Compiling Techniques

Using Knowledge-Based Techniques for Parallelization on Parallelizing Compilers*

Chao-Tung Yang, Shian-Shyong Tseng,
Cheng-Der Chuang, and Wen-Chung Shih

Dept. of Computer and Information Science, National Chiao Tung Univ.
Hsinchu, Taiwan 300, Republic of China
Phone: 886-35-715900, Fax: 886-35-721490
E-mail: ctyang@aho.cis.nctu.edu.tw
E-mail: sstseng@cis.nctu.edu.tw

Abstract. In this paper we propose a knowledge-based approach for solving data dependence testing and loop scheduling problems. A rule-based system, called the K test, is developed by repertory grid and attribute ording table to construct the knowledge base. The K test chooses an appropriate testing algorithm according to some features of the input program by using knowledge-based techniques, and then applies the resulting test to detect data dependences for loop parallelization. Another rule-based system, called the KPLS, is also proposed to be able to choose an appropriate scheduling by inferring some features of loops and assign parallel loops on multiprocessors for achieving high speedup. The experimental results show that the graceful speedup obtained by our compiler is obvious.

1 Introduction

Parallelizing compilers [1, 2, 14, 18, 21] analyze sequential programs to detect hidden parallelism and use this information for automatic restructuring of sequential programs into parallel subtasks on multiprocessors by using loop scheduling algorithms [7, 13, 17]. In particular, loops are such a rich source of parallelism that their parallelization would lead to considerable improvement of efficiency on multiprocessors [14, 21]. Therefore, we investigate the possibility of solving the problem on two fundamental phases, data dependence testing and parallel loop scheduling on loops, in parallelizing compilers.

In brief, the *data dependence testing* problem is that of determining whether two references to the same array within a nest of loops may reference to the same element of that array [5, 12, 10, 15]. Traditionally, this problem has been formulated as *integer programming*, and the best integer programming algorithms are $O(n^{O(n)})$ where n is the number of loop indices. Obviously, these algorithms are too expensive to use. For this reason, a faster, but not necessarily exact, algorithm might be more desirable in some situations. In this paper, we propose

* This work was supported in part by National Science Council of Republic of China under Grants No. NSC83-0408-E-009-034 and NSC84-2213-E-009-090.

a new approach by using knowledge-based techniques for data dependence testing [3]. A rule-based system, called the *K test* [16], is developed by repertory grid and attribute ordering table to construct the knowledge base. The K test can choose an appropriate test according to some features of the input program by using knowledge-based techniques [8], and then apply the resulting test to detect data dependences on loops for parallelization. Furthermore, as for system maintenance and extensibility, our approach is obviously superior to others, for example, if a new testing algorithm or testing technique is proposed, then we can integrate it into the K test easily by adding knowledge base and rules.

Another fundamental phase, parallel loop scheduling, is a method that schedules the parallel loops on multiprocessors. In a shared-memory multiprocessors, scheduling decision can be made either statically at compile time or dynamically at runtime. Traditionally, the parallelizing compiler dispatches the loop by using only one scheduling algorithm, maybe static or dynamic. However, a program has the different kind of loops including uniform workload, increasing workload, decreasing workload, and random workload, every scheduling algorithm can achieve good performance on different loop styles and system status [17]. To reduce the overhead and enhance the load balancing, the knowledge-based approach becomes another solution to parallel loop scheduling. In this paper, another rule-based system, named *Knowledge-Based Parallel Loop Scheduling* (KPLS), is also developed by repertory grid analysis, which can choose an appropriate scheduling according to some features of loops and system status, and then apply the resulting algorithm to assign parallel loops on multiprocessors for achieving high speedup. The experimental results show that the graceful speedup obtained by using KPLS in our compiler is obvious.

