Evaluating SmallBoom and LargeBoom CPU performance with EEMBC:CoreMark

Lalit Prasad Peri
ECE5504 Fall2023
Virginia Polytechnique Institue and
State University, Blacksburg
lalitprasad@vt.edu

Abstract— Synthetic microbenchmarks like Dhrystone have long been leveraged by CPU architects to fine-tune micro-architecture performance but it is shown to be susceptible to gaming by compiler optimizations to gain performance without any tangible hardware changes to the CPU pipeline. Also, as it doesn't include its library and has vague score reporting, the need for other portable benchmarks like EEBMC CoreMark was necessitated.

This study evaluates performance characteristics of RISC-V CPU Architecture with aid of CoreMark Benchmark. While we study compiler primitives to isolate hardware optimizable features in CPU micro-architecture, this study also focuses on the optimization of these design parameters to arrive at a "golden" configuration that will be best suited for single-threaded workloads that display similar characteristics as CoreMark.

This study replicates these experiments on two different classes of CPU architectures – SmallBoom, which represents power-efficient micro-controller class of applications, and LargeBoom which represents highly-performant compute and server class of applications.

As a result of this analysis, it was found the CoreMark scores remain agonistic to gcc riscv compiler driven optimizations, with best CoreMark score obtained with -O2 optimizations. Also, highest scores obtained was 24.4 CoreMark/MHz for LargeBoom and 8.91 CoreMark/MHz for SmallBoom CPUs, which is uplift of over 2.35x over baseline RISC-V configurations.

Keywords—*RISCV, CPU, EEMBC, CoreMark*

I. Introduction

Antiquated synthetic benchmarks like Dhrystone were highly susceptible to compiler and library optimizations, which would artificially boost DMIPS scores (Dhrystone million instructions per second) without any tangible design optimizations. To overcome this compiler gaming, EEMBC (Embedded Microbenchmark Consortium) came up with CoreMark for CPUs which is portable yet rigorous enough to provide fair comparison for performance comparison and characterizations. Along with compilation rules, EEMBC also specifies score reporting rules, which essentially avoid efforts to game the benchmark for CPU score projection.

CoreMark, though having a small program footprint, has a set of diverse algorithms to evaluate performance – i) List processing (search and sort), ii) matrix manipulation, iii) finite state machine, and iv) CRC (cyclic redundancy checks). Benchmark was designed with 64-bit processors in view.

CoreMark benchmark is frequency agonistic as the results are reported in the number of CoreMark iterations per second per MHz. This isolates any IPC performance gains in CPU which can be attributed to frequency boost alone.

Due to the portable yet versatile characteristics of CoreMark, this project evaluates RISCV CPUs in two different configurations – i) SmallBoom and ii) LargeBoom, where the sensitivity of micro-architecture parameters like cache sizing and replacement policies, TLB sizes and policies, prefetcher, branch-prediction schemes, and instructions-scheduling.

The expected outcome of this evaluation is to demonstrate performance sensitivity for RISCV CPUs with permutations of micro-architectural parameters which are ideal for workloads that have CoreMark-like characteristics. Simulation results and simulator artifacts will be published on github[3].

II. Related work

CoreMark is widely adopted in the CPU industry and used as one of the key benchmarks to demonstrate generation-on-generation performance uplifts. This project evaluates CoreMark performance inspired by a host of studies performed on x86 and Arm CPUs but primarily from a whitepaper published by EEMBC [1].

III. proposed method

This in study, we focus on two methods of evaluating CoreMark performance i) static analysis ii) dynamic analysis.

Static Analysis primarily focuses on program analysis using gcc-riscv compiler optimizations [8], with various combinations of optimization parameters that can vary the instruction mix and hence the nature of the workload itself. For example, a mix of 32-bit and 64-bit instructions, loop-unrolling, static-branch predictions, and register renaming, static instruction scheduling link tree optimizations, etc.

Dynamic analysis focuses on simulating and observing the sensitivity of benchmarks with micro-architecture parameter variations. This study proposes to use chip-yard project [7] which instantiates scala-models for CPUs and supports LLC and main memory, with parameterizable characteristics.

IV. Evaluation Plan

The evaluation plan for static analysis is to generate disassembly for various combinations of compiler primitives using gcc-riscv on Guacamole compute clusters, without executing riscv binaries and extract interesting observations, caused by these variances in compilations.

