EPJ Web of Conferences **214**, 05016 (2019) https://doi.org/10.1051/epjconf/201921405016
CHEP 2018

Porting the LHCb Stack from x86 (Intel) to aarch64 (ARM) and ppc64le (PowerPC)

Laura Promberger[1,*], *Marco* Clemencic[2], *Ben* Couturier[2], *Aritz Brosa* Iartza[3], and *Niko* Neufeld[2] on behalf of the LHCb collaboration

[1]Fakultät für Informatik und Wirtschaftsinformatik - Fachgebiet Informatik, Hochschule Karlsruhe - Technik und Wirtschaft, Karlsruhe, Germany
[2]EP, CERN, Meyrin, Switzerland
[3]Escuela de Ingeniería Informática, Universidad de Oviedo, Oviedo, Asturias, Spain

Abstract. LHCb is undergoing major changes in its data selection and processing chain for the upcoming LHC Run 3 starting in 2021. With this in sight several initiatives have been launched to optimise the software stack. This contribution discusses porting the LHCb Stack from x86_64 architecture to both architectures aarch64 and ppc64le with the goal to evaluate the performance and the cost of the computing infrastructure for the High Level Trigger (HLT). This requires porting a stack with more than five million lines of code and finding working versions of external libraries provided by LCG. Across all software packages the biggest challenge is the growing use of vectorisation - as many vectorisation libraries are specialised on x86 architecture and do not have any support for other architectures. In spite of these challenges we have successfully ported the LHCb High Level Trigger code to aarch64 and ppc64le. This contribution discusses the status and plans for the porting of the software as well as the LHCb approach for tackling code vectorisation in a platform independent way.

1 Introduction

In 2021 the LHCb experiment will undergo a major upgrade for Run 3. With the increased luminosity provided by the LHC and the introduction of new sub-detectors, the LHCb experiment will be able to research new phenomena and known ones more in detail. At the same time this results in an increase of the raw data detector output by a factor of 100 from 50 GB/s to approximately 4 TB/s. And the output data rate of the final selection of the events being written to disk will increase from 0.7 GB/s to 2 - 10 GB/s.

To cope with the large increase of data volume a combination of upgrading the hardware resources of the HLT computing farm and increasing the performance of the software stack is necessary. The increase of performance can be achieved by optimizing the logic of algorithms and exploiting techniques which maximize hardware efficiency. Mainly to be named are multi-threading and vectorisation.

The upgrade of the computing farm will include a new data center and new compute nodes. For the most competitive cost-performance solution it is important to have several

*e-mail: laura.promberger@cern.ch

options. For this LHCb decided to extend the architecture support from Intel x86_64 to ARM aarch64 and PowerPc ppc64le.

1.1 The LHCb software stack

The LHCb software stack is made up of multiple, large projects. These projects can be divided into three groups. The LCG project provides external dependencies (e.g. Oracle, ROOT, Python). On top of these is the experiment-independent project Gaudi which is being used by different experiments at CERN. Last there are the experiment-specific projects LHCb, Lbcom, Rec and Brunel which alone sum up to about five million lines of code.

For this study we used the LHCb stack with Brunel v53r1 [1]. First, with the predefined LCG version 91 [2], but later LCG was upgraded to version 92 [3]. This version of the stack was selected because vectorisation is considered to be the largest challenge for porting the code. The selected version contains less vectorised code than the software stack being currently developed for Run 3. At the same time this version has the disadvantage that it is not multi-threaded, yet.

2 Vectorisation

Before being able to port the software stack, several vectorisation libraries being used by LHCb are evaluated for their cross-platform support. The two libraries used are Vc and Vcl. An overview of their features is shown in Table 1. Both libraries do not have any support for either aarch64 (ARM) or ppc64le (PowerPc). However, Vcl being a light low-level wrapper on top of the intrinsic functions allows an implementation of the missing architectures in a limited time. Whereas Vc is a more high-level approach making its usage easier but with a more complex internal structure. Therefore it was decided for the port to add the cross-platform support to Vcl for all needed functions and replace the usage of Vc by either Vcl or a generic scalar implementation.

