SCIENCE CHINA
Information Sciences

• Supplementary File •

Identifying superword level parallelism with extended directed dependence graph reachability

Jie ZHAO* & Rongcai ZHAO

National Digital Switching System Engineering & Technological Research Center, Zhengzhou 450001*, China*

Appendix A Theorems Proofs

As shown in Figure 1, for any u_i $(1\leqslant i\leqslant m)$ and $v_j(1\leqslant j\leqslant n)$, it is manifest that u_i can reach w_1 while v_j is also reachable to w_2. On the basis of the definition of data dependence, there must be at least one statement storing into the memory location when there is a data dependence from statement S1 to S2. Hence the dependence edges between S1 and S2 will certainly pass through either of w_1 and w_2. Therefore, u_i is reachable to v_j iff any u_k $(k\neq i)$ can reach v_j. Likewise, v_j is reachable to u_i iff any v_l $(l\neq j)$ can reach u_i. According to this fact, a compiler only need to analyze the first read reference node of each statement, u_1 and v_1 respectively, rather than visit each node, and that will carry its point.

Before visiting edg, we should eliminate its redundant dependence edges. In terms of SIMD vectorization, a dependence is invalid if its dependence distance is not less than the vectorization factor. Dependence testing techniques can determine dependence distances, which are carried by dependence edges of edg. So we can get the information and determine whether the distance of a dependence is less than the vectorization factor, and eliminate dependence edge from edg accordingly. Next, we turn to determine dependence information with the reachability of nodes from different SCCs. We have the following theorems.

Theorem 1. For any nodes $u_i(1\leqslant i\leqslant m)$ and $v_j(1\leqslant j\leqslant n)$, if u_i is reachable to v_j but v_j cannot reach u_i, then S1 and S2 are SLP vectorizable but S1 must be executed before S2.

Proof. First of all, since v_j cannot reach u_i, no dependence edge from S2 to S1 exists, saying S1 does not depend on S2. Moreover, u_i is reachable to v_j, which implies S2 depends on S1, so we should consider two cases.

(1) If S2 appears after S1, then all the dependences are either loop-independent or forward loop-carried. In this case, S1 and S2 are SLP vectorizable.

(2) If S1 appears after S2, then all the dependences are loop-carried. Otherwise, there would be dependence edges deriving from S2 to S1, which violates the precondition that v_j cannot reach u_i. According to the Fundamental Theory of Dependence, reordering the execution sequence of S1 and S2 will preserve every dependence in this program, thereby preserving the meaning of the program. After such a reordering transformation, all dependences will be transformed into forward, so S1 and S2 are also SLP vectorizable in this case, but S1 must be executed before S2.

Therefore, when u_i is reachable to v_j but v_j cannot reach u_i, S1 and S2 are SLP vectorizable but S1 must be executed before S2.

Theorem 2. For any nodes $u_i(1\leqslant i\leqslant m)$ and $v_j(1\leqslant j\leqslant n)$, if u_i and v_j are mutually reachable and none of the paths deriving from S1 to S2 passes through w_2, then S1 and S2 are SLP vectorizable but node splitting has to be applied to S2.

Proof. u_i and v_j are mutually reachable, indicating that there exists at least one dependence cycle between S1 and S2. Because none of the paths deriving from S1 to S2 passes through w_2, neither anti-dependence edge from S1 to S2 nor true dependence edge from S2 to S1 exists. As a consequence, the situation can only be the following. The dependences deriving from S1 to S2 are true while the dependences from S2 to S1 are anti-dependences. Consider the true dependences and we have the following cases.

(1) If the true dependences from S1 to S2 are loop-independent, S1 will certainly appear before S2. Otherwise, there would be no loop-independent dependences from S1 to S2. So the anti-dependences from S2 to S1 have to be loop-carried and backward. Apply node splitting transformation to S2 and exchange the statement whose read references carry the anti-dependence with S1, then all the dependences will be transformed into forward. S1 and S2 are thus SLP vectorizable.

