Automatic Loop Parallelization via Compiler Guided Refactoring

Larsen, Per; Ladelsky, Razya; Lidman, Jacob; McKee, Sally A.; Karlsson, Sven ; Zaks, Ayal

Publication date:
2011

Automatic Loop Parallelization via Compiler Guided Refactoring

Per Larsen*, Razya Ladelsky‡, Jacob Lidman†, Sally A. McKee†, Sven Karlsson* and Ayal Zaks‡
*DTU Informatics
Technical U. Denmark, 2800 Kgs. Lyngby, Denmark
Email: {pl,ska}@imm.dtu.dk
† Computer Science Engineering
Chalmers U. Technology, 412 96 Gothenburg, Sweden
Email: lidman@student.chalmers.se, mckee@chalmers.se
‡IBM Haifa Research Labs Mount Carmel, Haifa, 31905, Israel
Email: {razya,zaks}@il.ibm.com

***Abstract*—For many parallel applications, performance relies not on instruction-level parallelism, but on loop-level parallelism. Unfortunately, many modern applications are written in ways that obstruct automatic loop parallelization. Since we cannot identify sufficient parallelization opportunities for these codes in a static, off-line compiler, we developed an interactive compilation feedback system that guides the programmer in iteratively modifying application source, thereby improving the compiler's ability to generate loop-parallel code. We use this compilation system to modify two sequential benchmarks, finding that the code parallelized in this way runs up to 8.3 times faster on an octo-core Intel Xeon 5570 system and up to 12.5 times faster on a quad-core IBM POWER6 system.**

Benchmark performance varies significantly between the systems. This suggests that semi-automatic parallelization should be combined with target-specific optimizations. Furthermore, comparing the first benchmark to hand-parallelized, hand-optimized pthreads and OpenMP versions, we find that code generated using our approach typically outperforms the pthreads code (within 93-339%). It also performs competitively against the OpenMP code (within 75-111%). The second benchmark outperforms hand-parallelized and optimized OpenMP code (within 109-242%).

I. Introduction

Parallel programming is the biggest challenge faced by the computing industry today. Yet, applications are often written in ways that prevent automatic parallelization by compilers. Opportunities for optimization are therefore overlooked. A recent study of the production compilers from Intel and IBM found that 51 out of 134 loops were vectorized by one compiler but not the other [10].

However, most optimizing compilers can generate reports designed to give a general idea of the issues encountered during compilation. The task to determine the particular source code construct that prevents optimization is left to the programmer.

We extended a production compiler to produce an interactive compilation system. Feedback is generated during compilation. It guides the programmer to refactor the source code. This process leads to code that is amenable to auto-parallelization.

This paper, evaluates the speedups after applying our compilation system on two benchmarks. We compare the speedups with hand-parallelized and optimized versions using an octo-core Intel Xeon 5570 system and a quad-core IBM POWER6 SCM system.

Our contributions are as follows.

- First, we present our interactive compilation system.
- Second, we perform an extensive performance evaluation. We use two benchmark kernels, two parallel architectures and also study the behavior of the benchmarks.

After modification with our compilation system, we find that the two benchmarks runs up to 6.0 and 8.3 times faster on the Intel Xeon 5570 system. On the IBM POWER6 system they run up to 3.1 and 12.5 times faster respectively.

We compared the benchmarks to hand-parallelized, hand-optimized POSIX threads [5], pthreads, and OpenMP [7] versions. For the first benchmark, we found that code generated using our approach delivers up to 339% of the performance of the pthreads version. It also performs competitively against the OpenMP code (up to 111%). The second benchmark delivered up to 242% of the performance of the OpenMP version.

The following section introduces two benchmarks which exemplify the problems faced by production compilers. The problems observed via the benchmarks are then related to the steps in the auto-parallelization process in Section III. Section IV describes the interactive compilation system which guides code refactoring to facilitate auto-parallelization. The refactoring which enabled auto-parallelization of the two benchmarks are discussed in Sections V and VI respectively. Experimental results are presented in Sections VII and Section VIII surveys related work. Section IX concludes.

II. Exposing Compiler Problems

Programs can be written in many ways. Some ways obstruct automatic loop parallelization. Throughout the paper, two benchmark kernels will exemplify the major problems production compilers face. We will start by briefly outlining the kernels.

The first kernel is a realistic image processing benchmark. It was kindly provided by STMicroelectronics Ottawa and Polytechnique Montreal [4]. The kernel interpolates sensor

TABLE I
LOOP NESTS IN THE EDGE DETECTION KERNEL THAT WERE AUTO-PARALLELIZED BY FIVE DIFFERENT COMPILERS. THE COMPILERS ARE UNLIKELY TO PARALLELIZE THE SECOND OR THIRD LOOP NEST.

Origin	Compiler	Loop nests		
		loop1	*loop2*	*loop3*
FOSS	`gcc`	✓		✓
Intel	`icc`	✓	(✓)	✓
FOSS	`opencc`	✓		✓
PGI	`pgcc`	✓		
Oracle	`suncc`	✓	✓	

data from a color filter mosaic [17] in a process called *demosaicing*. Its execution time is concentrated in twelve loop nests whose iterations are independent. In addition to the sequential code, we received a hand-parallelized and optimized `pthreads` version.

We also studied the edge detection kernel in the UTDSP benchmarks [16]. This program detects edges in a grayscale image and contains three loop nests that may be parallelized. A single loop nest accounts for the majority of the execution time.

Both kernels are difficult to parallelize automatically. We tested with the `gcc` [9] and `opencc` [6] open-source compilers and `icc` from Intel [12], `pgcc` [22] from Portland Group and `suncc` [19] from Oracle. The highest available optimization levels and inter-procedural optimization were selected to produce the best results. Level `-O2` were tried in addition to `-O3` with `gcc` since that may produce better results.

None of the loop nests in the demosaicing code were parallelized by any of the compilers. The results for the edge detection code are shown in Table I. Only `icc` managed to parallelize the second loop nest where parameter-aliasing is an issue. Interestingly, `icc` succeeds because it makes an inlining decision which rules out aliasing among function parameters. If the function is not inlined, aliasing also prevents `icc` from parallelizing the loop.

