Saint Petersburg State University

Software engineering

Vladimir Miloserdov

Flexible report generation system development for LLVM LNT

Graduation Thesis

Scientific supervisor:
Senior Lecturer Iakov Kirilenko

Reviewer:
Associate Professor Anton Korobeynikov

Saint Petersburg
2019

Contents

Introduction

Modern industry and science progression demand a constant growth of computational power [1]. Present-day applications utilize emerging technologies such as artificial intelligence systems, blockchain, virtual and augmented reality [2] and 5-th generation networks. Often they apply harsh limits to software and underlying hardware. Such limits include program execution time, power [3] and code size efficiency, device dimensions, portability, and reliability. To meet these expectations it is in turn required to constantly improve both processors and development tools – compilers, debuggers, simulators, etc [4].

Testing and performance analysis plays a major role in the tools development process. It is worth noting that software and hardware and tools themselves are being changed at the same time. Because of the great complexity of all components, huge test suites and many different benchmarks are involved across multiple teams. This generates a lot of effort spent on such casual tasks like running tests in trying to reproduce the desired result and localizing errors. Many of both performance and functional regressions are being detected too late in the process which bumps the difficulty of debugging and correcting mistakes. Moreover, in many cases, it is not trivial to understand which component – for instance, a test suite or hardware caused the trouble.

Under such conditions, a unified testing system capable of running various types of tests and benchmarks as well as providing convenient tools, such as producing developer reports to work with their results, helps to greatly facilitate the testing and development processes.

This work is mainly focused on improving one such system – LLVM LNT which is the part of LLVM compiler project. It is a tool that allows to conduct compiler testing and visualize its results. In particular, the goal of the work is to create a flexible, configurable subsystem for developer report generation which should become a part of both LNT and the testing automation flow.

Developer reporting is a useful feature that allows compiler developers not to spend too much time gathering testing information after each commit they make. It also prevents tehm, to some extend, from missing regressions after the change and, generally, facilitates error detection on earlier stages.

Acknowledgments

I would like to thank Iakov Kirilenko for overseeing my work and for helping me write this thesis.

I would also like to thank Sergey Yakushkin without whom this work would have been impossible for supervising the project, contributing ideas and providing valuable feedback throughout the process.

More thanks go to Dmitry Koznov for providing his feedback on writing.

Special thanks go to Anton Korobeynikov for reviewing this work.

Also, I would like to thank Stanislav Sartasov and Dmitry Luciv for the inspiration to study and advice.

Finally, I would like to thank my sister – Liubov Miloserdova, and friends – A. Polyakov, P. Shumilov, V. Shabanov, M. Kostycin, I. Tyulandin, G. Volkov, A. Minaev, K. Smirnov, and A. Chugaev for their feedback and incredible moral support.

1 Background

This chapter provides a reader with the necessary knowledge about compilers and their testing, LLVM, main concepts of LNT testing framework and developer reports.

1.1 Introduction to LLVM

The LLVM infrastructure – an umbrella of multiple projects used for building compilers. It includes Clang frontends, backends, linker, implementation of the C++ standard library and a JIT engine. [5]

An LLVM-based compiler – a compiler built partially or completely with the LLVM infrastructure.

For the purpose of this work, we will consider universal optimizing LLVM-based compilers built completely on top of the LLVM infrastructure. Typical structure of such compiler can be illustrated in Figure-1 below.

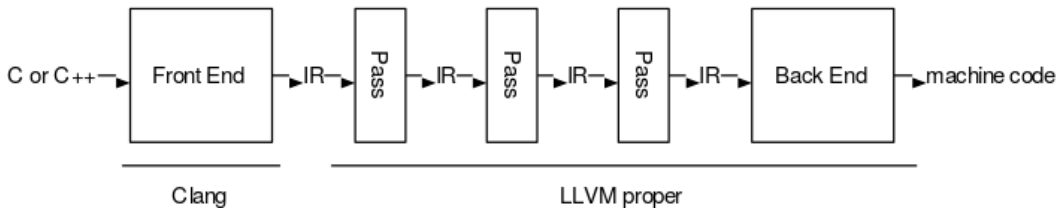


Figure 1: LLVM-based compiler structure

One of the key features of modern compilers, and in particular LLVM, is the modular architecture. A compiler is usually divided into frontend, middlend, and backend. Such a structure is shown in Figure-1.