* Corresponding author (email: zjbc2005@163.com)

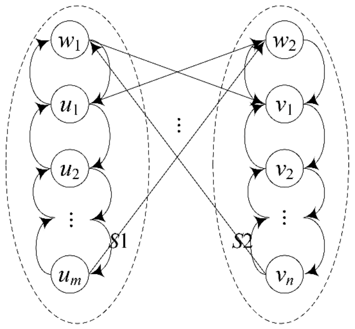

Figure A1 An edg of arbitrary statements S1 and S2.

(2) If the true dependences from S1 to S2 are loop-carried and S1 appears before S2, these dependences are forward. In this case, all anti-dependences from S2 to S1 are backward loop-carried. Thus, we can apply node splitting transformation to S2 and exchange the locations of the statements whose read references carry the anti-dependence with S1, then all the dependences will be transformed into forward.

(3) If S1 appears after S2, reordering S1 and S2 maintains the Fundamental Theory of Dependence when anti-dependences from S2 to S1 are loop-carried, and it can transform the backward true dependences into forward. As a result, S1 and S2 are SLP vectorizable.

(4) If the anti-dependences from S2 to S1 are loop-independent, we can apply node splitting transformation to S2 and exchange the locations of the statements whose read references carry the true dependences with S1, then all the dependences will be transformed into forward. S1 and S2 are therefore SLP vectorizable.

Consequently, when u_i and v_j are mutually reachable and none of the paths between S1 and S2 passes through w_2, S1 and S2 are SLP vectorizable but node splitting has to be applied to S2.

Note that a compiler does not need to check all dependence edges to determine whether any path between u_i and v_j passes through w_2, since whether a dependence edge enters or comes out from w_2 can be acquired from adg easily. The information has also been copied into edg, so it can be satisfied when the number of w_2's edges is 0.

Theorem 3. For any nodes $u_i(1\leqslant i\leqslant m)$ and $v_j(1\leqslant j\leqslant n)$, if w_1 appears before w_1 on each loop path from u_i passing through v_j, then S1 and S2 are not SLP vectorizable.

Proof. Similar to Theorem 2, there exists at least one dependence cycle between S1 and S2. Since w_1 appears before w_2 on each loop path from u_i passing through v_j, u_i can reach S2 only by passing through w_1 and v_j can reach S1 only by passing through w_2. It indicates that all dependences between S1 and S2 are true dependences, so we still have to consider two cases.

(1) If there are loop-independent dependences, then S1 and S2 cannot be exchanged. Hence, S1 and S2 are not SLP vectorizable.

(2) If there are no loop-independent dependences, which implies that all dependences between S1 and S2 are loop-carried, S1 and S2 can be exchanged safely. However, backward true dependences will still exist even after reordering transformation is applied, so S1 and S2 are not SLP vectorizable.

Therefore, when w_1 appears before w_2 on each loop path from u_i passing through v_j, S1 and S2 are not SLP vectorizable.

Theorem 4. For any nodes $u_i(1\leqslant i\leqslant m)$ and $v_j(1\leqslant j\leqslant n)$, if w_1 appears after w_2 on each loop path from u_i passing through v_j, then S1 and S2 are SLP vectorizable but node splitting has to be applied to S2.

Proof. Due to the same reasons in Theorem 2, there are dependence cycles between S1 and S2. Since w_1 appears after w_2 on each loop path from u_i passing through v_j, all dependences between S1 and S2 are anti-dependences. Consider the following cases.

(1) When there are loop-independent dependences, there is no harm to assume all the loop-independent dependences are from S1 to S2. So we can apply node splitting transformation to S2 and exchange the locations of the statements whose read references carry the anti-dependence with S1, then all the dependences will be transformed into forward. At this point, S1 and S2 are SLP vectorizaible.

