically parsed from the rv8 simulator [1].

Usage of an Intermediate Representation (IR) of tests

- Decouples microbenchmark description from code generation
- Any backend can then be targeted, e.g., C, assembly, raw binary

User approach to code generation

- User defines a *generation policy*—a sequence of compiler-like passes
- An example policy:
  1. Add a building block of 1000 instructions
  2. Fill the building block with instructions using the floating point unit (FPU)
  3. All instructions accessing memory should only hit the L1
  4. Set operands to avoid instruction dependencies
Example

Approach

1. Start with a Question
   - *What are the latencies and throughputs of RISC-V instructions in Rocket?*
   - Or: *Can I reverse engineer the Rocket implementation?*

2. Write a policy
   - *Loop over all instructions while varying dependency distance*

3. Emit benchmarks
   - *MicroProbe handles this for you . . .*

4. Run benchmarks

5. Interpret results
Example: Dependency Distance Policy

Figure: Example benchmark generation via a user-written policy. The user describes a microbenchmark as transforms over an intermediate representation (IR) for describing benchmarks.
Example: Dependency Distance Policy

target = import_definition("riscv_v22-risc_v_generic-riscv64_linux_gcc")

Figure: Example benchmark generation via a user-written policy. The user describes a microbenchmark as transforms over an intermediate representation (IR) for describing benchmarks.
Example: Dependency Distance Policy

target = import_definition("riscv_v22-risc_v_generic-riscv64_linux_gcc")

instrs = ['ADD_V0', 'DIV_V0', 'MUL_V0', 'FADD.S_V0', 'FDIV.S_V0', 'FMUL.S_V0', 'FADD.D_V0', 'FDIV.D_V0', 'FMUL.D_V0']

loopCount, loopSize, distances = 100, 100, [1, 2, 3, 4, 5]
reserved_registers = ['x1', 'x2', 'x3', 'x4', 'x8', 'x30']

Figure: Example benchmark generation via a user-written policy. The user describes a microbenchmark as transforms over an intermediate representation (IR) for describing benchmarks.
Example: Dependency Distance Policy

target = import_definition("riscv_v22-risc_v_generic-riscv64_linux_gcc")

instrs = ['ADD_V0', 'DIV_V0', 'MUL_V0',
          'FADD.S_V0', 'FDIV.S_V0', 'FMUL.S_V0',
          'FADD.D_V0', 'FDIV.D_V0', 'FMUL.D_V0']

loopCount, loopSize, distances = 100, 100, [1, 2, 3, 4, 5]
reserved_registers = ['x1', 'x2', 'x3', 'x4', 'x8', 'x30']

for instr in filter(lambda x: x.name in instrs, target.isa.instructions.values()):
    for d in distances:
        cwrapper = microprobe.code.get_wrapper("CLoopGen")
        synth = microprobe.code.Synthesizer(target, cwrapper(loopCount))

Figure: Example benchmark generation via a user-written policy. The user describes a microbenchmark as transforms over an intermediate representation (IR) for describing benchmarks.
Example: Dependency Distance Policy

```python
target = import_definition("riscv_v22-risc_v_generic-riscv64_linux_gcc")

instrs = ['ADD_V0', 'DIV_V0', 'MUL_V0',
          'FADD.S_V0', 'FDIV.S_V0', 'FMUL.S_V0',
          'FADD.D_V0', 'FDIV.D_V0', 'FMUL.D_V0']

loopCount, loopSize, distances = 100, 100, [1, 2, 3, 4, 5]
reserved_registers = ['x1', 'x2', 'x3', 'x4', 'x8', 'x30']

for instr in filter(lambda x: x.name in instrs, target.isa.instructions.values()):
    for d in distances:
        cwrapper = microprobe.code.get_wrapper("CLoopGen")
        synth = microprobe.code.Synthesizer(target, cwrapper(loopCount))

        passes = [microprobe.passes.structure.SimpleBuildingBlockPass(loopSize),
                  microprobe.passes.instruction.SetRandomInstructionTypePass([instr]),
                  microprobe.passes.initialization.ReserveRegistersPass(reserved_registers),
                  microprobe.passes.register.DefaultRegisterAllocationPass(dd=d)]

        for p in passes:
            synth.add_pass(p)

bench = synth.synthesize()
synth.save("build/" + instr.name + "_" + str(d) + ".c", bench=bench)
```