Table 1: Vectorisation libraries

	VCL	VC
AVX2	Yes	Yes
AVX512	Yes	In development
ALTIVEC	No	No
NEON	No	In development
DOCUMENTATION	Yes	Yes
EXAMPLES	Yes	Yes
MASKED FUNCTIONS	Shuffle, blend and permute	Nearly every function (`where` construct)
UNSUPPORTED FUNCTIONS AND ARCHITECTURES	No	Officially yes, but not implemented
LICENSE	GPL3 or payed for use in proprietary software	BSD-3-Clause
DISTRIBUTION	Own website[4]	Github[5]

Table 1: Vectorisation libraries

	Vcl	Vc
Main Developer	Agner Fog	Matthias Kretz
Vectorisation Style	Wrapper for intrinsic	Targeted for horizontal vectorisation
Problems	Only support for Intel architecture, not so many masked functions	Does not cover all use cases; hard to implement vertical vectorisation; vector width not always identifiable
Expandability for New Intrinsics	Medium (no unit tests)	Complex
Future		Version 2 will be integrated in the C++ standard

3 Porting to aarch64 (ARM)

The LHCb stack is first ported to aarch64. For LCG it requires changing compile flags and versions of the external dependencies. Some optional dependencies, like Oracle, are not supported on the architecture, so they can be deactivated without creating any problems. For the other projects (Gaudi, LHCb, ...) compile flags also have to be changed. Additionally, all usage of Vc is replaced either by Vcl or a generic scalar implementation.

During the port two major problems occurred. First, there are platform-specific differences when casting `double` to `unsigned int`. Intel does not have a specific instruction to cast from `double` to `unsigned int`. Instead Intel cast from `double` to `signed int` and then reinterprets it to `unsigned int`. ARM on the other hand has an instruction to cast from `double` to `unsigned int`. As a result casting the number -2.3 to `unsigned int` is not valid on ARM. It will result in a floating-point exception stating that the operation is invalid as the value is out of range for `unsigned int`. To solve this problem the behavior of Intel is mimicked explicitly: cast `double` to `signed int` and then to `unsigned int`. The mimicking was selected as enforcing the proper data range would have resulted in an unreasonable amount of code breaking changes for this study. The second problem is also a platform-specific problem. It is about the default expansion of `char`. On Intel `char` is expanded to `signed char` and on ARM it is expanded to `unsigned char`. This resulted within the LHCb stack in a wrongly calculated hash function. To solve the problem the Gcc compile flag `-fsigned-char` has to be used, forcing ARM to behave like Intel: expanding `char` to `signed char`.

After successfully building the LHCb stack on aarch64, the results of the full reconstruction test (Brunel) were validated for their numerical accuracy as the values fluctuate depending on the compiler version and the platform being used. The validation returned that the results are inside the accepted range.

4 Porting to ppc64le (PowerPc)

The port to ppc64le (ppc64le is little endian) is based on the aarch64 port. This is due to prior work showing that changes when porting between aarch64 and ppc64le are smaller compared to the changes required porting from x86_64.

Here, the main work was related to optimizing compile flags for ppc64le and changing guards from checking whether it is an aarch64 platform to checking that it is not a x86_64 platform. Further changes included replacing `__linux` by `__linux__` as the macro `__linux` is not defined on the PowerPc platform. This change mainly affected the external dependencies provided by LCG, for which the changes had to be recorded in form of patches to apply to the original code.

5 Performance

The performance benchmark is the full reconstruction which runs for about 1 to 1.5 hours. This run time length allows to neglect the different performance behavior which can occur when the machine is warming up. Table 2 contains details of the machines used for the benchmark. Depending on their performance the benchmark was either run over 6,000 or 12,000 events to reach the required run-duration. As the LHCb stack in this study is not multi-threaded, multiple processes are started in parallel. NUMA-binding is used to balance the load on all available NUMA-nodes and prevent performance-deteriorating effects due to cross NUMA-node communication.

Table 2: Machines used for the benchmark

	ThunderX2	E5-2630 v4	Power8+	Power9
Architecture	ARM	Intel	PowerPc	PowerPc
Platform	aarch64	x86_64	ppc64le	ppc64le
Compiler	GCC 7.2	GCC 6.2	GCC 7.3	GCC 7.3
Number logical cores	224	40	128	176
Threads per core	4	2	8	4
Cores per socket	28	10	8	22
Sockets/Numa nodes	2	2	2	2
Ram (GB)	256	64	256	128
Largest intrinsic set	NEON	AVX2	Altivec	Altivec
Cpu performance	top-notch high-tier	cost-efficient mid-tier		

The scalability of the stack is shown in Figure 1. Here, the Intel E5-2630 v4 and the POWER8+ have the same speed-up even though the Intel E5-2630 v4 has four more real cores. Whereas the ThunderX2 and the POWER9 have a slower speed-up in the beginning but on the long run better performance due to the higher number of cores. The measurement of the POWER9 does only cover around half of the all the logical cores due to limitations of the amount of RAM currently available in the machine.

