

Minimizing Inter-Iteration Dependencies for Loop Pipelining*

Timothy W. O’Neil
Dept. of Comp. Science & Eng.
University of Notre Dame
Notre Dame, IN 46556

Edwin H.-M. Sha
Dept. of Computer Science
Erik Jonsson School of Eng. and C.S.
Box 830688, MS EC 31
University 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 this paper we propose a novel approach for eliminating redundant data dependencies from code. We develop new terminology for expressing and studying dependencies and construct a polynomial-time algorithm for reducing the set of dependencies to its absolute minimum. Finally we demonstrate the implications of our work to several areas.

Index terms: Data dependence analysis, redundant dependence elimination, compiler optimization.

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.

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. Eliminating such dependencies permits the more efficient use of system resources, reduces the number of processor stalls and makes code easier to maintain. It also reduces the run-time complexity of loop scheduling algorithms and the hardware complexity of dynamically scheduled processors. In this paper, we will propose a novel approach for finding redundant dependencies, thus facilitating the parallelization of sequential code.

A great deal of work has been completed to date regarding the study of data hazards in programs. Traditional methods similar to ours [1,3] seek to minimize dependencies before execution begins. Our approach differs from these results chiefly by

being more general, and thus adaptable to more sophisticated areas of application. In this we are most like Rong and Tang [11]. However, in addition to simplifying their notation and proofs, we have also provided a polynomial-time algorithm that yields an optimal solution. Another, more complicated *a priori* minimization method that attempts to eliminate dependencies by recognizing the overall pattern of the program in question appears in [7]. Other work seeks to break dependencies at runtime via educated guesses of what value will be produced [2,8] or what instruction will be executed next [10]. Such estimates are made on the spot during execution, when a decision must be made immediately where to proceed next.

As a general overview of this work, consider the loop of Figure 1(a). In the next section, we will demonstrate that there are 11 data dependencies hidden in this code which hinder parallel execution of the loop iterations. However, through systematic examination, we find that only 2 of these dependencies are necessary, the other 9 being combinations of this pair. Thus we need only obey two restrictions to generate the loop’s parallel schedule in Figure 1(b).

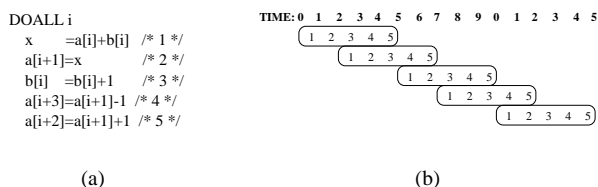


Figure 1: (a) An example; (b) Its schedule.

In this article we demonstrate a new approach to expressing and studying data dependencies in programs. We will show that some dependencies are redundant and are unneeded restrictions as we parallelize code. We will demonstrate a polynomial-time algorithm for eliminating such dependencies, reducing our set of restrictions to the bare minimum before attempting to schedule our program for parallel execution. We conclude by demonstrating how these concepts lay the foundation for new work in dependence analysis that can be applied to a variety of research areas, such as software pipelining and VLIW architectures.

*This work is partially supported by NSF grants MIP95-01006 and MIP97-04276, and by the A. J. Schmitt Foundation.

2 Background

Before proceeding, we should pause for a moment and review the concept of data dependence and the terminology used in this study. All of this material has appeared in print elsewhere [4,12].

Our overall goal is to maximize parallelism in a program. To do this, we need instructions in our program which can simultaneously execute in the computer’s pipeline without causing the pipeline to stall. However, if one instruction depends on the output of another, they can only execute serially, which reduces the amount of parallelism inherent in our program. Furthermore, two such instructions must be executed in their given order and cannot be reordered, as parallel instructions can. We reiterate: dependencies in code keep the pipeline from operating at peak efficiency, prevent a compiler from rearranging code to speed execution, and reduce a program’s parallelism.

