An Efficient Technique of Instruction Scheduling on a Superscalar-Based Multiprocessor†

Rong-Yuh Hwang
Department of Electronic Engineering, National Taipei Institute of Technology,
Taipei, Taiwan, R. O. C.
E-mail: ryh@en.tit.edu.tw

Abstract
An instruction scheduling approach is proposed for performance enhancement on a superscalar-based multiprocessor. The traditional list scheduling approach is not suitable for the environment because it does not consider the effect of synchronization operation. According to the LBD loop theorem, the system performance is very concerned with the position of synchronization operation. Therefore, the scheduling of synchronization operation has the highest priority in this technique. There are two aspects of performance enhancement for the instruction scheduling approach: 1) converting LBD into LFD, and 2) reducing the damage of LBD. Experimental results show that the enhancement is significant.

1. Introduction

Parallelism has been explored at various processing. The lower the level, the finer the granularity of the software processes. In general, the execution of a program may use several levels. During these levels, loop level and instruction level parallelism are two important issues in the study of parallel compiler and architecture. Do loop is an important source of loop level parallelism. However, some statistical results [2, 21] display that most of the loops belong to Doacross loop; Doall loop, in fact, is an ideal loop. Because Doacross loops exist one or more loop-carried dependences in an iteration, so synchronization operations are inserted into the original program to execute Doacross loop in parallel. There are two actions for synchronization operations. 1) generate a send statement immediately following dependence source S in the program text: Send_Signal(S), and 2) generate a wait statement immediately before dependence sink S': Wait_Signal(S, i−d), where d is dependence distance and i−d is iteration number [6, 11]. An example of do loop is shown in Fig. 1 (a). There are two dependences in this figure. Array element A[I], dependence source, in statement S3 depends on array elements A[I−2] and A[I−1], dependence sinks, in statements S1 and S2. In order to execute this loop in parallel, two pairs of synchronization operations are inserted into this loop. The loop after synchronization operation insertion is shown in Fig. 1 (b). Several synchronization schemes had been proposed to maintain correct order dependence during executing Doacross loop in parallel [1, 5, 10, 20]. On the other hand, several hardware and software techniques were proposed to exploit instruction level parallelism [7, 12, 13]. Bulter et al. have shown that single-instruction-stream parallelism is grater than two [9]. Wall finds that the average parallelism at instruction level is around five, rarely exceeding seven, in an ordinary program [3]. Superscalar processors are designed to exploit more instruction level parallelism in user program. Only independent instructions can be executed in parallel without causing a wait state. In the past, people studies either on loop level or on instruction level independently. However, some new problems need to be resolved if both instruction level and loop level are exploited in the superscalar-based multiprocessor. Loop level parallelism is exploited by executing one iteration for each superscalar processor. In this environment, if we want to execute instruction scheduling, then some problems will be occurred. After instruction scheduling, the instruction sequence is not the same as the original one. Therefore, dependence sink may be scheduled before its corresponding Wait_Signal. This action will have a chance to access stale data [18]. Similarly, the effect is the same as above if a dependence source is behind its corresponding Send_Signal. On the other hand, according
to LBD loop theorem [17], the parallel execution time is very concerned with the number of instruction between a Send_Signal and its corresponding Wait_Signal. The major goal of traditional instruction scheduling is to find more instruction parallelism. However, these instruction scheduling approaches are not suitable for the superscalar-based environment. In this paper, we will propose a new instruction scheduling technique to schedule instruction without accessing stale data and to enhance system performance. The rest of this paper is organized as follows: The preliminary knowledge about this problem is described in Section 2. New instruction scheduling technique is proposed in Section 3. Experimental results are made in Section 4. Finally, a conclusion is made in Section 5.

\[
\begin{align*}
\text{DOACROSS } & I = 1, N \\
\text{DO } & I = 1, N \\
S_2: & G[I-3]=A[I-1]-E[I+2]; \\
\text{ENDDO}
\end{align*}
\]

(a) A Do loop

(b) Synchronization operation insertion for (a)

**Fig. 1 An example of synchronization operation insertion**

### 2. Preliminary knowledge

In this section, we described the related problem and notations used in this paper. Firstly, some notations are defined [18]. \( \text{Src} \) : Dependence source, \( \text{Snk} \) : Dependence sink, \( \text{Sig} \) : A synchronization instruction "Send_Signal", and \( \text{Wat} \) : A synchronization instruction "Wait_Signal". Let \( S_i \) and \( S_j \) be arbitrary statements.

**Definition:** \( S_i \) bef \( S_j \) iff \( S_i \) occurs textually 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 forward iff \( S_i \) bef \( S_j \). Any dependence that is not forward is a backward dependence.

