Register Requirement for Exploiting Loops' Maximum Instruction-Level Parallelism[*]

Jian Wang[†] Andreas Krall M. Anton Ertl

Institut für Computersprachen
Technische Universität Wien
Argentinierstr. 8
A-1040 Vienna, Austria

Abstract

This paper studies the interaction between register requirement and loops' maximum Instruction-Level Parallelism (ILP). First, we present the minimum data dependence graph (MinDDG) of a loop to represent the maximum ILP available in the loop. Then we analyze the influence of register requirement on the data dependence graph and the MinDDG, and on the basis of which we present an approach to estimate the upper bound on register requirement for exposing loops' maximum ILP. Finally, the preliminary experimental results are given to verify our approach.

Keywords: Parallel Processing, Instruction-level Parallelism, Loop Scheduling, Software Pipelining, Register Allocation, Data Dependence Graph

1 Introduction

Exploiting Instruction-Level Parallelism (ILP) within loops has become a key compilation issue for the instruction-level parallel processors like Very Long Instruction Word (VLIW) and superscalar machines [1, 2, 3]. Software pipelining is an efficient compilation technique to exploit ILP for loops, which initiates successive iterations before previous iterations complete [4, 5, 6, 7].

Register Allocation is another key compilation issue [8, 9, 10, 11]. It has been well known that register allocation may introduce anti-dependences due to the re-use of registers, which limit the loops' ILP to be exploited by software pipelining [11, 3].

The interaction between register allocation and loop-free code scheduling has been studied since the mid 1980s [12, 13, 8, 14, 15]. The register allocation for software pipelined loop and lifetime-sensitive software pipelining approaches have been studied by some researchers and some efficient techniques have been proposed [16, 17, 11, 10, 18, 19].

However, the interaction between register requirement and loops' maximum ILP is still less understood and is lately considered in few studies. Mangione-Smith, et al. developed a lower bound on the number of registers needed for a given modulo scheduled loop [20]. Ning and Gao have presented a framework of register allocation for software pipelining by which they deduce

[*]This work was supported by the Lise Meitner Stipendium funded by the Austrian Science Foundation (FWF) and the Austrian Science and Research Ministry.

[†]Email: jian@mips.complang.tuwien.ac.at; Tel: 43-1-588014474; Fax: 43-1-5057838.

the minimum number of registers needed for finding the optimal software pipelined loop [21], but they do not consider the resource constraints. Their result can be used as a lower bound on the register requirement for loops' maximum ILP.

In this paper, we study the interaction between register requirement and loops' maximum ILP from a new perspective. Before software pipelining a loop, we focus on an interesting problem, that is, how many registers are needed for this loop such that its ILP can be maximally exploited? Thus, our result is general, independent of any specific loop schedule and can be used as the upper bound on register requirement for loops' maximum ILP.

Although it has been well known that the re-use of registers introduces new anti-dependence edges to the loop's data dependence graph(LDDG) and limits the loop's ILP, we have found that not all anti-dependences caused by the re-use of registers limit the loop's ILP. Thus, we first present a novel representation of a loop's maximum ILP based on the framework of decomposed software pipelining(DESP) [22, 23] . This representation is called Minimum Data Dependence Graph of a loop (referred to as MinDDG). Those anti-dependence edges which are introduced to LDDG but do not cause new dependence edges to MinDDG do not limit the loop's ILP. Then we theoretically develop an approach to determine the minimum number of registers which is required for no new dependence edge being introduced to MinDDG, which is used as the upper bound.

This paper is organized as follows. The next section first gives an outline of decomposed software pipelining and then present the concept of MinDDG. Section 3 discusses the influence of register requirement on LDDG and MinDDG. In Section 4, we use MinDDG as a basis to develop the upper bound. In Section 5, we give and discuss the preliminary experimental results. We conclude this paper in Section 6.

2 A Representation of Loop's Maximum ILP

The maximum ILP within a program is limited by the data dependences among operations and the available machine resources. The data dependences of a loop-free code can be represented by a Directed Acyclic Graph (DAG). DAG gives a direct representation of maximum ILP within a loop-free code, as it is easy to see, from DAG, which operations can be executed simultaneously (if we have enough machine resources).

However, the data dependences of a loop can not be represented by a DAG, but by a Loop Data Dependence Graph (denoted as LDDG), (O, E, λ, δ), where O is the operation set and E the dependence edge set; the **dependence distance** λ and the **delay** δ are two non-negative integers associated with each edge. For example, $e = (op, op')$ and $(\lambda(e), \delta(e))$ denote that op' can only be issued $\delta(e)$ cycles after the start of the operation op of the $\lambda(e)$th previous iteration [2, 24]. Obviously, the LDDG may include cycles so it does not give a direct representation of maximum ILP within a loop.

This section aims at a direct representation of maximum ILP within a loop under the framework of DEcomposed Software Pipelining (denoted as DESP) [22, 23]. In order to make this paper self-contained, we first give a brief introduction to DESP and some concepts which will be used in the following sections.