(2) When there is no loop-independent dependence, all dependences between statements are loop-carried. So the reordering transformation is valid. There is no harm to assume all dependences from S1 to S2 are forward, and we can apply node splitting transformation to S2 and exchange the locations of the statements whose read references carry the anti-dependence with S1, then all the dependences will be transformed into forward. So S1 and S2 are thus SLP vectorizaible.

Therefore, when w_1 appears after w_2 on each loop path from u_i passing through v_j, S1 and S2 are SLP vectorizable but node splitting has to be applied to S2.

Theorem 5. For any nodes $u_i(1\leqslant i\leqslant m)$ and $v_j(1\leqslant j\leqslant n)$, if neither u_i is reachable to v_j nor v_j can reach u_i, then S1 and S2 are SLP vectorizable.

Proof. Neither u_i is reachable to v_j nor v_j can reach u_i, which indicates that there are no dependence edges between S1 and S2. In other words, S1 and S2 can be executed parallel in any order. Hence, S1 and S2 are SLP vectorizable.

Theorem 6. For any given read reference node $u_i(1\leqslant i\leqslant m)$ and a write reference node w_1, if there is a dependence edge from w_1 to u_i, then S1 is not SLP vectorizable.

Proof. There is a dependence edge from w_1 to u_i, so S1 has a true self-dependence. As can be seen from Figure 4(a), the transformed code which has been applied node splitting is shown in Figure 4(c). In this case, there is a dependence cycle between the statements, each of the edges is true dependence. According to Theorem 3, the transformed code is not SLP vectorizable, indicating that S1 in original code is not SLP vectorizable.

Theorem 7. For any given read reference node $u_i(1\leqslant i\leqslant m)$ and a write reference node w_1, if there is no dependence edge from w_1 to u_i, then S1 is SLP vectorizable.

Proof. We need to discuss the following two cases.

(1) If there is no dependence edge from u_i to w_1, then no self-dependences exist in this case. Hence S1 is SLP vectorizable.

(2) If there is one dependence edge from u_i to w_1, then an anti-self-dependence exists, as shown in Figure 4(b). The transformed code that has been applied node splitting is listed in Figure 4(d). In this case, there are no dependence cycles and all dependences are forward. Therefore, the transformed code can be SLP vectorizable, which implies that S1 of original code is SLP vectoriziable.

Appendix B Evaluation Details

To verify the technique proposed in this work, we implemented the algorithm in Open64-5.0 that has been equipped with SLP algorithm. In the first instance of this section, we will compare the power of SLP vectorization identification of the original Open64-5.0 with the optimized version, which has implemented our technique. Next, we will compare the ability of the optimized Open64-5.0 compiler with GCC and Intel ICC. At last, we will compare the performance of our generated programs with current mainstream vectorizing compilers, so as to demonstrate the advantages of our approach.

Appendix B.1 Comparison with the original Open64 compiler

To demonstrate how our technique improves the power of the Open64-5.0 compiler to identify SLP vectorization, we designed 15 micro-kernel programs to compare the results before and after our method is used. We take all kinds of dependences into account, including forward/backward, true/anti[1], loop-carried/-independent, self/inter-statement and cyclic/acyclic of dependences. Cases in which dependence distance is not less than vectorization factor are also included. We believe these programs can reflect a compiler's ability to identify SLP vectorization in any complicated cases. As can be seen, the evaluation results have been listed in Table 1. The kernel codes of each program are also shown in the table. We use original-Open64 to represent the Open64-5.0 compiler without implementing our method while optimized-Open64 is one that has implemented the technique proposed in this paper. We use *-O*3 *-LNO*:*slp*=1 options to enable advanced optimizations and SLP vectorization in Open64 compiler. The data type used in these examples is 64-bit integer, and we assume the vector register width is 256 bits, which means the vector factor is 4 for these programs.

