Guiding Programmers to Higher Memory Performance

Nicklas Bo Jensen[1], Per Larsen[1], Razya Ladelsky[2], Ayal Zaks[2], and Sven Karlsson[1]

[1] DTU Informatics, Technical University of Denmark
s082973@student.dtu.dk
{pl,ska}@imm.dtu.dk
[2] IBM Haifa Research Labs
razya@il.ibm.com

Abstract. Modern compilers use complex optimizations. It is often a problem for programmers to understand how source code should be written to enable optimizations. Interactive tools which guide programmers to higher performance are very important. We have developed such a tool that helps programmers modify their code to allow for aggressive optimization. In this paper, we extend it to support high level memory optimizations such as matrix reorganization. We evaluate the tool using two benchmarks and four different compilers. We show that it can guide the programmer to 22.9% higher performance.

1 Introduction

Optimizing compilers are complex and difficult for programmers to understand. Programmers often do not know how to write code that the compilers can optimize well. We have developed a tool that helps non expert programmers write code that the compiler can better understand [1] in an interactive way. It uses feedback generated by a production compiler, `gcc` from the GNU Compiler Collection [2]. We modified `gcc` to improve the quality and precision of the feedback. Our tool then interprets the feedback and presents it in a human readable form directly into the Eclipse integrated development environment, the Eclipse *IDE*. In principle, the techniques could have been applied to any compiler and *IDE*.

In this paper, we extend our tool to support high level memory optimizations such as matrix reorganization. These optimizations change the way matrixes are accessed resulting in improved memory hierarchy performance. We perform an performance evaluation using two benchmarks from SPEC2000 and four different compilers. We show that our enhanced tool can guide the programmer to 22.9% higher performance.

Although one can argue that compilers should be better at optimizing code, programmers required to write performance sensitive code often cannot wait for a more advanced compiler to be released. Our tool helps non expert programmers change their code to fully utilize all the optimizations an existing production

compiler can offer. Our work is thus complementary to the work on more precise analysis methods and aggressive compiler optimization passes.

In short, we make the following contributions:

- We have extended our tool significantly. It can now handle the matrix reorganization memory optimization.
- We have developed a code refactoring wizard which helps programmers apply changes directly to their code.
- We evaluate the extended tool and show substantial performance improvements on two SPEC2000 benchmarks and with several compilers.

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

2 Related Work

Compiler memory optimizations includes matrix-reorg [3] where matrixes can be automatically flattened and transposed based on profiling data. Other data layout transformations include structure layout optimizations [4,5]. They optimizes structures by decompositioning them into separate fields, substructures or reordering the fields in a structure. Other compiler optimizations focus on improved data locality through loop restructuring [6]. All these optimizations are part of the normal compiler optimization flow. Our tool provides feedback to the programmers so that they can change their most performance sensitive code to take full advantage of compiler optimizations.

Leung and Zahorjan have presented their implementation of array restructuring at run time [7]. They compare it with common forms of loop restructuring and find that it has comparable performance and in some cases even superior performance.

Most compilers include options to generate optimization reports. Both `gcc` and `xlc` includes optimization reports for matrix reorganization. These mostly state only what was done, and not what could have been done. `gcc` reports often refer to intermediary representation and without any explanations. In contrast, our tool provides human readable feedback directly into the integrated development environment.

This paper extends our previous work [1] in that we have significantly expanded the tool to provide feedback on high level memory optimizations.

3 Memory Optimization

Compilers have to generate correct code. To this end, compilers often conservatively decide not to apply optimizations in ambiguous cases where the intent of the programmer is not clear. The general idea behind our tool is to help programmers fully utilize modern advanced compiler optimizations. We propose that programmers will work actively with the performance critical sections

of their code. Our tool interactively provides the programmers with hints and suggestions for changing the code so that source code ambiguities are removed thereby facilitating the application of additional compiler optimizations. The tool consists of several parts. First, we link `gcc` [2] with a special library that we have developed. The library will augment `gcc`'s diagnostic dump files with information on the optimizations performed but also on optimizations not performed. These dump files are then read by an Eclipse plug-in that we have developed as well. The plug-in interprets and displays the information. The plug-in also suggests refactoring changes based on the extracted information [1]. Refactoring is a technique for modifying source code in a structured way, without changing the program's external behavior [8].

