TIP: Time-Proportional Instruction Profiling

Internals of CPU Profiling with different instruction sets and architectures

Gufran Baig
#G01387617

How to evaluate CPU performance?

Basic idea behind this paper is to understand the different ways to profile the CPU performance.

- We want to know how execution time is distributed across the instructions of a program.
- Different CPU architectures and instruction sets makes it more difficult to do profiling.
- Highly parallel execution model of processors makes analysis being unnecessarily challenging.

Outline

- Introduction
- Motivation & Background
- Methodology
- Experimental Setup
- Results
- Case Study
- Conclusion

Introduction

Profilers

Profilers can be broadly divided into two categories

- **Software Level Profilers**
- **Hardware Supported Profilers**

Software Level Profilers

- Software level profilers (Linux perf) interrupt the application and retrieve the address of the instruction.
- Execution will resume from after the interrupt has been handled.
- Gathering profiles software level profilers is inherently inaccurate

Hardware Supported Profilers

- Hardware-supported profiling enables sampling in-flight instructions without interrupting the application.
- All hardware profilers rely on sampling.
- Profiler collects instruction address at regular time intervals.
- Collection and their instruction selection policies differ.

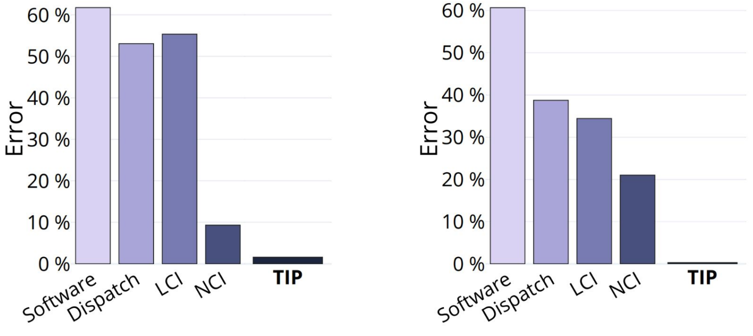
Hardware Supported Profilers

Sample collection policies for hardware based profiles

- **Next Committing Instruction (NCI)** heuristic.
- **Last Committed Instruction (LCI)** heuristic.
- **Dispatch heuristic**: Tag an instruction at dispatch and then retrieve the sample when the instruction commits.

Motivation & Background

Comparing profiler error rates



(a) Average error. (b) Imagick.

Figure 1: Instruction-level profile error of state-of-the-art profilers compared to our Time-Proportional Instruction Profiler (TIP). *Existing profilers are inaccurate due to lack of ILP support and systematic latency misattribution.*

Software-level profiling:

- 61.8%

dispatch tagging heuristic:

- 53.1%

LCI-heuristic:

- 55.4%

NCI-heuristic:

- 9.3%

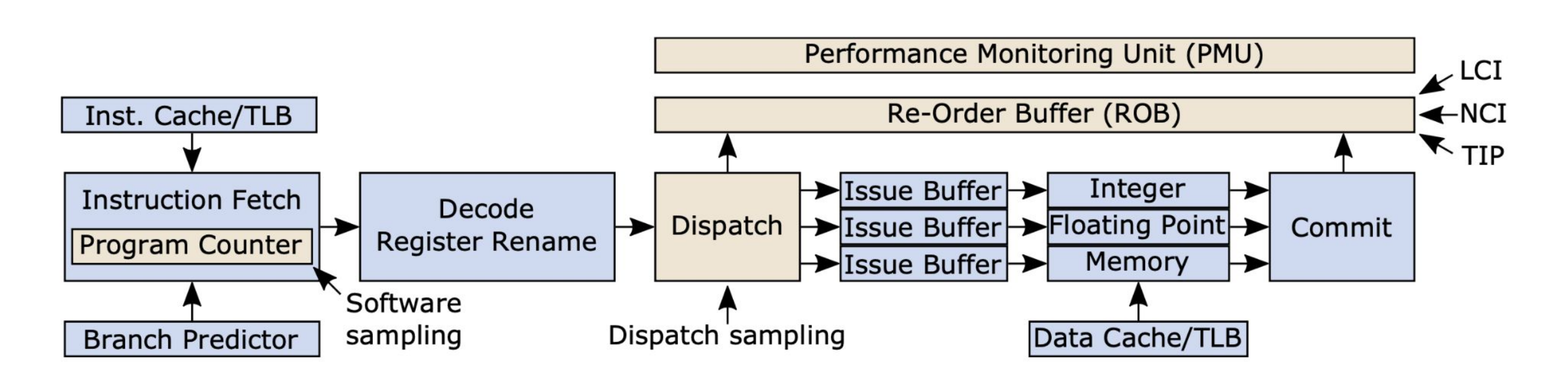
Issues with state-of-the-art profilers

Existing profilers are inaccurate due to lack of ILP support and systematic latency misattribution

- **They do not account for ILP** — incorrectly attributing the latency of co-committed instructions to only one of the instructions.
- **Suffer from systematic misattribution** — attributing the latency of processor stalls to a different instruction than the one that caused the stall.
- A CPU commits w instructions cycle — implies processor has been able to hide all of the instruction latency except for $1/w$ cycles.