**Figure:** Example benchmark generation via a user-written policy. The user describes a microbenchmark as transforms over an intermediate representation (IR) for describing benchmarks.
Example: Dependency Distance Policy

target = import_definition("riscv_v22-risc_v_generic-riscv64_linux_gcc")

instrs = ['ADD_V0', 'DIV_V0', 'MUL_V0',
          'FADD.S_V0', 'FDIV.S_V0', 'FMUL.S_V0',
          'FADD.D_V0', 'FDIV.D_V0', 'FMUL.D_V0']

loopCount, loopSize, distances = 100, 100, [1, 2, 3, 4, 5]
reserved Registers = ["x1", "x2", "x3", "x4", "x8", "x30"]

for instr in filter(lambda x: x.name in instrs, target.isa.instructions.values()):
    for d in distances:
        cwrapper = microprobe.code.get_wrapper("CLoopGen")
        synth = microprobe.code.Synthesizer(target, cwrapper(loopCount))

        passes = [microprobe.passes.structure.SimpleBuildingBlockPass(loopSize),
                  microprobe.passes.instruction.SetRandomInstructionTypePass([instr]),
                  microprobe.passes.initialization.ReserveRegistersPass(reserved_registers),
                  microprobe.passes.register.DefaultRegisterAllocationPass(dd=d) ]

        for p in passes:
            synth.add_pass(p)

        bench = synth.synthesize()
        synth.save('build/' + instr.name + '_' + str(d) + '.c', bench=bench)

Figure: Example benchmark generation via a user-written policy. The user describes a microbenchmark as transforms over an intermediate representation (IR) for describing benchmarks.
Example: Dependency Benchmarks

Figure: Generating, compiling, and running microbenchmarks
Example: Dependency Benchmarks

> ./risc-v_ipc.py

Figure: Generating, compiling, and running microbenchmarks
Example: Dependency Benchmarks

```plaintext
> ./risc-v_ipc.py
> ls build/* | xargs -n5 | column -s ' ' -t
build/ADD_V0_1.c build/ADD_V0_2.c build/ADD_V0_3.c build/ADD_V0_4.c build/ADD_V0_5.c
build/DIV_V0_1.c build/DIV_V0_2.c build/DIV_V0_3.c build/DIV_V0_4.c build/DIV_V0_5.c
build/FADD.D_V0_1.c build/FADD.D_V0_2.c build/FADD.D_V0_3.c build/FADD.D_V0_4.c build/FADD.D_V0_5.c
build/FADD.S_V0_1.c build/FADD.S_V0_2.c build/FADD.S_V0_3.c build/FADD.S_V0_4.c build/FADD.S_V0_5.c
build/FDIV.D_V0_1.c build/FDIV.D_V0_2.c build/FDIV.D_V0_3.c build/FDIV.D_V0_4.c build/FDIV.D_V0_5.c
build/FDIV.S_V0_1.c build/FDIV.S_V0_2.c build/FDIV.S_V0_3.c build/FDIV.S_V0_4.c build/FDIV.S_V0_5.c
build/FMUL.D_V0_1.c build/FMUL.D_V0_2.c build/FMUL.D_V0_3.c build/FMUL.D_V0_4.c build/FMUL.D_V0_5.c
build/FMUL.S_V0_1.c build/FMUL.S_V0_2.c build/FMUL.S_V0_3.c build/FMUL.S_V0_4.c build/FMUL.S_V0_5.c
build/MUL_V0_1.c build/MUL_V0_2.c build/MUL_V0_3.c build/MUL_V0_4.c build/MUL_V0_5.c
```

