Less is More: Exploiting the Standard Compiler Optimization Levels for Better Performance and Energy Consumption

Kyriakos Georgiou, Zbigniew Chamski, Kerstin Eder

University of Bristol

Imperial College London (UK), March 26 2019

TEAMPLAY

Time, Energy and security Analysis for Multi/Many-core heterogeneous PLAtforms

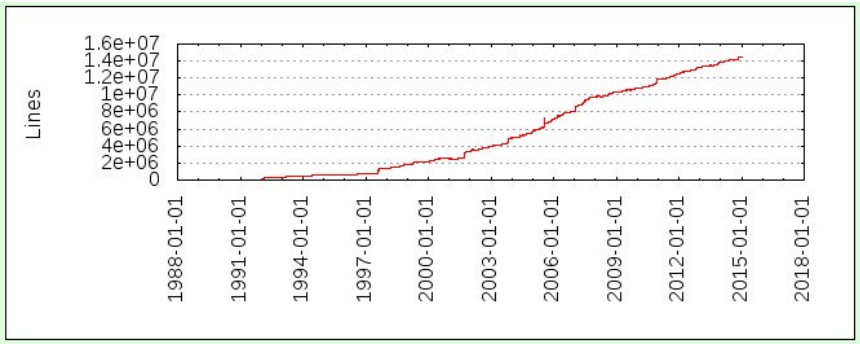
University of BRISTOL

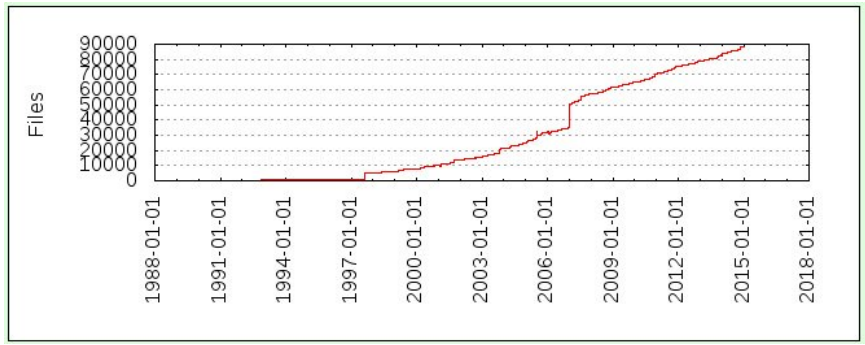
Compilers – The Good and the bad side

- They are in the heart of software development.
- Abstract away hardware complexity and increase productivity.
- But, probably, some of the most complex software.
- Need to support a variety of programming languages and architectures.
- Have to keep up with with hardware and languages advancements.

Compiler Engineer is a tough job!

GCC size in numbers (2015)[1].





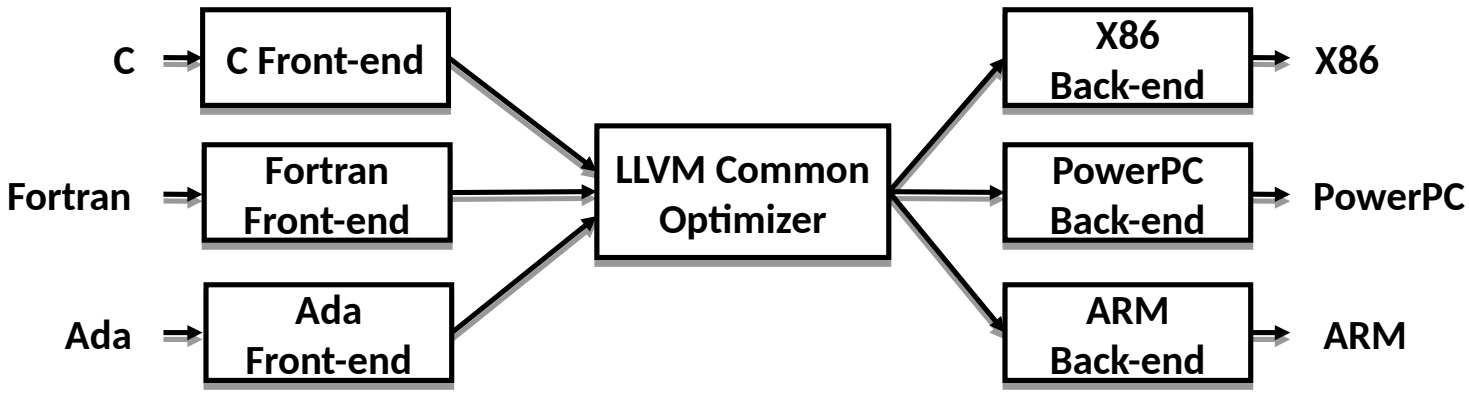
- GCC is spread across 88.5k files with its 14.5 million lines.
- LLVM/Clang with just under four million lines.
- The entire Linux kernel was over 19 million lines.

Tuning the compiler optimizations to perform well across all possible applications is impractical.

[1] "GCC Soars Past 14.5 Million Lines Of Code & I'm Real Excited For GCC 5" at: https://www.phoronix.com/scan.php?page=news_item&pxM̄Tg3OTQ

Mitigation strategies (1/2)

- Modern compilers introduced three-phase design (LLVM).



- Introduced a series of transformation and analysis passes available via flags.
- The challenges:
  1. selection problem
  2. phase-ordering problem

Mitigation strategies (2/2)

- GCC v4.7 has 2^{82} possible optimization configurations[2].
- It is impractical to explore the whole optimization-configuration space to find optimal configurations.
- Standard optimization levels: -O1, -O2, -03, -Os, -Oz

Each standard optimization level offers a predefined sequence of optimizations, which are proven to perform well based on a number of micro-benchmarks and a range of mainstream architectures.

