RISC-V Code Generation Comparison

Michael M. Pozulp and Yukio Miyasaka

Audience: RISC-V Summit 2021 Attendees

***Abstract*—We ran the SPEC CINT2006 benchmarks on the SiFive HiFive Unleashed and found that the benchmarks run up to 18% faster when we compiled with gcc than when we compiled with clang. The gcc-compiled benchmarks executed up to 31% fewer instructions and 35% fewer instruction bytes than the clang-compiled benchmarks. We also studied the effect of the RISC-V Bitmanip Extension and found that it decreased the dynamic instruction count of the benchmarks by up to 17%, which is comparable to the reduction achieved in previous work that used macro-op fusion.**

Our results indicate that 1) gcc often generates better RISC-V code than clang, 2) fewer dynamic instructions does not always mean faster runtimes, and 3) the RISC-V Bitmanip Extension is about as good as macro-op fusion at reducing dynamic instruction counts.

RISC-V toolchains are rapidly developing, so our results are a snapshot of the state as of May 2021 and future results could differ from ours.

I. Introduction

The RISC-V community is fortunate to have upstream implementations of the RV64G and RV64GC targets in both gcc and clang. Downstream implementations at UC Berkeley began as early as 2013. The downstream gcc implementation was accepted upstream and announced with the release of gcc 7.1 in 2017. Upstream development in clang (i.e. llvm) began in 2016 and was promoted from "experimental" to "official" with the release of clang 9.0.0 in 2019.

Previous work on RISC-V often employed gcc instead of clang. In the report which motived our work, researchers used gcc to compare the dynamic instruction counts and dynamic instruction bytes fetched for popular proprietary Instruction Set Architectures (ISAs) to the free and open RISC-V RV64G and RV64GC ISAs when running the SPEC CINT2006 benchmarks [3].

One metric to consider when choosing between compilers is the performance of the generated code. Another metric is extensibility: a researcher developing a custom extension would probably want to modify a compiler to emit the new instructions. Code size can be important and the compiler license can be important too. Our study focuses on performance, which has the advantage of being somewhat easier to quantify than the other metrics.

Michael M. Pozulp is with the Applied Science & Technology graduate program at the University of California, Berkeley, Berkeley, CA 94720 USA. (email: pozulp@berkeley.edu)

Yukio Miyasaka is with the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley, Berkeley, CA 94720 USA. (email: yukio_miyasaka@berkeley.edu)

II. Methodology

We compared the performance of gcc and clang for RISC-V targets by running the SPEC CINT2006 benchmarks [4]. We obtained runtimes, dynamic instruction counts, dynamic instruction bytes, and PC histograms. We targeted 4 ISAs with 2 compilers, 2 input sets, and 2 platforms, yielding $4 \times 2 \times 2 \times 2 = 32$ combinations, not all of which are possible. While RV64GC and RV64G have 8 valid combinations each, RV64GB and RV64GCB have only 2, which provides a total of $8 + 8 + 2 + 2 = 20$ valid combinations, of which we considered 11. The 11 cases are marked with ✓ in Table I. Section II-A Compilers, Section II-B Inputs, and Section II-C Platforms explain the eponymous table entries. Section II-D Targets describes how we chose the 4 ISAs to target.

TABLE I
SUMMARY OF THE DATA COLLECTED FOR THIS REPORT

	Compilers		Inputs		Platforms	
	gcc	clang	test	ref	spike	hifive
RV64GC	✓	✓	✓	✓	✓	✓
RV64G	✓		✓		✓	
RV64GB	✓	X	✓		✓	X
RV64GCB	✓	X	✓		✓	X

A. Compilers

For RV64GC and RV64G we compiled the benchmarks using gcc 10.2.0 and clang 11.0.1 at `-O3` and statically-linked them with a gcc-compiled glibc using the GNU Binutils 2.35 BFD linker.

For RV64GB and RV64GCB we compiled using unreleased gcc and binutils which were modified to support the RISC-V Bitmanip Extension (B-extension). Because the base versions of the modified gcc and binutils are older than the versions we used for the other targets, when we compared RV64GB and RV64GCB against the other targets we recompiled everything with the older versions of gcc and binutils.

