HL-LHC-UK-2017
02/10/2017

M1.4 Milestone Report: Profiling MERLIN

S. Rowan[1], S. Tygier[2], R. Appleby[2], and R. Barlow[1]

[1]University of Huddersfield
[2]University of Manchester

Abstract

MERLIN, a C++ particle tracking software package, design-focused for LHC collimation studies, has been thoroughly profiled to identify and quantify the severity of any performance and/or sustainability issues. Identification of such hotspots allows for code refactoring and/or redesign priorities to be defined, improving the efficiency of the development process. The following report details all profiling results as well as the developers intentions to improve the code base, accordingly.

1 Objectives

1. Profile the MERLIN code base to identify and quantify the severity of any performance and/or sustainability issues.
2. Establish code reform/refactoring priorities based on profiling results.

2 Introduction

MERLIN is a C++ accelerator physics library (∼26,500 logical lines of code across 438 source files), designed for accelerator-scale particle tracking simulations [1]. MERLIN was originally developed at DESY, Germany, for International Linear Collider (ILC) studies [2]. In more recent years, *circa* 2009 onwards, MERLIN has become design-focused for High-Luminosity Large Hadron Collider (HL-LHC) collimation studies [3]−[5]. Recent developments include implementation of circular accelerator lattice structures, proton tracking, single-diffractive scattering physics and both conventional (dynamic aperture jaw-based) and advanced (Hollow Electron Lens (HEL)) collimation options [6].

Code Sustainability

The increase in code functionality is inherently accompanied by an increase in the size of the code base, bringing with it a number of additional challenges. In particular, code complexity, readability and maintainability, *i.e.* code sustainability as a whole, all become more relevant. As it is the intention to maintain MERLIN for long-term use in both HL-LHC and Future Circular Collider (FCC) studies, *i.e.* order of decades, code sustainability is not only an important topic of discussion and investigation, but fast becoming a necessary one.

Code Profiling & Refactoring

It is the intention of the MERLIN developers to identify and remove/improve upon any performance and/or sustainability issues within the code base. Conventional (memory leaks and execution/function call times) as well as more advanced (cyclomatic complexity, dependency crossings, interface instabilities) profiling analyses have been carried out with the intention of quantifying the severity of such issues.

3 Benchmarking

Prior to profiling, it was important to properly benchmark the software, establishing a baseline set of compiler flags.

Representative and Consistent Testing

All benchmarking was carried out on a representative particle tracking and collimation simulation lhc_collimation_test.cpp. lhc_collimation_test constructs the full HL-LHC lattice frame, with all known aperture dimensions and utilizes HL-LHC v1.2 magnet optics settings. A restricted number of particles per simulation, N_{p}, are tracked around 10 turns, with collimators set to deliberately interact with the beam halo such that all forms of scattering are likely involved. $N_{\mathrm{p,\,max}}$ is CPU dependant and is defined by as the maximum number of particle that can be simulated without encountering slow-down due to CPU cache limitations.

Accurate run-time statistics are acquired by running the above-mentioned test 10 times on the same system for each configuration, presenting only the mean and standard deviation for each data set. Furthermore, due to processer warm-up times, potential inconsistencies and small variations in available memory resources for each run, the first test is always excluded and any further outliers are removed in accordance with Tukey's method. The same single physical CPU core is assigned the process each time to prevent variations in memory latency.

Number of Particles Per Process

There are a number of factors to consider when determining $N_{\mathrm{p,\,max}}$, such as the start-up costs from loading input files, constructing the lattice and setting-up cross-section tables. There are also other overheads that are independent of N_{p}, such as iterating through the lattice and checking which physics processes to apply. As a result, when N_{p} is small, the start-up time and overheads dominate, which would suggest that larger N_{p} simulations would be more efficient. However, one will quickly run into the aforementioned CPU cache limitations, due to the memory required to store each particle's spacial and momenta parameters. As a result, to determine the optimal number of particles per process, a parametric sweep of N_{p} was carried out on several systems with varying CPU cache sizes and the tracked particles/second was measured. Results are shown in Figure 1. Note that, as the memory required to store particle information exceeds L3 cache limits, some particles' information get stored in main memory, resulting in greater latency costs and a saturation in tracked particles/second.