[2] James Pallister, Simon J. Hollis, and Jeremy Bennett. 2015. Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms. Comput. J.58, 1 (2015), 95–109. https://doi.org/10.1093/comjnl/bxt129

Can we do better?

Two main approaches are proven that can perform better:

- Random-based iterative compilation.
  - Expensive to run.
  - In many cases produces malfunctioning executables.

- Machine-learning-based compilation.
  - Expensive training phase.
  - Typically account only for one resource.
  - Hardly portable to new architectures or different compiler versions (or even a patch).
  - In many cases produces malfunctioning executables.

Motivation

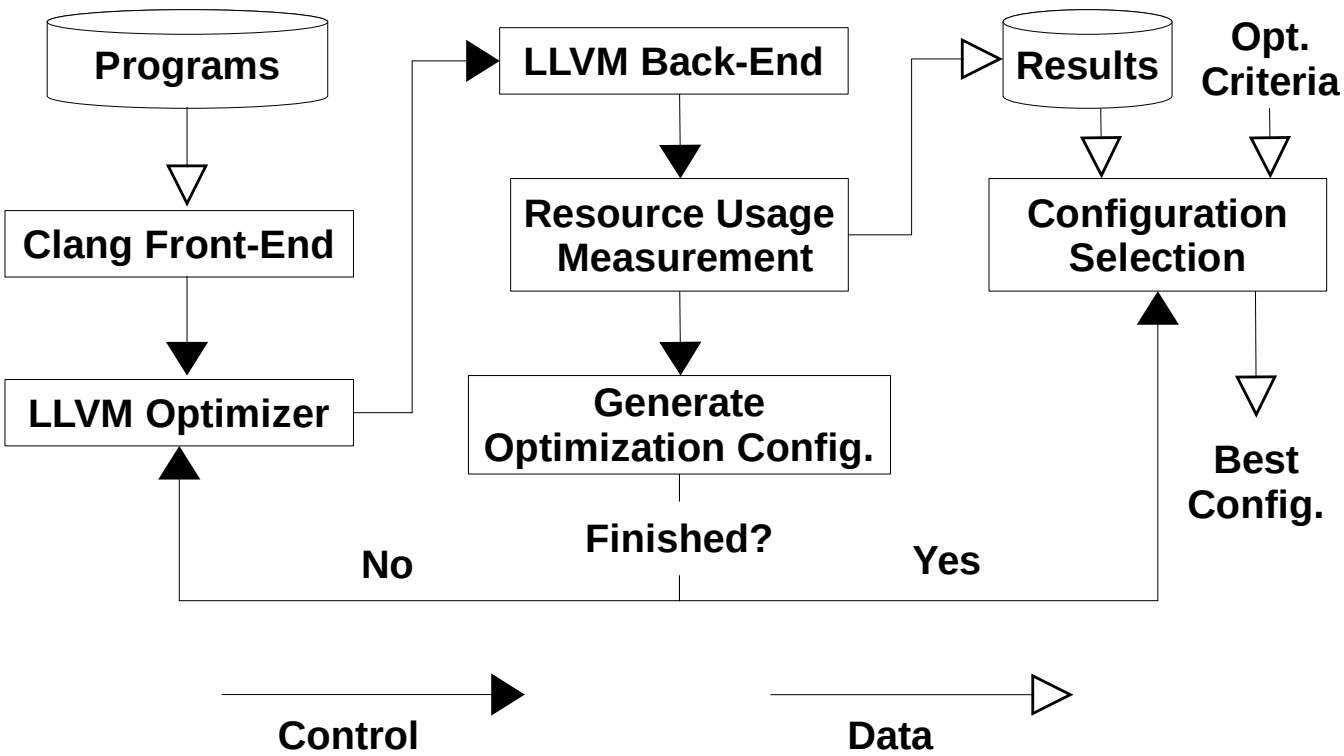
Many of the "good" produced optimizations are subsequences of the predefined code optimization levels.

- In a recent work based on iterative compilation and machine learning using LLVM 3.8[3]:
  - 79% of the -O3 optimization flags were part of a single subgroup with a fixed ordering that is similar to that used in the -O3 configuration.
- Maybe after all the standard optimization levels can be a good starting point!

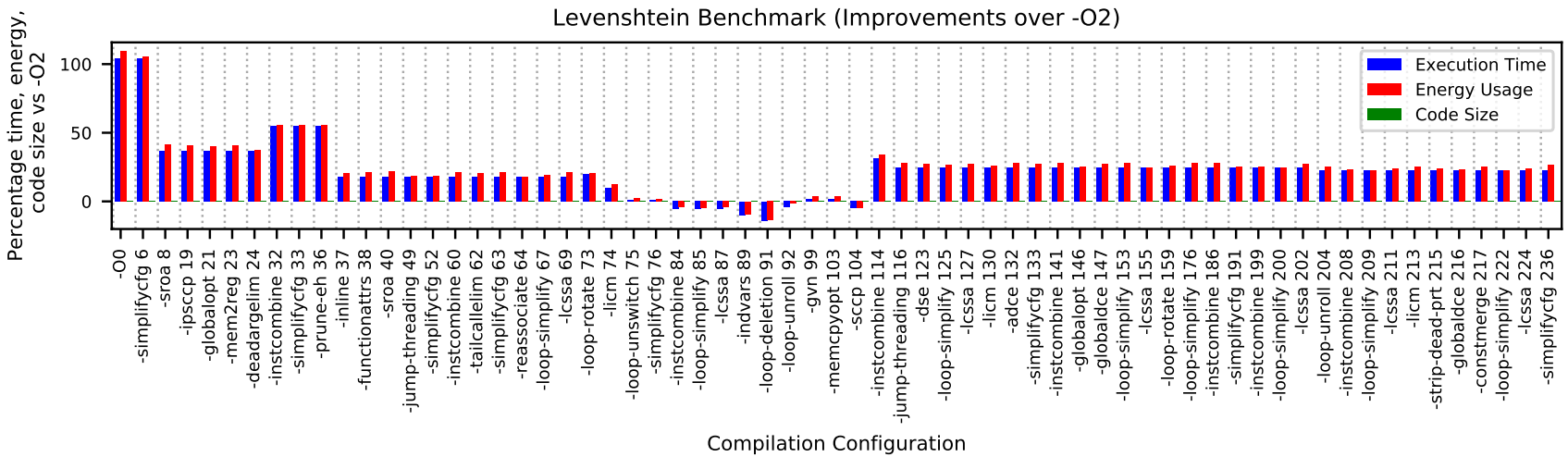
[3] Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. 2017. MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning. ACM Trans. Archit. Code Optim. 14, 3, Article 29 (Sept. 2017), 28 pages. https://doi.org/10.1145/3124452

