A Static Data Dependence Analysis Approach for Software Pipelining

Lin Qiao, Weitong Huang, Zhizhong Tang

Department of Computer Science and Technology, Tsinghua University, Beijing, 100084, PR China
{qiaolin, hwt}@cic.tsinghua.edu.cn; tzz-dcs@tsinghua.edu.cn

Abstract. This paper introduces a new static data dependence constraint, called dependence difference inequality, which can deal with coupled subscripts for multi-dimensional array references. Unlike direction vectors, dependence difference inequalities are related to not only the iteration space for a loop program but also the operation distance between two operations. They are more strict than other methods, and can act as additional constraints to each variable in a linear system on their own or with others. As a result, the solution space for a linear system can be compressed heavily. So long as dependence difference inequalities do not satisfy simultaneously, the loop can be software-pipelined with any initiation interval even if there exists a data dependence between two operations. Meanwhile, by replacing direction vectors with dependence difference inequalities some conservative estimations made by other traditional data dependence analysis approaches can be eliminated.

1 Introduction

Data dependence analysis plays an important role in automatic detection of implicit parallelism in programs written in conventional sequential languages. Dependence analysis techniques estimate, at compile-time, the run-time interactions between different operations or between different instances of the same operation [1].

It is at the core of data dependence analysis strategies to estimate data dependence between two operations in which multi-dimensional array references are involved. General speaking, the question of whether multi-dimensional array references with coupled linear subscripts can be parallelized depends upon the resolution of multi-dimensional array aliases. The resolution of multi-dimensional array aliases is to ascertain whether or not the two references to the same multi-dimensional array within a general [nested] loop can refer to the same element of that multi-dimensional array [2].

The paper focuses on a new data dependence analysis technique for an interlaced inner and outer loop software pipelining algorithm. Our approach, called dependence difference inequalities, can deal with coupled subscripts for multi-dimensional array references statically. This paper is organized as follows. Section 2 introduces related work and background, while Section 3 discusses dependence difference inequalities. Section 4 draws a conclusion.

2 Related Work and Background

This section introduces a novel software pipelining algorithm and gives a brief description of data dependence analysis techniques.

2.1 Interlaced Inner and Outer Loop Software Pipelining Algorithm

Software pipelining algorithms currently pursued in the world exploit the instruction-level parallelism of loop program by overlapping the operations of different loop bodies. Using software pipelining, a loop is transformed into a semantics-equivalent program consisting of a new loop, a prologue and an epilogue.

Several effective software pipelining algorithms have been presented to optimize innermost loops, such as Modulo Scheduling [3] and GURPR* [4]. In most cases, however, actual programs always contain nested loops. Optimization performance of existed algorithms is fairly insufficient when they are used to optimize nested loop programs, so it is the key to develop new algorithms that can efficiently optimize these nested loop programs.

Interlaced inner and outer Loop Software Pipelining (ILSP) is an efficient algorithm that can optimize operations in nested loops with various loop structures. In order to make the ILSP algorithm work efficiently and correctly, corresponding control mechanism, which combines software pipelining techniques with several hardware features, is introduced in [5]. In [6], Rong and his co-operators introduces a single-dimension software pipelining algorithm, which can be outlined as a brief version of the ILSP algorithm. Their algorithm chooses the most profitable loop level in the loop nest and software-pipelines it, which has been implemented as a tool set on an IA-64 Itanium workstation.

The ILSP algorithm is different from any traditional software pipelining algorithms of nested loops. ILSP does not execute the nested loops in the traditional sequence of completing the inner loop first and then executing the outer loop. It breaks the boundary of the different loop bodies of the nested loop, and can overlap the inner loop bodies of different outer loop bodies. Thus, ILSP makes it possible to optimize nested loops with various loop structures.

Consider the nested loop example as shown in Fig. 1. It can be performed well using the ILSP, as shown in Table 1.

