Compiler Driven Code Comments and Refactoring

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

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

Abstract. Helping programmers write parallel software is an urgent problem given the popularity of multi-core architectures. Engineering compilers which automatically parallelize and vectorize code has turned out to be very challenging. Compilers are not powerful enough to exploit all opportunities for optimization in a code fragment – rather, they are selective with respect to the coding patterns they optimize. However, even minor code changes may enable optimizations to succeed where they previously would not.

We present an interactive approach and a tool set which leverages advanced compiler analysis and optimizations while retaining programmer control over the source code and its transformation. Our tool-set provides comments describing why optimizations do not apply to a code fragment and suggests workarounds which can be applied automatically. We demonstrate the ability of our tool to transform code, and suggest code refactoring that increase its amenability to optimization.

Experiments with an image processing benchmark shows that automatic loop parallelization of the original code result in minor speed-ups or significant slow- downs with the GNU C compiler, `gcc`. With our approach, automatic parallelization yields 8.6 best-case speedup over the sequential version. The transformations and suggestions are based on the loop parallelization and matrix reorganization capabilities in the latest production release of `gcc`.

1 Introduction

Programmers can no longer depend on increasing single-thread performance. In recent years, execution speed of a single processor core has only improved marginally. Instead, the number of processor cores has increased. To fully utilize these cores, software must be made parallel.

The programmer must shoulder additional burdens to ensure correctness and efficiency when writing parallel software. Compilers can assist by automatically parallelizing [5], vectorizing [11,12,15] and improving locality of memory references [7,8] of some code fragments.

Analysis done by compilers rely on simplifying assumptions to make optimization possible. This means that different translations of an algorithm into source code are not optimized with the same probability. Effectively, compilers

are very selective when choosing code fragments to optimize. A recent study of vectorizing compilers from Intel and IBM showed that 51 out of 134 code fragments were optimized by one compiler but not the other [6].

When the integrated development environment, IDE, can inform the programmer why a code pattern cannot be parallelized or vectorized, she stands a better chance of making it optimizable. In addition, the effort required to make a code pattern optimizable can be reduced by suggesting how the code can be refactored *and* automating the transformations including the ability to preview and undo changes.

This paper describes a tool-set which transforms the traditional compilation process. The compiler no longer acts as a passive component which merely optimizes the input. Rather, it also generates comments which explain to non-expert programmers where and why optimizations did not apply. The compiler also generates comments to show where source-code optimizations are applicable.

Our work takes advantage of the built-in capabilities of IDEs to highlight the code comments directly in the source code alongside compiler errors and warnings. In addition, the tool may suggest to the programmer how to transform the source code based on the code comments. Lastly, the refactoring infrastructure of the IDE is used to automate the code transformations which are accepted by the programmer.

The benefits are threefold. First, it is possible to explain in detail why a code fragment was not optimized and, in some cases, suggest how the programmer can change the code to allow the optimization in question. This helps increasing the portion of the codebase that the compiler is able to optimize.

Second, our *programmer-in-the-loop* approach to optimization is able to take advantage of the programmers understanding of conceptual, algorithmic and structural aspects of a program. Consider loop parallelization for instance. Two questions must be answered to decide if iterations of a loop should execute in parallel or not. Can the loop iterations be grouped such that there are no true data-dependences among them? If so, is parallelization likely to be profitable? The recent releases of the GNU Compiler Collection [4], `gcc`, contains state-of-the-art algorithms [5,14] to compute data dependences in loops. The heuristic that estimates the profitability of parallelization only considers iteration count, however. We expect the programmer to have a richer understanding of the amount of work embodied in a loop – and, in general, have enough knowledge of the target architecture to determine profitability of parallelization.

Third, the tool makes it possible to apply and track optimizations at the source level, i.e. not only at the level of machine code. Some developers refrain from turning on traditional optimizations to maintain accurate debug information in spite of the associated performance loss. Applying optimizations on the source level allows performance improvements even in such cases. It also enables the source code to carry optimizations from one compiler to another. Consider a product that must be compiled with compiler X which lacks an important optimization such as automatic loop parallelization. With our approach it is possible to use a another compiler, Y, which supports loop parallelization and apply the

