Pipelining-Dovetailing: A Transformation to Enhance Software Pipelining for Nested Loops*

Jian Wang and Guang R. Gao

School of Computer Science
McGill University
3480 University Street
Montréal, QC
Canada, H3A 2A7
email: {*jwang, gao*}*@acaps.cs.mcgill.ca*

Abstract. The objective of software pipelining is to generate code which can maximally exploit *instruction-level parallelism* (ILP) in modern multi-issue processor architectures, such as VLIW and superscalar processors. Since the amount of ILP is usually fixed to a small number, four - eight, using state-of-the-art software pipelining scheduling techniques, modern compilers have been able to schedule instructions in a small window of successive iterations and keep the machine resources usefully busy. To maximally take advantage of software pipelining, it is beneficial if the number of iterations of the loops to be software pipelined is large (called *trip counts* in this paper). Therefore, software pipelining of nested loops becomes important, especially when the innermost loops have smaller trip counts.

This paper presents a loop transformation which extends software pipelining from the innermost loops to the enclosing loop nests. Unlike some popular loop transformation techniques (e.g. *unimodular transformation*) targeted to multi-processor machines (where the goal has been to maximally expose loop-level parallelism i.e. the transformed loop nests have maximum number of doall loops), the goal of our transformation, *pipelining-dovetailing*, is to extend the software pipelining of the innermost loop to the surrounding loop nests. Thus all iterations of the loop nests can be smoothly software pipelined through, and the number of effective trip counts is maximized. We also define the condition under which pipelining-dovetailing is valid. As a result, a software pipelining framework is derived for loop nests which integrates software pipelining and pipelining-dovetailing together.

Keywords: Instruction-Level Parallelism, Fine-Grain Parallelism, Software Pipelining, Loop Scheduling, Nested Loop, Very Long Instruction Word(VLIW), Superscalar

* This work was supported by research grants from NSERC, Micronet – Network Centers of Excellence, Canada.

1 Introduction

Exploiting Instruction-Level Parallelism [1] for loop programs has become a major challenge in the design of optimizing compilers for high-performance computer architectures. To this end, software pipelining has been proposed to schedule instructions from several consecutive iterations of an innermost loop for overlapped execution. Over the past decade, software pipelining has been widely studied and it is now successfully used in modern compilers to effectively exploit instruction-level parallelism under the resource and register constraints [2, 3, 4, 5, 6, 7, 8, 9].

Up to now, most proposed software pipelining approaches have only focused on the innermost loops and little work has been done to apply software pipelining across a whole loop nest[2]. For a uniprocessor architecture, which has been the primary target of software pipelining, the available hardware parallelism is quite limited and software pipelining of the innermost loops is quite effective. Note that in order to take advantage of software pipelining, it is desirable that a loop to be software pipelined should have a large number of iterations (called *trip counts* in this paper). In many applications the innermost loops do have reasonably large trip counts, making software pipelining beneficial.

There are three reasons that software pipelining of outer loops is becoming increasingly important. First, since the speeds of innermost loops with long trip counts have already been improved significantly by software pipelining, handling of those with smaller trip counts may become important – an incentive to extend software pipelining to outer loops. Secondly, modern compilers may introduce loop transformations such as *tiling* to improve data locality, as a result the transformed loop tiles may have a much smaller trip count at the innermost level. Finally, for multiprocessor machines, a loop nest may be partitioned to different processors, consequently the loops to be executed on one processor may also have smaller trip counts at the innermost level.

In this paper, we study a software pipelining method applicable to nested loops. A main feature of our methodology is to retain the existing (and quite mature) framework of software pipelining techniques for innermost loops, and investigate how such a framework can be naturally *extended* from the innermost loops to the enclosing loops. The key question to be answered is how can such an extension be done in a smooth and efficient fashion ?

At the center of our approach there is the following observation: for a majority of innermost loops to which software pipelining has been successfully applied, there is enough instruction-level parallelism across a few iterations of the loop body to fully utilize the hardware parallelism within the resource and register constraints in modern uniprocessor architectures. Therefore, our objective here is not to further "widen" the parallelism of the software pipeline (already being exploited very well for the innermost loop) by globally scheduling instructions from several outer-loop iterations together. Instead, taking a 2 dimensional loop nest as an example, we will apply software pipelining to the innermost loop as

[2] In this paper, we use "loop nest" and "nested loop" interchangeably.

before, but try to "dovetail" the end of one outer loop iteration – an already software pipelined loop body – to the beginning to the next (also software pipelined). In other word, we would like to effectively stretch the whole loop nest as a long software pipeline with a fixed width (the same as the innermost one), such that the total trip count is now increased to the number of iterations of the entire loop nest.

This paper will present a new transformation – called *pipelining-dovetailing* to make the above "dovetail" possible. However, this transformation is quite different from some popular loop transformation techniques (e.g. *unimodular transformations*) targeted to multiprocessor machines where the goal has been to *maximally* expose loop-level parallelism i.e. the transformed loop nests should have maximum numbers of **doall** loops [10, 11, 12, 13]. The goal of our transformation is to effectively extend the software pipelining of the innermost loop to the surrounding loop nests so that all iterations of the loop nests can be smoothly software pipelined through, and the number of effective trip counts is maximized. Our loop transformation does not need to make the innermost loop or any of the enclosing loops into **doall** loops! In other word, our loop transformation does not perform any global scheduling and that explains the simplicity of the resulting algorithm.