2 Background

2.1 A Review of Data Dependence Testing

A *data dependence* is said to exist between two statements S_1 and S_2 if there is an execution path from S_1 to S_2, both statements access the same memory location and at least one of the two statements writes the memory location [5]. There are three types of data dependences:

- *True (flow) dependence* (δ) occurs when S_1 writes a memory location that S_2 later reads.
- *Anti-dependence* ($\bar{\delta}$) occurs when S_1 reads a memory location that S_2 later writes.
- *Output dependence* (δ_o) occurs when S_1 writes a memory location that S_2 later writes.

Data dependence testing is the method used to determine whether dependences exist between two subscript references to the same array in a nested loop. The index variables of the nested loop are normalized to increase by 1. Suppose that we want to decide whether or not there exists a dependence from

statement S_1 to S_2. Let $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ and $\beta = (\beta_1, \beta_2, \ldots, \beta_n)$ be the integer vectors of n integer indices within the range of the upper and lower bounds of the n loops. There is a *dependence* form S_1 to S_2 if and only if there exist α and β, such that α is lexicographically less than or equal to β and the dependence equations are satisfied as $f_i(\alpha) = g_i(\beta)$, for $1 \leq i \leq m$. In this case, we say that the system of equations is *integer solvable* with the loop-bounds constraints. Otherwise, the two array reference patterns are said to be *independent*.

The data dependence tests can be classified into three classes: single dimensional tests (e.g., *GCD Test*, *Banerjee Test* [21], and *I Test* [10]), multiple dimensional tests, (e.g., *Extended-GCD Test* [1], *λ Test* [11], *Power Test* [19], and *Ω Test* [15]), and classification tests [5, 12]. We find these two papers [5, 12] are similar to our approach (*K Test*) in some aspects. Both of them collect a small set of test algorithms, and try to use them to solve the problem both efficient and exact in practical cases. However, our approach is different from theirs in essence.

- *Practical Test* [5]: The test is based on classifying pairs of subscripted variable references. The major difference between the Practical test and our approach is that the Practical test is essentially designed for practical input cases, and its strategy is fixed. However, our approach is not constraint to some kind of input cases.
- *MHL Test* [12]: The major difference between the MHL test and our approach is that the MHL test is a cascaded method; that is, the Extended-GCD test is tried first; if it fails, the next test is applied, and so on. However, our approach uses only the appropriate after the conclusion is drawn.

2.2 A Review of Parallel Loop Scheduling

A loop is called as a DOALL[1] loop if there is no data dependence among all iterations. *Parallel loop scheduling* is used to assign a DOALL loop into each processor as even as possible. There are two kind of parallel loop scheduling strategies which can be made either statically at compile time or dynamically at runtime. *Static scheduling* may be applied when the loop iterations take roughly the same amount of execution time, and the compiler must know how many iterations are run in advance. *Dynamic scheduling* adjusts the schedule during execution, so we use it whenever it is uncertain how many iterations to be run, or each iteration takes different amount of execution time, due to a branch statement inside the loop. Dynamic scheduling is more suitable for load balancing between processors, but the runtime overhead and memory contention must be considered. Up to now there are several loop scheduling algorithms, for example, *SS*, *GSS* [13], *CSS*, *Factoring* [7], and *TSS* [17]. We use N and P to denote the number of iterations and the number of processors, respectively. Assume the size of i_th partition is K_i. The formulas for the calculation of K_i in different algorithms are listed in Table 1, where the CSS/k algorithm partitions the DOALL loop into k equal-sized chunks.

[1] Iterations can be executed in any order or even simultaneously

Scheme	Formulas
SS	$K_i = 1$
CSS(k)	$K_i = k$
CSS/k	$K_i = \lceil \frac{N}{k} \rceil$
GSS	$K_i = \lceil \frac{R_i}{P} \rceil$, $R_0 = N$, $R_{i+1} = R_i - K_i$
Factoring	$K_i = (\frac{1}{2})^{\lceil \frac{i}{P} \rceil} \frac{N}{P}$
TSS(f, l)	$K_i = f - i\delta$, $I = \lceil \frac{2N}{f+l} \rceil$, $\delta = \frac{f-l}{I-1}$

Table 1. Various loop scheduling algorithms.