We can see from Table 1 that the original Open64-5.0 compiler can recognize none of the programs listed in Table 1 as SLP vectorizable, while the optimized Open64-5.0 compiler is able to vectorize most of them, out of which the programs *no*.1 and *no*.12 are not SLP vectorizable. The cases are same with the situations described in Theorem 3 and 6, so they are not SLP vectorizable. For all the other programs, optimized-Open64 can identify them as SLP vectorizable and generate correct and effective vector codes.

Among these programs, the last case may confuse the readers. It looks like none of the theorems introduced is related to this case. As a matter of fact, the program represents a case that dependences of multi-dimensional arrays are held. It seems that there is one dependence cycle in this program and both dependences are true. original-Open64 cannot recognize it as SLP vectorizable, since there are loop-carried dependences. However, the two dependences do not form a cycle in this case, although both of the two dependences are true. This cannot be recognized correctly by original-Open64. optimized-Open64 can figure out that the dependence of array a is carried by loop i while the other dependence of array b is carried by loop j. The levels of these dependences are different with each other, so the optimized compiler will not treat them as a cycle and thus can identify this case correctly. From Table 1 we can conclude that the optimized Open64-5.0 compiler that has implemented our technique can enhance the power of exploiting SLP vectorization in practice.

Appendix B.2 Comparisons with GCC and ICC

In this section, we will compare the SLP vectorization of the optimized Open64-5.0 with those of GCC and Intel ICC. For the illustration purposes, we perform the experiment from two aspects.

In the first place, we still use these three compilers to identify the hand-coded programs described in last subsection. We use GCC4.9 and Intel ICC14.0, both of which are the latest versions released by their developers recently. To enable optimizations and SLP vectorization, *-O*3 is used for GCC4.9 and *-O*3 *-vec-report*3 are used for ICC14.0. As can be seen from Table 1, their results are also listed. The items filled labelled with "*" are those not recognized as SLP vectorizable by neither GCC4.9 nor ICC14.0 but identified as SLP vectorizable by optimized Open64-5.0. For the other programs, optimized Open64-5.0 performs as well as GCC4.9 and ICC14.0, so we will discuss the programs in which our compiler prevails over the others[2].

1) We do not consider output dependence, as it will be optimized or does not impact on vectorziation when it cannot be eliminated.

2) ICC14.0 con only vectorize program 3 partially because it distributes the loop over the statements and reorders the loops and vectorizes them as opposed to reordering the statements, strip-mining, and vectorizing.

Table B1 SLP vectorzation recognition of the hand-coded programs

Items	Kernel codes	Dependence types	original-Open64	GCC4.9	ICC14.0	optimized-Open64
1	a[i+4] = b[i-4]; b[i] = a[i];	cyclic, true dependences	×	×	×	×
2	a[i+4] = c[i]; b[i] = a[i];	acyclic, forward true dependences	×	√	√	√
3*	a[i] = b[i]; b[i+1] = c[i];	acyclic, backward true dependences	×	×	×	√
4	a[i-4] = b[i+4]; b[i] = a[i];	cyclic,anti-dependences	×	√	√	√
5	a[i] = c[i+4]; c[i] = b[i];	acyclic, forward anti-dependences	×	√	√	√
6*	a[i] = c[i]; b[i] = a[i+1];	acyclic, backward anti-dependences	×	×	×	√
7	a[i+4] = b[i]; b[i-4] = a[i];	forward true & forward anti	×	√	√	√
8	a[i+4] = b[i];c[i] = a[i] + a[i+8];	forward true & backward anti	×	√	√	√
9*	c[i] = a[i] + a[i+2]; a[i+1] = b[i];	backward true & forward anti	×	×	×	√
10*	a[i-1] = b[i]; b[i+1] = a[i];	backward true & backward anti	×	×	×	√
11	a[i+4] = a[i];	true self-dependence	×	√	√	√
12	a[i+1] = a[i];	true self-dependence	×	×	×	×
13	a[i] = a[i+4];	anti-self-dependence	×	√	√	√
14*	a[i] = a[i+1];	anti-self-dependence	×	×	×	√
15*	a[i][j] = b[i][j-1]; b[i][j]=a[i-1][j-1];	multi-dimensional case	×	×	×	√

