Compiler Feedback using Continuous Dynamic Compilation during Development

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

Published in:
Proceedings - Workshop on Dynamic Compilation Everywhere

Publication date:
2014

A

Compiler Feedback using Continuous Dynamic Compilation during Development

Nicklas Bo Jensen, Technical University of Denmark
Sven Karlsson, Technical University of Denmark
Christian W. Probst, Technical University of Denmark

Optimizing compilers are vital for performance. However, compilers ability to optimize aggressively is limited in some cases. To address this limitation, we have developed a compiler guiding the programmer in making small source code changes, potentially making the source code more amenable to optimization. This tool can help programmers understand what the optimizing compiler has done and suggest automatic source code changes in cases where the compiler refrains from optimizing. We have integrated our tool into an integrated development environment, interactively giving feedback as part of the programmers development flow.

We have evaluated our preliminary implementation and show it can guide to a 12% improvement in performance. Furthermore the tool can be used as an interactive optimization adviser improving the performance of the code generated by a production compiler. Here it can lead to a 153% improvement in performance, indicating the feasibility of the tool as a performance adviser for a production compiler.

General Terms: Compiler design; compiler driven feedback; automatic refactoring; automatic vectorization and parallelization

ACM Reference Format:
DCE 2014 V, N, Article A (January 2014), 12 pages.

1. INTRODUCTION

Arguably we all rely on optimizing compilers for performance. When they optimize well, they can often generate code which outperform hand optimized code. However, optimizing compilers are complex and hard for programmers to understand. Programmers are often not aware how their programs are optimized and how they should be written to allow the compiler to optimize well. We have created a compiler, that guides programmers in modifying their programs, potentially making them more amenable to optimization.

Optimizing compilers and integrated development environments both perform analysis of code, regrettably usually with no code sharing between these. Therefore we have embedded a small optimizing compiler into an IDE. The compiler is part of the programmers development flow continuously giving feedback to the programmer as it is integrated with the IDE.

We open up the black box the compiler is today, exposing valuable information to the programmer using compiler driven feedback. In contrast, other related tools can also report on optimization issues encountered during compilation. The programmer then has to understand why a given optimization was not performed. Our tool is integrated into the IDE, dynamically giving quick feedback to the programmer during development. The integration improves the automatic refactoring we provide.

The benefit of the feedback is twofold; the feedback lets programmers know how their code how been optimized, and a reason if the compiler refrains from optimizing. Compilers often refrains from optimizing, especially due to limitations of program analysis. Programmers can unintentionally prevent optimization by unfortunate coding choices. If the programmer is aware of the problems, they can be mitigated by modifying the source code, often without affecting other software engineering principles.

An early implementation of the introduced compiler have been evaluated on 12 kernel benchmarks and we show that the feedback can lead to speedups of up to 153%.

In short, we make the following contributions:

— We show that the traditional compiler structure can be redesigned by reusing existing IDE technology in the compiler.
— We show that we can provide better automatic refactoring by integrating the IDE and the compiler.
— We have evaluated the tool and show that compiler driven feedback can lead to performance speedups. For example the suggested automatic refactorings can improve performance with up to 153% when used with a production compiler.

The paper is laid out as follows. We will discuss related work next in Sect. 2. The tool is presented in Sect. 3 and 4. Experimental results are analyzed in Sect. 5. We discuss future work in Sect. 6 and conclude the paper in Sect. 7.

2. RELATED WORK

Compiler driven feedback is not a new idea and is used in related systems for improving interactive feedback in integrated development environments [Ng et al. 2012], commentary about applied compiler optimizations [Oracle 2013; Du et al. 2012] and optimization advice [Larsen et al. 2012; 2011c; Larsen et al. 2011b; Jensen et al. 2012; Intel 2013; Du et al. 2012].

IDE Support. IDEs help the programmer with interactive feedback on syntax errors, type errors and code completion. Furthermore, they often provided a rich set of automatic refactorings for otherwise tedious coding tasks. These features require an understanding of the program and many IDEs have implemented technology which is also found in compilers. There is an opportunity for reusing code from compilers in the IDE, providing IDE support. One example of this is the Microsoft Roslyn compiler [Ng et al. 2012]. It tackles the issue of serving intelligent integrated development environments with data for auto-completion information, refactorings and jumping around in source code such as finding definitions. This is all information where compilers could open up and through APIs to their internal data structures make these services available in IDEs.

Optimization Commentary. The Oracle Solaris Studio [Oracle 2013] has a unique feature called Compiler Commentary. It annotates the source code with details on which optimizations were applied. Some comments are easy to understand, such as duplicate expressions or a loop being fissioned. Other comments only makes sense to experienced developers or compiler engineers. Furthermore, the tool cannot give advice on source code changes enabling further optimization. Our tool reports applied optimizations directly in the IDE, making it more user friendly.

Optimization Advisers. Programmers often prevent optimization with their coding choices, but small source code modification which does often not affect software engineering principles can help the compiler optimize further. Based on this realization a series of tools have been implemented, guiding the programmer in how to modify programs allowing the compiler to optimize more aggressively.