Our approach (1/2)

Aims to preserve the empirical knowledge built into the ordering of flags for the standard optimization levels.



Our approach (2/2)



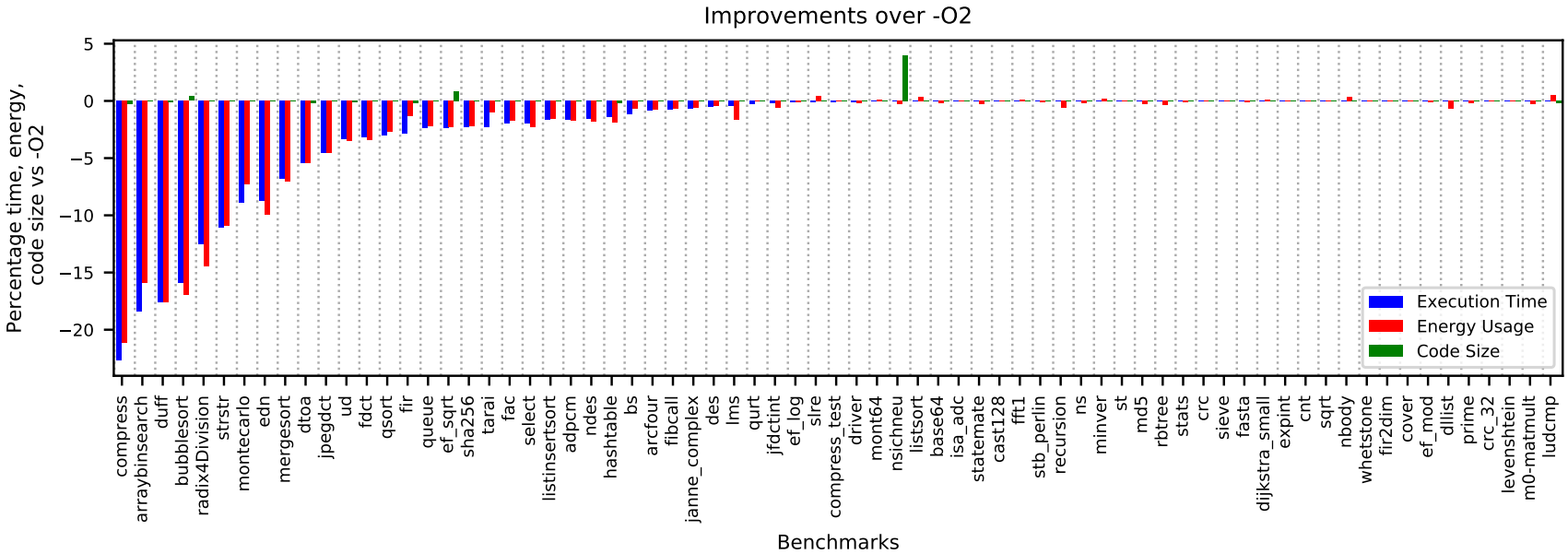
50 configurations exercised for the Cortex M0 and 64 for the Cortex M3.

Benefits

- Portability: The architecture and the compiler are treated as a black box.
- No expensive training phase similar to the ones needed by the machine learning approaches is required.
- Provides valuable insights to the software engineer on how each optimization flag affects the resource of interest.
- Avoids reordering that typically breaks the compilation or create a malfunctioning executable;
- Multi-criteria optimizations are possible without needing to train a new model for each resource.

Experimental evaluation (1/2)