The evaluation plan for dynamic analysis is carried out simulation using verilator simulator using chipyard-clean [7] scala-base classes to develop and extend the configurations for SmallBoom and LargeBoom Configurations. In these simulations plan is to carry out regressions with sweeps of micro-architectural parameters analyze performance trends

and highlight underlying phenomena that might create performance bottlenecks.

These parameter sweeps and policies are listed in detail in Table. I.

Table I

Micro-architecture features	Policies and parameter sweeps
Caches	Replacement policies – LRU, SLRU, Random Sweep - Line-sizes, sets/ways
Prefetchers	NLP, NLP+N, Pointer-chase.
TLB	iTLB/dTLB sizes. PageTable Entries.
Branch-prediction	PHT/BTB sizes; SwPred, TAGE, Alpha
Instruction-Scheduling	INT/FP phy-registers, ROB size, Ldq/Stq

V. Evaluation Method

Build and compile toolchain: For static analysis, which focuses on ISA, compiler and code optimization aspects of the benchmark, this project uses gcc riscv extension riscv64-unknown-elf-gcc over baseline version of v9.4 available of ece computer server. Also, for the object analysis, it uses riscv64-unknown-elf-objdump to extract the object file. After analyzing instruction mix generated by compiler optimizations flags (-O1, -O2, -O3, -Ofast) it was found that a more complex formal object de-compilation tool - Ghidra[11] is more suitable, as it not only decompiles 16/32 bit riscv binaries more accurately but also generates detailed analysis statistic like instruction count, memory foot-print, branch-density and call-function-graphs.

Simulations and regressions setup: For dynamic simulation analysis which sweeps across the range of micro-architecture parameters, changes over baseline SmallBoomConfig and LargeBoomConfig were made. Simulating configurations over verilator simulator with embedded performance monitors both the CPU performance characteristics was profiled. These simulations were also profiled using Linux-perf to capture simulation cycle time for accurate extrapolation of benchmark scores.

VI. Evaluation Results

Static Analysis, Instruction Mix & De-compilation.

As this part of analysis targets code optimization possible by introducing incrementally aggressive compiler optimizations supported by gcc riscv extension.

O0, no-optimization – This flag generates the riscv binary which is completely unoptimized and contains the inherent code introduced by the benchmark and gcc header files.

O1, single-pass optimization – This flag optimizes the binary generated in O0 by single pass of profile guided optimization. Benchmark score is expected to be slightly better than O0 binaries, by simulation on SmallBoomConfig it is observed to be 1.24x faster.

O2/O3 – double and triple pass optimizations. These flags introduce some additional inbuilt flags like loop-unrolling, link-tree-optimization which will significantly improve the characteristics of baseline binary and also tune further to reduce branch instructions and improve register renaming. Benchmark score is expected to be significantly better than O0 binaries, by simulation on SmallBoomConfig it is observed to be 2.35x faster.

Unroll-loops – This flag optimized loops in benchmark to reduce the number of branch instructions. Though side-effect of this optimization is increase in program size. Benchmark score is expected to be better than O0 binaries, by simulation on SmallBoomConfig it is observed to be 2.48x faster.

Ofast – This flag applies for most aggressive possible combinations, over multiple passes of profile guided optimization, such that overall execution time of benchmark can be reduced. This optimization adds more instructions over baseline increasing memory footprint of the benchmark. Benchmark score is expected to be significantly better than O0 binaries, by simulation on SmallBoomConfig it is observed to be 2.66x faster.

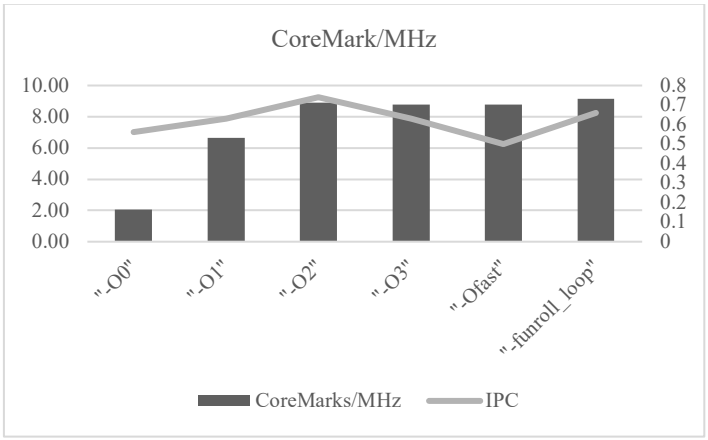

Figure 1. Coremark/MHz and IPC (instruction per cycle) for SmallBoom baseline.

