SCIENCE CHINA
Information Sciences

• LETTER •

January 2017, Vol. 60 019103:1–019103:3
doi: 10.1007/s11432-015-0163-7

Identifying superword level parallelism with extended directed dependence graph reachability

Jie ZHAO[1,2*] & Rongcai ZHAO[1]

[1]*National Digital Switching System Engineering & Technological Research Center, Zhengzhou* 450001*, China;*
[2]*Computer Science Department, École Normale Supérieure, Paris* 75005*, France*

Received March 19, 2016; accepted June 3, 2016; published online October 13, 2016

Citation Zhao J, Zhao R C. Identifying superword level parallelism with extended directed dependence graph reachability. Sci China Inf Sci, 2017, 60(1): 019103, doi: 10.1007/s11432-015-0163-7

Dear editor,

The increasing trend of multimedia applications' proliferation brings about multimedia extensions' wide use among existing microprocessors of mainstream manufacturers, e.g., Intel's MMX, AMD's 3DNow!, IBM's VMX/Altivec, etc. However, the implementation details behind these multimedia extensions vary from one architecture to another; hence, it generally comes in form of SIMD (single instruction multiple data) instructions, providing a mechanism to accelerate the performance of various application programs.

SLP (superword level parallelism) [1] is a widely used method to exploit SIMD vectorization. The principle of this technique is identifying isomorphic statements first, and then packing them together as a superword statement for concurrent execution. An isomorphic statement pair refers to such a statement pair that all their operations are in the same order and all the operands in the corresponding positions have the same data type.

The principle behind the recognition process is to determine the dependence information of the analyzed loops. Existing methods first construct adg (array dependence graph) according to the compiler's dependence testing techniques, and then construct sdg (statement dependence graph) based on adg. They need to determine whether statements can be vectorized by checking dependences between each statement pair, which requires the compiler to go back and forth between the adg and sdg. During the process, the compiler has to figure out both the source array reference node and the sink statement node carrying the same dependence, which makes itself error-prone.

To address this problem, we propose an edg (extended dependence graph) reachability based SLP recognition approach. We first extend adg and sdg to construct edg, which includes both dependences between each array pair and those of each statement pair. A compiler only needs to traverse edg when determining the SLP vectorization possibility. Besides, the dependence information can be figured out by only traversing the first read reference in edg, which simplifies the implementation and makes the process easy to use. We implement the method in Open64-5.0 compiler and the evaluation results show that the performance of our generated codes outperforms that of existing commercial compilers and the state of the art.

We implemente the SLP algorithm in Open64-5.0 compiler and performed an experiment on 100 representative programs covering 317 loops from the gcc-vect benchmarks. As a result, only 113 out of 317 loops can be identified as SLP vectorizable by the compiler. Among the remaining 204

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

The authors declare that they have no conflict of interest.

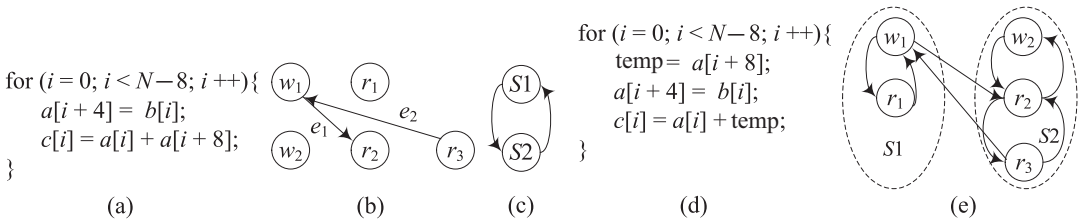

Figure 1 Observation on the SLP vectorization of Open64-5.0. (a) Code example; (b) array dependence graph; (c) statement dependence graph; (d) code after node splitting; (e) extended dependence graph.

loops recognized as unvectorziable by the SLP algorithm, existing commercial vectorizing compilers can vectorize 24% while 76% are really unvectorizable loops.

The SLP vectorization of a loop is determined by checking the parallelism of its enclosed basic blocks, whose recognition relies on the adg and sdg. The drawback of this method is that it does not take all kinds of dependence information into account. For example, Figure 1(b) and (c) are the adg and sdg of the example shown in Figure 1(a). Vectorizing compilers have to traverse the graphs of Figure 1(b) and (c) alternately as well as searching the array reference nodes and statement nodes carrying dependences iteratively. It does not discuss the dependence types here, which may also exert an important influence on the vectorization. To simplify the complicated analysis process, we first apply node splitting [2] to eliminate anti-dependences and then construct its edg as shown in Figure 1(e).

