Guiding instruction scheduling with synchronisation markers on a superscalar based multiprocessor

R.-Y. Hwang
F. Lai

Indexing terms: Data dependence, Instruction scheduling, Shared memory multiprocessor, Superscalar, Synchronisation, Synchronisation marker

Abstract: Exploiting loop parallelism is an important way to enhance system performance. For loop-carried dependence, the original DO loop is converted into a DOACROSS loop to function concurrently. In general, synchronisation operations are inserted to maintain order dependence during parallel execution. For each processor in a shared memory multiprocessor, if the executing sequence is the same as the original source program, the action of synchronisation operation is correct; however, if each processor is used out of order, such as in the superscalar machine, the action of synchronisation operation may be incorrect. The synchronisation marker insertion method proposed resolves this problem in two steps: (i) proper synchronisation markers are appended into the array element of dependences, and (ii) synchronisation markers are generated during intermediate code generation. Finally, algorithms are proposed to prevent error during instruction scheduling.

1 Introduction

A parallel compiler exploits loop level parallelism and instruction scheduling exploits instruction level parallelism. In the past, people studied either on loop level or on instruction level independently but some new problems need to be resolved where we consider loop level and instruction level simultaneously. For example, consider a shared memory multiprocessor in which each processing element is a superscalar processor: we need to exploit loop level and instruction level simultaneously. In this paper, we discuss a synchronisation problem in which synchronisation operation is inserted at loop level but the order dependence may be broken during instruction scheduling.

Parallel loops in a program, whose iterations can be executed concurrently on different processors, provide the greatest potential of parallelism to be exploited by multiprocessor systems [11, 13]. If the iterations of a parallel loop are independent, it is called a DOALL loop. If there are data dependences across iterations of a DO-loop (loop-carried dependence), its iterations can still be executed concurrently on different processors, provided that the data dependences are enforced by synchronisation across the processors during the execution. This kind of parallel loop is called a DOACROSS loop [2, 14]. There have been some compiler techniques on multiprocessor data synchronisation for DOACROSS loops. Midkiff [10] and Wolfe [16] suggested inserting statement level synchronisation instructions such as set/wait and send/wait in the loop body to enforce data dependences. These schemes can only handle constant distance data dependences and single-nested DOACROSS loops. Su and Yew proposed a process oriented data synchronisation scheme for constant distance data dependences [12]. Li provides two operations, POST and WAIT, on a logical event variable to execute a DOACROSS loop in parallel [8]. The operation POST(v) sets the event variable v to TRUE; The operation WAIT(v) busy-waits until v becomes TRUE. The initial value of an event variable is FALSE. The scheme of event variable synchronisation is suitable for some variable distances and nested loops. It can also handle single loop synchronisation. Tang, Yew and Zhu present an algorithm based on special counters [15]. They use two data oriented *synch read* and *synch write* to replace the regular read and write in the original sequential program, respectively. The ordering number for each data access is decided at compilation time. This scheme considers a subset of loop bounds and subscript in the event variable synchronisation method.

From the discussion above, we know that Li's approach is a most powerful one. Fig. 1 is an example of event variable synchronisation [8]. A nested loop is shown in Fig. 1*a*, and its synchronisation operation insertion is shown in Fig. 1*b*. In Fig. 1*b*, EV is a bit array to record whether its corresponding iteration has finished or not. Because the dependence sink should not wait in all loop iterations, a condition must be checked in each iteration to see if a WAIT must really be executed. This condition is called mask predicate, which is tested by the IF statement in Fig. 1*b*. On the other hand, because the distance of dependence relation is variable, the distance between dependence source and sink must be determined.

Paper 1519E (C1), first received 4th October 1993 and in revised form 11th July 1994

R.-Y. Hwang is with the Department of Electrical Engineering, National Taiwan University Taipei, Taiwan, Republic of China

F. Lai is with the Department of Electrical Engineering and Department of Computer Science and Information Engineering, National Taiwan University Taipei, Taiwan, Republic of China

This work is supported in part by the National Science Council under grant NSC 82-0408-E-002-093.

The correspondence between two dependent references is called contact, which should be maintained by indexing the event array correctly in POST and WAIT as in Fig. 1*b*. For details about deciding mask predicate and contact, refer to Reference 8.

```
DO I1 = 1, N
  DO I2 = I1 + 1, N + 1
    ...
    B[I1, I2] := A[I1, I2 - 2]
    ...
    A[I1 + I2, I2] := C[I1 + 3, I2]
    ...
  ENDDO
ENDDO
```

a

```
DOALL I1 = 1, N
  DOALL I2 = I1 + 1, N + 1
    WT: IF ((I2.EQ.I1 + 1).AND.(I1.GE.3))
      WAIT(EV[I1 - I2 + 2, I2 - 2])
    S1: B[I1, I2] := A[I1, I2 - 2]
    ...
    S2: A[I1 + I2, I2] := C[I1 + 3, I2]
    PO: POST(EV[I1, I2])
    ...
  ENDDO
ENDDO
```