Frontend translates source code from a high-level programming language into lower-level compiler's intermediate representation (IR). For instance, Clang frontend translates C or C++ code into LLVM IR.

Middlend is often understood as a system of various code optimizers working on the intermediate level. It is responsible for architecture-independent transformations of IR code. For example, dead code elimination pass is considered to be part of middlend.

Backend produces low-level code for targets, such as assembly or binary code from IR. It also includes some architecture-dependent transformations.

1.2 Testing

Software testing – the process of dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain. [6]

Throughout this work, we are interested in functional testing and performance testing.

Functional testing – software testing for the purpose of checking the correct implementation of functional requirements for this software.

Performance testing – software testing to verify its compliance with various performance requirements, as well as to evaluate performance indicators of this software (for example, the operating time consumed by the RAM).

1.2.1 Compiler testing

The compiler should at least produce correct code. In modern reality, the performance of the generated code and the compilation time are also of great importance.

A systematic approach is needed for testing and evaluation of the above. To verify the correctness, sets of functional tests are commonly used, which include various cases of programs according to the specification of the compiler. It is important to check as many of these cases as possible by providing the largest test coverage. [7]

For performance evaluation, special test programs that measure metrics (characteristics) in different execution scenarios – benchmarks. Metrics used in compiler development can include the number of processor cycles spent on benchmark execution, generated code size, as well as cache misses count, pipeline stalls and others.

1.3 Introduction to LLVM LNT

LNT[1] – a tool for performance and regression testing of compilers. Originally developed for testing LLVM-based compilers, it, however, can be used for testing any other compilers and applications.

The software is implemented in Python and has a Flask-based[2] web application for accessing testing data as well as a command-line user interface. The web application uses Jinja2[8] as template engine while the database layer uses SQLAlchemy[9] for its object-relational mapping (ORM)[3] and supports multiple backends such as SQLite and PostgreSQL. [10]

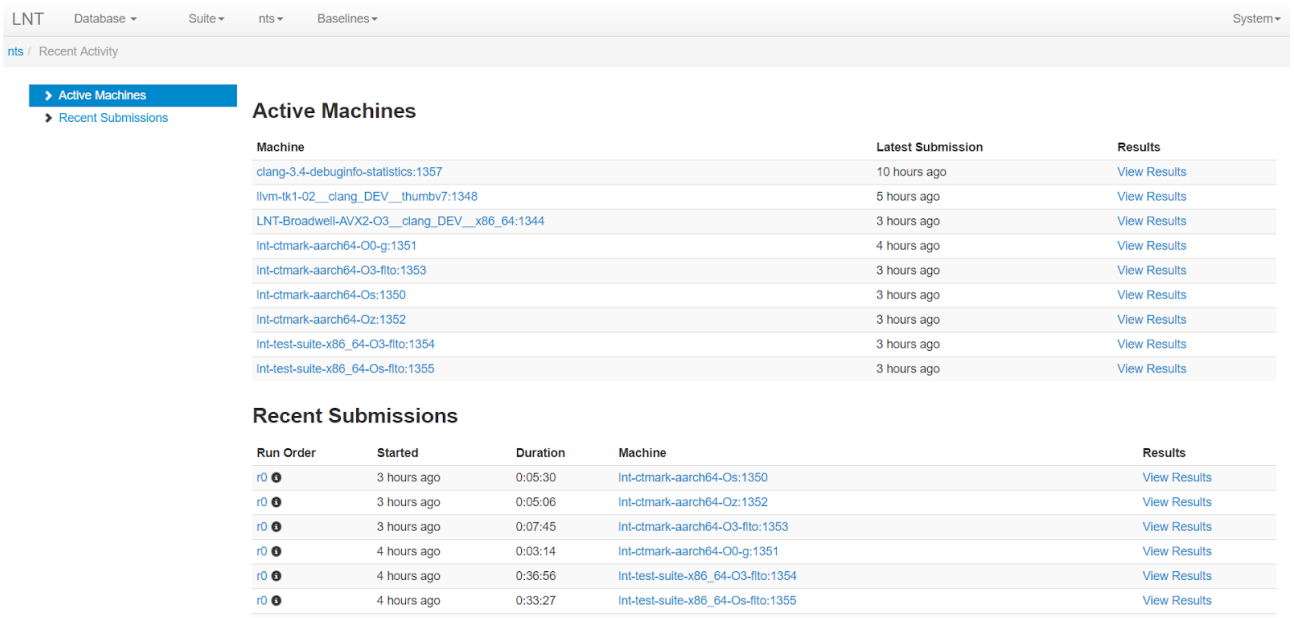


Figure 2: LNT web application screenshot

For the context of this work, we will need to be familiar with the following concepts.