In this paper, we have extended our work to support `gcc`'s matrix reorganization [3] framework which applies to dynamically allocated matrixes. The framework consists of two optimizations: matrix flattening and matrix transposing. When a matrix is flattened an m-dimensional matrix is replaced with an n-dimensional matrix where $n < m$. This leads to fewer levels of indirection for matrix accesses. Part of such an optimization can be seen in Fig. 1. The matrix transposing optimization swaps rows and columns resulting in better cache locality depending on access patterns. Profiling is used by `gcc` to make decisions on what matrixes to transpose.

```
a = (int**) malloc (N)
for (i=0; i<N; i++)
   a[i] = (int *) malloc (M)
a[i][j]++
```

(a) Dynamically allocated non flattened two-dimensional C array.

$\longrightarrow$

```
a = (int*) malloc (N * M)


a[i*M+j]++;
```

(b) Dynamically allocated flattened one dimensional C array.

Fig. 1: Dynamically allocated arrays in C showing both non flattened and flattened versions.

Reorganizing matrixes is an intrusive operation: declarations, allocation, matrix access and deallocation sites have to be updated. The `gcc` compiler will therefore refrain from using the optimization unless it can analyze exactly how the matrix is used. In many cases, the compiler cannot fully analyze how a matrix is accessed. The matrix *escapes* analysis. This may happen if the matrix is an argument to a function, even if it would be safe to optimize. Here the analysis is conservative and chooses not to analyze even local functions. Only global dynamically allocated arrays are optimized. Manual vector expressions could lead to errors. Therefore, matrixes are not reorganized when vector or assembler operations exists. Additional restrictions exist. For example, `gcc` assumes only one matrix allocation and one or more accesses, therefore matrixes with multiple allocation sites will not be optimized. In general, many matrices are not optimized.

Our tool can help the programmer when the compiler refrains from optimizing. It does so by giving reasons why the compiler did not optimize and suggests changes if applicable. There exist many reasons why the optimizations do not apply so our tool prioritizes the information so that the most useful hints are shown first. One example is missing compiler options. If the correct options are not used, it will present a hint in the Eclipse IDE suggesting the programmer to change options.

As mentioned, if the matrix escapes the analysis it will often not be optimized. One common scenario is when a matrix is an argument to an external function or some other function for which the source code cannot be analyzed. Here our tool points out all escaping matrixes to the programmer. It will also describe why each matrix escapes as well as suggestions for how code can be rewritten. This may include using annotations to get the compiler to inline functions. More invasive refactorings can also be attempted. Instead of passing the matrix to a function, it might be possible to pass a temporary variable if individual matrix elements or their addresses are passed. This will often help the compiler analysis understand the source code better. Another solution, not implemented, is to use data copying in places where it is known where a matrix escapes and where it returns. In the escaping region the original version of the matrix can be used. This adds some overhead synchronizing data.

Many programmers have issues using the profiling features of `gcc` used by the matrix reorganization framework to determine whether matrixes should be transposed and how. It is easy to detect in the compiler whether profile guided optimization are disabled. Our tool shows messages to the programmer explaining the required steps to use profile guided optimization. This is seen in Fig. 2.

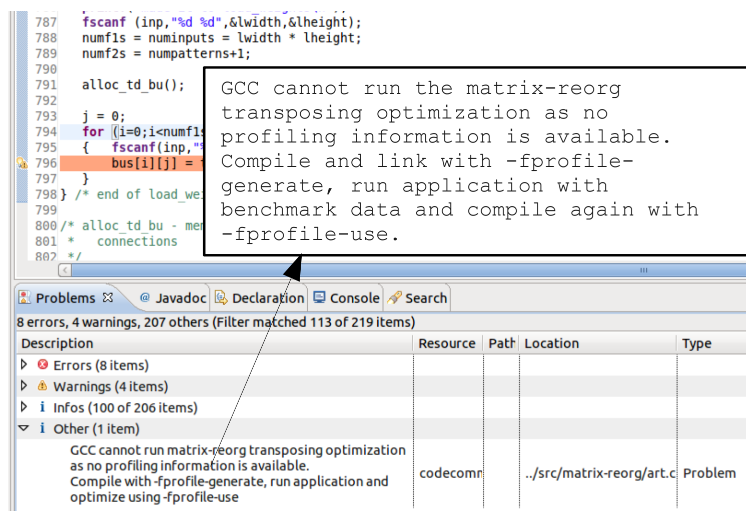


Fig. 2: Marker helping the programmer use profiling.