Through the combination of compiler optimization, we can observe from figure 1, that beyond -O2, any further optimization passes do not lead to better CoreMark score. For further simulation and regression these binaries are leveraged.

This also concludes that CoreMark is resilient towards compiler gaming by using optimization flags.

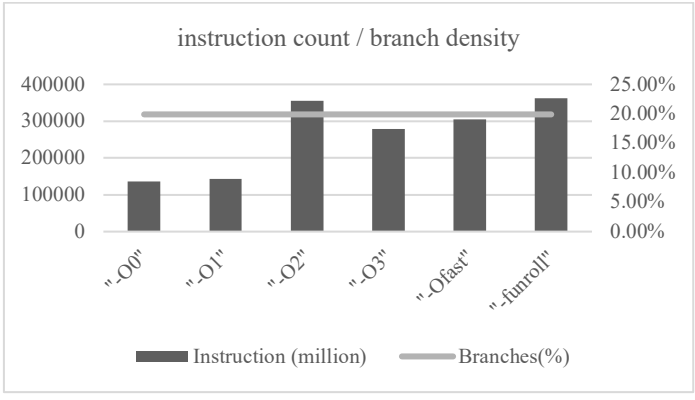

Figure 2. Coremark instruction count and branch density.

Dynamic Analysis via simulation and regression focuses on empirically analyzing the benchmark score uplifts by sweeping the micro-architecture parameters.

Firstly, by analyzing the bottlenecks by profiling the number of instruction stalls/retiring, it could be narrowed down, which parameter changes could uplift the performance significantly.

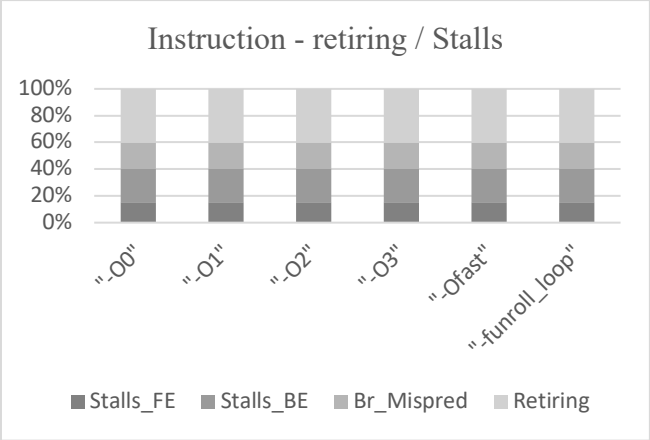

Figure 3. Top-Down Analysis of instruction stalls, SmallBoom CPU

From Figure 3. Top-Down Analysis of instruction stalls, it could be observed over 24% instruction stalls are from back-end i.e memory bound while front-end stalls are 15% and branch mis-prediction stalls are 19% of total instructions.

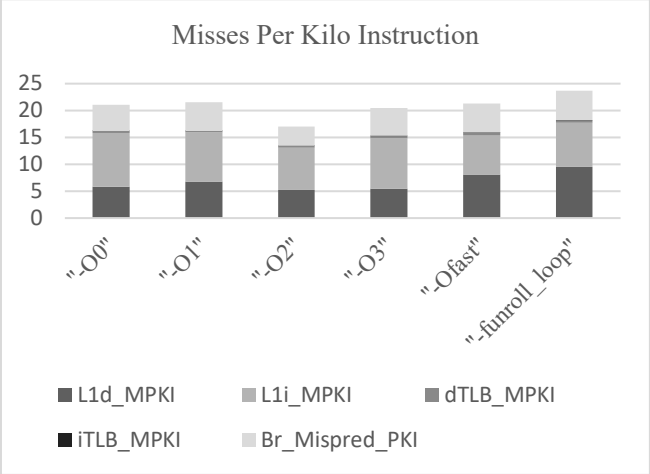

Figure 4. Miss Rates, LargeBoom CPU

Further analyzing the stalls by analyzing Misses per Kilo instructions (from hpm performance counters) from figure 4, it could be observed for SmallBoom and LargeBoom L1d and L1i cache miss rates contributes to back-end bottlenecks while Branch mis-prediction rates are identical to 19% as seen in top-down breakdown.