Figure: Generating, compiling, and running microbenchmarks
Example: Dependency Benchmarks

```bash
> ./risc-v_ipc.py
> ls build/*.c | xargs -n5 | column -s ' ' -t
build/ADD_V0_1.c  build/ADD_V0_2.c  build/ADD_V0_3.c  build/ADD_V0_4.c  build/ADD_V0_5.c
build/DIV_V0_1.c  build/DIV_V0_2.c  build/DIV_V0_3.c  build/DIV_V0_4.c  build/DIV_V0_5.c
build/FADD.D_V0_1.c  build/FADD.D_V0_2.c  build/FADD.D_V0_3.c  build/FADD.D_V0_4.c  build/FADD.D_V0_5.c
build/FADD.S_V0_1.c  build/FADD.S_V0_2.c  build/FADD.S_V0_3.c  build/FADD.S_V0_4.c  build/FADD.S_V0_5.c
build/FDIV.D_V0_1.c  build/FDIV.D_V0_2.c  build/FDIV.D_V0_3.c  build/FDIV.D_V0_4.c  build/FDIV.D_V0_5.c
build/FDIV.S_V0_1.c  build/FDIV.S_V0_2.c  build/FDIV.S_V0_3.c  build/FDIV.S_V0_4.c  build/FDIV.S_V0_5.c
build/FMUL.D_V0_1.c  build/FMUL.D_V0_2.c  build/FMUL.D_V0_3.c  build/FMUL.D_V0_4.c  build/FMUL.D_V0_5.c
build/FMUL.S_V0_1.c  build/FMUL.S_V0_2.c  build/FMUL.S_V0_3.c  build/FMUL.S_V0_4.c  build/FMUL.S_V0_5.c
build/MUL_V0_1.c  build/MUL_V0_2.c  build/MUL_V0_3.c  build/MUL_V0_4.c  build/MUL_V0_5.c

> ls build/*.c | xargs -n1 -IX riscv64-unknown-elf-gcc X -o X.rv
```

Figure: Generating, compiling, and running microbenchmarks
Example: Dependency Benchmarks

```
> ./risc-v_ipc.py

> ls build/*.c | xargs -n5 | column -s ' ' -t
build/ADD_V0_1.c  build/ADD_V0_2.c  build/ADD_V0_3.c  build/ADD_V0_4.c  build/ADD_V0_5.c
build/DIV_V0_1.c  build/DIV_V0_2.c  build/DIV_V0_3.c  build/DIV_V0_4.c  build/DIV_V0_5.c
build/FADD.D_V0_1.c build/FADD.D_V0_2.c build/FADD.D_V0_3.c build/FADD.D_V0_4.c build/FADD.D_V0_5.c
build/FADD.S_V0_1.c build/FADD.S_V0_2.c build/FADD.S_V0_3.c build/FADD.S_V0_4.c build/FADD.S_V0_5.c
build/FDIV.D_V0_1.c build/FDIV.D_V0_2.c build/FDIV.D_V0_3.c build/FDIV.D_V0_4.c build/FDIV.D_V0_5.c
build/FDIV.S_V0_1.c build/FDIV.S_V0_2.c build/FDIV.S_V0_3.c build/FDIV.S_V0_4.c build/FDIV.S_V0_5.c
build/FMUL.D_V0_1.c build/FMUL.D_V0_2.c build/FMUL.D_V0_3.c build/FMUL.D_V0_4.c build/FMUL.D_V0_5.c
build/FMUL.S_V0_1.c build/FMUL.S_V0_2.c build/FMUL.S_V0_3.c build/FMUL.S_V0_4.c build/FMUL.S_V0_5.c
build/MUL_V0_1.c  build/MUL_V0_2.c  build/MUL_V0_3.c  build/MUL_V0_4.c  build/MUL_V0_5.c

> ls build/*.c | xargs -n1 -IX riscv64-unknown-elf-gcc X -o X.rv

> ls *.rv | xargs -IX -n1 emulator-freechips.rocketchip.system-DefaultConfig-debug pk X 2>&1 | tee X.log
```

Figure: Generating, compiling, and running microbenchmarks
Figure: Rocket-Chip$^1$ Instructions per Cycle (IPC) as a function of inter-instruction dependency distance

$^1$Using Default Config of 61ef560
Example: Interpreted Results

Figure: Rocket Chip per-instruction IPC

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MUL</td>
<td>4</td>
<td>0.25</td>
</tr>
<tr>
<td>DIV</td>
<td>&gt; 22</td>
<td>&lt; 1/22</td>
</tr>
<tr>
<td>FADD.S</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>FMUL.S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDIV.S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FADD.D</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>FMUL.D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDIV.D</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
Takeaways

As a user of MicroProbe . . .

- Write microbenchmark generators, not benchmarks
Open Source Availability

Caveat

- Future location: https://github.com/ibm/microprobe
- Pending ongoing legal approval...
Acknowledgments

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.