

# Using Retiming to Minimize Inter-Iteration Dependencies\*

Timothy W. O’Neil  
Computer Science Dept.  
University of Akron  
Akron, OH 44325-4002

Edwin H.-M. Sha  
Computer Science Dept.  
Univ. of Texas at Dallas  
Richardson, TX 75083-0688

## Abstract

Since data dependencies greatly decrease instruction level parallelism, minimizing dependencies becomes a crucial part of the process of parallelizing sequential code. Eliminating all unnecessary hazards leads to the more efficient use of resources, fewer processor stalls and easily maintainable code. In [8] we proposed a novel approach for eliminating redundant data dependencies from code. In this paper, we review this method and show how this elimination technique may be combined with retiming so as to parallelize code even further.

**Key words:** Data dependence analysis, redundant dependence analysis, compiler optimization, retiming.

## 1 Introduction

The age of parallel computing brought with it the need for compilers that examine sequential code and optimize it to execute on parallel machines. Since loops are typically the most expensive part of a program in terms of execution time, an optimizing compiler must explore the parallelism hidden in loops. They must be able to identify those loops whose iterations can run simultaneously and schedule them to execute in parallel. This requires the use of sophisticated tests for detecting data dependencies in programs during compilation, and possibly advance planning and analysis prior to compilation.

A variety of methods exists for discovering data dependencies in programs. Once uncovered, the compiler must enforce such restrictions by explicit synchronizations within the optimized code, thus ensuring that the order of memory accesses remains satisfied. However,

such a synchronization would be unnecessary if the dependence relation it enforces were satisfied by other dependencies. Therefore, discovering and eliminating redundant data dependencies becomes an important priority in our compiler. In [8], we proposed one such approach for finding redundant dependencies.

In addition, a great deal of research has been done concerning the examination and manipulation of code in order to enhance parallelism. One of the most useful ideas has been to model a program as a weighted graph, then transform this graph into an optimized equivalent. This altered model then corresponds to a new version of the original program which performs the same task but does so more efficiently. Typically the focus has been on minimizing execution time, with it implied that dependencies among statements were being maneuvered in a beneficial way. There has been little said about the explicit effect of graph transformation on the data dependencies of the depicted program. In this paper, we will consider one of the most popular graph transformation techniques (retiming) and outline how it can be used in conjunction with previous ideas to reduce required data hazards.

Much work has been completed regarding the optimization of code through transformation techniques. Wolf and Lam [12] considered restructuring techniques like loop interchange, reversal, skewing and tiling [11] which could be performed on the original code without translation to and from a graphical model. Graph transformation techniques like retiming [6] and unfolding [10] have been considered extensively [9] for extracting the underlying parallelism from code, resulting in an optimal parallel schedule [2]. However, none of this work considered the explicit effect of graph transformation on the data dependencies within code. To this point, the idea of modeling a loop nest in graphical form, manipulating the graph via retiming, and producing an enhanced version of the code specifically to facilitate the elimination of redundant hazards has not been explored.

In this paper we review our original method from [8]

---

\*This work was partially supported by NSF grants MIP95-01006 and MIP97-04276; and by the A.J. Schmitt Foundation while the authors were with the University of Notre Dame. It was also supported by the University of Akron; and by the TI University Program, NSF ETA 0103709, Texas ARP 009741-0028-2001.

for expressing and studying loop-carried dependencies. We demonstrate that not all dependencies in a program need considered and present an approach for eliminating those that are unnecessary. Finally, we combine our efforts with the established method of re-timing so as to parallelize code even further.

```
DO i=1,N
  x   =a[i]+b[i] /* 1 */
  a[i+1]=x      /* 2 */
  b[i]  =b[i]+1  /* 3 */
  a[i+3]=a[i+1]-1 /* 4 */
  a[i+2]=a[i+1]+1 /* 5 */
ENDDO
```

Figure 1: An example.

## 2 Eliminating Redundant Data Dependencies

We wish to begin by reviewing not only the concept of data dependence and the terminology used in its study, but also our reduction techniques. All of this material has appeared elsewhere [3, 8, 11].