The Intel Performance Guide included in the Intel Composer XE [Intel 2013] can guide the programmer in profiling for hotspots, suggest optimization flags and give advice for source code modifications enabling auto-parallelization. It is a very user friendly tool, however it only gives suggestions when it is confident, thus often not showing any advice. Complementary IBM's XL C/C++ and Fortran compilers can generate XML reports describing applied optimization, and in some cases suggest modifications to enable further optimization [Du et al. 2012]. Our tool provide feedback on a broader range of issues and present the feedback directly in the IDE.

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 an IDE [Larsen et al. 2012; 2011c; Larsen et al. 2011b]. 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 super-linear speedups of parallel code parts were reported due to positive side-effects of modifications [Larsen et al. 2012]. The tool have issues with mapping internal GCC data structures to the real source code which is not always possible, and they generate many false positive comments due to GCC, requiring extensive comment filtering which is still an open research problem [Larsen et al. 2011a].

3. COMPILER INFRASTRUCTURE

We have designed and implemented an optimizing compiler infrastructure to address the issue of giving precise compiler feedback on the original source code. The compiler is embedded inside the Eclipse IDE. The compiler reuses existing Eclipse technology in the front-end by using the infrastructure already implemented for code analysis and supporting the programmer during development. In this way, the implemented compiler integrates into the normal Eclipse development flow, continuously giving feedback to the programmer during development. Furthermore, internal Eclipse data structures constructed for existing source knowledge tools can be reused. This decreases the load of executing our compiler during development in contrast to running an entire production compiler.

The compiler is constructed as a series of passes where most consume a single intermediary representation. The architecture and flow through the compiler is presented in figure 1. The infrastructure is organized as a series of modules with clear interfaces between them. The C front-end is constructed using Eclipse language tooling. The front-end takes the Eclipse representation of the source code and generates the internal representation inside the compiler for optimization. The optimizer consumes the internal intermediary representation, IR, and produces IR. The optimizer interfaces with the feedback-visualization Eclipse plugin by generating data on applied optimization and when an optimization was not applied. Last the code generator consumes the IR and emits assembly code compatible with GCC [Foundation b] calling convention, which can be assembled and linker by GNU Binutils [Foundation a].

Only the C language is supported and the C front-end uses the Eclipse CDT C/C++ language tooling [CDT 2013]. The idea behind the compiler infrastructure is not restricted to C and there exists Eclipse language tooling for many other languages.

The intermediary representation used for feedback and optimizations is based on Static Single Assignment, SSA, form [Cytron et al. 1991]. It has the special property that variables are only assigned at most once. This form is argued to improve many types of optimization and is found in many optimizing compilers. The IR is on three address form and consists of blocks with sequential instructions and special blocks for control flow.

3.1. Optimizer

The compiler is intended to be executed during the development, constraining the type of optimizations that can be executed fast enough without disturbing the responsiveness of the IDE. However, arguably the more optimization passes we implement the better feedback we can provide. We have implemented a series of scalar optimizations on the SSA form: Copy propagation [Skeppstedt 2012], constant folding, sparse conditional constant propagation [Wegman and Zadeck 1991], arithmetic simplifications, partial redundancy elimination [Kennedy et al. 1999], global value numbering [Torczon and Cooper 2011], operator strength reduction [Skeppstedt 2012], loop

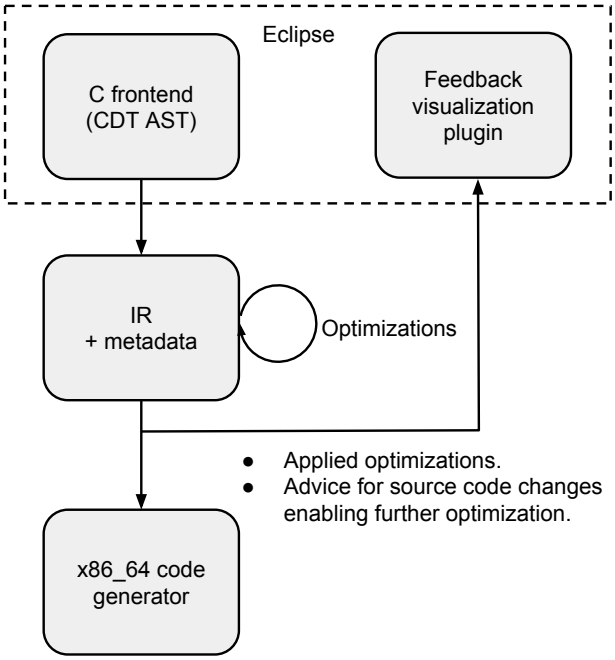


Fig. 1: Compilation framework architecture.

interchange and inlining. All optimization are performed in the order suggested by Muchnick [Muchnick 1997].