Order represents the state of the software to be tested (i.e. compiler). For example, different versions or submissions can be used as Orders.

Machine represents the testing configuration used. For instance, it can be an actual hardware machine configuration, a set of compilation parameters or a combination of them.

[1]LNT stands for LLVM Nightly Testing

[2]Flask is a python web framework. Project homepage: `http://flask.pocoo.org`

[3]Object-relational mapping (ORM) is a technique in which data from object code is connected to a relational database.

Run represents a pass through some set of tests. It has an Order on which it was run, set of tests and a Machine – the testing configuration. As a result of Run, we get Samples.

Sample represents test result data point for a specific test and metric. For example, Sample `('core/benchmarks/zlib', 473811, 'cycle_count')` displays that on some test named `zlib` the resulting number of cycles was `473811`.

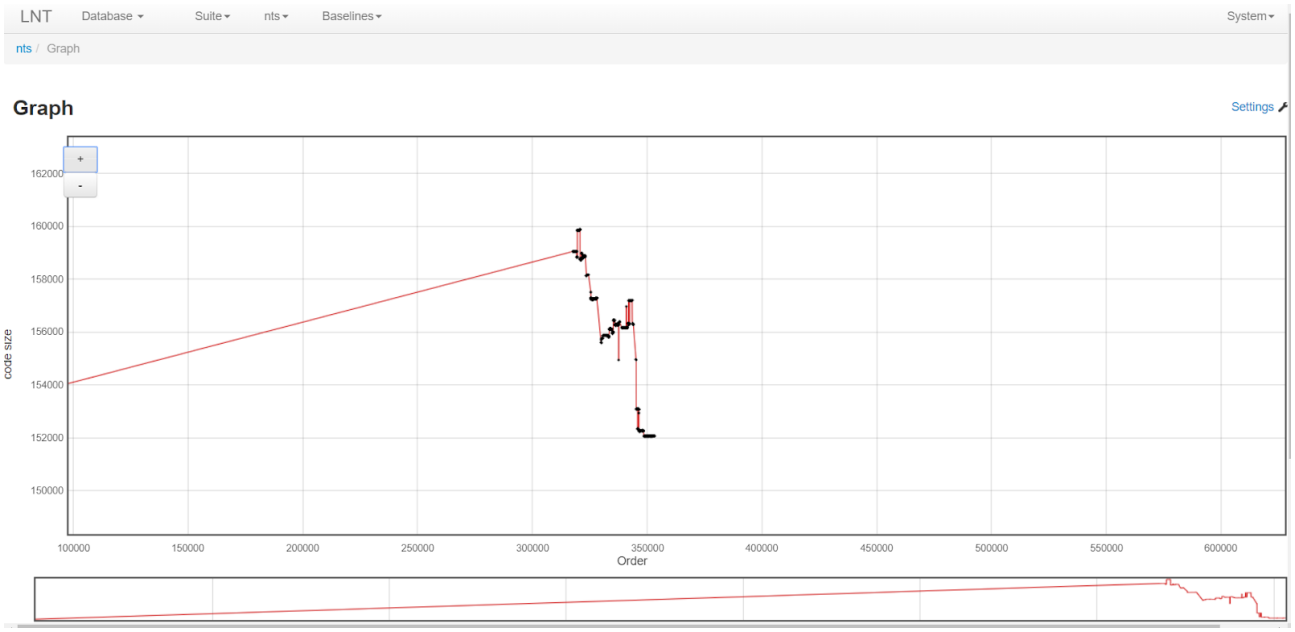


Figure 3: LNT web application screenshot – graph of code_size metric in the single test over different Orders

When considering two Run comparison on the single test and the specified metric, the following test statuses are possible:

Status name	Description
Failed	Test or benchmark failed, e.g. during the execution or compilation
Passed	Test passed
Regressed	Benchmark shows regression by metric value
Improved	Benchmark shows improvement by metric value

Table 1: Test comparison statuses

2 Statement of the problem

The immediate goal of this work was to produce the system capable of generating configurable compiler testing reports for developers.