By the example, we can describe the basic principle of the ILSP as follows. The ILSP is a software pipelining algorithm that is suitable for the nested loops with various loop structures. When each loop body of the nested loops is pipelined, the ILSP pipelines the nested loop as if the inner loops of this loop were executed only once. Whenever a new execution pattern made up of operations of an inner loop appears, in other words, the inner loop becomes active, the execution of the outer loop will be temporarily stopped. At this moment the software pipelining of the inner loop will be continuously executed until it is ready to enter its epilogue stage and to return to its outer loop or until another inner loop becomes active. When the inner loop begins to execute, the outer loop gives all its function units to the inner loop, and when the inner loop is completed, it will give all function units back to the outer loop, and the execution of the outer loop will be continued.

In one word, the foundation of the ILSP algorithm is to perform the nested loops as a whole. In order to form an effective pipeline along different loop bodies of the nested loop, it is very necessary to feed it with enough operations. A data dependence analysis approach has to meet the demand.

```
for( i=1; i<=n; i++ ){
  op1;
  for( j=1; j<=m; j++ ){
    op2;
    op3;
    op4;
  }
  op5;
}
```

(a) A Loop Program

$$\begin{tikzcd}
op_1 \arrow[d] \\
op_2 \arrow[d] \\
op_3 \arrow[d] \\
op_4 \arrow[d] \\
op_5
\end{tikzcd}$$

(b) Data Dependence Graph

Fig. 1. A nested loop example

Table 1. Execution result of the example

Clock	Operations					Comments
0	op$_1$(1,–)					Prologue of the outer loop begins
1	op$_1$(2,–)	op$_2$(1,1)				Prologue of the inner loop begins
2	op$_1$(3,–)	op$_2$(2,1)	op$_3$(1,1)			
3	op$_1$(4,–)	op$_2$(3,1)	op$_3$(2,1)	op$_4$(1,1)		The inner becomes active; switches to it
4		op$_2$(1,2)	op$_3$(3,1)	op$_4$(2,1)		The inner executes; the outer pauses
5		op$_2$(2,2)	op$_3$(1,2)	op$_4$(3,1)		
…	…	…	…	…		
3*m*		op$_2$(3,*m*)	op$_3$(2,*m*)	op$_4$(1,*m*)		The inner completes; system returns
3*m*+1	op$_1$(5,–)	op$_2$(4,1)	op$_3$(3,*m*)	op$_4$(2,*m*)	op$_5$(1,–)	The first epilogue of the inner begins
3*m*+2	op$_1$(6,–)	op$_2$(5,1)	op$_3$(4,1)	op$_4$(3,*m*)	op$_5$(2,–)	Epilogue of the outer begins
3*m*+3	op$_1$(7,–)	op$_2$(6,1)	op$_3$(5,1)	op$_4$(4,1)	op$_5$(3,–)	The second epilogue of the inner begins
3*m*+4		op$_2$(4,2)	op$_3$(6,1)	op$_4$(5,1)		The inner executes again; the outer pauses
…	…	…	…	…	…	
nm		op$_2$(*n*,*m*)	op$_3$(*n*–1,*m*)	op$_4$(*n*–2,*m*)		
nm+1			op$_3$(*n*,*m*)	op$_4$(*n*–1,*m*)	op$_5$(*n*–2,–)	Epilogue of the whole nested loop
nm+2				op$_4$(*n*,*m*)	op$_5$(*n*–1,–)	
nm+3					op$_5$(*n*,–)	

2.2 Data Dependence Analysis

Suppose op$_1$ and op$_2$ be two operations within a *n*-nested loop which refer to a *m*-dimensional array simultaneously. Each iteration of the loop is identified by an *iteration vector* whose elements are the values of the iteration variables for that iteration. We have