3.2. Backend

The backend of a compiler is very important for its performance. The implemented backend is simple and does not generate optimal code. The backend target the x86_64 instruction set. It does instruction selection using the greedy maximal munch algorithm [Appel 1997] and instruction scheduling using a simple list scheduling implementation [Skeppstedt 2012]. The register allocator is based on the simple Linear Scan working on SSA form [Wimmer and Franz 2010]. It allocates registers for one linear block at a time, linearly assigning register by choosing a register not used in the period where the register is alive. The allocator only looks a single block at a time, thus the allocation is not optimal globally. Furthermore, no low level optimization is performed during code generation. For simplicity all the features of the x86_64 instruction set is not used.

4. FEEDBACK INFRASTRUCTURE

To display feedback to the programmer and suggest automatic refactorings, we extend and use the infrastructure in Eclipse. This includes markers, yellow sticky notes in Eclipse used to present tasks, problems or general information about a source code construct [Glozic and McAffer 2001]. Eclipse contains views used for navigating or displaying a hierarchy of information. We have implemented a view, giving an overview of all the feedback presented to the programmer. This allows the programmer to get an overview of all the feedback given by the compiler.

An example of the kind of feedback we provide to the programmer is seen in figure 2. Here we inform the programmer that a function was not inlined and suggest an au-

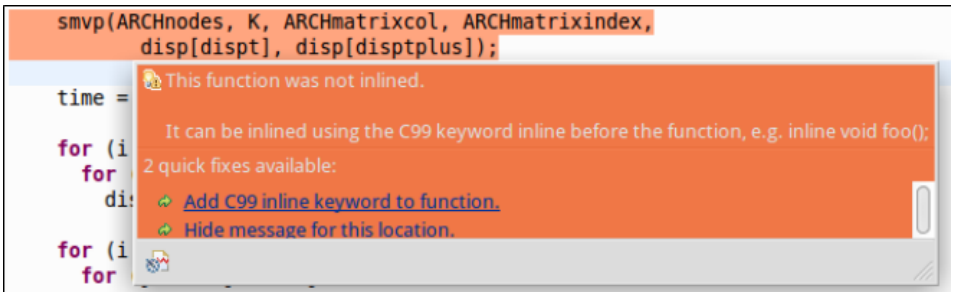


Fig. 2: Simple marker suggesting adding the C99 `inline` keyword to potentially enable inlining of the `smvp` function. Executing the suggested automatic refactoring gives a 22.6% improvement with GCC 4.8.1.

tomatic refactoring. This simple refactoring lead to a 22.6% improvement using GCC 4.8.1.

All optimizations in the compiler generate data on applied optimizations and data on missed optimizations. These show to the programmer how the compiler has optimized. Furthermore, the compiler gives details on why a given optimization was not applied and suggest an automatic refactoring. The modifications can be applied automatically, simply by accepting the compilers suggestion.

The tool can currently provide automatic refactorings for:

— Removing dead code
— Applying the `inline` keyword to functions
— Permuting the loop order
— Applying the `restrict` C99 keyword to pointers, even if matrix notation is used.

Mapping the internal representation used by the optimizer to the source code is important. Therefore, we have integrated with the IDE and keep Eclipse metadata in the IR during compilation, relating the IR to the Eclipse internal datastructures.

5. RESULTS

5.1. Setup

We evaluate our implementation on a platform based on a dual core 1.9Ghz Intel Core i7-3517U and a total of 6GB of DDR3 ram. The operating system is Linux with kernel version 3.8.0.

Furthermore the performance is compared to GCC 4.8.1 with no optimizations `-O0` and with the aggressive optimization level `-O3`.

The compiler have been evaluated on 12 kernel benchmarks. Three very simple kernels and nine kernel benchmarks from the Spec2000 benchmark suite [Henning 2000] and the Java Grande C benchmarks [Bull et al. 2001]. The benchmarks are of varying size, art and equake are the largest with 1042 lines of C code as measured using SLOCCount [Wheeler 2013]. The kernel benchmarks are chosen to represent resource demanding programs testing different elements of the system as they have high processing, I/O or memory demands.

In addition, the power of the tool as an optimization adviser for a production compiler is evaluated. This have been evaluated on an edge detecting program from the University of Toronto DSP Benchmark Suite. This benchmark have only been evaluated using GCC due to a bug in the compiler.