Sampling the Instructions

In software based profiler, sampling is done at the dispatch, while in LCI, NCI, and TIP sampling is done at the commit phase



Issues in Dispatch and Software Profiling

- Dispatch and Software do not sample at commit phase.
- Different instructions spend more time in some pipeline stages than others.
- Time an instruction spends at the head of the ROB directly impacts the execution time.

Issues with LCI and NCI Profiling

- NCI and LCI misattribute time.
- Time spent resolving a mispredicted branch must be attributed to the branch and not some other instruction.
- NCI systematically blames the instruction after a pipeline flush for stalls due to mis-speculation.

Profiling with TIP

- TIP account for ILP at the commit stage and pipeline stall, flush and/or drain latencies.
- Every clock cycle is attributed to the instruction(s) that the processor exposes the latency of.
- The average instruction-level profile error of TIP is merely 1.6%.
- TIP reduces average error by 5.8×, 34.6×, 33.2×, and 38.6× compared to NCI, LCI, Dispatch, and Software profiling, respectively.

Methodology

Oracle Profiling (TIP)

- Oracle which is time-proportional by design.
- Oracle attributes each clock cycle during program execution to the instruction(s) which the processor exposed the latency of in this cycle.
- Oracle attributes $1/n$ clock cycles to each of the n committing instructions.
- Oracle profiling consists of four fundamental states for each clock cycle.

Oracle clock cycle attribution

- If the ROB contains instruction(s), and if the processor is committing (an) instruction(s) in this cycle. Processor is in the **Computing state**.
- If the processor is not committing instructions and there are instructions in the ROB, processor is in **Stalled state.**
- If ROB is empty due to mis- speculation, the processor is in the **Flushed state**
- If front-end is not supplying instructions, due to an instruction cache or instruction TLB miss, the processor is in the **Drained state**

Oracle clock cycle attribution

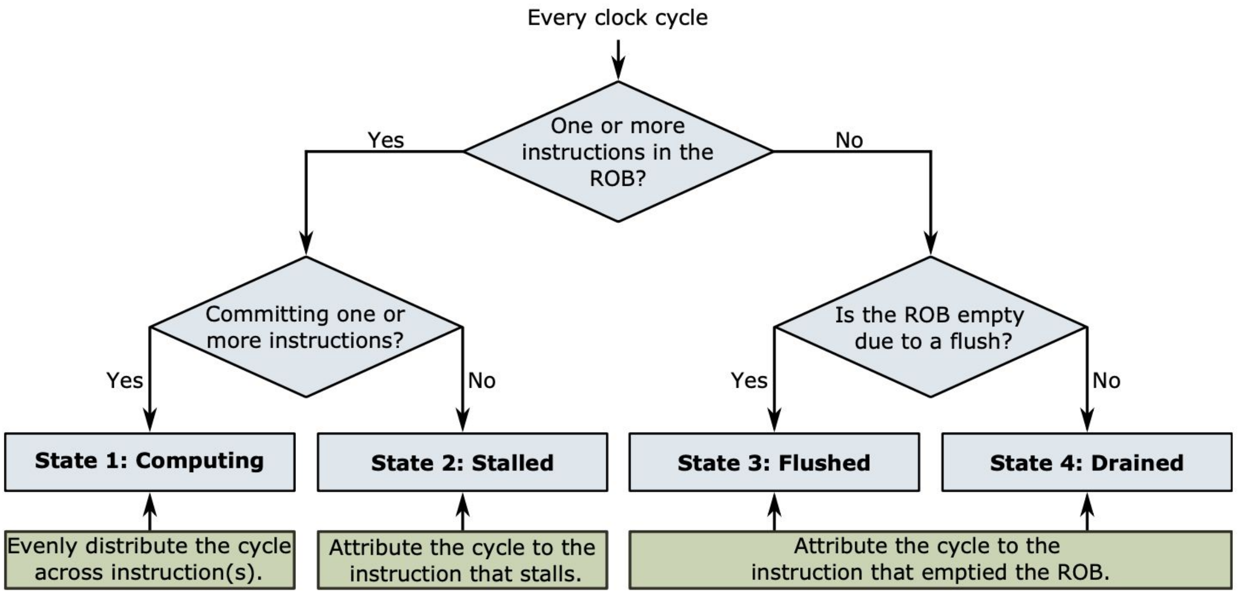
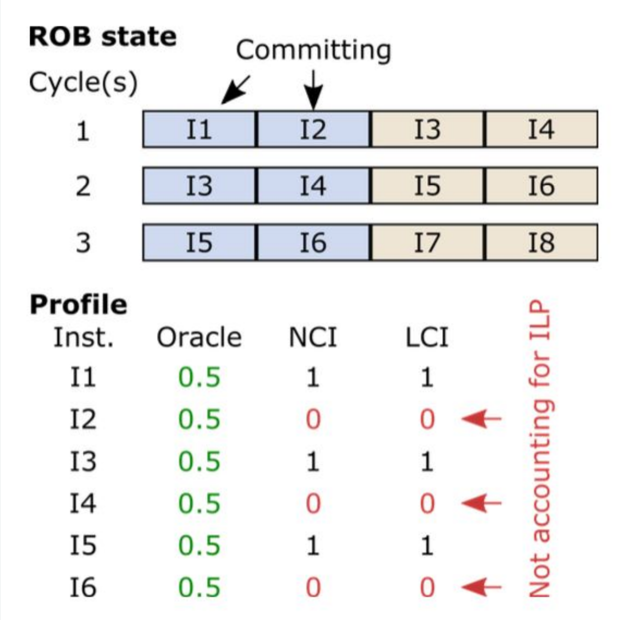