The obstacles which prevents auto-parallelization of the benchmark kernels are:

- Function parameters that may point to the same memory locations.
- Function calls in loop bodies. These may have side effects.
- Loop counters that may overflow or lead to out-of-bound array accesses.
- Loop bounds that can not be analyzed by the compiler.
- Array access patterns that are too complex for the compiler to analyze.
- Loops that may contain insufficient work for parallelization to be profitable.

To understand why a compiler may refrain from auto-parallelizing the benchmarks, a single production compiler was studied. We based our work on the widely used, open-source `gcc` compiler which is being rigorously tested and enhanced by a vibrant and pragmatic developer community.

III. AUTOMATIC PARALLELIZATION WITH `gcc`

Automatic parallelization [24] involves numerous analysis steps. Every optimizing compiler must perform similar steps. The concrete implementations may vary and this leads to different strengths and weaknesses among compilers. This section explains where and why the analysis steps in `gcc` release 4.5.1 had problems parallelizing the benchmarks.

A. Alias Analysis

Alias analysis determines which storage locations may be accessible in more than one way [11]. Aliasing of pointers and function parameters may create dependencies among loop iterations so this analysis is instrumental to auto-parallelization.

The alias analysis implemented in `gcc` is a fast variant. It does not account for the context in which function calls are made nor does it take the flow-control in functions into account. Hence, function parameters of array and pointer types are conservatively assumed to alias. Also, if a statement aliases two pointers at some point in a function, the pointers are assumed to alias not just in the following statements but at all statements in the function. Both types of alias-analysis inaccuracies prevented auto-parallelization of the benchmark kernels studied. Our interactive compilation system can point to array accesses which are assumed to alias and suggest how to remove this assumption.

B. Number of Iterations Analysis

Loop iterations must be countable for auto-parallelization to proceed. The compiler must therefore analyze the upper and lower bounds and loop increments. If these values are not constant, it must discover which variables hold the values at runtime.

The demosaicing kernel contained several loops where the loop counter is incremented by two in each iteration. This prevented `gcc` from computing the number of iterations and was reported by our interactive compilation system. The loop increment can be *normalized* – or changed to one – by multiplying all uses of the loop counter with the original increment. *Gcc* id not normalize the loops in the demosaicing benchmark so manual refactoring was required.

C. Induction Variable Analysis

Induction variables are the loop counters and scalars which are affine functions of the loop counter variables. Auto-parallelization works by distributing loop iterations among threads. To do so, each thread must be have private copies of the induction variables. Threads must also have private copies of reduction variables.

In the demosaicing benchmark, `gcc`was either unable to determine the variables that serve as induction variables or how the values of induction variables change in successive loop iterations. This prevented auto-parallelization. The problems were mostly due to the use of a one dimensional array to represent two dimensional data. This programming pattern is known as *de-linearization* and is common to C programs.

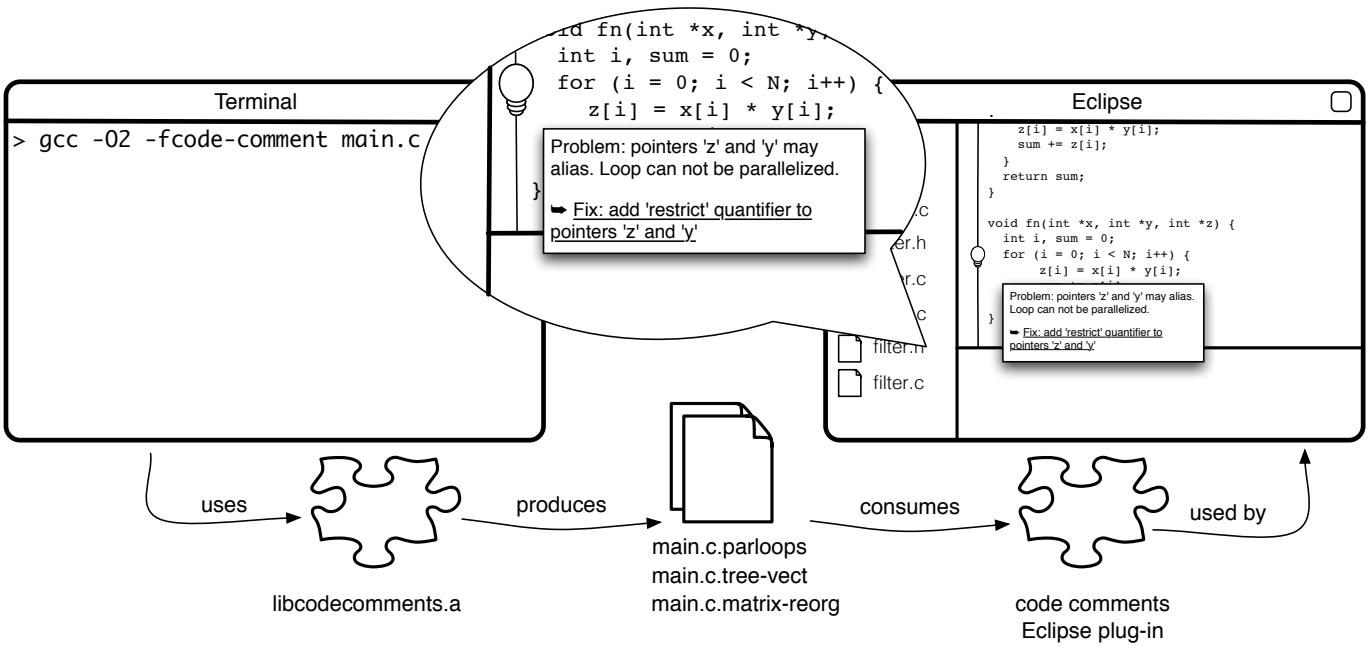


Fig. 1. Illustration of our compilation feedback system. A library extends `gcc` to generate code comments in its diagnostic dump files. A plug-in for the Eclipse CDT environment provides Eclipse with the functionality to i) read the code comments containing feedback ii) display them at appropriate places in the source code and iii) provide refactoring support for the changes suggested by the compiler feedback.

It effectively complicates indexing expressions. Using two-dimensional arrays or pointers to pointers to represent two dimensional data simplifies the indexing expressions.