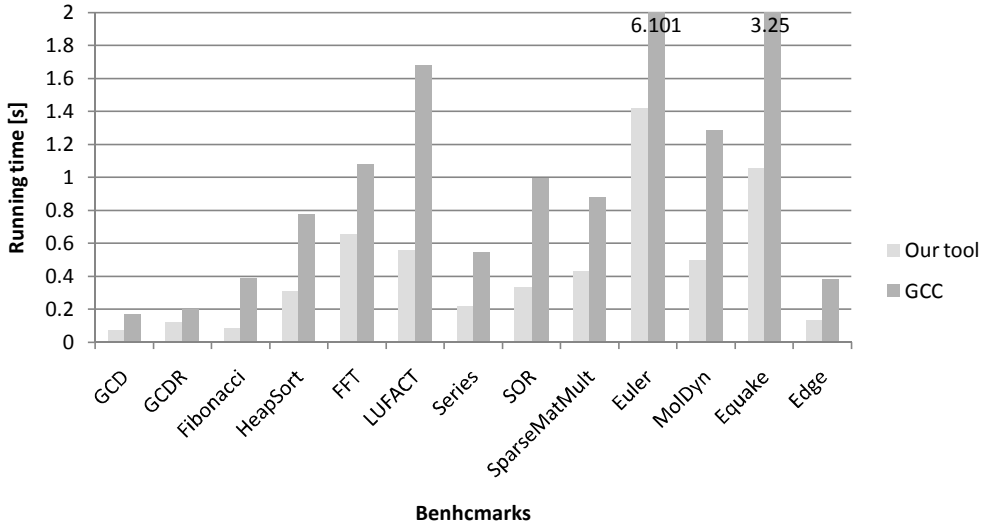


Fig. 3: Execution time for generating feedback and GCC compilation time. Lower is better.

5.2. Compilation Speed

The tool is embedded inside the Eclipse IDE and the tool integrates into the normal software analysis flow of Eclipse CDT. This means that we can reuse existing Eclipse data that is created in the IDE. Therefore our tool can give quick feedback on the source code during development. To this end, the tool must be fast to be integrated into the programmers flow.

The time used by our tool to generate feedback to the programmer for the benchmarks is shown in figure 3. In the figure the compilation time using GCC is also shown. When comparing the execution time of our tool and GCC it is clear that our tool is faster and can be used to generate comments faster.

Evaluation whether the tool is fast enough to integrate into the programmers development flow is harder, as it is a subjective opinion. It is fast enough to be used when the programmer pauses in writing, but for larger programs it might have an annoying delay while typing.

5.3. Compiler Performance

First we consider the performance of the compiler used as a static offline compiler, similar to how GCC normally would be used. I.e. we do not consider the compiler feedback in this section, only the quality of the generated code. The compiler have been evaluated using all 12 kernel benchmarks. The performance is reported first without any optimizations and second with all the implemented optimizations.

The performance data is presented in figure 4, where the running time of each benchmark is reported using our compiler and GCC. GCC outperforms our compiler. Even for the simple Fibonacci benchmark, GCC is 1.8 times faster. In the fibonacci benchmark GCC performs many optimizations on the source code. It inlines the recursive calls and use tail recursion optimization to turn the call into a loop. This decreases the amount of recursion significantly. In contrast we can only do simple scalar optimizations on the fibonacci kernel.

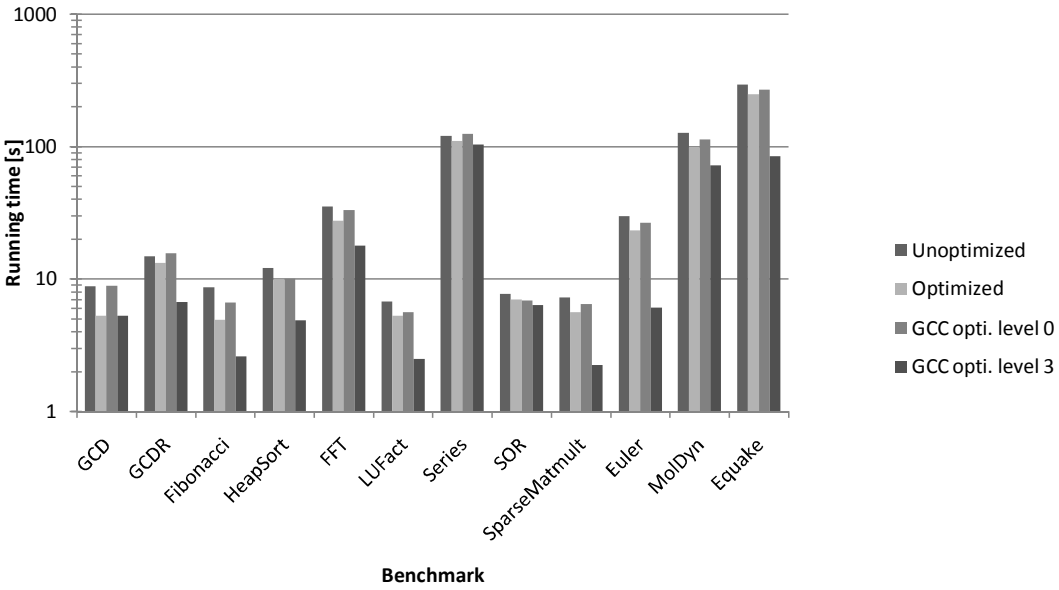


Fig. 4: Raw performance of our compiler and GCC 4.8.1. Lower is better.

Our compiler is designed to be lightweight and executed during development, for giving quick feedback to the programmer. In contrast GCC use optimizations that are significantly slower, but optimize more aggressively.