We have developed an automatically refactoring wizard. The wizard helps the programmer apply matrix flattening and transposing directly to the source code. This makes it possible for the programmer to turn off matrix reorganization in the compiler. This is sometimes necessary. For example, we have discovered at least one bug in `gcc` which forced us to manually rewrite the code. The wizard also allows us to apply code transformations using other compilers. This might be useful as many compilers do not have matrix reorganization optimizations. The wizard is implemented using Eclipse's refactoring framework and so have a familiar look and feel. The wizard will be offered to the programmer if the compiler found that the matrix could be optimized. Both full and partial flattening and transposing are supported. An example of the wizard can be seen in Fig. 3.

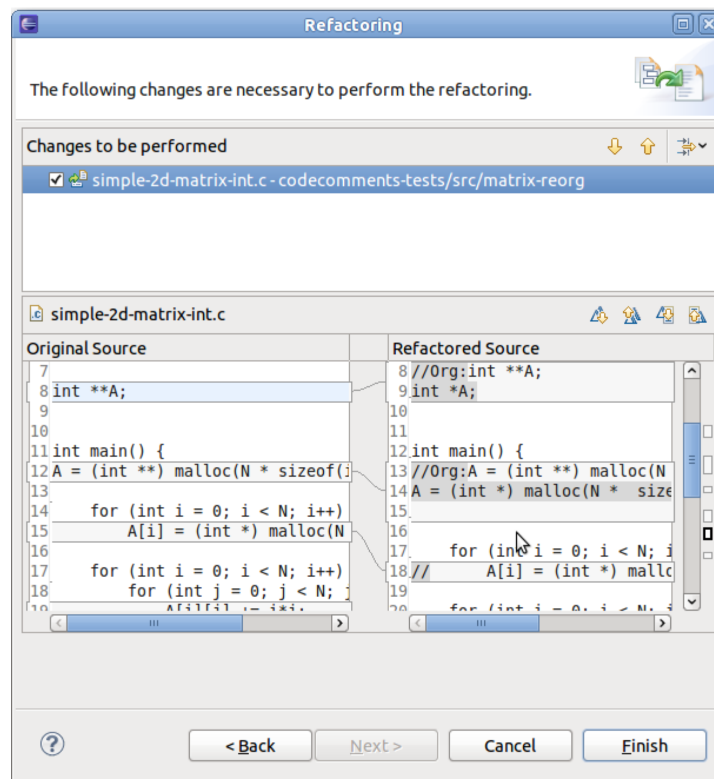


Fig. 3: The refactoring as it appears in the Eclipse IDE.

4 Experimental Results

We have applied our tool on two kernel benchmarks from the SPEC2000 benchmark suite: 179.art [9] and 183.equake [10]. The benchmarks have previously

been used to evaluate the matrix reorganization capabilities in `gcc` [3]. Good results were shown. A total of 35% improvement on 179.art and 9% on 183.equake. This indicates that the standard optimizations in `gcc` already are effective.

We will use our tool to further optimize the benchmarks. First by mitigating issues preventing optimizations and then by applying the optimization directly at the source code.

We have run experiments on two machines. We have used a Dell Poweredge 1900, with one Quad-core Intel Xeon E5320 processor at 1.86GHz and a level two cache of 2x4MB. The machine, called *xeonserver*, runs Debian 6.0.2.1. The second machine is an iMac G5 with a 1.8GHz PowerPC 970fx processor. It is called *g5server* and runs Red Hat Enterprise Linux 5.5.

We have used four different compilers. An overview of compilers, versions and host systems is shown in Table 1. The compiler flags used are shown in Table 2. All experiments have been run 50 times and the average benchmark execution time has been used.

Table 1: Name, version and host machine of used compilers.