Figure 3: Oracle profiler clock cycle attribution overview.

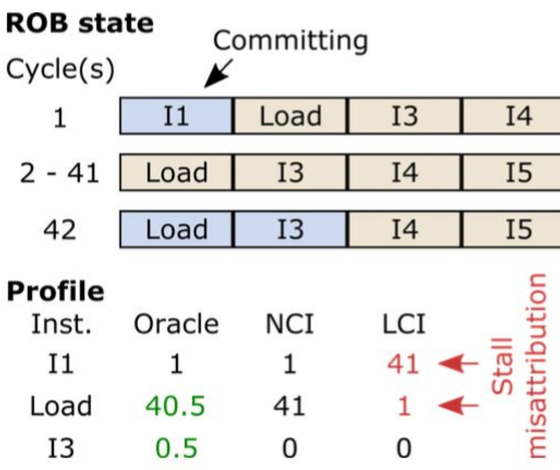
Computing State



(a) Computing.

- Oracle accounts $1/n$ cycles to each committed instruction where n is the number of instructions committed in that cycle.
- In cycle 1, instructions I1 and I2 are committing and Oracle hence accounts 0.5 cycles to both.
- In contrast, NCI and LCI select a single instruction to attribute the clock cycle to

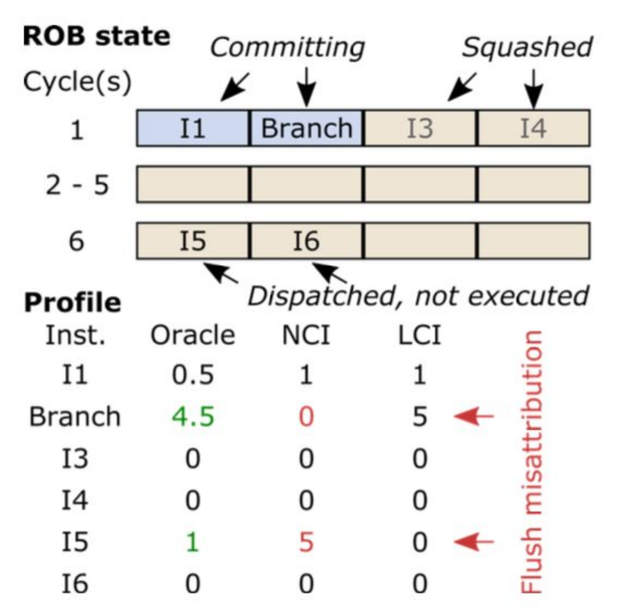
Stalled State



(b) Stalled.

- I1 is committed in cycle 1 before commit stalls for 40 cycles on the load instruction from cycle 2 to 41
- Oracle attributes the 40 cycles where the processor is stalled to the oldest instruction that the processor is stalling on
- LCI, completely misattributed the load stall as I1 is the last-committed instruction from cycle 1 to cycle 41

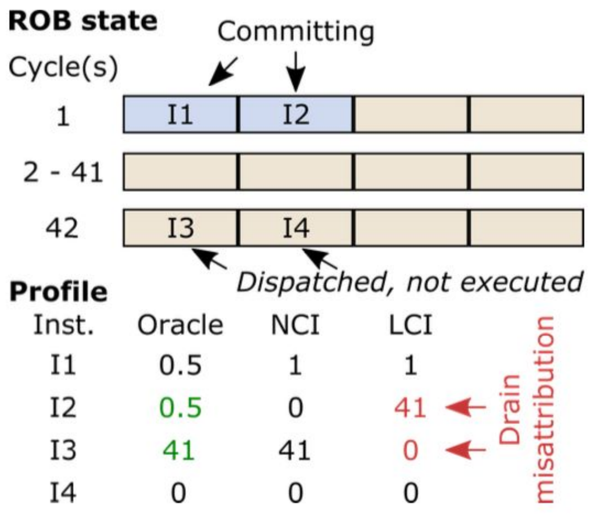
Flushed State



(c) Flushed.