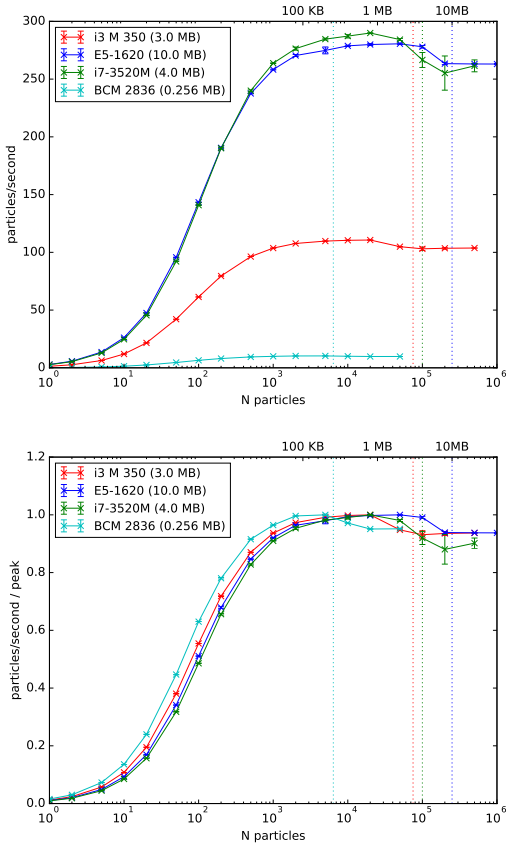

Figure 1: Absolute (top) and normalized (bottom) tracking speed for a range of CPUs with different L3 cache sizes. L3 cache size shown in brackets.

Keeping the storage size of the bunch within CPU cache limits is shown to give a 5-10% performance gain. Note that for benchmarking, only a single CPU core was used. In practice, cache is shared between all CPU cores and the results would vary accordingly.

Optimization of how particle bunch vectors were stored and updated was carried out to improve cache limits by preventing unnecessary bunch copies being made. A brief review of the study, including a before/after comparison can be found in Appendix A.

Compiler Optimization Flags

Modern compilers have an array of auto-optimization options available. To establish a baseline for benchmarking MERLIN, a series of comparisons between various default optimization compiler flags was carried out. The compiler used was gcc 6.3.0.

The default optimization flags, -O0 (no optimization), -O1 (minimum optimization), -O2 (nominal optimization) and -O3 (full optimization) were compared. It was found that relative to -O0 (∼407 s), -O1 was considerable faster averaging ∼ 73.7 s. -O2 was 1.8% faster still (∼ 72.4 s) but use of -O3 was in fact detrimental to the order of 6%. This is not uncommon, however, as some -O3 optimizations are unstable for some programs and may also increase the binary size. Results are shown in Figure 2. Overall, it was decided that profiling be carried out using the -O2 flag.

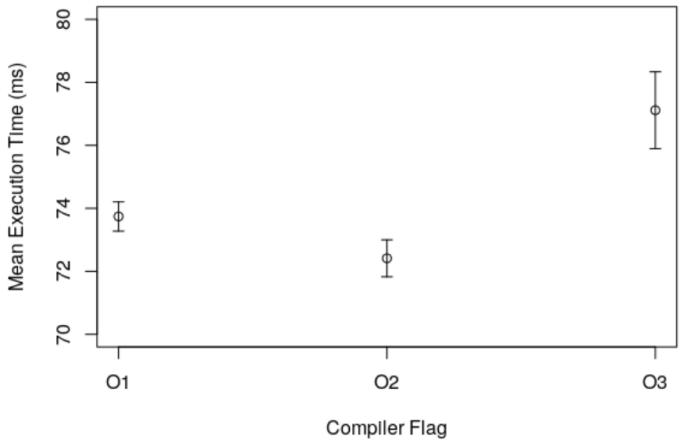

Figure 2: A comparison of gcc compiler optimization flags. Optimization flags show -O2 to be ∼0.5% and ∼6% faster than -O1 and -O3, respectively. -O0 results (mean ∼407 ms) not show to prevent obfuscation.

Note that both -O2 and -O3 include some auto-vectorization flags. Looking at -O2 specifically, *i.e.* no detrimental optimization options, the gain in performance from auto-vectorization is minimal. This shows that MERLINs main algorithms do not explicitly take advantage of this process. In theory, packaged-vectorization scales linearly with the available vector register bit-size. In the case of AVX and AVX-512 compatible CPUs, 256/512 -bit vector registers, there are significant performance gains to be made, ∼4-8x [8]. This will also scale with future hardware advances. Explicit use of vectorization throughout MERLIN will be a focus of investigation for performance improvements in general.

A full compiler flag study, looking at all flag options individually, will be carried out post code reform/refactoring.

Parallelization

MERLIN's tracker does not implement intra-bunch particle-to-particle coupling. As a result, simulating high numbers of particles (order of $N_{\mathrm{p}}{\sim}10^8$) is easily and consistently parallelizable by means of multi-process computing. Simulations are split into N_{j} jobs of $N_{\mathrm{p,\,max}}$ particles and assigned to a batch system such as HTCondor [15]. The cumulative loss-map result is acquired by summing the output loss-maps of each batch job. Performance gains adhere to weak scaling efficiency levels assuming similar compute power among batch processors. However, this is not normally the case in practice.