B. Inputs

SPEC CINT2006 is composed of 12 benchmarks and 35 workloads which process integer data for some typical desktop or server applications such as artificial intelligence, compilation, compression, search, and simulation. The benchmarks are CPU-intensive and use less than 2 GB of memory. We ran the `reference` and `test` inputs for the RV64GC target. To save time we ran only the `test` inputs for the other 3 targets since the `test` inputs run about 1 order of magnitude faster than the `reference` inputs. We ran the inputs to completion for all 4 targets.

C. Platforms

We collected runtimes on the SiFive HiFive Unleashed (HFU) which has 4 RV64GC "application cores" and 1 RV64IMAC "monitor core" on a SoC with 8 GB of DDR4 RAM. Each core has a single-issue in-order execution pipeline with a peak sustainable execution rate of 1 IPC. We ran the benchmarks on the 4 RV64GC cores, each of which has a 32 KiB 8-way instruction cache and a 32 KiB 8-way data cache. Both L1 caches hold 64-byte cache lines. All 5 cores share a 2 MiB 16-way coherent L2 cache.

We used gcc and Buildroot to compile and run a Linux 4.15 kernel as an initramfs on the HFU. We ran up to 4 benchmarks simultaneously and timed them using the `time` command. We ran the benchmarks in lexicographic order, so `400.perlbench` ran first and `483.xalancbmk` ran last. We waited until all the gcc-compiled benchmarks had finished running before running the clang-compiled benchmarks.

In addition to running the benchmarks on the HFU, we also ran them on the `spike` ISA simulator which allowed us to collect the following 3 metrics: dynamic instruction counts, dynamic instruction bytes, and PC histograms. We ran `pk`, the RISC-V Proxy Kernel, a user-level VM which handles system calls by intercepting and proxying them to the host machine. We modified `pk` to exclude instructions in system calls from our 3 metrics and we disabled on-demand program paging. Thus, our invocations looked like `spike -g pk -p -s benchmark input`.

D. Targets

We targeted 4 64-bit RISC-V ISAs: GC, G, GB, and GCB. We targeted GC because it is popular. GC is the ISA that SiFive chose for the HFU and GC is also the hardware baseline for general-purpose binary Linux distributions that was agreed upon by representatives from Debian, Fedora, and the RISC-V Foundation [1]. We targeted G to assess the effectiveness of instruction compression by looking at the dynamic instruction bytes fetched for G and GC. Targeting G also provides a sanity check because the dynamic instruction count should be the same for G and GC.

We targeted GB and GCB to assess the effect of the B-extension instructions on the dynamic instruction count and dynamic instruction bytes. The version of B-extension currently supported by the compiler is between 0.92 and 0.93, where the support for shadd instructions was added on top of version 0.92. The B-extension implementation in the compiler is partial in the sense that not all of the B-extension instructions are being used by the compiler.

The B-extension is a proposal for a standard extension to the Unprivileged ISA which provides bit manipulation instructions. Examples include Count Leading Zeros (`clz`), which counts the number of 0 bits at the MSB end of the argument, and Count Bits Set (`cpop`), which counts the number of 1 bits in a register. The B-extension design criteria require each new instruction to add enough value to cover the marginal cost of adding a new instruction to the ISA. For example, a new instruction which replaces at least three instructions would probably cover its cost. This is similar to *macro-op fusion*, where several ISA instructions are fused in the decode stage of the pipeline and handled as one internal operation. Some researchers prefer macro-op fusion to adding new instructions [3].