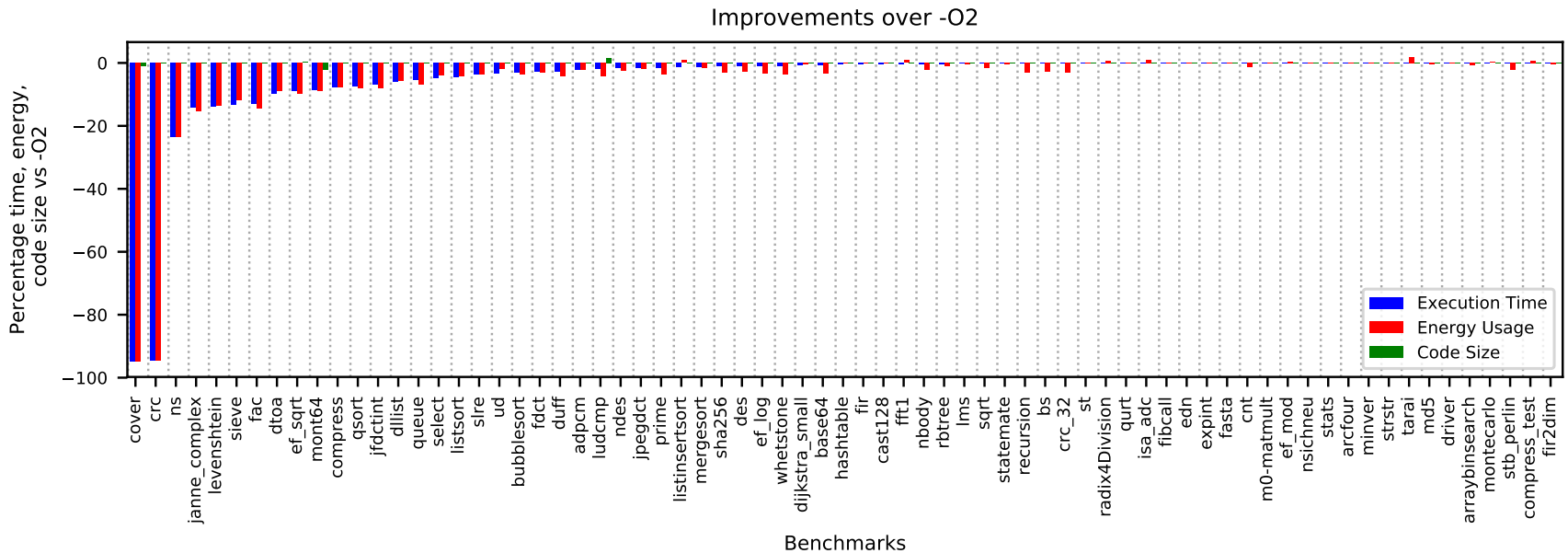
Results for the Cortex-M0 processor and the LLVM v3.8 compilation framework.



- average reduction in execution time of 2.5%.
- 29/71 benchmarks seeing execution time improvements over -O2 ranging from 1% to 23%.
- similar savings were observed for energy.

Experimental evaluation (2/2)

Results for the Cortex-M3 processor and the LLVM v5.0 compilation framework.



- average reduction in execution time of 5.3%.
- 31/71 benchmarks seeing execution time improvements over -O2 ranging from 1% to 90%.
- similar savings were observed for energy.

Comparing to existing works

Using Inductive Logic Programming to find good configurations[4].

- using the same benchmark suite, same board (Cortex-M3) and same measurement setup, but GCC.
- average saving in execution time 8% over -O3.
- 1000 random configurations tested for each benchmark = one week training time.
- hardly portable.
- accounts only for one resource.

[4] Craig Blackmore, Oliver Ray, and Kerstin Eder. 2015. A logic programming approach to predict effective compiler settings for embedded software. Theory and Practice of Logic Programming 15, 4-5 (2015), 481–494. https://doi.org/10.1017/S1471068415000174

Comparing to existing works

Used Fractional Factorial Design to explore the large optimization space (2^{82}combinations for the GCC version used)[5]

- using the same benchmark suite, same boards and same measurement setup, but GCC.
- 2048 tested configurations using each benchmark.
- demonstrated savings only on a couple of benchmarks over -O3.
- hardly portable.

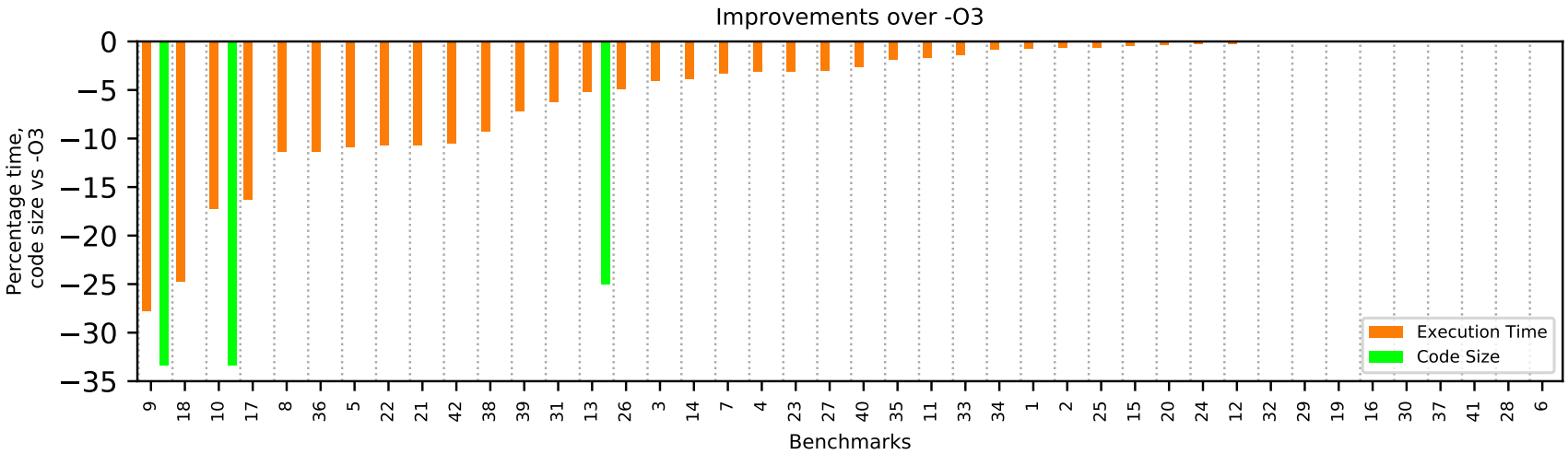
[5] James Pallister, Simon J. Hollis, and Jeremy Bennett. 2015. Identifying Compiler Options to Minimize Energy Consumption for Embedded Platforms. Comput. J.58, 1 (2015), 95–109. https://doi.org/10.1093/comjnl/bxt129

Latest developments

Motivation: Existing auto-tuning techniques are only suitable for tuning compilers settings, not for routine testing and compiler improvement.