Furthermore, for the Intel E5-2630 v4 and the Cavium ThunderX2 a cost-performance estimation is done. The Power8+ is not part of it as it is a machine being part of IBM's "scale-up" offering, and not the cloud-oriented "scale-out" offering. It is designed for machine learning on GPUs and therefore as such the results cannot be representative for systems which might realistically enter the data-centre. While to have a and therefore of no design which would be considered for the computing farm. And for the Power9 the price was not known. The results are shown in Figure 2. The numbers are in favor for Intel. But there are three things to be considered: One, the ported version of the stack uses less vectorisation

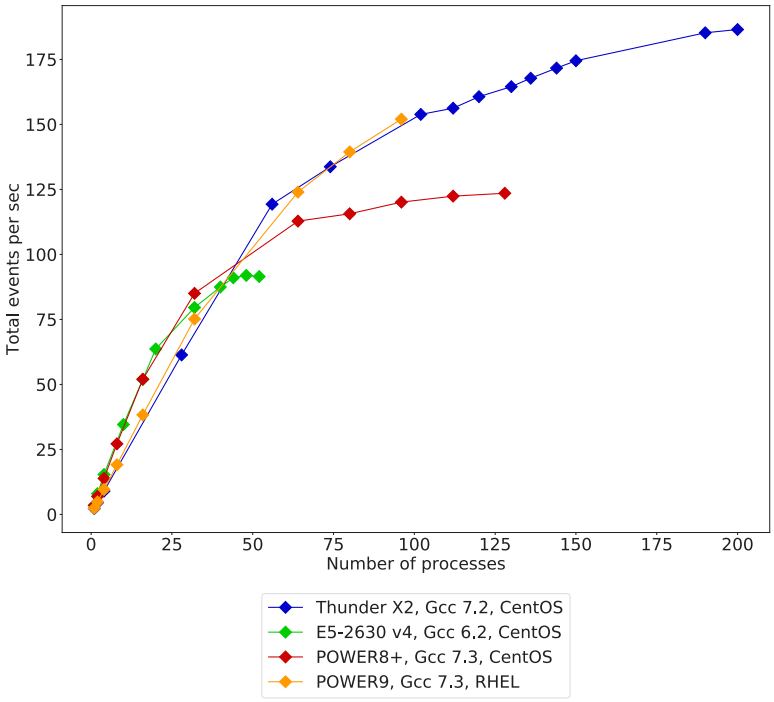

Figure 1: Performance based on average time of one event per process

as some parts were not replaced by Vcl but by a generic scalar code. Two, the stack is not multi-threaded which prevents positive effects of cache-locality. It therefore penalizes a high number of hyper-threads and makes machines with many cores expensive as each process requires about 1.5 GB RAM. This disadvantages the ThunderX2 as it has 224 logical cores and four hyper-threads compared to 40 logical cores and two hyper-threads of the E5-2630 v4. Three, the pricing for buying the machines is not fair. The E5-2630 v4 is a middle tier machine especially selected for its cost-performance ratio and was bought in a bulk of several hundreds of servers. Whereas the ThunderX2 is the top-notch high tier solution from Cavium and the price of it is the price for buying a single machine. Nonetheless, the numbers are not off by factors. They suggest that in a competitive tender both, ARM and Intel, could be viable options.

6 Future work

The main goal is to have a cross-platform support in LHCb software stack for Run 3. However, to be able to do this the major problem of cross-platform vectorisation has to be solved. For this aside from Vc and Vcl additional vectorisation libraries like UMESIMD and boost.simd where explored for their long-term maintainability, functionality and cross-platform support. None of those libraries was able to fulfill all requirements.