- Branch misprediction lead to the ROB being empty for 3.5 cycles on average.
- LCI correctly attributes the stall cycles to the mispredicted branch whereas NCI does not.
- NCI attributes the empty ROB cycles to I5 as it will be the next instruction to commit
- NCI attributes zero cycles to the branch instruction since it is committed in parallel with I1.

Drained State



(d) Drained.

- In cycle 1, I1 and I2 are committed. This leaves the ROB empty until cycle 42.
- Oracle attributes 41 cycles to I3;
- 40 cycles is due to the drain and one cycle is attributed because I3 is stalled at the head of the ROB in cycle 42.
- LCI misattributed the empty ROB cycles to I2.

Putting all together

- We have discussed the four fundamental states of the commit stage independently.
- Instructions often accumulate cycles across multiple states.
- Oracle will account time to the preceding instructions according to the time they spend at the head of the ROB.

Oracle Structural overview

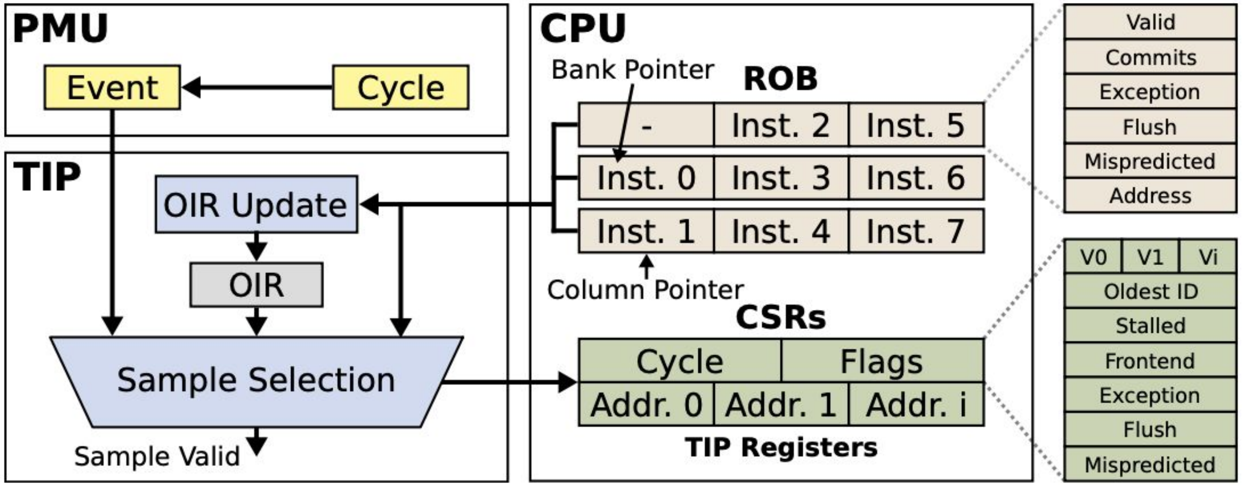


Figure 5: Structural overview of our Time-Proportional Instruction Profiler (TIP). *TIP is triggered by the PMU, collects a sample, and finally exposes the sample to software.*

Experimental Setup

Simulator

- We use the FireSim cycle-accurate FPGA-accelerated full-system simulator to evaluate the different performance profiling strategies.
- The simulated model uses the BOOM 4-way superscalar out-of-order core.
- Hardware profilers are enabled when the system boots and profile until the system shuts down.
- A modified version of FireSim is used to trace instruction address and the valid, commit, exception, flush, and mispredicted flags.

Simulated Configuration

Part	Configuration
Core	OoO BOOM: RV64IMAFDCSUX @ 3.2 GHz
Front-end	8-wide fetch, 32-entry fetch buffer, 4-wide decode, 28 KB TAGE branch predictor, 40-entry fetch target queue, max 20 outstanding branches
Execute	128-entry ROB, 128 int/fp physical registers, 24-entry dual-issue MEM queue, 40-entry 4-issue INT queue, 32-entry dual-issue FP queue
LSU	32-entry load/store queue
L1	32 KB 8-way I-cache, 32 KB 8-way D-cache w/ 8 MSHRs, next-line prefetcher from L2
L2/LLC	512 KB 8-way L2 w/ 12 MSHRs, 4 MB 8-way LLC w/ 8 MSHRs
TLB	Page Table Walker, 32-entry fully-assoc L1 D-TLB, 32-entry fully-assoc L1 I-TLB, 512-entry direct-mapped L2 TLB
Memory	16 GB DDR3 FR-FCFS quad-rank, 25.6 GB/s maximum bandwidth, 14-14-14 (CAS-RCD-RP) latencies @ 1 GHz, 8 queue depth, 32 max reads/writes
OS	Buildroot, Linux 5.7.0

Benchmarks

Cycle stacks is used to classify benchmarks

- Benchmark is classified as **Compute-Intensive** if it spends more than 50% of its execution time committing instructions.
- Benchmark spends more than 3% of its time on pipeline flushing, the benchmark is classified as Flush-Intensive.
- Rest of the benchmarks are classified as Stall-Intensive as they spend a major fraction of their execution time on processor stalls.