The key to maximizing parallelism in a program is to study the data dependencies in the program. We shall say that a *data dependence* or *read-after-write (RAW) hazard* occurs when one instruction requires the output of a previous instruction in order to execute. In other words, let  $I_k$  be the operation executed at time  $k$ . Then  $I_j$  is data dependent on  $I_i$  if either  $I_i$  produces a result used by  $I_j$ , or  $I_j$  is data dependent on  $I_k$  (according to the above definition) and  $I_k$  is data dependent on  $I_i$ . The only way to solve this dependence is to require that  $I_i$  complete its execution before  $I_j$  enters the “operand read” stage of the pipeline, which means reducing pipeline throughput and parallelism.

When a dependence exists between instructions in the same iteration of a loop, the dependence is *intra-iteration*; otherwise it is a *loop-carried* dependence. If a loop-carried dependence exists between instruction  $I_i$  in iteration  $x$  of a loop and  $I_j$  in a later iteration  $y$  of the same loop, the *distance* of the dependence is  $y - x$ . (Trivially, intra-iteration dependencies have distance 0.) We will refer to a dependence  $A$  from  $I_i$  to  $I_j$  having distance  $d$  using the notation  $A : (I_i \rightarrow I_j, d)$ .

For example consider our code fragment from Figure 1 below. Referring to each instruction by the number contained in the comments to the right of the code, we derive four intra-iteration RAW hazards quickly:  $(1 \rightarrow 2, 0)$ ,  $(2 \rightarrow 4, 0)$ ,  $(2 \rightarrow 5, 0)$ , and  $(3 \rightarrow 3, 0)$ . Similarly, to find the loop-carried dependencies for the same code fragment, we *unroll* the loop by replicating the loop body multiple times, taking care to adjust the indices when necessary. By doing this, we find 7 more RAW hazards:  $(2 \rightarrow 1, 1)$ ,  $(5 \rightarrow 4, 1)$ ,  $(5 \rightarrow 5, 1)$ ,  $(5 \rightarrow 1, 2)$ ,  $(4 \rightarrow 4, 2)$ ,  $(4 \rightarrow 5, 2)$  and  $(4 \rightarrow 1, 3)$ . However, we will demonstrate that many of these are redundant and need not be considered when we schedule the code fragment to execute in parallel.

In our example of Figure 1, we saw that we had an intra-iteration dependence between lines 2 and 4, and another between lines 2 and 5. However, if I assume that this code is being executed serially, then I am also assuming an implicit dependence from line 4 to line 5. My dependence  $(2 \rightarrow 5, 0)$  would be unnecessary in this scenario; it is the combination of my other dependencies  $(2 \rightarrow 4, 0)$  and  $(4 \rightarrow 5, 0)$ . Thus, when I parallelize this code, I can ignore  $(2 \rightarrow 5, 0)$  and focus on satisfying the other dependencies which cannot be dismissed in this fashion. I am reducing the number of constraints that bind me and, in the process, making the parallelization of my code much easier.

Given an instruction  $I$  in my program, let  $\tau(I)$  represent the time at which  $I$  executes. As discussed earlier, if I have a data dependence from  $I_i$  to  $I_j$ , it can only be resolved if  $I_i$  completes execution before  $I_j$ ; in other words, if  $\tau(I_i) < \tau(I_j)$ , or  $\tau(I_j) - \tau(I_i) > 0$ . When this is true, we will say that the dependence  $(I_i \rightarrow I_j, d)$  is *satisfied*. Throughout this paper we will assume that instruction  $I_i$  is executed at time  $i$ , simplifying our notation so that  $(I_i \rightarrow I_j, d)$  is satisfied if and only if  $j - i > 0$ .

As pointed out above, if  $(2 \rightarrow 4, 0)$  is satisfied in our program, then  $(2 \rightarrow 5, 0)$  is automatically also satisfied. Whenever the satisfaction of a dependence  $A$  guarantees the satisfaction of another dependence  $B$ , we will say that  $A$  *subsumes*  $B$  and denote it by  $A \supseteq B$ . (In our example we would write  $(2 \rightarrow 4, 0) \supseteq (2 \rightarrow 5, 0)$ .) If  $A \supseteq B$  and  $A \subseteq B$ , we will say that  $A$  *equals*  $B$ , which we will of course denote by  $A = B$ .