Thus, the key to maximizing parallelism in a program is to study the dependencies in the program. There are three different types of dependence:

1. A *data dependence* or *read-after-write (RAW) hazard* occurs when one instruction requires the output of a previous instruction in order to execute. Formally, 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.
2. An *antidependence* or *write-after-read (WAR) hazard* between I_i and I_j happens when I_j writes data that I_i reads and I_i is executed before I_j . Depending on the ordering of the “read” and “write” stages of the pipeline, this may or may not be a problem.
3. An *output dependence* or *write-after-write (WAW) hazard* takes place when both I_i and I_j write to the same register or memory location. In this case, the ordering between the two instructions must be preserved.

More formally, given an instruction I_i , define the set of input operands to I_i as the domain of I_i , and the set of output operands as the range of I_i . Denote the domain and range of I_i as $D(I_i)$ and $R(I_i)$, respectively. Then a RAW hazard exists between instructions I_i and I_j (i.e., I_j is data dependent on I_i) if $R(I_i) \cap D(I_j) \neq \emptyset$; a WAR hazard is present between I_i and I_j if $D(I_i) \cap R(I_j) \neq \emptyset$; and a WAW hazard occurs between I_i and I_j if $R(I_i) \cap R(I_j) \neq \emptyset$.

Sometimes data dependencies are referred to as *true dependencies* while anti- and output dependencies are called *name dependencies*. This is due to the fact that instructions involved in a name dependence only use the same register or memory location; there is no exchange of data between the instructions, as happens in a true dependence. Thus, if the name used in these instructions were changed so that the instructions don’t conflict, the instructions could be executed in parallel or reordered. For this reason we will focus primarily on true dependencies as we study maximizing parallelism.

Finally, 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(a). Referring to each instruction by the number contained in the comments to the right of the code, we see that $R(1) = D(2) = \{x\}$, $R(2) = D(4) = D(5) = \{a[i+1]\}$ and $D(3) = R(3) = \{b[i]\}$, giving us our four intra-iteration RAW hazards quickly, as shown in Figure 2.

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. Since the highest index in any of our original range values is $i + 3$, we will need to unroll up to three times, permitting us to examine four iterations of our loop simultaneously. We do this in Figures 3(a), 3(b) and 3(c). Once the loop has been unfolded, it is simple to examine the domains and ranges of the instructions and discover all loop-carried RAW hazards, as we have shown in the figures.

To summarize, we have found 11 RAW hazards for the code fragment in Figure 1(a): $(1 \rightarrow 2, 0)$, $(2 \rightarrow 4, 0)$, $(2 \rightarrow 5, 0)$, $(3 \rightarrow 3, 0)$, $(2 \rightarrow 1, 1)$, $(5 \rightarrow 4, 1)$, $(5 \rightarrow 5, 1)$, $(5 \rightarrow 1, 2)$, $(4 \rightarrow 4, 2)$, $(4 \rightarrow 5, 2)$ and $(4 \rightarrow 1, 3)$. In the sections to come, we will demonstrate that many of these are redundant and need not be considered when we schedule the code fragment to execute in parallel.

3 Dependencies Across Single Distances

In our example of Figure 1(a), we saw that we had a intra-iteration dependence between lines 2 and 4, and another between lines 2 and 5. However, if we assume that this code is being executed serially, then we are 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 the other dependencies $(2 \rightarrow 4, 0)$ and $(4 \rightarrow 5, 0)$. Thus, when we parallelize this code, we can ignore $(2 \rightarrow 5, 0)$ and focus on satisfying the other dependencies which cannot be dismissed in this fashion. We are reducing the number of constraints that bind us and, in the process, making the parallelization of the code much easier.

In this section and the next we will develop a formal method for studying a code fragment and determining which dependencies are absolutely necessary and which are merely combinations of others. Along the way we will also develop new terminology to describe our situation. We assume that instructions within iterations are being executed serially throughout the remainder of

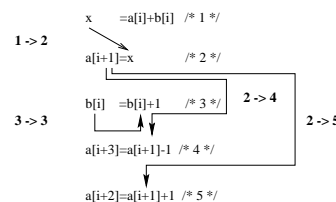


Figure 2: The intra-iteration RAW hazards for the code in Figure 1(a)