III. Results

This section is separated into two parts. The first part, Section III-A Comparing GCC and Clang, compares the performance of gcc and clang on the SPEC CINT2006 benchmarks. The second part, Section III-B Comparing B-extension to Macro-op Fusion, uses gcc to compare the performance of the B-extension to macro-op fusion using SPEC CINT2006. Our results presentation style is to give values for every benchmark in a plot and state the extrema and mean in prose. Eq. (1) is our formula for the mean, denoted $\bar{x}$.

$$\bar{x} = \left(\prod_{j=1}^{12} \left(\prod_{i=1}^{n_j} x_{ji} \right)^{\frac{1}{n_j}} \right)^{\frac{1}{12}}, \sum_{j=1}^{12} n_j^{\text{ref}} = 35, \sum_{j=1}^{12} n_j^{\text{test}} = 25 \tag{1}$$

The interior product calculates the geometric mean of the n_j workloads for benchmark j, where x_{ji} is the value for workload i of benchmark j. The exterior product calculates the geometric mean of the 12 benchmarks. There are 35 `reference` workloads and 25 `test` workloads.

A. Comparing GCC and Clang

Individual benchmarks ran up to 18% faster on the HFU when we compiled with gcc for the RV64GC target than when we compiled with clang (see Fig. 1). The mean runtime speedup of all the benchmarks was 5.7%. Compiling with gcc reduced dynamic instruction counts by up to 31% compared with clang (see Fig. 2). The mean dynamic instruction count reduction of all the benchmark was 10.8%.

Given two programs that do the same thing, we assume that the one with fewer instructions finishes first. We tested our assumption by comparing the data in Fig. 1 and Fig. 2 (see Fig. 3). The points outside the shaded areas violate our assumption. For example, `429.mcf` executed 13% fewer instructions with gcc, but the runtimes were about the same with gcc and clang. This indicates that `429.mcf` could be limited by the memory system. Another example is `456.hmmer`, which executed 3% more instructions with gcc, but the gcc runtime was 3% faster. We do not know why.

B. Comparing B-extension to Macro-op Fusion

Compiling with the B-extension reduced dynamic instruction counts by up to 17% (see Fig. 4). The mean dynamic instruction count reduction of all the benchmarks was 5.7%.

Macro-op fusion is a hardware optimization where several ISA instructions are fused in the decode stage of the pipeline and handled as one internal operation. The researchers in [3] counted the internal operations by parsing PC histograms for fusion pairs. The Load Effective Address (LEA) and Clear

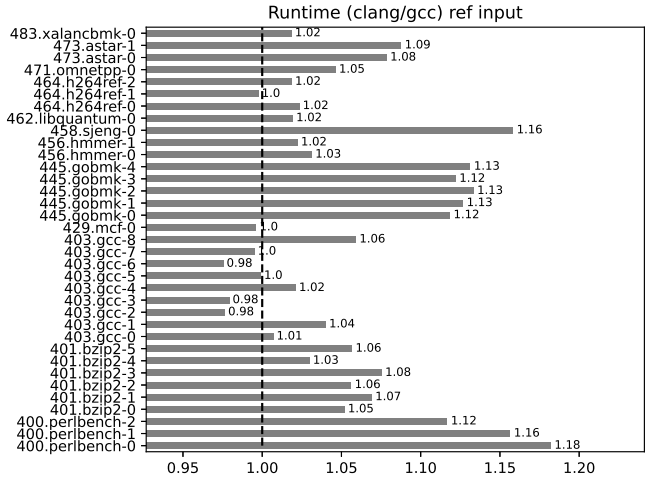

Fig. 1. Ratio of **runtimes** on the HFU for each workload in the `reference` input for each benchmark compiled with clang and gcc for the RV64GC target.

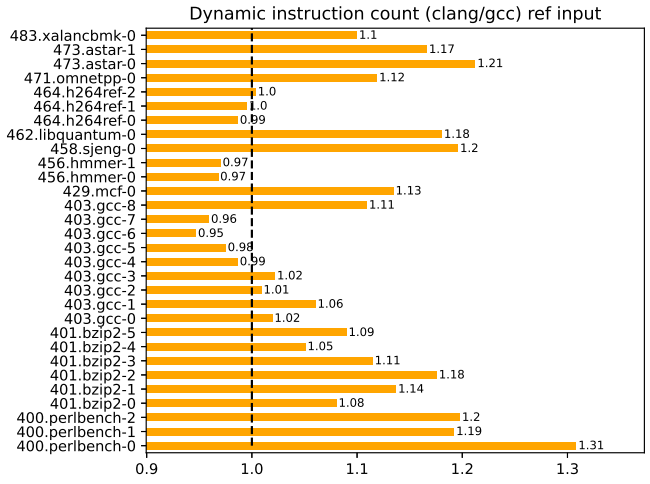

Fig. 2. Ratio of **dynamic instruction counts** on `spike` for each workload in the `reference` input for each benchmark compiled with clang and gcc for the RV64GC target.