With this terminology in mind, it is very easy to show the *translation property* of data dependencies, as in [8]:

**Corollary 2.1** *Let  $d$  be an iteration distance, and let  $A : (I_i \rightarrow I_j, d)$  and  $B : (I_m \rightarrow I_n, d)$  be two data dependencies. Then  $A = B$  if  $i - j = m - n$ .*

Because of the translation property, the dependence relation  $(I_i \rightarrow I_j, d)$  is the same as  $(I_{i-j} \rightarrow I_0, d)$ , so we need only keep track of the difference in start times between the two instructions of a data dependence. This difference, which we will designate  $\lambda$ , is called the *characteristic value* of the data dependence. Because

of this, we will henceforth refer to the dependence  $A : (I_i \rightarrow I_j, d)$  as  $A : (\lambda_A, d)$  where  $\lambda_A = i - j$ .

As in [8], we can now show that, given an iteration distance  $d$  and data dependencies  $A : (\lambda_A, d)$  and  $B : (\lambda_B, d)$ ,  $A \supseteq B$  if and only if  $\lambda_A \geq \lambda_B$ . With this in mind, consider now all data dependencies having a given iteration distance  $d$ . Because of this idea, the dependence from this set having the largest characteristic value will subsume every other dependence in the set. This “maximal” dependence is called the *characteristic data dependence for distance  $d$*  and is denoted by  $(\Lambda_d, d)$  where  $\Lambda_d = \max\{\lambda : (\lambda, d) \text{ is a data dep.}\}$ . Returning to our example from Figure 1, recall that we had 11 data dependencies. Because of this concept we can immediately cut this list down to the four characteristic dependencies:  $(0, 0)$ ,  $(1, 1)$ ,  $(4, 2)$  and  $(3, 3)$ . Better still, due to our assumption of serial execution, we can ignore intra-iteration dependencies at this point and concentrate on loop-carried dependencies.

So far, we have seen that dependencies which are subsumed by other dependencies can be dismissed from consideration as we attempt to maximize parallelism. Similarly, if a dependence is subsumed *by some combination* of other dependencies, it should also be removed. For example, in Figure 1, the dependence  $(3, 3)$  is subsumed by three properly-placed copies of  $(1, 1)$ . Specifically,  $(4 \rightarrow 1, 3)$  is subsumed by the combination of  $(4 \rightarrow 3, 1)$  plus  $(3 \rightarrow 2, 1)$  plus  $(2 \rightarrow 1, 1)$ . We will now develop this case wherein different dependencies are combined to subsume some other dependence.

The problem lies in decomposing a distance as a sum of other distances and studying all resulting sums of characteristic dependencies. Consider the set of all dependencies for a given distance formed by adding characteristic dependencies having varying distances. We will adopt the language of [7] and informally define the *dominant data dependence for distance  $d$*  to be that dependence from this set which subsumes all other dependencies in this set. As before, it suffices to choose that dependence with the largest characteristic value. Thus, the dominant data dependence for distance  $d$  is the data dependence  $(\Delta_d, d)$  where

$$\Delta_d = \max \left\{ \left( \sum_{j=1}^n \Lambda_{i_j} + 1 \right) - 1 : \sum_{j=1}^n i_j = d \right\}.$$

It is clear that the calculation of all such sums would take far too long if  $d$  gets very large. Fortunately we can greatly decrease our work load via our method of [8], where we calculate  $\Delta_d$  inductively from the values of  $\Delta_1, \Delta_2, \dots, \Delta_{d-1}$ . Finally, it is clear that all

data dependencies having iteration distance  $d$  can be eliminated if and only if  $\Delta_d > \Lambda_d$ .

All of this leads to the formalized elimination method given as Algorithm 1 below. It is divided into two phases. First, for each iteration distance, we find the maximum characteristic value among all the dependencies having the given distance. The dependence which has this maximum is retained while all others are removed from further consideration. Next we explore redundancy across distances via dynamic programming. For each iteration distance  $d$ , we first find the largest sum of dominant dependencies whose distances add to  $d$ . We then compare this number to the value of the characteristic dependence for  $d$ . If the characteristic value is the larger of the two figures, the dependence remains in our set. Otherwise it is eliminated. Applying this polynomial-time algorithm to our dependencies for Figure 1 reduces our initial set to  $\{(1, 1), (4, 2)\}$ , as we have seen.

