WHITE PAPER

Protecting Investment in Code Optimisation with Toolchain CI

Linaro Toolchain Team

June 2022

Abstract

Modern compilers usually do an excellent job of producing a good baseline level of performance. In important cases this may not be enough and the skilled developer learns the optimisation techniques that the compiler uses and can work in partnership with it to achieve the highest performance for a particular high throughput or low latency application. The resulting code also comes with a maintenance burden. Optimisations which worked hand-in-hand with the previous compiler version may immediately come unstuck when a new version of the same compiler is released. This happens because compiler developers do not have a clear picture of how their development affects all scenarios or architectures.

This white paper from the Linaro Toolchain Team examines the problem space and the Toolchain CI project. This project carries out automated performance regression testing with the evolving compiler and can immediately flag any regressions to the compiler development community. In this way, performance regressions impacting optimised code caused by new compiler versions can be caught and fixed much more quickly than if they enter an official compiler release. As a result, organisations responsible for optimised codebases can see significantly reduced risk of performance regression when moving to new compiler versions.

Contents

Code Optimisation - an Uneasy Partnership with the Compiler

Software performance is ever more important in our world from the data centre all the way to portable devices. Portable devices are becoming smaller and have to do more with less battery capacity. Software performance and efficiency become increasingly more important in our lives. There's no programming language that can make code run fast, but programming languages can give the programmer a tremendous amount of control and unleash potentially powerful compiler optimization technology [1].

The compiler becomes an active partner in the optimisation process and the developer should take a good compiler and help it to do a great job of optimising the code. There are many optimization techniques that compilers can use, ranging from simple transformations, such as constant folding, to extreme transformations, such as instruction scheduling [2].

The UK's Super Computing Service - HECTOR, and in its Good Practice Guide [3] describes two main avenues which can be followed when trying to optimise an application:

- Optimisations that DO NOT involve modifying the source code (modification may not be desirable): optimisation consists of searching for the best compiler, set of flags and libraries.

- Optimisations that DO involve modifying the source code: in the first instance the programmer must evaluate if a new algorithm is necessary, followed by writing or rewriting optimised code ... If this is not possible the programmer should write the code using techniques that help the compiler to generate a fast executable.

Both cases emphasise the importance of partnership with the compiler. However, there are two thorny issues lurking in this partnership, especially for long-lived codebases. Firstly, compilers are actively developed projects evolving to support new processors, being tuned to better address new use cases, and being supported to fix bugs. Secondly, compiler project teams are completely unable to exhaustively test all the optimisation cases that could be thrown at them, and although the open source compiler projects do an excellent job, regressions do escape and make it into official compiler releases.

The standard practice for testing and benchmarking performance-critical applications is to use compiler and toolchain releases on a bi-annual or annual cadence. It makes short-term efficient use of developers' time, but it leaves little opportunity for addressing performance problems which sneak into new compiler releases. All major open-source compiler communities allow changes to release branches that fix code correctness issues, but they do not risk de-stabilizing release branches with fixes for code-performance problems. Organisations developing code that achieves a hard-won level of performance improvement, almost always through partnership between developer and the compiler, have seen some or all of a particular performance improvement evaporate when they move to a newer version of the compiler.

If performance or code size optimisations to a codebase are a significant part of an organisation's engineering investment, then there are real and demonstrable risks and implications of compiler performance regression.

sidebar

The Continuing Importance of Optimisation

Wirth's Law (also called Page's Law) states "software is getting slower more rapidly than hardware is becoming faster.". This was originally coined by Niklaus Wirth but has been re-affirmed by Larry Page and Sergey Brin of Google [4]. There have been a few variants on the statement that directly tied it to the historic regular increases in hardware performance described in Moore's Law. The general consensus is that Moore's Law is now over, at least in anything like its original form [5]. The implication is that the need for code optimisation, which never went away under Moore's Law, is stronger than ever.

New Compiler - New Performance Regression?