2.1 Decomposed Software Pipelining

DESP is a novel modulo scheduling approach, and its idea can be illustrated by Figure 2.1 as an example. First, we modify the LDDG by removing some edges so that the graph becomes acyclic; secondly, we apply the list scheduling technique [1, 2] on the modified graph to generate the software pipelined loop body under the resource constraints, and use the row-number to denote the cycle-number of each operation in the loop body; thirdly, we determine the iteration-number (denoted as column-number in the context of DESP) of each operation such that all data dependences in LDDG are satisfied.

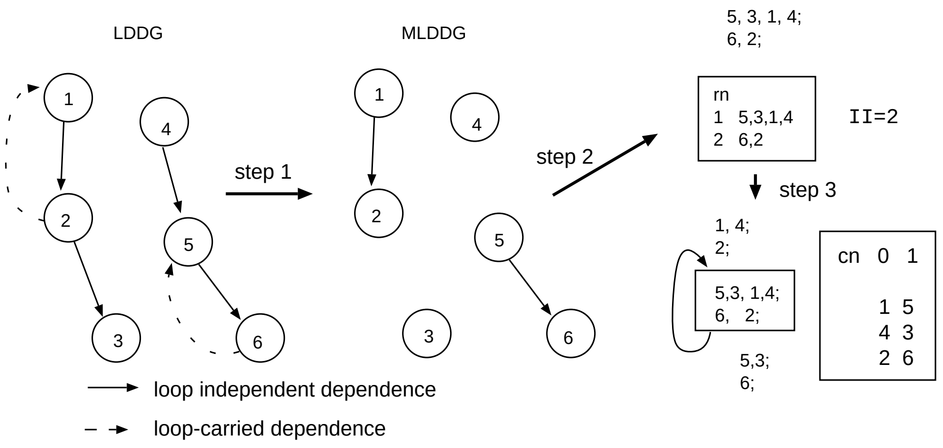


Figure 2.1 Decomposed Software Pipelining

Formally, DESP theoretically decomposes the loop schedule σ into two functions, row-number and column-number.

Definition 2.1 Let $G = (O, E, \lambda, \delta)$ be the LDDG of a loop, and σ a valid loop schedule for G with initiation interval II[1]. We define the row-number rn and the column-number cn, two mappings from O to N (non-negative integer set), such that

$$\sigma(op, 1) = rn(op) + II * (cn(op) - 1) \quad and \quad \sigma(op, i) = \sigma(op, 1) + II * (i - 1).$$

□

Thus, software pipelining can be described below with the concepts of row-number and column-number.

Definition 2.2 (Decomposed Software Pipelining) Let $G = (O, E, \lambda, \delta)$ be the LDDG of a loop, we say that the row-number, rn, and the column-number, cn, are valid for the loop, if and only if the following constraints are satisfied:

1. resource constraints: $\forall op_i, op_j \in O$, if $rn(op_i) = rn(op_j)$, then op_i and op_j must not conflict with respect to the resources[2];

2. dependence constraints:

$$\exists II \in N, \quad rn(op') - rn(op) + II * (\lambda(e) + cn(op') - cn(op)) \geq \delta(e), \quad \forall e = (op, op') \in E.$$

II is called the initiation interval or the length of the software pipelined loop body. The goal of

[1]That is, a new iteration of the loop can be issued every II cycles

[2]Here, we only consider the pipelined operations and the single-cycle operations, but the definition is easily extended to the case of multi-cycle non-pipelined operations.

decomposed software pipelining is to find valid row-number and column-number with minimum II. □

In this paper, we assume that $min(rn(op)) = 1$ and $min(cn(op)) = 0$. In previous papers [22, 23, 25], we have proven the following theoretical results.

Theorem 2.1 For a given LDDG, suppose we have constructed row-number rn which satisfies the resource constraints. We can construct column-number cn such that the data dependence constraints are also satisfied, if and only if, for each cycle C of the LDDG,

$$\sum_{\forall e \in C} \tau(e) \leq 0$$

where $\tau(e) = -\lambda(e) + \lceil (\delta(e) + rn(op) - rn(op'))/II \rceil$, $e = (op, op')$. □

Theorem 2.1 implicitly points out the following corallary.

Corallary 2.1 For a LDDG without cycle, if we have constructed row-number taking into account the resource constraints, then we can always construct column-number such that the data dependence constraints are also satisfied. □

The column-number is an important parameter to control the register requirement of each variable. For example, if the longest definition-use path[3] of variable x is from op_i to op_j, then $cn(op_j) - cn(op_i)$ gives the estimate of the lifetime of x and thus determines the register requirement of x.

2.2 Minimum DDG

Under the framework of DESP, we now deduce a direct representation of maximum ILP available within a loop. As mentioned in the last subsection, DESP approach includes three steps. The step 2 construct row-number and determines initiation interval (II). Provided we have enough hardware resources, the II is only limited by the modified LDDG (referred to as MLDDG). Thus, we expect that the step 1 can generate a MLDDG which is of the minimum height and the minimum number of edges and satisfies the following two conditions:

1. MLDDG is acyclic so that the step 2 can work;
2. MLDDG is sensitive to the condition of Theorem 2.1 so that the step 3 can work.