Benchmarks

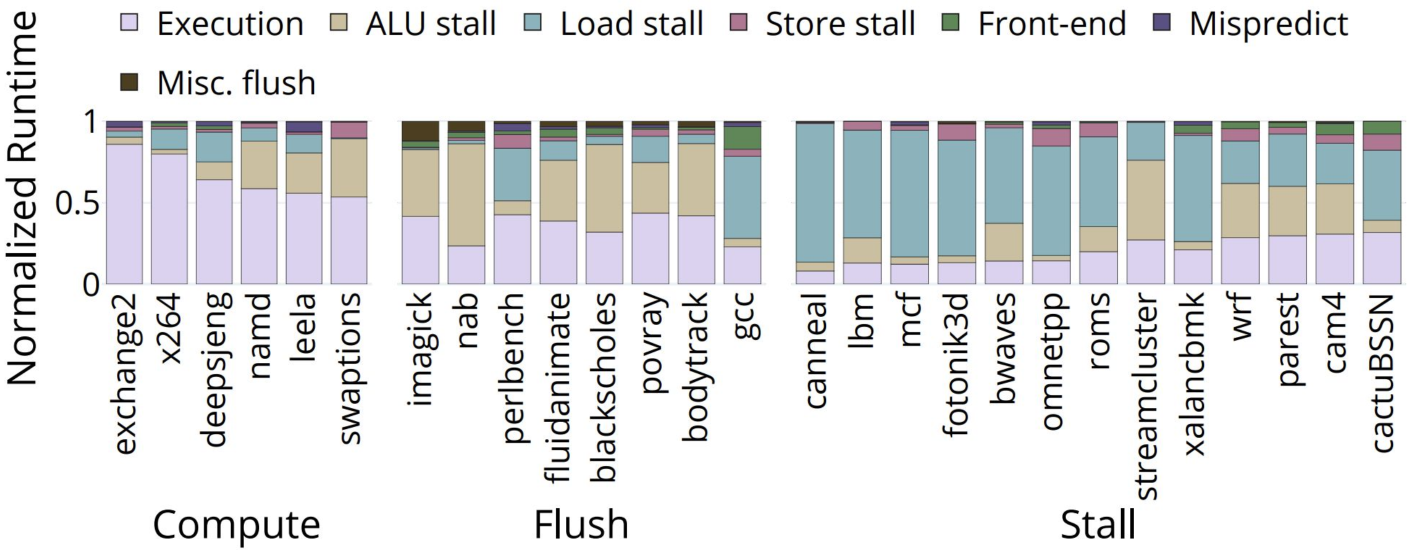


Figure 7: Normalized cycle stacks collected at commit.

Results

Comparing profiler errors

Profiling can be done at different level of granularity

- **Function-level profiling** reports error at the function level across all the profilers considered.
- **Basic-block-level profiling** Correctly attributing samples to functions does not necessarily mean that a performance analyst will be able to identify the most performance-critical basic blocks.
- **Instruction-level profiling** Performance analysts need profiling information that is even more detailed than the basic block (and function).

Function-level profiling

- All profilers, except Software and Dispatch, are accurate at function-level granularity.
- Software and Dispatch are inherently inaccurate, because tagging instructions at fetch and dispatch creates significant bias

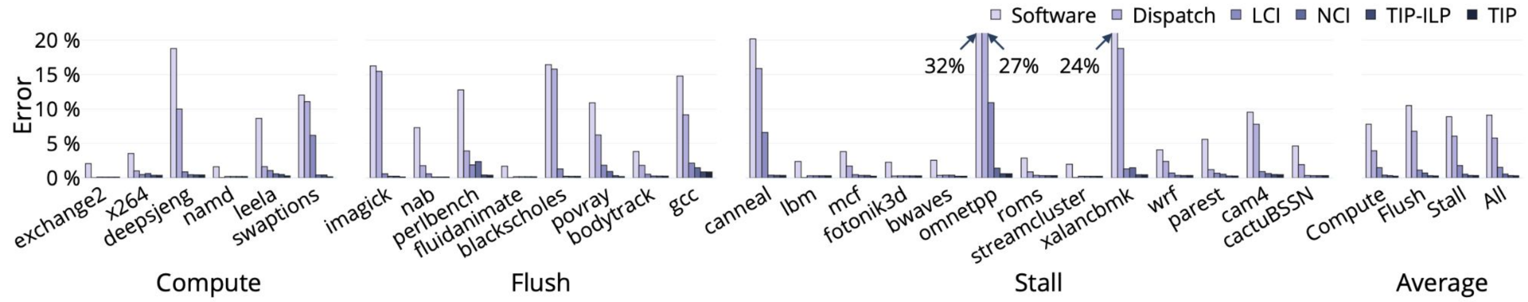


Figure 8: Function-level errors for the different profilers. *TIP, TIP-ILP, NCI, and LCI are accurate at the function level, while Software and Dispatch are largely inaccurate.*