Overview

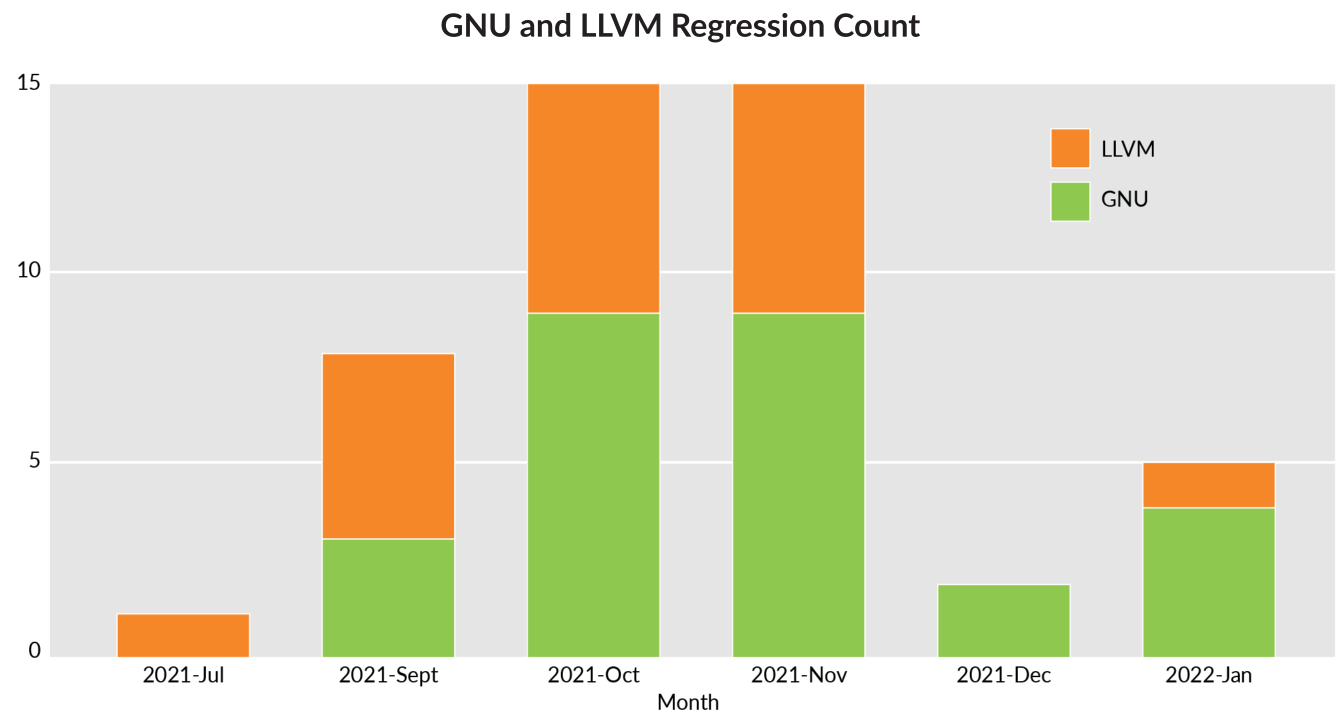
Where hard-fought performance or other efficiency improvements have been implemented in a codebase it's important to look at what circumstances trigger the potential risk to this investment. Typically a new compiler version is likely to be mandated when moving to a new distro version. Functional improvements, meeting commercial or technical standards, the need for security updates and general availability of support drives the cycle of updates to new compiler/distros. It can be a struggle to manage the work involved because of potential regressions in performance or other efficiency issues.

At the same time, the organisation's engineering teams often aren't able or don't have time to follow changes to the open source compilers that might affect them in the future. Regressions in performance due to compiler changes can come as an unpleasant surprise. Even worse, if an organisation reports a regression in the toolchain to the compiler team when they encounter it in a release, it can take a long time to get it fixed and recover the lost performance.