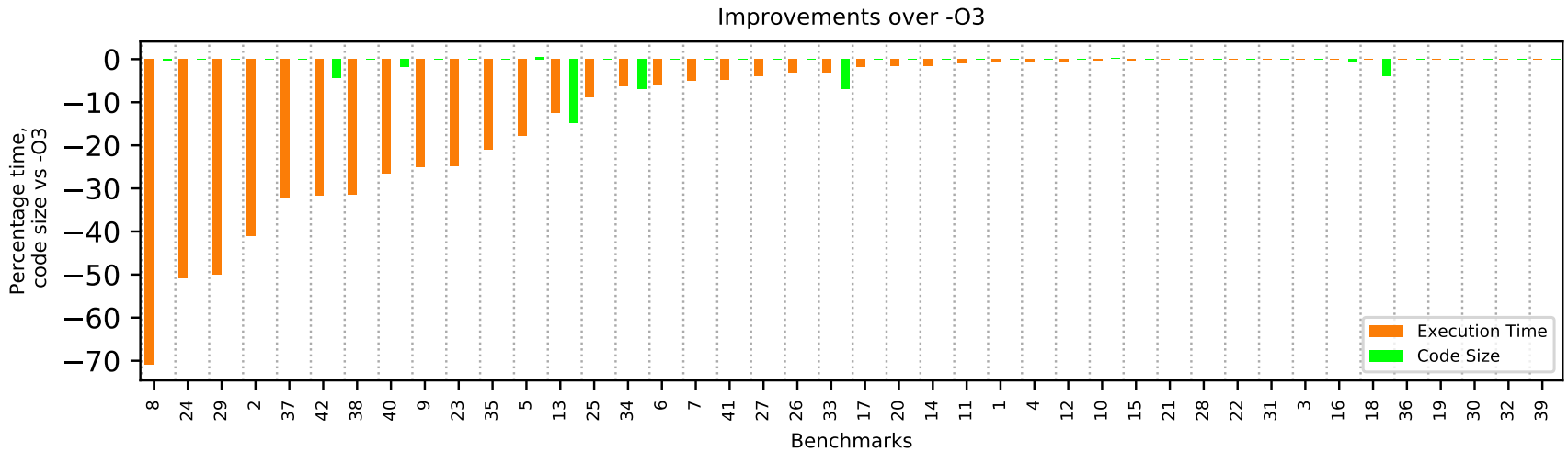
- Ported to new architectures and compiler version.
  - The Intel i5-6300U, widely used in portable PCs.
  - The ARM Cortex-A53-based Broadcom BCM2837 used in the Raspberry Pi 3B+.
  - LLVM v6.0.
- Introduced an enhanced nightly regression system that allows for systematic compiler improvement and tuning.
- Benchmarks used are part of the LLVM test-suite.

Results for the Cortex-A53 processor



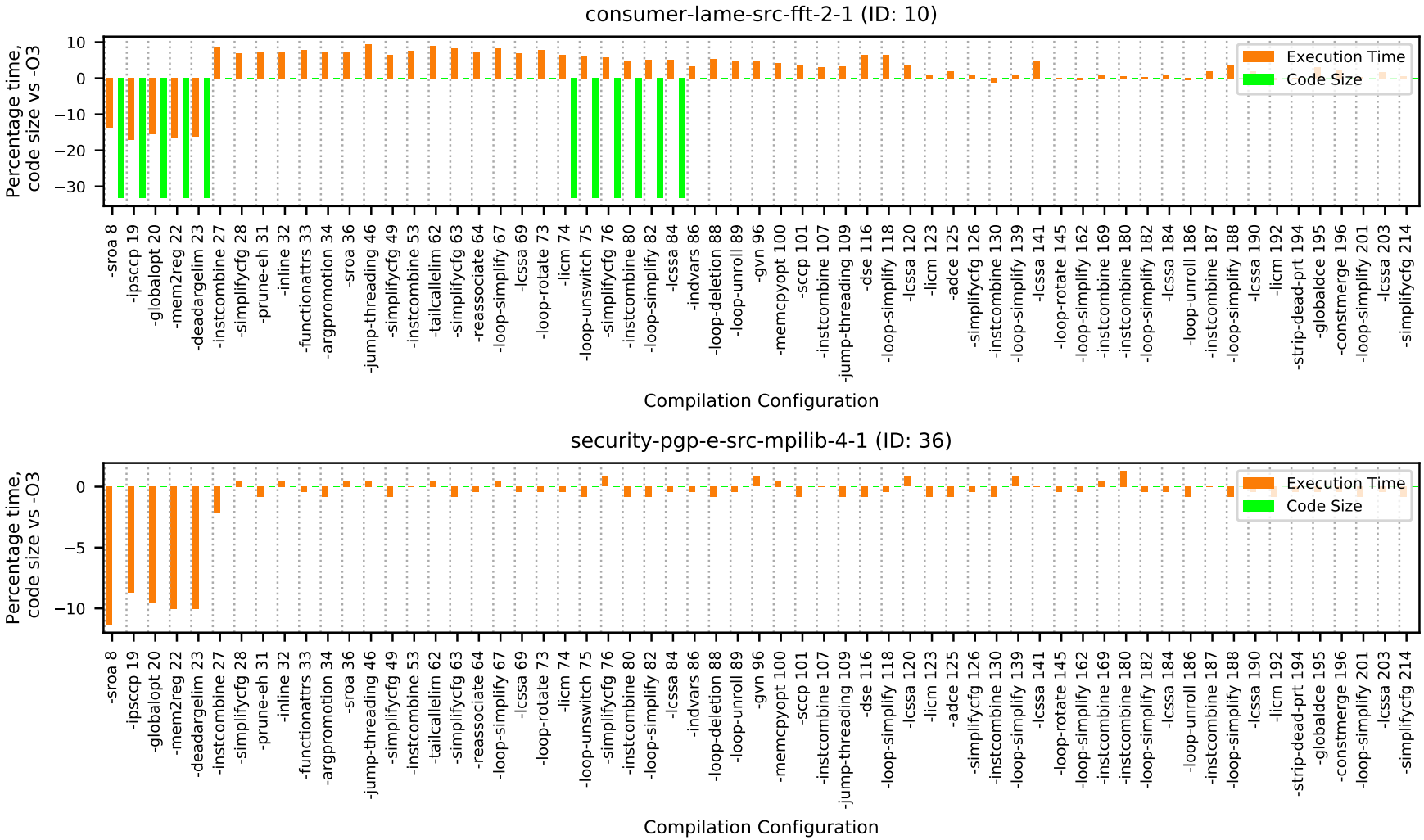
- average reduction in execution time of 5.1%.
- 26/42 benchmarks seeing execution time improvements over -O3 ranging from around 1% to 28%.