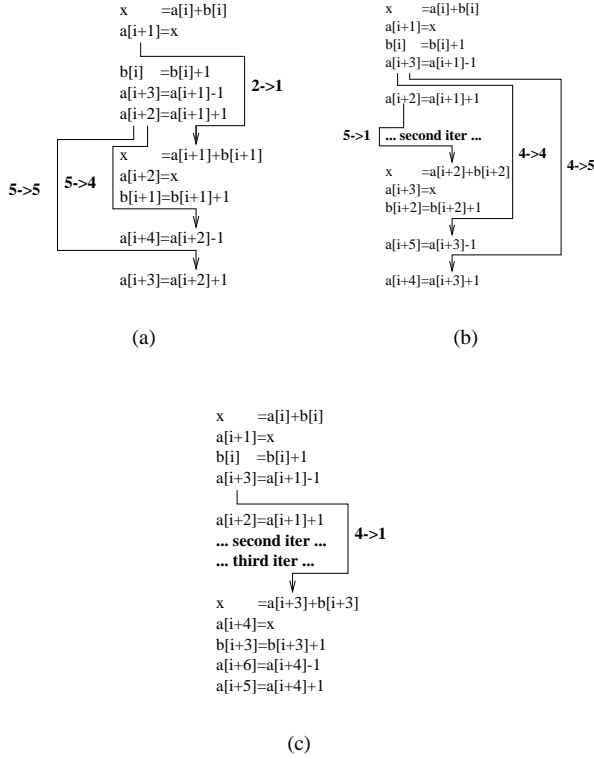


Figure 3: The loop-carried RAW hazards for the code in Figure 1(a): (a) across 1 iteration; (b) across 2; (c) across 3

this paper and focus solely on eliminating redundant loop-carried dependencies.

3.1 Characteristic Values and Subsumption

Given an instruction I in a program, let $\tau(I)$ represent the time at which I executes. As discussed earlier, if we have a data dependence from I_i to I_j , it can only be solved 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 $j - i > 0$.

In the preamble to this section, we found that the data dependence $(2 \rightarrow 5, 0)$ for the code in Figure 1(a) was rendered unnecessary by the dependence $(2 \rightarrow 4, 0)$ and the understanding that our program was being executed serially. In other words, 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 two dependencies A and B subsume each other, i.e. $A \supseteq B$ and $A \subseteq B$, we will say that A *equals* B , which we will of course denote by $A = B$. Finally, we will say that A *properly subsumes* B , written $A \supset B$, if and only if $A \supseteq B$ and $A \neq B$.

With this terminology in mind, it is very easy to show:

Lem 3.1 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 $A \supseteq B$ if $i - j \geq m - n$ and $A \supset B$ if $i - j > m - n$

Pf: Suppose that $A \not\supseteq B$, so that A is satisfied but B isn't. If A is satisfied then $j - i > 0$; if B is not satisfied, $n - m \leq 0$. Therefore $j - i > n - m$, or $i - j < m - n$. By contrapositive argument, $i - j \geq m - n$ implies $A \supseteq B$. The second part is proved similarly. \square

From this we can quickly derive the *translation property* of data dependencies:

Cor 3.1 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 λ , 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 $\lambda_A = i - j$.

From our Lemma we can now show the following:

Thm 3.1 Given an iteration distance d and data dependencies $A : (\lambda_A, d)$ and $B : (\lambda_B, d)$, then:

1. (Trichotomy Property) Exactly one of the following is true: $A \supset B$, $A \subset B$ or $A = B$.
2. $A \supseteq B$ if and only if $\lambda_A \geq \lambda_B$; $A = B$ if and only if $\lambda_A = \lambda_B$; and $A \supset B$ if and only if $\lambda_A > \lambda_B$.