Formally, we give the definitions as follows:

Definition 2.3 Let $G = (O, E, \lambda, \delta)$ be the LDDG of a loop, we remove some edges from G and get $MLDDG$, $MLDDG$ is valid for the loop if and only if

1. $MLDDG$ is acyclic; and
2. for any schedule σ on $MLDDG$, we can always find a non-negative integer II such that, for each cycle C of G, $\sum_{\forall e \in C} \tau(e) \leq 0$, where $\tau(e)$ is defined in Theorem 2.1. □

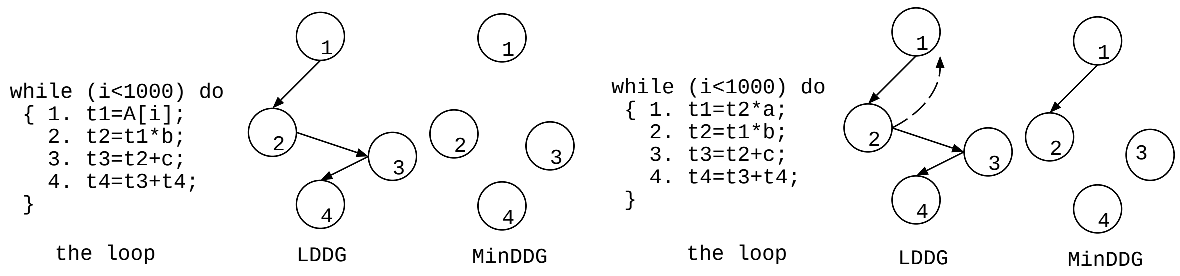
Definition 2.4 Let $G = (O, E, \lambda, \delta)$ be the LDDG of a loop, we remove some edges from G and get $MinDDG$, $MinDDG$ is called the minimum DDG of the loop if and only if

1. $MinDDG$ is valid for the loop; and
2. the height of $MinDDG$ is not greater than the minimum II of the loop; and

[3]A definition-use path is defined as a path from the operation writing a variable to the operation referencing the variable in LDDG.

3. if we remove any edge from $MinDDG$, then the resulting graph is not valid for the loop any more. □

The performance of a software pipelined loop is evaluated by its initiation interval (II) and the II is actually the length of the pipelined loop body. The pipelined loop body is generated with a list scheduling to the minimum DDG (referred to as MinDDG) in the second step of decomposed software pipelining, as shown in Figure 2.1. Provided we have enough available machine resources, the maximum ILP of a loop is represented by its MinDDG. Figure 2.2 gives two examples. For the first without dependence cycle, its MinDDG includes no dependence edge so the ILP of the loop is maximally exposed. For the second with a dependence cycle (1-2-1), its MinDDG only includes one dependence edge so the ILP of the loop is also maximally exposed.



```
while (i<1000) do
 { 1. t1=A[i];
   2. t2=t1*b;
   3. t3=t2+c;
   4. t4=t3+t4;
 }
```

```
while (i<1000) do
 { 1. t1=t2*a;
   2. t2=t1*b;
   3. t3=t2+c;
   4. t4=t3+t4;
 }
```

(a) The case without dependence cycle (b) The case with dependence cycle

Figure 2.2 Examples of Loops' LDDGs and MinDDGs

In our previous papers [22, 23], we have developed a method for approximately generating a MinDDG from the LDDG of a loop:

(1) find out all strongly connected components (SCCs) in the LDDG, remove all edges which are not included in the SCCs;

(2) under the unlimited resource constraints, generate a software pipelined loop for the SCCs, denoted as (rn_0, cn_0);

(3) for each edge $e = (op_i, op_j)$ of SCCs, if $rn_0(op_j) - rn_0(op_i) < \delta(e)$, then remove e from the SCCs.

The remaining graph satisfies the first and the second conditions of Definition 2.4, which approximately represents the MinDDG.

3 Influence of Register Requirement on LDDG and MinDDG

This section discusses the influence of register requirement on LDDG and MinDDG, and addresses our important observation, i.e. not all anti-dependences caused by the re-use of registers limit the ILP avaliable in a loop. Those anti-dependences which are introduced to LDDG but do not cause new dependences to MinDDG do not limit the ILP avaliable in a loop which can be exploited by software pipelining.

In the case of register allocation for a software pipelined loop, more than one register could be allocated to one variable. For simplicity, we assume that the registers are well-distributed to different iterations if one variable is allocated with more than one register, as shown in Figure 3.1.