3 Using Knowledge-Based Techniques for Data Dependence Testing

3.1 Knowledge-Based Approach

Knowledge-based systems are systems that depend on a vast base of knowledge to perform difficult tasks. The knowledge is saved in a knowledge base separately from the inference component. This makes it convenient to append new knowledge or update existing knowledge without recompiling the inferring programs. The *rule-based* approach is one of the commonly used form in many knowledge-based systems. The primary difficulty in building a knowledge base is how to acquire the desired knowledge. To ease acquisition of knowledge, one primary technique among them is *Repertory Grid Analysis* (RGA) [9]. RGA is easy to use, but it suffers from the problem of *missing embedded meanings* [8]. For example, when a doctor expresses the features of catching a cold are headache, cough and sneeze, he means if a person catches a cold, he may has those features. However, in RGA, a person is not considered to catch a cold except that he gets all of the features. To overcome the problem, the concept of *Attribute Ordering Table* (AOT) is employed to elicit embedded meanings by recording the importance of each attribute to each object [8].

3.2 The Anatomy of the K Test

The processes of knowledge-based data dependence testing can be described as follows. First, the input, a set of dependence equations, is fed into the inference component. Then, the inference component reasons about knowledge and draw a conclusion, a test. Finally, the resulting test is applied to detect dependence relations for loop parallelization, and generate the answer whether the loop is parallelizable or not. An implementation, called the *K test*, is proposed to demonstrate the effectiveness of the new approach. The K test is a rule-based system. The primary reason we choose a rule-based system is that this type of system is easy to understand; in addition, rule-based inference tools are widely available, which simplify the work of implementation.

The organization of the K test is shown in Figure 1 that the three components are replaced by actual software. We describe them briefly.

- **Knowledge Base:** The knowledge base is constructed as a rule base, i.e., the knowledge is expressed in the form of production rules. These rules can be coded by hand or generated by a translator. In our K test, the latter is adopted. A translator, GRD2CLP, is utilized to translate the repertory grid and attribute ordering table to CLIPS's production rules.
- **Inference Component:** An expert system shell, called CLIPS [4], is used as the inference component. CLIPS, a forward reasoning rule-based tool, is very efficient, and does not increase the execution time of the K test too much.
- **Testing Algorithm Library:** We include four tests in the library. There are GCD test, Banerjee test, I test and Power test for solving the data dependence problem.

It should be noted that the knowledge base and the testing algorithm library shown in Figure 1 are flexible; that is, they are not fixed. You can modify these two components so long as the efficiency and precision of the system are retained. The repertory grid of the K test contains four attributes and four objects which are four existing data dependence tests. The four attributes of the K test are described below:

- Unity_Coef: whether the coefficients of variables are 1, 0, or -1 or not.
- Bound_Known: whether the loop bounds are known or not.
- Multi_Dim: whether the array reference is multi-dimensional or not.
- Few_Var: whether the number of variables in the equation is small or not.

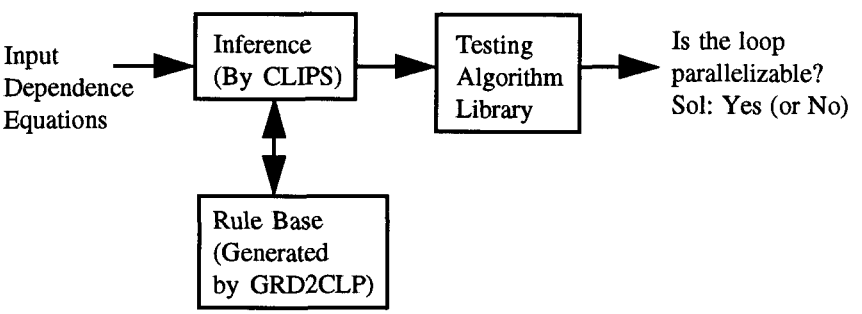

Fig. 1. Components of the K test.

In order to elicit the embedded meanings of RGA of the K test, we construct the AOT. The RGA/AOT of the K test is shown in Table 2. The process is described in dialog form. For example,
Q: If Bound_Known is not equal to 5, is it possible for the Banerjee test to be applied?