Pf: In (1), for integers λ_A and λ_B , exactly one of three cases can occur: $\lambda_A > \lambda_B$, $\lambda_A < \lambda_B$ or $\lambda_A = \lambda_B$. If $\lambda_A > \lambda_B$ then $A \supset B$ by Lemma 3.1. If $\lambda_A < \lambda_B$ then $A \subset B$ by similar logic. Finally, if $\lambda_A = \lambda_B$ then $A = B$ by the corollary to Lemma 3.1. For (2), that $\lambda_A \geq \lambda_B$ implies $A \supseteq B$ was established in Lemma 3.1. On the other hand, $\lambda_A < \lambda_B$ implies that $A \subset B$ by the same Lemma, or $A \not\supseteq B$ by the Trichotomy Property. The contrapositive of this implication proves our desired result. The other implications are derived similarly. \square

3.2 Characteristic Data Dependencies

Consider now all data dependencies having a given iteration distance d . Because of this last theorem, 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 (Λ_d, d) where $\Lambda_d = \max\{\lambda : (\lambda, d) \text{ is a data dep.}\}$ if there exists a data dependence with iteration distance d , and $-\infty$ otherwise. Returning to our example from Figure 1(a), 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 $(3, 3)$.

We can simplify this a little bit by considering our intra-iteration dependencies with more care. All of these dependencies involve a line of code which is dependent on a previous line of code from the same iteration. Therefore, all intra-iteration dependencies in a program should have characteristic values of at most zero. However, we have noted several times that the assumption

of serial execution adds an implicit intra-iteration dependence of $(-1, 0)$ to any program. Our characteristic data dependence for distance zero is then either $(-1, 0)$ or $(0, 0)$, neither of which is helpful to our cause. For this reason we have reduced our problem to that of studying only loop-carried dependencies and will attempt later to develop *ad hoc* methods for bringing out intra-iteration parallelism.

4 Combining Data Dependencies

Finally, we can begin eliminating some loop-carried dependencies by generalizing our theory to permit simple combinations of 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(a), 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.

4.1 Sums of Data Dependencies

On first glance, adding two data dependencies should yield an obvious solution. However, first impressions are deceiving, as we now show:

Thm 4.1 *Given two data dependencies (λ_a, d_a) and (λ_b, d_b) , then $(\lambda_a, d_a) + (\lambda_b, d_b) = (\lambda_a + \lambda_b + 1, d_a + d_b)$.*

Pf: By the Translation Property, the first of these ordered pairs represents a data dependence between instructions I_{λ_a+1} in iteration 0 and I_1 in iteration d_a . Assuming that the first instruction of the zeroth iteration starts at time 0 and that all instructions are executed serially and in order, we see that I_{λ_a+1} in iteration 0 begins at time λ_a , and so instruction I_1 in iteration d_a can start no sooner than time $\lambda_a + 1$. Now, applying the Translation Property to our second ordered pair yields a data dependence between instructions I_{λ_b+1} of iteration d_a and I_1 of iteration $d_a + d_b$. As before, if I_1 of iteration d_a begins at time $\lambda_a + 1$, then I_{λ_b+1} of the same iteration starts at time $\lambda_a + 1 + \lambda_b$, and so the earliest starting time for I_1 of iteration $d_a + d_b$ is $(\lambda_a + 1 + \lambda_b) + 1 = \lambda_a + \lambda_b + 2$. Since instruction $I_{\lambda_a+\lambda_b+2}$ of iteration zero began execution at timestep $\lambda_a + \lambda_b + 1$, our sum is equivalent to a data dependence between this instruction and I_1 of iteration $d_a + d_b$, which is represented by the ordered pair $(\lambda_a + \lambda_b + 1, d_a + d_b)$. \square

In general, if $A_i = (\lambda_i, d_i)$ is a sequence of n data dependencies, our above analysis can be continued inductively to show that

$$\sum_{i=1}^n A_i = \left(\left(\sum_{i=1}^n \lambda_i + 1 \right) - 1, \sum_{i=1}^n d_i \right).$$

Furthermore, if k is any positive integer, we can define $k \cdot A = (k(\lambda + 1) - 1, kd)$ for any data dependence $A = (\lambda, d)$. We apply this to see that we need only consider the dependencies $(1, 1)$ and $(4, 2)$ when parallelizing our code fragment in Figure 1(a), since $3 \cdot (1, 1) = (5, 3)$, which subsumes $(3, 3)$ by Theorem

3.1. Thus, when we schedule the iterations of our loop, we must ensure that instruction i of iteration d precedes both instruction $i - 1$ of iteration $d + 1$ and instruction $i - 4$ of iteration $i + 2$, as shown in Figure 1(b).