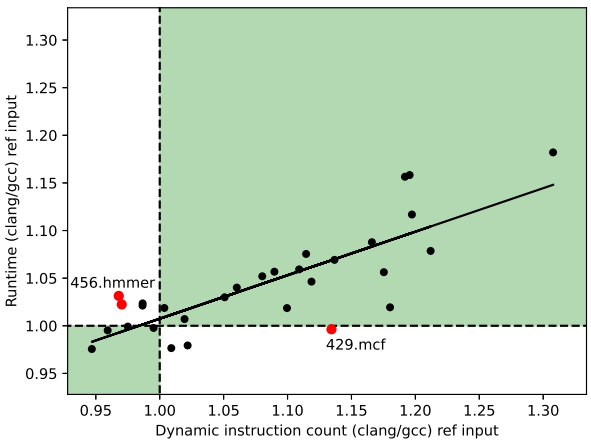

Fig. 3. The ratio of **dynamic instruction counts** on `spike` predicts the ratio of **runtimes** on the HFU. The Pearson correlation coefficient is 0.86. This figure combines the data from Fig. 1 and Fig. 2.

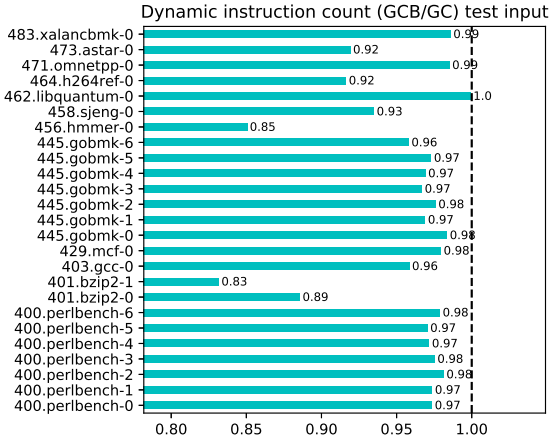

Fig. 4. Ratio of **dynamic instruction counts** on `spike` for each workload in the `test` input for each benchmark compiled with gcc for the RV64GC and RV64GCB targets.

Upper Word (CUW) fusions that they found somewhat correspond to the B-extension `sh{1,2,3}add[.uw]` (shadd) and `{add,sub,addi}wu` (wu) instructions, which allows us to compare our B-extension results to their macro-op fusion results.

Columns 3 and 4 of Table II show the percentage reduction of dynamic instruction count by B-extension (shadd and wu) and macro-op fusion, respectively. Column 1 shows the percent reduction in the dynamic instruction count, which we computed by taking the data from Fig. 4 and applying the arithmetic mean in order to produce one value per workload and then multiplying by 100 to get a percentage. Column 2 shows the percent of the reduction attributable to the shadd and wu instructions, which we computed by taking the sum of the shadd and wu instruction counts and dividing by the total count of B-extension instructions, then multiplying by 100 to get a percentage. Column 3 is the product of Columns 1 and 2 divided by 100, which approximately isolates the effect of the shadd and wu instructions from the effect of the other B-extension instructions.

Columns 3 and 4 indicate that the performance of the B-extension was better than macro-op fusion even though this is not a fair comparison for the following reasons: our isolation of the effect of the shadd and wu instructions is approximate, the shadd and wu instructions are not doing exactly the same replacement as the LEA and CUW fusions, the inputs are different, and the compiler versions are different.

Note that there is no instruction in the B-extension which corresponds to Indexed Load, another effective macro-op fusion proposed in [3]. Designers of high-end processors should consider combining such a fusion with B-extension in their designs.