Results for the Intel i5-6300U processor

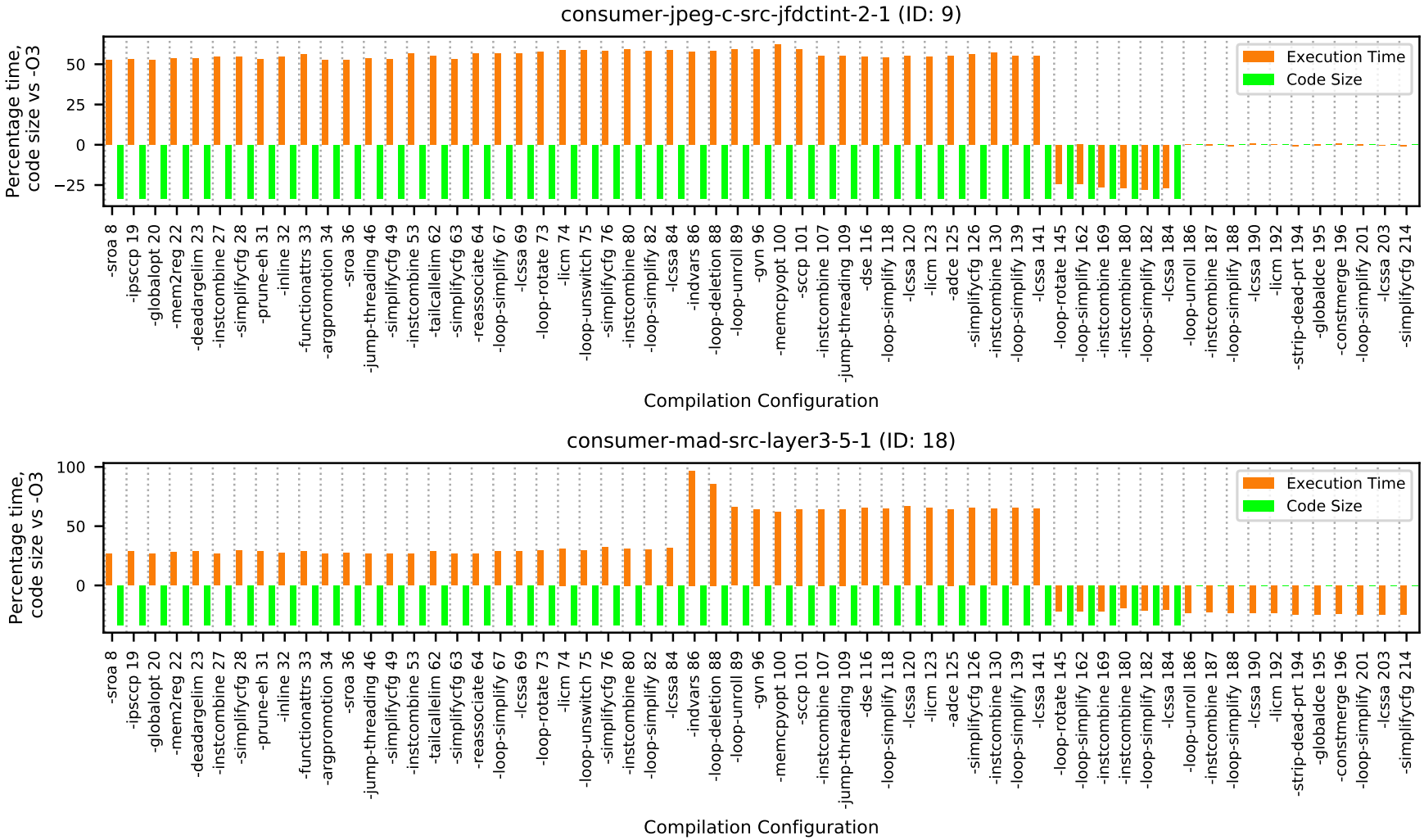


- average reduction in execution time of 11.5%.
- 26/42 benchmarks seeing execution time improvements over -O3 ranging from around 1% to 71%.