Definition 1. Let op$_1$($\boldsymbol{i}$) and op$_2$($\boldsymbol{j}$) respectively denote the instance of the operation op$_1$ during the iteration $\boldsymbol{i} = (i_1, i_2, \ldots, i_n)$ and that of the operation op$_2$ during the iteration $\boldsymbol{j} = (j_1, j_2, \ldots, j_n)$. There exists an *partial order* between the two operation instances, $\mathrm{op}_1(\boldsymbol{i}) < \mathrm{op}_2(\boldsymbol{j})$, if (*a*) for given an r where $1 \le r \le \min\{m, n\}$, $i_r = j_r$ and $i_k \le j_k$ $(1 \le k < r)$ hold, or (*b*) for $m \le n$, $i_r = j_r$ $(1 \le r \le m)$ holds.

Practically an partial order $\mathrm{op}_1(\boldsymbol{i}) < \mathrm{op}_2(\boldsymbol{j})$ means that $\mathrm{op}_1(\boldsymbol{i})$ precedes $\mathrm{op}_2(\boldsymbol{j})$. If the instance of the operation $\mathrm{op}_2(\boldsymbol{j})$ uses the element of the array defined first by the instance of the operation $\mathrm{op}_1(\boldsymbol{i})$, then $\mathrm{op}_2(\boldsymbol{j})$ is true-dependent or write-read-dependent on $\mathrm{op}_1(\boldsymbol{i})$. If the instance of the operation $\mathrm{op}_2(\boldsymbol{j})$ defines the element of the array used first by the instance of the operation $\mathrm{op}_1(\boldsymbol{i})$, then $\mathrm{op}_2(\boldsymbol{j})$ is anti-dependent or read-write-dependent on $\mathrm{op}_1(\boldsymbol{i})$. If the instance of the operation $\mathrm{op}_2(\boldsymbol{j})$ redefines the element of the array defined first by the instance of the operation $\mathrm{op}_1(\boldsymbol{i})$, then $\mathrm{op}_2(\boldsymbol{j})$ is output-dependent or write-write-dependent on $\mathrm{op}_1(\boldsymbol{i})$.

In general, suppose the n-nested loop have linear lower bounds and upper bounds, f_k denote the lower bound function for the k-th level nested loop, and g_k the upper bound function. It is obvious that $f_k \le i_k, j_k \le g_k$, $1 \le k \le n$, hold simultaneously. By replacing i_k with x_{2k-1} and j_k with x_{2k}, the problem mathematically can be reduced to that of checking whether or not a system of m linear equations with $2n$ unknown variables has a simultaneous integer solution, which satisfies the constraints for each variable in the system. The m linear equations in the system can be written as

$$\begin{cases} a_{1,0} + a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,2n}x_{2n} = 0 \\ a_{2,0} + a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,2n}x_{2n} = 0 \\ \cdots \\ a_{m,0} + a_{m,1}x_1 + a_{m,2}x_2 + \ldots + a_{m,2n}x_{2n} = 0 \end{cases} \tag{1}$$

where each $a_{i,j}$ is a constant integer for $1 \le i \le m$ and $1 \le j \le 2n$. $\boldsymbol{i} = (x_1, x_3, \ldots, x_{2n-1})$ and $\boldsymbol{j} = (x_2, x_4, \ldots, x_{2n})$ denote two iteration vectors, $\mathrm{op}_1(\boldsymbol{i})$ and $\mathrm{op}_2(\boldsymbol{j})$, respectively. Constraints to each variable in Eq. (1) can be represented as

$$\begin{cases} P_{1,0} \le x_1, x_2 \le Q_{1,0} \\ P_{k,0} + \sum_{s=1}^{k-1} P_{k,s}x_{2s-1} = f_k(x_1, x_3, \ldots, x_{2k-3}) \le x_{2k-1} \le g_k(x_1, x_3, \ldots, x_{2k-3}) = Q_{k,0} + \sum_{s=1}^{k-1} Q_{k,s}x_{2s-1},\ 2 \le k \le n \\ P_{k,0} + \sum_{s=1}^{k-1} P_{k,s}x_{2s} = f_k(x_2, x_4, \ldots, x_{2k-2}) \le x_{2k} \le g_k(x_2, x_4, \ldots, x_{2k-2}) = Q_{k,0} + \sum_{s=1}^{k-1} Q_{k,s}x_{2s},\ 2 \le k \le n \end{cases} \tag{2}$$