	GCD	Banerjee	I	Power
Unity_Coef	1/1	5/2	1/1	1/1
Bound_Known	1/2	5/D	1/1	5/2
Multi_Dim	1/1	1/1	1/2	5/2
Few_Var	5/1	5/1	1/2	1/2

Table 2. The RGA/AOT of the K test.

A: No.
The answer means that Bound_Known dominates the Banerjee test, and hence AOT[Bound_Known, Banerjee]='D'. In AOT, large integer number implies the attribute being more important to the object (e.g. 2¿1).

3.3 The Algorithm of the K Test

We now summarize the discussion of the K test into an algorithm. The algorithm consists of two phases.
Algorithm: K test
Input:

$(a_0^1, a_1^1, \cdots, a_n^1, M_1^1, N_1^1, \cdots, M_n^1, N_n^1,$
$\cdots$
$a_0^m, a_1^m, \cdots, a_n^m, M_1^m, N_1^m, \cdots, M_n^m, N_n^m,$
Unity_Coef, Bound_Known, Multi_Dim, Few_Var)

Output:

True: the input is integer solvable.
or False: the input is not integer solvable.
or Maybe: the input may be integer solvable.

Phase 1: calling CLIPS to draw a conclusion, that is, the most suitable dependence test.
Phase 2: calling the corresponding testing algorithm to check for data dependence.

4 Using Knowledge-Based Techniques for Parallel Loop Scheduling

If the parallelizing compiler can analysis a loop's attributes such as loop style, loop bound, data locality, etc.; then the suitable scheduling algorithms for the particular case should be applied. This leads to select scheduling algorithms by using knowledge-based approach. The processes of knowledge-based loop scheduling method can be described as follows. First, the compiler can get some

attributes about a loop by parsing input program. Then, the inference component reasons about knowledge and draw a conclusion, a parallel loop scheduling. Finally, the resulting scheduling is applied for loop partition.

4.1 The Anatomy of KPLS

In this session, we describe our new method, named *Knowledge-Based Parallel Loop Scheduling* (KPLS). We propose this method using *knowledge-based*, because it is easy to understand, implementation, maintenance and extension. This approach has great flexibility as we can add new algorithms to the repertory grid and attribute ordering table, and then use the conversion tool to convert tables to CLIPS rules. We do not need any modification in CLIPS source.

We describe the components of KPLS briefly in the following:

- **Knowledge Base:** The knowledge base is constructed as a rule base, i.e., the knowledge is expressed in the form of production rules. We also use GRD2CLP to translate the repertory grid and attribute ordering table to CLIPS's production rules.
- **Inference Component:** CLIPS is used as the inference component, which is very efficient for inferring, and does not increase the execution time of our KPLS too much.
- **Scheduling Algorithm Library:** There are six scheduling algorithms in the library including static scheduling, SS, CSS, GSS, Factoring, and TSS. It is also the advantage of expert system; whenever, we can easily modify the rules and adding the new scheduling strategy flexibly.

The repertory grid and attribute ordering table of KPLS are shown in Table 3. 'X' means that the attribute has no relation with the object. There are six algorithms and five attributes in both tables. We describe these attributes as follows:

- Loop_Style: means the different styles of loop (1:uniform workload, 2:increasing workload, 3:decreasing workload, or 4:random workload).
- Start_Time: means whether the starting time of each processors is equal or not, influencing the execution time of loop.
- Loop_Bound: means whether the loop bounds are known or not in compile time.
- Overhead: means the different overhead of synchronization primitives on system (0:none, 1:little, 2:fair or 3:high).
- Easy: means whether the implementation of algorithm is easy or not.

4.2 The Algorithm of the KPLS

In this section, we describe our algorithm of KPLS. The algorithm consists of three phases.

Algorithm: KPLS

Input:

The following information can be obtained from input file.