4 Profiling

Profiling software allows one to quantify code quality and identify detrimental hotspots via static and dynamic code analysis. Profiling is an important part of the development process as it provides a means for establishing reform/refactoring priorities. All static and dynamic analyses were carried out on dual-socket Intel Xeon E5-2620 v4 workstation, running Ubuntu 17.04. Code testing was carried out on mutiple different x86-64 systems across various operating systems.

4.1 Static Analysis

Static analysis was carried out using metriculator [9] to quantify, amongst others, logical source lines of code, cyclomatic complexity and efferent coupling. All analyses were carried out at the member function level. For perspective, MERLIN contains 5,332 member functions.

Logical Source Lines of Code

An excessive number of logical source lines of code (LSLOC) for a given member function can greatly decrease readability and maintainability. Convention advises that all member functions are structured such that the LSLOC remains

below 20 (strict) or 50 (lenient). The 6 least-conforming member functions are presented in Table 1. In total 224 (4.2%) member functions exceed strict limits, 48 (0.9%) exceed lenient limits and 14 exceeding 100+. A lot of improvements could be made in this area. It is the intention of the developers to look into restructuring all member functions exceeding 100+.

Table 1: List of member functions with highest numbers of logical source lines of code (LSLOC).

Member Function/Constructor	LSLOC
MADInterface::ReadComponent	253
ApertureConfiguration::ConfigureElementApertures	208
MaterialDatabase::MaterialDatabase	207
GenSectorBendTM	195
CollimatorDatabase::ConfigureCollimators	183
CCFailureProcess::DoProcess	174

Cyclomatic Complexity

Cyclomatic Complexity defines how complex as source file is from the perspective of the CPU. Minimizing code complexity is important for avoiding branch mispredictions as well as improving readability. Complexity can be increased by excessive use of if and switch statements and/or convoluted object-orientation. Cyclomatic Complexity is quantified in metriculator by the McCabe number [10]. Convention advises that a member functions' McCabe number should not exceed 15 (strict) or 20 (lenient). The 6 least-conforming member functions are presented in Table 2. In total 30 (0.56%) member functions exceed strict limits, 23 (0.43%) exceed lenient limits and 10 exceed 30+. Although the percentage of non-conforming functions is considered low, code complexity can affect both maintainability and performance. Furthermore, several of the least-conforming functions are also noted to have a high LSLOC. As a result, this is likely to be a priority in refactoring/code-redesign. It is the intention to bring all member functions' McCabe value below 30.

Table 2: List of member functions with highest levels of cyclomatic complexity (McCabe number).

Member Function/Constructor	McCabe
CCFailureProcess::DoProcess	82
MADInterface::ReadComponent	82
ApertureConfiguration::ConfigureElementApertures	66
svdcmp	58
collimatortable	41
CollimatorParticleProcess::DoCollimation	40

Efferent Coupling

Efferent Coupling looks at the dependency of certain class on externally defined classes. The more external classes required to construct a member function, the higher the efferent coupling value. High levels of inter-class dependencies decrease both maintainability and modularity. The 6 highest dependent member functions are presented in Table 3. Convention advises that no member function exceeds a dependency value of 5 (strict) or 10 (lenient). Notably, only one member function in MERLIN exceeds the strict limit and from this perspective MERLIN has been well designed.

Table 3: List of member functions with highest levels of efferent coupling.

Member Function/Constructor	Efferent Coupling
LinearFBSystem	5
ExtractTypedComponents	4
Klystron	4
MixtureProperties	4
ApplyATL	3
ATL2D	3

4.2 High-Level Dynamic Analysis

High-level dynamic analysis was carried out using Valgrind [11] to identify hotspots in function call times/frequency (callgrind) and memory leaks (memcheck). All high-level dynamic analysis was carried out on the aforementioned lhc_collimation_test, tracking 8,000 particles around 10 turns.

Function Call Times/Frequency

Analysing member function call times/frequency allows developers to prioritise any code refactoring/redesign from the perspective of performance improvements. The top 6 member functions in terms of percentage of total execution time are presented in Table 4. Standard library functions are excluded.

RTMap::Apply and RMap::Apply, accumulate over 46% of the total execution time, with over 1.35 billion function calls each. This is, however, as expected since the Apply functions update the vectors containing all particles' spacial and momenta parameters. Nonetheless, optimizing of the algorithms within is under investigation as small improvements may translate to significant performance gains overall.

The next observation lies with the two PointInside aperture functions. These functions check whether a particle remains with the adjacent aperture dimensions and are also called 1.35 billions times cumulatively. Note, however, that the interpolated aperture equivalent accounts for a significantly higher percentage of total execution time (4.29% with respect to 2.59%), whilst only being

Table 4: List of member functions call times in terms of percentage of total execution time and number of function calls for test lhc_collimation_test.