Basic-block-level profiling

- LCI is inaccurate with an average error of 11.9% and up to 56.1% because it incorrectly attributes stalls on long-latency instructions
- Error is higher at the basic block level compared to the function level for all profilers

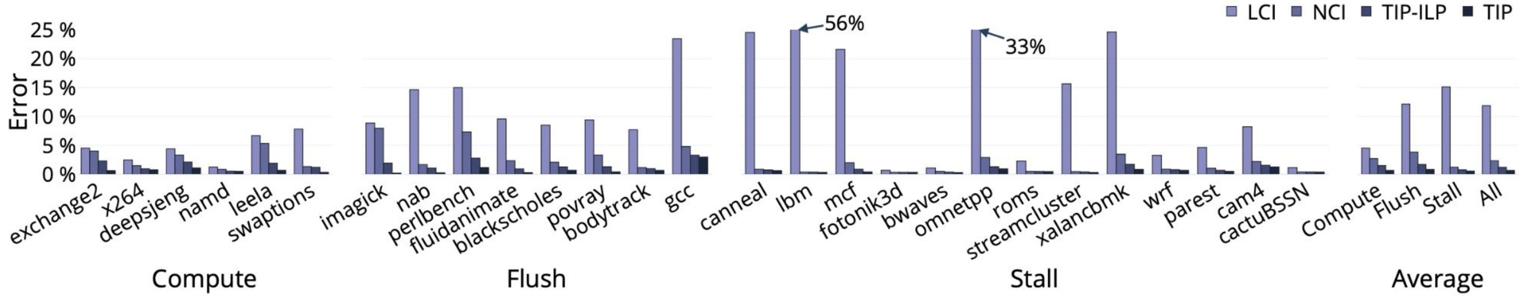


Figure 9: Basic-block-level errors for the different profilers. (Software and Dispatch are not shown because of their high error.) *TIP, TIP-ILP, and NCI are accurate at the basic block level, whereas LCI (and Software and Dispatch) are not.*

Instruction-level profiling

- TIP is the only accurate profiler at the instruction level, average profile error for TIP equals 1.6%
- It provides profiling information that is even more detailed than the basic block and function level profiles

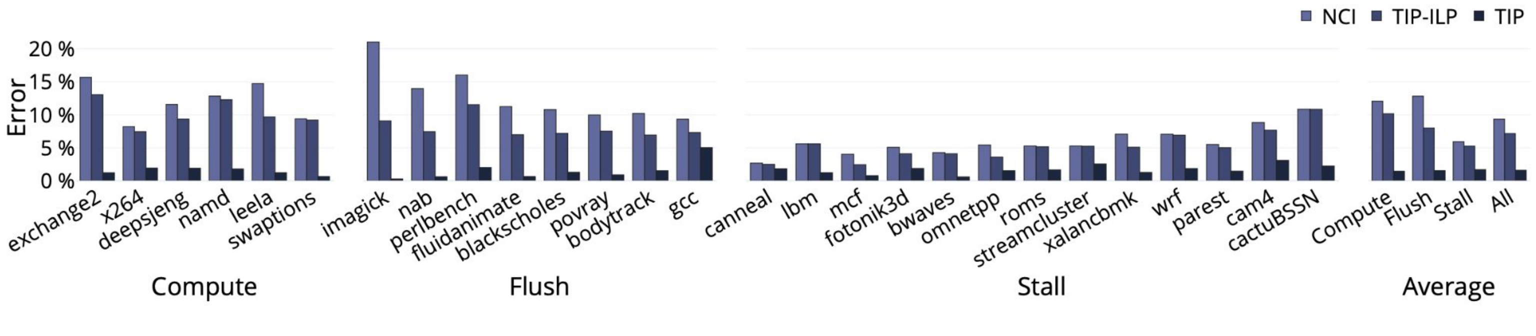
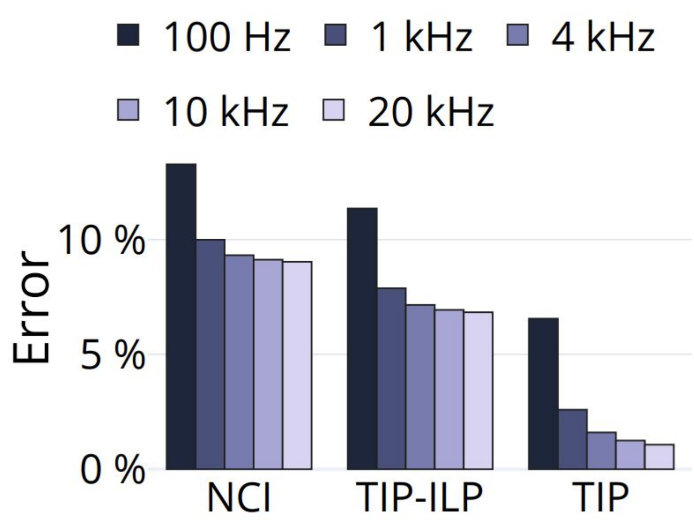