where $P_{r,0}$, $Q_{r,0}$, $P_{r,s}$, $Q_{r,s}$ are constant integers for $1 \le r \le n$. If each of $P_{r,s}$ and $Q_{r,s}$ is zero, the Eq. (2) will be reduced to

$$P_{k,0} \le x_{2k-1}, x_{2k} \le Q_{k,0},\ 1 \le k \le n\,. \tag{3}$$

That is, the bounds for each variables are constants.

Definition 2. A vector of the form $\boldsymbol{e} = (e_1, e_2, \ldots, e_n)$ is termed a direction vector from $\mathrm{op}_1(\boldsymbol{i})$ to $\mathrm{op}_2(\boldsymbol{j})$ if for $1 \le k \le d$, $i_k\, e_k\, j_k$, i.e., the relation e_k is defined by

$$e_k = \begin{cases} < & \text{if } i_k < j_k \\ = & \text{if } i_k = j_k \\ > & \text{if } i_k > j_k \\ * & \text{any one of } \{<,=,>\} \end{cases}\,. \tag{4}$$

There are several well-known data dependence analysis algorithms exploited for practical parallelizing compilers. The Banerjee Inequalities can handle one linear equation under the bounds of Eq. (3) and Eq. (4), and the Banerjee Algorithm can deal with one linear equation under the bounds of Eq. (2) and Eq. (4) [7]. When applied to practical cases, the Banerjee Test (the Banerjee Inequalities and the Banerjee Algorithm) may lose accuracy. The I Test and the Direction Vector I Test are a combination of the Banerjee inequalities and the GCD Test [8] [9]. They determine integer solutions for a linear equation with constant bounds and given direction vectors. The Lambda Test extends the Banerjee Inequalities to allow m linear equations in Eq. (1) under the constraints of Eq. (3) and Eq. (4) to be tested simultaneously [10]. And the Generalized Lambda Test allows m linear equations in Eq. (1) under the constraints of Eq. (2) and Eq. (4) to be tested simultaneously [2]. More precise results can be obtained by judging the consistency of a linear system of equations and inequalities inexpensively.

All of above data dependence analysis methods, however, are exploited for general parallelizing compilers, and they ignore the fact that the software pipelining technique *per se* has an impact on instruction-level parallelism. The next section will prove that for instruction-level parallelizing compilers more interesting results can be achieved under additional constraints of dependence difference inequalities.

3 Dependence Difference Inequalities

This section introduces a kind of additional constraints, called *dependence difference inequalities*, for software pipelining techniques. Under these additional constraints the solution space for a linear system can be compressed heavily.

3.1 Relationship between Software Pipelining and Data Dependence

In general, the ILSP algorithm overlaps adjacent iterations of a nested loop program. The *initiation interval* of these adjacent iterations, denoted by II, is only restricted by resource limit, denoted by $\mathrm{II}_{\mathrm{res}}$, and sequential semantics of the loop program, denoted by $\mathrm{II}_{\mathrm{sem}}$, i.e., $\mathrm{II} = \max\{\mathrm{II}_{\mathrm{res}}, \mathrm{II}_{\mathrm{sem}}\}$. For the sake of clarity the paper only concentrates on $\mathrm{II}_{\mathrm{sem}}$ since $\mathrm{II}_{\mathrm{res}}$ can always be released by using more function units.

Definition 3. Let op_1 and op_2 be two operations of a loop program. The number of operations between op_1 and op_2 plus 1 is referred to as *operation distance*, denoted by dis(op_1, op_2).