b

Fig. 1 *Example of event variable synchronisation*

a nested loop *b* synchronisation operation insertion

Li makes two assumptions to ensure that explicit synchronisation is needed only for loop carried dependences: (i) The processors executing a parallel loop may exit only after all iterations are completed, and (ii) statements in the same iteration of a parallel loop execute sequentially in their original order. However, if the program shown in Fig. 1*b* is executed on a multiprocessor in which each processor is executed out of order, such as the superscalar processor [4, 6, 7], the assumption will be invalid and some errors will be incurred. For superscalar machines, instruction scheduling can be done at compilation time. The compiler analyses the dependence relation and decides how to move instructions [6]. For example, there is no dependence relation between the statements S_2 and PO in Fig. 1*b*, and the original executing sequence of three-address code for statements S_2 and PO may be changed after instruction scheduling. It means that the action of POST can be finished before the access of array element $A[I_1 + I_2, I_2]$. For such a case, a processor has not written data into array A yet, but the corresponding bit array has been set. In the meantime, if a processor is waiting for the bit set, error is incurred. So we will get stale data. In this paper, we propose a technique to resolve this problem.

2 Synchronisation conditions

In this Section, we discuss the synchronisation conditions which prevent the instruction scheduling from error. First, some notations are defined.

Src = dependence source
Snk = dependence sink
Sig = a synchronisation instruction 'POST'
Wat = a synchronisation instruction 'WAIT'
P_{src} = the processor which excecutes the iteration of dependence source
P_{snk} = the processor which executes the iteration of dependence sink

Let S_i and S_j be arbitrary statements.

Definition: S_i bef S_j iff S_i occurs before S_j.

Definition: $S_i \delta S_j$ means that statement S_j is dependent on statement S_i.

Definition: In a loop, a dependence $S_i \delta S_j$ is said to be LFD (lexically forward dependence) iff S_i bef S_j. Any dependence that is not LFD is an LBD (lexically backward dependence).

Now, synchronisation conditions are described as follows.

For LFD and LBD

(i) A *Sig* can not precede the corresponding *Src*.

(ii) A *Wat* can not be behind the corresponding *Snk*.

If the above conditions are not satisfied, some errors will be incurred. For the LFD case, if condition (i) is violated, a *Sig* is issued before the corresponding *Src*. In the meantime, if the P_{snk} is waiting for this *Sig*, it will mistakenly act as if the corresponding *Src* has been accessed and the P_{snk} might access stale data. Similarly, when condition (ii) is violated, the P_{snk} always accesses the *Snk* without suspending, and the order dependence is violated. The *Wat* (or *Snk*) can go across the *Sig* (or *Src*), and there is no error because they are independent of each other inside the iteration. For an iteration *i*, *Sig* is to signal the dependence sink after iteration *i*; however, *Wat* is to wait for the dependence source after iteration *i*. Therefore, the deadlock can not happen. In such a case, the LFD is converted into the LBD. For the LBD case, if condition (i) or (ii) is violated, the error is the same as in the LFD case. Similarly, the *Src* (or *Sig*) can go across the *Snk* (or *Wat*), and no errors will be incurred. If *Sig* goes across the *Wat*, it means that *Src* must go across *Src*, *Snk* and *Wat*. The dependence relation still holds because it is a loop carried dependence. In such case, the LBD is converted into the LFD.

3 Implementation of synchronisation marker insertion

In this Section, we propose an approach to resolving the problem. The simplest solution for this problem is to constrain instruction moving across any synchronisation operation. However, this will seriously affect the function of instruction scheduling. According to Reference 5, the instruction parallelism in a basic block is scarce. If this problem is resolved in this way, the performance of the superscalar processor will degrade seriously. This problem has two features: (i) the dependence infomation is constructed during synchronisation operation insertion which is done at statement level, (ii) the order dependence maintained by synchronisation operation may be broken during instruction scheduling. From the features, it is clear that the solution is very troublesome if we try to resolve this problem at instruction level without any information provided by the statement level. Therefore, we convert the dependence information, which is constructed at statement level, into synchronisation markers which always attach to the dependence event, and then the error prevention algorithms are developed to guide the instruction scheduler for correct scheduling. Basically, if we can satisfy the synchronisation conditions above,

the problem is resolved. A marker is a pseudoinstruction, the format of which is either Δ_t or ∇_t. Δ_t, the upmarker, represents the situation that the synchronisation condition may be violated when the dependence event *Src*, *Snk*, *Sig*, or *Wat* immediately following Δ_t is moved up. Conversely, ∇_t, the downmarker, means that the synchronisation condition may be violated when the dependence event immediately beyond ∇_t is moved down. Variable t is to identify whether the dependence events belong to the same dependence relation. The problem is overcome if we deposit different synchronisation marker pairs between *Src* and its corresponding *Sig*, and *Snk* and its corresponding *Wat*. When the dependence event is moved up (down), the marker near to the event must be moved up (down) together with it as one unit. After the instruction scheduling is finished, all markers inserted are deleted. Therefore, this will not increase the original program size.