Member Function/Constructor	% of Total	No. Calls
RTMap::Apply	31.64	1,387,144,890
RMap::Apply	15.03	1,387,144,890
InterpolatedAperture::PointInside	4.21	168,069,641
ppDiffractiveScatter::PomeronScatter	3.29	56,861,753
RectEllipseAperture::PointInside	2.59	1,113,640,268
CollimatorParticleProcess::DoCollimation	2.32	463,081

called ∼15% of the time (0.167 to 1.1 billion). It is likely that significant performance gains could be made in optimization these algorithms. An example of improving such an algorithm is shown in Appendix B.

Finally, PomeronScatter and DoCollimation are also notably high in percentage of execution time, while being low in call count. It is also likely that significant performance gains can be obtained by revisiting and optimizing the algorithms within.

Memory Leaks

Memcheck results recorded 25 memory records, including 3 condition jump/move operations based on uninitialized values and 22 memory loss records of up to 2 MB. Focusing on the loss records, the 6 greatest losses and corresponding member functions are presented in Table 5.

Table 5: List of recorded memory losses when running lhc_collimation_test.

Origin Member Function/Constructor	Memory Loss [bytes]
TTrackSim::TTrackSim	2,178,643
ApertureConfiguration::ConfigureElementApertures	1,027,080
ApertureConfiguration::ConfigureElementApertures	786,864
ApertureConfiguration::ConfigureElementApertures	332,608
ApertureConfiguration::ConfigureElementApertures	286,656
CollimateParticleProcess::DoCollimation	178,020

Notably, 4 of the 6 loss record shown are due to losses within ApertureConfiguration::ConfigureElementApertures, a member function which far exceeded several static analysis minima and is clearly a priority for refactoring/redesign. The other losses shown are the result of member functions TTrackSim, which is responsible for tracking the particles movements and DoCollimation, another member function which has failed previous tests. Garbage collection of overwritten tracking information may not be implemented for these function. It is the intention to address all of the recorded memory issues in time. In particular, utilizing C++11's unique and shared pointers for automatic garbage collection.

4.3 Low-Level Dynamic Analysis

Low-level dynamic analysis was carried out using Intel's VTune Amplifier for cycles per instruction analysis and Valgrind's cachegrind tool for cache misses and branch mispredictions.

Cycles Per Instruction

Cycles per instruction (CPI) is a useful metric for determining if a software package is utilizing instruction level parallelism by means of superscalar operations and pipelining. In modern processors, it is possible for more than one arithmetic logic unit to be assigned an operation within a single CPU cycle. Today's CPUs have access to four ALUs allowing four simultaneous operations and a theoretical CPI of 0.25. A CPI greater than 1 is considered inefficient. One example of how to explicitly take advantage of instruction level parallelism is for-loop unrolling [12]. The 6 least-conforming member functions from the CPI analysis are presented in Table 6.

Table 6: Cycles per instruction as identified by Intel's VTune Amplifier.

Member Function/Constructor	CPI
ppDiffractiveScatter::GenerateDsigDtDxi	4.833
AcceleratorComponent::GetLength	3.0
InterpolatedCircularAperture::PointInside	2.0
OneSidedUnalignedCollimatorAperture::PointInside	2.0
SectorBendTM	2.0
ProcessStepManager::Track	2.0

Overall, 15 member functions utilized in the lhc_collimation_test are noted to have a CPI greater than 1, with several exceeding 2+. The average CPI for the test is 0.533, which is consider very good. Nonetheless, it is the intention of the developers to improve upon all functions with a CPI greater than 1.

Cache Misses/Branch Mispredictions

Modern CPUs speculate by design. CPU speculation comes in two forms, data speculation, *i.e.* prediction of which memory locations will be accessed next, and control speculation, *i.e.* prediction of the most likely path through an algorithm. Speculation can be greatly beneficial if correct, but also detrimental if incorrect. To minimize cache misses and branch mispredictions, code should be written to have low complexity and be inherently predictable. The results of the valgrind-cachegrind analysis are presented in Tables 7 and 8. Note that cachegrind only recorded events in the first and last levels of cache, in this case, L1 and L3, respectively.

Once again, several of the functions producing the most speculation misses have been noted for failing other profiling tests. The majority of speculation

issues, however, were from standard library functions such as std::pow and std::vector::operator. It is likely worth investigating alternatives for these functions where possible.

Table 7: Speculation misses as identified by Valgrind's cachegrind tool.

Speculation Issue	Number Misses
Branching Reference	607,298,772,259
L1 Branch Mispredictions	9,314,608 (0.0015%)
L3 Branch Mispredictions	7,633 (0.0000013%)
Caching reference	303,787,094,963
L1 Cache Misses	3,904,905,272 (1.29%)
L3 Cache Misses	259,928 (0.000085%)

Table 8: Most problematic member functions in terms of cache misses.