In fact, each program among those item with a star label “*” has a corresponding item with different dependence distances, whose statement order is contrary but is not labelled by “*”. For instance, item 3 and item 2 is such a program pair that the statement orders are contrary and each of their dependence distances is different with that of the other. We can reorder the statements of program 3 to transform it into the form of program 2. However, our compiler recognize program 3 as SLP vectorizable, but neither GCC4.9 nor ICC14.0 does. It happens again when each of program 2’s dependence distances less than 4, while the situation is contrary when it is equal to or greater than 4, as can seen from Table 1.

It illustrates that both GCC4.9 and ICC14.0 assume the width of the vector register is 256 bits, and eliminate a dependence edge when its distance is not less than the vectorzation factor. Our vector register’s width varies according to the demand of targeted architecture, which of course can be 256 bits or more. When the targeted architecture provides 512-bit vector registers, the vectorization factor will be larger than 4 for integer data. In this case, the dependence distance is less than vectorization factor. The difference between our method and the algorithms in GCC4.9 and ICC14.0 is that a loop may be recognized as SLP vectorizable even it carries a dependence whose distance is less than the vectorzation factor. In these cases, the compiler can pack data first and then shift the data stored in vector registers according to the generated offsets due to dependence distances. We escape from using the redundant packing and unpacking operations in this way, thereby generating more efficient vector codes.

For illustration purposes, consider program 9 as an example, whose corresponding example is program 8. Undeniably, program 9 is not identified as SLP vectorizable by neither GCC4.9 nor ICC14.0, since the distance of dependence carried by array a is less than 4, but optimized-Open64 recognizes it as SLP vectorizable. In comparison, all the three compilers recognize program 8 as SLP vectorizable. The optimized-Open64 compiler applies a series of transformations to program 8 and 9, including node splitting, statement reordering, register data shifting and so on, to vectorize them. On the one hand, GCC4.9 and ICC14.0 eliminate redundant dependence edges when their distances are large enough. This optimization is also implemented in our compiler. On the other hand, they did not considered various other optimizations, e.g., node splitting, statement reordering, etc. as mentioned above, which, however, have been taken into account in our compiler. As a result, the technique proposed in this work performs better than others when confronted with complicated data dependences as listed in Table 1.

In the second place, we still test the 100 programs, which cover 317 loops with the compilers. As we introduced above, these programs are extracted from the gcc-vect benchmarks. gcc-vect is a package of benchmarks provided by the developers of GCC compiler series. It is designed to evaluate the power of a vectorizing compiler to identify vectorization. There are hundreds of programs included in this package, which cover various practical application cases that a compiler should take into consideration. For instance, complicated dependence relations, data type conversions, function calls, etc.

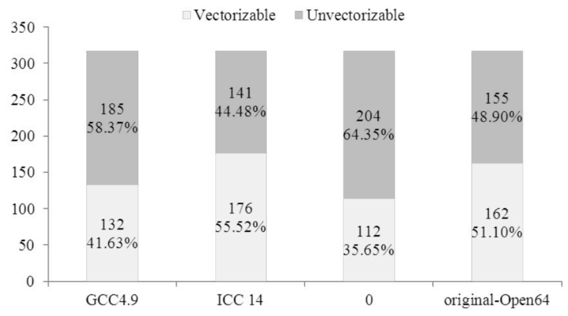

Figure B1 Comparison of SLP vectorization ability before optimized-Open64 considered the reduction operation.