Thus, the anti-dependence edges caused by the re-use of registers are introduced to LDDG in such a way that,

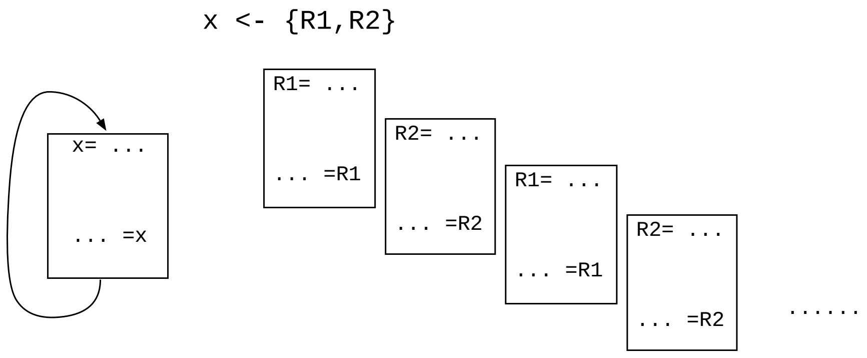


Figure 3.1 Register Allocation for Software Pipelined Loops

(1) If the variable v is first defined by op_i and then used by op_j in the original loop body – we call (op_i, op_j) a loop-independent dependence, and v is allocated with K_v registers, then one anti-dependence edge with the dependence distance of K_v (i.e. $\lambda((op_j, op_i)) = K_v$) is introduced to LDDG from op_j to op_i (e.g. the variable x in Figure 3.2);

(2) If the variable v is first used by op_j and then defined by op_i in the original loop body – we call (op_i, op_j) a loop-carried dependence, and v is allocated with K_v registers, then one anti-dependence edge with the dependence distance (i.e. $\lambda((op_j, op_i)) = K_v - 1$) is introduced into LDDG from op_j to op_i (e.g. the variable y in Figure 3.2).

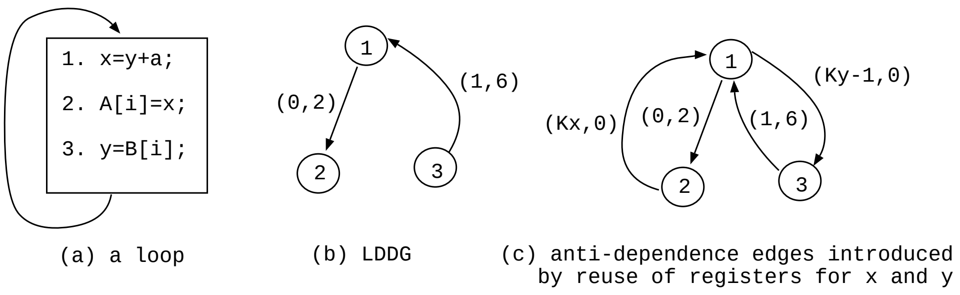


Figure 3.2 Anti-Dependence Edges Caused by Re-Use of Registers

Now we use Figure 3.3 as an example to discuss how the anti-dependences caused by the re-use of registers are introduced to MinDDG and its relationship with the register requirement.

In Figure 3.3, (a) is the LDDG and its MinDDG, and we only focus on variables x and y. x is defined by op_1 and used by op_3, y is defined by op_4 and used by op_5. The MinDDG includes no dependence edges.

First, if we allocate one register to x and another to y, i.e. $K_x = 1$ and $K_y = 1$, one anti-dependence edge (3,1) with the dependence distance of 1 and another (5,4) with the dependence distance of 1 are introduced to the LDDG, as shown in LDDG1 of (b). The new edge (3,1) causes a new cycle 1-2-3-1 and thus introduces two new edges, (3,1) with the dependence-distance of 0 and (1,2) with dependence-distance of 2, to the MinDDG, as shown in MinDDG1 of (b). The new edge (5,4) also causes a new cycle 4-5-4 but does not introduce any new edge to the MinDDG. Therefore, we say that the anti-dependence edge (3,1) caused by the re-use of register limits the loop's ILP, but the edge (5,4) does not.

Secondly, we still allocate one register to y but two to x. The dependence distance of the edge (3,1) becomes 2, thus one new edge (3,1) is only introduced to the MinDDG, as shown in LDDG2 and MinDDG2 of (c). We can observe that, more registers allocated to the variables,