TABLE II
COMPARISON BETWEEN B-EXTENSION AND MANUAL MACRO-OP FUSION

Benchmark	Percentage of dynamic instruction count			
	Reduction by B-ext	#shadd +#wu over #B-ext	Reduction by shadd +wu	Reduction by fusing LEA +CUW [3]
400.perlbench	2.51	74.03	1.86	1.71
401.bzip2	14.13	92.14	13.02	10.25
403.gcc	4.16	58.17	2.42	0.80
429.mcf	2.07	77.37	1.60	0.31
445.gobmk	2.93	98.12	2.88	2.75
456.hmmer	14.94	4.49	0.67	0.03
458.sjeng	6.51	95.62	6.23	4.89
462.libquantum	0.09	89.20	0.08	0.01
464.h264ref	8.37	96.97	8.11	5.72
471.omnetpp	1.50	81.57	1.22	0.63
473.astar	8.08	95.12	7.69	6.05
483.xalancbmk	1.42	86.28	1.23	0.11
Arithmetic mean	5.56	79.34	3.92	2.77

IV. FUTURE WORK

It would be interesting to try to modify clang to emit code for the RV64GC target which runs as fast as gcc-generated code on the HFU, and then submit the patches for inclusion in upstream clang. Repeating this study for SPEC CFP2006, which requires a Fortran compiler, could reveal even bigger opportunities for performance improvement. Repeating this study for 2017 SPEC benchmarks would provide useful data as well. Also, we would like to understand why gcc-compiled `456.hmmer` executed more instructions but ran *faster* than clang-compiled `456.hmmer`.

The comparable effects of the B-extension and macro-op fusion on the dynamic instruction count could be examined more closely in a future study which considers runtimes. It would be interesting to augment the Rocket core [2] with the B-extension and macro-op fusion, then synthesize the 2 new cores on FireSim [5] and run the benchmarks to collect runtimes.

Recompiling and recollecting dynamic instruction counts with the RISC-V Vector Extension (V-extension) would be interesting. We are not aware of any auto-vectorizers for RISC-V targets, but manually vectorizing important glibc functions like memcpy could provide most of the anticipated benefits without the expense required to get the auto-vectorizer in gcc or clang to emit V-extension instructions.

Another interesting study would assess the argument that RV64G, not RV64GC, should be the hardware baseline for general-purpose binary Linux distributions. The C-extension reduces binary size. It also reduces the number of dynamic instruction bytes fetched, which reduces instruction cache misses, which could reduce runtimes if the benefit exceeds the cost of increasing the complexity of instruction decoding. It would be worthwhile to test the assumption that RV64GC runs faster than RV64G by recompiling Linux, glibc, and the benchmarks with RV64G and running them on the HFU. The test is not completely fair because the HFU was designed for RV64GC so we expect RV64G binaries to run slower, but it would be very interesting if they did not.

V. CONCLUSION

Our study of the SPEC CINT2006 benchmarks indicates that gcc generates slightly faster code than clang for the popular RV64GC target. The biggest runtime speedup was 18% and the mean was 5.7%, as measured on the HFU. The biggest dynamic instruction count reduction was 31% and the mean was 10.8%, as measured on `spike`. We found that fewer dynamic instructions does not always mean faster runtimes. We also found that the B-extension is about as good as macro-op fusion at reducing dynamic instruction counts.

ACKNOWLEDGMENTS

The authors thank Professor Krste Asanović, Albert Ou, and Jerry Zhou for mentoring this project and providing valuable feedback.

REFERENCES

[1] Debian risc-v hardware baseline and abi choice. https://wiki.debian.org/RISC-V#Hardware_baseline_and_ABI_choice.

[2] Krste Asanović et al. The rocket chip generator. Technical report, University of California at Berkeley, 2016. http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[3] Christopher Celio et al. The renewed case for the reduced instruction set computer: Avoiding isa bloat with macro-op fusion for risc-v. *Preprint*, 2016. https://arxiv.org/pdf/1607.02318.

[4] John L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[5] Sagar Karandikar et al. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, 2018.

APPENDIX A
DATA COLLECTED FOR THIS REPORT

The figures and tables in this report present ratios and percentages derived from the actual data that we collected. In this appendix we present the actual data.