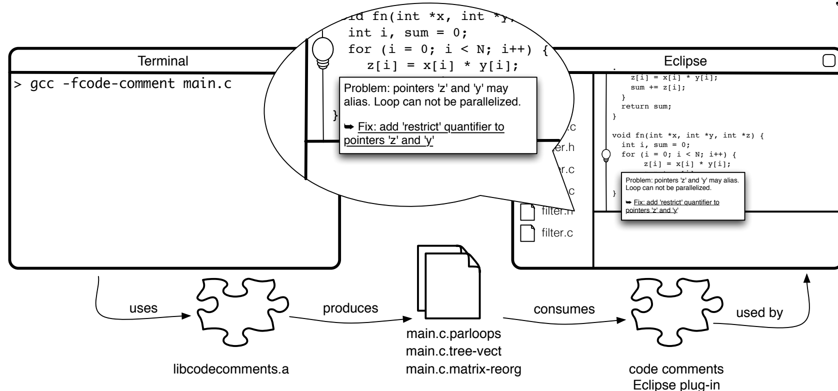

Fig. 1: Illustration of the code comments tool-set. A library extends `gcc` with the capability to output code comments in its diagnostic dump files. A plug-in extends the Eclipse CDT environment with the functionality to i) read the compiler generated code comments ii) display them at the proper places in the source code and iii) provide refactoring support for the changes suggested by the code comments.

parallelization achieved by compiler Y at the source level and subsequently compile the final product using compiler X. For instance, applications for the Apple iOS mobile operating system must be compiled with an older `gcc` release which does not support auto-parallelization.

Our prototype implementation currently supports comments for auto-parallelization, vectorization and matrix flattening optimizations in `gcc`. We currently target programs written in C/C++ but the core approach extends to all languages supported by the compiler. The current refactoring capabilities include insertion of OpenMP pragmas for loop parallelization, adding the `restrict` keyword to function parameters and, finally, declaring that functions are pure.

The edge detection application in the UTDSP benchmark suite [9] was used to measure the impact of our tool-set. Using the code comments, a single function signature was modified to work around a problem of possible aliasing of the function parameters. The change allows parallelization of the most computationally intensive loop and allows a best case speedup of 5 over sequential execution on a dual socket Intel machine and a 8.6 speedup on an IBM POWER6 blade. In both cases, the maximum speedup is obtained with 8 threads.

The rest of the paper presents the code comments tool in Sect. 2. Section 3 gives concrete examples how the tool helps programmers improve source code and Sect. 4 presents the preliminary experimental results. Finally, Sect. 5 surveys related work before Sect. 6 concludes.

2 Tool Overview

The compiler and an IDE are key components when developing software. Our tool extends both of these as illustrated in Fig. 1. Extending the compiler rather

than implementing our own analysis framework lets us take advantage of the existing, production-quality program-analysis facilities and optimization heuristics. Limitations in the compiler, of course, are inherited too.

The compiler is extended such that it produces informative messages - code comments - in addition to warnings and errors. As soon as an analysis pass determines that its corresponding optimization is possible or not, a code comment is generated. Whereas warnings and errors are usually written to the standard streams, code comments are written to compiler diagnostic files. Therefore comments are only generated when diagnostic or dump files are enabled for a particular analysis pass.

Currently, the analysis passes which implement auto-parallelization and matrix-reordering have been extended to generate code comments. An analysis pass typically consists of a sequence of steps to determine if an optimization is possible followed by a sequence of steps that transform the code. If one of the feasibility checks fails, a code comment is generated to describe the nature of the problem. If possible, any variables or expressions that are part of the problem and their locations in the source code are also output. If, on the other hand, an optimization is feasible, the code comment contains enough information to allow the IDE to transform the source code.

An analysis pass skips to the next code fragment as soon as an issue preventing optimization is discovered. Therefore, the code comments only report the first issue that prevents optimization although there may be several issue. The programmer must therefore resolve each issue to discover the next. Reporting multiple issues for the same code fragment would require a substantial re-engineering of the compilers analysis passes.

The analysis passes operate on a low-level, intermediate representation, IR, which differs substantially from the source code and therefore makes little sense to the programmer. Rather than describing an issue preventing optimization in terms of the intermediate representation, we reconstruct the source-level expressions. The reconstructed source-level expressions are not necessarily identical to the code written by the programmer because we do not take formatting and preprocessor definitions into account. Taking these things into account may be very helpful in practice. We have not done so, however, as it is mainly an engineering task of little research value.

A small library, `libcodecomments`, was added to `gcc`. It contains functionality that is commonly needed when generating code comments. This includes functions which convert the compilers intermediate representation into source-level expressions and code finding the source code location of loops, variables and functions.