Compiler	Version	Host
`gcc` [2]	The GNU Compiler Collection 4.5.1	xeonserver
`icc` [11]	Intel C++ Composer XE 2011 for Linux 12.0.4	xeonserver
`suncc` [12]	Oracle Solaris Studio 12.2 Linux	xeonserver
`xlc` [13]	IBM XL C/C++ for Linux 11.1	g5server

Table 2: Compiler options used.

Compiler	options
`gcc`	`-O3 -fipa-matrix-reorg -fwhole-program -std=c99`
`icc`	`-fast -ipo`
`suncc`	`-fast -xc99`
`xlc`	`-O2 -qhot -qipa=level=2`

4.1 Case study: 179.art

The 179.art kernel benchmark is using neural networks to recognize objects in thermal images. It consists of a training process where the neural network learns from test images and an analysis process where it is matching a thermal image against the training images [9]. The training process is short. We will only use the execution time for the analysis process when evaluating performance.

The program consists of 1042 lines of C code as measured using SLOC-Count [14]. It contains a number of matrixes but only three global multidimensional dynamically allocated matrixes are candidates for optimization.

gcc can optimize two of them, the tds and bus matrixes, automatically but the cimage matrix escapes. The matrix is not an argument to a function and therefore it should have been optimized by gcc. We found that the escape analysis in gcc has problems with char arrays. This issue is due to an internal limitation in the compiler's analysis which could be overcome by changing the datatype, e.g. to an integer.

When profiling the program, the bus matrix was transposed meaning that the dimensions of the matrix were swapped.

We also used the IBM XL C/C++ compiler, xlc, to compile the kernel. Here all three matrixes were flattened and two were chosen for transposing. xlc does not need profiling data to determine whether transposing is useful and statically evaluates this.

The optimizations each compiler performed automatically are shown in Table 3. One thing that stands out from this experiment is that xlc has a more precise program analysis. It will probably optimize more aggressively.

Table 3: Global multidimensional dynamically allocated matrixes in 179.art and whether they are optimized automatically by gcc and xlc.

Declaration	gcc	xlc
unsigned char **cimage	Not optimized, matrix escapes	Flattened
double **tds	Flattened	Flattened and transposed
double **bus	Flattened and transposed	Flattened and transposed

Our tool could not guide the programmer to a version where the cimage matrix in 179.art is optimized by the gcc compiler. Therefore results are only presented for the unmodified original program and a version where two of the matrixes are optimized at the source code level obtained using the wizard. The wizard optimizes the matrixes that gcc can automatically optimize as seen in Table 3.

This, however, did not give any significant speedup over gcc's own optimization. In Fig. 4 it can be seen that using gcc a speedup of 1.16 was possible. This is only due to the wizard utilizing the profiling data to transpose a matrix. Using profile guided optimization the compiler does an equally good job.

We also have made experiments with multiple compilers. The performance results for 179.art on xeonserver are summarized in Fig. 4. Using suncc a speedup factor of 1.36x was achieved over the original version. icc produced the fastest program but did not benefit from the optimization with a speedup of one. The last compiler is xlc. It has the matrix reordering optimization. However, it performed worse with the optimization as seen in Fig. 4 with a speedup of 0.8. One reason might be that gcc chooses only to transpose one of the matrixes and xlc transposes two of them. As the wizard applies optimization at the source code it prevents xlc from further optimizing matrixes.