Inspired by ROOT, LHCb is looking into a new solution named VecCore [6]. VecCore is a wrapper on top of multiple different vectorisation back ends. It is actively developed but still in an early-stage. VecCore will provide a uniform interface to use UMESIMD, Vc, Cuda or a scalar implementation as back end. ROOT plans to replace the direct Vc support by

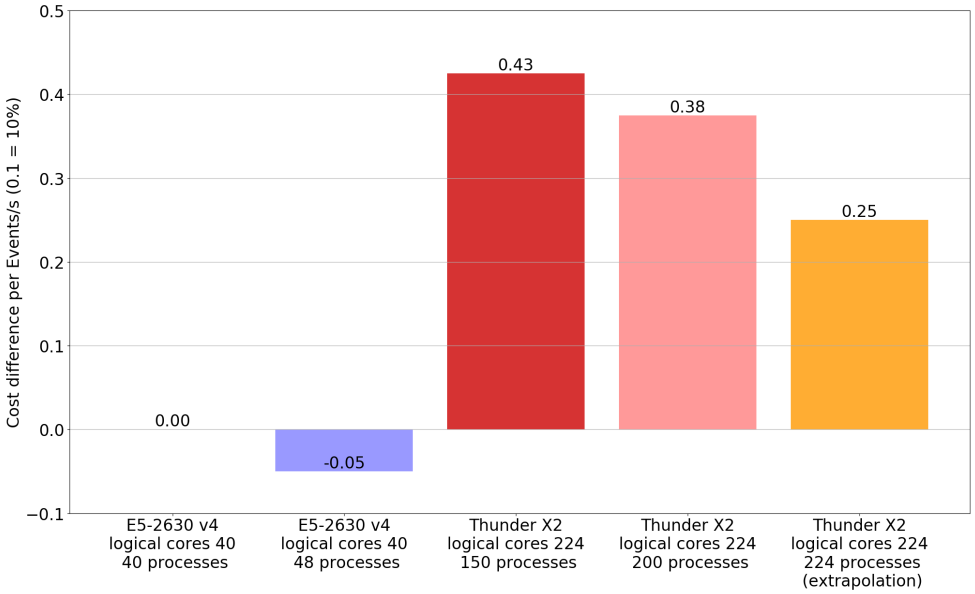

Figure 2: Cost-performance estimation based on the price for buying a server

VecCore. And LHCb has done a first pilot study on the original LHCb stack which was also used to do this cross-platform support study. Next steps will be to introduce VecCore to the development teams of the sub-detectors. So that missing functions are detected early enough to be able to be integrated in VecCore in time for the tender of the new HLT computing farm.

Further future work is a detailed analysis of the power-consumption for the different architectures. Such an analysis would improve the quality of the measurement of competitiveness.

7 Conclusion

This study is the first step to enable the LHCb collaboration to run their Run 3 HLT computing farm not only on Intel x86_64 machines but also on ARM aarch64 and PowerPc ppc64le. Throughout the work the first analysis that cross-platform support for vectorisation will be a major problem was proven to be true. However, as this study could prove that a port is feasible and that the cost-performance difference between the Intel E5-2630 v4 and the ARM Cavium ThunderX2 does not differ by factors we are confident that continuing this work will benefit the diversity and competitiveness of the tender for the HLT computing farm.

References

[1] L. Brunel, *Files · v53r1 · LHCb / Brunel · GitLab*, [Online; accessed October 11, 2018], `https://gitlab.cern.ch/lhcb/Brunel/tree/v53r1`

[2] LHCb, *Files · LCG_91_aarch64 · LHCb Core Software / lcgcmake · GitLab*, [Online; accessed October 11, 2018; LCG 91 with aarch64 modifications], `https://gitlab.cern.ch/lhcb-core/lcgcmake/tree/LCG_91_aarch64`

[3] LHCb, *Files · LCG_92_ppc64le_based_on_aarch64 · LHCb Core Software / lcgcmake · GitLab*, [Online; accessed October 11, 2018; LCG 92 with aarch64 and

ppc64le modifications], https://gitlab.cern.ch/lhcb-core/lcgcmake/tree/LCG_92_ppc64le_based_on_aarch64

[4] A. Fog, *Software optimization resources. C++ and assembly. Windows, Linux, BSD, Mac OS X*, [Online; accessed January 19, 2018], http://www.agner.org/optimize/

[5] M. Kretz, *GitHub - VcDevel/Vc: SIMD Vector Classes for C++*, [Online; accessed January 19, 2018], https://github.com/VcDevel/Vc

[6] root project, *GitHub - root-project/veccore: SIMD Vectorization Library*, [Online; accessed March 8, 2018], https://github.com/root-project/veccore