It is important to note that we should view our techniques and previous loop transformation techniques (e.g. unimodular transformation, tiling, etc.) as complementary to each other — each can be applied for their own purpose at a different phase in a compiler. For example, one can imagine that some unimodular transformation or tiling may be performed at an earlier phase of compilation for parallelization or storage optimization, then our technique can be applied at the later code generation phase.

This paper is organized as follows: The next section gives a brief introduction to software pipelining and the data dependence representation of loop nests, making this paper self-contained. Section 3 consists of the motivating examples which highlight the principle of pipelining-dovetailing and discusses the relation with unimodular loop transformation. In Section 4, we present a sufficient condition under which pipelining-dovetailing is valid. The performance analysis is given in Section 5. Section 6 compares our approach with the related work. People assume the last section is a conclusion.

2 Background

2.1 Software Pipelining

Software pipelining is an efficient instruction-level loop scheduling technique. It tries to overlap the execution of operations from several consecutive iterations of a loop under the constraints of data dependences and resources. Figure 2.1 gives an example of software pipelining. A software pipelined loop consists of three parts: the prelude and the postlude which are executed exactly one time, and the software pipelined loop body[3] which may be executed many times. The length of the software pipelined loop body is called *initiation interval(II)*. $II = 1$ for this example.

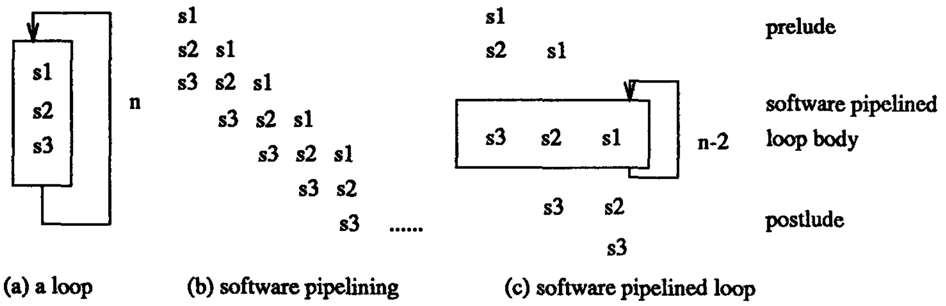


Figure 2.1 An example of software pipelining

Software pipelining can be combined with a loop unrolling transformation to improve its performance. We often consider a software pipelining technique with loop unrolling in such a way that we first unroll the loop and then software pipeline the unrolled loop without any unrolling.

The data dependences of a single-level loop can be represented by a Data Dependence Graph (DDG), (O, E, λ, δ), where O is the operation set, E is the dependence edge set, λ is the *dependence distance* and $\delta$is the *delay* . λ and δ are two non-negative integers associated with each edge. For example, $e = (op, op')$ and $(\lambda(e), \delta(e))$ denote that op' can be only issued $\delta(e)$ cycles after the start of the operation op of the $\lambda(e)$th previous iteration [14]. Not more formally, we can define software pipelining without loop unrolling as follows:

Construct a loop schedule σ, a mapping function from $O \times N$ to N (N is the positive integer set), $\sigma(op, i)$ denotes the execution cycle where the instance of operation op of ith iteration is issued. If the following constraints are satisfied:

1. Resource constraints: In each cycle, the same resource can not be used more than once.

2. Dependence constraint: $\forall e = (op_i, op_j) \in E,\ \forall k \in N,\ \sigma(op_i, k) + \delta(e) \leq \sigma(op_j, k + \lambda(e))$.

3. Cyclicity constraint: σ must be expressible in the form of a loop, that is, $\exists II \in N,\ \forall op \in O,\ \forall i \in N \text{ and } i > 0,\ \sigma(op, i) = \sigma(op, 1) + II * (i - 1)$.

then we say that σ is a valid loop schedule for the given loop. II is called the initiation interval of σ. The goal of software pipelining is to find a valid loop schedule with the minimum initiation interval.

[3] There are some other names, e.g. the steady state, the repeating pattern, the new loop body.

2.2 The Data Dependence Representation of Loop Nests

The data dependence representation of a single-level loop should be extended for loop nests. For simplicity of formulation, we only consider normalized perfect nested loops[4] [15] as shown in Figure 2.2, but the ideas presented in this paper can be extended to any nested loop.

```
for i1 = 1 to N1 do
 for i2 = 1 to N2 do
  ......
  for im = 1 to Nm do
   begin
    s1;
    s2;
    ......
    sn;
   end
```

Figure 2.2 The perfect nested loop

Each iteration in a nested loop of depth m is identified by its index vector $(i_1, i_2, ..., i_m)$, where i_k is the value of the kth loop index in the nested loop, numbered successively from the outermost loop to the innermost loop. In a sequential loop, the iterations are thus executed in *lexicographic order* of their index vectors [15].