Member Function	Cache Misses
ParticleTracking::...::DoCollimation	892,178,216
RMap::Apply	495,060,101
RTMap::Apply	369,647,043
ParticleTracking::...::ApplyDrift	177,389,390
ApertureConfiguration::ConfigureElementApertures	96,387,437
ParticleTracking::...::ApplyMap1	59,482,748

4.4 Code Testing

Code testing is key in maintaining a working code base. It is a method of ascertaining that new code has been developed and investigated correctly. The following discusses the current MERLIN test suite.

Test Suite & Code Coverage

In the recent months, the MERLIN developers have pushed for more code tests to be written. There are now 16 (previously zero) main tests, which cover a wide range of aspects of the code from simple aperture and particle bunch tests, through to representative particle tracking and collimation tests. The test suite can be called easily and used as verification of a correct installation via the cmake build option. A view of this being carried out is shown in Figure 3.

The cumulative code coverage of the above tests is ∼30%. Although this value is considered very low, it was decided that backdating tests for the entire code base would be too time consuming and that tests should only be written when new classes or applications are being developed.

The test suite is automatically run on a nightly basis on several server systems with various architectures (x86/ARM) across various operating systems

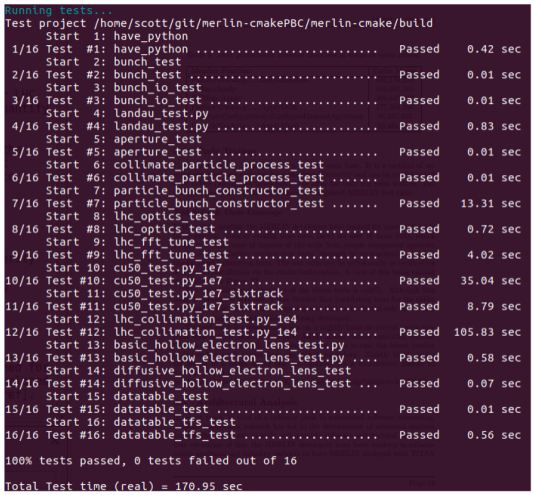

Figure 3: MERLIN nightly cmake build test suite.

(Fedora/Ubuntu/CentOS). Each time the test suite is run, the latest version of the code base is cloned from the github repository. Nightly build results are stored in a cdash server and are viewable via a web-browser, similar to SixTrack/MAD-X.

url: https://abp-cdash.web.cern.ch/abp-cdash/index.php?project=MERLIN

4.5 Architectural Analysis

In view of an industry push towards automated software refactoring, software engineering research has led to the development of advanced analysis frameworks which can identify and quantify software architectural issues. To take advantage of this, the MERLIN developers have been working in collaboration with academic partners at Drexel University, PA, USA, to have MERLIN analysed with their in-house DV8 (previously TITAN) architecture analysis framework [13], [14]. Architectural analysis can be vital in identifying otherwise unnoticed instabilities within the fundamental structure of a code base.

Architectural Measures

Propagation Cost (PC) aims to measure how tightly coupled a system is by manipulating the design structure matrix (DSM) of a project's dependencies, adding indirected dependencies until no more can be added. A lower PC is considered better.
Decoupling level (DL) measures how well an architecture is decoupled into modules. If a module influences all other files either directly or indirectly in lower layers, its DL is 0. Conversely, if a modules influences no other files, its DL is 1. A higher DL is considered better.

MERLIN's base-level architectural analysis results are presented in Table 9. Values presented are static, looking at the most recent version only, and dynamic, taking into account all logged changes on github.

Table 9: Decoupling level and Propagation Cost analysis for both current and historical MERLIN updates.

MERLIN	DL	PC
Current Version (Static)	0.79	0..11
With History (Dynamic)	0.27	0.11

The PC is low in both static analysis and with history and in this regard MERLIN is considered well structured (this compliments the Efferent Coupling analysis). DL notably only has a good score in the static analysis and has a potentially problematic score when taking into account the history. This suggests there has been a continual need to update fundamental files and is likely due to the presence of a numbers of architectural flaws.

Architectural Root Analysis

Architecture Roots are the groups of files responsible for the most changes in the system. These architectural roots contain files (and their relations) that have the most impact on the maintainability of the system, *i.e.* are error-prone and/or change-prone. All files within a given root are inherently coupled.

The four most impactful roots defined by the DV8 analysis are presented in Table 10. The main four roots account for 75% of the error/change-prone files in the code base. As a result, it is highly likely the change-proneness propagates with each bug-fix/update.

Table 10: Most impactful architectural roots within MERLIN as defined by DV8 and the corresponding coverage of error/change prone files (bugs).

	No. of Files	'Bug' Coverage (cumulative)
Root 1	183	0.39
Root 2	68	0.57
Root 3	46	0.68
Root 4	71	0.75

Architectural Debt

Architectural Debt is defined as the extra maintenance effort caused by the flawed relations among Architectural Roots. Architectural debts are quantified

by looking at the number of added/amended lines of code for each root. A list of the architectural debt metrics analysed and quantified by DV8 are as follows.