Table III and Table IV show the data that we collected for the RV64GC target using the `reference` inputs to SPEC CINT2006 compiled with gcc and clang, respectively. There are some missing values denoted by "N/A". The values are for `445.gobmk` which output messages like `cannot open or parse games/...` when we ran on `spike`, but not when we ran on the HFU. We're still investigating the issue.

Table V and Table VI show data for the RV64GC and RV64GCB targets using `test` inputs to SPEC CINT2006 compiled with the old version of gcc. Workload 7 for `400.perlbench` will not run because `pk` does not have a fork implementation.

TABLE III
REFERENCE INPUT DATA FOR RV64GC USING GCC

Benchmark-workload	Runtime (s)	Dynamic instruction count (billions)	Dynamic instruction bytes (billions)
483.xalancbmk-0	3265	879	2581
473.astar-1	2036	715	2232
473.astar-0	1464	337	1035
471.omnetpp-0	3352	570	1653
464.h264ref-2	6428	4442	13659
464.h264ref-1	697	487	1497
464.h264ref-0	798	580	1766
462.libquantum-0	9348	1239	3750
458.sjeng-0	4156	2691	8069
456.hmmer-1	4150	2681	8667
456.hmmer-0	2030	1279	4136
445.gobmk-4	631	N/A	N/A
445.gobmk-3	464	N/A	N/A
445.gobmk-2	652	N/A	N/A
445.gobmk-1	1199	N/A	N/A
445.gobmk-0	488	N/A	N/A
429.mcf-0	5327	283	866
403.gcc-8	157	63	182
403.gcc-7	1033	177	498
403.gcc-6	1039	199	558
403.gcc-5	801	156	435
403.gcc-4	517	115	321
403.gcc-3	345	103	290
403.gcc-2	568	141	391
403.gcc-1	407	165	473
403.gcc-0	271	82	234
401.bzip2-5	927	417	1261
401.bzip2-4	1807	911	2908
401.bzip2-3	1136	636	1940
401.bzip2-2	1037	354	1092
401.bzip2-1	464	200	620
401.bzip2-0	1025	530	1595
400.perlbench-2	1074	712	2236
400.perlbench-1	712	429	1212
400.perlbench-0	1899	1161	3501

TABLE IV
REFERENCE INPUT DATA FOR RV64GC USING CLANG

Benchmark-workload	Runtime (s)	Dynamic instruction count (billions)	Dynamic instruction bytes (billions)
483.xalancbmk-0	3325	966	2931
473.astar-1	2215	833	2545
473.astar-0	1579	408	1220
471.omnetpp-0	3508	637	1793
464.h264ref-2	6548	4458	12982
464.h264ref-1	696	484	1420
464.h264ref-0	817	573	1602
462.libquantum-0	9529	1463	3634
458.sjeng-0	4814	3217	9696
456.hmmer-1	4243	2601	8023
456.hmmer-0	2094	1238	3812
445.gobmk-4	714	N/A	N/A
445.gobmk-3	521	N/A	N/A
445.gobmk-2	739	N/A	N/A
445.gobmk-1	1352	N/A	N/A
445.gobmk-0	546	N/A	N/A
429.mcf-0	5307	321	892
403.gcc-8	167	70	203
403.gcc-7	1028	170	476
403.gcc-6	1014	189	537
403.gcc-5	800	152	426
403.gcc-4	528	113	317
403.gcc-3	338	105	296
403.gcc-2	554	142	394
403.gcc-1	423	175	503
403.gcc-0	272	84	238
401.bzip2-5	979	455	1377
401.bzip2-4	1861	957	3067
401.bzip2-3	1222	709	2150
401.bzip2-2	1096	416	1276
401.bzip2-1	496	228	702
401.bzip2-0	1078	572	1729
400.perlbench-2	1200	852	2695
400.perlbench-1	823	512	1495
400.perlbench-0	2244	1519	4725