Therefore, an instance of an operation op in a nested loop is represented as $(op, (i_1, i_2, ..., i_m))$. For any two instances, $(op, (i_1, i_2, ..., i_m))$ and $(op', (i'_1, i'_2, ..., i'_m))$, if there is a data dependence between them, then we say there is a data dependence between op and op' with a *distance vector* [15] of $(d_1, d_2, ..., d_m)$, where $d_k = i'_k - i_k$, $\forall k = 1, 2, ..., m$. Like the data dependences in a single-level loop, there is also a delay δ associated with each dependence in a nested loop.

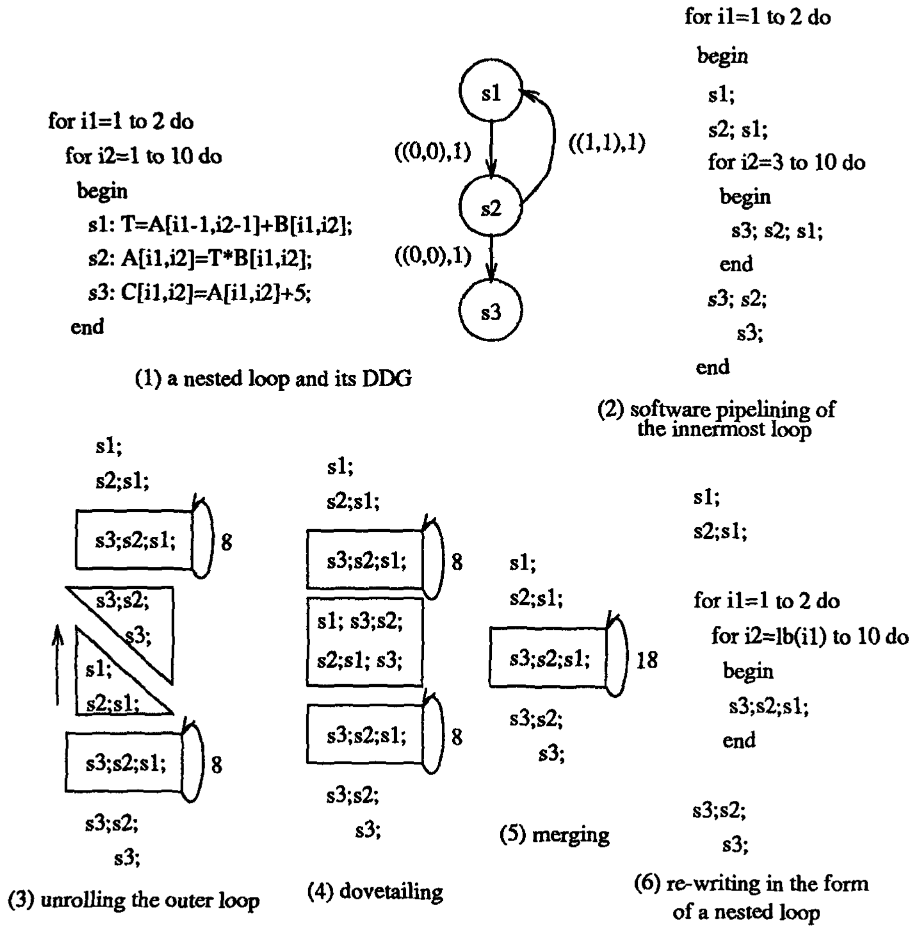
However, for some nested loops, their data dependences may not be representable with a finite set of distance vectors. In this case, extensions to include *direction vector* [15] are necessary.

The direction vector $(\theta_1, \theta_2, ..., \theta_m)$ of two iterations $(i_1, i_2, ..., i_m)$ and $(i'_1, i'_2, ..., i'_m)$ is defined by, for all j $(1 \le j \le m)$

$$\theta_j = \begin{cases} '<' \text{ if } i'_j - i_j > 0 \\ '=' \text{ if } i'_j - i_j = 0 \\ '>' \text{ if } i'_j - i_j < 0 \end{cases}$$

[4] In this paper, "nested loop" or "loop nest" refers to the normalized perfect nested loop unless it is specified.

A direction vector may contain the symbol '$*$' which stands for an arbitrary relationship between corresponding components of two iterations. $(=,<,*)$, for example, stands for the set $\{(=,<,<),(=,<,>),(=,<,=)\}$.



```
for i1=1 to 2 do
  for i2=1 to 10 do
    begin
      s1: T=A[i1-1,i2-1]+B[i1,i2];
      s2: A[i1,i2]=T*B[i1,i2];
      s3: C[i1,i2]=A[i1,i2]+5;
    end
```

(1) a nested loop and its DDG

```
for i1=1 to 2 do
  begin
    s1;
    s2; s1;
    for i2=3 to 10 do
      begin
        s3; s2; s1;
      end
    s3; s2;
      s3;
  end
```

(2) software pipelining of the innermost loop

(3) unrolling the outer loop

(4) dovetailing

(5) merging

```
s1;
s2;s1;

for i1=1 to 2 do
  for i2=lb(i1) to 10 do
    begin
      s3;s2;s1;
    end

s3;s2;
  s3;
```