---

#### Algorithm 1 Eliminating Redundant Data Dependencies

---

**Input:** A set  $S$  of data dependencies for a given program, the iteration distance  $D$

**Output:** The set  $S$  with all redundant dependencies removed  
/\* Eliminate redundancy for each iteration distance \*/

```

for  $i = 1$  to  $D$  do
   $\Lambda[i] \leftarrow -\infty$ 
  for all data dependencies  $(\lambda, i)$  do
    if  $\lambda > \Lambda[i]$  then
      if  $\Lambda[i] > -\infty$  then
        delete the data dependence  $(\Lambda[i], i)$  from  $S$ 
      end if
       $\Lambda[i] \leftarrow \lambda$ 
    else
      delete the data dependence  $(\lambda, i)$  from  $S$ 
    end if
  end for
end for
/* Eliminate redundancy for multiple distances */
 $\Delta[1] \leftarrow \Lambda[1]$ 
for  $i = 2$  to  $D$  do
   $M \leftarrow -\infty$ 
  for  $j = 1$  to  $\lfloor \frac{i}{2} \rfloor$  do
    if  $M < \Delta[j] + \Delta[i - j] + 1$  then
       $M \leftarrow \Delta[j] + \Delta[i - j] + 1$ 
    end if
  end for
  if  $\Lambda[i] \leq M$  then
     $\Delta[i] \leftarrow M$ 
    delete the data dependence  $(\Lambda[i], i)$  from  $S$ 
  else
     $\Delta[i] \leftarrow \Lambda[i]$ 
  end if
end for

```

---

### 3 The Effect of Retiming

*Retiming* [6] is a transformation technique which rebuilds iterations of a loop nest by redistributing instructions. Each instruction  $I_i$  is retimed by some integral amount  $r(I_i) \geq 0$  representing the offset between  $I_i$ 's original iteration and its iteration following retiming. Since retiming moves instructions forward in the execution flow of a program, it alters the distances of any dependencies involving said instructions. Thus we may be able to further reduce the loop-carried dependencies of a program by retiming the code by a properly selected function before applying our previous elimination technique.

As an example, consider the sample loop in Figure 1. Its flow of execution is pictured at the top of Figure 3. If we begin by retiming instruction 1 once, we shift each occurrence of instruction 1 forward by one iteration, as seen by the second line of Figure 3. In doing so, we reorder instructions so that instruction 1 moves from the head of the repeating nest to the tail. This shifting also takes the very first occurrence of instruction 1 out of the repeating loop nest entirely and makes it a code section all its own, to be executed before beginning the repeated iterations. This new section is called the *prologue* of the schedule. Similarly, shifting removes the last occurrence of instruction 1 from the final iteration, leaving us with an incomplete iteration to execute after our repeating loop terminates. This final section is called the *epilogue*. Figure 2(a) illustrates our revised code at this point, with one iteration of the loop nest removed and divided between prologue and epilogue.

```

x      =a[1]+b[1]      /* 1 iter 1 */
a[2]   =x              /* 2 iter 1 */
x      =a[2]+b[2]     /* 1 iter 2 */
DO i=1,N-2
    b[i] =b[i]+1      /* 3 iter i = 1' */
    a[i+3]=a[i+1]-1  /* 4 iter i = 2' */
    a[i+2]=a[i+1]+1  /* 5 iter i = 3' */
    a[i+2]=x          /* 2 iter i+1 = 4' */
    x    =a[i+2]+b[i+2] /* 1 iter i+2 = 5' */
ENDDO
b[N-1]=b[N-1]+1     /* 3 iter N-1 */
a[N+2]=a[N]-1       /* 4 iter N-1 */
a[N+1]=a[N]+1        /* 5 iter N-1 */
a[N+1]=x             /* 2 iter N */
b[N]   =b[N]+1       /* 3 iter N */
a[N+3]=a[N+1]-1     /* 4 iter N */
a[N+2]=a[N+1]+1     /* 5 iter N */

x      =a[1]+b[1]     /* 1 */
DO i=1,N-1
    a[i+1]=x          /* 2 */
    b[i]   =b[i]+1    /* 3 */
    a[i+3]=a[i+1]-1  /* 4 */
    a[i+2]=a[i+1]+1  /* 5 */
    x      =a[i+1]+b[i+1] /* 1 */
ENDDO
a[N+1]=x             /* 2 */
b[N]   =b[N]+1       /* 3 */
a[N+3]=a[N+1]-1     /* 4 */
a[N+2]=a[N+1]+1     /* 5 */

```

(a)

(b)

Figure 2: Fig. 1: (a) after retiming instruction 1 once; (b) after retiming is completed.