- *Clique* (moderate severity): A group files that are interconnected, forming a strongly connected 'component' but not belonging to a single module.
- *Package cycles* (moderate severity): Typically the package structure of a software system should form a hierarchical structure. A cycle among packages is therefore considered to be harmful.
- *Improper inheritance* (moderate-high severity): An inheritance hierarchy is considered problematic if it falls into one of the following cases: (1) a parent class depends on one or more of its children, (2) the client of the class hierarchy uses/calls both a parent and one or more of its children, violating the Liskov Substitution Principle.
- *Crossing* (moderate-high severity): If a file has many dependants and it also depends on many other files, this file is considered a crossing.
- *Unstable interface* (high severity): If a highly influential file is changed frequently with other files that directly or indirectly depend on it, it is likely an unstable interface.

The architectural debt analysis results are presented in Tables 11−14. There are a significant number of identified architectural issues. In particular, a number of unstable interfaces (high severity), a high number of improper inheritances (moderate severity) and a high number of crossings (moderate-high severity).

In looking in more detail at improper inheritances, crossings and unstable interfaces, it is clear that fundamental classes PSvector information) and AcceleratorComponent are found to be problematic at an architectural level. Both classes are found to be unstable interfaces and are also at the flagged with regard to other architectural issues. PSvector is seen to have 13 counts of improper inheritance, which can be highly detrimental to performance as well as maintainability. AcceleratorComponent is found to be the center of a crossing involving 55 other files. Due to the fundamental nature of these classes, it is possible that a great number of the issues mentioned in the report are as a result of a underlying instabilities within such fundamental classes. It is the intention of the developers to revisiting these fundamental classes, stabilizing them by means of refactoring and multiple levels of abstraction. Other classes which are likely to be revisited and similarly improved upon are ModelElement and ComponentTracker.

Overall, the DV8 analysis has provided vital information on unstable fundamental classes, which may be the reason for other inherently propagated issues. It is the intention to use the above profiling analysis results as a 'snapshot' of the current state of MERLIN and redoing the analysis ever few months to track and quantify the benefits of any carried out refactoring/restructuring.

Table 11: Summary of the DV8 architectural debt analysis.

Debt Metric	No. Issues	No. Files	LSLOC
Clique	2	38	7,335
Improper Inheritance	12	43	6,556
Package Cycle	19	173	35,926
Crossing	70	285	37,818
Unstable Interface	4	328	26,254

Table 12: List of improper inheritance usage identified by the DV8 analysis.

Improper Inheritance Files	No. Involved Files
PSvector.h	13
PSmoment.h	5
TCovMtrx.h	5
ReferenceParticle.h	5
Material.h	4
Transformable.h	4

Table 13: List of largest crossings identified by the DV8 analysis.

Crossing File	No. Involved Files
AcceleratorComponent.cpp	55
ComponentTracker.h	51
AcceleratorComponent.h	51
ParticleBunch.h	46
AcceleratorModel.h	40
Transform3D.h	36

Table 14: List of most unstable interfaces identified by the DV8 analysis.

Unstable Interface Files	No. File Changes
PSvector.h	327
AcceleratorComponent.cpp	319
ModelElement.h	319
merlin_config.h	27

5 Discussion & Conclusions

MERLIN was profiled with the intention of identifying any performance and/or sustainability issues. In-depth benchmarking, static profiling, low-level and high-level dynamic profiling was carried out using various profiling tools. Moreover, an architectural analyses was carried out using Drexel University's DV8 analysis framework.

In benchmarking, the number of particles per process was noted to be subject to CPU cache limitations. This led to an improvement in the methods with which particle bunch vectors are copied when updated. The -O2 flag was found to be the fastest default optimization flag.

Static analysis identified several code design issues, including excessive logical lines of code and high code complexity levels. Notably, MADInterface::ReadComponent, CCFailureProcess::DoProcess and ApertureConfiguration::ConfigureElementApetures were among the least-conforming in both metrics. Refactoring these member functions is a priority.

High-level dynamic analysis identified that RTMap::Apply and RMap::Apply account for 46% of the total execution time. Furthermore, RectEllipseAperture::PointInside and InterpolatedAperture::PointInside were also shown to accumulate a significant portion, 6.8%, of the overall execution time. Any improvement in either of the utilized algorithms could be greatly beneficial. Member functions PomeronScatter and DoCollimation have disproportionately low call counts but remain high in total percentage of execution time. High-level analysis also identified a number of memory leaks. Notably, 4 of the 6 top leaks where due to ApertureConfiguration::ConfigureElementApertures. DoCollimation was flagged here again.

Low-level dynamic analysis identified a number of members functions where superscaler operations and pipelining could be better utilised, notably several PointInside functions were flagged. In term of speculation, low-level analysis identified major sources of cache misses and branch mispredictions, with DoCollimation, Apply and ConfigureElementApertures all being flagged the worse.