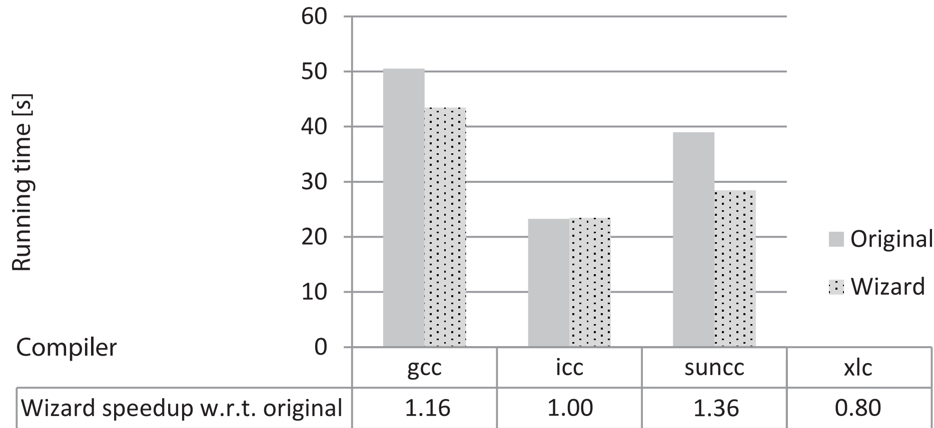


Fig. 4: Execution time and speedup of original and wizard optimized 179.art compiled with `gcc`, `icc` and `suncc`. Speedups obtained when using `xlc` are included. The most aggressive optimization options have been used but not profile guided optimization.

4.2 Case study: 183.equake

The 183.equake kernel is a program that simulates seismic waves propagation [10]. It contains ten global multidimensional dynamically allocated matrixes. All are candidates for optimization. Six out of the ten are automatically recognized as optimizable and the refactoring wizard is offered.

Two matrixes `ARCHcoord` and `ARCHvertex` escape as they are input to a function. If a matrix escapes we cannot always analyze all access sites. The compiler tends to choose to be conservative and back off when meeting a matrix passed as an argument to a function. However, it may be possible to inline the functions which will solve the problem. Using our tool it was identified that the `fscanf` C library function was the problem. Functions from the C library cannot be inlined using annotations. However, only the address of a single element in the matrix was passed as an argument. We chose to handle this situation by inserting a temporary variable as seen in Fig. 5. This is possible as only a single element of the matrix is passed as an argument. This allowed the compiler to optimize both `ARCHvertex` and `ARCHcoord` matrixes.

The matrixes `K` and `disp` also escape. Using our tool, which shows the function calls causing the matrixes to escape, it was possible to identify the function `smvp` as the main problem. In Fig. 6, we show how information is presented in the IDE. To resolve the problem, our tool proposes to inline the function if possible. We did that by using the `__attribute__((always_inline))` annotation which we applied to the function prototype. If only `inline` was used, the compiler determined that it is not advantageous to inline the function. However, with `always_inline` the compiler is forced to inline it. After the function was

```
fscanf(packfile, "%d",
  &ARCHvertex[i][j]);
```

(a) Original source.

```
int tmp;
fscanf(packfile, "%d", &tmp);
ARCHvertex[i][j] = tmp;
```

(b) Modified source.

Fig. 5: Modifications needed for `gcc` to optimize `ARCHvertex`. In (a) the compiler backs off but in (b) it is clear to the compiler that the matrix can be optimized.

inlined the matrixes are now chosen for optimization and it was thus possible to help the compiler when its analysis is too primitive or conservative.

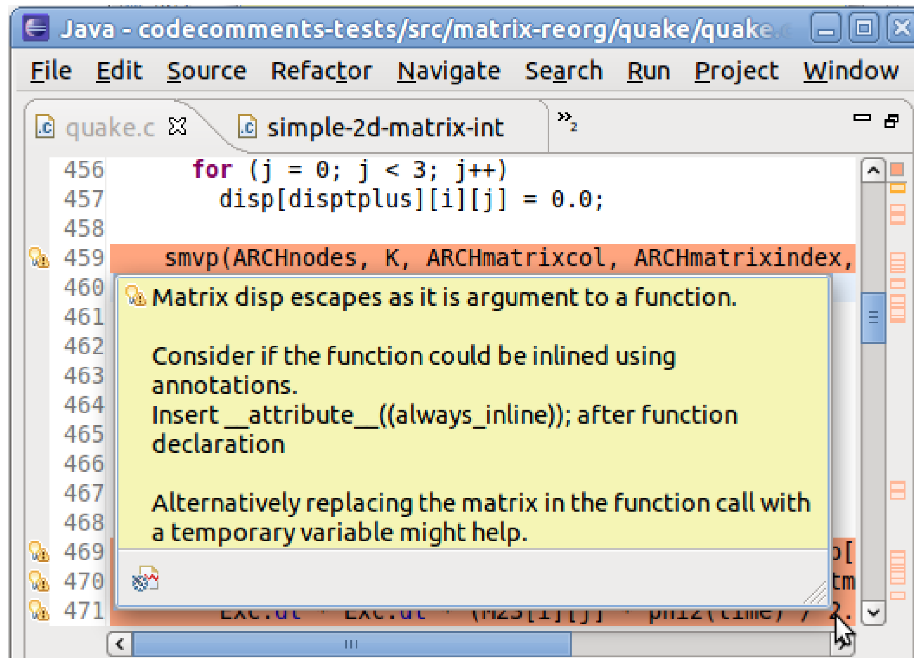


Fig. 6: IDE output showing which function call makes the matrix escape and suggested changes.