D. Data Dependence Analysis

Currently, `gcc` contains two different frameworks to analyze data dependencies. The *lambda* framework is the oldest and most mature of the two. It represents data dependences as distance vectors and implements the classical data dependence tests [2]. Much functionality, including the lambda framework is shared between the loop parallelization and vectorization optimizations. Hence, code transformations which enable `gcc` to parallelize a loop using the lambda framework will also help making it vectorizable.

`Gcc` is transitioning to GRAPHITE which is a newer and more capable data dependence framework [20]. The transition is advancing at slow but steady pace and much work remains. Even in the 4.6 release of `gcc`, auto-parallelization with GRAPHITE is only be able to handle innermost loops. Hence, the lambda framework was used in our experiments.

The goal of data dependence analysis is to determine if a loop iteration may depend on on data written in a previous loop iteration. In such cases auto-parallelization is usually prohibited. Often, data is read and written from arrays or pointers and this may lead to dependencies between loop iterations. Each iteration is identified by a vector in the loop iteration space. Subtracting the iteration vectors of two dependent iterations yields a distance vector. The lambda dependence analysis framework requires that all possibly dependent loop iterations have the same distance vector. If is not the case, auto-parallelization fails. The failure happens because the structure data accesses through subscripted variables is too complex to be modeled, even if the iterations of the loop nest are in fact independent. Some loop nests in the demosaicing benchmark was not auto-parallelized for this reason.

IV. INTERACTIVE COMPILATION FEEDBACK SYSTEM

The benchmark kernels exposed several coding patterns which obstructs auto-parallelization in production compilers. The previous section then used `gcc` to exemplify why a compiler may be prevented from auto-parallelizing a loop nest. We can now explain how we provide feedback and suggestions on how the programmer can make code amenable to auto-parallelization.

Our interactive compilation feedback system is illustrated in Fig. 1. It has two parts. The first part is a library, `libcodecomments`, and a set of patches to `gcc`'s auto-parallelization subsystems. This extension of `gcc` generates code comments containing compiler feedback. In contrast to stand-alone tools, the code comments leverage the production-quality program analysis and optimization functionality already present in the compiler.

The code comments are generated when one of the steps in the auto-parallelization optimization encounters an issue which prevents further analysis. The functionality in `libcodecomments` is then used to produce a human understandable problem description. This is important because program analysis often fail while processing compiler generated, temporary variables that are meaningless to the programmer. Most importantly, `libcodecomments` is used to reconstruct source level expressions (after preprocessing) and their file locations from compiler generated temporaries.

The generation of diagnostic dump files are controlled via existing compiler flags – our code comments are simply added to these dump files.

The second part of our system is a plug-in for the Eclipse C Development Tools, CDT [21]. The code comments plug-in enables CDT to parse the compiler feedback from dump files. The dump files are read by a custom step in the Eclipse build process and requires no programmer intervention besides adding the appropriate compiler flags. The raw code comments are subsequently 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 associated with the code comment. The comment may include a *quick fix* – i.e. a refactoring that automates the suggested transformation. For example, lines may be added or deleted around the construct. The comment in 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.

Not all the code comments which can be generated by our modified compiler contain concrete advice on how to resolve a given issue. The types of feedback currently available to a non-compiler expert are the following: aliasing comments, comments on function calls which prevent parallelization due to possible side-effects and comments on data-dependences among memory accesses. We consider these comments sufficient to address the most important compilation issues – those which are the least likely to be resolved in future releases of the `gcc` compiler.

V. Case Study: Demosaicing

Recall, `gcc` and the other compilers tested failed to parallelize any of the 12 original loop nests in the demosaicing kernel. Our compilation feedback system, however, succeeded in removing the issues preventing parallelization. It was accomplished by iteratively modifying and compiling the code until all relevant loop nests were auto-parallelized. The following sections describe how we refactored the code to accomplished this.

A. Loop Iteration Counts

Most of the loops in the demosaicing code have a stride of two. This caused `gcc`'s iteration count analysis to fail according to the compiler feedback. As a workaround, the loops were normalized to use unit strides and array indexing expressions were updated accordingly. For instance

```
int x, y, idx;
for(x=2+offset_red;x<H-2;x+=2) {
 for(y=2+offset_blue;y<W-2;y+=2) {
  idx=x*W+y; ... }}
```

was rewritten as:

```
unsigned int x, y, idx;
 for(x=1;x<(H-2)/2;x++) {
  for(y=1;y<(W-2)/2;y++) {
   idx=(2*x+offset_red)*W+2*y+offset_blue;
   ... }}
```

Additionally, the type of the loop counters, `x` and `y` were changed from signed to unsigned integers. Finally, we observed that writing the loop upper bound as `H/2-1` rather than `(H-2)/2` also caused number of iterations analysis to fail. A more powerful analysis can surely digest both variants properly.

B. Aliasing

As mentioned in section III-A, `gcc` employs a fast but imprecise alias analysis. Most importantly, the analysis does not analyze how function arguments are passed from callers to callees, which means that if a function contains several arguments having pointer or array types, `gcc` must assume they may alias. This assumption is made even for parameters of incompatible types, due to the weak type discipline employed in C. Its possible to change the assumption that pointers to objects of different types alias with the `-fstrict-aliasing` flag. In our experiments it eliminated potential aliasing in two out of the twelve loop nests.

When potential aliasing prevents parallelization, a code comment contains feedback on the data references that are potential aliases. The code comment as it appears in the IDE is shown in figure 2. The comment suggests that the problem can be resolved by annotating the relevant pointers with the `restrict` keyword. This type qualifier was added in the latest revision of the language standard [14]. It also includes an option to automatically transform the code such that the `restrict` type qualifier is added to the relevant pointers declarations. Semantically, if memory addressed by a `restrict` qualified pointer is modified, no other pointer provides access to that memory. It is left to the programmer to determine if the `restrict` qualifier can be added. Based on the suggestions provided by the code comments, we added `restrict` qualifiers to 6 pointer typed formal parameters in two function signatures.

A more precise, inter-procedural alias analysis is also available in `gcc`. It is enabled by the `-fipa-pta` flag. Contrary to our expectations, the inter-procedural alias analysis did not diminish the need to `restrict`-qualify function parameters.