We may now proceed in a similar fashion to retime instruction 2, resulting in the pattern seen in the third

line of Figure 3. Each copy of instruction 2 moves forward an iteration, with the first copy moving to the prologue and the last leaving the epilogue. We may also retime instructions within our nest. For example, retime instruction 1 a second time, as in the last line of Figure 3. The effect is to pick each copy of instruction 1 up out of the middle of its old iteration and append to the end of the previous iteration as shown. Retiming instruction 1 this second time also leaves us with two incomplete iterations at the end of our execution, which are merged to form the new epilogue. (In general, as noted in [1], there are  $r(I_i)$  copies of instruction  $I_i$  in the prologue and  $(\max_{I_j} r(I_j)) - r(I_i)$  copies in the epilogue once retiming is complete.) This final flow pattern from Figure 3 gives us the needed information for constructing a retimed version of our initial example, which is seen in Figure 2(b).

To this point, we have discussed the effect of retiming on code. Typically, however, retiming is viewed solely as an operation on directed graphs which represent control flow within code. These graphs, called *data-flow graphs* or DFGs, model a loop nest by assigning a vertex to each instruction of the loop nest and representing dependencies between relations by directed edges between nodes. These edges are weighted by *delays* which indicate what we have dubbed the dependence distance. For example, the DFG in Figure 4(a) models the behavior of the loop nest in Figure 1. Each RAW hazard with non-zero characteristic value found earlier corresponds to a weighted edge in this graph; we have excluded dependencies with value zero from this representation for reasons we will indicate shortly.

Retiming can now be viewed as pulling delays from a node's incoming edges and pushing them onto the node's outgoing edges. (Thus retiming will not affect the delay count of an edge from a node to itself, making the representation of zero-characteristic-value dependencies in a DFG a useless exercise which only clouds the issue.) For example, let  $(i, j)$  represent the directed edge from vertex  $i$  to vertex  $j$ . Retiming node 1 by 1 pulls a delay in from each of the edges  $(4, 1)$ ,  $(5, 1)$  and  $(2, 1)$  and deposits a delay onto edge  $(1, 2)$ . Retiming node 2 by 1 then draws this delay from  $(1, 2)$  in and pushes it onto edges  $(2, 1)$ ,  $(2, 4)$  and  $(2, 5)$ . Thus there are enough delays to retime vertex 1 once more, resulting in the retimed DFG of Figure 4(b). It is well-known that the effect of retiming by a function  $r$  is to alter the delay count of edge  $e : u \rightarrow v$  from  $d(e)$  to  $d(e) + r(u) - r(v)$ . (This retimed delay count is denoted  $d_r(e)$ .) Since an edge cannot be assigned a negative number of delays, we must have  $d_r(e) \geq 0$  for

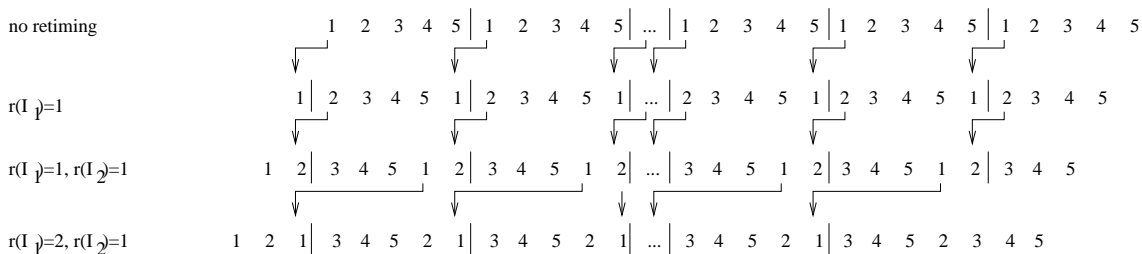


Figure 3: Alteration of execution flow pattern by retiming.

all edges  $e$  in order for a retiming to be *legal*.