Figure 10: Instruction-level errors for the different profilers. (Software, Dispatch, and LCI are omitted because of their large errors.) *TIP is the only accurate profiler at the instruction level.*

Sampling Rate

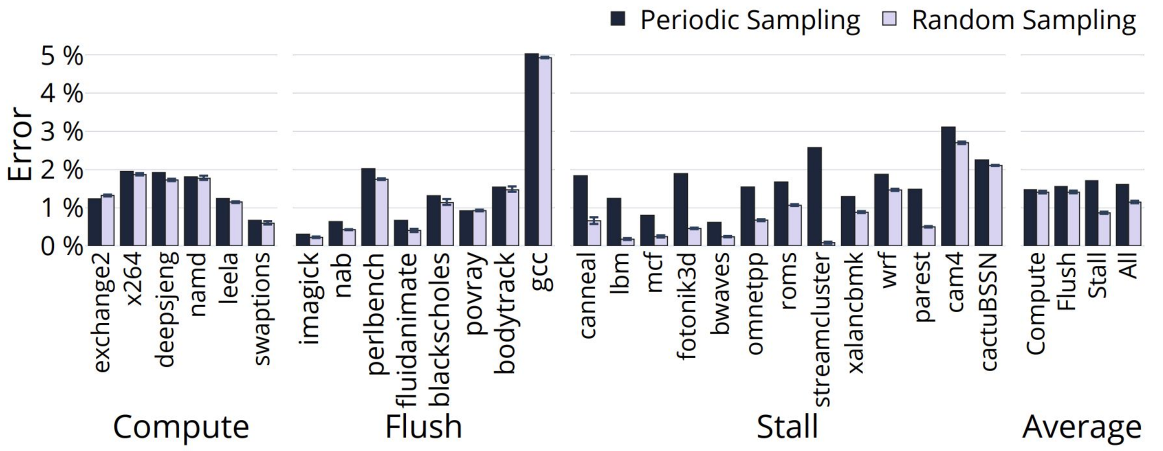


(a) Sampling frequency.

- Profiling error decreases with increasing sampling frequency; and this is true for all profilers.
- lower frequencies as these have more unsystematic error.
- TIP's accuracy continues to measurably improve as the sampling frequency is increased beyond 4 KHz

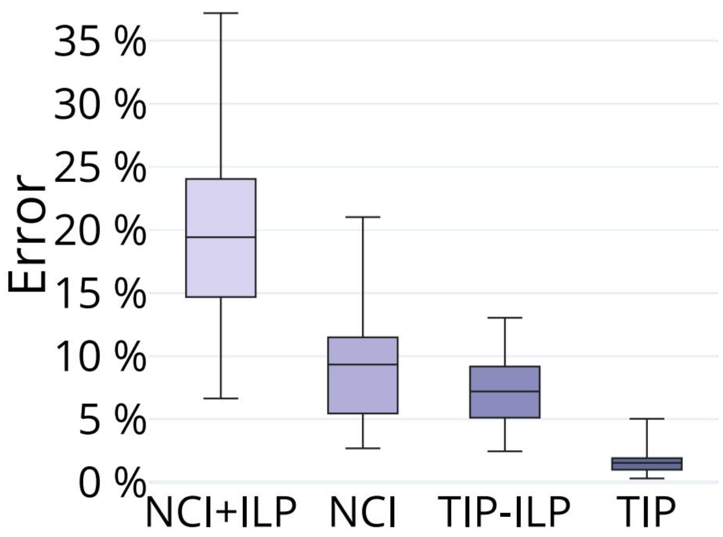
Sampling method

- Periodic sampling is only slightly more inaccurate than random sampling.
- Periodic sampling is simpler to implement in hardware as compared to random sampling.



(b) Periodic versus random sampling.

Commit-parallelism-aware NCI



(c) Parallelism-aware NCI+ILP

- TIP is more accurate than NCI.
- It correctly accounts for pipeline flushes and commit parallelism.
- Making NCI commit parallelism aware increases profile error, in contrast to TIP