5.4. Feedback

The compiler can guide the programmer with automatic refactorings, potentially making the code more amenable to optimization.

The feedback adviser can only handle a specific number of cases where the compiler refrains from optimizing. Thereby, we cannot generally optimize all programs. Two benchmarks, the simple fibonacci and GCD kernels, did not benefit from the feedback advice. The other 10 kernels yielded small speedups. The speedup obtained, by using the feedback advice for small source code changes is shown in figure 6. The baseline is the optimized results from figure 4.

For the LUFact and Euler benchmarks the tool suggest automatic refactorings for applying the `inline` keyword to loop intensive functions and interchanging the order of some loops. However, the compiler can not automatically analyze whether the transformation is safe due to limitations in the dependency analysis. Therefore, we suggest an automatic refactoring and leaves the task of whether doing so is safe to the programmer as seen in figure 5. Performing the loop interchange refactoring yields a speedup of 12% for LUFact and 8% for Euler.

For the Equake benchmark the feedback suggests adding the `inline` keyword a function call to the `smvp` function. This both reduce the function call overhead, but also makes some optimizations more effective. Inlining the `smvp` function call yields a speedup of 7.6%.

5.5. Evaluate Feedback on Production Compilers

We also evaluate the feedback using a production compiler. One intended use of our tool is to be used during development and the refactored source code is then subsequently used with a production compiler.

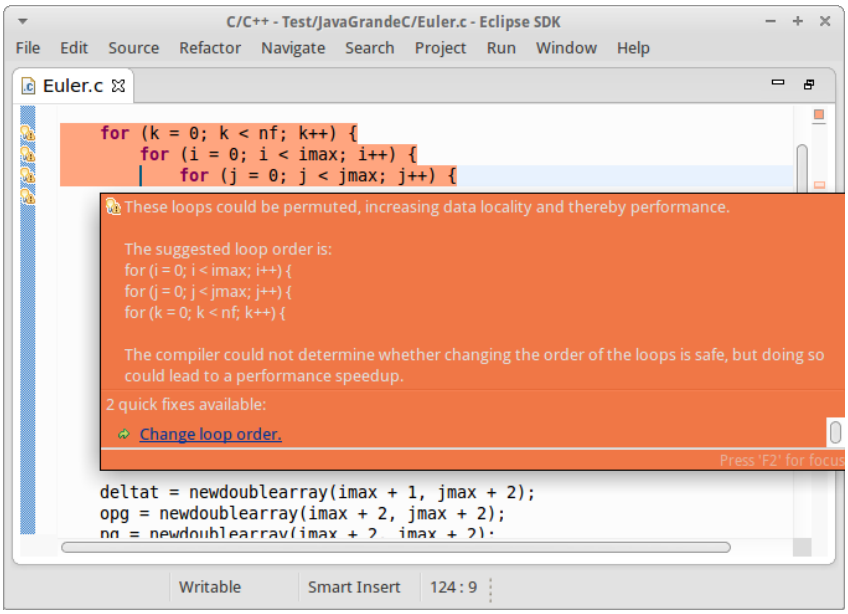


Fig. 5: Marker suggesting permuting the order of the loops, if the programmer determines doing so is safe.

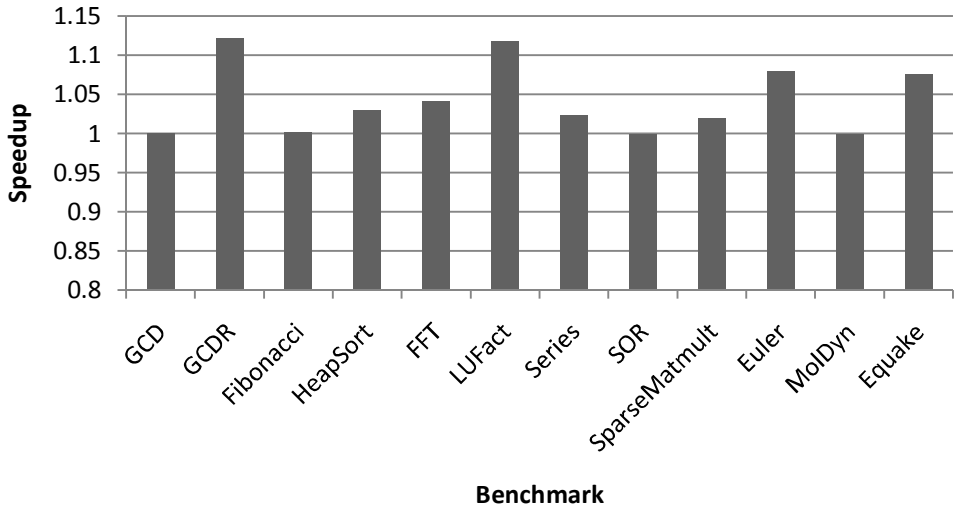


Fig. 6: Speedup using the feedback advice on each benchmark.

The kernel benchmarks have been optimized by applying the suggested automatic refactorings described in section 5.4 and then afterwards compiled with GCC 4.8.1 using the aggressive optimization level `-O3`.