Definition 4. If $\mathrm{op}_2(\boldsymbol{j})$ is dependent on $\mathrm{op}_1(\boldsymbol{i})$, $\boldsymbol{i} = (x_1, x_3, \ldots, x_{2n-1})$ and $\boldsymbol{j} = (x_2, x_4, \ldots, x_{2n})$, then $\boldsymbol{i} - \boldsymbol{j} = (x_1 - x_2, x_3 - x_4, \ldots, x_{2n-1} - x_{2n})$ is referred to as *dependence difference vector*, denoted by $\mathbf{dif}(\mathrm{op}_1, \mathrm{op}_2)$, and $i_k - j_k = x_{2k-1} - x_{2k}$ is referred to as the dependence difference in the k-th level nested loop, denoted by $\mathrm{dif}_k(\mathrm{op}_1, \mathrm{op}_2)$. If $\mathrm{dif}_k(\mathrm{op}_1, \mathrm{op}_2) = 0$ then $\mathrm{op}_2(\boldsymbol{j})$ is intra-loop-dependent on $\mathrm{op}_1(\boldsymbol{i})$ otherwise inter-loop-dependent.

For the ILSP algorithm intra-loop-dependences have no impact on $\mathrm{II}_{\mathrm{sem}}$ but those inter-loop-dependences may make a strong impact on $\mathrm{II}_{\mathrm{sem}}$.

Lemma 1 [11]. Let op_1 and op_2 be two operations, both belonging to the same k-th level nested loop. If $0 < \mathrm{dif}_k(\mathrm{op}_1, \mathrm{op}_2) \le \mathrm{dis}(\mathrm{op}_1, \mathrm{op}_2)$ does not hold, then the loop program can be software-pipelined with $\mathrm{II}_{\mathrm{sem}} = 1$.

Lemma 1 shows that the loop program can not be software-pipelined with $\mathrm{II}_{\mathrm{sem}} = 1$ where $0 < \mathrm{dif}_k(\mathrm{op}_1, \mathrm{op}_2) \le \mathrm{dis}(\mathrm{op}_1, \mathrm{op}_2)$ holds. However, it does not imply that the loop program can not be software-pipelined with a greater one.

Definition 5. An inequality of the form $1 \le \mathrm{dif}_k(\mathrm{op}_1, \mathrm{op}_2) \le \mathrm{dis}(\mathrm{op}_1, \mathrm{op}_2)$ is termed a *dependence difference inequality* for $1 \le k \le n$.

Theorem 1 [11]. Let op_1 and op_2 be two operations, both belonging to the same n-nested loop program whose loop labels are denoted by $L_1, L_2, \ldots, L_n$ in turn. The loop can be software-pipelined with any value of initiation interval if the following dependence difference inequalities

$$1 \le \mathrm{dif}_k(\mathrm{op}_1, \mathrm{op}_2) \le \mathrm{dis}(\mathrm{op}_1, \mathrm{op}_2),\ 1 \le k \le n, \tag{5}$$

do not satisfy simultaneously.

Theorem 1 implies that dependence difference inequalities can act as, on their own or with other constraints, additional constraints to each variable in a linear system. If there does not exist any integer solution for the linear system under these constraints, the loop can be software-pipelined with $\mathrm{II}_{\mathrm{sem}} = 1$ even if a data dependence between two operations exists.

3.2 Dependence Difference Inequalities vs. Direction Vectors

In general, a direction vector $\boldsymbol{e} = (e_1, e_2, \ldots, e_n)$ bounds the solution space for a linear system with $i_k < j_k$ or $i_k > j_k$ for $1 \le k \le n$. $i_k = j_k$ means two operations are intra-loop-dependent on each other, and thus the dependence can be ignored when the loop is software-pipelined by the ILSP algorithm.

