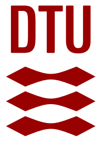
Downloaded from orbit.dtu.dk on: Apr 19, 2024



DTU Library

Code Commentary and Automatic Refactorings using Feedback from Multiple Compilers

Jensen, Nicklas Bo; Probst, Christian W.; Karlsson, Sven

Published in:
Proceedings of the 7th Swedish Workshop on Multicore Computing (MCC'14)

Publication date:
2014

Link back to DTU Orbit

Citation (APA):
Jensen, N. B., Probst, C. W., & Karlsson, S. (2014). Code Commentary and Automatic Refactorings using Feedback from Multiple Compilers. In *Proceedings of the 7th Swedish Workshop on Multicore Computing (MCC'14)*

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Code Commentary and Automatic Refactorings using Feedback from Multiple Compilers

Nicklas Bo Jensen
Technical University of Denmark
DTU Compute
nboa@dtu.dk

Christian W. Probst
Technical University of Denmark
DTU Compute
cwpr@dtu.dk

Sven Karlsson
Technical University of Denmark
DTU Compute
svea@dtu.dk

ABSTRACT
Optimizing compilers are essential to the performance of parallel programs on multi-core systems. It is attractive to expose parallelism to the compiler letting it do the heavy lifting. Unfortunately, it is hard to write code that compilers are able to optimize aggressively and therefore tools exist that can guide programmers with refactorings allowing the compilers to optimize more aggressively. We target the problem with many false positives that these tools often generate, where the amount of feedback can be overwhelming for the programmer. Our approach is to use a filtering scheme based on feedback from multiple compilers and show how we are able to filter out 87.6% of the comments by only showing the most promising comments.

1. INTRODUCTION
Writing programs that performs well on modern multi-core systems is a major challenge. Many aspects influence the performance, especially how well the optimizing compiler has transformed the code into a faster version. As parallel programming is hard it is attractive to expose the parallelism to the compiler. With auto-parallelization and auto-vectorization it can do the heavy lifting. Sadly, it is hard to write code that compilers can optimize well given the large complexity of compilers.

We have developed a tool that can assist programmers in understanding how compilers optimize and even give advice on source code changes that could allow for more aggressive optimization. Many such tools exist, but we believe that one key difference will make our tool more usable, namely the amount of false positives and true positives. Programmers will only use a tool if it is cost effective and a good use of their time. If only one out of many comments can successfully be applied it is not effective. In contrast to other similar tools, we use input from multiple compilers, allowing us to produce fewer false positives while still producing many true positives. We believe this is key for a wider adoption of optimization advice tools.

Our tool works by parsing the optimization reports from multiple compilers and use insight that if one compiler succeeded in optimizing, the others could potentially as well. In this way, if one compiler succeeds in optimizing, even just partially, we might be able to modify the original source code such that more optimizations can be applied. We are able to filter the number of comments generated by three compilers with 87.6%, resulting in an amount of comments that is easier to handle and focus on for the programmer.

Our current implementation supports the optimization reports from three major production compilers ICC [6], GCC [4] and Clang [13]. Each of these has optimization reports, which describe the applied optimizations and the missed optimization. We analyse these reports in our plugin, built into the Eclipse Integrated Development Environment. Combined with our own analysis we can suggest automatic source code refactorings to the programmer. We also visualize how the different compilers have optimized the code by coloring the source code, giving a very quick overview of where the programmer's time is best spent.

To summarize, we make the following contributions:

- We propose a novel compiler driven feedback model based on input from multiple compilers
- A preliminary implementation supporting optimization reports from ICC, GCC and Clang
- Show how we are able to filter 87.6% percent of the compiler generated comments

The paper is laid out as follows. We discuss related work in the next section 2. As motivation related compiler optimization studies are discussed in section 3. The tool is described in section 4. Experimental results are analyzed in section 5. Last section 6 concludes the paper.

2. STATE OF THE ART IN CODE COMMENTARY AND OPTIMIZATION ADVISERS
Numerous systems exist describing how the source code have been optimized by compilers and give advice on source code changes that could potentially allow for more aggressive optimization using compiler driven feedback.

The Oracle Solaris Studio [12] has a feature called Compiler Commentary. It textually annotates the source code with details on which optimizations were applied.

Source code is often written in a way that prohibits some optimizations like automatic vectorization or parallelization. However, small changes can make the code more amenable to aggressive compiler optimization, often without affecting software engineering principles.

The Intel Performance Guide included in the Intel Composer XE [6] can give advice for source code modifications allowing the compiler to optimize more aggressively. It can guide the programmer, by first profiling for hotspots, suggest optimization flags and suggest small source code changes. It is a user-friendly tool, however it only gives suggestions when it is confident, thus often not showing any advice. Similarly IBM's XL C/C++ and Fortran compilers can generate XML reports describing applied optimization, and suggest modifications to enable more aggressive optimization [3].