	Static	SS	CSS	GSS	TSS	Factoring
Loop_Style	{1}/D	X/X	{1}/D	{2,4}/D	X/X	X/X
Start_Time	YES/D	X/X	YES/D	X/X	X/X	X/X
Loop_Bound	YES/D	X/X	NO/D	X/X	X/X	X/X
Overhead	X/X	{0}/D	{1,2,3}/D	{1}/2	{2,3}/1	{2,3}/1
Easy	X/X	X/X	X/X	NO/2	X/X	NO/2

Table 3. The RGA/AOT of the KPLS.

1. What kind of loop style? (1-4 styles)
2. Are the start time of processors roughly equal? (Yes/No)
3. Is the loop bound known during compiler time? (Yes/No)
4. What is the synchronization overhead level? (None/Low/Fair/High)
5. Use easy-to-implement methods only? (Yes/No)

A *certainty factor* (CF) [8] value for each question to express the question's importance is given.

Output:

What kind of loop scheduling strategy will be applied. If there are more than one suggestion, the one with maximal CF value will be chosen.

Phase 1: Get the loop attributes from parallelism detector.

Phase 2: Call CLIPS to draw a conclusion by using rules; that is, the most suitable loop scheduling method.

Phase 3: S2m [6] uses the appropriate loop scheduling to partition the DOALL loop on multiprocessors.

5 Experiments

5.1 Integrating K Test and KPLS in PFPC

We integrate K test and KPLS in PFPC to generate the efficient object codes for multiprocessors. The K test is used to treat the data dependence relations and then restructure a sequential FORTRAN source program into a parallel form, i.e., if a loop can be parallelized, then parallelism detector (K test) converts it into DOALL loop. In the previous version of our compiler, Parafrase-2 (p2fpp) is used to treat the data dependence analysis [20, 6]. For improving the capacity of loop partition module in s2m, the KPLS is used, instead of the previous version, to build an intelligent loop scheduling method. Because the restriction of the OS scheduling, the system call for binding any thread onto the appropriate processor is not avaliable and only dynamic scheduling is employed. We can only partition loops and encapsulate with data into threads by s2m and let the OS dynamically choose the threads to run on multiprocessors. The experiments only concerned the performance of KPLS in PFPC.

5.2 Experimental Programs

We show the performance gained by using our parallelizing compiler on the following four examples which are different styles of loops.

- The first example is matrix multiplication, the outer two loops can be parallelized. Since the example is highly load balanced, this kind of loop is called *uniform workload*. The matrix size is 600×600.
- The second example is adjoint convolution which exploits significant load imbalance. This kind of loop is called *decreasing workload*. We choose the problem size to be 150×150.
- The third example is reverse adjoint convolution which also exploits significant load imbalance. This kind of loop is called *increasing workload*. The problem size is 150×150.
- The fourth example is transitive closure. The characteristic of this program is that the workload is dependent on the input data. This kind of loop is called *random workload*. We select the different matrix sizes for testing alone and combining four programs by using 1000×1000 and 500×500, respectively.

5.3 Experimental Results

There are two parts of the experiments: the first part concerns each execution time and speedup of above four programs, and the other is a combined program, including those four programs. The execution time of four programs is shown in Table 4, and the corresponding speedup in Figure 2 shows that GSS performs poorly when iterations have an decreasing workload like adjoint convolution. CSS/4 is suitable for the uniformly workload like matrix multiplication, and Factoring is suitable for reverse adjoint convolution. Among those scheduling algorithms, none of them is suitable for every case. KPLS can choose an appropriate scheduling and have good results for all programs except transitive closure. In the case of transitive closure, our approach does not choose the fastest one, CSS/4, but chooses TSS, because the imbalance workload in this program is not so obvious, because the control flow is related to the input data, and because the matrix size is 500×500, which is divided exactly by the number of CPUs. On most cases, KPLS makes a better choice than other scheduling even in single loop.