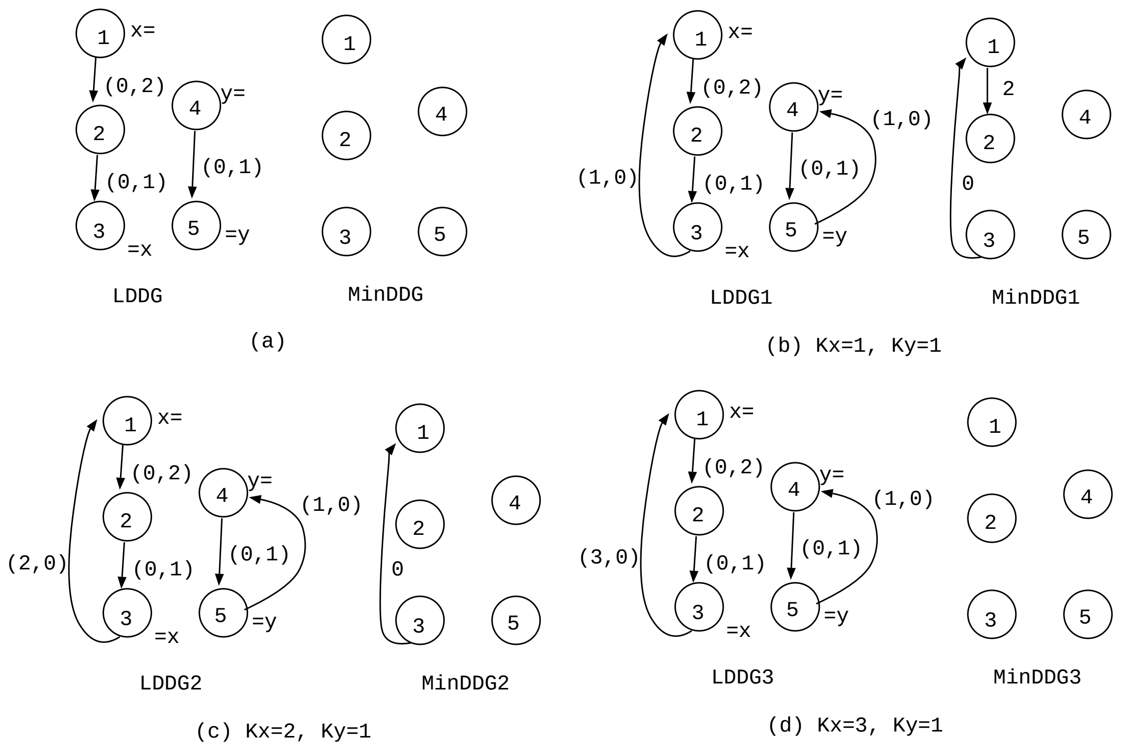


Figure 3.3 Influence of Register Requirement on LDDG and MinDDG

less new dependence edges introduced to the MinDDG and thus less limitation on the loop's ILP.

Finally, we increase the registers of x by 1, the dependence distance of the edge (3,1) becomes 3. Now, any new dependence edge is not introduced to the MinDDG, as shown in LDDG3 and MinDDG3 of (d).

The above observation gives us a hint that, for each variable[4] of a loop, there exists a minimum register requirement for that no new dependence edge is introduced to the loop's MinDDG and thus the loop's ILP can be maximally exposed.

The next section will quantitively estimate this minimum register requirement, which is the upper bound on the register requirement for exploiting loops' maximum ILP.

4 The Upper Bound on Register Requirement for Loops' Maximum ILP

We are ready to estimate the upper bound on register requirement for loops' maximum ILP. First, we give a formal description of the problem. Then an approximate solution is developed.

4.1 The Formal Description

We first define a new graph, $LDDG^+$, to conclude the analysis of the last section.

[4]In this paper, we only consider the loop-variant variables.

Definition 4.1 Given the LDDG, (O, E, λ, δ), of a loop, we define $LDDG^{+} = (O, E \cup E_{lid} \cup E_{lcd}, \lambda, \delta)$, where

(1) $E_{lid} = \{(op_i, op_j) | op_i, op_j \in O;$ op_j first defines a variable and then op_i uses the variable in the loop body. $\}$; $E_{lcd} = \{(op_i, op_j) | op_i, op_j \in O;$ op_i first uses a variable and then op_j defines the variable in the loop body. $\}$;

(2) $\forall e \in E_{lid} \cup E_{lcd}, \delta(e) = 0;$

(3) $\forall e \in E_{lid}, \lambda(e) = K_v$; $\forall e \in E_{lcd}, \lambda(e) = K_v - 1$. Where K_v is the number of registers allocated to v and v is the corresponding variable. □

Figure 3.2 is an example, where (b) is LDDG and (c) is $LDDG^{+}$. Actually, $LDDG^{+}$ is the graph extended by all anti-dependence edges caused by the re-use of registers to LDDG.

Now, given $LDDG = (O, E, \lambda, \delta)$, let K_v represent the number of registers allocated to v, we can construct $LDDG^{+} = (O, E \cup E_{lid} \cup E_{lcd}, \lambda, \delta)$ and $MinDDG = (O, E_{min}, \delta)$. Any decomposed software pipelined loop $\{II, rn, cn\}$ which is based on the MinDDG should satisfy the data dependence constraints below:

$$\begin{aligned} rn(op_j) - rn(op_i) + II * (\lambda(e) + cn(op_j) - cn(op_i)) \geq \delta(e),\ \forall e = (op_i, op_j) \in E; \\ rn(op_j) - rn(op_i) \geq \delta(e), \forall e = (op_i, op_j) \in E_{min}; \\ rn(op_j) - rn(op_i) + II * (K_v + cn(op_j) - cn(op_i)) \geq 0, \forall (op_i, op_j) \in E_{lid}; \\ rn(op_j) - rn(op_i) + II * (K_v - 1 + cn(op_j) - cn(op_i)) \geq 0, \forall (op_i, op_j) \in E_{lcd}; \end{aligned}$$

If II and rn are given, then the minimum register requirement can be found by solving the following integer programming problem.