Data dependences include flow dependence, anti-dependence and output dependence [18]. Alternatively, data dependences are separated into two types: forward and backward dependences. The LFD(Lexically Forward Dependence) and LBD(Lexically Backward Dependence) are used as representing the forward and backward dependences respectively. In order to maintain the correct order dependence between \( \text{Sig} \) (\( \text{Wat} \)) and its corresponding \( \text{Src} \) (\( \text{Snk} \)), the synchronization conditions are described as follows [18].

(1) A \( \text{Sig} \) can not precede the corresponding \( \text{Src} \).
(2) A \( \text{Wat} \) can not be behind the corresponding \( \text{Snk} \).

A synchronization marker method had been proposed to maintain the synchronization conditions [18]. For a LFD, the Send_Signal is issued before its corresponding Wait_Signal, so the parallel execution time is equal to the time of executing one iteration. However, for a LBD, the Send_Signal is issued after its corresponding Wait_Signal, so the parallel execution time is equal to \( \lceil \frac{a}{d} \rceil \ast (i-j)+1 \), where \( i \), \( j \), \( d \), and \( l \) represent the position of Send_Signal, Wait_Signal, dependence distance, and number of instruction in an iteration respectively [15, 17, 19]. The parallel execution time of the LFD loop is always smaller than the parallel execution time of the LBD loop. Therefore, performance enhancement is achieved if LBD can be converted into LFD. On the other hand, performance enhancement is also achieved if the position of Send_Signal and its corresponding Wait_Signal are as close as possible. In the next section, the related knowledge will be used to propose a new instruction scheduling technique.

### 3. New instruction scheduling technique

#### 3.1. Construction of Sig, Wat, and Sigwat graph

According to the discussion in the last section, the instruction sequence is not the same as the original one after instruction scheduling. However, if Send_Signal(S) is scheduled before its corresponding dependence source, we might access stale data. Similarly, Wait_Signal can not be behind its dependence. To follow the synchronization conditions, an extra dependence arc is inserted between a synchronization operation and its corresponding dependence event. The three address codes of Fig. 1(b) are as shown in Fig. 2. Observing Fig. 2, the corresponding three address codes of array elements A[I], A[I-1] and A[I-2] are instructions 26, 16, 5 respectively. To avoid accessing stale data, several extra flow dependences need to be created for instructions 11 and 16, 1 and 5, and 26 and 27. Fig. 3 is the data flow graph of Fig. 2. The extra dependence arc insertion is shown in Fig. 3. In this figure, up-triangle is used to represent Send_Signal. Similarly, down-triangle is used to represent Wait_Signal. The number attached in these triangles are used to represent whether the synchronization operations belong to the same pair of synchronization operations. For example, array element A[I] exists two pairs of synchronization operations with array elements A[I-1] and A[I-2]. Therefore, all triangles in Fig. 3 have the same number \( i \). Next, we make some definitions of data flow graph with synchronization operations.
Definition: A Sig (Wat) graph is a subset of a data flow graph. All of its nodes are contiguous and contain one or more Send_Signal (Wait_Signal) instructions.

Definition: A Sigwat graph is a subset of a data flow graph. All of its nodes are contiguous and contain one or more Send_Signal and Wait_Signal instructions.

Therefore, a Sig (Wat) graph represents the statements where each one contains at least one dependence source (sink) and its corresponding Send_Signal (Wait_Signal). Sigwat graph is formed by several cases. 1) If Src and Snk are in the same statement, then this statement is formed a Sigwat graph. 2) Two statements are in the same Sigwat graph if they exist inside loop dependence or use same array index expression, and some synchronization operations consisting of Sig and Wat are in the two statements. For example, statements S3 and S1 in Fig. 1 exist an inside loop dependence, B[I]. A[I] in S3 is Src, and A[I-2] in S1 is Snk. Therefore, statements S3 and S1 will be formed a Sigwat graph shown in Fig. 3. Similarly, statement S2 and Wait_Signal(S3, I-1) are formed a Wat graph shown in Fig. 3.