TABLE V
TEST INPUT DATA FOR RV64GC USING OLD GCC

Benchmark-workload	Dynamic instruction count (thousands)	Dynamic instruction bytes (thousands)
483.xalancbmk-0	305584	892051
473.astar-0	21137308	68043620
471.omnetpp-0	1659430	4477637
464.h264ref-0	101485835	307916461
462.libquantum-0	176172	534558
458.sjeng-0	18143629	54845637
456.hmmer-0	19442807	62154964
445.gobmk-6	43563415	131299669
445.gobmk-5	717726	2158878
445.gobmk-4	136952	413110
445.gobmk-3	15987902	47938089
445.gobmk-2	102128	306234
445.gobmk-1	3374635	10030219
445.gobmk-0	271071	796967
429.mcf-0	3211501	9667915
403.gcc-0	5040322	14410669
401.bzip2-1	22847381	70495326
401.bzip2-0	11981295	36649293
400.perlbench-7	N/A	N/A
400.perlbench-6	7912	23272
400.perlbench-5	4213	12385
400.perlbench-4	2302	6725
400.perlbench-3	258793	746868
400.perlbench-2	304181	864706
400.perlbench-1	8716	25767
400.perlbench-0	13313	39238

TABLE VI
TEST INPUT DATA FOR RV64GCB USING OLD GCC

Benchmark-workload	Dynamic instruction count (thousands)	Dynamic instruction bytes (thousands)
483.xalancbmk-0	301242	886130
473.astar-0	19429112	64512790
471.omnetpp-0	1634576	4455704
464.h264ref-0	92995415	305215074
462.libquantum-0	176012	529947
458.sjeng-0	16961923	53577069
456.hmmer-0	16537867	56691429
445.gobmk-6	41731815	129583143
445.gobmk-5	698275	2136683
445.gobmk-4	132715	408238
445.gobmk-3	15456762	47372977
445.gobmk-2	99659	303302
445.gobmk-1	3269659	9926505
445.gobmk-0	266578	791270
429.mcf-0	3145143	9520569
403.gcc-0	4830425	14127327
401.bzip2-1	19003462	63510790
401.bzip2-0	10611134	34011521
400.perlbench-7	N/A	N/A
400.perlbench-6	7744	23063
400.perlbench-5	4089	12223
400.perlbench-4	2237	6640
400.perlbench-3	252374	739588
400.perlbench-2	298462	860401
400.perlbench-1	8485	25481
400.perlbench-0	12960	38805

APPENDIX B
BUILD COMMANDS

In this appendix we include the commands required to build the toolchains and the tools that we used to collect our data. We include Listing 1 Commands for Building GCC and Clang, Listing 2 Commands for Building Spike, Listing 3 Commands for Building the Proxy Kernel, Listing 4 Commands for Building and Running SPEC CINT2006 with Speckle, Listing 5 Commands for Building B-extension toolchain.

We modified Speckle to add options for the input type and the config file. We also modified the run command for `483.xalancbmk` to give the real paths of the input files because we had an issue using `realpath` with `pk`.

Listing 1. Commands for Building GCC and Clang

```
git clone https://github.com/riscv/riscv-gnu-toolchain.git
    ↪ #3db4fdb
cd riscv-gnu-toolchain
mkdir build
cd build
../configure --prefix=/path/to/install
make -j linux

git clone https://github.com/llvm/llvm-project.git llvm
cd llvm
git checkout llvmorg-11.1.0 #1fdec59bffc1
cmake \
  -S llvm \
  -B build \
  -G Ninja \
  -DCMAKE_BUILD_TYPE="Release" \
  -DBUILD_SHARED_LIBS=True \
  -DLLVM_USE_SPLIT_DWARF=True \
  -DLLVM_OPTIMIZED_TABLEGEN=True \
  -DLLVM_BUILD_TESTS=False \
  -DDEFAULT_SYSROOT="/path/to/install/sysroot" \
  -DLLVM_DEFAULT_TARGET_TRIPLE="riscv64-unknown-linux-gnu"
      ↪ \
  -DLLVM_ENABLE_PROJECTS="clang" \
  -DLLVM_TARGETS_TO_BUILD="RISCV" \
  -DCMAKE_INSTALL_PREFIX:PATH=/path/to/install
cmake --build build --target install
```