C. Induction Variables

Normalizing loop strides to one and simplifying the expressions governing loop bounds as described in section V-A in effect complicated the expressions for the induction variables. For instance, an induction variable that was previously computed as `idx=x*W+y` became `idx = (2*x+offset_red)*W+ 2*y+offset_blue`. However, the compilation feedback helped us understand how to refactor the loops so that induction variable analysis did not prevent auto-parallelization. The original demosaicing kernel uses linearized arrays to represent variable size, two-dimensional image data. The arrays are passed into the three kernels as function parameters. Changing the types of these function parameters allowed us to cast the

linearized arrays as two-dimensional arrays. This in turn allowed a simplification of the indexing expressions. By changing a parameter `int *restrict red_array` to `int (*restrict red_array)[W]` where `W` is scalar holding the image width, we changed the indexing expressions from

```
idx = (2*x+offset_red)*W+2*y+offset_blue;
red_array[idx]= RBK_3x3_1(
 red_array[idx-W-1], red_array[idx-W+1],...);
```

to

```
red_array[2*x+offset_red][2*y+offset_blue] =
 RBK_3x3_1(
  red_array[2*x+offset_red-1][2*y+offset_blue-1],
  red_array[2*x+offset_red-1][2*y+offset_blue+1],
 ...);
```

De-linearizing the array accesses arguably increased the readability of the code.

It was also necessary to move the loop-invariant variables `offset_red` and `offset_blue` out of the loop. This was accomplished by introducing a temporary, `restrict`-qualified pointer defined as `tmp_red_array= red_array+W*offset_red+offset_blue`

Finally, due to an analysis limitation when computing the scalar evolution of expressions containing integers of different sizes we had to suffix the integer literals with `L`'s since we used a 64-bit build environment. Continuing the code example, we arrived at:

```
int (*restrict tmp_red_array)[W]=
 red_array + W*offset_red + offset_blue;
for(x=1;x<(H-2)/2;x++) {
 for(y=1;y<(W-2)/2;y++) {
  tmp_red_array[2L*x][2L*y] =
   RBK_3x3_1(
    tmp_red_array[2L*x-1L][2L*y-1L],
    tmp_red_array[2L*x-1L][2L*y+1L], ...
```

D. Data Dependences

After transforming the code to allow all preceding analysis steps to succeed, `gcc` was able to perform data dependence analysis on the loop nests. Although iterations of all loop nests in the benchmark are independent, eight of these loop nests update elements in place to reduce memory requirements. The in-place updates are possible when a loop nest writes only "odd" elements and reads only "even" elements or vice versa. From the code comments, however, we could determine that the data dependence analysis failed to discover this. For instance, a possible data dependence was reported between the references

```
tmp_blue_array[x*2+1][y*2-1]
```

and

```
tmp_blue_array[x*2-1][y*2+1]
```

Data dependence analysis fails for this pair of references, because the lambda framework can not compute a distance vector which represents their dependence relation. A possible dependence between these two references must therefore be assumed in lieu of a more precise data dependence analysis.

To avoid reading and writing to the same array – the memory addressed by `tmp_blue_array` in the example – a new temporary array was allocated to hold the writes. This effectively sidesteps a compilers inability to analyze non-overlapping accesses to the same array. However, it also means that updates are no longer done in-place which decreases the spatial locality of the kernel and increases the memory consumption by approximately 8%. Finally, for each of the eight loop nest with in-place updates, a simple "copy" loop was added to write data from the new temporary array back to its original destination. These loops were fairly easy to add and were readily parallelized due to their simplicity.

An alternative solution exists: the programmer could have introduced additional `restrict`-qualified pointers until all potential data dependencies are ruled out. This solution does affects neither the data access pattern nor the memory consumption so performance would be unaffected. This shows that the programmer may need to chose among several alternative ways to refactor – each having a different performance impact. For our experiments, we pessimistically assume that the programmer choose refactoring that is most costly in terms of performance.

E. POSIX *Threads Version*

We optimized the pthreads code received from STMicroelectronics to execute all relevant loops in parallel and to minimize synchronization and management overhead. The parallelization strategy of the `pthreads` version differs from the auto-parallelized version. Two of the twelve loop nests in the sequential code were fused.

The distribution of iterations among threads also differ. The auto-parallelized version only distributes iterations of the outer loops among threads. The `pthreads` version, however, divides the two-dimensional picture into a number of tiles and assigns each tile to a single thread thereby increasing cache affinity.

Finally the `pthreads` version exploits the task-level parallelism that exists among the eight computationally intensive loop nests. It does so by executing them in pairs of two. The auto-parallelized version executes all loop nests one after another so it only exploits data-level parallelism.

F. OpenMP Version

Temporary arrays and extra loop nests were introduced in the auto-parallelized version to work around limitations in `gcc`'s data dependence analysis. Auto-parallelization also uses the combined work-sharing construct `omp parallel for` in OpenMP whereas an expert performance-programmer may enclose several loop nests with `omp for` directives in a single `omp parallel` region to reduce synchronization among threads.

To measure the resulting performance if the above mentioned deficiencies were removed, we hand-parallelized the

demosaicing code with OpenMP pragmas. Like the auto-parallelized version, the OpenMP version only exploits data-parallelism but performs updates of the arrays in-place instead of using temporary arrays.

Furthermore, we minimized the entries and exists to and from parallel regions. It was done by using separate `omp parallel` and `omp for` directives in place of the `omp parallel for` directive. This reduced the number of times a parallel section was entered from twelve to five. Using the `nowait` clause on the `omp for` directives finally allowed us to remove three implicit barriers in total.

VI. Case Study: Edge Detection

The program consists of a `main` function which calls the function `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`. The problem is aliasing between three arrays which are passed as parameters to the `convolve2d` function. Feedback from the compilation system reported an aliasing problem between pairs of data references and is illustrated in Figure 2.

A programmer who understands the roles of the pointers in the edge detection code knows that these will never point to the same memory. As with the demosaicing code, the lack of aliasing between the function parameters must be communicated using the `restrict` keyword and again `gcc`'s inter-procedural alias analysis did not help. The fact that only pointers can be qualified with `restrict` complicates the situation. Before the `restrict`-qualifier can be used, the parameters to the `convolve2d` function, must be changed as shown below:

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

to

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

The edge detection benchmark was subsequently parallelized by `gcc` without further problems.