Now, we consider how to insert the synchronisation markers into the dependence relation. First, we illustrate the example shown in Fig. 1*b*. In Fig. 1*b*, according to the synchronisation conditions discussed in the last Section, we need to insert a downmarker ∇_i which immediately follows the dependence source $A[I_1 + I_2, I_2]$, and an upmarker Δ_i which immediately precedes the synchronisation operation POST. Similarly, an upmarker $\Delta_j (j \neq i)$ is deposited immediately before the dependence sink $A[I_1, I_2 - 2]$, and a downmarker ∇_j is deposited immediately after the synchronisation operation WAIT(EV$[I_1 - I_2 + 2, I_2 - 2]$). An n-dimensional array element $A[I_1, I_2, \ldots, I_n]$ must be converted into one dimension address to store in memory. Basically, there are two fundamental forms, row-major and column-major, to finish this work. Therefore, an array element can be translated into several three-address codes. Assume A is a 10×10 array and a work is 4 bytes: the array element $A[I_1, I_2 - 2]$ in Fig. 1*b* can be translated as in Fig. 2*a*. Observing Fig. 2*a*, we find that the correspondence of array element $A[I_1, I_2 - 2]$ at instruction level is instruction i: $t_4 \leftarrow A[t_3]$. So, the synchronisation marker Δ_j needs to be deposited immediately before instruction i in Fig. 2*a* and is shown in Fig. 2*b*.

a	*b*
$t_1 \leftarrow I_2 - 2$	$t_1 \leftarrow I_2 - 2$
$t_2 \leftarrow I_1 * 10$	$t_2 \leftarrow I_1 * 10$
$t_2 \leftarrow t_2 + t_1$	$t_2 \leftarrow t_2 + t_1$
$t_3 \leftarrow 4 * t_2$	$t_3 \leftarrow 4 * t_2$
i: $t_4 \leftarrow A[t_3]$	Δ_j
	i: $t_4 \leftarrow A[t_3]$

Fig. 2 *Three-address codes with synchronisation marker insertion*
a three-address codes for array element $A[I_1, I_2 - 2]$
b after synchronisation marker insertion

With a dependence event, we can classify the event as one of the five types of dependence relation. The five types are (i) single dependence source, (ii) single dependence sink, (iii) multiple dependence source, (iv) multiple dependence sink and (v) one or more dependence sources and sinks. Let A and B be *Src*, *Snk*, *Sig*, *Wat*, or Wat_n (Wat_n represents n *Wats*). An A-B marker is a synchronisation marker pair into which the downmarker ∇_i and the upmarker Δ_i are inserted immediately following and preceding A and B, respectively. If A is Wat_n, n identical downmarkers are inserted immediately following n *Wats*. The synchronisation marker insertion of the five types is shown in Fig. 3. With the single dependence source *Src* in Fig. 3*a*, we need only insert a *Src–Sig* marker to prevent violating synchronisation conditions. Similarly, with the single dependence sink *Snk* in Fig. 3*b*, the multiple dependence source Src_n in which there are n corresponding dependence sinks in Fig. 3*c*, and the multiple dependence sink Snk_n in which there are n corresponding dependence sources in Fig. 3*d*, we need only insert *Wat–Snk* marker, *Src–Sig* marker, and Wat_n–*Snk* marker, respectively. With a data dependence event Src_m/Snk_n in which there are m corresponding dependence sinks and n corresponding dependence sources in Fig. 3*e*, we insert two different synchronisation marker pairs Wat_n-*Snk* marker and *Src–Sig* marker. Therefore, for a dependence source (or sink) in a dependence relation, there exist, at most, two synchronisation markers to maintain the correct execution.

			Wat_1	Wat_1
			∇_j	∇_1
			...	...
Src	*Wat*	Src_n	Wat_2	Wat_n
∇_i	∇_i	∇_i	∇_j	∇_1
Δ_i	Δ_i	Δ_i	...	Δ_1
Sig	*Snk*	*Sig*	Wat_n	Src_m/Snk_n
			∇_j	∇_k
			Δ_j	Δ_k
			Snk_n	*Sig*
a	*b*	*c*	*d*	*e*

Fig. 3 *Synchronisation marker insertion for five types of dependence relation*
a single dependence source case: *Src–Sig* marker
b single dependence sink case: *Wat–Snk* marker
c multiple dependence source case: *Src–Sig* marker
d multiple dependence sink case: Wat_n–*Snk* marker
e dependence source/sink case: Wat_n–*Snk* and *Src–Sig* marker

From the discussion above, we insert, at most, two different marker pairs to resolve all kinds of dependence cases. Now, we show how to insert synchronisation markers into the program. This is done in two steps:

(i) append synchronisation markers into the dependence event of the source program;

(ii) generate the synchronisation markers during the intermediate code generation.