4.2 Dominant Data Dependencies

Earlier we developed the idea of a characteristic data dependence for a given iteration distance and showed how it subsumed all other dependencies for that distance. We now have the possibility that a sum of characteristic dependencies could subsume another characteristic dependence, so that all dependencies of a given distance vanish. The problem lies in decomposing a distance as a sum of other distances and studying all resulting sums of characteristic dependencies. We will now develop the terminology and methods needed for dealing with this problem.

Consider the set of all dependencies for a given distance formed by adding characteristic dependencies having varying distances. We will adopt the language of [6] 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. By Theorem 3.1, it suffices to choose that dependence with the largest characteristic value. Thus, the dominant data dependence for distance d is the data dependence (Δ_d, d) where

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

For example, it is clear from the definition that $\Delta_1 = \Lambda_1$, $\Delta_2 = \max\{\Lambda_2, \Lambda_1 + \Lambda_1 + 1\}$ and $\Delta_3 = \max\{\Lambda_3, \Lambda_1 + \Lambda_2 + 1, \Lambda_1 + \Lambda_1 + \Lambda_1 + 2\}$. To compute Δ_4 requires comparing 5 such sums, and Δ_5 is one of 7 such sums. 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 by noting the following:

Thm 4.2 *For a given iteration distance $d > 1$, let $M = \max\{\Delta_i + \Delta_{d-i} + 1 : i = 1, 2, \dots, \lfloor \frac{d}{2} \rfloor\}$. Then $\Delta_d = \max\{\Delta_d, M\}$.*

Pf: For any integer i between 1 and $\lfloor \frac{d}{2} \rfloor$, $(\Delta_i + \Delta_{d-i} + 1, d)$ is a data dependence of distance d and is therefore subsumed by (Δ_d, d) . In short, $\Delta_d \geq \Delta_i + \Delta_{d-i} + 1$ for any such i , and so $\Delta_d \geq M$, the maximum of all such sums. On the other hand, if $\Delta_d \neq \Lambda_d$, then

$$\begin{aligned} \Delta_d &= \left(\left(\sum_{j=1}^{\lfloor n/2 \rfloor} \Lambda_{i_j} + 1 \right) - 1 \right) \\ &+ \left(\left(\sum_{j=\lfloor n/2 \rfloor + 1}^n \Lambda_{i_j} + 1 \right) - 1 \right) + 1 \quad (1) \end{aligned}$$

for some sequence of indices $\{i_j\}_{j=1}^n$ such that $\sum_{j=1}^n i_j = d$. However, $\sum_{j=1}^{\lfloor n/2 \rfloor} i_j = k < d$ for some integer k , and so each of the partial sums in (1) constitutes the characteristic value of a data dependence, the first with distance k and the second with distance $d - k$. Since each of these must be bounded from above by the characteristic values of their respective dominant dependencies, $\Delta_d \leq \Delta_k + \Delta_{d-k} + 1 \leq M$, the maximum of all such sums.

Hence either $\Delta_d = \Lambda_d$ or $\Delta_d = M$ and we choose Δ_d to be the larger value by definition. \square

We can therefore calculate Δ_d inductively if we know 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 $\Delta_d > \Lambda_d$.

5 The Elimination Algorithm

Our formalized method is 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.

Let us now consider time complexity. As below, let D be the maximum iteration distance. Let n_d be the cardinality of the set of data dependencies having iteration distance d for $d = 1, 2, \dots, D$, while $N = \max n_d$. The **for** loop which eliminates redundancy within each iteration distance executes in $O(ND)$ time, while the **for** loop which eliminates redundancy across multiple distances executes in $O(D^2)$ time. Hence our algorithm is polynomial time.

6 Examples

To demonstrate our algorithm, we now apply it to several variations of our above sample program.