A. OpenMP Version

To compare the auto-parallelized edge-detection code with a hand-parallelized and optimized version, we inserted OpenMP directives in the sequential code. Similar to demosaicing, separate `omp parallel` and `omp for` directives were used to increase the performance.

VII. Experimental Results

We ran measurements on the two benchmarks. The differences in sequential performance between the original and modified versions were measured and found to be negligible in both cases. The reference input for the edge detect benchmark is an image with 128x128 pixels which we scaled to 4096x4096 to increase running times well above the timing resolution. A large and a small color image was used as input to the demosaicing kernel. The large image had 5616x3744, 24-bit pixels while the small image consisted of 768x512, 24-bit pixels. The inputs are summarized in Table II.

TABLE II
Characteristics of the benchmark inputs.

Benchmark	Large input	Small input
Demosaicing	5616x3744, 24-bit color	768x512, 24-bit color
Edge Detection	4096x4096, 8-bit grayscale	-

TABLE III
Characteristics of the processing elements and memory hierarchies in the systems used for benchmarking.

Sys.	#Procs.	#Cores	#Threads	Freq.
Intel	2	8	16	2.93 GHz
IBM	2	4	8	4.0 GHz
Sys.	L1D	L2/Core	L3/Core	DRAM
Intel	32 KB	256 KB	2 MB	12 GB DDR3
IBM	64 KB	4 MB		8 GB DDR2

Two different systems were used to evaluate the impact of our modifications to the benchmarks. The Intel system was a dual-socket server equipped with two quad-core 2.93 GHz Xeon 5570 CPUs and a total of 12 GB DDR3 RAM. It contains eight cores each of which supports two hardware threads. It had 32 KB L1 instruction cache, 32 KB L1 data cache, 256 KB L2 cache per core and 8 MB shared L3 cache per CPU. The operating system was Linux using the 2.6.36 kernel. The IBM system was a JS22 (7998-61X) blade with two dual-core 4.0 GHz POWER6 SCM processors. Like the Intel Xeon system, each core supports two hardware threads. The system had 8 GB DDR2 RAM, 64 KB L1 instruction cache, 64 KB L1 data cache and 4 MB L2 cache per core. The operating system kernel was Linux 2.6.27. The characteristics of the processing units are summarized in Table III

Version 4.5.1 of `gcc` with our modifications to generate compiler feedback was used for all experiments. The `-O2` compilation flag was used for optimization since the auto-parallelization does not always succeed at `-O3`. Measurements were made for 2-16 threads on the x86 platform and 2-8 on the POWER platform. Numbers were calculated as averages over three consecutive program executions on an unloaded system. The time spent on IO was excluded from the measurements.

A. Demosaicing Speedups for Intel Xeon

We measured the speedups of parallelizing the modified demosaicing code relative to the original, sequential code. We also measured the speedups of the hand written `pthreads` and OpenMP versions with respect to the sequential version. Two images were used as input for the demosaicing benchmark: a high resolution 21 mega-pixel image and a small image with 768x512 pixels. The speedups on the x86-64 platform are summarized in Fig. 3a and Fig. 3b.

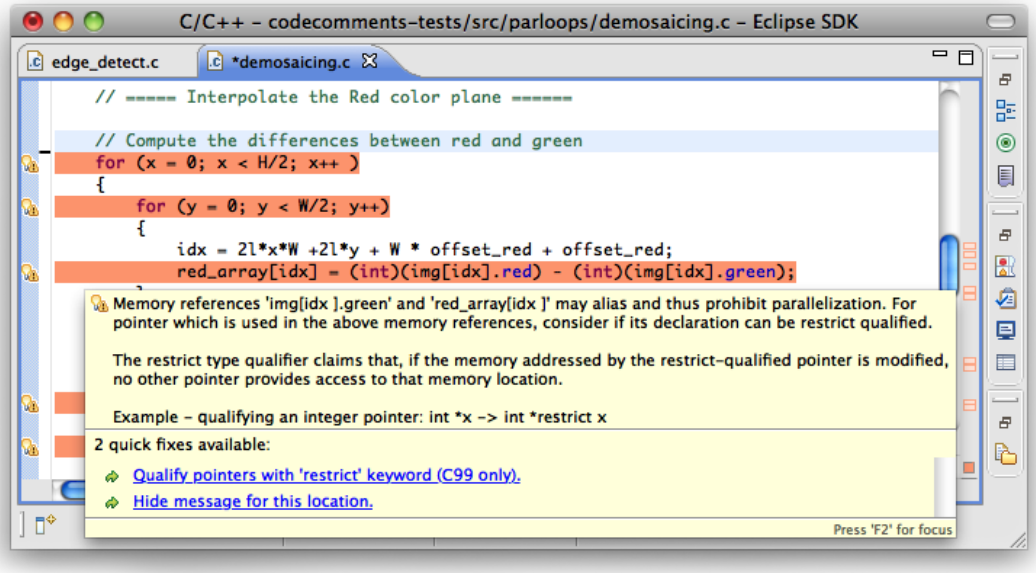


Fig. 2. A code comment generated by our compilation feedback system and shown in the Eclipse editor. Lines with comments are highlighted with an orange background color and with small lightbulbs in the gutter area. Placing the cursor on a source line with a comment will show an overlay with an explanatory message and a list of available refactoring suggestions (if any). The problem view in the bottom shows code comments in addition to regular warnings and errors.

When auto-parallelizing the unmodified code no loops are parallelized, and therefore no speedups are shown. When auto-parallelizing the modified code, the observed speedups range from 1.9 to 6 for 2-16 threads for the low resolution image. The highest speedup of 6.0 was obtained using 16 threads. A speedup of 5.6 is already obtained using 8 threads and the using 12 threads did not produce more than a 5.3 speedup over the sequential code. The meager increases after 8 threads are most likely explained by the fact that we are not adding more cache capacity past 8 threads since the system uses two-thread simultaneous multi-threading and has SMT-aware scheduling.

The high resolution image shows speedups ranging from 1.9 to 5.3 for 2-16 threads. The speedup using 8 threads, which is 5.1, is once again close to the maximal speedup of 5.3 on 16 threads. The weaker scaling when using a high resolution image may be explained by the fact that the effect of the temporary arrays become more pronounced once the working set sizes exceed the capacity of the last level caches.