Step 1 is done by a parallel compiler during the insertion of synchronisation operations [8]. A parallel compiler does dependence analysis to decide whether the dependence relation exists. If any dependence relation is found, the synchronisation operations POST and WAIT are inserted into the dependence relation [9]. At the same time, we append an adequate marker pair between *Src*(*Snk*) and its corresponding *Sig*(*Wat*). The synchronisation marker insertion is shown in Algorithm 1. For an array element A_1 which is a dependence source, A_1 is replaced with string A_1 & i, the corresponding synchronisation marker of which is ∇_i, and string i&POST(EV$[i_1, i_2, \ldots, i_n]$) is inserted after the statement that issues A_1. Similarly, the array element A_2, which is a dependence sink, is replaced with string j&A_2 and statement IF ρ WAIT(ε)&j is inserted before the statement which issues the sink reference A_2 (ρ is *mask predicate* and ε is *contact*). The terminal symbol & is a special symbol which is to combine synchronisation marker and array element as one unit. The synchronisation markers are generated during intermediate code generation. For example, the synchronisation operation insertion shown in Fig. 1*b* can have some markers appended as shown in Fig. 4. In this Figure, array element $A[I_1 + I_2, I_2]$ is a dependence source, so the string &j is appended after it.

Its corresponding POST is replaced with j&POST(EV[I_1, I_2]). Similarly, string i& is inserted before the array element $A[I_1, I_2 - 2]$, which is a dependence sink, and string &i is appended after the corresponding WAIT (EV[$I_1 - I_2 + 2, I_2 - 2$]).

```
DOALL I1 = 1, N
  DOALL I2 = I1 + 1, N + 1
    ...
    WT: IF ((I2.EQ.I1 + 1).AND.(I1.GE.3))
        WAIT(EV[I1 - I2 + 2, I2 - 2])&i
      S1: B[I1, I2] := i&A[I1, I2 - 2]
      ...
      S2: A[I1 + I2, I2]&j := C[I1 + 3, I2]
    PO: j&POST(EV[I1, I2])
    ...
  ENDDO
ENDDO
```

Fig. 4 *Example for appending synchronisation marker in Fig. 1b*

For Step 2, we write a grammar to deposit the synchronisation marker inserted in Step 1 into three-address codes. A translation scheme is a context-free grammar in which program fragments called semantic actions are embedded within the right sides of productions [1]. The translation scheme for array element and statements POST and WAIT is written in Fig. 5. In this Figure, the three-address code is produced by the *generate* procedure invoked in the semantic actions. Assume that *newtemp* generates a new temporary name each time a temporary is needed. A sequence of input characters that comprises a single token is called a *lexeme*. The *lexeme* for the name represented by *id* is given by attribute $id._{place}$. The $Elist._{ndim}$ records the number of dimensions in the Elist. The function *limit*(array, j) returns nj, the number of elements along the jth dimension of the array whose symbol table entry is pointed to by *array* which is stored at $Elist._{array}$. $Elist._{place}$ denotes the temporary value computed from index expression in Elist. $L._{place}$, $F._{place}$, $T._{place}$, $P._{place}$, and $E._{place}$ are pointers which point to the symbol table entry for that name. $L._{offset}$ is a new temporary that holds w times the value of $Elist._{place}$ (w is the number of bytes in a word).

The grammar in Fig. 5 generates the three-address code of array element and inserts synchronisation markers which are immediately after or before the dependence events of three-address code. The semantic action of each rule is listed at right-hand side of the rule. Production rule 1 shows that a statement can be an assignment, PO, or WT statement. Rule 2 describes an assignment statement with simple identifier ($F := E$), array element ($L := E$), or dependence event ($F\&L := E$, $L\&F := E$, or $F_1\&L\&F_2 := E$) on the left-hand side of ':=', respectively. Rules 3, 4, 5, and 6 show the arithmetic operation in which the nonterminal symbol P is a simple identifier ($P \rightarrow F$), array element ($P \rightarrow L$) or dependence event ($P \rightarrow F\&L$, $P \rightarrow L\&F$, or $P \rightarrow F_1\&L\&F_2$). Rules 7, 8, and 9 calculate the address of array element. Rules 10 and 11 generate the synchronisation instruction and its corresponding marker. HWAIT and HPOST are two instructions which are supported by hardware. For the rule $P \rightarrow F\&L$, $F._{place}$ records the variable of the synchronisation marker, and F, a synchronisation marker identifier, is placed before L, which is an array element. Therefore, the upmarker $\Delta F._{place}$ is inserted before L. The corresponding semantic action generates $\Delta F._{place}$ and $P._{place} \leftarrow L._{place}[L._{offset}]$.