The organisation risks facing a difficult choice of staying on an old compiler and distro for longer, with customers complaining about slow pace of updates, or alternatively accepting the performance hit of moving to the new version.

Maintaining Code Performance Across Arm and x86

The recent emergence of Arm as an architecture in the server and desktop space, manifested by the Amazon Graviton and the Apple M1, has meant that application code and compiler optimisations need to be maintained across both Arm and x86. This is a new and more complex commercial environment and developers are struggling to keep track of work across architectures. There are recent cases of X86 compiler changes having a detrimental effect on Arm code performance, and as stated above, this can only become apparent in an



official release when it's too late to back out the change to the compiler.

Many organisations now have an incomplete picture of compiler behaviour based only on x86 from the days when the Arm architecture was mainly deployed for embedded and mobile, but working across architectures is now mainstream, rather than a niche activity. Making an application developed on x86 run on Arm architectures is, mostly, a one-time effort. Making an application developed on x86 run fast on Arm architectures is a continuous effort – due to developers still, overwhelmingly, using x86 machines for write-test-debug process. This makes continuous integration, testing and benchmarking a must-have requirement for cross-architecture software projects.

The Challenge of Specific Use cases or Configurations

Many engineering businesses benefit from deep knowledge and service of a market niche. Extensive application knowledge in one area is a key differentiator versus new entrants. This specialisation can mean working on very specific workloads for demanding customers. Unfortunately, the more narrow the use case or the more specialist the configuration, the less likely that it will be represented in testing by the compiler community and its performance is therefore vulnerable to upstream changes to the toolchain.

Organisations Shipping Their Own Compilers

High Performance Compute (HPC) is a market where a commercial compiler offer is often carefully tuned to give best performance results on HPC workloads on high performance hardware clusters. Maintaining a compiler is a major and costly undertaking and it's likely that the development team may have to focus on a small set of workloads. Performance testing is a major overhead and without significant automation it can be difficult to track performance regressions vs previous versions of the compiler product. In addition, frequent and extensive testing is necessary to get a good picture of the in-house compiler product performance versus the latest state of open source compiler evolution.

A performance benefit should be the primary reason to develop and maintain a proprietary compiler, but without regularly measuring the performance delta there's no obvious way to differentiate and justify commercial sales.

Keeping on Top of Compiled Code Performance

As shown above, investment made in achieving high performance for an in-house codebase or an external project on internal hardware is potentially at risk from changes as open source compilers evolve. Because of this, organisations can accumulate technical debt with respect to falling performance without realising, and at the point that they move the latest compiler, an unanticipated support burden to fix code for newer compilers can appear.

To address this, a process is needed to de-risk movement to a new toolchain version - typically driven from a move to a new distro. In the new cross architecture world, the process needs specific management to handle the more fluid case of Arm tools and optimisations vs the incumbent x86. It needs to be able to look across performance benchmarks for both x86 and Aarch64 => Add Aarch64 benchmarks. It also needs to be able to track a specific configuration or edge case.

The value from such a process would include providing insurance for investment in code optimisations and a more timely movement to new toolchains or distros.

Key aspects of such a process with respect to the GCC and LLVM open source compilers should be:

- To allow a timely dialogue with the ecosystem if changes are affecting in-house performance
- Get a better understanding of the dynamics involved in compiler changes that can affect the organisation
- Stand a better chance of pushing back on adverse compiler changes instead of re-doing optimisations
- Access to results of relevant (niche) benchmarks

And additionally for any organisation developing in-house commercial compiler products, it can be highly valuable to be able to access an infrastructure/process for continuous testing that gives hard numbers to enable a comparison with open source as a baseline.