The report generation system that resulted from this work is actually just one stage in a much larger testing automation system built upon Jenkins 2 for testing Synopsys MetaWare™ [11] compiler.

This work can be divided into the following tasks:

1. Producing system specification
2. Conducting an overview of existing approaches
3. Designing and implementing the solution
4. Integrating the system into the current testing pipeline
5. System testing and evaluation

3 System specification

As a part of this work, system specification was produced by analyzing feedback collected by surveying potential users of the system. In total, 7 compiler developers had responded and completed the survey.

The author came to the conclusion that the report generation system should be implemented as an extension of LNT and provide the following functionality:

- Ability to generate testing status reports for developers
- Ability to customize report sections
  - Customization of displayed test suites and benchmarks
  - Customization of displayed metrics
  - Customization of displayed buckets customization
  - Customization of display options
- Support of multiple hardware configurations and compiler option sets
- Support of configuring report generator by using configuration files
- Support of report sections with miscellaneous data (e.g. changelogs)
- Support of the command-line interface

3.1 Developer report specification

For the purpose of this work, **developer report** – customized summary of testing results at the specific state of the development process. Report should be divided into sections, each section displays a comparison summary of 2 LNT Runs on the specified test suite or benchmark on some LNT Machine.

Each section, as well as the report itself should have the valid status in according to the table below.

Status name	Priority	Description
Failed	1	At least 1 failed test
Fixed	2	At least 1 test no longer fails
Unstable	3	Combination of performance improvements and regressions
Regressed	4	At least 1 performance regression
Improved	5	At least 1 performance improvement
Stable	6	Default status

Table 2: Valid statuses

Developer report should contain header, status, summary and one or more section. It should follow the structure as described in Figure 4:

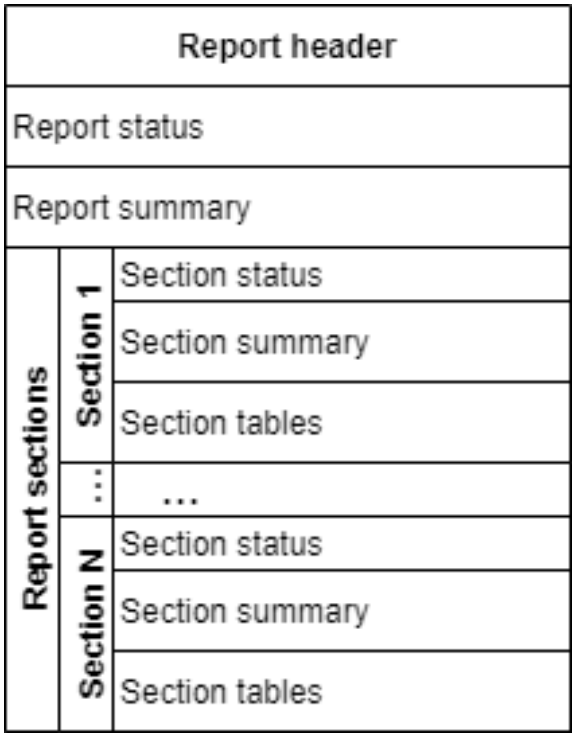


Figure 4: Developer report structure

4 System design

4.1 Justification of choosing LNT as a base system

LLVM LNT already provides a convenient way of handling the testing data and has re-usable built-in Jinja templates as well as some other useful functionality.

Therefore, the author considers the requirement to build the report generation system on top of the LNT framework well-founded and justified.