TLB miss rates in both SmallBoom and LargeBoom CPUs are under 1%, which can be attributed to very small data-set size of CoreMark Benchmark.

Caches configuration sweeps (Line sizes, sets/ways) could be summarized in figure 5. It could be observed that on increasing cache line size from 64B to 128B and 16Way-16Set to 32Way-8Set configurations L1d Misses per kilo instruction was reduced by 15%, 24% for LargeBoom and SmallBoom respectively. This can be attributed more localized data structures used in CoreMark.

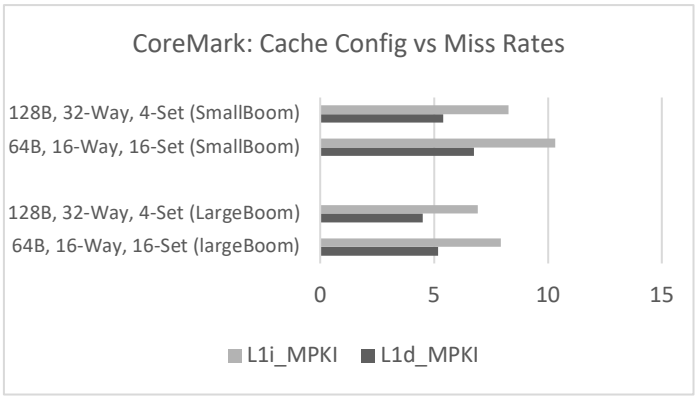

Figure 5. Cache Miss Rates for LargeBoom / SmallBoom

Address Translation (TLB sizing), as observed from the figure 4, TLB misses contribute to less than 1% of the instructions being stalled. Similar behaviour was observed in sweep simulations for 4KB, 8KB and 16KB TLB pages. Hence no significant performance uplifts were observed over baseline 4KB pages in LargeBoom and SmallBoom CPU.

Branch-Prediction, branch mis-predictions per kilo instruction seems to make up to 19% of total instruction stalls.
In these configuration analysis, three branch prediction schemes – TAGE, G-Share and Alpha predictors were compared for both CPUs. From figure 5, it can observe that G-share reduced branch mis-prediction rates by 9% and Alpha-BPD by 37% for SmallBoom CPUs. These uplifts are measured without application of loop-unrolling compiler directive.

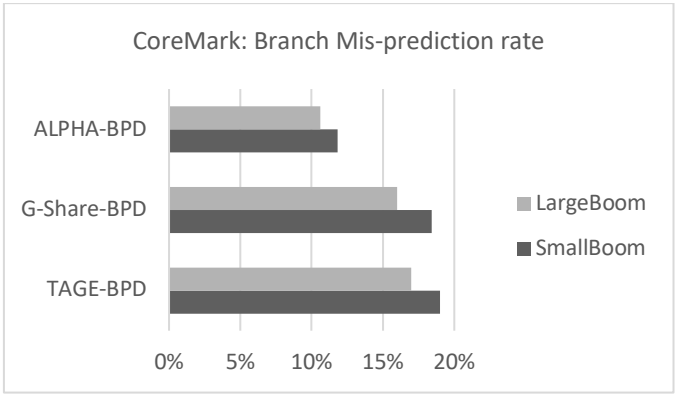

Figure 6. Branch-predictions vs Branch-miss rates LargeBoom / SmallBoom

Also, by crossing optimal branch-predictor configuration with cache sweep results, uplifts in CoreMark performance could summarized in Figure 7, CoreMarks/MHz score is uplifted from 8.9 to 9.6 from baseline SmallBoom CPU and from 14.6 to 23.36 for LargeBoom CPU.