Architectural analysis was carried out and identified instability issues with several fundamental classes, including PSvector and AcceleratorComponent. In particular, the analysis showed the true extent of the dependency issues within MERLIN, which not only hinder maintainability and modularity, but also propagate error-prone and change-prone issues. It is likely that a number of the prior discussed issues are as a results of propagated instabilities.

In conclusion, the MERLIN code base has been successfully and thoroughly profiled for performance and sustainability issues. The vast majority of classes are notably well within profiling metric limits, however, a select few classes and member functions are in need of reform, regularly being flagged in the top few for various different metric analyses. Furthermore, a few fundamental classes were flagged as problematic at an architectural level. As a result, it is the intention of the developers to focuses on stabilising the fundamental classes as well as working towards refactoring/redesigning classes and member functions which were flagged in multiple metrics tests. Profiling will also be redone on a

regularly basis (every few months) to track and quantify the performance and sustainability improvements as a result of the proposed code optimizations.

References

[1] MERLIN-Collaboration, *https://github.com/MERLIN-Collaboration*, 2017.

[2] J. Molson *et al.*, *Advances with Merlin - A Beam Tracking Code*, Proceedings of IPAC'10, Kyoto, Japan, 2010.

[3] M. Serluca *et al.*, *Hi-Lumi LHC Collimation Studies with Merlin Code*, Proceedings of IPAC'14, Dresden, Germany, 2014.

[4] H. Rafique, *MERLIN for LHC Collimation*, Proceedings of IPAC'15, Shanghai, China, 2015.

[5] S. Tygier, *Recent Developments and Results with the MELRIN Tracking Code*, Proceedings of IPAC'17, Copenhagen, Denmark, 2017.

[6] H. Rafique, *MERLIN for High Luminosity Large Hadron Collider Collimation*, Ph.D. Thesis, The University of Huddersfield, 2016.

[7] K. Cooper and L. Torczon, *Engineering a Compiler*, Morgan Kaufmann, ISBN: 978-1558606982, 2003.

[8] N. G. Dickinson *et al.*, *Importance of explicit vectorization for CPU and GPU software performance*, Journal of Computational Physics, vol. 230, pp. 5383-5389, 2011.

[9] Ueli Kunz and Julius Weder, *metriculator CDT metric plug-in*, Thesis, University of Applied Sciences Rapperswil, 2011.

[10] T. J. McCabe, *A Complex Measure*, IEEE Trans. Soft. Eng., vol. SE-2, 1976.

[11] Valgrind Developers, *http://valgrind.org/*, 2017.

[12] M. Booshehri *et al.*, *An Improving Method for Loop Unrolling*, Int. Jour. Comp. Sci. Inf. Sec., vol. 11, 2013.

[13] L. Xiao, *Titan: A Toolset That Connects Software Architecture with Quality Analysis*, Proceedings of the 22nd ACM SIGSOFT FSE2014, pp. 763-766, 2014.

[14] ArchDia, *https://www.archdia.com/*, 2017.

[15] HTCondor Developers, *https://research.cs.wisc.edu/htcondor/*, 2017.

Appendix A

CPU Cache Limitation/Unnecessary Bunch Copies

Merlin stores the particle bunch in a C++ std::vector. This is a very efficient data structure as it is continuous in memory and the location of any particle can be calculated in O(1) time. However, during aperture checks, any particle can be lost from the bunch and a vector does not support efficient erasure of the elements. Each particle that is erased from the middle of the bunch would cause all the particles following it to be moved along by 1 position to preserve the continuous layout. In order to avoid having to make multiple moves at each aperture, MERLIN avoids using the erase method. Instead, it creates a new bunch and copies the surviving particles to the new bunch. This means that particle information is only moved in memory once at any given aperture. However, this has a side effect of increasing cache use, as two copies of bunch will exist simultaneously. It is also speculate that such movement is detrimental to the CPUs caching algorithm as the location of the bunch changes.

To minimise these effects, a pre-check was added to make sure that the bunch is only copied when needed. If no particles are lost at a given aperture, there is no need to copy the bunch. Figure 4 shows the performance comparison for simulations with and without unnecessary bunch copying. There is a clear and significant performance increase in implementing the pre-check.

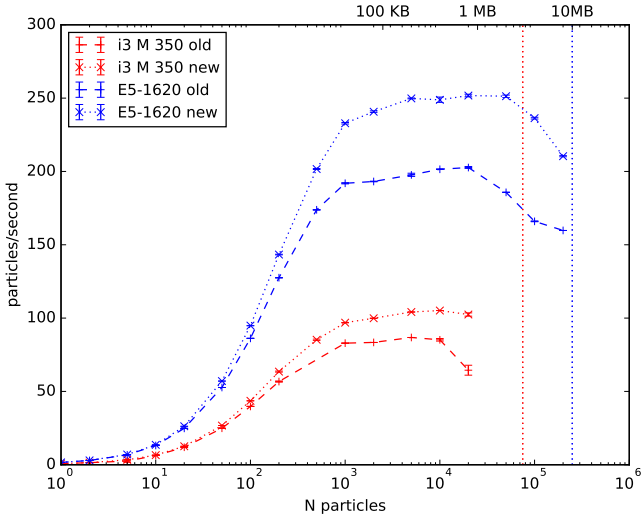

Figure 4: Performance comparison across two CPU types, showing particles/second curves, both with and without preventing unnecessary bunch copies.