4.2 System architecture

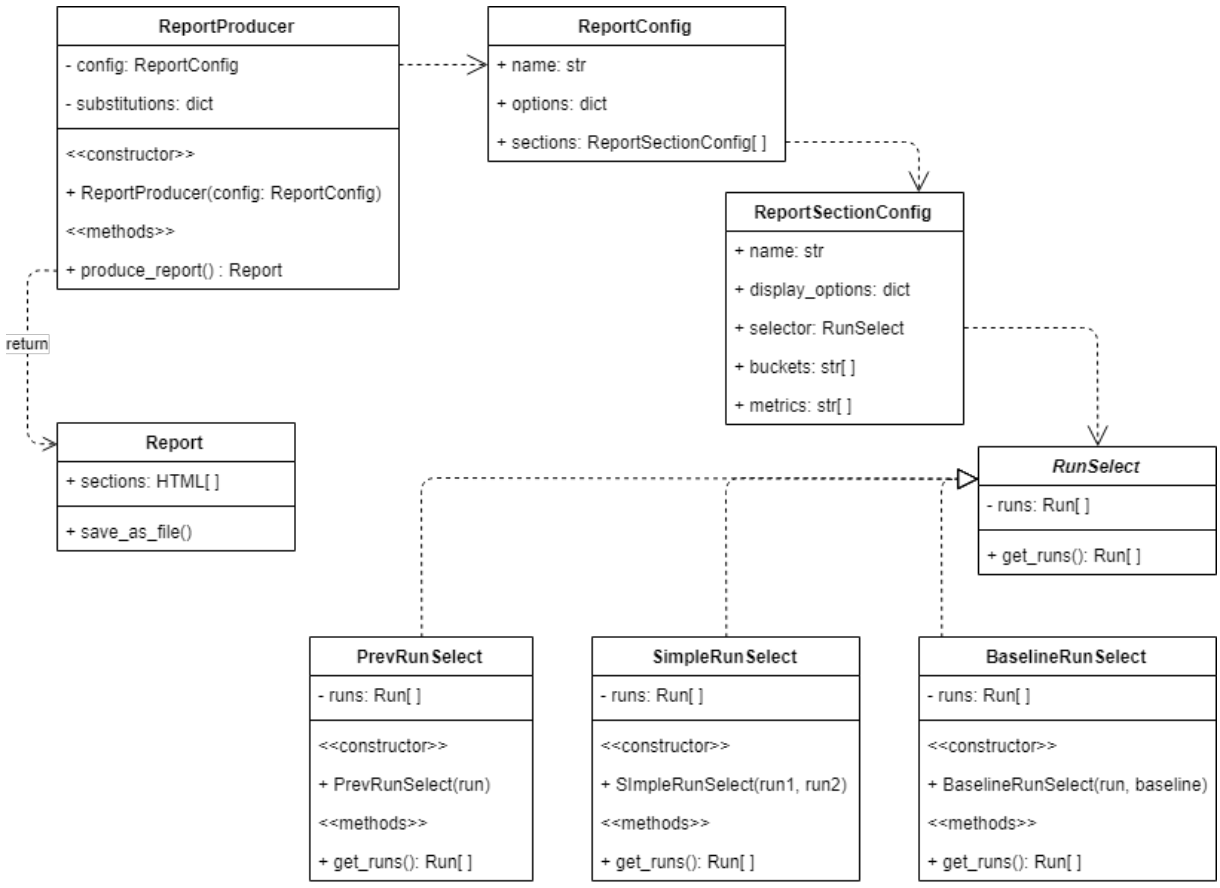


Figure 5: Architecture of the report generation system

In Fig. 2 above `Report` class represents the finished report which can be saved as an HTML file. Reports are produced by `produce_report` method of `ReportProducer` class which, in turn, is initialized with `ReportConfig`

instance – configuration of the report. In this class, report section configurations are stored as `ReportSectionConfig` instances.

One aspect worth spending attention is `RunSelect`. It is responsible to determine LNT runs for comparison in the section. It is an abstract class and has many implementations for different scenarios.

5 System integration

This chapter describes the production testing flow to which the author integrated his solution. It also covers changes of this flow which were required for successful integration.

5.1 Existing testing flow overview

As a part of this work, it was required to integrate reporting generation system into the current system testing flow.

For automation, Jenkins 2 framework is used. The flow structure is shown below.

