Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Journal of Systems Architecture 50 (2004) 729–742
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

Exploitation of parallelism to nested loops with dependence cycles

Weng-Long Chang [a,*], Chih-Ping Chu [b,1], Michael (Shan-Hui) Ho [a,2]

[a] Department of Information Management, Southern Taiwan University of Technology, Tainan County 710, Taiwan, ROC
[b] Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan City 701, Taiwan, ROC

Received 24 March 2001; received in revised form 6 June 2004; accepted 10 June 2004
Available online 25 August 2004

Abstract

In this paper, we analyze the recurrences from the breakability of the dependence links formed in general multi-statements in a nested loop. The major findings include: (1) A *sink variable renaming* technique, which can reposition an undesired anti-dependence and/or output-dependence link, is capable of breaking an anti-dependence and/or output-dependence link. (2) For recurrences connected by only true dependences, a *dynamic dependence* concept and the derived technique are powerful in terms of parallelism exploitation. (3) By the employment of global dependence testing, link-breaking strategy, Tarjan's depth-first search algorithm, and a topological sorting, an algorithm for resolving a general multi-statement recurrence in a nested loop is proposed. Experiments with benchmark cited from Vector loops showed that among 134 subroutines tested, 3 had their parallelism exploitation amended by our proposed method. That is, our offered algorithm increased the rate of parallelism exploitation of Vector loops by approximately 2.24%.

Keywords: Parallelizing compilers; Vectorizing compilers; Loop optimization; Data dependence analysis; Dependence cycle; Parallelism exploitation

1. Introduction

In high speed computing [20], there are the two most popular parallel computational models, distributed memory multiprocessors and shared memory multiprocessors. Because the technique to memory hardware [20] is improved, therefore, the access time of shared memory for a system of multiprocessors is obviously decreased and the

* Corresponding author. Tel.: +886 6 6891 421/2533131x 4300; fax: +886-6-2541621.
E-mail addresses: changwl@mail.stut.edu.tw, changwl@csie.ncku.edu.tw (W.-L. Chang), chucp@csie.ncku.edu.tw (C.-P. Chu), mhoincerritos@yahoo.com (M. (Shan-Hui) Ho).
[1] Tel.: +886 6 2757575x62527; fax: +886 6 2747076.
[2] Tel.: +886 6 2533131x4300; fax: +886 6 2541621.

1383-7621/$ - see front matter © 2004 Published by Elsevier B.V.
doi:10.1016/j.sysarc.2004.06.001

system has been increasingly used for scientific and engineering applications. However, the major shortcoming of shared memory multiprocessors is the difficulty in programming because programmers are responsible for analyzing data dependence relations among statements in programs and exploiting the parallelism of statements in programs among shared memory multiprocessors.

A successful vectored/paralleled compiler is capable of exploiting the parallelism of a program. Three features of a vectored/paralleled compiler determine its level of parallelism exploitation: (1) an accurate data dependence testing, (2) efficient loop optimization and (3) efficient removals of undesired data dependences.

Techniques for dependence analysis algorithms, which directly support data dependence testing, have been developed and used quite successfully [2,5,7–10,20,22–27]. Computationally expensive programs in general spend most of their time in the execution of loops. Extracting parallelism from loops in an ordinary program therefore has a considerable effect on the speed-up. Many loop optimizational methods have been developed and broadly fallen into two classes: loop vectorization and loop parallelization [3,4,20,21,28–30]. In terms of the reduction of data dependences, most researches concentrate on loop optimization by the front-end of vectored/paralleled compilers such *as scalar renaming, scalar expansion*, *scalar forward-substitution* and *dead code elimination* [20]. Studies on the back-end of vectored/paralleled compilers primarily deal with separation of parallelism execution and sequential execution of the statements. Relatively less attention has been given to data dependence elimination [1,9,11].

Recurrence is a type of π-blocks in a general nested loop, which is extracted at the time of loop distribution, a back-end phase [17]. Statement(s) involved in a recurrence are strongly connected via various dependence types. Famous techniques, such as *node splitting, thresholding, cycle shrinking*, etc., are able to eliminate data dependence of a recurrence to a certain extent, depending upon specific dependence types [1,17,18,20,21].