Suppose $i_k > j_k$. When using a direction vector as a constraint we have $f_k(x_2, x_4, \ldots, x_{2n}) \le j_k < i_k \le g_k(x_1, x_3, \ldots, x_{2n-1})$, i.e.,

$$1 \le i_k - j_k \le g_k(x_1, x_3, \ldots, x_{2n-1}) - f_k(x_2, x_4, \ldots, x_{2n}). \tag{6}$$

On the other hand, by using dependence difference inequalities as constraints we have

$$1 \le i_k - j_k \le \mathrm{dis}(\mathrm{op}_1, \mathrm{op}_2). \tag{7}$$

It is shown that dependence difference inequalities are more strict than direction vectors since in most cases $g_k(x_1, x_3, \ldots, x_{2n-1}) - f_k(x_2, x_4, \ldots, x_{2n})$, as an iteration counter, is far greater than $\mathrm{dis}(\mathrm{op}_1, \mathrm{op}_2)$. In one word, these dependence difference inequality constraints make our data dependence analysis algorithm more powerful.

Table 2 gives a practical loop example where op_2 is anti-dependent on op_1 and $\mathrm{dis}(\mathrm{op}_1, \mathrm{op}_2) = 3$. The corresponding data dependence equation of the loop program is $2i_1 + 1 = j_1$, i.e., $2i_1 - j_1 = -1$. Because $\gcd(2, 1) = 1$, the GCD Test draws a conclusion that op_1 is dependent on op_2 and the loop can not be parallelized. On the other hand, it can be derived that $-90 \le 2i_1 - j_1 \le 195$ from $5 \le i_1, j_1 \le 100$, namely, $-90 \le -1 \le 195$, which makes the Banerjee Test also draws a conclusion that the loop can not be parallelized. Furthermore, when a direction vector $i_1 < j_1$ is applied we first have $-95 \le i_1 - j_1 \le -1$, and second $-106 \le i_1 - j_1 \le -6$ from $i_1 - j_1 = -i_1 - 1$. The Banerjee Test still draws the same conclusion since the interaction of the solution spaces for the two inequalities is not empty.

Table 2. Dependence difference inequalities are more strict than direction vectors

```
for(L1=5;L1<=100;++L1)
{
  op1: X = A[2L1+1];
  Y = X + 5;
  Z = Y * 3;
  op2: A[L1] = Z;
}
```

(a) A loop program

Clock	Iteration			
	$i = 5$	$i = 6$	$i = 7$	$i = 8$
1	`X=A[11];`			
2	`Y=X+5;`	`X=A[13];`		
3	`Z=Y*3;`	`Y=X+5;`	`X=A[15];`	
4	`A[5]=Z;`	`Z=Y*3;`	`Y=X+5;`	`X=A[17];`
5		`A[6]=Z;`	`Z=Y*3;`	`Y=X+5;`
6			`A[7]=Z;`	`Z=Y*3;`
7				`A[8]=Z;`

(b) Correct software pipelining with $\mathrm{II}_{\mathrm{sem}} = 1$

When replacing the direction vector with a dependence difference inequality, we can clearly find the interaction of the solution spaces for the two inequalities, $1 \le i_1 - j_1 \le \mathrm{dis}(\mathrm{op}_1, \mathrm{op}_2) = 3$ and $-106 \le i_1 - j_1 \le -6$, is empty. Thus our data dependence analysis algorithm determines that the loop can be paralleled, as shown in Table 2.

4 Conclusion

This paper has presented a new static data dependence analysis approach, called dependence difference inequality, for our software pipelining algorithm ILSP for nested loops. Our data dependence analysis approach can deal with coupled subscripts for multi-dimensional array references statically.

Conceptually, unlike a direction vector, a dependence difference inequality is not only related to the iteration space for a loop program but also related to the operation distance between two operations. Dependence difference inequalities can act as additional constraints to each variable in a linear system on their own or with other constraints, such as direction vectors. They are more strict than a direction vector and make our data dependence analysis algorithm more powerful. As a result, the solution space for the linear system can be compressed heavily.