Linaro's long-standing involvement in Open Source toolchains

Year	Company milestones	GCC	LLVM
2010	♦ Linaro founded		
2011	♦ Linaro toolchain project processes defined	♦ gcc-linaro-4.6-2011.10 on Arm	
2012		♦ gcc 4.7 ♦ using gcc auto-vectorizer	
2013		♦ Engineering build of gcc 4.8	
2014	♦ NEON testing	♦ eglibc & glibc ♦ gcc 4.9 ♦ Member and product driven gcc optimisations	♦ LLVM community releases and roadmap
2015	♦ Ongoing quarterly toolchain releases ♦ Advanced toolchain usage tutorials ♦ Benchmarking best practice ♦ Performance improvements ♦ GDB and LLDB roadmap		
2016	♦ Undefined Behavior and Compiler Optimizations		♦ Introducing LLVM
2017	♦ ILP32 and FDPIC ♦ SVE	♦ GCC toolchain transition to Arm	♦ LLVM Internals
2018	♦ Coremark regression approaches ♦ String optimization in glibc		
2019	♦ Code size improvement work		
2020			♦ LLVM and GDB contributions
2021	♦ Windows on Arm native development		♦ Reducing LLVM code size on 32-bit Arm targets

Introducing Linaro's Toolchain CI

Project Context and Goals

Toolchain benchmarking and analysis is a key element of the work that Linaro carries out on open source tools for the Arm architecture. It's part of overall toolchain quality and CI efforts and it includes detection of code-speed regressions, code-size regressions and also detection of build/boot breakages. These quality and CI efforts are part of Linaro's open source toolchain community citizenship. They ensure the quality of open source toolchains, bring value across all areas of the Arm Ecosystem and also secure our members' investment in Linaro's own toolchain development work for architecture enablement and Arm architecture optimisation.

In summary the project goals are:

- Secure Linaro members' investment in toolchain optimisation
- Contribute to the overall sustainability of the open source toolchain community
- Bring value to the Arm architecture ecosystem

In order to achieve these project goals, Linaro has worked with our members and with the toolchain community and built a state-of-the-art benchmarking CI which can identify benchmark slow downs and code size increases, automatically identifies regressions down to highlighting a single toolchain commit and is able to track and benchmark many different configurations of upstream toolchains.

Toolchain CI to the Rescue! A Case Study of the CI in use

This is a short case study taken from many examples where an actual regression case was caught by the Toolchain CI after a patch was submitted to the compiler. In this case an optimisation for another architecture (x86) adversely affected the performance of Arm compiled code.

