Code Transformation Impact on Compiler-based Optimization: A Case Study in the CMSSW

Teerit Ploensin 1, Krerk Piromsopa1, Norraphat Srimanobhas2

1 Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Thailand
2 Department of Physics, Faculty of Science, Chulalongkorn University, Thailand
E-mail: teeritploensin@gmail.com; Krerk.P@chula.ac.th; Norraphat.S@chula.ac.th

Abstract. In this paper, we study the benefit of applying loop transformations to a part of module in the CMS software. Particularly, we focus at the effect of loop transformations in term of performance improvement from the optimization process of compilers. Loop optimizations have been considered at low-level phase, such as loop unrolling using compiler directive. For high-level code transformations such as index set splitting and loop reordering, we adopt the polyhedral model to simplify the transformations. In this study, our loop optimization has been evaluated quantitatively. We study the impact on loops execution speed up and its instruction executed. Our observation shows that high-level loop optimizations can reduce both execution time and the number of instruction. This behavior suggested that simple loop transformations can trigger other optimizations. Moreover, we not only improve the overall performance, but also reduce the number of instruction. The results show that loop optimizations yield the speed up between 1.5 and 1.7.

1. Introduction
For scientific application, computational cost and time are mostly dedicated to program loops. Scientific simulation code and their mathematical models are steadily evolved with respect to new insights and requirements. Consequently, loop optimization is significantly important for improving execution time and reducing loops overhead. Thus, many techniques are specifically used to provide high-performance code.

Loop optimization [1, 2] can be described as a sequence of specific loop transformations that perform to the source code or intermediate representation of their languages. Normally, a transformation must preserve all dependencies. Compilers have a wide range of loop optimizations, which are different. Directives for loop optimizations, such as vectorizing, are implemented in most mainstream compilers. Compilers will either perform the transformation automatically, or upon requested via the command-line flag. Nevertheless, compilers may achieve significantly lower performance than those of manually optimized programs [3]. A better approach is to search for restructuring techniques that enable other optimizations while preserving semantics of the original source code. This paper investigates the use of these techniques applying to Compact Muon Solenoid Software (CMSSW) by making manual changes, examining whether or not the optimizations can be done by the compilers and comparing runtime performance among them.
Tuura, et.al. [4] suggested that the CMS software, physics simulation software written in C++ language, requires optimization to fit the resource budget, which allows a single event to process faster. Moreover, its current version is targeted by many loop optimization patterns, discussed in this paper. Therefore, we aim to experiment on how loop transformation may improve performance.

This paper is divided into sections as follows: Section 2 provides the CMS software overview. Section 3 discusses patterns of interests to apply transformation for triggering loop optimizations. Section 4 evaluates the impact of applying loop optimizations in the CMS software on its performance. Then, we conclude our analysis in section 5.

2. The CMS Software Overview

The CMS Software (CMSSW) [5] is an overall collection of software using for simulation, reconstruction and data analysis of the CMS collaboration. Its workflow is basically defined configuration, written in Python language, to the CMSSW to perform different types of analysis with data collected from particle experiments.

In this paper, we interest in the CMSSW data format of reconstructed tracks, which is a set of configuration code that is responsible for output tasks. We aim to make deeper optimization by exploring loops in the modules. Our study finds that loops do not get much benefit from current optimization. Therefore, we apply appropriate code transformations, which eventually allow compiler to perform better optimizations.

3. Patterns to Transform

Usually, compilers are able to generate the optimized versions of loops. However, they sometimes require restructuring to help with the optimization. The first optimization pattern is based on the polyhedral model, which is a mathematical model attended to analyze and to transform nested loops. Such technique is independent of compilers or target hardware. We adopt the polyhedral model to investigate on how nested loops should be transformed. The second optimization pattern is loops with branches. It is normally noticed when parts of code doing something repeatedly with conditional statements. As a result, this section focuses on patterns that can benefit from rewriting, restructuring or using compiler options.

3.1. Polyhedral model

Polyhedral model [6, 7] is an algebraic representation about nested loops which combine transformation flexibility to design optimizations. A nested loop with affine bounds and conditional expressions can be represented as a polyhedral representation. Such transformations are operated on the geometric representation of loop iteration called iteration space. Figure 1 represents an example of polyhedral representation for iteration space of loop nests. The compositions of a transformation, like the transformation legality, are easily checked. The polyhedral method treats each iteration as a point within mathematical objects called Polyhedra; and affine or non-affine transformations can be applied to transform depending on the optimization goal.