1. Stmt → S | PO | WT
2. S → (F | L | F&L | L&F | F_1&L&F_2) := E
   {generate($F._{place}$ ← 'E.$_{place}$');} |
   {generate($L._{place}$'['$L._{offset}$']'' ← 'E.$_{place}$')} |
   {generate('Δ'F.$_{place}$); generate($L._{place}$'['$L._{offset}$']'' ← 'E.$_{place}$')} |
   {generate($L._{place}$'['$L._{offset}$']'' ← 'E.$_{place}$); generate('∇'F.$_{place}$)} |
   {generate('Δ'$F_{1\cdot place}$); generate($L._{place}$'['$L._{offset}$']'' ← 'E.$_{place}$);
   generate('∇'$F_{2\cdot place}$)}
3. E → (E_1 + T | E_1 − T | T)
   {E.$_{place}$:= newtemp; generate(E.$_{place}$ ← '$E_{1\cdot place}$' + 'T.$_{place}$')} |
   {E.$_{place}$:= newtemp; generate(E.$_{place}$ ← '$E_{1\cdot place}$' − 'T.$_{place}$')} |
   {E.$_{place}$:= T.$_{place}$}
4. T → T*P | P
   {T.$_{place}$:= newtemp; generate(T.$_{place}$ ← 'T.$_{place}$'*'P.$_{place}$')} |
   {T.$_{place}$:= P.$_{place}$}
5. P → (F | L | F&L | L&F | F_1&L&F_2 | (E))
   {P.$_{place}$:= F.$_{place}$} |
   {P.$_{place}$:= newtemp; generate (P.$_{place}$ ← 'L.$_{place}$'['L.$_{offset}$']'} |
   {P.$_{place}$:= newtemp; generate('Δ'F.$_{place}$);
   generate (P.$_{place}$ ← 'L.$_{place}$'['L.$_{offset}$']'} |
   {P.$_{place}$:= newtemp; generate (P.$_{place}$ ← 'L.$_{place}$'['L.$_{offset}$']';
   generate('∇'F.$_{place}$)} |
   {P.$_{place}$:= newtemp; generate('Δ'$F_{1\cdot place}$);
   generate(P.$_{place}$ ← 'L.$_{place}$'['L.$_{offset}$']'; generate('∇'$F_{2\cdot place}$)} |
   {P.$_{place}$:= E.$_{place}$}
6. F → id
   {F.$_{place}$:= id.$_{place}$}
7. L → Elist]
   {L.$_{place}$:= Elist.$_{array}$; L.$_{offset}$:= newtemp;
   generate (L.$_{offset}$ ← 'w*Elist.$_{place}$')}
8. Elist → $Elist_1$, E
   {t := newtemp; m := $Elist_1$ + 1; Elist.$_{place}$:= t; Elist.$_{ndim}$:= m
   generate(t' ← '$Elist_{1\cdot array}$'*' limit($Elist_{1\cdot array}$, m));
   generate(t' ← 't'+'E.$_{place}$'); Elist.$_{array}$:= $Elist_{1\cdot array}$}
9. Elist → id [E
   {Elist.$_{place}$:= E.$_{place}$; Elist.$_{ndim}$:= 1; Elist.$_{array}$:= id.$_{place}$}
10. PO → F&POST(E){generate('Δ'F.$_{place}$); generate('HPOST('E.$_{place}$')')}
11. WT → WAIT(E)&F{generate('HWAIT('E.$_{place}$')'); generate('∇'F.$_{place}$)}

Fig. 5 *Translation scheme for addressing array element with synchronisation marker insertion*

The grammar shown in Fig. 5 is LALR (1) [1], so we know that this grammar is realisable. It is left recursive; therefore, bottom-up parsing is employed and it is implemented with shift reduce parser. A *handle* of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ. That is, if $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A \omega \underset{rm}{\Rightarrow} \alpha\beta\omega$, the $A \rightarrow \beta$ in the position following α is a *handle* of $\alpha\beta\omega$. Basically, shift-reduced parsing is used to find a *handle*. The action 'shift' is applied if a *handle* can not be found; or the action 'reduce' is applied, and it will reduce the production in which a *handle* is found. The three-address code of statements S_2 and PO in Fig. 4 is shown in Fig. 6. In this Figure, the correspondence of dependence event

```
      t1 ← I1 + I2
      t2 ← t1 * 10
      t2 ← t2 + I2
      t3 ← 4 * t2
   l: t4 ← I1 + 3
      t5 ← t4 * 10
      t5 ← t5 + I2
      t6 ← 4 * t5
      t7 ← C[t6]
   l: A[t3] ← t7
         ∇j
      t8 ← I1 * 10
   m: t8 ← t8 + I2
      t9 ← 4 * t8
      t10 ← EV[t9]
         Δj
      HPOST(t10)
   n: ...
```

Fig. 6 *Three-address code for statements S2 and PO in Fig. 4*

$A[I_1 + I_2,\ I_2]\&j$ in instruction level is instruction l: $A[t_3] \leftarrow t_7$ and ∇_j. Similarly, the corresponding instructions of synchronisation operation PO in instruction level are Δ_j and HPOST(t_{10}). Therefore, HPOST(t_{10}) and $A[t_3] \leftarrow t_7$ can not violate synchronisation conditions if we do not allow synchronisation pair Δ_j and ∇_j to cross each other. In the next Section, we propose three efficient error prevention algorithms to prevent the instruction scheduling from error.