Under constraints of dependence difference inequalities, so long as these inequalities do not satisfy simultaneously, the loop can be software-pipelined with any value of initiation interval even though there exists a data dependence between two operations. The paper has also shown that some conservative estimations made by other traditional data dependence analysis approaches can be eliminated by replacing a direction vector with a dependence difference inequality.

Further experimental results are reported in [12]. On the other hand, a dynamic data dependence analysis approach is presented in [13], which can work with this method together to coping with data dependencies for software pipelining.

Acknowledgement

This work was partially supported by National Nature Science Foundation, grant number 60173010, of *P. R. China*.

References

1. Petersen, P. M., Padua, D. A.: Static and Dynamic Evaluation of Data Dependence Analysis Techniques. IEEE Transactions on Parallel and Distributed Systems 7 (1996) 1121–1132
2. Chang, W. L., Chu, C. P., Wu, J.: The Generalized Lambda Test: A Multi-Dimensional Version of Banerjee's Algorithm. International Journal of Parallel and Distributed Systems and Networks 2 (1999) 69–78
3. Rau, B.R.: Iterative Modulo Scheduling. Technical Report HPL-94-115. Hewlett-Packard Laboratory, Palo Alto, CA (1994)
4. Su, B., Ding, S., Wang, J., Xia, J.: GURPR — A Method for Global Software Pipelining. ACM SIGMICRO Newsletter 19 (1988) 32–36
5. Qiao, L., Tang, Z. Z., Wang, S. Y.: Control Strategies of Software Pipelining: Dealing with the Prologue and the Epilogue of Nested Loops. In: Zhou, X., Xu, M., Lou, S., Yang, X. (eds.): Proceedings of the 3rd Workshop on Advanced Parallel Processing Technologies, 19-21 Oct. 1999, Changsha, China. Publishing House of Electronics Industry, Beijing (1999) 177–181
6. Rong, H. B., Tang, Z. Z., Govindarajan, R., Douillet, A., Gao, G. R.: Single-Dimension Software Pipelining for Multi-Dimensional Loops. In: Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization, 21-24 Mar. 2004, San Jose, CA. IEEE Computer Society, Los Alamitos, CA (2004) 163–174
7. Banerjee, U.: Dependence Analysis. Kluwer Academic Publishers, Norwell MA (1997)
8. Kong, X, Klappholz, D., Psarris, K.: The I Test. IEEE Transactions on Parallel and Distributed System 2 (1991) 342–359
9. Kong, X, Klappholz, D., Psarris, K.: The Direction Vector I Test. IEEE Transactions on Parallel and Distributed System 4 (1993) 1280–1290
10. Li, Z., Yew, Y. C., Zhu, C. Q.: An Efficient Data Dependence Analysis for Parallelizing Compilers. IEEE Transactions on Parallel and Distributed System 1 (1990) 26–34
11. Qiao, L.: On Data Dependencies in Software Pipelining. Doctorial Dissertation, Department of Computer Science, Tsinghua University, Beijing (2001)
12. Qiao, L., Huang, W. T., Tang, Z. Z.: Coping with Data Dependencies of Multi-Dimensional Array References. In: Jin, H., Reed, D., Jiang, W. (eds.): Proceedings of IFIP International Conference on Network and Parallel Computing, Beijing, Lecture Notes in Computer Science. Springer-Verlag, Berlin Heidelberg New York (2005) accepted by NPC'05
13. Qiao, L., Huang, W. T., Tang, Z. Z.: A Dynamic Data Dependence Analysis Approach for Software Pipelining. In: Jin, H., Reed, D., Jiang, W. (eds.): Proceedings of IFIP International Conference on Network and Parallel Computing, Beijing, Lecture Notes in Computer Science. Springer-Verlag, Berlin Heidelberg New York (2005) accepted by NPC'05