(6) re-writing in the form of a nested loop

Figure 3.1 The Principle of Pipelining-Dovetailing

3 Pipelining-Dovetailing: Motivating Examples

3.1 The Principle of Pipelining-Dovetailing

Let us first describe the principle of pipelining-dovetailing with a simple nested loop shown in Figure 3.1(1), where the value on each dependence edge denotes the distance vector and the delay. The distance vector (0,0) represents a loop-independent dependence. Assume we software pipeline the innermost loop as shown in Figure 3.1(2). Imaging that we fully unroll the outer loop as shown in Figure 3.1(3). Now, we present a transformation, called *dovetailing*, to transform Figure 3.1(3) to (4). In Figure 3.1(3), the prelude of the second software pipelined loop can be moved upward to fit together with the postlude of the first software pipelined loop, thus generating Figure 3.1(4). It is easy to check that all data dependences are satisfied in Figure 3.1(4) so the dovetailing is valid for this example. Guaranteeing a valid dovetailing will be theoretically detailed in the next section. After dovetailing, the loop in Figure 3.1(4) can be *merged* as shown in Figure 3.1(5), which is always valid since we do not change the execution order of the loop iterations. We re-write the merged loop in the form of a nested loop as shown in Figure 3.1(6), where $lb(i1)$ means that $i2$ should count from 3 if $i1 = 1$, otherwise from 1. We call *"pipelining-dovetailing"* the transformation from Figure 3.1(2) to Figure 3.1(6).

3.2 Two Full Examples of Pipelining-Dovetailing

Next we discuss two examples to illustrate the applications of pipelining-dovetailing. The first example illustrates how pipelining-dovetailing improves the efficiency of software pipelining of the innermost loop. We can also see the simplicity, the low computation complexity and the low implementation cost of pipelining-dovetailing in the first example. The second example can not be pipelining-dovetailed due to data dependence constraint.

The first example is shown in Figure 3.2(a). After the innermost loop is software pipelined, we get the nested loop shown in Figure 3.2(b). Although the instruction-level parallelism within the innermost loop is fully exploited, its prelude and postlude are executed $N1 * N2$ times, thus the whole nested loop can not be efficiently executed on a VLIW/superscalar processor. If we apply pipelining-dovetailing to it, however, then the final nested loop is shown in Figure 3.2(c) where the prelude and the postlude are only executed once. The lower bound of $i3$ should be rewritten so that it counts from 5 instead of 1 when $i1 = 1$ and $i2 = 1$. We can see that the whole nested loop is in the form of software pipelining in Figure 3.2(c). It is not difficult to check that all dependences are satisfied in the loop of Figure 3.2(c) so pipelining-dovetailing is valid for this example.

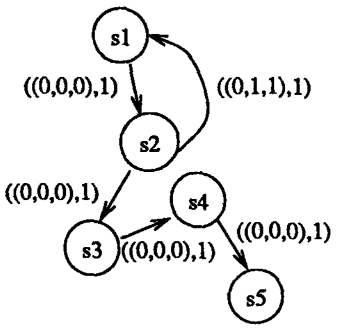
Now we do a simple quantitative analysis to see how pipelining-dovetailing improves the efficiency of software pipelining. Let $N1 = N2 = N3 = 10$, and the execution time of each operation is one cycle, then the original nested loop in Figure 3.2(a) needs $T_{ori} = (1 + 1 + 1 + 1 + 1) * 10 * 10 * 10 = 5000$ cycles; the loop in Figure 3.2(b) needs $T_{inn} = (4 + 4) * 10 * 10 + 1 * 10 * 10 * 6 = 1400$ cycles; the loop in Figure 3.2(c) needs $T_{pd} = 4 + 4 + 1 * (10 * 10 * 10 - 4) = 1004$ cycles. Therefore, software pipelining of the innermost loop gets the speedup of $T_{ori}/T_{inn} = 3.57$ over the original loop; but software pipelining plus pipelining-dovetailing can get the speedup of $T_{ori}/T_{pd} = 4.98$ over the original loop, and an improvement of 28.29% over software pipelining the innermost loop only.

We give the second example in Figure 3.3. The nested loop in Figure 3.3(a) can not be pipelining-dovetailed due to the dependence edge from s_2 to s_1.

Readers can easily check that this edge will be violated if we software pipeline the innermost loop and then do pipelining-dovetailing directly.

The next section will develop a condition under which pipelining-dovetailing is guaranteed to be valid.

```
for i1=1 to N1 do
 for i2=1 to N2 do
  for i3=1 to N3 do
   begin
    s1: T1=A[i1,i2-1,i3-1]+C1;
    s2: A[i1,i2,i3]=T1+C2;
    s3: T2=A[i1,i2,i3]*C3;
    s4: T3=T2*C4;
    s5: B[i1,i2,i3]=T3+C5;
  end
```



(a) a nested loop and its DDG

```
for i1=1 to N1 do
 for i2=1 to N2 do
  begin
   s1;
   s2; s1;
   s3; s2; s1
   s4; s3; s2; s1;
   for i3=5 to N3 do
    begin
     s5; s4; s3; s2; s1;
   end
    s5; s4; s3; s2;
        s5; s4; s3;
            s5; s4;
                s5;
  end
```

(b) software pipelining of the innermost loop

```
s1;
s2; s1;
s3; s2; s1;
s4; s3; s2; s1;
for i1=1 to N1 do
  for i2=1 to N2 do
    for i3= L(i1,i2) to N3 do
     begin
      s5; s4; s3; s2; s1;
     end
s5; s4; s3; s2;
    s5; s4; s3;
        s5; s4;
            s5 ;
```