4 Error prevention algorithm

4.1 Simple and efficient prevention algorithm

In this Section, we propose algorithms to ensure correct instruction scheduling. The scheduling fails if marker pair Δ_i and ∇_i meet together during the scheduling. To identify dependent events, the synchronisation markers are always attached to the dependent events. There is a directed edge from block B_1 to block B_2 if B_2 can immediately follow B_1 in some execution sequence. We say that B_1 is a *predecessor* of block B_2, and B_2 is a *successor* of B_1, and B_1 precedes B_2 if there are directed edges from B_1 to B_2. Algorithm 2 describes how to maintain correct scheduling with out of order execution. In this algorithm, assume that the instruction I_x in $BB(A)$, basic block A with offset x, is moved to the position y of $BB(B)$ and called I_y thereafter. This is shown in Fig. 7. First, we need to know what instructions are around dependence event I_x. If neither I_{x+1} nor I_{x-1} is a synchronisation marker, the scheduling succeeds. Otherwise, each dependence event contains at most two markers: Instruction I_{x+1} must be ∇_t if it exists and instruction I_{x-1} must be Δ_t if it exists. With marker ∇_t, there is no problem if the moving direction is up during the scheduling; however, if the

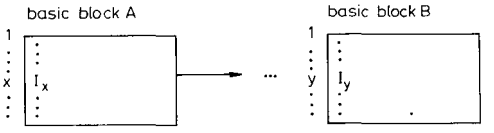

Fig. 7 *Example of instruction scheduling*
assuming instruction scheduler decides to move instruction Ix in basic block A with offset x to position y of basic block B

direction is down, it is necessary to check from $BB(A, x)$ to $BB(B, y)$ whether marker Δ_t is encountered during the scheduling. In this algorithm, function *Converse*(I) is to find the corresponding marker of I, where I is either an up or down marker. For example, if I is ∇_t, *Converse*(I) is Δ_t. The scheduling fails if the two markers Δ_t and ∇_t meet together. For example, if instruction l in Fig. 6 is to be moved across instruction m, this is an allowable movement; however, if it is to be moved across instruction n, this movement fails because the two markers, Δ_j and ∇_j, meet together.

4.2 Two modified prevention algorithms

In this Section, we slightly modify Algorithm 2 by separating the dependence event from the basic block to improve efficiency. Let d be the number of quadruples between instructions I_x and I_y in Algorithm 2. The time complexity of Algorithm 2 is O(d) if I_x is a dependence event and the moving direction is the same as its attached marker; otherwise, the complexity is O(1). Can the scheduling efficiency be improved again? First, we separate *Sig*(*Wat*) and its attached synchronisation marker, synchronisation block, from the original basic block and name it Algorithm 3. By definition, a basic block begins execution at its top and executes all instructions in sequence, then ends with a conditional or unconditional branch. No branches into the middle of a basic block are allowed. A subblock is a set of contiguous three-address codes in a basic block. A synchronisation block is a subblock in which there are only two instructions: HPOST or HWAIT and its nearby marker. The attributes of a synchronisation block consist of *type*, and *mid*. The *type* shows the type of block, zero for subblock and one for synchronisation block. The *mid* is used for recording the corresponding identifier of synchronisation for the synchronisation block. For such an arrangement, we need only check whether the synchronisation conditions are violated by checking the type of block. If the marker pair meet together, scheduling fails. However, block splitting and merging are required in Algorithm 3 and the number of blocks will increase. What is the number of blocks in Algorithm 3? Let the number of original basic blocks in a flow graph be b. Assume there are m dependence events and n of them are multiple dependence source/sinks. From the discussion in the last Section, if a dependence event is not multiple dependence source/sink, we need only one synchronisation marker; otherwise, two synchronisation markers are needed. For $(m - n)$ dependence events, we need at least $(m - n)$ synchronisation blocks. Similarly, for n multiple dependence events, we need at least $2n$ synchronisation blocks. Therefore, the low bound is $b + (m - n) + 2n = b + m + n$. On the other hand, if a synchronisation block is moved to the middle of the original basic block,

each block will be divided into two subblocks. Therefore, the upper bound of total number of blocks will be $b + (m - n) + (m - n) + 2n + 2n = b + 2(m + n)$. From the discussion above, the range of total number of blocks including HPOST and HWAIT synchronisation blocks is between $b + m + n$ and $b + 2(m + n)$. Therefore, the total number of blocks in Algorithm 3 is greater than the original blocks and dependent on the number of dependence events and the position of synchronisation blocks. For example, the flow graph of Fig. 6 is shown in Fig. 8*a*. In this Figure, the HPOST synchronisation block is isolated from the original block. The instruction in position l can be moved to position m because blocks A and B are in the same block. In another case, assuming the instruction in position l is to be moved to position n, we inspect HPOST synchronisation block and find that a synchronisation marker pair meet together. Therefore, the scheduling fails. We only inspect one synchronisation block, and this is more efficient than Algorithm 2. However, if instruction HPOST(t_{10}) is to be moved to position m, the scheduling succeeds but block splitting and merging are executed. Now the inspection should be done statement by statement instead of block by block. After block splitting and merging, the changed flow graph is shown in Fig. 8*b*.