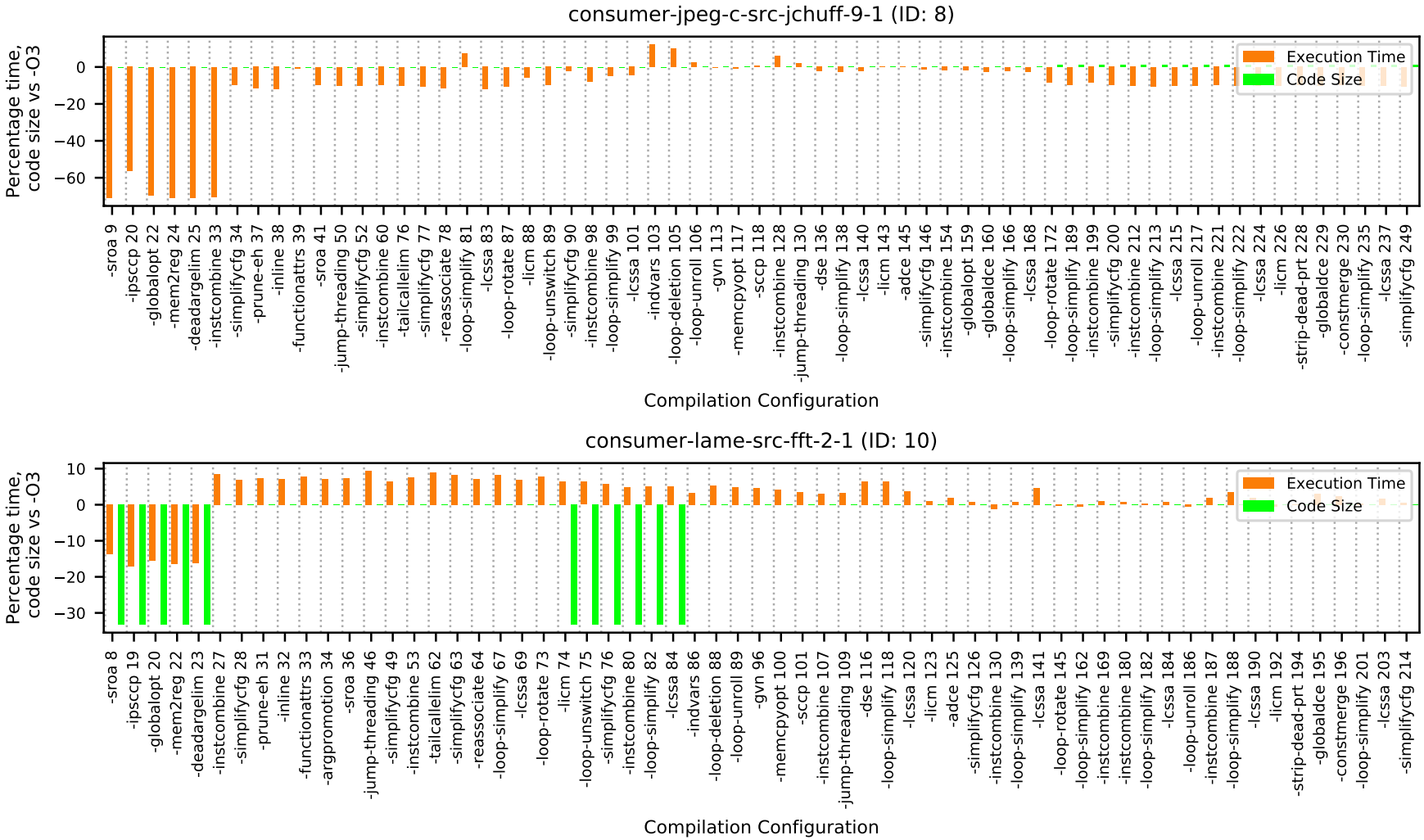
Potential Optimization Opportunities for Cortex-A53



Potential Optimization Opportunities for Cortex-A53



Cross-Architecture Potential Optimizations



Classic Nightly Regression Testing

- Standard approach to evaluate compilers during development.
- Compares test results to a previous regression run that performed the best so far.
- Traditionally tracks the validation of test results and their overall resource usage.
- Does not indicate the source of a performance regression.
- Does not indicate optimization opportunities.
- Performance regression investigation is a time consuming task.

i5-6300U Nightly Regression Report

Benchmark ID	First Config. Better than -O3	Config. Removing Gains	Best Overall Config.	Execution Time Reduction %
8	sroa - 9	simplifycfg - 34	instcombine - 33	-70.98
2	sroa - 9	simplifycfg - 34	instcombine - 33	-40.98
37	sroa - 9	simplifycfg - 34	instcombine - 33	-32.34
23	sroa - 9	simplifycfg - 34	instcombine - 33	-24.76
13	sroa - 9	simplifycfg - 34	sroa - 9	-12.53
25	sroa - 9	simplifycfg - 34	sroa - 9	-8.82
7	sroa - 9	simplifycfg - 34	sroa - 9	-5.11
42	sroa - 9	simplifycfg - 34	ipsccp -20	-31.61
35	sroa - 9	simplifycfg - 34	instcombine - 221	-21.05
24	sroa - 9	simplifycfg - 90	functionattrs - 39	-50.79
34	instcombine - 33	lcssa - 83	instcombine - 33	-6.25
33	instcombine - 33	lcssa - 83	instcombine - 33	-3.13
29	no pattern	no pattern	jump-threading - 130	-50.00
38	sroa - 9	instcombine - 60	instcombine - 33	-31.53
40	sroa - 9	loop-rotate - 87	ipsccp -20	-26.51
9	loop-unroll - 217	after simplifycfg -249	mem2reg - 24	-25.00
5	no pattern	no pattern	loop-simplify 138	-17.82
6	sroa - 9	globaldce - 229	loop-rotate - 87	-6.00
41	reassiciate - 78	indvars - 103	loop-rotate - 87	-4.76
27	sroa - 9	lcssa - 101	ipsccp -20	-3.92
26	loop-rotate - 87	instcombine - 98	loop-rotate - 87	-3.17