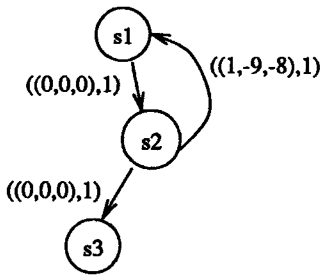
(c) valid pipelining-dovetailing

Figure 3.2 Example 1

3.3 Relation with Unimodular Loop Transformation

It is important to highlight relation between the pipelining-dovetailing transformation we develop in this paper and the unimodular loop transformation [16, 10]

```
for i1=1 to 10 do
  for i2=1 to 10 do
    for i3=1 to 10 do
      begin
        s1: T=A[i1-1,i2+9,i3+8]+C1;
        s2: A[i1,i2,i3]=T*C2;
        s3: B[i1,i2,i3]=A[i1,i2,i3]+C3;
      end
```



(a) a nested loop and its DDG

```
for i1=1 to 10 do
  for i2=1 to 10 do
    begin
      s1;
      s2; s1;
      for i3=3 to 10 do
        begin
          s3; s2; s1;
        end
      s3; s2;
         s3;
    end
```

(b) software pipelining of the innermost loop

```
s1;
s2; s1;
for i1=1 to 10 do
  for i1=1 to 10 do
    for i3= L(i2,i1) to 10 do
      begin
        s3; s2; s1;
      end
s3; s2;
   s3;
```

(c) invalid pipelining-dovetailing

Figure 3.3 Example 2

which is well known in the field of optimizing and parallelizing compilers. Unimodular loop transformation provides a novel matrix representation to combine loop transformations such as loop interchange, loop reversal and loop skewing. Its goal has been to maximally expose loop-level parallelism, i.e. the transformed loop nests have maximum number of doall loops. Consider the example as shown in Figure 3.1(1), since the innermost loop is already a **doall** loop in effect, unimodular loop transformation may not do anything to further transform it for exposing instruction-level parallelism in the innermost loop. However, our method would perform dovetailing to extend the software pipelining to the outer-loop.

In fact, we can view pipelining-dovetailing and unimodular loop transformation to be complement with each other — each can be applied for their own purpose at a different phase in a compiler. For example, one can imagine that some unimodular transformation or tiling may be performed at an earlier phase of compilation for parallelization or storage optimization, then pipelining-dovetailing can be applied at the later code generation phase.

4 The Condition for Valid Pipelining-Dovetailing

As discussed in the last section, pipelining-dovetailing is not always valid due to data dependence constraint. In this section, a sufficient condition will be presented under which pipelining-dovetailing is valid.

We first illustrate three concepts – innermost DDG, linearized DDG and pipelining-depth – which will be used as a basis in the following discussion.

The *innermost DDG* is the DDG of the innermost loop, which retains only those dependences which have zeros on all outer dimensions. Figure 4.1(a) is an example of the innermost DDG of the nested loop shown in Figure 3.2(a). Innermost DDG is traditionally used when software pipelining is applied to the innermost level only.

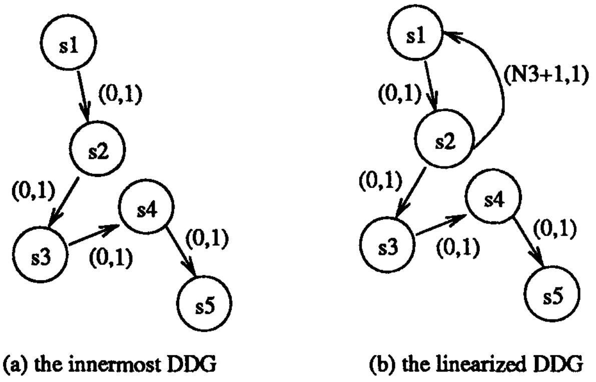


Figure 4.1 The innermost DDG and the linearized DDG of the nested loop in Figure 3.2(a)

The *linearized DDG* is a new concept we present in this paper, which has the same node set and the same edge set as the DDG of a nested loop, but each dependence edge's distance vector is *linearized* into a scalar. For example, the dependence distance (0,1,1) of the edge from s2 to s1 in Figure 3.2(a) is linearized into $0 * N2 * N3 + 1 * N3 + 1 = N3 + 1$, which is the dependence distance of the edge from s2 to s1 in Figure 4.1(b). The linearized DDG can be exactly defined as:

Given a nested loop and its DDG, $G = (O, E, (d_1, d_2, ..., d_m), \delta)$ where O is the node set, E is the edge set, $(d_1, d_2, ..., d_m)$ is the distance vector and δ the delay on each edge, the linearized DDG of G, $G' = (O', E', \lambda', \delta')$, is defined by:

(1) $O' = O, E' = E$;

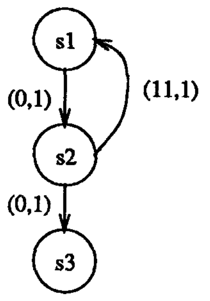
(2) For each edge e,

$\lambda'(e) = N_2 * N_3 * ... * N_m * d_1(e) + N_3 * N_4 * ... * N_m * d_2(e) + ... + N_m * d_{m-1}(e) + d_m(e)$;

(3) For each edge e, $\delta'(e) = \delta(e)$.