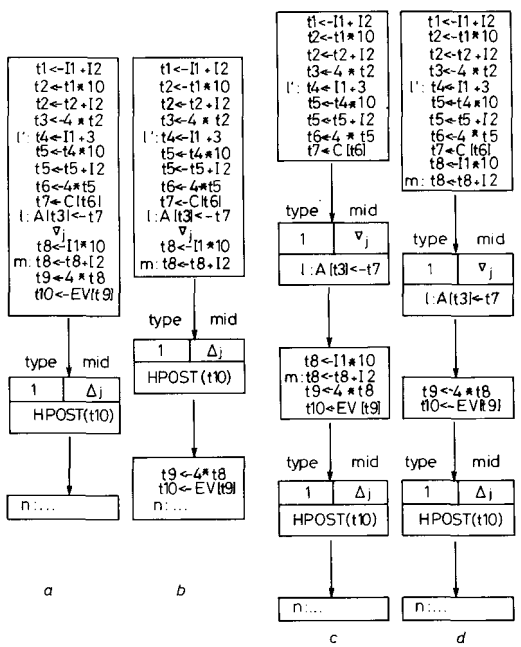

Fig. 8 *Flow graph for algorithms 3 and 4*

a flow graph for Algorithm 3

b flow graph for *a* after block splitting and merging (instruction HPOST($t10$) is moved to position m)

c flow graph for Algorithm 4

d flow graph for *c* after block splitting and merging (the instruction in position l is moved to position m)

Similarly, we can also improve the scheduling efficiency when I_x is *Sig* or *Wat*. If all *Srcs* (*Snks*) and those neighbouring markers are isolated from the original block, the problem is conquered. This method is named Algorithm 4. For such an arrangement, we can make the scheduling decision from the attributes of each block. The error checking is done in a synchronisation block because the dependence events had been isolated in a synchronisation block. Therefore, we inspect synchronisation conditions only when I_x is in a synchronisation block. The time complexity for I_x being a dependence event is O(c), where c is the number of blocks between blocks A and B. If every dependence event is formed as a synchronisation block, what is the range of total number of blocks including synchronisation blocks? For $(m–n)$ dependence events, there are at least $(m–n)$ synchronisation blocks and $(m - n)$ blocks which consist of dependence events and its marker. Similarly, for n dependence events which are multiple dependence source/sink, we need at least $2n$ synchronisation blocks and n blocks. Therefore, the low bound of total number of blocks is $b + (m - n) + (m - n) + 2n + n = b + 2m + n$. Similarly, if a synchronisation block is moved into the middle of the original basic block, each block will be separated into two subblocks. The upper bound of total number of blocks is $b + 2 * [(m - n) + (m - n)] + 2 * (n + n + n) = b + 4m + 2n$. Therefore, if every dependence event is formed as a synchronisation block, the range of total number of blocks including synchronisation blocks is between $b + 2m + n$ and $b + 4m + 2n$. From the discussion above, the number of blocks in Algorithm 4 is more than in Algorithm 3. This will increase the operation of block splitting and merging. However, the time complexity is O(c) if I_x is a dependence event, where c is the number of blocks. For example, the flow graph of the program segment in Fig. 6 is shown in Fig. 8*c*. In this Figure, the action of movement from position l to position m or n is similar to Algorithm 3 and block splitting and merging would be executed. The flow graph after moving the instruction in position l to position m is shown in Fig. 8*d*. To move instruction HPOST (t_{10}) to position l', we need only check two blocks to find out that the scheduling is not allowable. For Algorithms 2 and 3, this must be inspected statement by statement. In general, Algorithm 2 is a simple but efficient error prevention algorithm. However, if I_x is a dependence event and the number of instructions between I_x and I_y is large, the performance of Algorithms 3 and 4 is better than that of Algorithm 2.

5 Conclusion

The major goal of the superscalar based multiprocessor is to exploit loop and instruction parallelism. This requires reconsideration of several interesting problems such as error prevention, discussed in this paper, scheduling approaches, and other related compiler techniques. We have proposed an approach to resolving the problem of instruction scheduling with out of order execution. The most important contribution of this paper is that it shows how to prevent the instruction scheduling from error by providing synchronisation markers to guide the instruction scheduler for correct scheduling. In most cases, all algorithms proposed are very efficient (O(1)). The synchronisation marker method is suitable for post scheduling and prescheduling.