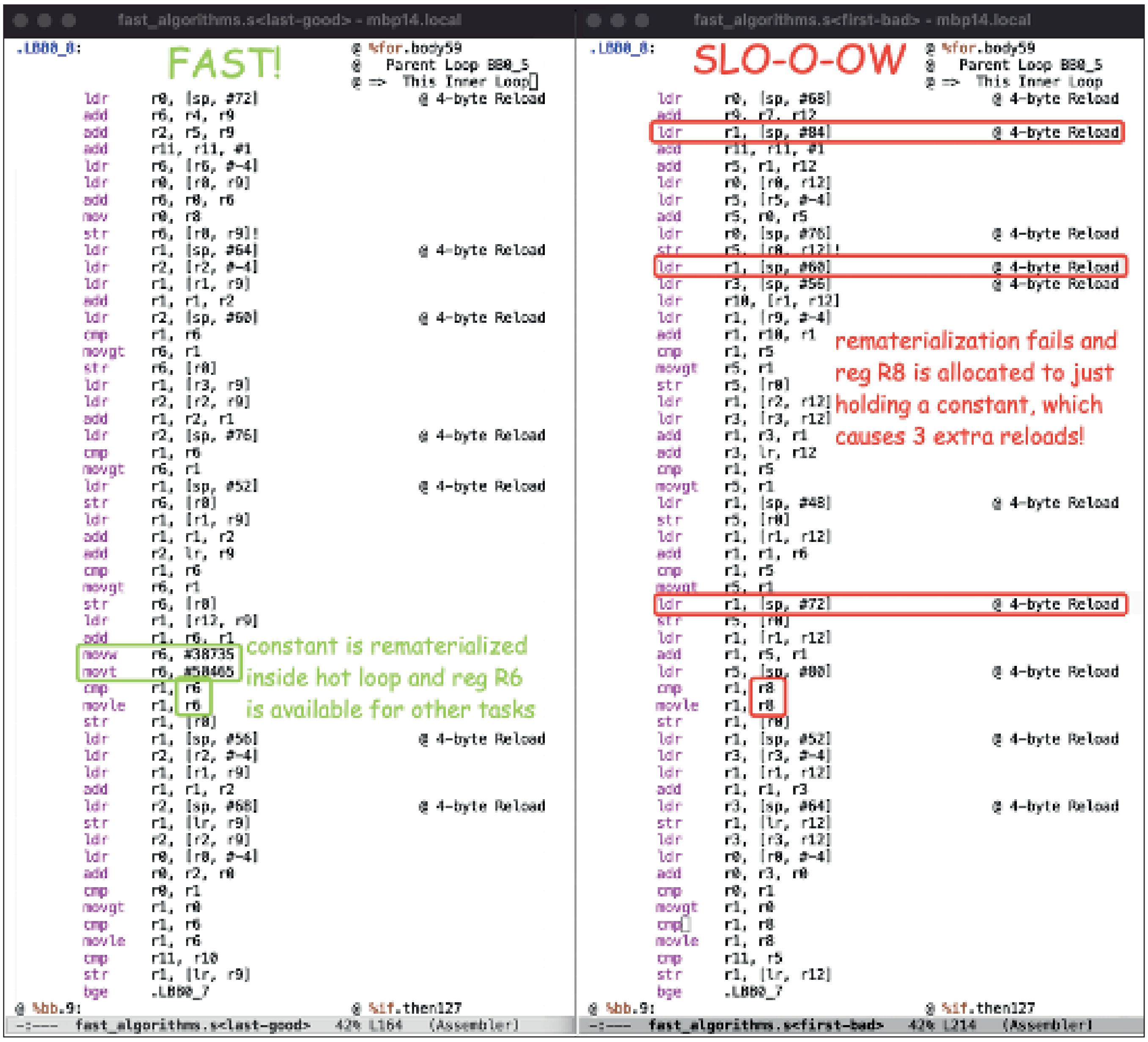
Timeline of a performance regression:

22nd September 2021 - Linaro Toolchain CI detects a 6% slowdown in SPEC CPU2006 benchmarks for LLVM caused by a patch submitted by an x86 developer who was not familiar with the Arm ISA. After bisecting, the CI was able to provide the developer with the last good run (parent commit) and first bad run (failing commit).

23rd September 2021 - Linaro gives some advice to the x86 developer on how the change affected the Arm implementation.

24th September 2021 - the commit is reverted and performance is restored.

Note that the issue was highlighted by toolchain CI and resolved by the community between 22nd and 24th September.