All global matrixes can be optimized after applying the aforementioned changes suggested by our tool. We changed three source code lines and added four lines. The modifications are minor and do not affect readability.

Unfortunately, we found a bug in `gcc` when using the optimized source code. Under certain circumstances the matrix reorganization optimization might introduce a wrong malloc allocation size. This means that the generated executable will return with an error and stop execution. This is not a bug in our tool,

but purely in the optimization pass of the gcc compiler. When flattening the matrix allocation sites, it will not include the allocation statement for all dimensions. This bug has been reported to the GCC Bugzilla [15]. We changed the benchmark so that the affected matrix, disp, is not optimized. Therefore in the presented results, the disp matrix has not been flattened.

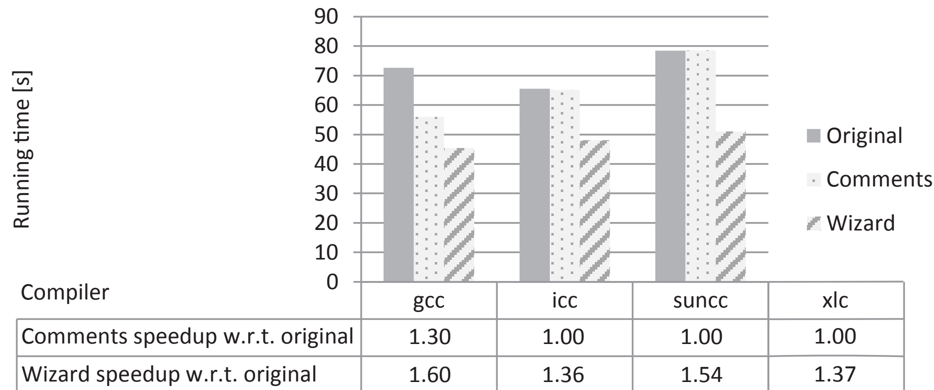
The 183.equake kernel was also optimized using xlc. It can optimize local matrixes and not just global. In our case, it could optimize two matrixes which gcc could not. Both gcc and xlc could optimize the same global dynamically allocated matrixes in the original source code. However after our tool has been used, gcc performs better. Now eight out of the ten matrixes are optimized. The reason for xlc not optimizing the last matrixes, like gcc did, appears to be that the compiler will only optimize two dimensional matrixes. gcc chooses not to transpose any matrixes with profile guide optimization. However, xlc chose to transpose some of the matrixes. Table 4 shows how each matrix were optimized by each compiler.

Table 4: Global multidimensional dynamically allocated matrixes in 183.equake and whether they are optimized automatically by gcc and xlc.

Matrix declaration	gcc	xlc
double **ARCHcoord	Flattened with help	Flattened with help
int **ARCHvertex	Flattened with help	Flattened with help
double **M	Flattened	Flattened and transposed
double **C	Flattened	Flattened and transposed
double **M23	Flattened	Flattened and transposed
double **C23	Flattened	Flattened and transposed
double **V23	Flattened	Flattened and transposed
double **vel	Flattened	Flattened and transposed
double ***disp	Flattened with help	Not flattened
double ***K	Flattened with help	Not flattened

Our tool could guide the programmer using code comments to rewrite the code in a simple way which led to more matrixes being optimized. We saw a speedup of 1.30 when gcc was used. A speedup of 1.6 was achieved, using gcc, if the optimizations were applied on the source code level using the wizard. This speedup is not only the result of the compiler optimizing more matrixes. In this case, the changes to the source code made it possible for the compiler to apply more aggressive optimizations. The results, seen in Fig. 7, show that with minor programming effort significant improvements in performance are possible.