Appendix B

Optimization of the PointInside Algorithm

The current method of checking whether or not a particle is within the aperture boundaries is shown in Figure 5 (Note that a RectElliptical aperture is modelled in MERLIN as the superposition of a rectangle and an ellipse).

```
bool RectEllipseAperture::PointInside (double x, double y, double z) const
{
    if( (x*x + y*y*HV) > EHH2)
    {
        return false;
    }
    else if(std::fabs(x) > RectHalfWidth || std::fabs(y) > RectHalfHeight)
    {
        return false;
    }
    else
    {
        return true;
    }
}
```

Figure 5: MERLIN RectEllipseAperture::PointInside code.

The function first checks if the particle lies within the ellipse, then subsequently if it lies within the rectangle. A faster method is to use a more in-depth algorithm, utilizing other simple geometries, such as 45-degree-rotated squares (diamonds). Such an algorithm should also be ordered taking into account the most common particle scenario. The devised algorithm is graphically depicted and shown in Figure 6.

A series on additional steps are added, each checking the particle lies within ever growing diamonds. This allows for most checks to be passed by calculations involving additions rather than more computationally taxing multiplications. Several check algorithms were benchmarked against the current one. The average of 10 loops of 1 billion function calls is taken for each. Results are shown in Table 15.

Table 15: Benchmarking results of various check algorithms.

Check Type	Time Avg [s (stdv)]	Difference [%]
Current	36.65 (0.49)	n/a
Current (swapped)	37.2 (0.95)	+1.49
Swapped + 1 diamond	34.52 (0.66)	-5.80
Swapped + 2 diamonds	34.58 (0.74)	-5.65

It is clear that the addition of a diamond check results in a significant 4.09% performance improvement. A further diamond step, however, in fact decreases performance, though only by 0.15% and is within error margins. It is therefore concluded that a single diamond step is optimal and that any further change due to additional steps would either be negligible or detrimental. It is also noted

```
91  bool checkwithin::newcheck2diamond(double x, double y){
92      double ax = fabs(x);
93      double ay = fabs(y);
94
95      if (ax+ay<minDim) return true;
96      if (ay>=rectH) return false;
97      if (ax+ay<maxEllip) return true;
98      if (ax>=ellipH) return false;
99      if ((x*x + y*y*ellipH2overV2) < ellipH2) return true;
100     return false;
101 }
102
```

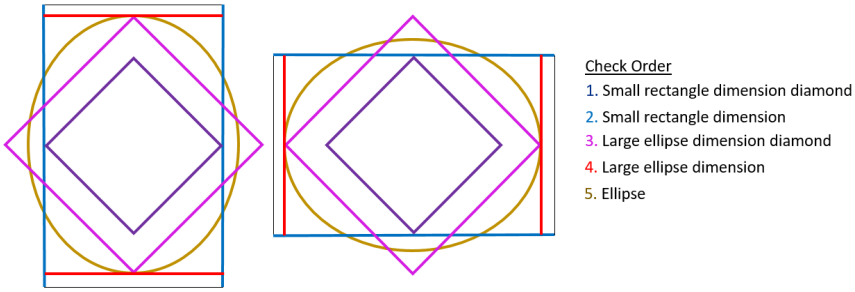

Figure 6: Revised MERLIN RectEllipseAperture::PointInside code (top) and corresponding graphical depiction (bottom).

that, although multiplications are more computationally taxing, simply swapping the check order to exclude particles outside the rectangle without additional diamond steps decreases performance by 1.49%. This is a prime example of the importance of taking into account probability, i.e. where the particles are most likely to be.

Callgrind results following implementation of the above changes are shown in Table 16. There is a clear reduction in the cumulative percentage of execution for all PointInside functions. Moreover, a significant reduction in overall execution time of ∼2% is noted. As a result, it is the intention that similar algorithmic optimization process be done for all aperture geometries in MERLIN.

Table 16: Callgrind function calls results post-optimization.

Function Name	Function Calls	Call Time [%]
RTMap::Apply	1,378,425,116	33.40
RMap::Apply	1,378,425,116	15.86
InterpolatedRectEllipseAperture::PointInside	167,051,974	3.64
ppDiffractiveScatter::PomeronScatter	50,000,469	3.08
RectEllipseAperture::PointInside	1,106,895,569	2.89