In this paper, we study a parallelism exploitation for loops with dependence cycles on basis of breaking dependence links. Formally the breaking strategies for dependence cycles were surveyed in a single loop [11]. Their method is extended to break the dependence links in a nest of loops. In Section 2, the concept of data dependence is reviewed. In Section 3, an analysis of the formation of dependence cycles is provided and three dependence links on the breaking-strategy basis are derived. For each link pattern, its features, breaking techniques and applications are introduced in detail. An algorithm is developed for the resolution of a general multi-statement recurrence. Experimental results showing the advantages of the proposed method are given in Section 4. Finally, Section 5 contains a conclusion.

2. Data dependence

It is assumed that there are two statements within a general loop. The general loop is presumed to contain n common loops. Statements are postulated to be embedded in n common loops. An array A is supposed to appear simultaneously within statements and if a statement S_2 uses the element of the array A defined first by another statement S_1, then S_2 is true-dependent on S_1. If a statement S_2 defines the element of the array A used first by another statement S_1, then S_2 is anti-dependent on S_1. If a statement S_2 redefines the element of the array A defined first by another statement S_1, then S_2 is output-dependent on S_1. Another dependence, control dependence, which arises due to control statements, is not addressed in this paper.

Each iteration of a general loop is identified by an iteration vector whose elements are the values of the iteration variables for that iteration. For example, the instance of the statement S_1 during iteration $\vec{i} = (i_1, \ldots, i_n)$ is denoted $S_1(\vec{i})$; the instance of the statement S_2 during iteration $\vec{j} = (j_1, \ldots, j_n)$ is denoted $S_2(\vec{j})$. If $(i_1,\ldots,i_n)$ is identical to $(j_1,\ldots,j_n)$ or $(i_1,\ldots,i_n)$ precedes $(j_1,\ldots,j_n)$ lexicographically, then $S_1(\vec{i})$ is said to precede $S_2(\vec{j})$, denoted $S_1(\vec{i}) < S_2(\vec{j})$. Otherwise, $S_2(\vec{j})$ is said to precede $S_1(\vec{i})$, denoted $S_1(\vec{i}) > S_2(\vec{j})$. In the following, Definitions 2.1–2.7, cited from [3,4,11], will be used later.

Definition 2.1. *Loop-independent dependence* refers to the dependence confined within each single iteration. Loop-independent dependences include loop-independent true-dependence (denoted δ^{t}), loop-independent anti-dependence (denoted δ^{a}) and loop-independent output-dependence (denoted δ^{o}). These relations are represented by the set (denoted Δ), i.e., $\Delta = \{\delta^{t}, \delta^{a}, \delta^{o}\}$.

Definition 2.2. Consistent loop-carried dependence refers to the dependence occurring across the iteration boundaries. Consistent loop-carried dependences include consistent loop-carried true-dependence (denoted $[\delta^{t}]$), consistent loop-carried anti-dependence (denoted $[\delta^{a}]$) and consistent loop-carried output-dependence (denoted $[\delta^{o}]$). These relations are represented by the set (denoted $[\Delta]$), i.e., $[\Delta] = \{[\delta^{t}], [\delta^{a}], [\delta^{o}]\}$.

Definition 2.3. A vector of the form $\vec{\theta} = (\theta_1, \ldots, \theta_n)$ is termed as a direction vector. The direction vector $(\theta_1, \ldots, \theta_n)$ is said to be the direction vector from $S_1(\vec{i})$ to $S_2(\vec{j})$ if for $1 \leqslant k \leqslant n$, $i_k \theta_k j_k$, i.e., the relation θ_k is defined by

$$\theta_k = \begin{cases} < & \text{if } i_k < j_k, \\ = & \text{if } i_k = j_k, \\ > & \text{if } i_k > j_k \\ {}^{*}\text{the relation of } i_k \text{ and } j_k \text{ can be ignored,} \\ \quad \text{i.e., can be any one of } \{<, =, >\}. \end{cases}$$

We remember $S_1\ \delta_{\vec{\theta}}\ S_2$, where $\delta \in \Delta \cup [\Delta]$ and $\vec{\theta} = (\theta_1, \theta_2, \ldots, \theta_n)$.