We also tried the modified code with multiple compilers. The speedups are shown in Fig. 7. Only gcc could benefit from the code changes suggested by our tool and they did not affect the other compilers. All compilers did benefit from the changes made by the wizard with improvements in performance.



Compiler	gcc	icc	suncc	xlc
Comments speedup w.r.t. original	1.30	1.00	1.00	1.00
Wizard speedup w.r.t. original	1.60	1.36	1.54	1.37

Fig. 7: Execution time and speedup of original, comments and wizard optimized 183.equake compiled with `gcc`, `icc` and `suncc`. Speedups obtained when using `xlc` are included. The most aggressive optimization options have been used but not profile guided optimization.

5 Conclusions

We have extended an interactive compilation tool to support high level memory optimizations – matrix reorganization. The tool presents information in the Eclipse IDE guiding programmers to write source code which can be aggressively optimized. The tool uses feedback generated by the `gcc` compiler. We have complemented the tool with a refactoring wizard which applies the compiler optimization directly into the source code. This allows for optimized source code to be used with multiple compilers.

We have evaluated our tool using two SPEC2000 benchmarks. Our results show that it is not always possible to present good hints to the programmer. This was noticed for the 179.art benchmark, where the compiler has issues with a code pattern and no good solution is available. Modified source code was evaluated with four different compilers. We could show that two compilers benefited from the optimized source and two did not.

Better results were possible for the second benchmark 183.equake. Here the tool helped us rewrite the code, in a very simple manner, to allow all possible matrixes to be optimized. The modified source code yield a speedup of 1.3 using `gcc`. Using the refactoring wizard to make more invasive changes resulted in a larger speedup of 1.6. The refactored code was also compiled with multiple compilers resulting in speedups for the executable code of 1.36 to 1.6.

Acknowledgment

The research leading to these results has received funding from the ARTEMIS Joint Undertaking under grant agreement number 100230 and from the national

programmes / funding authorities. The authors acknowledge the HiPEAC2 European Network of Excellence, and thank Gadi Haber for initial discussions and motivation.

References

1. Larsen, P., Ladelsky, R., Karlsson, S., Zaks, A.: Compiler Driven Code Comments and Refactoring. In: Proc. of Fourth Workshop on Programmability Issues for Multi-Core Computers. (2011)
2. Free Software Foundation: GNU Compiler Collection. http://gcc.gnu.org. Accessed on 22/7/2011.
3. Ladelsky, R.: Matrix flattening and transposing in GCC. In: Proc. of GCC Developer's Summit. (2006)
4. Kistler, T., Franz, M.: Automated data-member layout of heap objects to improve memory-hierarchy performance. ACM Transactions on Programming Languages and Systems **22** (2000) 490–505.
5. Golovanevsky, O., Ladelsky, R.: Struct-reorg: Current status and future perspectives. In: Proc. of GCC Developer's Summit. (2007)
6. Berlin, D., Edelsohn, D.: High-level loop optimizations for gcc. In: Proc. of GCC Developers Summit. (2004)
7. Leung, S., Zahorjan, J.: Optimizing data locality by array restructuring. Technical report, University of Washington - Department of Computer Science and Engineering. (1995)
8. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (1999)
9. Roberson, C., Domeika, M.: 179.art SPEC CPU2000 Benchmark Description File. http://www.spec.org/cpu2000/CFP2000/179.art/docs/179.art.html. Accessed on 14/6/2011.
10. O'Hallaron, D.R., Kallivokas, L.F.: 183.equake spec cpu2000 benchmark description file. http://www.spec.org/cpu2000/CFP2000/183.equake/docs/183.equake.html. Accessed on 14/6/2011.
11. Intel: Composer XE 2011 for Linux. http://software.intel.com/en-us/articles/intel-composer-xe/. Accessed on 22/7/2011.
12. Oracle: Oracle solaris studio. http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/. Accessed on 22/7/2011.
13. IBM: XL C/C++ for Linux. http://www-01.ibm.com/software/awdtools/xlcpp/linux/. Accessed on 28/7/2011.
14. Wheeler, D.A.: Sloccount. http://www.dwheeler.com/sloccount/. Accessed on 14/6/2011.
15. Jensen, N.B.: Gcc bugzilla bug 49916. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=49916. Accessed on 31/7/2011.