fast_algorithms.s<last-good> - mbp14.local
FAST!
constant is rematerialized inside hot loop and reg R6 is available for other tasks
fast_algorithms.s<first-bad> - mbp14.local
SLO-O-OW
rematerialization fails and reg R8 is allocated to just holding a constant, which causes 3 extra reloads!

```
.LBB0_8:                                        @ %for.body59
                                                @   Parent Loop BB0_5
                                                @ =>  This Inner Loop
        ldr     r0, [sp, #72]                   @ 4-byte Reload
        add     r6, r4, r9
        add     r2, r5, r9
        add     r11, r11, #1
        ldr     r6, [r6, #-4]
        ldr     r0, [r0, r9]
        add     r6, r0, r6
        mov     r0, r8
        str     r6, [r0, r9]!
        ldr     r1, [sp, #64]                   @ 4-byte Reload
        ldr     r2, [r2, #-4]
        ldr     r1, [r1, r9]
        add     r1, r1, r2
        ldr     r2, [sp, #60]                   @ 4-byte Reload
        cmp     r1, r6
        movgt   r6, r1
        str     r6, [r0]
        ldr     r1, [r3, r9]
        ldr     r2, [r2, r9]
        add     r1, r2, r1
        ldr     r2, [sp, #76]                   @ 4-byte Reload
        cmp     r1, r6
        movgt   r6, r1
        ldr     r1, [sp, #52]                   @ 4-byte Reload
        str     r6, [r0]
        ldr     r1, [r1, r9]
        add     r1, r1, r2
        add     r2, lr, r9
        cmp     r1, r6
        movgt   r6, r1
        str     r6, [r0]
        ldr     r1, [r12, r9]
        add     r1, r6, r1
        movw    r6, #38735
        movt    r6, #58465
        cmp     r1, r6
        movle   r1, r6
        str     r1, [r0]
        ldr     r1, [sp, #56]                   @ 4-byte Reload
        ldr     r2, [r2, #-4]
        ldr     r1, [r1, r9]
        add     r1, r1, r2
        ldr     r2, [sp, #68]                   @ 4-byte Reload
        str     r1, [lr, r9]
        ldr     r2, [r2, r9]
        ldr     r0, [r0, #-4]
        add     r0, r2, r0
        cmp     r0, r1
        movgt   r1, r0
        cmp     r1, r6
        movle   r1, r6
        cmp     r11, r10
        str     r1, [lr, r9]
        bge     .LBB0_7
@ %bb.9:                                        @ %if.then127
-:----  fast_algorithms.s<last-good>   42% L164   (Assembler)
```

```
.LBB0_8:                                        @ %for.body59
                                                @   Parent Loop BB0_5
                                                @ =>  This Inner Loop
        ldr     r0, [sp, #68]                   @ 4-byte Reload
        add     r9, r7, r12
        ldr     r1, [sp, #84]                   @ 4-byte Reload
        add     r11, r11, #1
        add     r5, r1, r12
        ldr     r0, [r0, r12]
        ldr     r5, [r5, #-4]
        add     r5, r0, r5
        ldr     r0, [sp, #76]                   @ 4-byte Reload
        str     r5, [r0, r12]!
        ldr     r1, [sp, #60]                   @ 4-byte Reload
        ldr     r3, [sp, #56]                   @ 4-byte Reload
        ldr     r10, [r1, r12]
        ldr     r1, [r9, #-4]
        add     r1, r10, r1
        cmp     r1, r5
        movgt   r5, r1
        str     r5, [r0]
        ldr     r1, [r2, r12]
        ldr     r3, [r3, r12]
        add     r1, r3, r1
        add     r3, lr, r12
        cmp     r1, r5
        movgt   r5, r1
        ldr     r1, [sp, #48]                   @ 4-byte Reload
        str     r5, [r0]
        ldr     r1, [r1, r12]
        add     r1, r1, r6
        cmp     r1, r5
        movgt   r5, r1
        ldr     r1, [sp, #72]                   @ 4-byte Reload
        str     r5, [r0]
        ldr     r1, [r1, r12]
        add     r1, r5, r1
        ldr     r5, [sp, #80]                   @ 4-byte Reload
        cmp     r1, r8
        movle   r1, r8
        str     r1, [r0]
        ldr     r1, [sp, #52]                   @ 4-byte Reload
        ldr     r3, [r3, #-4]
        ldr     r1, [r1, r12]
        add     r1, r1, r3
        ldr     r3, [sp, #64]                   @ 4-byte Reload
        str     r1, [lr, r12]
        ldr     r3, [r3, r12]
        ldr     r0, [r0, #-4]
        add     r0, r3, r0
        cmp     r0, r1
        movgt   r1, r0
        cmp     r1, r8
        movle   r1, r8
        cmp     r11, r5
        str     r1, [lr, r12]
        bge     .LBB0_7
@ %bb.9:                                        @ %if.then127
-:----  fast_algorithms.s<first-bad>   42% L214   (Assembler)
```

It's unfortunately very easy to lose 6% of performance due to a problematic change and very hard to later find 6% performance improvement due to careful (re-) optimisation. If CI had not been available to catch this problem, and the issue appeared in an official release of the compiler, it would have taken many months just to back out the problematic commit.