Larsen et al. also describes their tool, consisting of a modified version of GCC, outputting information on why a given optimization was not applied and displays this information in the Eclipse IDE [8, 10, 11]. In this way, their tool reuse existing aggressive compiler optimizations for feedback and help the programmer understand why a given optimization was not applied. In a parallelization study, cases with superlinear speedups of parallel code parts were reported due to positive side effects of modifications [11].

Jensen et al. has built static analysis into an Eclipse plugin, which can suggest automatic source code refactorings without depending on any compilers [7]. This allows for quicker turn-around times and more advanced refactorings.

Last Aguston et al. shows their tool, which uses code skeletons for parallelization hints [1]. Using skeletons allows Aguston et al. to make larger code transformation and leave the question of whether doing so is safe to the programmer. Applying their skeletons on a subset of SPEC benchmarks suggest a speedup of 30%, however whether doing so is safe is not clarified.

The main difference between these tools and this paper is the use of feedback from multiple compilers allowing us to have fewer false negatives while still giving speculative advice.

3. COMPILER OPTIMIZATION STUDIES

This paper's motivation is to show how we can reduce the number of false positives using feedback from multiple compilers. Multiple studies have shown how different compilers optimize different loops, which is the fundamental driver behind our filtering approach.

Callahan et al. studied 100 loops written in Fortran with the purpose of testing the vectorization effectiveness of 19 compilers [2]. On average the compilers vectorized or partly vectorized 61% of the loops, the best compiler vectorized 80% of the loops.

A more recent study presented by Maleki et al. looked at three newer compilers, ICC from Intel, XLC from IBM and GCC. Maleki et al. complement the loops used by Callahan et al. with additional loops. The original Fortran loops were rewritten into C and modified such that the compilers had as much information as possible for a total of 151 loops. They found that by applying changes to the source code level or using vector intrinsics XLC could vectorize 82% and ICC 84% of the loops such that the loops performed better than manual vectorization. GCC failed to vectorize many of these loops, for example GCC could only vectorize 60% of the loops that XLC and ICC combined could vectorize.

Table 1: Supported compiler versions and compiler flags

Compiler	Version	Flags
ICC	Intel Composer XE 2015 [6]	-opt-report
GCC	GCC 4.9.1 [4]	-fopt-info-optall
Clang	Clang 3.5.0 [13]	-Rpass=.*

Similar to the two previous studies Larsen [9] presented how four compilers, ICC from Intel, XLC from IBM, PGCC from Portland Group and SUNCC from Oracle, optimized the loops in the EEMBC benchmark suite. The motivation was to show how synergies between compilers can be used for categorizing missed optimization as resolvable. Out of the 3490 missed optimizations generated by the four compilers, 43% could be categorized as potentially resolvable or unprofitable. These are all very promising results. We see how even if a loop is not optimized by one compiler, it will often be optimized by another due to the strength and weaknesses of the individual compiler optimizations. We can use this information to guide the programmer focus on the loops that we know are possible to optimize more aggressively.

4. MULTI-COMPILER FEEDBACK TOOL

Our tool is based on the optimization reports from production quality compilers. These can report applied optimizations and missed optimization.

We currently support input from three compilers, namely ICC from Intel and the two production quality open source compilers GCC and Clang as seen in table 1. The versions are the newest at the time of writing. The optimization report feature is new in GCC and Clang and thus has limited support for the number of optimization passes it can produce feedback from. Therefore, we focus on automatic vectorization of loops, an optimization that is very important for good performance on modern processors.

There are many limitations to automatic vectorization as it involves numerous advanced analysis steps. Every compiler performs roughly the same steps, however as the implementations vary they each has strengths and weaknesses. Some of the types of analysis that needs to succeed are: identification of loop bounds and stride, induction variable analysis to determine dependencies between loop iterations and alias analysis to again to determine dependencies within and between loop iterations. These three analyses are used as input into the actual data dependency analysis. For each of these analyses there exists many weaknesses, some of which can be addressed at the source code level. We propose automatic refactorings to the programmer to mitigate these limitations. One example is how we can help alias analysis by specifying that two memory locations are distinct using the C99 restrict keyword. We reuse many of the automatic refactorings shown to be effective in earlier research [7, 11]. The tool process is twofold as seen in figure 1:

1. First the programmer has to manually change the build system, such that the program is compiled with multiple programs and with the additional compiler flag for producing optimization reports. We are working on automating this step. The flags used for all later