Definition 4.2 Given the LDDG, (O, E, λ, δ), of a loop, if we have found II and rn, let $LDDG^{+} = (O, E \cup E_{lid} \cup E_{lcd}, \lambda, \delta)$ and $MinDDG = (O, E_{min}, \delta)$, then the problem of finding the minimum register requirement can be modelled as an integer programming problem as follows:

$$\min \sum_{\forall v \in V} K_v$$

Subject to

$$\begin{aligned} cn(op_j) - cn(op_i) \geq -\lambda(e) + \lceil (\delta(e) - rn(op_j) + rn(op_i))/II \rceil, \forall e = (op_i, op_j) \in E - E_{min}; \\ cn(op_j) - cn(op_i) \geq -\lambda(e), \forall e = (op_i, op_j) \in E_{min}; \\ K_v + cn(op_j) - cn(op_i) \geq \lceil (rn(op_i) - rn(op_j))/II \rceil, \forall (op_i, op_j) \in E_{lid}; \\ K_v + cn(op_j) - cn(op_i) \geq 1 + \lceil (rn(op_i) - rn(op_j))/II \rceil, \forall (op_i, op_j) \in E_{lcd}; \\ K_v, cn(op)\ integers,\ \forall op \in O, u \in V. \end{aligned}$$

Where V is the set of all loop-variant variables. The minimum register requirement corresponding to $\{II, rn\}$ is denoted as $RR_{min}(II, rn)$. □.

In order to find the upper bound of register requirement, we can not assume that II and rn are known. Generally, the problem of finding the upper bound can be described as follows:

Definition 4.3 Given the LDDG, (O, E, λ, δ), of a loop, let $LDDG^{+} = (O, E \cup E_{lid} \cup E_{lcd}, \lambda, \delta)$ and $MinDDG = (O, E_{min}, \delta)$, the upper bound of register requirement, RR^{up}, is defined as

$$RR^{up} = \max_{II,rn}(RR_{min}(II, rn)).$$

Where II and rn satisfy

(1) $II_{low} \leq II \leq II_{up}$, II_{low} is the lower bound on the initiation interval and II_{up} the upper bound;

(2)$rn(op_j) - rn(op_i) \geq \delta(e),\ \forall e = (op_i, op_j) \in E_{min}$;

(3) For any cycle C of the LDDG,

$$\sum_{\forall e=(op_i,op_j)\in C} (-\lambda(e) + \lceil(\delta(e) + rn(op_i) - rn(op_j))/II\rceil) \leq 0.$$

□

II_{up} can be determined by the length of the original loop body. II_{low} can be determined by the critical cycle of the LDDG and the usage of the critical resources in the loop body.

The following three theorems are important to develop a solution. In order not to make the paper too long, we cut off their proofs. Readers can refer to our detailed research report [26] for more.

Theorem 4.1 If II and rn satisfy that, for any cycle C of the LDDG,

$$\sum_{\forall e=(op_i,op_j)\in C} (-\lambda(e) + \lceil(\delta(e) + rn(op_i) - rn(op_j))/II\rceil) \leq 0.$$

Then $RR_{min}(II, rn)$ exists.□

Theorem 4.2 Given a loop, RR^{up} is the minimum register requirement which is sufficient and necessary for no new dependence edge being introduced into the MinDDG. □

Theorem 4.3 The constraint matrix in the integer programming problem of Definition 4.2 is totally unimodular. □

Theorem 4.3, which is directly derived from the work of Ning and Gao [21, 27], points out that the integer programming problem of Definition 4.2 can be solved as a linear programming problem and the optimal solution is guaranteed to be integral. Therefore, in order to solve our integer programming problem, we can use general linear programming algorithms such as simplex, ellipsoid or interior point methods [28, 29, 30]. Also, Ning and Gao presented a more efficient algorithm whose computation complexity is $O(n^3 \log n)$, where n is the number of nodes in LDDG.

4.2 The Approximate Solution

The exact solution to the problem of finding the upper bound remains open. In this subsection, we develop an approximate solution. The "upper bound" we find may be greater than the real upper bound.

First, we compute II by the following formula

$$II = \max_{\forall C\in LDDG}\lceil \sum_{e\in C} \delta(e) / \sum_{e\in C} \lambda(e)\rceil.$$

That means we use the lower bound on initiation interval which is determined by the critical cycle of the LDDG.

Secondly, note that, for each edge (op_i, op_j) of LDDG,

$$-II + 1 \leq rn(op_i) - rn(op_j) \leq II - 1.$$

We consider the worst case where we set $rn(op_i) - rn(op_j) = II - 1$.