The Eclipse C Development Tools, CDT, was extended to provide IDE support. A plug-in enables CDT to read the code comments from dump files generated by `gcc` during compilation. The reading of the dump files is done by extending the Eclipse build process and requires no programmer intervention besides enabling the code comments. This is currently done on a project-by-project basis. Code comments read from the compiler dump files are converted into *markers*,

which are shown as icons in the left margin of the code in the Eclipse source editor. The markers automatically track the source code construct, say a loop or variable, which is associated with the comment. The comment may include an option for resolution (refactoring), in which case the source code is changed. For example, lines may be added or deleted around the construct. A description of the marker is shown in the *Problems* view in Eclipse, and pops up when the cursor hovers over the marked code as shown in the call-out in Fig. 1. Similar to compiler warnings and errors, the code comments are automatically updated after each full or incremental build.

3 Code Comments and Refactoring

The code comments tool currently generates comments for the following analyses: loop auto-parallelization, auto-vectorization and matrix reorganization. This section showcases the current capabilities of our tool.

3.1 Auto Parallelization

Recent versions of `gcc` have added support for automatic parallelization of regular loop nests. It includes support for recognition of reductions and private variables. Loop parallelization works by inserting calls to the OpenMP runtime just as if the programmer had inserted directives in the source code. However, with `gcc` auto- parallelization, no directives are inserted at the source code level. parallelization pass builds on the existing support for explicit parallelization with OpenMP [13] and the data-dependency analysis used for auto-vectorization.

We extended the loop auto-parallelization pass to output its analysis results. This allows us to parallelize the loop at the source level. Parallelization is done by inserting the `omp parallel for` directive before the loop nest to be parallelized. The compiler dump file was extended with the following information for each parallelizable loop: i) file name and line number of the loop, ii) a list of zero or more `private` clauses; and iii) a list of zero or more `reduction` clauses. The `private` clauses are used to instruct the subsequent OpenMP analysis pass to create per-thread copies of variables mentioned in these clauses. For example, induction variables in every sub-loop nested inside the loop being parallelized need to be private. The `reduction` clauses cause special code to be generated for reduction variables recognized by the loop parallelization pass. Figure 2 gives an example which includes the use of these two clauses.

Each thread should have its own copy of the induction variables `j` and `i`. The variable `j`is implicitly made private by the `parallel for` directive. It is easy to forget to also declare the induction variable `i` as private when parallelizing the loop manually.

Special cases are loops where the reduction operation is *minimum* or *maximum* of two values. The compiler recognizes these operations in the C code, but the `reduction` clause in OpenMP for C only allows the use of primitive, commutative operators such as addition and multiplication [13, Section

```
 1 int sum_matrix(int A[N][N]) {
 2   int i, j, x[N], sum = 0;
 3
 4   for (j = 0; j < N; j++) {
 5     for (i = 0; i < N; i++) {
 6       x[i] = A[i][j];
 7     }
 8     sum += x[j];
 9   }
10   return sum;
11 }
```

accept

```
 1 int sum_matrix(int A[N][N]) {
 2   int i, j, x[N], sum = 0;
 3
 4 #pragma omp parallel for \
     reduction(sum, +) private(i)
 5   for (j = 0; j < N; j++) {
 6     for (i = 0; i < N; i++) {
 7       x[i] = A[i][j];
 8     }
 9     sum += x[j];
10   }
11   return sum;
12 }
```

Fig. 2: Refactoring which parallelizes a loop nest. If user accepts the refactoring an OpenMP clause is introduced with the `reduction` and `private` clauses necessary to ensure correct execution on multiple threads.

2.9.3.6]. The code may still be parallelized by inserting several OpenMP directives in the source code. Instead of inserting a single `#pragma omp parallel for reduction(...)`, three directives are used instead: `#pragma omp parallel`, `#pragma omp for` and `#pragma omp critical`.

3.2 Matrix Reorganization

The matrix flattening and transposing pass in `gcc` can optimize the data layout of matrices. This reduces the level of indirection when accessing multi-dimensional arrays in C [8]. This is done by replacing a multi-dimensional matrix with a lower-dimensional one and, optionally, transposing the elements.