The results are seen in figure 7. The baseline is the original source code, aggressively optimized with `-O3` using the same version of GCC.

Three of the kernels shows decent speedups, namely LUFact, Euler and Equake of 11%, 5% and 23% respectively. More inlining have widened the scope of GCC's opti-

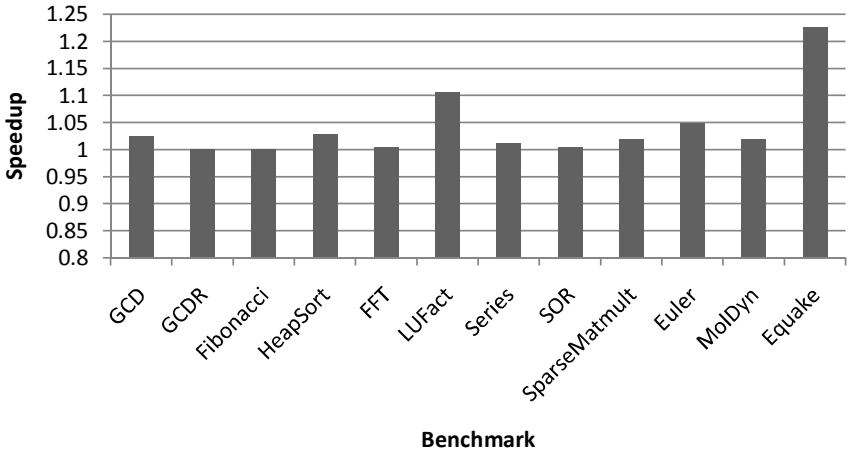


Fig. 7: Speedup using the feedback advice on each benchmark using GCC.

mizer, allowing it to optimize the compute intensive parts of the kernels more aggressively. The manually interchanged loops also yield small speedups, but as the loops are not part of the compute intensive part of the kernels the speedup is moderate.

5.5.1. Parallelization Adviser. Earlier research have shown the benefit of applying the `restrict` keyword to C pointers, mitigating issues preventing optimization [Larsen et al. 2011b]. The `restrict` keyword states that only the pointer itself, or a value derived directly from the it, is used for all accesses to that object. This makes the programmers intent clearer to the compiler, mitigating many issues preventing optimizations. It can benefit automatic parallelization, vectorization and loop transformations.

For evaluating the feedback when performing automatic parallelization, we study an edge detection application from the University of Toronto DSP Benchmark Suite. Our compiler can provide optimization advice on the edge detection kernel as seen in figure 8. Applying the C99 `restrict` keyword is simple on pointers, but may require more work on arrays. For example the array arguments are rewritten as: `int (*restrict input_image)[N]`. The suggested refactoring can be applied automatically by the tool.

The result have been evaluated on the system described in section 5 using GCC 4.8.1 and the compiler flags `-O3 -ftree-parallelize -loops=#threads -fopenmp` as seen in figure 9. The reported time is excluding the I/O activity in the beginning and end of the program. Three numbers are reported for the Intel i7 platform one and two threads respectively; the sequential execution, the speedup of GCC's automatic parallelization and the speedup after performing the automatic refactoring.

The best speedup is achieved with two threads, where the modified version is 16% faster than the automatically parallelized unmodified version.

We have also analyzed the edge detection application on a 2.66 GHz quad-core Intel Xeon X5550 system, with 24GB of ram running Linux kernel 2.6.32. The available compiler on this system was GCC 4.7.2. The evaluation is shown in figure 9. All loops were parallelized in the modified version, but most of the speedup comes from automatic vectorization. GCC chose to parallelize the inner loop and therefore the full potential of the parallelization is not achieved. The best speedup was 2.62 with eight threads compared to the sequential version.

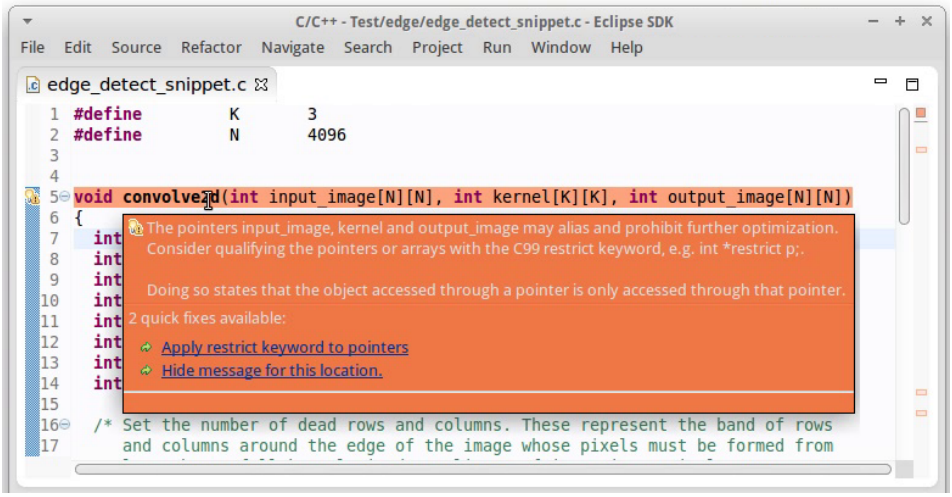


Fig. 8: Marker suggesting adding the restrict keyword to the three array arguments.