1: Wait_Signal(S3, I-2);
2: t1 ← 4*I;
3: t2 ← I-2;
4: t3 ← 4*t2;
5: t4 ← A[t3];
6: t5 ← I+1;
7: t6 ← 4*t5;
8: t7 ← E[t6];
9: t8 ← 4+I7;
10: B[t1] ← t8;
11: Wait_Signal(S3, I-1);
12: t9 ← I-3;
13: t10 ← 4+I9;
14: t11 ← I-1;
15: t12 ← 4*I11;
16: t13 ← A[t12];
17: t14 ← I+2;
18: t15 ← 4+I4;
19: t16 ← E[t15];
20: t17 ← I3-I16;
21: G[t9] ← t17;
22: t18 ← B[t11];
23: t19 ← I+3;
24: t20 ← 4*I19;
25: t21 ← C[t20];
26: A[t1] ← t18+t21;
27: Send_Signal(S);

Fig. 2 The three address codes in Fig. 1(b)

3.2. New instruction scheduling technique

According to the definitions above, the corresponding Wat for a Sig graph is in Sigwat or Wat graph. Similarly, the corresponding Sig for a Wat graph is also in a Sigwat or Sig graph. The Sig, Wat, and Sigwat graph do not depend on each other, so the dependence relation in Sig or Wat graphs can be converted into LFD by scheduling Sig (Wat) graphs before (after) all Sigwat graphs. This means that all dependence relations whose synchronization operations are in Sig or Wat graphs can be converted into LFD. A path which starts from a Wat node to its corresponding Sig node in a Sigwat graph is called a synchronization path SP(Wat, Sig). The LBD loop can not be converted into LFD loop if there exists a synchronization path in a Sigwat graph. For such a case, we need to reduce the damage of LBD loop. The synchronization path is the shortest distance from a Wait_Signal to its corresponding Send_Signal. Therefore, the scheduling rule for a Sigwat graph is that the node in a synchronization path needs to be scheduled contiguously. For such scheduling policy, the parallel execution time of LBD is minimum.

Now, we propose a new scheduling approach, which benefits for the parallel execution time. Firstly, we find all synchronization paths SP(Wati, Sigi) in Sigwat graph and sort SP(Wati, Sigi) by the value \((nd_i) \times SP(Wati, Sigi)\) in descending order, where \(d_i\) and \(n\) are the dependence distance for that dependence relation and number of iterations in a loop respectively. If some synchronization paths have the same nodes (i.e. the intersection of nodes with two synchronization paths is not null), then they need to be scheduled simultaneously; otherwise, the distance of Sig and its corresponding Wat increase and the parallel execution time will be increased.

Consider two synchronization paths SP(Wati, Sigi) and SP(Watj, Sigj), if SP(Wati, Sigi) \(\cap\) SP(Watj, Sigj) \(\neq\) \(\emptyset\) and \((nd_i) \times SP(Wati, Sigi) > (nd_j) \times SP(Watj, Sigj)\), then the parallel execution time \(T_{SP_i}\) is larger than \(T_{SP_j}\). For such a case, SP(Wati, Sigi) and SP(Watj, Sigj) need to be scheduled by following the scheduling rule simultaneously, and SP(Wati, Sigi) is scheduled firstly because its parallel execution time is larger than the parallel execution time of SP(Watj, Sigj). After all the synchronization paths are scheduled, the remaining nodes in Sigwat graph are to be scheduled by following the order dependence of data flow graph.
scheduling a Sig graph, we find the position of their corresponding Wait_Signal, and then schedule Send_Signals in Sig graph immediately before its corresponding Wait_Signal. The other nodes in Sig graph are scheduled by the list scheduling. Similarly, to get LFD in Wat graph, the Wait_Signals in Wat graph need to be scheduled after their corresponding Send_Signal. Therefore, we find the position of Send_Signal, which is the corresponding Send_Signal in Wat graph, and schedule Wait_Signal after its corresponding Send_Signal. The remaining nodes in Wat graph are then scheduled. Finally, the nodes which are not in Sig, Wat, and Sigwat graph are scheduled.

\[
\begin{align*}
(1, 2, 3, -) & \quad \text{Wat} \\
(4, 6, 11, -) & \quad \text{Wat2} \\
(5, 7, 12, -) & \quad (1, 4, 8, 23) \quad \text{Wat} \\
(8, 13, 14, -) & \quad (5, 24, -, -) \\
(9, 15, -) & \quad (9, 2, 25, -) \\
(10, 17, -) & \quad (10, -, -, -) \\
(16, 18, 23, -) & \quad (22, 17, -, -) \\
(19, 24, -) & \quad (26, -, -, -) \\
(20, 22, -) & \quad (27, 14, 18, -) \quad \text{Sig} \\
(21, -, -) & \quad (11, 15, 19, -) \quad \text{Wat2} \\
(25, -, -) & \quad (16, -, -) \\
(26, -, -) & \quad (20, 12, -, -) \\
(27, -, -) & \quad (21, 13, -, -) \\
\end{align*}
\]