The matrix reorganization pass is an invasive optimization in the sense that it must have access to the whole program to change all access sites of the matrix. This means that the optimization, when applied to the source code, potentially cause many files to be modified and the programmer must assert that the compiler can discover all matrix access sites during compilation. By leveraging the built-in refactoring support, Eclipse can be modified to preview changes before they are applied and changes can be undone with a single keystroke as long as the IDE stays open. This is convenient if it is subsequently determined that the program does not profit from that particular optimization. Figure 3 illustrates the flattening of a 3-dimensional matrix at the source code level. Refactoring previews are yet to be implemented in our tool.

3.3 Missed Opportunities for Optimization

There are many situations where compilers do not apply optimizations, regretfully. Some cases are essential while others are not. The major causes we encountered during this work are i) pointer-aliasing, ii) data which escapes from the unit of compilation, iii) complex control flow, iv) complex data dependencies, v) function calls inside loop bodies; and vi) code containing vector intrinsics or inline assembly.

```
 1 int mat3d() {
 2   A = (int ***)malloc(N * sizeof(int**));
 3   for (int i = 0; i < N; i++) {
 4     A[i] = (int **)malloc(N * sizeof(int *));
 5     for (int j = 0; j < N; j++)
 6       A[i][j] = (int *)malloc(K * sizeof(int));
 7   }
 8   for (int i = 0; i < N; i++)
 9     for (int j = 0; j < N; j++)
10       for (int k = 0; k < K; k++)
11         A[i][j][k] += 1;
12   printf("%d\n", A[N-1][N-1][K-1]);
13   for (int i = 0; i < N; i++)
14     free(A[i]);
15   free(A);
16   return EXIT_SUCCESS;
17 }
```

accept

```
 1 int mat3d() {
 2   A = (int *)malloc(4096);

 4   for (int i = 0; i < N; i++)
 5     for (int j = 0; j < N; j++)
 6       for (int k = 0; k < K; k++)
 7         A[64*i+j*4+k] = A[64*i+j*4+k] + 1;
 8   printf("%d\n", A[1023]);

 9   free(A);
10   return EXIT_SUCCESS;
11 }
```

Fig. 3: Refactoring which flattens a multi-dimensional matrix. If user accepts the refactoring suggestion, the three dimensional matrix is flattened to a one dimensional array and all memory allocation, access and memory deallocation sites are transformed as shown. Symbols `N` and `K` are compile time constants with values 16 and 4.

In a number of circumstances, it is possible to transform the code such that it no longer fails the compiler's test that determines if an optimization is applicable. Due to space constraints, we will only discuss one such transformation – marking a function *pure* so as to allow a loop to be parallelized even though its body contains a call to a function defined outside the current translation unit. Pure functions return a value which is calculated based on given parameters and global memory and do not change the values of global variables and memory.

We use the code shown in Fig. 4 as an example. It contains a loop that is not parallelized due to a function call. Function inlining would have allowed parallelization to proceed but inlining `max` is not done because only the function declaration, not the function definition, is seen when compiling a single translation unit at a time. Nor can it be determined that the function call is free from side effects for the same reason. The fact that `max` is pure should be obvious to the programmer upon inspection of the code and the build process (not shown).

The code comments assist the programmer as follows. Markers are shown in two places in the source code: at the call site and at the function declaration. Also, when hovering the marked code, a text in a pop-op window explains how to determine if a function is pure or not. Figure 5 shows the appearance of code comments. Finally, a refactoring is suggested at the declaration site. It applies an annotation to inform the compiler that all function implementations of the declaration are guaranteed to be pure. The refactoring eliminates the need for the programmer to be familiar with the compiler specific syntax to declare functions as pure. When the refactored code is compiled again, the loop will get parallelized.

Link-time optimization could also resolve the issue by making the `max` function definition available at optimization time. Changing the build system to do link-time optimization, however, is non-trivial with complex, cross-platform build- systems.

```
proc.c
 1 #include "util.h"

42 for (j = 0; j < N; j++) {
43   for (i = 0; i < N; i++) {
44     x[i] = max(x[i], 42.0);
45   }
46   sum += x[j];
47 }
```

```
util.c
1 double
2 max(double a, double b) {
3   return a > b ? a : b;
4 }
```

```
util.h
1 double
2 max(double a, double b);
3
```

accept

```
util.h
1 double
2 max(double a, double b)
3 __attribute__((pure));
```

Fig. 4: Suggestion to the programmer to consider if the declaration of a function can be marked as pure since automatic loop parallelization would otherwise not process the loop in `proc.c`.