Definition 2.4. The dependence distance vector from $S_1(\vec{i})$ to $S_2(\vec{j})$ is denoted by $\text{dist}(\vec{i}, \vec{j}) = (j_1 - i_1, \ldots, j_n - i_n)$.

Definition 2.5. The dependence distance matrix of nested loops is a matrix whose columns are the dependence distance vectors of all the dependences in nested loops.

Definition 2.6. For an inter-statement or intra-statement dependence, the source variable of the dependence refers to the instance of an indexed variable to be accessed first; the sin$k$ variable of the dependence refers to the instance of indexed variable to be accessed later.

Definition 2.7. The direction vector matrix of nested loops is a matrix whose columns are the direction vectors of all the dependences in nested loops.

In the following, Lemma 2.1, cited from [12], introduces a dependence relation for which direct vectorization of the code is available. Lemma 2.2, cited from [1,21], emphasizes an important fact. That is, no matter how complicated dependence relations in statements are, so long as dependence relations do not form a cycle, the statements can always be vectorized via statement reordering.

Lemma 2.1. *A nest of loops without inconsistently dependent statement(s) can be fully vectorized directly if the following two conditions hold.*

1. *There does not exist a single statement S_1 such that $S_1(\vec{i})[\delta^{t}]S_1(\vec{j})$.*
2. *There does not exist a pair of statements S_1 and S_2, where $S_1 < S_2$, such that $S_2(\vec{j})\delta S_1(\vec{i})$, where $\delta \in [\Delta]$.*

Lemma 2.2. *A nest of loops with two statements S_1 and S_2, where $S_1 < S_2$, $S_2(\vec{j})\delta S_1(\vec{i})$ and $\delta \in [\Delta]$, is the only dependence relation in the statements, can be vectorized via the statement reordering technique, i.e., reorder S_1 and S_2 to become $S_2 < S_1$.*

The vectorization of statements in a nest of loops is inhibited if dependence relations in statements form a dependence cycle. Statements involved in a dependence cycle are strongly connected by various dependence edges. There exists at least one path between any pairs of statements in the dependence graph. The vectorizability of the dependence cycle(s) depends on whether the dependence cycle(s) can be broken or the level of dependence links that can be eliminated [1,6,11,21].

3. The breaking strategy and exploitation of parallelism

In general, we can exploit the parallelism of a nest of loops as long as one of the existing dependence links in a dependence cycle is breakable. If we break one dependence cycle and new dependence links satisfy the condition of Lemma 2.1, then these statements involved in the dependence cycle can be vectorized. If we break one dependence cycle and new dependence links satisfy the condition of Lemma 2.2, then these statements involved in the dependence cycle can be vectorized via statement reordering. So, to deal with the parallelism exploitation of a dependence cycle, we only need to consider the breakability of its dependence links in a dependence cycle.

If we examine the possible dependence links for a statement $S_2(\vec{j})$ on a statement $S_1(\vec{i})$, then we find seven dependence links which can exist: (1) true-dependence, (2) anti-dependence, (3) output-dependence, (4) true- and anti-dependences, (5) true- and output-dependences, (6) anti- and output-dependences and (7) true-, anti- and output-dependences [11]. With respect to breaking strategy of a dependence link of a dependence cycle in a nest of loops, the dependence types of a link can be classified as three patterns: (1) anti-dependence link, (2) output-dependence link and (3) true-dependence links including any possible dependence link. The first two link patterns can be broken while the third pattern of dependence link is unbreakable. In order to prove the correction of breaking strategy, we need to use a general dependence cycle to study the breaking strategy of the first two link patterns. Suppose we have n statements $S_0, S_1, \ldots, S_{n-1}$ in a nest of loops, where $S_0 < S_1 < \cdots < S_{n-1}$. These n statements are involved in a dependence cycle in a nest of loops. If there exists a pair of statements S_a and S_b, where $0 \leqslant a, b \leqslant n-1$ and $a \neq b$, and the dependence link from S_a to S_b is one of the first two link patterns, the breaking techniques and applications are described below.