Cortex-A53 Nightly Regression Report

Benchmark ID	First Config. Better than -O3	Config. Removing Gains	Best Overall Config.	Execution Time Reduction %
10	sroa - 8	instcombine - 27	sroa - 8	-17.18
36	sroa - 8	instcombine - 27	sroa - 8	-11.35
42	sroa - 8	instcombine - 27	sroa - 8	-10.48
31	sroa - 8	instcombine - 27	sroa - 8	-6.25
7	sroa - 8	instcombine - 27	sroa - 8	-3.23
5	loop-rotate - 73	jump-threading - 109	instcombine - 80	-10.82
22	loop-rotate - 73	jump-threading - 109	instcombine - 80	-10.71
21	loop-rotate - 73	jump-threading - 109	memcopyopt - 100	-10.71
39	loop-rotate - 73	instcombine - 80	loop-rotate - 73	-7.14
26	loop-rotate - 73	instcombine - 80	simplifycfg - 76	-4.92
13	loop-unswitch - 75	instcombine - 80	loop-unswitch - 75	-5.16
23	loop-unswitch - 75	instcombine - 80	simplifycfg - 76	-3.07
9	loop-rotate - 145	loop-unroll - 186	loop-simplify - 182	-27.72
18	loop-rotate - 145	no pattern	strip-dead-prot - 194	-24.68
17	sroa - 8	loop-rotate - 73	ipsccp - 19	-16.23
8	sroa - 8	instcombine - 80	globalopt - 20	-11.38
38	sroa - 8	instcombine - 53	sroa - 8	-9.22
3	no pattern	no pattern	licm - 192	-4.00
14	sroa - 8	indvars - 86	functionattrs - 33	-3.85
4	no pattern	no pattern	strip-dead-prot - 194	-3.07

Added Value

- Exposes optimization opportunities that standard nightly regression can not capture.
- Can demonstrate recurring patterns instead of isolated performance regressions.
- Pinpoints to the flags (compiler code) responsible for the performance degradations or gains.
- Keeps all the consecutive IR and Object files.
- Validates test results for sub-sequences of the standard opt. levels.

Some Exposed Potential Optimizations

- If-conversion heuristics (branches vs predications) – GI.
- Dead code in unrolling (catch-up loops left behind after fully unrolling) – GI.
- Not making use of multiply-accumulate instruction – TS.
- Tuning of unrolling parameters – TA.

Categories: `GI: General Improvement, TA: Target-Aware heuristic tuning, TS: Target-Specific optimization refinement`

Future work

- Currently testing the approach on HPC using OpenMP.
- Combine the approach with performance counters reports.
- Apply machine learning to tune the compiler settings for a specific architecture.
- Building the ultimate compiler debugger.

Related publications

- Kyriakos Georgiou, Craig Blackmore, Samuel Xavier-de Souza, and Kerstin Eder. Less is more: Exploiting the standard compiler optimization levels for better performance and energy consumption. In Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems, SCOPES '18, pages 35–42, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3207719.3207727, doi:10. 1145/3207719.3207727 and arxiv version URL: https://arxiv.org/abs/1802.09845

- Kyriakos Georgiou, Zbigniew Chamski, Andres Amaya Garcia, David May, Kerstin Eder. Lost in translation: Exposing hidden compiler optimization opportunities. Submitted to The Computer Journal and currently under review. arxiv version URL: https://arxiv.org/abs/1903.11397

Thank you!

Questions?

Kyriakos.Georgiou@bristol.ac.uk

ID	Benchmark Name	ID	Benchmark Name
1	automotive-basicmath-cubic-3-1	2	automotive-basicmath-isqrt-1-1
3	automotive-qsort1-src-qsort-1-1	4	automotive-susan-e-src-susan-10-1
5	automotive-susan-e-src-susan-2-1	6	automotive-susan-s-src-susan-1-1
7	consumer-jpeg-c-src-jcdctmgr-13-1	8	consumer-jpeg-c-src-jchuff-9-1
9	consumer-jpeg-c-src-jfdctint-2-1	10	consumer-lame-src-fft-2-1
11	consumer-lame-src-newmdct-10-1	12	consumer-lame-src-newmdct-3-1
13	consumer-lame-src-psymodel-17-1	14	consumer-lame-src-quantize-7-1
15	consumer-lame-src-quantize-pvt-6-1	16	consumer-lame-src-takehiro-16-1
17	consumer-lame-src-takehiro-5-1	18	consumer-mad-src-layer3-5-1
19	consumer-mad-src-layer3-6-1	20	consumer-tiff2rgba-src-tif-predict-4-1
21	consumer-tiffdither-src-tif-fax3-8-1	22	consumer-tiffdither-src-tif-fax3-9-1
23	consumer-tiffdither-src-tiffdither-1-1	24	consumer-tiffmedian-src-tiffmedian-1-1
25	consumer-tiffmedian-src-tiffmedian-3-1	26	consumer-tiffmedian-src-tiffmedian-4-1
27	consumer-tiffmedian-src-tiffmedian-5-1	28	consumer-tiffmedian-src-tiffmedian-6-1
29	network-dijkstra-src-dijkstra-large-5-1	30	office-ghostscript-src-gdevpbm-1-1
31	office-rsynth-src-nsynth-5-1	32	office-rsynth-src-nsynth-9-1
33	security-pgp-d-src-mpilib-1-1	34	security-pgp-e-src-mpilib-1-1
35	security-pgp-e-src-mpilib-3-1	36	security-pgp-e-src-mpilib-4-1
37	telecomm-adpcm-c-src-adpcm-1-1	38	telecomm-adpcm-d-src-adpcm-1-1
39	telecomm-fft-fftmisc-5-1	40	telecomm-fft-fourierf-3-1
41	telecomm-gsm-src-rpe-4-1	42	telecomm-gsm-src-short-term-2-1