4 Preliminary Results

To measure the effect of using our tool-set, we generated code comments for the edge detection application in the UTDSP benchmarking suite [9]. The program detects edges in an 8-bit grayscale image. The unmodified code uses a 128x128 image which we changed to 4096x4096 to increase running times well above the timing resolution.

The program consists of the `main` function which calls `convolve2d` repeatedly with 3x3 Gaussian and Sobel kernels to do edge detection. The `main` method contains two loop nests but the bulk of the computation takes place in `convolve2d`'s second loop nest. During compilation, `gcc` can parallelize the loop nests in the `main` method but not the work intensive loop in `convolve2d`.

A code comment is shown in the Eclipse editor on the source line containing the code which the auto parallelization optimization can not analyze without programmer assistance. Figure 5 shows the code comment as it appears in the Eclipse editor. As can be seen from the comment, the problem is aliasing between the arrays `kernel`, `output_image` and `input _image`. The tool currently generates three comments reporting an aliasing problem between a pair of memory references, but such sets of comments could be merged in to a single one for clarity.

Any programmer with a basic understanding of the semantics of the arrays in the edge detection code knows that these can never point to the same memory. To convey this information to the compiler, the `restrict` keyword can be used. It was introduced in the C99 standard but `gcc` also accepts `__restrict__` when compiling in C89 mode. The fact that only pointers can be qualified with `restrict` complicates the situation. Thus, to use the `restrict` qualifier, the declaration of the arrays, which are parameters to the `convolve2d` function, must be changed to use pointers. The function signature was therefore changed as shown in Table 1.

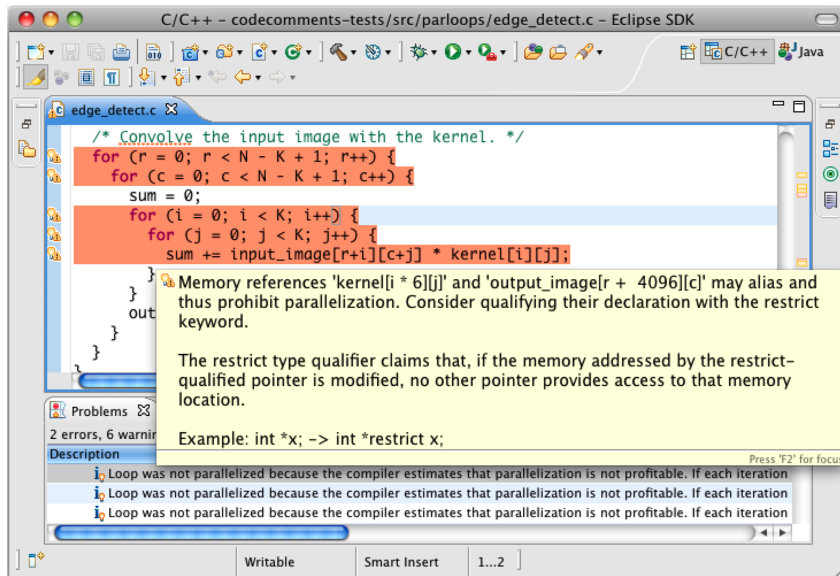

Fig. 5: Code comments generated with our tool shown in the Eclipse editor. Lines with comments are highlighted with an orange background and with small lightbulbs in the gutter area. By hovering the mouse on a source line with a comment, an overlay shows an explanatory message and a list of refactoring suggestions (if any). The problem view in the botton shows code comments in addition to regular warnings and errors.

4.1 Experimental Method and Setup

We measured the speedup enabled by modifying the edge detection application to allow automatic parallelization of the most computationally intensive loop nest.

Two different machines were used to measure the speedup. The first was a dual-socket server equipped with two quad-core 2.93 GHz Intel Xeon 5570 CPUs and a total of 12 GB DDR3 RAM. It contained eight cores each of which supports two hardware threads. It had 256 KB L2 cache per core and 8 MB shared L3 cache per CPU. The operating system was Linux using the 2.6.30 kernel patched to support hardware performance counters.

The second was an IBM JS22 (7998-61X) blade with a quad core 4.0 GHz POWER6 SCM processor. Each processing core supports two hardware threads. The machine includes 8 GB DDR2 SDRAM, 64 KB I-cache and 64 KB D-cache L1 cache per core, and 4 MB L2 cache per core. The operating system kernel was Linux 2.6.27 for PowerPC.