(a) list scheduling (b) our scheduling result

**Fig. 4 Scheduling result for Fig. 3**

The main advantage of this scheduling algorithm is that it never degrades the system performance. Conversely, it significantly improves the system performance. Now, we illustrate an example to show the scheduling algorithm. In Fig. 3, assuming 4-issue is employed for the superscalar machine, there are following function units: Load/store, adder, shifter, multiplier, and divider. The result of list scheduling with 4-issue in Fig. 3 is listed in Fig. 4 (a). In this figure, we find that nodes 1, 2, 3, 6, 11, 12, 14, 17, and 23 are available. Therefore, nodes 1, 2, and 3 are arranged in an instruction. Node 6 is conflicted with node 3 because of using the same function unit. According to the resource constraint and data flow graph, we can get the scheduled code shown in Fig. 4 (a). Observing Fig. 4 (a), there exist two LBDs. The longest distance from Sig to Wat2 has 12 instructions; therefore, the parallel execution time is \((12 \times N) + 13\), where \(N\) represents \(N\) iterations because the dependence distance for Wat2 is 1. The scheduling result of our approach is listed in Fig. 4 (b). First, we find a synchronization path in Sigwat graph. The synchronization path contains nodes 1, 5, 9, 10, 22, 26, and 27. Therefore, we schedule these nodes in the contiguous instruction. Then the remaining nodes in Sigwat graph are scheduled. To get LFD, the Wait_Signal, node 11, in the Wat graph needs to be scheduled after its corresponding Send_Signal, node 27. Similarly, the remaining nodes in the Wat graph are scheduled. Observing Fig. 4 (b), there exists only one LBD, and the parallel execution time is \(\frac{N+13}{2}\).
time of the traditional list scheduling and our scheduling approach in a loop.

According to the statistics in [14], there are 6 types of DOACROSS loop. These types are control dependence, anti-output dependence, induction variable, reduction operation, simple subscript expression and others that are not included in the above categories. Now, we explain some statistics gathered by using types 3 (induction variable), 4 (reduction operation), 5 (simple subscript expression), and part of type 6.

![Fig. 5 Statistical model](image_url)

### 4.2. Statistical results

The characteristics of these perfect benchmarks are as listed in table 1. In this table, we find that benchmarks FLQ52, QCD, and TRACK are all LBD. And almost all LBDs are flow dependences [19]. We had made a comparison between our instruction scheduling technique and list scheduling [16]. Assuming the type of function unit in the superscalar processor include load/store unit, integer unit, floating-point unit, multiplier, divider, and shifter unit. Multiplier and divider take 3 and 6 cycles respectively, and other function units take one cycle. There are 100 iterations in each loop. Let $T_{X,Y,Z}$ be the parallel execution time, where $x$ is "a" or "b" for list scheduling or new instruction scheduling respectively, $y$ is 2 (4) for 2-issue (4-issue), and $z$ is 1 (2) for the number of each function unit being 1 (2). Now, the statistics are divided into four cases. **Case 1:** 2-issue and the number of each function unit is one. This is used to calculate $T_{a,2,1}$ and $T_{b,2,1}$ of each benchmark. **Case 2:** 2-issue and the number of each function unit is two. This is used to calculate $T_{a,2,2}$ and $T_{b,2,2}$ of each benchmark. **Case 3:** 4-issue and the number of each function unit is one. This is used to calculate $T_{a,4,1}$ and $T_{b,4,1}$ of each benchmark. **Case 4:** 4-issue and the number of each function unit is two. This is used to calculate $T_{a,4,2}$ and $T_{b,4,2}$ of each benchmark.

The statistical result is as shown in table 2. In this table, we find that the performance enhancement of each benchmark is significant. Some observations from this table is made as follows:

1) The parallel execution time of new instruction scheduling for each case is much the same. The reason is that the new instruction scheduling has the shortest synchronization path at any case, and the dependence is serious in the DLX code. Therefore, the parallel execution time is not significant difference between 2-issue and 4-issue.

2) For the list scheduling, $T_{a,2,i}$ of some benchmarks are smaller than their corresponding $T_{a,4,i}$. The reason is that synchronization operations, $Wat$, does not depend on other elements except their corresponding dependence events. In order to increase instruction level parallelism, synchronization operations can be easily scheduled in the list scheduling approach. And this action will lengthen the synchronization path. Therefore, the parallel execution time of list scheduling with 4-issue will be increased.

3) The improved percentage of parallel execution time between list scheduling and new instruction scheduling is shown in table 3. There is a significant improvement in the statistics. The reason is that the distance from a $Wat$ to its corresponding $Snk$ is so far. For an assignment statement, we find that $Snk$ is almost at the first element of the left-hand side of ':='; and the precedence associated with $Snk$ has lower priority. On the other hand, delayed Load technique is employed to effectively use the limited registers. In fact, $Snk$ is a Load instruction. Because of the use of delayed Load, $Snk$ is placed near the end of the assignment statement but its corresponding $Wat$ is far away from it. This means that a significant improvement is possible. In summary, after new instruction scheduling, there are about 83.37% and 85.1% performance enhancement for 2-issue and 4-issue respectively.