Speedups for the OpenMP version range from 1.8 to 7.2 for 2-16 threads for the low resolution image and from 1.9 to 7.0 for the high resolution image. Performance increases steadily for 2,4,8 and 16 threads whereas there is little difference between 8 and 12 threads. Interestingly, the auto-parallelized version outperforms the OpenMP version by 8% on 2 threads and performs similarly with 4 threads. It is outperformed by 4% and 17% on 8 and 16 threads respectively.

The speedups obtained by the `pthreads` version were: 1.8-5.7 for the high resolution image and 1.3-3.2 for the low resolution image, executed by 2-16 threads. On the big image, the auto-parallelized code performed within 93-108% of the pthreads code – outperforming it on all but 16 threads. On the small image the auto-parallelized version outperformed the pthreads code by 44%-239%.

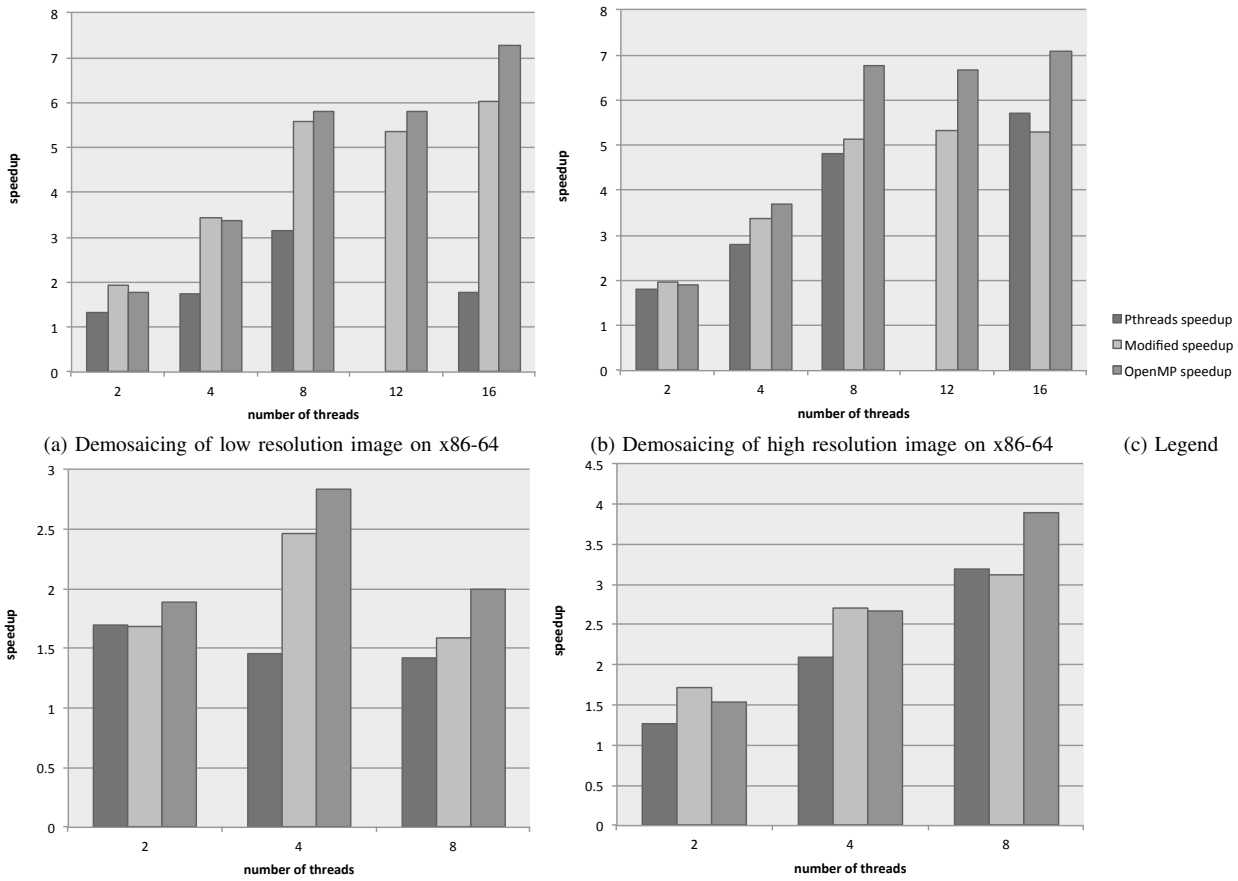
B. Demosaicing Speedups for IBM POWER

An anomaly was encountered when compiling the demosaicing code on POWER. Four of the 20 loop nests in the modified benchmark were not parallelized by `gcc`. All loops are successfully parallelized with Linux or Mac OS X on Intel platforms. It was also ensured that `gcc` were configured and built identically on the two platforms. This leads us to believe that the differences are caused by target dependent optimization decisions.

As a work-around, we inserted `parallel for` pragmas manually where the auto parallelization step in `gcc` would have done the same. It was verified that the workaround where 16 loops are auto-parallelized and 4 hand parallelized performs identical to the version where all loops were auto-parallelized on the Intel platform.

The experimental runs of the demosaicing benchmark were repeated on the POWER platform using the same high and low resolution images and the same compiler version. The speedups on this platform are summarized in Fig. 3d and Fig. 3e. The demosaicing code generally scaled significantly worse on the POWER platform which suggest that the benchmark needs additional tuning – e.g. improving the use of the memory hierarchy though loop tiling – to make the best use of this platform.

For the version we modified to allow auto-parallelization, the speedups on the low resolution image range from 1.7 to 2.5 for 2-8 threads with the best performance observed for 4 threads. The speedups for the high resolution image the



(a) Demosaicing of low resolution image on x86-64

(b) Demosaicing of high resolution image on x86-64

(c) Legend

(d) Demosaicing of low resolution image on POWER

(e) Demosaicing of high resolution image on POWER

Fig. 3. Demosaicing speedups on x86 and POWER platforms. Three parallelized versions are compared to the original, sequential program version: a version modified and auto-parallelized by `gcc`, a hand written version using `pthreads`, and a hand written OpenMP version. The `pthreads` version does not take advantage of all parallelism inherent in the benchmark. Also, it does not support 12 threads, so this data point is unavailable. Figures 3a and 3d speedups results for a image with a resolution of 768x512 and 3b and 3e show speedups for an image with a resolution of 5616x3744 pixels.

speedups were 1.7-3.1 and here the best result used all 8 threads.