On the Intel platform, `gcc` version 4.5.1 was used, which at the time of writing is the latest production release. For the IBM platform we used `gcc` 4.6 experimental. The `-O2` compilation flag was used for optimization for two

Table 1: Original (left) and modified (right) function signature for the `convolve2d` function. The modifications were done based on code comments and allows auto parallelization of an important loop nest.

```
void convolve2d(             void convolve2d(
int input_image[N][N],       int (*__restrict__ input_image) [N],
int kernel[K][K],            int (*__restrict__ kernel)[K],
int output_image[N][N])      int (*__restrict__ output_image) [N])
```

reasons. First, auto-vectorization, which is enabled at optimization level `-O3` may interfere with auto-parallelization and secondly the auto parallelization does not succeed at `-O3` for reasons which we have yet to investigate. Measurements were made for 2-16 threads. No auto parallelization is done when less than two threads in `gcc`. We also measured the difference in sequential performance between the original and modified edge detect application and found it to be zero.

To measure the average execution time of each program version, three warmup runs and ten benchmark runs were performed on an otherwise idle server. The edge detection program spends most of the execution time loading the image from disk and saving the output image due to the simple but inefficient file format and IO routines. Consequently, the time spent on IO was excluded from our measurements.

4.2 Results on Intel Xeon Platform

We measured the speedup relative to sequential execution when `gcc` parallelized the original edge detection code. The edge detection code was subsequently modified based on the code comments from our tool as described in the previous section. We also measured the speedup when `gcc` parallelized the modified code relative to sequential execution and relative to execution of the unmodified, auto-parallelized code. The speedups on the Intel based server are summarized in Fig. 6.

When auto parallelizing the unmodified edge detection code, a small speedup is obtained when using 2 to 8 threads with 4 threads providing the best result. Using 12 or 16 threads, however, results in a as much as a 28% slowdown.

When auto parallelizing the code with modification based on the code comments, all three loop nests are transformed by `gcc`. Once again, the biggest gain is not realized with the highest number of threads. Using eight threads we observed a factor 5 speedup over the sequential version and a factor 4.8 over the original edge detection code when auto-parallelized. Super-linear speedups was observed over both the sequential and original, auto parallelized code when using 2-4 threads. We believe that this effect is due to the cache system. By adding cores, we effectively reduce the working set of each core thus allowing it to hold a larger fraction of its working set in the L1 data cache. Thus, the application benefits not only from the added computational resources but also from the additional cache capacity. We have not yet had the time to verify this hypothesis by measuring the cache miss rates with hardware performance counters.

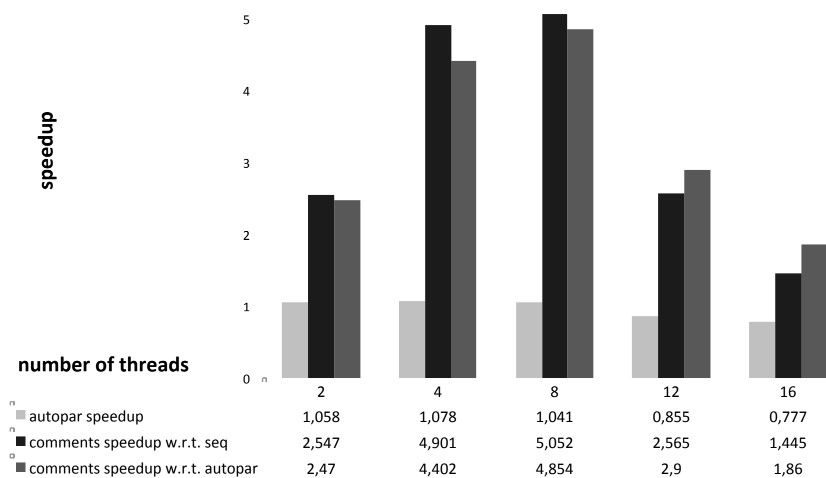

Fig. 6: Speedups on Intel Xeon platform. In the version modified on the basis of comments, all three loop nests in the program are parallelized by `gcc`, in the unmodified version, only the two loop nests in `main` are parallelized.

4.3 Results on POWER6 Platform

We also ran our experiments on an IBM POWER6 based server. The speedups that was observed on this machine are summarized in Fig. 7. When running the unmodified, auto parallelized edge detection code, we observed a slowdown of approximately 6-7% on 2-8 threads.