### 5. Conclusion

A new instruction scheduling technique is proposed in this paper. This technique has not been considered by any instruction scheduling algorithms. However, the performance enhancement after this new scheduling technique is significant. Two major contributions have been made in this paper: 1) the correct synchronization conditions during instruction scheduling is maintained. 2) The system performance is enhanced by converting LBD loop into LFD loop or reducing the damage of LBD.
Table 1 Characteristics of the Perfect benchmarks

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>FLQ52</th>
<th>QCD</th>
<th>MDG</th>
<th>TRACK</th>
<th>ADM</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>lines parsed by Parafrase</td>
<td>1794</td>
<td>1664</td>
<td>835</td>
<td>1483</td>
<td>3809</td>
<td>9585</td>
</tr>
<tr>
<td>total no. of loops</td>
<td>82</td>
<td>81</td>
<td>20</td>
<td>51</td>
<td>148</td>
<td>382</td>
</tr>
<tr>
<td>No. of Dall loops</td>
<td>27</td>
<td>10</td>
<td>1</td>
<td>22</td>
<td>25</td>
<td>85</td>
</tr>
<tr>
<td>lines generated by DLX compiler</td>
<td>719</td>
<td>173</td>
<td>2926</td>
<td>537</td>
<td>2328</td>
<td>6683</td>
</tr>
<tr>
<td>total no. of LFD</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>total no. of LBD</td>
<td>7</td>
<td>2</td>
<td>32</td>
<td>4</td>
<td>14</td>
<td>59</td>
</tr>
</tbody>
</table>

Table 2 Statistic results

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>2-issue(#FU=1)</th>
<th>2-issue(#FU=2)</th>
<th>4-issue(#FU=1)</th>
<th>4-issue(#FU=2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLQ52</td>
<td>Ta-2-1 2726</td>
<td>Tb-2-1 2730</td>
<td>Ta-2-2 1846</td>
<td>Tb-2-2 1844</td>
</tr>
<tr>
<td>QCD</td>
<td>Ta-4-1 443</td>
<td>Tb-4-1 443</td>
<td>Ta-4-2 1889</td>
<td>Tb-4-2 1884</td>
</tr>
<tr>
<td>MDG</td>
<td>Ta-2-1 5742</td>
<td>Tb-2-1 5743</td>
<td>Ta-2-2 4540</td>
<td>Tb-2-2 4543</td>
</tr>
<tr>
<td>TRACK</td>
<td>Ta-4-1 1846</td>
<td>Tb-4-1 1848</td>
<td>Ta-4-2 1328</td>
<td>Tb-4-2 1326</td>
</tr>
<tr>
<td>ADM</td>
<td>Ta-2-1 116165</td>
<td>Tb-2-1 116165</td>
<td>Ta-2-2 93652</td>
<td>Tb-2-2 93652</td>
</tr>
<tr>
<td>Total</td>
<td>229138</td>
<td>229138</td>
<td>229138</td>
<td>229138</td>
</tr>
</tbody>
</table>

Table 3 Improved percentage for the statistics

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>2-issue(#FU=1)</th>
<th>2-issue(#FU=2)</th>
<th>4-issue(#FU=1)</th>
<th>4-issue(#FU=2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLQ52</td>
<td>89.23% 87.36%</td>
<td>89.74% 88.66%</td>
<td>89.74% 88.66%</td>
<td>89.74% 88.66%</td>
</tr>
<tr>
<td>QCD</td>
<td>34.95% 0.32%</td>
<td>55.37% 47.88%</td>
<td>55.37% 47.88%</td>
<td>55.37% 47.88%</td>
</tr>
<tr>
<td>MDG</td>
<td>88.89% 86.63%</td>
<td>89.67% 88.8%</td>
<td>89.67% 88.8%</td>
<td>89.67% 88.8%</td>
</tr>
<tr>
<td>TRACK</td>
<td>90.14% 86.48%</td>
<td>91.03% 89.89%</td>
<td>91.03% 89.89%</td>
<td>91.03% 89.89%</td>
</tr>
<tr>
<td>ADM</td>
<td>81.97% 79%</td>
<td>82.6% 81.85%</td>
<td>82.6% 81.85%</td>
<td>82.6% 81.85%</td>
</tr>
</tbody>
</table>

References