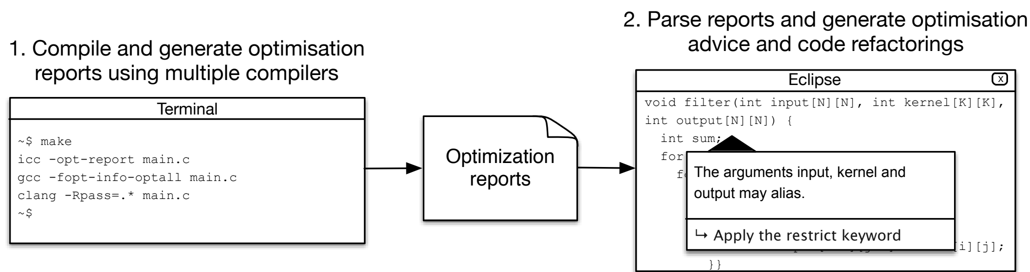


Figure 1: Overview of tool process.

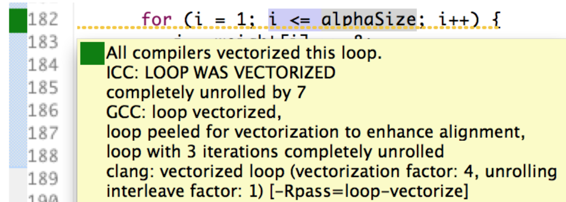


Figure 3: Screenshot of the visualization of how the different compilers vectorize. By hovering over the source line the compiler feedback will be displayed.

examples are shown in table 1.

2. Second we parse the optimization reports in our Eclipse plugin, aggregate all the comments based on loops, filter them based on how each compiler optimized and display the overview and proposed automatic refactorings to the programmer.

The plugin is based on Eclipse Kepler 4.3. The first step is to parse the optimization reports generated by the compilers. Correlating the comments presents a challenge in itself, for multiple reasons. Different coding styles must be handled as seen in figure 2. Comments from different compilers may refer to same loop, but different source code lines. We handle this by relating each comment to a loop instead of a source line. This is based on a simple algorithm that first finds loops and loop nests, and their corresponding source code line ranges. This implementation assumes that we only have one loop per source code line. One last issue handled is how comments for inlined function calls are handled. Depending on the compiler, these may be described as corresponding to the call site or the function itself.

After aggregating comments, we classify how each compiler has optimized into three categories: not vectorized, partially vectorized or fully vectorized. We present this classification directly in the IDE to the programmer using colored source code lines. We color the Eclipse marker bar either green, orange or red depending on how many compilers optimized. In this way we do not overwhelm the programmer with too much information and if more information is desired, hovering over a marker bar will present the classification and the individual compilers comments as seen in figure 3.

5. RESULTS

We have studied the C benchmarks from the SPEC2006 benchmark suite [5] in total 11 benchmarks: 401.bzip2, 403.gcc,

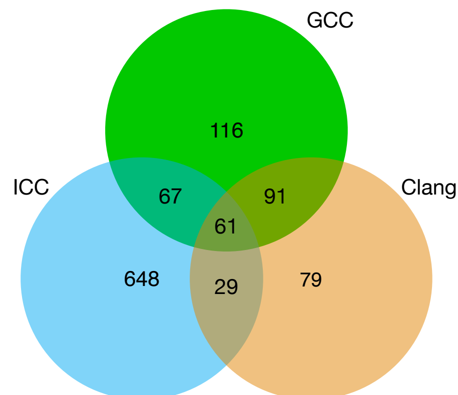


Figure 4: Venn diagram of how many loops the three compilers, ICC, GCC and Clang have vectorized among the loops in the 11 C benchmarks from SPEC2006.

429.mcf, 433.milc, 445.gobmk, 456.hmmer, 458.sjeng, 462.libquantum, 464.h264ref, 470.lbm and 482.sphinx3. These benchmarks consist of 494.709 lines of C code and contain 30.370 loops that can potentially be optimized. We use the compilers ICC, GCC and Clang with the versions seen in table 1. Each compiler target the Intel Haswell platform. Out of the total 30.370 loops in the benchmarks, only 8829 loops produce any comments by some compiler. This is mainly due to the large amount of optimization done in the compilers, e.g. full loop unrolling will eliminate a loop.

The number of loops each compiler vectorized is shown in figure 4. We see how ICC from Intel is clearly dominating with 805 vectorized loops, where GCC and Clang vectorized 335 and 260 loops respectively. We also see how GCC and Clang optimize many loops that ICC does not, it may be a missed optimization in ICC or that it is actually not beneficial to do so.