The OpenMP code showed slightly better scaling with speedups ranging from 1.9-2.8 on 2-8 threads for the low resolution image and produced the best speedup using 4 threads. For the high resolution image, speedups were from 1.5 to 3.8 and the best result was obtained using 8 threads similar to the auto-parallelized version. Comparing the performance of the OpenMP and auto-parallelized versions shows that the latter delivers 79-89% of the performance of the former with the low resolution image and 80-111% for the high resolution image. Again the auto-parallelized version compares most favorably to the OpenMP version with 2-4 threads.

The `pthreads` version showed more modest speedups on the POWER platform. With the low resolution image, speedups were 1.7 on 2 threads but only 1.5 and 1.4 on 4 and 8 threads respectively. With the high resolution image, speedups were: 1.3 on 2 threads, 2.1 on 4 and 3.2 on 8 threads. With the small image, the auto-parallelized code delivered the same performance on 2 threads and 111-167% on 4 and 8 threads. With the big image, auto-parallelization delivers 130-135% on 2 and 4 threads and the same performance on 8 threads.

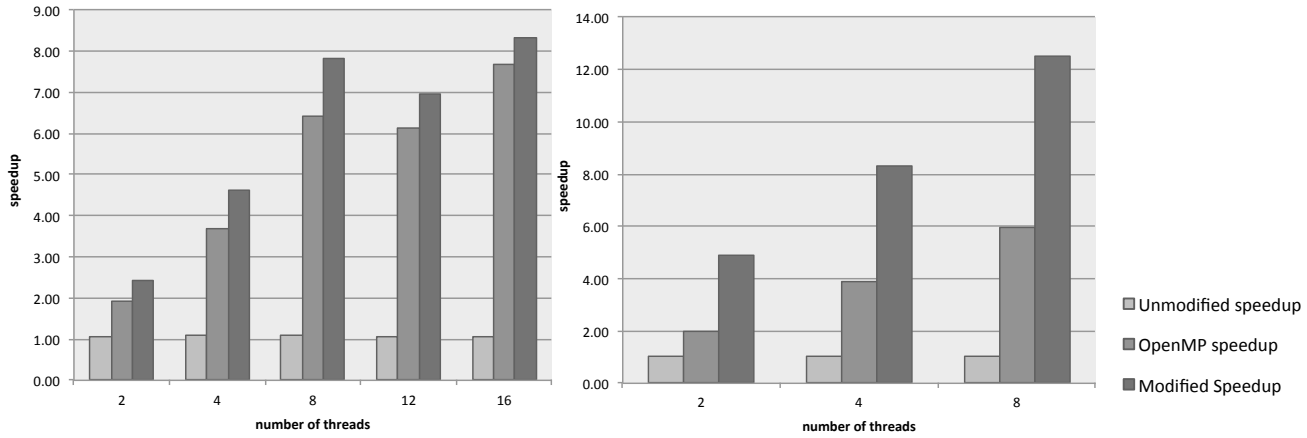
C. Edge Detection Speedups for Intel Xeon

We measured the speedups when `gcc` parallelized the original edge detection code and the modified code relative to sequential execution and relative the performance of the unmodified, auto-parallelized code. Finally, the results of auto-parallelization are compared with hand-parallelized OpenMP code. The speedups on the Intel Xeon system are summarized in Fig. 4a.

When auto-parallelizing the unmodified edge detection code with `gcc`, speedups are within 5%-10% since the most work intensive loop is not parallelized.

When auto-parallelizing the modified code, all three loop nests are transformed. The highest speedup of 8.32 used 16 threads, but a speedup of 7.8 is already obtained at 8 threads and 12 threads only resulted in a speedup of 7.0. Speedups on 2 and 4 threads are 2.44 and 4.62 which is super-linear.

The speedups of the OpenMP version ranged from 1.92 on 2 threads to 7.67 on 16 threads. Super-linear scaling was not observed. In effect, the auto-parallelized code outperformed the OpenMP code by 9-27% and the difference was greatest on 2 threads. The reason, we discovered, was that `gcc` was able to unroll the most frequently executed inner loop in the auto-



(a) Speedups on Intel Platform. (b) Speedups on POWER Platform. (c) Legend

Fig. 4. Edge detection speedups on x86 and POWER platforms. Speedups for modified and auto parallelized code is shown with respect to the sequential performance and with respect to the unmodified, auto-parallelized code. 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.

parallelized version. The use of manually inserted OpenMP pragmas on the other hand seems to prevent such unrolling.

D. Edge Detection Speedups for IBM POWER

The speedups that were observed on the POWER6 machine are summarized in Fig. 4b. When running the unmodified, auto-parallelized edge detection code, we observed a performance improvement of 2% on 2-8 threads.

When auto-parallelizing the code with modification based on the code comments, however, we observe speedups ranging from 4.9 on two threads and up to 12.5 on 8 threads. In contrast, the OpenMP version saw speedups of 2.0 on 2 threads, 3.9 on 4 and 6.0 on 8 threads. Hence, the performance of the auto-parallelized version was 210-242% relative to the OpenMP version. Again, we attribute the difference to `gcc`'s unrolling of the inner loop in the auto-parallelized version.

VIII. Related Work

Early work which pioneered user interaction in an auto-parallelization process include the ParaScope Editor, SUIF Explorer and PAT [1], [15], [18]. They parallelize sequential FORTRAN codes based on stand-alone analysis and user-interaction. Our work leverages the extensive analysis capabilities of a production compiler. This means that compiler feedback will adjust in response to improvements in the compiler analysis and in response to the use of different compiler flags. Our integration with a production compiler is also important since the analysis of loop nests benefits from scalar optimizations such as if-conversion and function inlining and from optimizing at link time.

The mechanisms used to rule out potential data dependencies also differ. ParaScope and PAT store information on potential data dependencies which the programmer has suppressed outside the source code so this information can be obsoleted by changes to the source code. Our compiler feedback, on the other hand, suggest that the `restrict` keyword is used to eliminate sets of dependencies. This is a standardized mechanism understood by most compilers. It also works when the code is changed.