Thirdly, for each edge (op_i, op_j) in $LDDG^+$ but not in the LDDG, in order to get a more precise solution, we do not simply set $rn(op_i) - rn(op_j) = II - 1$; instead, set $\lceil (rn(op_i) - rn(op_j))/II \rceil$ to 1 or 0 in terms of the situation of the edges in the LDDG. Here we denote $\lceil (rn(op_i) - rn(op_j))/II \rceil$ as $G(op_i, op_j)$. $G(op_i, op_j)$ may take 1 or 0.

Thus, an approximate upper bound can be found as follows:

$$\min \sum_{\forall v \in V} K_v$$

Subject to

$$\begin{aligned}
cn(op_j) - cn(op_i) \geq -\lambda(e) + \lceil (\delta(e) + II - 1)/II \rceil, \forall e = (op_i, op_j) \in E - E_{min};\\
cn(op_j) - cn(op_i) \geq -\lambda(e), \forall e = (op_i, op_j) \in E_{min};\\
K_v + cn(op_j) - cn(op_i) \geq G(op_i, op_j), \forall (op_i, op_j) \in E_{lid};\\
K_v + cn(op_j) - cn(op_i) \geq 1 + G(op_i, op_j), \forall (op_i, op_j) \in E_{lcd};\\
K_v, cn(op)\ integers,\ \forall op \in O, u \in V.
\end{aligned}$$

Where V is the set of all loop-variant variables. $LDDG = (O, E, \lambda, \delta)$, $LDDG^+ = (O, E \cup E_{lid} \cup E_{lcd}, \lambda, \delta)$ and $MinDDG = (O, E_{min}, \delta)$,

Obviously, the above integer programming problem is of the same properties as that of Definition 4.2, so it can be also solved efficiently.

Finally, we should point out that, for some cycle, if we set $rn(op_i) - rn(op_j) = II - 1$ for all its edges, then no solution exists since the condition of Theorem 4.1 is not satisfied. In this case, we should carefully handle the cycle. In our current method, we only set $rn(op_i) - rn(op_j) = II - 1$ for loop-independent dependence edges, while compute $rn(op_i) - rn(op_j)$ for loop-carried dependence edges to make the condition of Theorem 4.1 satisfied.

5 Preliminary Experimental Results

We select six examples to verify our approach. Except for example 1(as shown in Figure 5.1(1)), the other five examples are selected from the Livermore benchmarks. As our preliminary experiments are mainly conducted by a manual simulation, we try to select some simple loops in a random way. The machine model we use in the experiments is shown in Figure 5.1(2).

The results are presented in Table 1. In the last column, we give the register requirement for generating an optimal software pipelined loop by DESP software pipelining approach. The upper bound we find approach to the real register requirement for exploiting loops' maximum ILP.

It should be pointed out that, in this paper, we do not address another important problem – which variables can share the same registers. We only address the register requirement of

The Original Loop:

```
for i=1 to n do
s=s+a[i]
a[i]=s*s*a[i]
enddo
```

The Code of the Loop Body:

```
1. t0=t0+1;
2. t1=a[t0];
3. s=s+t1;
4. t2=s*s;
5. t3=t1*t2;
6. a[t0]=t3
```

(1) The Loop of Example 1

Pipeline	Number	Operation	Latency
Memory port	2	Load	13
		Store	1
Address ALU	2	Add/Sub	1
Adder	1	FAdd/FSub	1
		IAdd/ISub	1
Multiplier	1	FMUL	2
		IMUL	2

(2) The Machine Model

Figure 5.1 Example 1 and the Machine Model

Table 1. Register Requirement for Loops' Maximum ILP

Example	L	II	The upper bound	DESP
1	20	2	28	27
2	22	3	41	39
3	17	1	30	30
4	18	3	22	21
5	16	1	29	29
6	17	2	29	27

note 1: L = the length of the longest dependence path in the loop body.
note 2: II = the estimated initiation interval.

each variable and consider the loop's register requirement as the sum of all variables' register requirements. This simple approach is efficient for the short loops and the well-software pipelined loops, where the operations from different iterations are fully overlapped and the live ranges of all variables interfere. For the big loops, however, our simple approach should be extended in the following two ways:

1. Construct the weighted interference graph, the weight on each node represents the register requirement of the corresponding variable. Thus, the number of registers can be counted by the conventional method [9];

2. Check any two variables whose live ranges do not interfere: Supposing they use the same registers, check whether new dependence edges is introduced to the MinDDG.

6 Conclusion

In this paper, we first present the minimum data dependence graph (MinDDG) of a loop to represent the maximum ILP available in the loop and analyze the influence of register requirement on the data dependence graph and the MinDDG, then on the basis of which we present an approach to estimate the upper bound on register requirement for exposing loops' maximum ILP. The preliminary experimental results show the efficiency of our approach.

References

[1] J.A. Fisher, D. Landskov, and B.D. Shriver. Microcode compaction: Looking backward and looking forward. In *proceedings of 1981 National Computer Conference*, 95-102 1981.

[2] F. Gasperoni. Compilation techniques for vliw architectures. Technical Report TR435, New York University, March 1989.