Listing 2. Commands for Building Spike

```
wget https://git.kernel.org/pub/scm/utils/dtc/dtc.git/
    ↪ snapshot/dtc-1.6.0.tar.gz
tar xf dtc-1.6.0.tar.gz
cd dtc-1.6.0
make -j
cp dtc /path/to/install/bin

git clone git@github.com:pozulp/riscv-isa-sim.git #
    ↪ modified spike based on 21684fd
cd riscv-isa-sim
mkdir build
cd build
PATH=$PATH:/path/to/install/bin ../configure --prefix=/path
    ↪ /to/install --enable-histogram
make -j
make install
```

Listing 3. Commands for Building the Proxy Kernel

```
git clone git@github.com:pozulp/riscv-pk.git # modified pk
    ↪ based on 75bbd1e
cd riscv-pk
mkdir build
cd build
PATH=$PATH:/path/to/install/bin ../configure --prefix=/path
    ↪ /to/install --host=riscv64-unknown-linux-gnu
PATH=$PATH:/path/to/install/bin make -j
make install
```

Listing 4. Commands for Building and Running SPEC CINT2006 with Speckle

```
git clone git@github.com:pozulp/Speckle.git # modified
    ↪ Speckle based on b650d4b
cd Speckle

export PATH=/path/to/install:$PATH
export PATH=/path/to/install/bin:$PATH
export PATH=/path/to/install/riscv64-unknown-linux-gnu/bin:
    ↪ $PATH
export SPEC_DIR=/path/to/spec2006install

./gen_binaries.sh --compile --input ref --config gnu
./gen_binaries.sh --compile --input ref --config llvm
./gen_binaries.sh --run --input ref --config gnu
./gen_binaries.sh --run --input ref --config llvm
#./gen_binaries.sh --compile --input test --config gnu
#./gen_binaries.sh --compile --input test --config llvm
#./gen_binaries.sh --run --input test --config gnu
#./gen_binaries.sh --run --input test --config llvm
```

Listing 5. Commands for Building B-extension toolchain

```
git clone https://github.com/riscv/riscv-gnu-toolchain.git
    ↪ #d45cfc6
cd riscv-gnu-toolchain
git submodule update -i riscv-gcc riscv-binutils
cd riscv-gcc
git checkout riscv-bitmanip #3a004f3 based on 49f75e0
cd ..
cd riscv-binutils
git checkout riscv-bitmanip #c870418 based on 612aac6
cd ..
mkdir build
cd build
../configure --prefix=/path/to/install
make -j linux

git clone -b cs252a_cgc_project_b git@github.com:pozulp/
    ↪ riscv-isa-sim.git # modified spike based on cab796f
cd riscv-isa-sim
mkdir build
cd build
PATH=$PATH:/path/to/install ../configure --prefix=/path/to/
    ↪ install --enable-histogram --with-isa=RV64IMAFDCB
make -j
make install
```

TABLE VII
GCC COMPILER FLAGS

all	`-O3 -static -DSPEC_CPU_LP64`
400.perlbench	`-DSPEC_CPU_LINUX_X64 -std=gnu89`
401.bzip2	∅
403.gcc	∅
429.mcf	∅
445.gobmk	∅
456.hmmer	∅
458.sjeng	∅
462.libquantum	`-DSPEC_CPU_LINUX`
464.h264ref	`-fsigned-char`
471.omnetpp	∅
473.astar	∅
483.xalancbmk	`-DSPEC_CPU_LINUX`

TABLE VIII
CLANG COMPILER FLAGS

all	`-O3 -static -DSPEC_CPU_LP64`
400.perlbench	`-DSPEC_CPU_LINUX_X64 -std=gnu89`
401.bzip2	∅
403.gcc	∅
429.mcf	∅
445.gobmk	∅
456.hmmer	∅
458.sjeng	∅
462.libquantum	`-DSPEC_CPU_LINUX`
464.h264ref	`-fsigned-char`
471.omnetpp	∅
473.astar	`-std=c++98`
483.xalancbmk	`-DSPEC_CPU_LINUX -std=c++98`