We intend to use this data in a loop ranking mechanism that ranks all loops based on how likely it is a suggested refactoring is going to succeed and improve performance. We use profiling data to rank hot loops and answer how worthwhile a closer inspection is. We use compiler reports to rank loops on how likely it is a refactoring is going to succeed. This means we can focus the programmers attention on hot loops that are not optimized fully by his chosen compiler, but by another compiler. These loops are good candidates for making the suggested automatic refactorings cost effective.

```
1  for (int i=0; i < N; i++)
```

(a) No opening brackets

```
1  for (int i=0; i < N; i++)
2  {
```

(b) Brackets on the subsequent lines

```
LOOP BEGIN at file.c(1)
 remark #15300: LOOP WAS VECTORIZED
LOOP END
```

(c) IICC comment referring to line 1

```
file.c:3: note: loop vectorized
```

(d) GCC comment referring to line 3

Figure 2: Issues encountered when correlating compiler comments. In (a) and (b) different coding styles must be handled. In (c) and (d) comments from different compilers refers to the same loop, but different lines numbers in the source code.

In this way see how for example ICC could have vectorized 286 additional loops, GCC could have vectorized 756 additional loops and last Clang could have vectorized 831 additional loops. If the programmer only target one specific compiler, the amount of compiler feedback is possible to handle given that these are derived from 11 benchmarks. For all compilers the loops with feedback have reduced the number of loops with feedback from 8829 loops to 1091, an 87.6% reduction.

We have previously shown the benefit of automatic refactorings that allows the compilers to optimize more aggressively [7]. One result is the speedup achieved by adding the `restrict` keyword from C99 to an edge detection kernel from the UTDSP benchmark suite. This comment is not filtered out by our tool, and can help GCC with an automatic refactoring allowing it to vectorize one loop giving a 140% speedup [7]. Multiple related works presents other use cases with good speedups [8, 9, 11]. These automatic refactorings combined with our filtering is a promising avenue.

To give more precise data it would be very relevant to add more compilers and platforms. This could include XLC from IBM on the Power architecture. With input from more compilers it would be possible to give extra priority to loops that are optimized by multiple other compilers.

6. CONCLUSIONS

Many applications rely on optimizing compilers for performance. Unfortunately they are often not written in a way that allows compilers to optimize aggressively. Tools that help programmers write code in a way that the compilers can understand are important. However, tools that do this often have many false positives leading to programmers not using them, as they are simply not cost effective.

To this end we introduce how the feedback of multiple compilers can be used as a filtering mechanism, reducing the amount of false positives by only showing the most promising comments. Using a simple filtering we are able to achieve an 87.6% reduction in comments. This is a significant step in the direction of making compiler driven automatic refactorings cost effective.

7. ACKNOWLEDGMENTS

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement number 332913 for project COPCAMS.

8. REFERENCES

[1] C. Aguston, Y. Ben Asher, and G. Haber. Parallelization hints via code skeletonization. In *Symposium on Principles and Practice of Parallel Programming*, PPoPP, 2014.

[2] D. Callahan, J. Dongarra, and D. Levine. Vectorizing compilers: A test suite and results. In *Conference on Supercomputing*, Supercomputing, 1988.

[3] Y. Du, K. Vinayagamoorthy, K. Yuen, and Y. Zhang. Explore Optimization Opportunities with XML Transformation Reports in IBM XL C/C++ and XL Fortran for AIX Compilers. IBM developerWorks, 2015.

[4] Free Software Foundation. GNU Compiler Collection. `http://gnu.gcc.org`. Accessed on 24/9/2014.

[5] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News*, 2006.

[6] Intel. Intel Composer XE 2015. `http://software.intel.com/en-us/intel-composer-xe`. Accessed on 24/9/2014.

[7] N. B. Jensen, S. Karlsson, and C. W. Probst. Compiler feedback using continuous dynamic compilation during development. In *Workshop on Dynamic Compilation Everywhere*, DCE, 2014.

[8] N. B. Jensen, P. Larsen, R. Ladelsky, A. Zaks, and S. Karlsson. Guiding programmers to higher memory performance. In *Workshop on Programmability Issues for Heterogeneous*, MULTIPROG, 2012.

[9] P. Larsen. *Feedback Driven Annotation and Refactoring of Parallel Programs*. PhD Thesis. Technical University of Denmark, 2011.

[10] P. Larsen, R. Ladelsky, S. Karlsson, and A. Zaks. Compiler driven code comments and refactoring. In *Workshop on Programmability Issues for Heterogeneous Multicores*, MULTIPROG, 2011.

[11] P. Larsen, R. Ladelsky, J. Lidman, S. A. McKee, S. Karlsson, and A. Zaks. Parallelizing more loops with compiler guided refactoring. In *International Conference on Parallel Processing*, ICPP, 2012.

[12] Oracle. Oracle Solaris Studio. `http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html`. Accessed on 17/5/2013.

[13] The LLVM Foundation. clang: a C language family frontend for LLVM. `http://clang.llvm.org`. Accessed on 24/9/2014.