Table 5 shows the experimental execution time and Figure 3 shows the corresponding speedup for the big program integrated from the above four programs. Traditionally, every scheduling algorithm uses only one method through the entire program. But the KPLS can always choose an appropriate scheduling algorithm according to the behaviors of the loop among one program. Like the second part experiment, KPLS chooses different style of loop scheduling for each loop in the combined program. For example, according to the loop behaviors, KPLS selects TSS for the adjoint convolution part of the combined program and CSS/4 for matrix multiplication, instead of only one scheduling method. We concern about the runtime cost of loops. During the execution time, it's important to

select a good loop scheduling algorithm by considering about the runtime cost; once compiler specifies the right loop scheduling, program can save the time on execution. Furthermore, for the inference cost of knowledge-based approach, the KPLS only spends a little time. Since no single scheduling algorithm performs well across all applications, our method can now give a good solution to make a compiler more flexible and efficient in loop scheduling.

	Serial	CSS/1	CSS/4	CSS/8	CSS/16	GSS	Factoring	TSS	KPLS
Mat_Mul	854.46	848.18	225.96	230.92	232.73	250.94	259.32	230.74	as CSS/4
Adj_Conv	769.50	776.38	350.80	256.70	239.90	369.54	245.01	237.68	as TSS
Rev_Adj	604.51	632.37	281.01	215.19	171.63	182.01	168.93	177.57	as Factoring
Trans_Col	2310.96	2221.81	695.73	708.83	702.60	730.30	713.62	710.68	as TSS

Table 4. The execution time of four different kind of programs (sec).

	Serial	CSS/1	CSS/4	CSS/8	CSS/16	GSS	Factoring	TSS	KPLS
All	2185.59	2167.99	872.34	683.85	636.38	763.67	651.52	644.43	582.47

Table 5. The execution time of combined program (sec).

6 Conclusions and Further Directions

In this paper we have proposed a new approach by using knowledge-based techniques, which integrates existing data dependence testing algorithms and loop scheduling algorithms to make good use of their advantages for loop parallelization. A rule-based system, called the K test, was developed by RGA and AOT to construct the knowledge base. The K test could choose an appropriate testing algorithm by knowledge-based techniques, and then apply the resulting test to detect data dependences on loops. Another rule-based system, called the KPLS, was also developed by RGA and AOT, which was embedded in our s2m, that could choose an appropriate scheduling and then apply the resulting algorithm for assigning parallel loops on multiprocessors to achieve high speedup. The experiments have shown that the KPLS can apply more suitable loop scheduling strategy. Once we choose the right method for loop scheduling, the program can save more execution time. The experimental results also have shown that the graceful speedup obtained by our compiler is obvious. Furthermore, as for system maintenance and extensibility, our approach is obviously superior to others. In addition, we are going to study whether knowledge-based approaches may be applied to guide the wide variety of loop transformation for parallelization in parallelizing compilers.

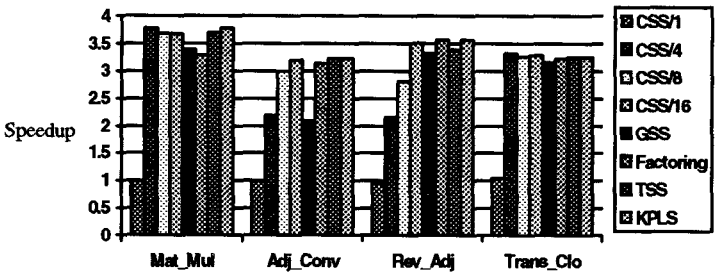

Fig. 2. The speedup for different kind of programs.

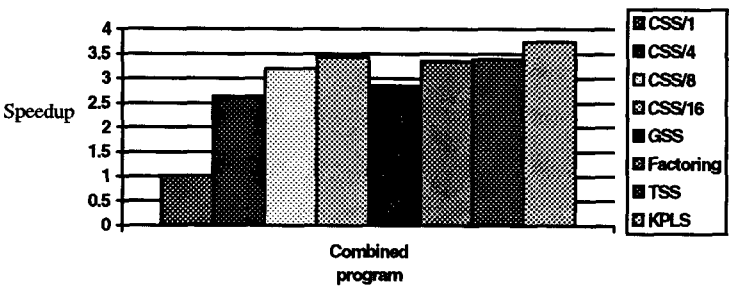

Fig. 3. The speedup for the combined of four program.