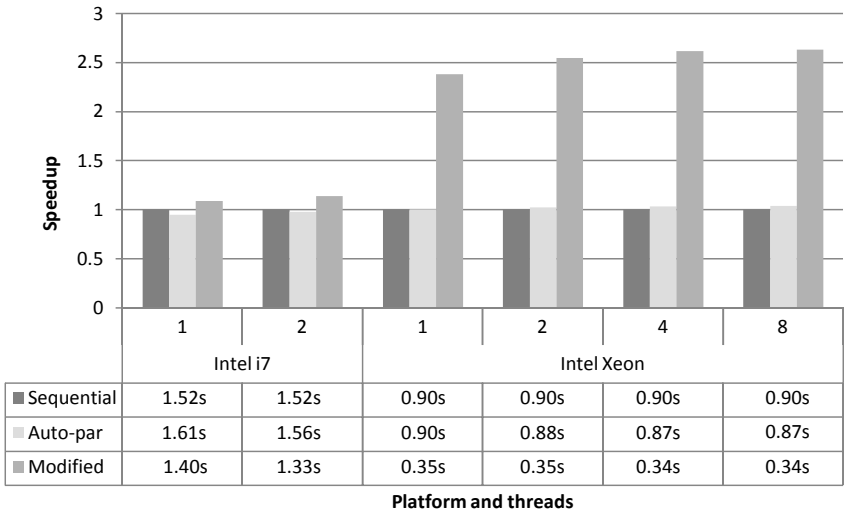
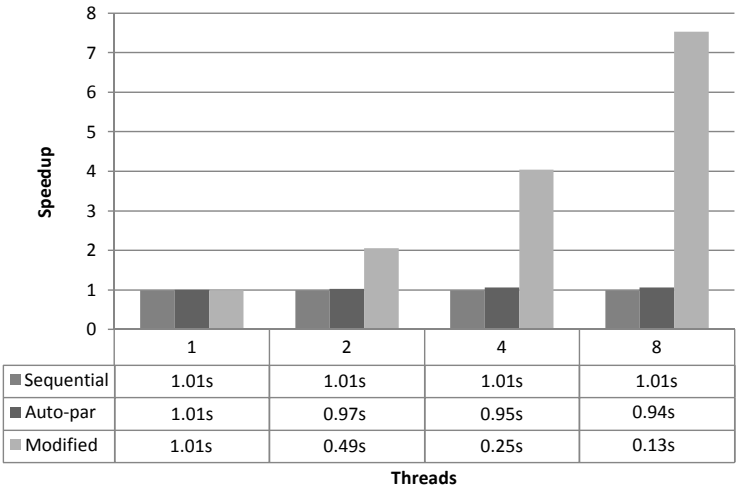


Fig. 9: Speedup and executing time of the edge-detection kernel on the (a) Intel i7 platform and (b) Intel Xeon platform. In the modified version using the feedback suggestions, all loops were parallelized. In the unmodified the outer loop were not parallized.

6. FUTURE WORK

One big challenge for feedback systems is when and what to display to the programmer. Feedback system can often generate many comments, but many are false positives and only a few will lead to a speedup. Therefore, one future direction is to filter the feedback by automatically applying the suggested automatic refactorings and validate whether the compilation benefited. The generated assembly can also be used to statically approximate the running time and use this information for filtering.

Another related research problem is which compiler flags to use. Compiling with a production compiler can be complicated, for example GCC 4.8.1 has over 250 optimiza-



	1	2	4	8
Sequential	1.01s	1.01s	1.01s	1.01s
Auto-par	1.01s	0.97s	0.95s	0.94s
Modified	1.01s	0.49s	0.25s	0.13s

Fig. 10: Speedup and execution time achieved with `-O2` and disabled vectorizer on the Intel Xeon Platform.

tion flags with many possible combinations. On the edge detection program, if we use the `-O2` optimization flag and disable the vectorizer, we can parallelize all loops giving a speedup of up to 7.46 as shown in figure 10. From a programmers perspective this combination of automatic refactorings and optimization flags is far from obvious.

Future work could take input from one or multiple production compilers, that can produce feedback on its optimization. Here multiple combinations of optimization flags could be used, by specifying known combinations that have been identified to be successful. This work could be integrated into our tool by reusing the existing refatorings and supplement the existing analysis. This will of course add extra compilation time, therefore this slower option could be run on the programmers request.