When auto parallelizing the code with modification based on the code comments, however, we observe speedups ranging from 3.5 to 8.6 over the sequential version and 3.7 to 9.2 relative to the original, auto-parallelized version. Again, super linear speedups are observed – this time for all thread counts. This strengthens our belief that the application benefits from the added cache capacity as well as from the additional cores. As on the Intel platform, we have not investigated this.

5 Related Work

ReLooper is an Eclipse plug-in that can help the programmer parallelize regular loop nests in Java code [2]. Parallelization is done using the `ParallelArray` framework and not OpenMP. The former is limited to handle fewer kinds of parallel loops. Like our tool, ReLooper also relies on static data-dependence analysis to detect parallelism – but unlike in `gcc` the data-dependence analysis is inter-procedural.

ReLooper puts the programmer in the loop in the sense that she picks a "target array" and ReLooper then analyses the loops which access the array and reports if they can be parallelized safely. The programmer can then choose

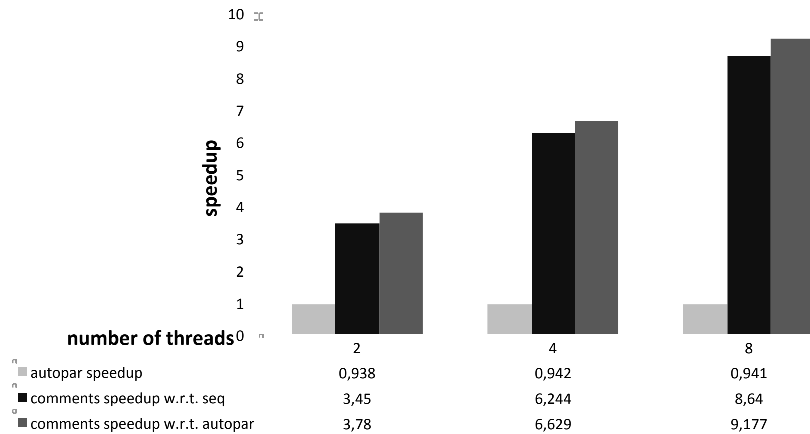

Fig. 7: Speedups on POWER6 platform. In the version modified on the basis of comments, all three loop nests in the program are parallelized by `gcc`, in the unmodified version, only the two loop nests in `main` are parallelized.

to parallelize unsafe loops as is or make changes before re-running the analysis. This mode of interaction differs from our tool since we may also give refactoring suggestions that remove obstacles to optimization.

Sean Rul et al. proposed the Paralax infrastructure which also exploits programmer knowledge for optimization [16]. Paralax is comprised of tree parts. The first is a compiler for automatic parallelization of outer loops containing coarse- grain pipeline-style parallelism. The second is a set of annotations which primarily supply data-dependency information not inferred by static analysis in the compiler and which are verified dynamically. The final part is a tool which suggests how the programmer may add annotations to the program.

The Paralax approach is complimentary to ours. It parallelizes irregular, pointer- intensive codes whereas we are primarily concerned with regular codes that are amenable to parallelization, vectorization and locality enhancement with slight modifications. Also, the suggestions generated by the Paralax tool rely on both static analysis and profiling information whereas our suggestions, so far, do not require the programmer to perform program profiling.

FASThread [10] is a commercial tool from Nema Labs that helps the programmer parallelize code by identifying issues which prevent parallelization. It also shows "generic" examples of workarounds to the programmer and integrates with the IDE. Our knowledge of the tool is limited as the tool is in closed beta-testing. The core approach seems similar to the part of our work which address auto- parallelization. However, the approach presented here includes automated refactoring of certain optimization issues.

Suggestions for locality optimizations, SLO, provides refactoring suggestions at the source level aiming to reduce reuse distances and thus the number of cache misses [1]. The suggestions are based on cache profiling runs and are

complimentary to the types of refactoring offered by our tool. For instance, SLO does not help the programmer expose parallelism in the source code.

Xlanguage [3] is a pragma language for C which allows automatic exploration of program optimizations. The programmer inserts pragmas, or directives, to indicate where transformations such as loop unrolling, loop interchange and tiling are legal. Multiple transformations may be legal for each loop nest. The best combination of transformations for a particular machine can then be found automatically. With this approach, the programmer explicitly directs the source transformation, whereas we supply semantic information but let the compiler decide if an optimization is legal. A combination of the two approaches may be possible. Code comments could help the programmer write code such that the compiler can determine where xlanguage pragmas are safe to insert.