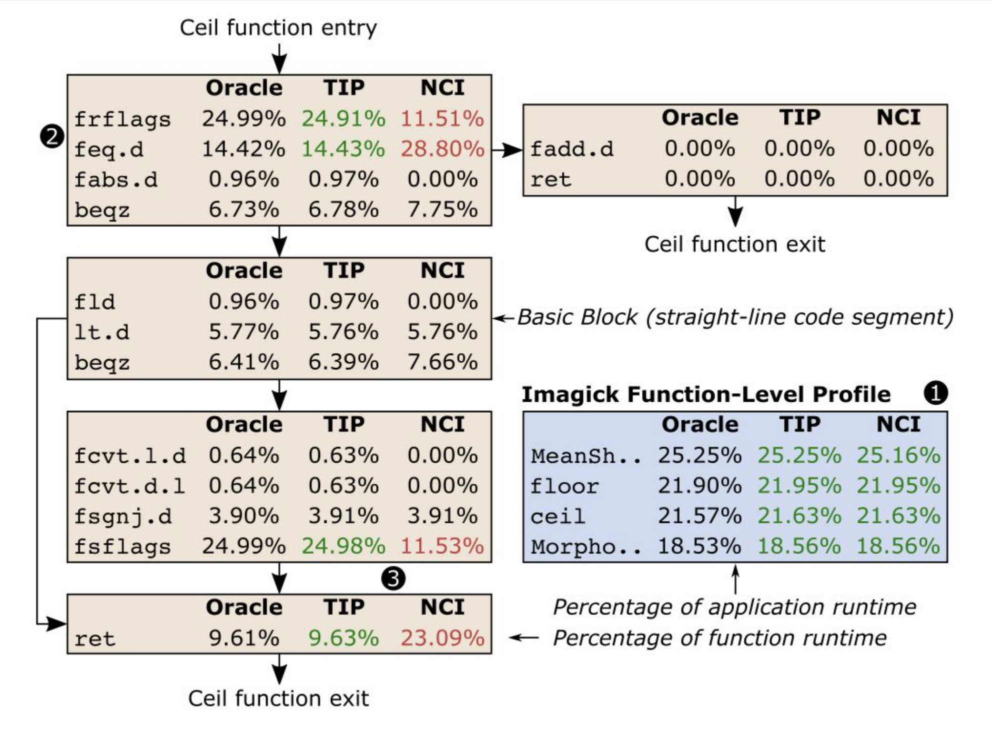
Case Study

SPEC CPU2017 benchmark Imagick

Perform SPEC CPU2017 benchmark Imagick to illustrate how TIP pinpoints the root cause of performance issues.

- The function-level profile does not clearly identify any performance problem.
- The instruction-level NCI profile attributes most of the execution time to the *feq.d* and the *ret* instructions.
- TIP correctly reports that most of the time in ceil is spent on the *frflags* and *fsflags* instructions.

SPEC CPU2017 benchmark Imagick



Optimized Imagick benchmarks

Perform SPEC CPU2017 benchmark with optimized Imagick

- Both ceil and floor spend significant time on ALU stalls and front-end stalls.
- Optimized Imagick's code by replacing *frflags* and *fsflags* in *ceil* and *floor* with *nop* instructions to remove the unnecessary status register operations.
- Optimized version improves performance by 1.93× compared to the original version.
- Speedup is based on the fraction of time spent executing the *frflags* and *fsflags* instructions.

Optimized Imagick benchmarks

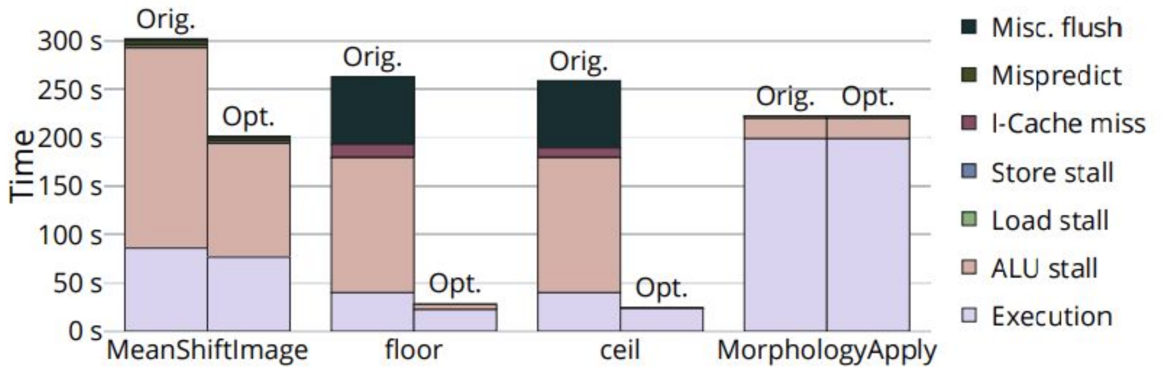


Figure 13: Time breakdown for the four most runtime-intensive functions in Imagick **comparing the original to our optimized version.** *The 1.93× speed-up is primarily due improved processor utilization.*

Conclusion

- Existing profilers fall short because they they lack ILP support and systematically misattribute instruction latencies.
- TIP account for ILP at the commit stage and pipeline stall, flush and/or drain latencies.
- TIP is highly accurate profiler, average instruction-level profile error of TIP is merely 1.6%

Questions ?

Thank You :)

References :)

- https://doi.org/10.1145/3466752.3480058%20MICRO%202021