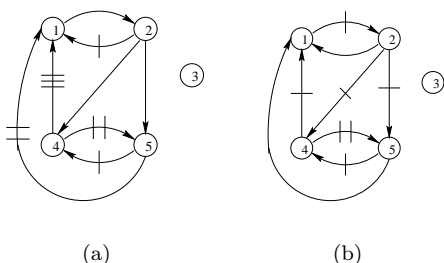


Figure 4: The DFG for Fig. 1: (a) before retiming; (b) after retiming.

While retiming does not change the data dependencies in a program, the sliding of instructions forward will change how a dependence is represented in our notation and thus may aid our elimination strategy. As pointed out above, retiming changes the distance of each dependence ( $I_i \rightarrow I_j, d$ ) from  $d$  to  $d+r(I_i)-r(I_j)$ . However, since retiming also reorders the instructions of the loop nest in a manner not easy to predict, it is not possible at this time to formalize the complete effect that retiming has on a dependence’s characteristic value. Therefore, in order to derive the retimed characteristic values, we must simply translate each dependence from the original to the retimed code. This is done for our initial example in Table 1.

We now apply our elimination method from [8] in order to reduce our set of loop-carried dependencies, first to  $\{(2, 1), (0, 2)\}$ , then finally to the singleton set  $\{(2, 1)\}$ . By reducing the distances (and possibly by altering the characteristic values) of certain dependencies, retiming has redefined them to the point where they may be subsumed by other elements of the set. The cost is that we have introduced many more dependencies into our overall code, namely those involving the prologue and epilogue. However, any such new restrictions must be dealt with only once. The benefit of simplifying the repeated execution of our loop

nest will more than offset this problem if a sufficient number of iterations are executed.

From this exercise, it appears that in general we want to retime our DFG in such a way as to better distribute delays, reducing the delay counts on edges which are too heavily weighted while slightly increasing delay counts on other lighter edges in an attempt to “even out” all delay assignments. In short, we wish to minimize the standard deviation of the retimed edge weights, a very difficult problem. However, let  $\sigma$  be this standard deviation, with  $\mu$  the mean of the retimed weights. We note that  $\sigma \leq \mu\sqrt{|E| - 1}$  since all retimed edge weights must be zero or larger. Thus it appears that a reasonable alternative is to minimize the mean of the edge weights. This is equivalent to minimizing the overall delay count of a retimed DFG by the definition of the mean. As in [6], we can see that

$$\begin{aligned} \sum_{e \in E} d_r(e) &= \left( \sum_{e \in E} d(e) \right) \\ &+ \left( \sum_{u \in V} r(u) \cdot \text{outdegree}(u) \right) \\ &- \left( \sum_{v \in V} r(v) \cdot \text{indegree}(v) \right). \end{aligned}$$

We note that the first of these summations is fixed for any DFG. Since our retiming must still be legal, we must also have  $0 \leq d_r(e) = d(e) + r(u) - r(v)$  for all edges  $e : u \rightarrow v$ . Thus the general problem may be expressed as the linear programming problem

$$\begin{aligned} &\text{Minimize} \\ &\sum_{v \in V} r(v) (\text{outdegree}(v) - \text{indegree}(v)) \\ &\text{subject to } r(v) - r(u) \leq d(e) \text{ for all edges} \\ &e : u \rightarrow v \text{ in } E. \end{aligned}$$

As noted in [6], the dual of this problem can now be cast as a minimum-cost network-flow problem and solved via methods such as those outlined in [4] or [5].

Org. Dep.	Ret. Dep.	Org. Dep.	Ret. Dep.	Org. Dep.	Ret. Dep.
$(1 \rightarrow 2, 0) = (-1, 0)$	$(5' \rightarrow 4', 1) = (1, 1)$	$(2 \rightarrow 1, 1) = (1, 1)$	$(4' \rightarrow 5', 0) = (-1, 0)$	$(5 \rightarrow 1, 2) = (4, 2)$	$(3' \rightarrow 5', 0) = (-2, 0)$
$(2 \rightarrow 4, 0) = (-2, 0)$	$(4' \rightarrow 2', 1) = (2, 1)$	$(5 \rightarrow 4, 1) = (1, 1)$	$(3' \rightarrow 2', 1) = (1, 1)$	$(4 \rightarrow 4, 2) = (0, 2)$	$(2' \rightarrow 2', 2) = (0, 2)$
$(2 \rightarrow 5, 0) = (-3, 0)$	$(4' \rightarrow 3', 1) = (1, 1)$	$(5 \rightarrow 5, 1) = (0, 1)$	$(3' \rightarrow 3', 1) = (0, 1)$	$(4 \rightarrow 5, 2) = (-1, 2)$	$(2' \rightarrow 3', 2) = (-1, 2)$
$(3 \rightarrow 3, 0) = (0, 0)$	$(1' \rightarrow 1', 0) = (0, 0)$			$(4 \rightarrow 1, 3) = (3, 3)$	$(2' \rightarrow 5', 1) = (-3, 1)$

Table 1: Dependencies of Fig. 1 translated to Fig. 2(b).