The vectorization pass in `gcc` already offers an option to generate compilation reports. However, since problem descriptions are very terse and often reference compiler generated intermediaries rather than source code variables, we believe these reports are of more use to compiler writers than programmers.

6 Conclusions

This paper presents the first iteration of our code comments and refactoring tool. It increases the level of integration between the IDE and the compiler to aid the programmer in optimizing software for embedded, multi-core architectures.

It offers two kinds of feedback to the programmer. First it allows optimizations to be applied to the source level. This is beneficial in scenarios where the final code must be compiled with optimizations disabled to ensure accurate debugging information and it allows optimizations to be carried by the source code from one compiler to another.

Second, we bring attention to problems which obstruct particular optimizations and a particular optimization to succeed and optionally suggest workarounds in the form of refactoring.

The preliminary experiments on an edge detection application shows that without the code comments, the application either runs slightly faster or slower depending on the platform and number of threads. When modifying the code to allow an important loop nest to be parallelized, however, both platforms show significant speedups from 5 to 8.6 relative to the sequential code and from 4.8 to 9.1 over the unmodified, auto- parallelized code.

The tools are work in progress and much work remain. Our primary goals are to expand the types of analysis supported and increase the number of refactoring options for situations where an optimization fails for reasons which may be corrected by the programmer. Also, any codebase is likely to contain a lot of code which is essentially not amenable to optimization. Consequently, not all code comments represent true optimization opportunities. To help the programmer focus on the most promising code comments, we believe it would be helpful to prioritize code comments which relate to code which is frequently executed and also code comments which offer automated refactoring.

Finally, one could apply the refactoring suggestions speculatively to determine which ones actually enable additional optimization and use that information as an additional ranking criteria. When the prototype implementation matures, we plan to make our tool-set publicly available as free and open source software.

Acknowledgments This work was done while the first author was on HiPEAC internship at IBM Haifa. The authors thank Gad Haber at IBM Haifa whose efforts have greatly contributed to this work. The research made use of the University of Toronto DSP Benchmark Suite, UTDSP. Finally, the authors are grateful for the clear and plentiful constructive feedback provided by an anonymous reviewer.

References

1. K. Beyls and E. D'Hollander. Refactoring for data locality. *IEEE Computer*, 42(2):62–71, 2 2009.
2. D. Dig et al. ReLooper: Refactoring for loop parallelism. Technical report, University of Illinois Urbana-Champaign, 2009. https://netfiles.uiuc.edu/dig/papers/ReLooper.pdf.
3. S. Donadio et al. A language for the compact representation of multiple program versions. In *Int. Workshop on LCPC*. Springer-Verlag, 2005.
4. Free Software Foundation. GNU Compiler Collection. http://gnu.gcc.org. Date accessed: September 11th 2010.
5. Free Software Foundation. Wiki page on auto-parallelization capabilities in GCC. http://gcc.gnu.org/wiki/Graphite/Parallelization. Date accessed: September 14th 2010.
6. M. J. Garzaran et al. Program optimization through loop vectorization. http://sc10.supercomputing.org/schedule/event_detail.php?evid=tut140, 2010. Date accessed: December 19th 2010.
7. O. Golovanevsky and A. Zaks. Struct-reorg: Current status and future perspectives. In *Proc. of GCC Developer's Summit*, 2007.
8. R. Ladelsky. Matrix flattening and transposing in GCC. In *Proc. of GCC Developer's Summit*, 2006.
9. C. Lee et al. UTDSP benchmark suite. http://www.eecg.toronto.edu/corinna/DSP/infrastructure/UTDSP.html, 1998. Date accessed: July 4th 2009.
10. Nema Labs. FASThread product page. http://www.nemalabs.com/?q=node/7. Date accessed: October 18th 2010.
11. D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for SIMD. In *Proc. of PLDI*, 2006.
12. D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short SIMD architectures. In *Proc. of PACT*, 2008.
13. OpenMP Architecture Review Board. OpenMP application program interface, version 3.0. Technical report, 2008.
14. S. Pop et al. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proc. of GCC Developer's Summit*, 2006.
15. K. Trifunovic et al. Polyhedral-model guided loop-nest auto-vectorization. In *Proc. of PACT*, 2009.
16. H. Vandierendonck, S. Rul, and K. D. Bosschere. The paralax infrastructure: Automatic parallelization with a helping hand. In *Proc. of PACT*, 2010.