Acknowledgments

We would like to thank the anonymous reviewers for suggesting of improvements, and offering of encouragements.

References

1. U. Banerjee, *Dependence Analysis for Supercomputing*, Norwell, Kluwer Academic Publishers, MA, 1988.
2. U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, "Automatic program parallelization," *Proc. IEEE*, 81(2):211-243, Feb. 1993.
3. B. M. Chapman and H. M. Herbeck, "Knowledge-based parallelization for distributed memory systems," *Parallel Computing, in Proc. of the First International ACPC Conference*, vol. 591, pp. 77-89, Springer-Verlag, Salzburg, Austria, 1991.
4. J. C. Giarratano and G. Riley, *Expert Systems: Principles and Programming*, PWS-Kent Publishing Company, Boston, 1993.

5. G. Goff, K. Kennedy, and C. W. Tseng, "Practical dependence testing," in *Proc. of the ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, Toronto, Canada, pp. 15-29, June 1991.
6. M. C. Hsiao, S. S. Tseng, C. T. Yang, and C. S. Chen, "Implementation of a portable parallelizing compiler with loop partition," in *Proc. 1994 ICPADS*, Hsinchu, Taiwan, R.O.C. pp. 333-338, Dec. 1994.
7. S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A method for scheduling parallel loops," *Commun. ACM*, 35(8):90-101, Aug. 1992.
8. G. J. Hwang and S. S. Tseng, "EMCUD: A knowledge acquisition nethod which captures embedded meanings under uncertainty," *Int. J. Man-Machine Studies.*, vol. 33, pp. 431-451, 1990.
9. G. A. Kelly, *The Psychology of Personal Constructs*, vol. 1, New York, NY: W. W. Norton, 1955.
10. X. Kong, D. Klappholz, and K. Psarris, "The i test: An improved dependence test for automatic parallelization and vectorization," *IEEE Trans. Parallel Distrib. Syst.*, 2(3):342-349, July 1991.
11. Z. Li, P. C. Yew, and C. Q. Zhu, "An efficient data dependence analysis for parallelizing compilers," *IEEE Trans. Parallel Distrib. Syst.*, 1(1):26-34, Jan. 1990.
12. D. E. Maydan, J. L. Hennessy, and M. S. Lam, "Efficient and exact data dependence analysis," in *Proc. of the ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, Toronto, Canada, pp. 1-14, June 1991.
13. C. D. Polychronopoulos and D. J. Kuck, "Guided self-scheduling: A practical self-scheduling scheme for parallel supercomputers," *IEEE Trans. Comput.*, 36(12):1425-1439, Dec. 1987.
14. C. D. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, MA, 1988.
15. W. Pugh, "A practical algorithm for exact array dependence analysis," *Commun. ACM*, 35(8):102-114, Aug. 1992.
16. W. C. Shih, C. T. Yang, and S. S. Tseng, "Knowelwdge-based data dependence testing on loops," in *Proc. 1994 Int. Computer Symposium*, Hsinchu, Taiwan, R.O.C. pp. 961-966, Dec. 1994.
17. T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers," *IEEE Trans. Parallel Distrib. Syst.*, 4(1):87-98, Jan. 1993.
18. M. Wolfe, *Optimizing Supercompilers for Supercomputers*. Pitman Publishing Co. London, and MIT Press, MA, 1989.
19. M. Wolfe and C. W. Tseng, "The power test for data dependence," *IEEE Trans. Parallel Distrib. Syst.*, 3(5):591-601, Sep. 1992.
20. C. T. Yang, S. S. Tseng, and C. S. Chen, "The anatomy of parafrase-2," *Proceedings of the National Science Council Republic of China (Part A)*, 18(5):450-462, Sep. 1994.
21. H. P. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, Addison-Wesley Publishing, New York, ACM Press, 1990.