From Figure 1, iteration space can be written as:

\[ I_s = \{ S(i, j) \mid 0 \leq i \leq n, 0 \leq j \leq i \}. \] (1)

The use of polyhedral representation in optimizing typically focuses on loop transformation. The polyhedral model relies on three steps:

- representing the loop in the geometrical view called iteration space;
- performing a geometrical transformation in that space;
Figure 1. Nested loops that fit the polyhedral model, which can be represented as mathematical (constraint-based) objects (geometrically visualized) [8].

- converting the set of iteration domain of that iteration space back to generate optimized code.

3.2. Loops with Branches
When appearing in a loop, if and if-else statement might compromise performance. Because the if-else statement is usually translated into extra instructions like jump/branch instruction that causes a control hazard in pipelining [9]. Listing 1 shows an example of loops coming with branch statement. Thus, eliminating the condition removes control dependency and gives compiler the ability to pipeline more arithmetic operations.

```
void extraInstruction() {
    for (int i = 0; i < 100; i++) {
        if (i > 0) {
            continue;
        }
    }
}
```

Listing 1. Example of a loop with branch statement in high-level source code.

```
extraInstruction:
push rbp
mov rbp, rsp
mov DWORD PTR [rbp -4] , 0
.L4:
cmp DWORD PTR [rbp -4] , 99
je .L5
cmp DWORD PTR [rbp -4] , 0
add DWORD PTR [rbp -4] , 1
jmp .L4
.L5:
nop
pop rbp
ret
```

Listing 2. Example of a loop with branch statement in x86 assembly.

4. Program Transformation
This section mainly discusses on loop optimizations that enable other optimization to the compiled source code. We provide a briefly information on how the optimization works for each loop optimization.

4.1. Loop Unrolling
Loop Unrolling [1] is a loop optimization technique that basically increases the program’s speed by eliminating loop control instructions such as the end of loop tests on each iteration. This technique is also known as low-level loop optimizations at code generation phase. With this technique, the loop body is replicated with the exact number of times as well as the loop increments [2]. This amount of number is called the unrolling factor. Listing 3 shows an example of loop unrolling by factor 2. Nevertheless, this technique is not applicable to legacy code.
4.2. Index Set Splitting

Index Set Splitting [10] is a technique to decompose iteration domain by partitioning the index sets of all statements into a fixed number of parts, and computing the schedule for each part. With this technique, a loop is divided into multiple fractions, each responsible for its fraction of the original loop’s iteration domain. This method is done to remove dependencies. Thus, the resulting transformation provides more parallelism. Listing 4 shows an example of this optimization.

Listing 4. Example of Index Set Splitting.

4.3. Loop Interchange

Loop Interchange [1] also known as loop reordering, is a transformation that is normally used with a nested loop by exchanging iteration variables between an inner and an outer loop. Normally, a loop interchange is used to enhance the performance of program to gain more parallelism or enable vectorization. Listing 5 illustrates an example of loop interchange. Nevertheless, loop interchange is occasionally cumbersome, because data dependencies need to be considered before applying [11].

Listing 5. Example of Loop Interchange.
5. Evaluation

Usually, the CMSSW is compiled using GNU GCC compiler by default with optimization option (e.g., -O2). The -O level option tells GCC to turn on compiler optimization, when the specified value of level is in effect. With gcc -O2, compiler will explicitly invoke level two optimization, which improves the performance of the output binary, while avoiding numerical accuracy issues. From our evaluation, we are considering data format of reconstructed tracks which consists of 35 target loops in 3 previously discussed patterns. Table 1 represents an overview of the loop optimizations and the number of test programs that target these optimizations.

| Loop optimization   | Number of an optimized pattern |
|---------------------|--------------------------------|
| Loop unrolling      | 25                             |
| Index set splitting | 9                              |
| Loop reordering     | 1                              |

5.1. Benchmark Results

Firstly, we analyze our performance by using Callgrind, a profiling tool, to collect the number of instructions executed. Table 2 represents results for a selection of functions contained loops. The results are divided into two parts: initialization and loop optimizations.

| Function name                     | Initialization(-O2) | Optimized loops   |
|-----------------------------------|---------------------|-------------------|
| fillCovariance                    | 49,617,744          | 13,250,193        |
| TrackExtra                        | 10,953,225          | 6,761,250         |
| TrackBase                         | 9,801,004           | 5,183,644         |
| VertexCompositePtrCandidate       | 11,995,424          | 2,249,142         |