[3] B. R. Rau and J.A. Fisher. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing*, 7(1), January 1993.

[4] B.R. Rau and C.D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *proceedings of the 14th International Symposium on Microprogramming and Microarchitectures (MICRO-14)*, pages 183–198, October 1981.

[5] A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. In *proceedings of European Symposium on Programming, Lecture notes in Computer Science, No.300*, pages 221 –235. Spring-Verlag, June 1988.

[6] P. Y. T. Hsu. *Highly Concurrent Scalar Processing.* PhD thesis, University of Illinois, Urbana-Champaign, 1986.

[7] A.E. Charlesworth. An approach to scientific array processing: The architecture design of the ap-120b/fps-164 family. *Computer*, pages 18–27, September 1981.

[8] D.G. Bradlee, S. J. Eggers, and R.R. Henry. Integrated register allocation and instruction scheduling for riscs. In *proceedings of the 4th International Conference on ASPLOS*, 1991.

[9] G. J. Chaitin. Register allocation and spilling via graph coloring. In *proceedings of ACM SIGPLAN Symp. on Compiler Construction*, 1982.

[10] L.J. Hendren, G.R. Gao, E. R. Altman, and C. Mukerji. Register allocation using cyclic interval graph: A new approach to an old problem. Technical Report ACAPS Technical Memo 33, McGill University, 1992.

[11] B. R. Rau, M. Lee, P.P. Tirumalai, and M.S. Schlansker. Register allocation for software pipelined loops. In *proceedings of PLDI*, 1992.

[12] R.F. Touzeau. A fortran compiler for the fps-164 scientific compute. In *proceedings of ACM SIGPLAN Symposium on Compiler Construction*, 1984.

[13] J. R. Goodman and W. Hsu. Code scheduling and register allocation in large basic blocks. In *proceedings of International Conference on Supercomputing*, 1988.

[14] S.A. Mahlke, W.Y. Chen, P.P. Chang, and W.W. Hwu. Scalar program performance on multiple-instruction-issue processors with a limited number of registers. In *proceedings of the 25th HAWAII International Conference on System Sciences*, January 1992.

[15] Wolfgang Ambrosch, M. Anton Ertl, Felix Beer, and Andreas Krall. Dependence-conscious global register allocation. In *proceedings of PLSA*, April 1994.

[16] C. Eisenbeis, W. Jalby, and A. Lichnewsky. Compile-time optimization of memory and register usage on the cray-2. In *proceedings of the second Workshop on Languages and Compilers*, 1989.

[17] M.S. Lam. *A Systolic Array Optimizing Compiler.* PhD thesis, CMU, 1987. CMU-CS-87-187.

[18] R. Huff. Lifetime-sensitive modulo scheduling. In *proceedings of ACM SIGPLAN PLDI*, pages 258–267, June 1993.

[19] Jian Wang, Christine Eisenbeis, and Philippe Canalda. Using timed petri net to analyze the maximum computation rate for loop programs. In *proceedings of the 3rd International Conference for Young Computer Scientists*, July 1993.

[20] William Mangione-Smith, S. G. Abraham, and E. S. Davidson. Register requirements of pipelined processors. In *proceedings of 1992 ACM International Conference on Supercomputing*, 1992.

[21] Qi Ning and Guang R. Gao. A novel framework of register allocation for software pipelining. Technical Report ACAPS Technical Memo 42, McGill University, 1993.

[22] Jian Wang and Christine Eisenbeis. Decomposed Software Pipelining: A new approach to exploit instruction level parallelism for loop programs. In Michel Cosnard, Kemal Ebcioglu, and Jean-Luc Gaudiot, editors, *proceedings of IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 3–15. IFIP, North-Holland, January 1993.

[23] Jian Wang and Christine Eisenbeis. Decomposed Software Pipelining. Reseach Repport RR-1838, INRIA-Rocquencourt, France, 1993.

[24] Bogong Su and Jian Wang. Loop-carried dependence and the general URPR software pipelining approach. In *proceedings of the 24th Annual Hawaii International Conference on System Sciences*, pages 366–372. IEEE and ACM, January 1991.

[25] Jian Wang, Christine Eisenbeis, Martin Jourdan, and Bogong Su. Decomposed Software Pipelining: A new perspective and a new approach. *International Journal of Parallel Programming*, 22(3):357–379, 1994.

[26] Jian Wang, Andreas Krall, and M. Anton Ertl. Register requirement for exposing loops' maximal instruction-level parallelism. Technical Report TR 1851/94/3, Institut für Computersprachen, TU Wien, Austria, 1994.

[27] Q. Ning and G.R. Gao. A novel framework of register allocation for software pipelining. In *proceedings of POPL*, January 1993.

[28] V. Chvatal. *Linear Programming.* W.H. Freeman and Company, 1983.

[29] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, (4), 1984.

[30] L.G.. Khachian. A polynomial algorithm in linear programming. *Soviet Math. Doklady*, (20), 1979.