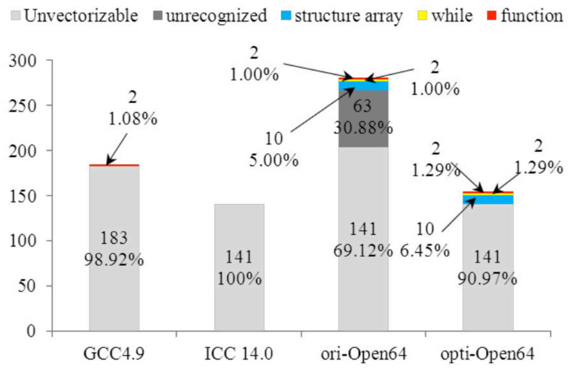

Figure B2 Comparison of SLP vectorization ability after optimized-Open64 considered the reduction operation.

It is generally believed that evaluation derived from this package can reflect a compiler's ability of identifying vectorizaion properly. We tested each program in the package, but only 100 representative ones are shown in this section for illustration purposes, since the remaining programs are derived by modifying those from these 100 programs slightly. The experimental results of the 317 loops included in these programs are summarized in Figure 2.

As can be seen from this figure, our approach can vectorize 46.37% of the total amount, while the vectorizable loops recognized by GCC4.9 and ICC14.0 account for 41.64% and 55.52% respectively. That is to say, our algorithm can identify more vectorizable loops than that recognized by GCC4.9, but still falls behind ICC14.0. Note that in terms of a loop nest, the GCC4.9 compiler is likely to vectorize outer loops while optimized-Open64 and ICC14.0 prefers inner loops.

We continue to analyze the unvectorizable loops that ICC14.0 is able to vectorize but optimized-Open64 cannot. We find that these loops fall into four categories, which are (a) while loops, (b) loops including arrays of structures, (c) loops invoking function calls and (d) loops containing statements with reduction operations, respectively. For case (a), as Open64 always tries to rewrite an input loop in Fortran DO style before analysis, it may happen that a while loop cannot be transformed correctly and then is recognized as unvectorizable. However, we can identify correctly those, which are transformed successfully. The situation comes down to "point to" problems for case (b), which we have not considered at present. We just continue to use the analysis implemented in PathScale without any modification; hence, cases that arrays of structures or pointers are not recognized properly happened during the evaluation. Loops in case (c) invoke mathematical function calls like $pow()$ function embedded in the operating system. Consequently, we transform the function into other forms written as $sqrt()$ or $square()$ functions and add them into a list of vectorizable operations, thereby leading to successful recognition. Now let us come into the last case.

As described above, both VP and SLP have been implemented in GCC4.9 and ICC14.0, whilst the optimized Open64-5.0 compiler had only implemented VP at the very beginning. After appending the SLP algorithm in optimized-Open64, we try to recognize the loops including statements with reduction operations as SLP vectorizable, but the packing operations become very time-consuming. On the other hand, these loops are vectorized easily and the compiling is much more efficient when we use VP method of Open64-5.0, which does not necessitate packing operations. As a result, we vectorize such loops with VP instead of with SLP algorithm under compilation option -LNO:$simd$=1, which is also adopted by GCC4.9 and ICC14.0. So we enable the VP algorithm of optimized-Open64 as well and test these 100 programs again. At this time, the number of our vectorizable loops reaches a proportion of 51.10% as indicated in Figure 3.

From the figure we can see that our method becomes comparable with that of ICC14.0. Next, we analyze the unvectorizable loops, which cannot be vectorized by all of these compilers. The results are shown in Figure 4. It indicates that our compiler is restricted to cases such as while loops, function calls, arrays of structures, etc. while GCC4.9 and ICC14.0 take them into account. In terms of complicated dependences, the optimized Open64-5.0 compiler can vectorize loops that carry some loop-carried true dependences, since we fulfill its potential by applying numerous transformations and optimizations. In comparison, GCC4.9 is not good enough in this respect. The data in Figure 7 also demonstrates that our technique is