In general, given a nested loop, its linearized DDG and its innermost DDG have the same node set. Moreover, the edge set of the innermost DDG is a subset of the edge set of the linearized DDG. While the innermost DDG is the data dependence constraint when software pipelining is only applied to the innermost loop, the linearized DDG is the data dependence constraint when software pipelining is applied to the ***linearized*** loop of the nested loop. By "linearizing a loop nest", we mean that the loop nest is first completely unrolled and then re-rolled into a single level loop. Figure 4.2(1) is an example of a "linearized loop". As we can see, the two dimensional iteration space (i1,i2) in Figure 3.1(1) is linearized into a one-dimensional iteration space ((i1,i2)$\rightarrow$ ($i1 * 10 + i2$)) in Figure 4.2(1).

```
for k=1 to 20 do
 begin
  s1: T=A[i1-1,i2-1]+B[i1,i2];
  s2: A[i1,i2]=T*B[i1,i2];
  s3: C[i1,i2]=A[i1,i2]+5;
 end
 where i1= the floor of k/10
       i2= (k mod 10)+1
```



(1) the linearized loop and the linearized DDG of the nested loop in Figure 3.1(1)

```
s1;
s2; s1;
for k=3 to 20 do
  begin
    s3; s2; s1;
  end
s3; s2;
    s3;
```

(2) software pipelining the linearized loop

Figure 4.2 Linearizing a loop nest and software pipelining

The ***pipelining-depth*** is defined as the number of different iterations which are overlapped in the software pipelined loop body. For example, in Figure 3.2(b), the pipelining-depth is 5 since there are 5 different iterations being overlapped in the software pipelined loop body; while in Figure 3.3(b), the pipelining-depth is 3. Pipelining-depth is of an important property: If the dependence distance of a loop-carried dependence is greater than or equal to the pipelining-depth, then this loop-carried dependence can be omitted during software pipelining. This property is very useful when we software pipeline a loop nest: After the loop nest is linearized, some dependence edges may have long dependence distances and removal of these edges may simplify the data dependence graph. For example, in Figure 4.2, the dependence distance of the edge from s2 to s1 is 11, which is much greater than the pipelining-depth, 3.

Now we derive a sufficient condition on the basis of the above three concepts under which pipelining-dovetailing is valid. Let us recall the principle of pipelining-dovetailing illustrated in Figure 3.1. That is, we first software pipeline the innermost loop and then do pipelining-dovetailing. This can be regarded as to first linearize the loop nest into a single level loop and then software pipeline the linearized loop, as shown in Figure 4.2.

We now see that if the linearized DDG and the innermost DDG are the same, pipelining-dovetailing is definitely valid. If the linearized DDG has more dependence edges than the innermost DDG and those edges have large dependence distances (greater than or equal to the pipelining-depth), according to the property of pipelining-depth, pipelining-dovetailing is still valid. Let us take the example of Figure 4.2. Although the linearized DDG has an edge from s2 to s1 which is not in the innermost DDG, the dependence distance of this edge is 11, greater than the pipelining-depth, 3, thus pipelining-dovetailing is valid for this loop nest. Therefore, we have the following theorem.

Theorem 4.1 Given a nested loop, let its innermost DDG be (O, E, λ, δ), the linearized DDG be $(O, E', \lambda', \delta')$, assume *pd* is the pipelining-depth. If $E = E' - \{e | e \in E' \text{ and } \lambda'(e) \geq pd\}$, Then the nested loop can be validly pipelining-dovetailed.

Readers can find the proof in the Appendix.

Although Theorem 4.1 can be used to determine if pipelining-dovetailing is valid, it needs to construct innermost DDG and linearized DDG. Our question is: can we directly check each dependence distance vector of a given nested loop to determine if pipelining-dovetailing is valid? Theorem 4.1 itself provides a hint that we can give the question a positive answer.

The sufficient condition stated in Theorem 4.1 actually requires that the (linearized) DDG of the nested loop does not include the edges which are not in the innermost DDG and whose (linearized) dependence distances are less than the pipelining-depth. We have found that the distance vectors of those edges can be expressed in the form of $(0, ..., 0, 1, 1 - N_{i+1}, ..., 1 - N_{m-1}, d_m)$ where $1 \leq i < m$[5], d_m is a negative integer of less than $(pd - N_m)$, N_j $(1 \leq j \leq m)$ is the number of iterations of the level-j loop and *pd* is the pipelining-depth. We call those edges ***dovetailing-preventing edges*** since they prevent the valid pipelining-dovetailing. Figure 3.3 gives an example of dovetailing-preventing edges, where the edge from s2 to s1 has the distance vector $(1, -9, -8)$, in the form of $(1, 1 - 10, -8)$ where $d_m = -8$ is less than $pd - N_m = 3 - 10 = -7$. As discussed in section 3.2, This edge prevents the valid pipelining-dovetailing. Thus, we have Theorem 4.2 (readers can find the proof in the Appendix).

Theorem 4.2 Given a nested loop, if there is not any dovetailing-preventing edge in its DDG, then the nested loop can be validly pipelining-dovetailed.