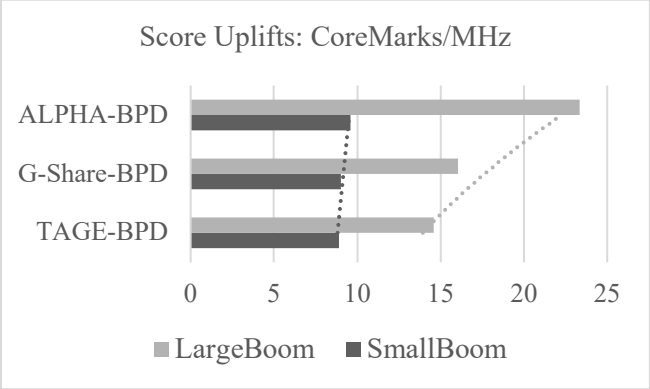

Figure 7. CoreMark/MHz score uplift with Branch-predictions. LargeBoom / SmallBoom

VII. TIMELINE & EXECUTION

Timeline for proposed study is divided into weekly milestones and planned to be completed in seven weeks.

Milestones and associated deliverables are summarized in Table II.

Table II

Milestones	Planned	Actual
Background research & simulation setup	Week1 (Nov.1)	Week1 (Nov.1)
Static Analysis (Gcc-riscv)	Week2 (Nov.8)	Week2 (Nov.8)
Simulations & Analysis (SmallBoom)	Week3 (Nov.15)	Week3 (Nov.15)
Simulations & Analysis (LargeBoom)	Week4 (Nov.22)	Week5 (Nov.22)
Simulation & Regression Analysis	Week5 (Nov.29)	**Week6 (Nov.29)
Documentation & Presentation	Week6 (Dec.6)	**Week7 (Dec.6)
Presentation & Archival	Week7 (Dec.14)	**Week7 (Dec.14)

**Challenges and workarounds to overcome them

GCC extension for riscv compilation used with gcc version 9.4 is deprecated, which posed challenges in correct interpretation of C headers such as time.h to accurate score calculations. To overcome these alternate methods like using linux-perf and embedding boom-CPU performance counters inside the base code was done, which is not ideal and not recommended by EEMBC but necessary to generate riscv executable.

Also porting the CoreMark codebase from EEMBC to be run on boom-CPU models, which do not support semi-hosting to pass runtime arguments like randomization seeds was again a challenge. This was overcome by using a static seed in benchmark code.

**Godel server outages and slowness of shared compute cluster lead to challenges in incremental compile of verilator simulator, simulation regression to sweep all the configurations of interest and preserve & archive all the artifacts related to this project.

VIII. CONCLUSION & FUTURE WORKS

This analysis proves the versatility and portability of EEMBC CoreMark for riscv CPU evaluation. Also, it can be established that unlike legacy micro-benchmarks, CoreMark is resilient to compiler driven performance gaming, with substantiated results.

Due to small instruction footprint, Coremark proves to focus on back-end (memory bound) and front-end (core-bound), evenly, which make it ideal for micro-architecture explorations like as seen on SmallBoom and LargeBoom CPUs.

Though not in purview, of this project, possible future directions could be to perform comparative analysis of contemporary micro-benchmarks like Dhrystone, Whetstone, CoreMark and also SPEC2006. Another interesting direction could be to explore EEMBC CoreMarkPro, which is a more sophisticated multi-threaded evolution of single-threaded CoreMark for system performance exploration.

References

[1] Shay Gal-On, Markus Levy, “Exploring CoreMark™ – A Benchmark Maximizing Simplicity and Efficacy” EEMB Whitepaper

[2] Jim Turley “Dhrystone Is Dead; Long Live CoreMark!”, EEJournel, June’2009

[3] CoreMark-Cortex-M4. https://developer.arm.com/Processors/Cortex-M4

[4] SimulationArtifacts: https://github.com/lalitprasadperi/Evaluating_CoreMark_on_BoomCore

[5] RISC-V Insstruction Set Architecture v2.2 https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf

[6] BOOM CPU Specification https://docs.boom-core.org/en/latest/

[7] Chipyard Project Documentation https://chipyard.readthedocs.io/en/latest/Generators/BOOM.html

[8] Gcc-riscv toolchain https://wiki.riscv.org/display/HOME/Toolchain+Projects

[9] David A Patterson and John L. Hennessy, Computer Architecture: A Quantitative Approach 5th edition.

[10] Course Slides, ECE5504 Fall2023

[11] Ghidra decompilation tool: https://ghidra-sre.org/