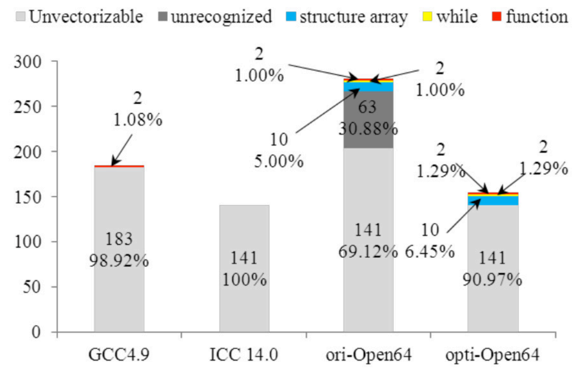

Figure B3 Details of unvectorizable loops.

able to recognize as many vecotrizable loops as the ICC14.0 compiler when not considering the above 4 cases. In other words, the number of our unvectorizable loops caused by complicated dependences is the same with that of ICC14.0.

Appendix B.3 Performance comparison of practical applications

To illustrate how our technique exerts influence on practical applications, we experiment on some applications with the optimized Open64-5.0, ICC14.0, PathScale 5.0.0 compilers. Besides, we also implement the state-of-the-art recognition algorithm proposed by Liu et al. What we shall do is timing the serial applications first, and then recording execution time of the vector codes generated by our method as well as those vectorized by these compilers and algorithms. At last, we calculate their speedups and compare the performance.

The experimental methodology is as follows. First, compile and time the tested applications by ICC14.0 and PathScale5.0.0 without vectorization, and then vectorize these applications with these compilers again and record the execution time of the vector codes. Second, run the tested applications using a native compiler embedded in targeted operating system without any optimizations[3)], and take down their serial execution time. After that, transform these applications into vector versions with the Open64-5.0 compiler together with our method and Liu et a"s algorithm individually, and compile and time the generated codes with the native compiler. Finally, calculate and compare the speedups respectively[4)]. The compilation options are the same with the above subsections.

We choose one micro-kernel FFT program, one large-scale application OpenCFD and numerous benchmarks from the SPEC2006 benchmark and NAS Parallel Benchmarks as the test suites. Details about these programs are summarized in Table 2.

The original-Open64 and GCC4.9 can rarely identify vectorizable loops of these programs, so it failed to vectorized all of the applications of these applications and we thus do not show their result.

All the compilers and algorithms return that there are loop-carried true dependences among the iterations of the FFT's kernel loop. As the dependence distance is less than 4, all the others recognizes this loop as unvectorizable, while our work vectorize it successfully. As a result, the other compilers execute the program serially while our vector codes experience an improvement of 46.4%.

The ICC14.0 and PathScale 5.0.0 compilers, and the Liu et al's algorithm cannot vectorize 31 loops of OpenCFD, as they are assumed to hold loop-carried dependences with distance less than 4. Fortunately, our technique is able to vectorize 12 loops out of them successfully. So our work recognizes and vectorizes more loops than these methods, generating an improvement of 80.4%. Likewise, the performances of the other generated benchmarks outperform those of ICC14.0 , PathScale 5.0.0 and the state of the art. The side effect of the use of our technique is the overhead. As we need to construct edg and traverse it to recognize SLP vectorization, the compilation overhead of the whole SLP vectorization recognition process is increased by about 23%.

However, by contrary, the speedups of benchmarks *hmmer* and *libquantum* suffer from declines by 7.6% and 12.1% respectively when compared to ICC14.0. Unlike the above programs, the number of vectorizable loops of our work falls behind that of ICC14.0's vector codes, although the compiler also vectorizes some loops, which ICC14.0 cannot vectorize. The reason is as follows. Our technique is restricted to some cases as described above, and these cases appear frequently in these two programs. As a consequence of this, our codes are defeated by those of ICC14.0. Liu et al's algorithm performs best for the benchmarks *poverty* and *calculix*, as they applies an instruction scheduling scheme to exploit full potential of the data layout of the program. Our method falls behind their work by 2.4% and 1.4% for these benchmarks. The results of these programs are shown in Figure 5.

3) We do like this to remove the effect of the native compiler.