Theorem 4.2 gives a very feasible method for applying pipelining-dovetailing.

Finally, we want to point out that, the above results can be easily extended to the nested loops whose data dependences may include direction vectors. Here we only use a simple example to illustrate the extension.

Given the nested loop, for simplicity, let the number of levels, $m = 2$. A direction vector, say $(=, <)$, is a set of distance vectors, $\{(0, 1), (0, 2), ..., (0, N_2 - 1)\}$. When we construct the DDG of the innermost loop and the linearized DDG, we only take the worst case, that is $(0, 1)$. When we consider the data dependences in Theorem 4.2, we still only take the worst case, $(0, 1)$.

The direction vector, $(=, >)$, is also a set of distance vectors, $\{(0, -1), (0, -2), ..., (0, -N_2 + 1)\}$. When we construct the DDG of the innermost loop and the

[5] If $i = m$, the edge is in the innermost DDG.

linearized DDG, we only take the worst case, that is $(0, -N_2 + 1)$. When we consider the data dependences in Theorem 4.2, we still only take the worst case, $(0, -N_2 + 1)$.

5 Applicability and Performance Improvement

The applicability of pipelining-dovetailing depends on how many loop nests in practical programs satisfy the condition of Theorem 4.2. According to the form of the distance vector of a dovetailing-preventing edge, the condition is quite relaxed and we expect that most loop nests may not include any dovetailing-preventing edge and satisfy the condition.

In order to verify the applicability of our method we studied three well known benchmarks from SPECfp92 – fppp.f, tomcatv.f and ora.f. We have identified all loop nests which are suitable for software pipelining. For each such loop nest we examined the dependence relation of the loop body and checked it against the condition for dovetailing outlined in the last section. We found that all such loop nests in these three programs satisfy the condition so that the dovetailing can be successfully applied [6].

It is beyond the scope of this paper to fully assess the performance impact of pipelining-dovetailing. However, we found it useful to provide some preliminary analysis based on a straight-forward back-to-the-envelop calculation as detailed below.

Given an m-level nested loop, let the execution time of the original loop body be T, $N^0 = N_1 * N_2 * ... * N_{m-1}$ where N_j is the number of iterations of level-j loop, then the execution time of the original nested loop is $t_0 = N^0 * N_m * T$.

Assume we only software pipeline the innermost loop, let II be the initiation interval, then the execution time of the software pipelined loop is

$$t_{sp} = N^0 * 2II * (\lceil \frac{T}{II} \rceil - 1) + N^0 * (N_m - \lceil \frac{T}{II} \rceil + 1) * II = N^0 * (N_m + \lceil \frac{T}{II} \rceil - 1) * II.$$

We pipelining-dovetail the software pipelined loop. It is not difficult to compute the execution time of the loop which is software pipelined and pipelining-dovetailed

$$t_{sp-pd} = 2II * (\lceil \frac{T}{II} \rceil - 1) + N^0 * N_m - \lceil \frac{T}{II} \rceil + 1) * II$$

Provided $N^0 * N_m$ is large enough compared to $\lceil \frac{T}{II} \rceil$, we get the following result approximately $t_{sp-pd} = N^0 * N_m * II$.

We can compute the performance improvement below

$$\Delta = \frac{t_0}{t_{sp-pd}} - \frac{t_0}{t_{sp}} = \frac{T}{II} * \frac{1}{\eta + 1} \qquad .... (6.1)$$

[6] We also develop a loop transformation to transform those loop nests which include dovetailing-preventing edges such that they satisfy the condition. However, due to the limitation of the paper's length, we can not present this work in this paper.

Where $\eta = \frac{N_m}{\lceil \frac{T}{II} \rceil - 1}$.

From equation (6.1), we directly have the following conclusions:

1. The efficiency of pipelining-dovetailing only depends on $\frac{T}{II}$ and N_m;
2. When $\frac{T}{II}$ is large and N_m is small, the efficiency of pipelining-dovetailing is significant;
3. When $\frac{T}{II}$ is small and N_m is large, it is not necessary to do pipelining-dovetailing;
4. After software pipelining the innermost loop, we can get $\frac{T}{II}$, thus make a decision whether or not it is worth to do pipelining-dovetailing;
5. Note that, in order to transform the innermost loop into a doall loop, the leading loop transformation techniques [10] tend to cause a large $\frac{T}{II}$ and a small N_m of the innermost loop, which indicating that pipelining-dovetailing is very promising.

6 Related Work

Most of existing software pipelining methods have been focused on the innermost loops. In [17] and [4], an approach has been presented to deal with nested loops (in [4], it is called hierarchical reduction); that is, first software pipeline the innermost loop, then reduce the software pipelined loop as a single node in the body of outer loop so that the outer loop can be software pipelined further.