6 References

1 AHO, A.V., SETHI, R., and ULLMAN, J.D.: 'Compilers principles, techniques, and tools' (Addison-Wesley, Reading MA, 1986)

2 ALLEN, J.R., and KENNEDY, K.: 'Automatic translation of Fortran programs to vector form', *ACM Trans.*, 1987, **PLS-9**, (4), pp. 491–542

3 ALLEN, J.R., and KENNEDY, K.: 'Conversion of control dependence to data dependence'. Conference record of the tenth ACM symposium on principles of programming languages, 1983, pp. 836–844
4 CHANG, P.P., and MAHLKE, S.A.: 'IMPACT: An architectural framework for multiple-instruction-issue processors'. Proceedings of the 1991 international conference on supercomputing, 1991, pp. 266–275
5 GOODMAN, J.R., and HSU, W.C.: 'Code scheduling and register allocation in large basic blocks'. Proceedings of the 1988 international conference on supercomputing, St Malo, 1988
6 HWU, W.M., and CHANG, P.P.: 'Exploiting parallel microprocessor microarchitectures with a compiler code generator'. Proceedings of the 15th annual international symposium on computer architecture, Honolulu, Hawaii, 1988, pp. 45–53
7 JOUPPI, N.P., and WALL, D.W.: 'Available instruction-level parallel for superscalar and superpipelined machines'. 3rd international symposium on architecture support for programming languages, 1989, pp. 272–282
8 LI, Z.: 'Compiler algorithms for event variable synchronization'. Proceedings of the 1991 ACM international conference on supercomputing, 1991, pp. 85–95
9 MIDKIFF, S.P., and PADUA, D.A.: 'Compiler generated synchronization for DO loops'. Proceedings of the 1986 international Conference on parallel processing, 1986, pp. 544–551
10 MIDKIFF, S.P., and PADUA, D.A.: 'Compiler algorithm for synchronization', *IEEE Trans.*, 1987, **C-36**, (12), pp. 1485–1495
11 POLYCHRONOPOULUS, C.: 'Toward auto-scheduling compilers'. Illinois CSRD report 789, University of Illinois at Urbana-champaign, 1988
12 SU, H.M., and YEW, P.C.: 'On data synchronization for multiprocessors'. Proceedings of the 16th annual international symposium on computer architecture, 1989, pp. 416–423
13 TANG, P., and YEW, P.C.: 'Processor self-scheduling for multiple-nested parallel loops'. Proceedings of the 1986 ACM international conference on parallel processing, 1986, pp. 528–535
14 WOLFE, M.J., and BANERJEE, U.: 'Data dependence and its application to parallel processing', *Int. J. Parallel Programming*, 1987, **16**(2), pp. 137–178
15 WOLFE, M.J.: 'Optimizing supercompiler for supercomputers'. PhD. thesis, University of Illinois at Urbana-champaign, 1982
16 WOLFE, M.J.: 'Multiprocessor synchronization for concurrent loops', *IEEE Software*, 1988, **5**, pp. 34–42

7 Appendix

7.1 Algorithm 1: Appending synchronisation marker (Fig. 4)

Input: An array reference A_1 which is a dependence source in a loop. Suppose the indices of the enclosing loops are $i_1, i_2, \ldots, i_n$.

Output: Synchronisation code for every dependence from A_1.

1. Creat a new variable i.
2. Replace A_1 with $A_1\&i$ in the source program.
3. After the statement that issues A_1, insert

   "i&POST(EV[$i_1, i_2, \ldots, i_n$])".

4. For every dependence sink A_2 from A_1 do the following:
   4.1 Create a new variable j.
   4.2 Replace A_2 with $j\&A_2$ in the source program.
   4.3 Formulate the *mask predicate* ρ and the *contact* ε.
   4.4 Before the statement which issues the sink reference, insert
   'IF ρ WAIT(ε)&j'.

7.2 Algorithm 2: Simple and efficient error prevention method (see Fig. 6)

Input: The instruction I_x in $BB(A)$ and the instruction I_y in $BB(B)$/* Assuming that the instruction I_x in $BB(A)$ is moved to the position y of $BB(B)$*/

Output: SCHE/*FALSE if scheduling fails; TRUE if scheduling succeeds*/ SCHE ← TRUE;
if I_{x+1} is downmarker and ((A precedes B) or ($A = B$ and $I_x < I_y$))
  for each quadruple (three-address code) I between I_x and I_y do
    if I = *Converse* ($I_x + 1$) then SCHE ← FALSE; exit; endif;
  endfor;
endif;
if I_{x-1} is upmarker and ((B precedes A) or ($A = B$ and $I_x > I_y$))
  for each quadruple I between I_x and I_y do
    if I = *Converse*(I_{x-1}) SCHE ← FALSE; exit; endif;
  endfor;
endif;