7. CONCLUSIONS

Many applications rely on automatic compiler optimization for optimal performance. Regrettably, programmers are often not aware of how their programs are optimized and especially which small refactorings can lead further optimization by the compiler. To address this issue we have developed our dynamic feedback compilation system, which can give feedback to the programmer during development directly in the IDE. Compared to other compiler driven feedback systems the advantage of the presented tool is dynamically executing the embedded compiler, as part of the normal Eclipse development feedback system and the IDE integration benefiting the automatic refactoring.

The implemented compiler have been evaluated on 12 C benchmarks, but compared to GCC does not yield good performance. This is expected, as the tool is designed to be fast and be executed continously as the programmer works. The optimization advise system have also been evaluated on the benchmarks. The feedback on inlining of function calls and automatically changing the order of loops lead to simple source code modifications, where 8 out of 12 benchmarks yield a speedup. The system can guide to a 12% improvement over the compiler optimized version.

The tool can also be used as a feedback adviser for a production compiler. Here 9 out of 12 of the benchmarks benefited from the feedback. On the Equake benchmark the tool can guide to a 23% improvement. Given the little work required by the program-

mer to obtain the speedup, the compiler driven feedback is cost effective. Furthermore it was also investigated on an edge detection program, how the feedback could improve the automatic parallelization performance using a production compiler. Here the tool could guide to a 153% performance improvement.

ACKNOWLEDGMENTS

This work has been partially funded by the ARTEMIS Joint Undertaking as part of the COPCAMS project (http://copcams.eu) under Grant Agreement number 332913. The authors acknowledge the HIPEAC3 European Network of Excellence.

REFERENCES

APPEL, A. W. 1997. *Modern Compiler Implementation in C: Basic Techniques*. Cambridge University Press, New York, NY, USA.

BULL, J. M., SMITH, L. A., POTTAGE, L., AND FREEMAN, R. 2001. Benchmarking java against c and fortran for scientific applications. In *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*.

CDT. 2013. Eclipse C/C++ Development Tooling. http://www.eclipse.org/cdt/. Accessed on 18/5/2013.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*.

DU, Y., VINAYAGAMOORTHY, K., YUEN, K., AND ZHANG, Y. 2012. Explore Optimization Opportunities with XML Transformation Reports in IBM XL C/C++ and XL Fortran for AIX Compilers. IBM developerWorks. Accessed on 17/5/2013.

FOUNDATION, F. S. GNU Binutils. http://www.gnu.org/s/binutils/. Accessed on 7/8/2013.

FOUNDATION, F. S. GNU Compiler Collection. http://gcc.gnu.org. Accessed on 7/8/2013.

GLOZIC, D. AND MCAFFER, J. 2001. Mark My Words. http://www.eclipse.org/articles/Article-Mark My Words/mark-my-words.html. Accessed on 7/4/2013.

HENNING, J. L. 2000. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer 33,* 7, 28–35.

INTEL. 2013. Intel Composer XE 2013. http://software.intel.com/en-us/intel-composer-xe. Accessed on 17/5/2013.

JENSEN, N., LARSEN, P., LADELSKY, R., ZAKS, A., AND KARLSSON, S. 2012. *Guiding Programmers to Higher Memory Performance*.

KENNEDY, R., CHAN, S., LIU, S.-M., LO, R., TU, P., AND CHOW, F. 1999. Partial redundancy elimination in ssa form. *ACM Trans. Program. Lang. Syst.*.

LARSEN, P., KARLSSON, S., AND MADSEN, J. 2011a. *Feedback Driven Annotation and Refactoring of Parallel Programs*. IMM-PHD-2011. Technical University of Denmark (DTU).

LARSEN, P., LADELSKY, R., KARLSSON, S., AND ZAKS, A. 2011b. *Compiler Driven Code Comments and Refactoring*.

LARSEN, P., LADELSKY, R., LIDMAN, J., MCKEE, S., KARLSSON, S., AND ZAKS, A. 2011c. *Automatic Loop Parallelization via Compiler Guided Refactoring*. IMM-Technical Report-2011. Technical University of Denmark.

LARSEN, P., LADELSKY, R., LIDMAN, J., MCKEE, S., KARLSSON, S., AND ZAKS, A. 2012. *Parallelizing More Loops with Compiler Guided Refactoring*.

MUCHNICK, S. S. 1997. *Advanced compiler design and implementation*. San Francisco, CA, USA.

NG, K., WARREN, M., GOLDE, P., AND HEJLSBERG, A. 2012. The Roslyn Project: Exposing the C# and VB compilers code analysis. Whitepaper. Microsoft.

ORACLE. 2013. Oracle solaris studio. http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html. Accessed on 17/5/2013.

SKEPPSTEDT, J. 2012. *An Introduction to the Theory of Optimizing Compilers*.

TORCZON, L. AND COOPER, K. 2011. *Engineering A Compiler* 2nd Ed. San Francisco, CA, USA.

WEGMAN, M. N. AND ZADECK, F. K. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst. 13,* 2, 181–210.

WHEELER, D. 2013. Sloccount.

WIMMER, C. AND FRANZ, M. 2010. Linear scan register allocation on ssa form. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*.