To illustrate our point, consider the above example. The in- and outdegrees of most nodes in the DFG in Figure 4(a) match, so in order to find a retiming for this graph, we must minimize  $2(r(I_2) - r(I_1))$  subject to the constraints

$$\begin{aligned}
r(I_1) - r(I_2) &\leq 1 & r(I_5) - r(I_2) &\leq 0 \\
r(I_2) - r(I_1) &\leq 0 & r(I_5) - r(I_4) &\leq 2 \\
r(I_1) - r(I_4) &\leq 3 & r(I_4) - r(I_5) &\leq 1 \\
r(I_4) - r(I_2) &\leq 0 & r(I_1) - r(I_5) &\leq 2
\end{aligned}$$

The first two inequalities of the left column tell us that  $-1 \leq r(I_2) - r(I_1) \leq 0$ , so we can achieve our minimum by setting  $r(I_2) - r(I_1) = -1$ . Assuming that all other retimings will be zero, the final constraint in the right column yields two possible answers:  $r(I_1) = 1$  and  $r(I_2) = 0$ , or  $r(I_1) = 2$  and  $r(I_2) = 1$ . At this stage, the only way to tell which of these is preferable is to retime Figure 4(a) by each, compute the standard deviation of the retimed edge weights for both, and select the function which results in the smaller number. The reader can verify that we should retime by the second of these functions for this reason, leading to the retimed DFG of Figure 4(b) and the retimed data dependencies of Table 1.

In summary, retiming a loop nest may be beneficial in that a properly chosen retiming alters the distances of certain dependencies, which may in turn expedite elimination by our method. The distances change in a consistent, predictable pattern but it remains an open problem to derive a general formula for the characteristic values resulting from retiming. While we are essentially seeking to minimize the sum of the retimed dependence distances, the set of constraints which restrict how we do this appears incomplete, in that we have demonstrated via an example that two answers may be derived by our current method, one of which is clearly superior to the other upon closer scrutiny.

## 4 Conclusion

In this paper we have reviewed our original method from [8] for expressing and studying loop-carried de-

pendencies. We have demonstrated that not all dependencies in a program need considered and have presented an approach for eliminating those that are unnecessary. We then concluded by combining our efforts with the established method of retiming so as to parallelize code even further.

## References

- [1] L.-F. Chao. *Scheduling and Behavioral Transformations for Parallel Systems*. PhD thesis, Dept. of Comput. Sci., Princeton Univ., 1993.
- [2] L.-F. Chao and E. H.-M. Sha. Scheduling data-flow graphs via retiming and unfolding. *IEEE Trans. Parallel & Distributed Syst.*, 8:1259–1267, 1997.
- [3] J.P. Hayes. *Computer Architecture and Organization*. WCB/McGraw-Hill, 1998.
- [4] M. Klein. A primal method for minimal cost flows with applications to the assignment and transportation problems. *Management Science*, 14:205–220, 1967.
- [5] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.
- [6] C.E. Leiserson and J.B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.
- [7] Z. Li and W. Abu-Safah. On reducing data synchronization in multiprocessed loops. *IEEE Trans. Comput.*, C-36:105–109, 1987.
- [8] T.W. O’Neil and E. H.-M. Sha. Minimizing inter-iteration dependencies for loop pipelining. In *Proc. ISCA 13th Int. Conf. Parallel & Distributed Computing Syst.*, pages 412–417, 2000.
- [9] K.K. Parhi. Algorithm transformation techniques for concurrent processors. *Proc. IEEE*, 77:1879–1895, 1989.
- [10] K.K. Parhi and D.G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Trans. Comput.*, 40:178–195, 1991.
- [11] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
- [12] M.E. Wolfe and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel & Distributed Syst.*, 2(4):452–471, 1991.