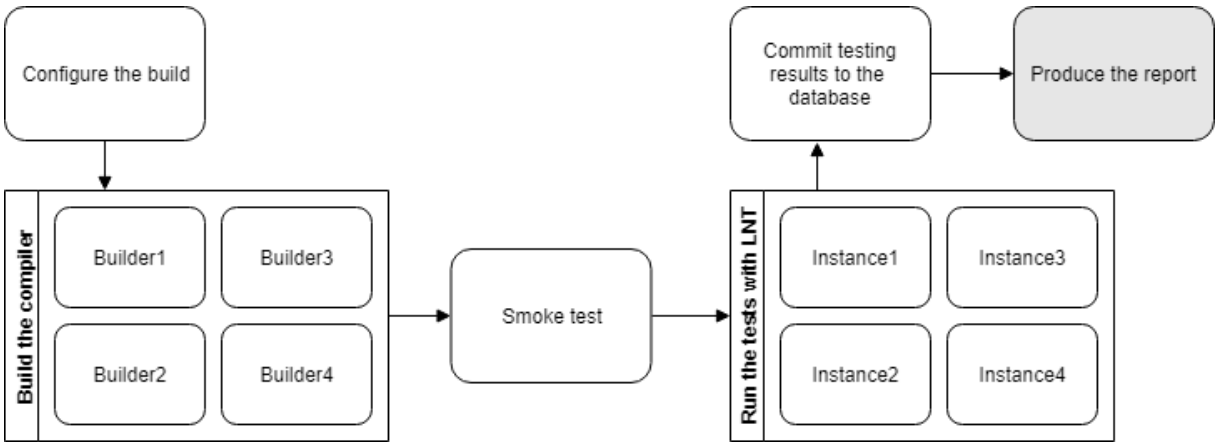


Figure 6: Main stages of testing automation flow

As one can notice, there are 2 parallel stages – `Building compiler` and `Running tests with LNT`. After each of them, a synchronization barrier is used before starting the next stage.

For the implementation of above there is a Groovy[1] library of shared functions and each stage has its own dedicated class. The component added during this work is colored in grey.

[1]Groovy – an optionally typed and dynamic Java-based programming language with static typing [12]

5.2 Jenkins pipeline modification

The above mentioned Jenkins 2 library was extended by developing several functions for handling reports. This handling can be decomposed as follows:

1. Check that the pipeline state is ready for producing reports (e.g. all actual testing is done)
2. Get the necessary LNT Run id-s to be included in the report
3. Prepare LNT parameters for invocation
4. Invoke LNT and wait for the subprocess to be completed
5. Process the result of invocation and handle errors if necessary
6. Deliver the report if the previous step succeeded. Otherwise, send an error report instead.

5.3 Report delivery

Generated reports are saved as HTML files to be processed by testing automation flow. Once the job is finished, it does several checks including that the report file is present and valid. Then reports are distributed via email in according to mailing list policies.

The function responsible for report delivery was developed as a part of this work in Groovy and integrated into Jenkins 2 pipeline.

6 Testing and evaluation

This chapter contains an overview of the system developed as a result of this work and its testing.

6.1 System evaluation

At earlier stages of system integration, corridor testing[1] and tune some aspects like changing configuration file format from JSON to Python to allow more powerful tweaking.

The system after completion and integration was evaluated by author and developer test group of 3 persons. Later, the system was launched a in testing environment and available for all compiler developers – reports were generated after each commit. At the moment of writing, the system is actually used in production. As one can see on the figure above, the produced