Note that this is not an uncommon situation and developers working on a particular architecture can't be expected to know the deep details of other architectures or configurations. It's up to stakeholders to be vigilant. Other regressions caught by Toolchain CI for specific benchmarks within the same week (27th Sept - 4th October 2021) were:

- Linaro's Toolchain Working group 470.lbm grew in size by 38% after gcc: aarch64: Improve size ...
- Linaro's Toolchain Working group 400.perlbench slowed down by 6% after llvm: [SimplifyCFG] Ignore ...
- Linaro's Toolchain Working group 471.omnetpp slowed down by 8% after gcc: Avoid invalid loop ...
- Linaro's Toolchain Working group 462.libquantum grew in size by 3% after llvm: [JumpThreading] ...

Value Generated by Toolchain CI Participation

Any organisation which depends on, or has itself invested in toolchain optimisations can immediately gain value from participating in Toolchain CI in order to become best-in-class for up-to-date tools and increase business confidence to continue to invest in optimisations. Specifically, businesses who are now moving to support Arm as well as x86 can be more confident in protecting their investment in code optimisations to be performant on the Arm architecture.

As well as end customers and users benefitting from the latest tools, in-house engineers become much more able to collaborate and share information across architectures and configurations. Through the ability to automatically detect specific problem commits, engineers can maintain a connection to the upstream project community and can feel more in control of events instead of just reacting to the impact of each official release.

Ultimately CI participation makes an organisation better able to target engineering resources where they are needed, and this is particularly beneficial to organisations who also ship proprietary toolchains in such performance-sensitive verticals as HPC.

References

[1] Understanding Compiler Optimization - Chandler Carruth Opening Keynote Meeting C++ 2015

[2] https://docs.microsoft.com/en-us/archive/msdn-magazine/2015/february/compilers-what-every-programmer-should-know-about-compiler-optimizations

[3] http://www.hector.ac.uk/cse/documentation/SerialOpt/

[4] Wikipedia - Wirth's Law

[5] We're Not Prepared for the End of Moore's Law https://www.technologyreview.com/2020/02/24/905789/were-not-prepared-for-the-end-of-moores-law/

How to find out more and participate

You can see examples of the testing output at Benchmarking CI regressions. Take a look at the benchmarks and contact us with any specific questions about the testing and results. For more information on the project contact us on toolchain-ci@linaro.org

There are multiple ways to participate in the project. Various tiers of membership are open to those interested in directly influencing the direction of the project to ensure it delivers the solutions they need. By becoming a member, your engineers get to work with Linaro's team of experts and other industry leaders on scoping and steering the solution. If you are keen to find out more about the Linaro Toolchain Team, you are welcome to visit the team page at linaro.org

About Linaro

Linaro leads collaboration in the Arm ecosystem and helps companies work with the latest open-source technology. The company has over 250 engineers working on more than 70 open-source projects, developing and optimizing software and tools, ensuring smooth product roll outs, and reducing maintenance costs.

Work happens across a wide range of technologies including artificial intelligence, automotive, datacenter & cloud, edge & fog computing, high performance computing, IoT & embedded and mobile. Linaro is distribution neutral: it wants to provide the best software foundations to everyone by working upstream, and to reduce costly and unnecessary fragmentation. The effectiveness of the Linaro approach has been demonstrated by Linaro consistently being listed as one of the top ten company contributors, worldwide, to Linux kernels since 3.10.

To ensure commercial quality software, Linaro's work includes comprehensive test and validation on member hardware platforms. The full scope of Linaro engineering work is open to all online. To find out more, please visit www.linaro.org and www.96Boards.org