3.1. Pattern I—anti-dependence link

For an anti-dependence link of a dependence cycle in a single loop, a $\sin k$ variable-renaming algorithm to break such a dependence cycle was developed [11,12]. We extend that algorithm to break such a dependence of a dependence cycle in a nest of loops. The algorithm is described below.

Algorithm 1. The Breaking Strategy of Link Pattern I

Input:

(1) A nest of loops $L = \{l_1, l_2, \ldots, l_n\}$.
(2) A set of statements $S = \{S_0, S_1, \ldots, S_{n-1}\}$ that are involved in a dependence cycle.
(3) The dependence link from S_a to S_b is an anti-dependence link.
(4) A direction vector matrix $D = (\vec{d}_1, \vec{d}_2, \ldots, \vec{d}_p)$.

Output: Generate the new dependence links not to form a new dependence cycle.

Method:
/*
Let S_a^{src} and $S_b^{\sin k}$ represent the source and $\sin k$ variables in the dependence link from S_a to S_b, respectively.
Let $\vec{d}_b$ represent the direction vector from S_a to S_b.
Let S_a^{index} and S_b^{index} represent the indexed expression of the source and $\sin k$ variables in the dependence link from S_a to S_b, respectively.
*/

1. If one of elements in a direction vector matrix, D, includes one direction vector '>', then the transformation is exited and all of the statements are preserved. Otherwise, the two variables Ω and index represent the $\sin k$ variable name and the indexed expression of the $\sin k$ variable in anti-dependence between one statement S_a and another statement S_b, respectively. Simultaneously, one Boolean variable is set to a false value.
2. Determine which variables is true-dependent on the $\sin k$ variable Ω in the anti-dependence. If a variable $S_{\mathrm{d}}^{\sin k}$ in a statement S_{d} is true-dependent on the $\sin k$ variable Ω and the variable $S_{\mathrm{d}}^{\sin k}$ cannot be the $\sin k$ variable of other true-dependences, then the name of the variable $S_{\mathrm{d}}^{\sin k}$ is changed and the Boolean variable is allocated to a true value. If the variable $S_{\mathrm{d}}^{\sin k}$ is the $\sin k$ variable of another true-dependence, then such a transformation is exited and all of the statements are preserved.

Table 1
The performance of the proposed methods to loops tested in Vector loops

Benchmark	Loop name	Scalar mode (s)	Parallel mode (s)	Speed-up
Vector loops	S231	3.485	0.085	41
Vector loops	S232	4.656	0.097	48
Vector loops	S221	5.559	0.109	51
Vector loops	S212	6.435	0.117	55
Vector loops	S211	6.897	0.121	57
Vector loops	S243	9.258	0.149	62
Vector loops	S241	8.875	0.136	65
Vector loops	S244	11.243	0.17	66
Vector loops	S222	14.697	0.213	69

than that of the transformed codes. For all of the subroutines in our experiments, the execution time of the original programs was indicated to take from 41 to 69 times longer than the execution time of the transformed programs. This indicates that the proposed scheme is very significant in term of speed-up, ranging from 41 to 69.

5. Conclusion

Parallelism exploitation for statements with dependence cycles in a nest of loops is necessary. The vectorizability of a dependence cycle depends primarily on its dependence links. There exist seven possible dependence relations for one statement on another statement. A dependence cycle with an anti-dependence link is breakable via node splitting [17]. In case the source variable for the anti-dependence relation is itself the sin$k$ variable of another true-dependence, a corrective strategy should be incorporated into the node splitting algorithm. An output-dependence link or an anti- and output-dependence link in a dependence cycle is breakable via a sin$k$ variable renaming technique. This technique is also applicable to break an anti-dependence link. In practice, node splitting and sin$k$ variable renaming techniques should be utilized in a complementary manner. For a dependence cycle formed only by links of true-dependence or all other dependences, a general, simple, but less-efficient partial vectorization algorithm is available. To improve the efficiency, a dynamic dependence concept and its derived technique are powerful. This approach is particularly efficient to deal with a simple dependence cycle. All of the recurrence-resolving strategies can be integrated with the dependence testing technique, Tarjan's depth-first search algorithm, and a topological sorting to develop an automatic recurrence-resolving system.