The SLP vectorizable recognition phase based on edg is the following. For read reference nodes r_1 and r_2 in Figure 2, there only exists one reachable path from r_1 to r_2. To be more specific, the path is (r_1, w_1, r_2). Similarly, the only reachable path from r_2 to r_1 is (r_2, r_3, w_1, r_1). The level of dependences and dependence distances are stored in dependence edges e_1 and e_2. r_1 and r_2 are mutually reachable implies there is a dependence cycle between the two statements. When both the paths pass through and only pass through one write reference node w_1, it implies that the dependence deriving from S1 to S2 is true while the other from S2 to S1 is an anti-dependence. The compiler can easily figure out whether a dependence is forward or backward by determining the relation between statement execution sequence and dependence direction. Hence we say that all the information can be obtained from edg. Compared with existing approaches, only the first read reference node of each statement needs to be analyzed, which simplifies the analysis process.

SLP vectorization analysis. For illustration purposes, we number the read reference nodes in an edg as ($u_1, \ldots, u_m$, $v_1, \ldots, v_n$), which means that there are m and n read reference nodes in the sentences of the interest. Arbitrary read reference node can be the candidate to determining dependence between statements. This is because it is manifest that u_i can reach w_1 while v_j is also reachable to w_2 for any u_i ($1 \leqslant i \leqslant m$) and $v_j (1 \leqslant j \leqslant n)$ according to the construction of edg.

In terms of SIMD vectorization, a dependence is invalid if its dependence distance is not less than the vectorization factor, which has been proved in [3], so we can eliminate redundant dependence edges before visiting edg. After that, we turn to determine dependence information with the reachability of nodes from edg and therefore have the following theorems. Please refer to the supplementary materials for the proof details of these 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.

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.

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.

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.

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.

The above theorems are applicable to the case that the source and sink of a dependence belong to different statements, which we define as normal case. We also need to consider self-dependent case. As a matter of fact, a self-dependent case can be transformed into a normal case after node splitting, and is thus amenable to these theorems.

For self-dependent cases, we have the following theorems.

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.

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.

Evaluations. To verify our technique, we implemente the algorithm in Open64-5.0. We conduct the experiment on 15 micro-kernel programs to compare the results. As a result, the Open64 compiler can recognize 13 more programs as vectorizable after applying our method. GCC 4.9 and Intel ICC14.0 can only vectorize 7 out of those programs.

We also experiment on 100 programs covering 317 loops extracted from the gcc-vect benchmarks. Our method can recognize 51.10% of the tested loops, while the vectorizable loop numbers of GCC 4.9 and ICC 14.0 are 41.64% and 55.52%. Our method performs better for sophisticated dependences cases, but falls behind ICC 14.0 in cases like while loops, array of structures, reduction operations, etc. This is due to the natural drawback of the intermediate representation of Open64 compiler rather than our method.

To illustrate how our technique exerts influence on practical applications, we experiment on 18 applications with our methods, ICC14.0, PathScale 5.0.0 and the state-of-the-art recognition algorithm [4]. The test suite is composed of a micro-kernel FFT program, a large-scale application OpenCFD and numerous benchmarks from the SPEC2006 benchmark suites and NAS Parallel Benchmarks.

Among the applications we used in the experiment, hmmer and libquantum suffer from declines by 7.6% and 12.1% when compared to ICC14.0. The reason is our technique is restricted to some cases as described above. Liu et al's algorithm performs best for 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. Our method outperforms other compilers and methods for all the remaining applications. Please refer to the supplementary materials for complete experimental results.

Conclusion. We proposed a novel approach to recognize SLP vectorization in this letter. This method is able to simplify the recognition process and efficiently determine all the dependence information relevant to SLP vectorization. The experimental results show that our approach prevails over the GCC4.9 compiler and is comparable with ICC14.0 and outperforms the state-of-the-art on practical applications. Our next research plan is to further improve the performance in those cases that we fall behind ICC 14.0.

Acknowledgements This work was supported by HEGAOJI Major Project of China (Grant No. 2009ZX01036-001-001-2) and Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (Grant No. 2013A11).

Supporting information The supporting information is available online at info.scichina.com and link.springer.com. The supporting materials are published as submitted, without typesetting or editing. The responsibility for scientific accuracy and content remains entirely with the authors.

References

1 Larsen S, Amarasinghe S. Exploiting superword level parallelism with multimedia instruction sets. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2000. 145–156
2 Padua D A, Wolfe M J. Advanced compiler optimizations for supercomputers. Commun ACM, 1986, 29: 1184–1201
3 Bulic P, Gustin V. D-test: an extension to Banerjee test for a fast dependence analysis in a multimedia vectorizing compiler. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS). Washington: IEEE Computer Society, 2004. 535–546
4 Liu J, Zhang Y, Jang O, et al. A compiler framework for extracting superword level parallelism. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM, 2012. 347–358