Sean Rul et al. proposed the Paralax infrastructure which also exploits programmer knowledge for optimization [23]. Paralax is comprised of tree parts i) a compiler for automatic parallelization of outer loops containing coarse-grain pipeline-style parallelism, ii) a set of annotations which annotate data-dependencies which can not be eliminated via static analysis and which are verified dynamically and iii) a tool which suggests how the programmer may add annotations to the program.

Paralax is complimentary to our work. It parallelizes irregular, pointer-intensive codes whereas we focus on codes amenable to automatic parallelization after some modification. The suggestions generated by the Paralax tool rely on both static analysis and profiling information whereas our suggestions, so far, do not require program profiling.

Suggestions for locality optimizations, SLO, provides refactoring suggestions at the source level aiming to reduce reuse distances and thus the number of cache misses [3]. The suggestions are based on cache profiling runs and are complimentary to the types of refactoring suggested by our tool. For instance, SLO does not help the programmer expose parallelism in the source code.

The latest releases of IBM XL C/C++ and Intel `icc` can generate compilation reports and may suggest changes in response to code which cannot be analyzed by the compiler. [8], [13]. This work is complimentary to ours and relies on vendor specific pragmas. We used `icc`'s *Guided Auto-Parallelism* feature on the demosaicing kernel by inserting `icc`-specific pragmas as suggested by the compiler. This allowed `icc` to parallelize a minority of the loops. The resulting performance varied from a marginal speedup to a sizable slowdown.

IX. Conclusions

For many parallel applications, performance relies on loop-level parallelism. Regrettably, many source codes are written in ways that prevent auto-parallelization of loops. To address this problem, we developed an interactive compilation feedback system that guides the programmer in iteratively modifying the source code.

We evaluate our infrastructure via two sequential kernel benchmarks that pose problems for current production compilers. By refactoring application source code, we enable greater auto-parallelization of relevant loop nests. We compare auto-parallelization with manual-parallelization across different program inputs, systems and benchmark programs. Auto-parallelization delivers the best result in 12 cases, while hand-parallelization remains better in 11 remaining situations. At low and medium thread counts auto-parallelization generally performs similar to or better than hand-parallelized and optimized codes.

We enable the programmer to chose among refactoring alternatives and show that these affect performance differently. We deliver speedups of up to 6.0 for a demosaicing kernel on the Intel Xeon system (and up to 3.1 on the IBM POWER6 system). We speed up an edge-detection kernel by factors of up to 8.3 the Intel Xeon system and up to 12.5 on the IBM POWER6 system.

Our results demonstrate the opportunities to extract more parallelism from many source codes. Based on the different improvements on our two platforms, we conclude that auto-parallelization should be combined with platform-specific tuning to extract additional performance. Prioritization of compiler feedback remains important: information from our compilation feedback system as well as other compilers are likely to consist of hundreds of individual comments, even when the code contains only a handful of missed opportunities for optimization. Discarding superfluous comments thus represents an important direction for ongoing research.

Acknowledgment

Parts of this work was done while the first author was on a HiPEAC^2 internship at IBM Haifa and was supported by the SMECY project. 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.

References

[1] B. Appelbe, K. Smith, and C. McDowell. Start/Pat: A parallel-programming toolkit. *IEEE Softw.*, 6:29–38, July 1989.

[2] U. K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988.

[3] K. Beyls and E. D'Hollander. Refactoring for data locality. *IEEE Computer*, 42(2):62–71, 2 2009.

[4] Y. Bouchebaba et al. MPSoC memory optimization for digital camera applications. In *Proceedings of DSD '07*, 2007.

[5] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[6] Computer Architecture and Parallel Systems Laboratory. Open64. http://www.open64.net/. Date accessed: March 13, 2011.

[7] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science and Engineering*, 5:46–55, 1998.

[8] Y. Du et al. Explore optimization opportunities with xml transformation reports in ibm xl c/c++ and fortran compilers for aix. http://public.dhe.ibm.com/software/dw/rational/emi/Explore_XL_CCplus_and_Fortran_Compilers_for_AIX_XML_Transformation_Reports_Options.pdf, 2010. Date accessed: January 13, 2011.

[9] Free Software Foundation. GNU Compiler Collection. http://gnu.gcc.org.

[10] M. J. Garzaran et al. Program optimization through loop vectorization. http://sc10.supercomputing.org/schedule/ event_detail.php?evid=tut140, 2010. Date accessed: December 19, 2010.

[11] M. Hind. Pointer analysis: Haven't we solved this problem yet. In *Proc. of PASTE '01*, 2001.

[12] Intel Corp. Intel C++ Composer XE. http://software.intel.com/en-us/articles/intel-compilers/. Date accessed: March 13, 2011.

[13] Intel Corp. Guided auto-parallelism (GAP). http://software.intel.com/en-us/articles/guided-auto-parallel-gap/, 2010. Date accessed: March 16, 2011.

[14] International Organization for Standardization. ISO/IEC 9899:1999, December 1999.

[15] K. Kennedy, K. S. McKinley, and C. W. Tseng. Interactive parallel programming using the parascope editor. *IEEE Trans. Parallel Distrib. Syst.*, 2:329–341, July 1991.

[16] C. Lee et al. UTDSP benchmark suite. http://www.eecg.toronto.edu/corinna/DSP/infrastructure/UTDSP.html, 1998. Date accessed: July 4, 2009.

[17] X. Li, B. Gunturk, and L. Zhang. Image demosaicing: a systematic survey. volume 6822. SPIE, 2008.

[18] S.-W. Liao et al. Suif explorer: an interactive and interprocedural parallelizer. In *Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '99, pages 37–48, New York, NY, USA, 1999. ACM.

[19] Oracle Corp. Oracle Solaris Studio. http://www.oracle.com/technetwork/server-storage/solarisstudio. Date accessed: March 13, 2011.

[20] S. Pop et al. In *Proc. of GCC Developer's Summit*.

[21] The Eclipse Foundation. Eclipse C Development Tools. http://eclipse.org/cdt/. Date accessed: March 13, 2011.

[22] The Portland Group. PGI C/C++ Workstation. http://www.pgroup.com/products/pgiworkstation.htm. Date accessed: March 13, 2011.

[23] H. Vandierendonck, S. Rul, and K. D. Bosschere. The paralax infrastructure: Automatic parallelization with a helping hand. In *Proc. of PACT*, 2010.

[24] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.