References

[1] R. Allen, K. Kennedy, Automatic translation of Fortran program to vector form, ACM Transactions on Programming Languages and Systems 9 (4) (1987) 491–542.
[2] U. Banerjee, Dependence, Analysis, Kluwer Academic Publishers, Norwell, MA, 1997.
[3] U. Banerjee, Loop, Parallelization, Kluwer Academic Publishers, 1994.
[4] U. Banerjee, Loop, Transformation for Restructuring, Compilers: The Foundations, Kluwer Academic Publishers, 1993.
[5] W. Blume, R. Eigenmann, Nonlinear and symbolic data dependence testing, IEEE Transaction on Parallel and Distributed Systems 9 (12) (1998) 1180–1194.
[6] P.-Y. Calland, A. Darte, Y. Robert, Plugging anti and output dependence removal techniques into loop parallelization algorithm, Parallel Computing 23 (1997) 251–266.
[7] W.-L. Chang, C.-P. Chu, The extension of the interval test, Parallel Computing 24 (14) (1998) 2101–2127.
[8] W.-L. Chang, C.-P. Chu, The generalized direction vector I test, Parallel Computing 27 (8) (2001) 1117–1144.
[9] W.-L. Chang, C.-P. Chu, J. Wu, The generalized lambda test: a multi-dimensional version of Banerjee's algorithm, International Journal of Parallel and Distributed Systems and Networks 2 (2) (1999) 69–78.
[10] W.-L. Chang, C.-P. Chu, The infinity lambda test: a multi-dimensional version of Banerjee infinity test, Parallel Computing 26 (10) (2000) 1275–1295.
[11] C.-P. Chu, D.L. Carver, An analysis of recurrence relation in Fortran do-loops for vector processing, in: Proc. Fifth Parallel Processing Symp, IEEE CS Press, Los Alamities, CA, 1991, pp. 619–625.