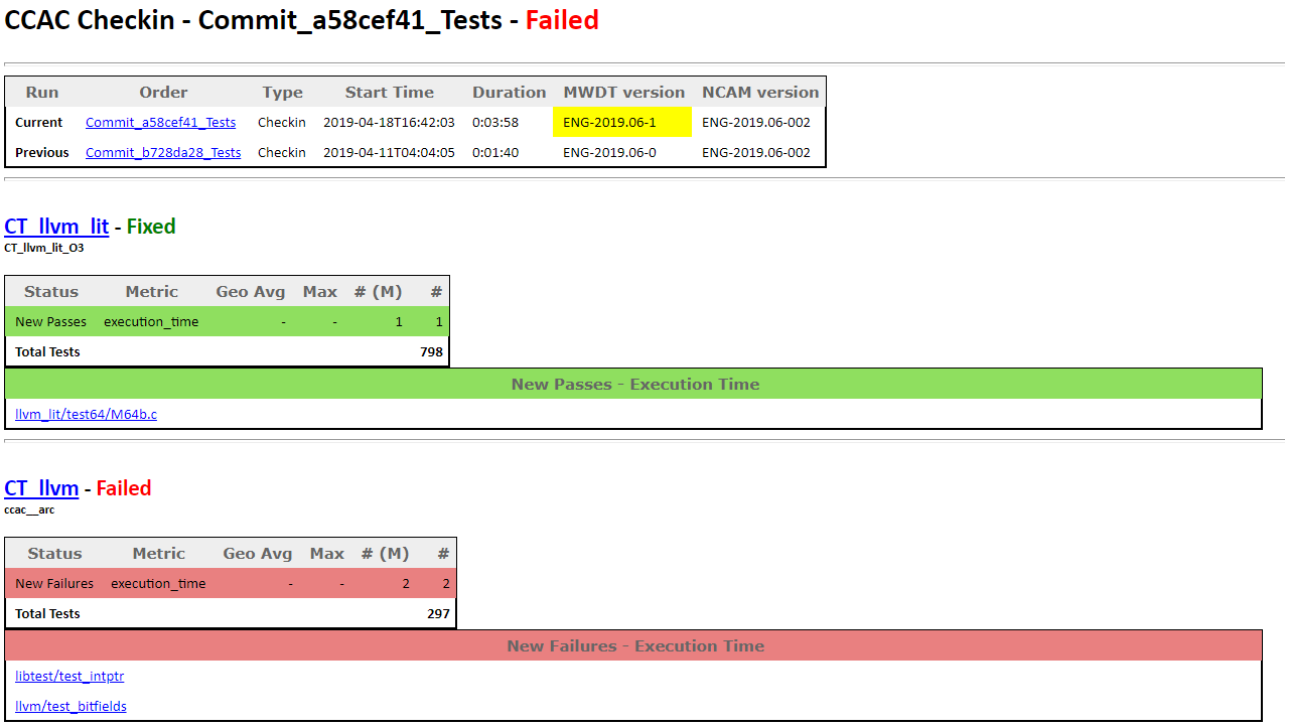
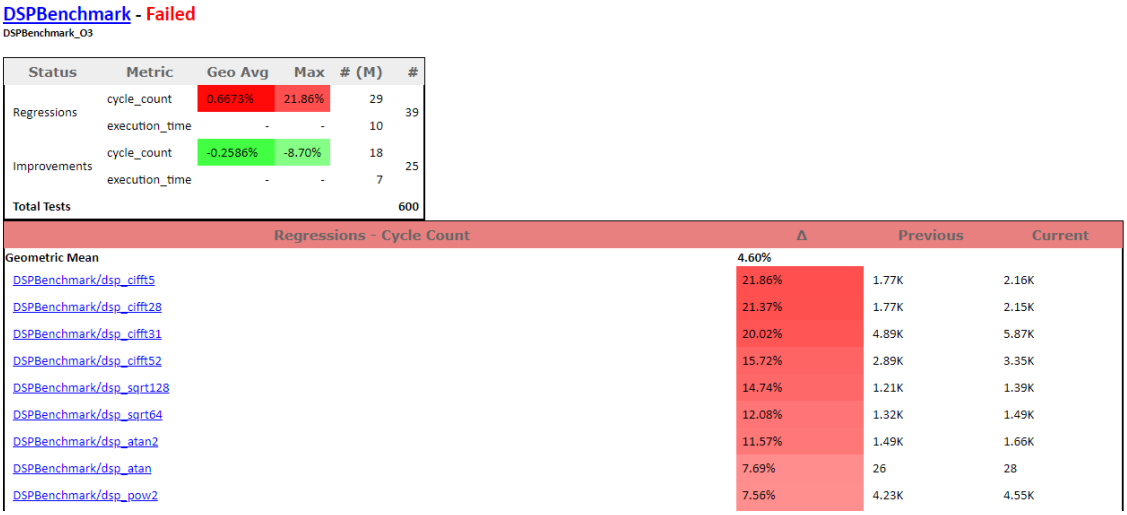


Figure 7: Sample developer report with 2 sections

report complies with declared report specification. It has its header, status,

[1]Corridor (or Hallway) testing – usability testing technique in which randomly selected people (e.g. colleagues) provide their feedback to tune the tested software.

version summary, and 2 sections – each with its own status, summary and data tables.