Manually applying loop optimizations to high-level source code is significantly smaller as shown in table 2. This is because a loop optimization might enable other loop optimizations, in our case are loop unrolling. Loop unrolling improves performance in two ways. First, the unrolled loops use fewer instructions, because they reduce the number jump instruction. Thus, there are fewer loop tests and branches. Second, the unrolled loops also enable other optimization like exposing instruction-level parallelism. Nevertheless, GCC needs to confirm that there is no loop-carried dependencies, which would prevent consecutive iterations. Thus, loop in a program needs to be restructured to improve loop performance.

Next, we evaluate the performance improvement in terms of timing speed up. Table 3 shows that the execution time per event processed has been improved both high-level loop optimizations, which are index set splitting and loop reordering, and low-level loop optimization, which is being done at code generation phase.

6. Conclusion

In this paper, we study the concepts of existing loop optimizations techniques. We present a novel code transformations, which resolve into better loop optimizations of large complicated software like the CMSSW. Our experimenting yield the following results. Using this technique, we improve the performance of each function containing loops in the modules of the CMSSW.
Table 3. Average execution time of repeatedly selected loop patterns in nanoseconds of CMSSW_10_2_3 compiled with GCC 7.3.1.

| Optimization description       | Initialization(-O2) | Optimized loops | Speed up |
|--------------------------------|---------------------|-----------------|----------|
| Index set splitting            | 125.34              | 79.67           | 1.57     |
| Unrolled loop with directive   | 122                 | 76.34           | 1.59     |
| Loop reordering                | 112                 | 64.67           | 1.73     |

We apply the polyhedral model to sequence of loops for simplification. This allows compiler to perform better optimizations. We also investigate the optimization option of GCC compiler. An important problem is that using -O2 alone does not normally give the expected performance without invoking other optimization techniques. The reason is that compiler cannot simplify the complexity of loops. Thus, it does not gain any benefit from this option. Our measurement is based on speed up and the number of instruction executed.

7. Acknowledgements
This work has been supported by C2F and CUniverse, Chulalongkorn University.

References
[1] Allen R, Kennedy K. Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann Publishers; 2001. Available from: https://books.google.co.th/books?id=X1QfogEACAAJ.
[2] Bacon DF, Graham SL, Sharp OJ. Compiler Transformations for High-Performance Computing. ACM Comput Surv. 1994 Dec;26(4):345–420. Available from: https://doi.org/10.1145/197405.197406.
[3] Pouchet LN, Bondhugula U, Bastoul C, Cohen A, Ramanujam J, Sadayappan P, et al. Loop Transformations: Convexity, Pruning and Optimization. ACM SIGPLAN Notices. 2011 05;46:549–562.
[4] Tuura LA, Innocente V, Eulisse G. Analysing CMS software performance using IgProf, OProfile and callgrind. J Phys Conf Ser. 2008;119:042030.
[5] The CMSSW Documentation Suite, The CMS Offline Workbook; [accessed June 23, 2020]. [Online]. Available from: https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBook.
[6] Griebl M, Lengauer C, Wetzel S. Code Generation in the Polytope Model. In: In IEEE PACT. IEEE Computer Society Press; 1998. p. 106–111.
[7] Benabderrahmane MW, Pouchet LN, Cohen A, Bastoul C. The Polyhedral Model Is More Widely Applicable Than You Think. In: Gupta R, editor. CC. vol. 6011 of Lecture Notes in Computer Science. Springer; 2010. p. 283–303. Available from: http://dblp.uni-trier.de/db/conf/cc/cc2010.htmlBenabderrahmanePCB10.
[8] Tobias Grosser, Johannes Doerfert. Analyzing and Optimizing your Loops with Polly. [Accessed July 22, 2020]. [Online]. Available from: http://www.llvm.org/devmtg/2016-03/Tutorials/applied-polyhedral-compilation.pdf.
[9] Muchnick SS. Advanced Compiler Design and Implementation. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 1998.
[10] Griebl M, Feautrier P, Lengauer C. On index set splitting. In: 1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425); 1999. p. 274–282.
[11] Allen JR, Kennedy K. Automatic Loop Interchange. SIGPLAN Not. 1984 Jun;19(6):233–246. Available from: https://doi.org/10.1145/502949.502897.