4) As the Open64 compiler is a source-to-source compiler, its generated codes should be recompiled by a native OS-provided compiler, while ICC compiler will generated the executable vectorized codes.

Table B2 Test Suite Details

No.	Programs	Descriptions
1	FFT	Fast Fourier Transformation
2	OpenCFD	Numerical simulation on compressible turbulent boundary flow
3	BT	Block tri-diagonal solver for computational and data movement
4	FT	3D partial differential equation solution using Fast Fourier Transformations
5	LU	Lower-upper Gauss-seidel solver for computational and data movement
6	SP	Scalar penta-diagonal solver for computational and data movement
7	433.milc	Simulations of four dimensional lattice gauge theory
8	434.zeusmp	Solver for equations of hydrodynamics and magnetohydrodynamics
9	435.gromacs	Simulation of the Newtonian equations of motion
10	436.cactusADM	Solver for the Einstein evolution equations
11	450.soplex	Solve a linear program using the Simplex algorithm
12	453.povray	Calculate an image of a scene
13	454.calculi	Solver for linear and nonlinear three- dimensional structural applications
14	456.hmmer	Search for patterns in DNA sequences
15	458.sjeng	Play chess and several chess variants
16	462.libquantum	Simulation of a quantum computer
17	464.h264ref	A reference implementation of H.264/AVC
18	470.lm	Use Lattice Boltzmann Method to simulate incompressible fluids

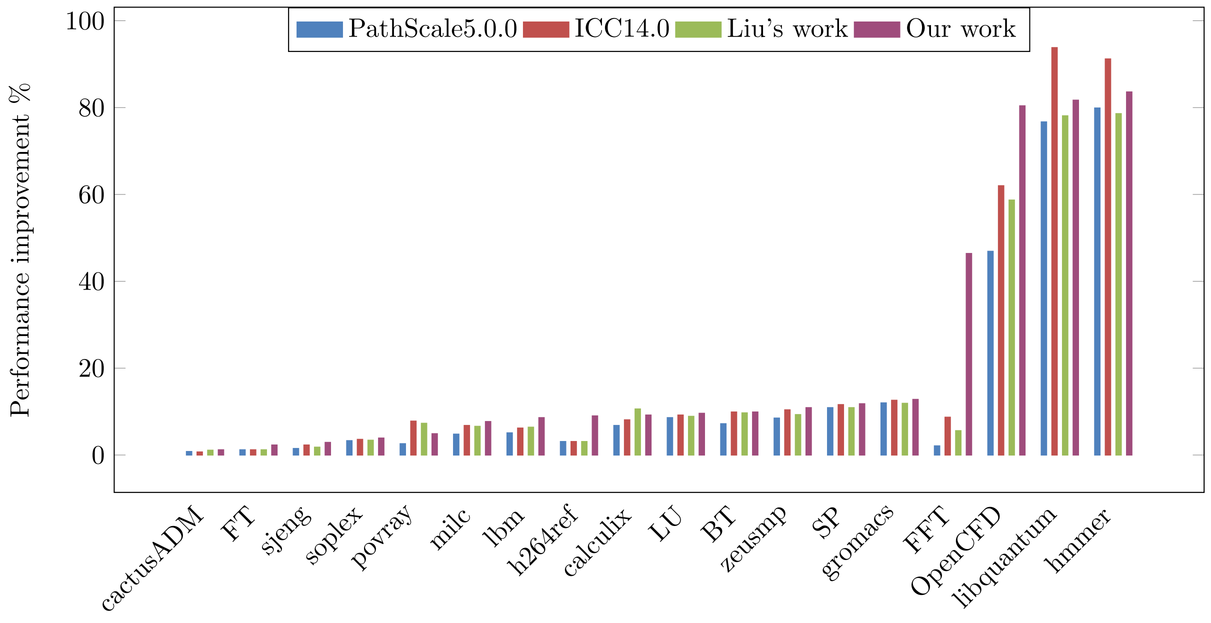

Figure B4 Performance comparison between different methods.