Another software pipelining method for nested loops has been presented in [18], where the loop schedule σ was defined as

$$\sigma(s_j, (i_1, i_2, ..., i_m)) = \lfloor \frac{T_1 * i_1 + T_2 * i_2 + ... + T_m * i_m + A(s_j)}{r} \rfloor$$

$T_1, T_2, ..., T_m, A(s_j), \forall j = 1, 2, ..., n$, are non-negative integers, r is a positive integer, called periodicity. For software pipelining without unrolling, $r = 1$ and the above formula can be simplified below

$$\sigma(s_j, (i_1, i_2, ..., i_m)) = T_1 * i_1 + T_2 * i_2 + ... + T_m * i_m + A(s_j)$$

In [18], a method has been presented to determine the optimal $T_1, T_2, ..., T_m$ and $A(s_j)$ while, in [19], a method has been presented to generate the new loop based on $T_1, T_2, ..., T_m$ and $A(s_j)$. The computational complexity of this method appears to be quite high. In addition, it remains to be a challenge to handle the resource constraint under this method.

Many loop transformation techniques have been proposed to exploit coarse-grain parallelism for multi-processor architectures and vector machines [15, 12, 20, 21, 11]. Unimodular loop transformation provides a framework to combine the loop transformations such as loop interchange, loop reversal and loop skewing to exploit the maximum degree of parallelism [16, 10]. In [13], the affine transformation technique has been presented to extend the work of [10]. We have already highlighted the differences of these work from ours in section 1 and 3.3.

7 Conclusion

In this paper, we have proposed pipelining-dovetailing as a simple method to extend software pipelining from the innermost loop to the enclosing loops in a loop nest. We have also formulated the condition under which pipelining-dovetailing is legal.

We anticipate that software pipelining will become increasingly important for future generation processor architectures with ample instruction-level parallelism. The method developed in this paper has the advantage that it can be built upon on the existing software pipelining method, and appears to be simple to implement. Nevertheless, much work remains to be done to assess its feasibility in practical compilers.

Acknowledgments

We would like to thank Dr. Christine Eisenbeis, Dr. Andreas Krall and Prof. Bogong Su for their comments, and thank Mr. Hisham Petry for proof-reading. We would also like to appreciate the helpful suggestions from Prof. Uwe Kastens and our anonymous referees for improving the presentation of this paper.

References

1. B. R. Rau and J.A. Fisher. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing*, 7(1), January 1993.
2. 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.
3. K. Ebcioglu and T. Nakatani. A new compilation technique for paralelizing loops with unpredictable branches on a vliw architecture. In A. Nicolau D. Gelernter and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 213–229. Pitman/The MIT Press, London, 1989.
4. M.S. Lam. *A Systolic Array Optimizing Compiler.* PhD thesis, CMU, 1987. CMU-CS-87-187.
5. 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.
6. A. Aiken and A. Nicolau. A realistic resource-constrainted software pipelining algorithm. In T.Gross A. Nicolau, D. Gelernter and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 274–290. Pitman/The MIT Press, London, 1991.
7. R. Huff. Lifetime-sensitive modulo scheduling. In *proceedings of ACM SIGPLAN PLDI*, pages 258–267, June 1993.
8. Q. Ning and G.R. Gao. A novel framework of register allocation for software pipelining. In *proceedings of POPL*, January 1993.

9. 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.
10. Michael E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), 1991.
11. U. Banerjee. *Loop Transformations for Restructuring Compilers.* Kluwer Academic, 1993.
12. A. Darte, L. Risset, and Y. Robert. Loop nest scheduling and transformations. In *proceedings of Environments and Tools for Parallel Scientific Computing*, 1992.
13. Amy W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *proceedings of LCPC'94*, 1994.
14. F. Gasperoni. Compilation techniques for vliw architectures. Technical Report TR435, New York University, March 1989.
15. Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.
16. U. Banerjee. Unimodular transformations of double loops. In *proceedings of the 3rd Workshop on Languages and Compilers for Parallel Computing*, 1990.
17. Bogong Su, Shiyuan Ding, Jian Wang, and Jinshi Xia. GURPR–a method for global software pipelining. In *proceedings of the 20th Annual International Workshop on Microprogramming (MICRO-20)*, pages 88–96. ACM and IEEE, November 1987.
18. Guang R. Gao, Qi Ning, and Vincent Van Dongen. Extending software pipelining techniques for scheduling nested loops. In *proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, 1993.
19. Ki chang Kim and Alexandru Nicolau. Parallelizing tightly nested loops. In *proceedings of International Conference on Parallel Processing*, 1991.
20. P. Feautrier. A collection of papers on the systematic construction of parallel and distributed programs. Technical Report Hors-serie, Lab. MASI, Universite P. et M. Curie, 1992.
21. M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, MA, 1989.

Appendix: The Proofs of Theorem 4.1 and 4.2

Theorem 4.1 Given a nested loop, let its innermost DDG be (O, E, λ, δ), the linearized DDG be $(O, E', \lambda', \delta')$, assume *pd* is the pipelining-depth. If $E = E' - \{e | e \in E' \text{ and } \lambda'(e) \geq pd\}$, Then the nested loop can be validly pipelining-dovetailed.

Proof: We first software pipeline the innermost loop and then do pipelining-dovetailing for the whole nested loop, which is equivalent to that we software pipeline the whole nested loop in its lexicographic order under the data dependence constraints given in the linearized DDG. Therefore, we only need to show that, given the condition of the theorem, the linearized DDG is satisfied if the DDG of the innermost loop is satisfied. From the property of pipelining-depth, we can remove those loop-carried dependences whose dependence distances are greater than or equal to *pd* since they will be automatically satisfied. Furthermore, from the condition of the theorem, after those loop-carried dependences