DSPBenchmark - Failed

DSPBenchmark_O3

Status	Metric	Geo Avg	Max	# (M)	#
Regressions	cycle_count	0.6673%	21.86%	29	39
	execution_time	-	-	10	
Improvements	cycle_count	-0.2586%	-8.70%	18	25
	execution_time	-	-	7	
Total Tests					600

Regressions - Cycle Count	Δ	Previous	Current
Geometric Mean	4.60%		
DSPBenchmark/dsp_cifft5	21.86%	1.77K	2.16K
DSPBenchmark/dsp_cifft28	21.37%	1.77K	2.15K
DSPBenchmark/dsp_cifft31	20.02%	4.89K	5.87K
DSPBenchmark/dsp_cifft52	15.72%	2.89K	3.35K
DSPBenchmark/dsp_sqrt128	14.74%	1.21K	1.39K
DSPBenchmark/dsp_sqrt64	12.08%	1.32K	1.49K
DSPBenchmark/dsp_atan2	11.57%	1.49K	1.66K
DSPBenchmark/dsp_atan	7.69%	26	28
DSPBenchmark/dsp_pow2	7.56%	4.23K	4.55K

Figure 8: Sample section with major regressions

Here we can more interesting section with performance changes – both regressions and improvements. That is why section status is Unstable (according to Table 2).

This report is basically the HTML file delivered by mail to all developers who may be possibly interested in this specific change in the compiler.

6.2 Regression testing

LNT, like most of the projects from LLVM umbrella, uses **lit tests**[1] for regression testing. Alternatively, there are custom shell tests implemented in Bash for Synopsys branch of LNT.

A decision was made to only implement lit tests because of their wider support from LLVM Community and easier maintenance compared to shell tests.

Fairly simple lit tests were developed as a result of this work. They were also Integrated to GitLab CI/CD used in our development environment.

[1]LIT *(stands for LLVM Integrated Tests)* – tool for executing specially-styled regression tests. More information on LLVM regression testing: `https://llvm.org/docs/TestingGuide.html#regression-test-structure`

Conclusion

As a result of this work flexible configurable report generation system was designed and implemented. At the moment of writing it is being used by 15 compiler developers of Synopsys, Inc.

The following tasks were completed:

1. Requirements were collected from compiler developers and analyzed
2. Solution based upon existing LNT infrastructure was designed and implemented
3. The system was integrated into current Jenkins test automation flow
4. Regression tests were developed and the system was evaluated by real users

References

[1] Progress In Digital Integrated Electronics : Rep. / Intel Corporation ; Executor: Gordon E. Moore : 1975.

[2] van Krevelen D.W.F. Augmented Reality: Technologies, Applications, and Limitations // Vrije Universiteit Amsterdam, Department of Computer Science. — 2007.

[3] Shekhar Borkar Andrew A. Chien. The future of microprocessors // Communications of the ACM, Vol. 54 Issue 5. — 2011.

[4] Rogier Wester John Koster. The Software behind Moore's Law // IEEE. — 2015. — March.

[5] Team LLVM. LLVM Project. — 2003. — URL: `http://llvm.org` (online; accessed: 09.05.2019).

[6] Bourque P., Fairley R.E. Guide to the Software Engineering Body of Knowledge, Version 3.0. — IEEE Computer Society, 2014. — URL: `www.swebok.org`.

[7] Alfred V. Aho Monica S. Lam Ravi Sethi Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. — 1986.

[8] Ronacher Armin. Jinja project homepage. — 2014. — URL: `http://jinja.pocoo.org` (online; accessed: 09.05.2019).

[9] Bayer Michael. SQLAlchemy project. The Python SQL Toolkit and Object Relational Mapper. — 2019. — URL: `https://www.sqlalchemy.org` (online; accessed: 09.05.2019).

[10] Dunbar Daniel. LNT Overview // LLVM Documentation. — 2019. — URL: `http://llvm.org/docs/lnt/intro.html` (online; accessed: 09.05.2019).

[11] Synopsys Inc. Synopsys MetaWare compiler. — 2019. — URL: https://www.synopsys.com/dw/ipdir.php?ds=sw_metaware (online; accessed: 09.05.2019).

[12] Foundation Apache. Groovy programming language. — 2019. — URL: http://groovy-lang.org (online; accessed: 09.05.2019).