[12] C.-P. Chu, D.L. Carver, Exploitation of parallelism in Fortran do-loops for vectoring processing, Technical Report 91-004, Department of Computer Science, LSU, 1991.
[13] MasPar Group, Parallel, Programming, language, Digital Equipment Corporation, Part number: AA-PV4QA-TE, 1993.
[14] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, M. Wolfe, Dependence graphs and compiler optimizations, in: Conference Record of 8th ACM Symposium on Princ. of Prog. Lang., 1981, pp. 207–218.
[15] D. Levine, D. Callahan, J. Dongarra, A comparative study of automatic vectorizing compilers, Parallel Computing 17 (1991) 1223–1244.
[16] D.E. Kunth, The, Art, of, Computer, Programming, vol. 1: Fundamental Algorithm, Addison-Wesley, Reading, MA, 1973.
[17] C.D. Polychronopoulos, Advanced loop optimizations for parallel computers, in: Proceedings of the 1987 International Conference on Supercomputing, 1987, pp. 255–277.
[18] W. Shang, M. O'Keefe, J. Fortes, On loop transformation for generalized cycle shrinking, IEEE Transaction on Parallel and Distributed Systems 5 (2) (1994) 193–204.
[19] R. Tarjan, Depth first search and linear graph algorithms, SIAM Journal on Computing 1 (2) (1972) 146–160.
[20] M. Wolfe, High, Performance, Compilers, for, Parallel, Computing, Addison-Wesley Publishing Company, Redwood City, CA, 1996.
[21] H. Zima, B. Chapman, Supercompilers for Parallel and Vector Computers, Addision-Wesley Publishing Company, 1991.
[22] W.-L. Chang, C.-P. Chu, J.-H. Wu, A precise dependence analysis for multi-dimensional arrays under specific dependence direction, The Journal of Systems and Software 63 (2) (2002) 99–107.
[23] W.-L. Chang, C.-P. Chu, J. Wu, A multi-dimensional version of the I test, Parallel Computing 27 (13) (2001) 1783–1799.
[24] W.-L. Chang, C.-P. Chu, J.-H. Wu, A polynomial-time dependence test for determininginteger-valued solutions in multi-dimensional arrays under variable bounds, Journal of Supercomputing, in press.
[25] M. Guo, W.-L. Chang, J. Cao, The non-continuous direction vector i test, accepted on the International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN2004), Hong Kong, 2004.
[26] W.-L. Chang, J.-W. Huang, C.-P. Chu, The non-continuous i test: an improved dependence test for reducing complexity of source level debugging for parallel compilers, in: The Third International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'02), Kanazawa Bunka Hall, Kanazawa, Japan, 4–6 September 2002, pp. 455–462.
[27] W.-L. Chang, B.-H. Chen, A proof method for the correctness of the interval test to be applied for determining whether there are integer-valued solutions for one-dimensional arrays with subscripts formed by induction variable, in: The Third International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'02), Kanazawa Bunka Hall, Kanazawa, Japan, 4–6 September 2002, pp. 52–57.
[28] A. Bik, M. Girkar, P. Grey, X. Tian, Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems, Intel Technology Journal Q1 (March) (2001) 1–9.
[29] D. Petkov, R. Harr, S. Amarasinghe, Efficient pipelining of nested loops: unroll-and-squash, in: 16th International Parallel and Distributed Processing Symposium, Fort Lauderdale, Florida, April, 2002.
[30] W.-L. Chang, C.-P. Chu, J.-H. Wu, A simple and general approach to parallelize loops with arbitrary control flow and uniform data dependence distance, The Journal of Systems and Software 63 (2) (2002) 91–98.

Weng-Long Chang received his Ph.D. degrees in Computer Science and Information Engineering from National Cheng Kung University, Taiwan, Republic of China, in 1999. He is currently an Assistant Professor at the Department of Information Management, Southern Taiwan University of Technology, Tainan County 710, Taiwan, Republic of China. Dr. Chang's research interests include molecular computing (including bio-molecular cryptography, bio-molecular computational complexity and bio-molecular computational theory), parallel and distributed processing and parallelizing compilers.

Chih-Ping Chu received a B.S. degree in agricultural chemistry from National Chung Hsing University, Taiwan, an M.S. degree in computer science from the University of California, Riverside, and a Ph.D. degree in computer science from Louisiana State University. He is currently a professor in the Department of Computer Science and Information Engineering of National Cheng Kung University, Taiwan. His research interests include parallelizing compilers, parallel computing, parallel processing, internet computing, and software engineering.

Michael (Shan-Hui) Ho, Associate Professor of Southern Taiwan University of Technology, has 25 years industrial and academic experience in the computing field. He had worked as a Sr. Software Engineer and Project Manager developing B2B/B2C/C2C web and multimedia applications and a Sr. database administrator for SQL clustered servers, Oracle, and DB2 databases including network/systems LAN/WAN system administration to US major corporations and government organizations such as the GAS Company, Fox, Fidelity, ARCO, US LA Hall Records,...etc. He is also certified in Oracle DBA/Developer (OCP), CCNA/CCNP, MCSE, Unix Solaris systems SA, and Dell server/computer technician engineer. Dr. Ho had more than 10 years of college teaching/research experience as an Assistant Professor and Research Analyst at Central Missouri State University, the University of Texas at Austin, and BPC International Institute. He earned a Ph.D. in IS/CS with minors Management and Accounting from the University of Texas at Austin and a M.S. degree from St. Mary's University. His research interests include algorithm and computation theories, software engineer, database and data mining, parallel computing, quantum computing and DNA computing.