6.1 First Example

For our original example (Figure 1(a)), we begin executing the algorithm with set of dependencies $S = \{(1, 1), (0, 1), (4, 2), (0, 2), (-1, 2), (3, 3)\}$. After executing the initial **for** loop, S is reduced to $\{(1, 1), (4, 2), (3, 3)\}$. The first statement of the second half sets $\Delta[1] = 1$. When $i = 2$ in the succeeding **for** loop, $j = 1$ only, which results in $M = 3$. Since this is smaller than $\Lambda[2] = 4$, we keep the dependence $(4, 2)$ and set $\Delta[2] = 4$. Finally, when $i = 3$, we again have only $j = 1$, which yields $M = 1 + 4 + 1 = 6$. Since this is bigger than $\Lambda[3] = 3$, we remove the dependence $(3, 3)$ and set $\Delta[3] = 6$. When the algorithm ends, $S = \{(1, 1), (4, 2)\}$, exactly the same answer we found earlier.

Let us now interchange statements 4 and 5 in our sample loop. Note that this program performs exactly the same function as the original. Carefully studying this reordered code yields an initial set of dependencies $S = \{(1, 1), (0, 1), (3, 2), (1, 2), (0, 2), (4, 3)\}$. Removing all but the characteristic dependencies in the first half of the algorithm's execution cuts this down to $S = \{(1, 1), (3, 2), (4, 3)\}$. As above,

Algorithm 1 Eliminating Redundant Data Dependencies

Input: A set S of data dependencies for a given program, the maximum 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 (λ, 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 (λ, 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

we pass through only three phases during the second half of the algorithm.

1. Begin by setting $\Delta[1] = \Lambda[1] = 1$.
2. When $i = 2$ and $j = 1$, $M = 3 = \Lambda[2]$ and so we may remove $(3, 2)$ from S and set $\Delta[2] = 3$.
3. Finally, when $i = 3$ and $j = 1$, $M = 1 + 3 + 1 = 5$. Since $M > \Lambda[3]$ we set $\Delta[3] = 5$ and delete $(4, 3)$ from S .

Thus only the data dependence $(1, 1)$ must be considered when parallelizing the loops for this sample program. This may be achieved by starting iteration i at time i for $i \geq 0$.

Note that our first example demonstrates that reordering code results in different sets of necessary data dependencies. There are two implications of this. First, a strict in-order execution of the code is necessary, eliminating the possibility that we can improve execution time by reordering commands “on the fly” during run time. Second, we must study our program carefully to determine a correct order of execution which minimizes data dependencies.

6.2 Second Example

To this point we have not addressed the issue of intra-iteration parallelism, and such a discussion is too large to be contained herein. However, we can study our example and demonstrate that our technique will work when commands within iterations execute in parallel. Recall that the intra-iteration dependencies for the program in Figure 1(a) require line 1 to complete execution

before line 2 begins, which in turn must complete before lines 4 and 5 begin. It therefore follows that we can execute lines 2 and 3 in parallel, followed by the parallel execution of 4 and 5. Figure 4(a) shows our code edited to specify this.

To develop an initial set of dependencies, we view each DOPAR as a single command and translate our original dependencies. For example, the dependence $(5 \rightarrow 1, 2)$ of the original program now represents a dependence from the second DOPAR to line 1. Under our revised notation, this second DOPAR is now line 3 of our program, so this original dependence becomes $(3 \rightarrow 1, 2)$ in the parallelized version of the program. The translation for each dependence from the original version of the program to the parallelized version is displayed in Table 4(b). Basically, for each dependence, occurrences of lines 2 or 3 are changed to 2, while lines 4 and 5 are mapped to the new line 3.

We can see now that our original set of dependencies for the parallelized code is $S = \{(1, 1), (0, 1), (2, 2), (0, 2), (2, 3)\}$, which is pared down to $(1, 1)$ by our algorithm. Therefore, when scheduling our parallelized program, we must insure that the first DOPAR completes execution before the next iteration of the loop starts, as shown in Figure 4(c). This sample program was small enough to study in this manner. It remains for us to develop a method for systematically studying code so that intra-iteration parallelism may be discovered and exploited, even in much larger programs. Such a method could then be combined with already existing techniques like retiming [5] and unfolding [9] to maximize such parallelism before applying the concepts we have developed here for maximizing parallelism of loop iterations.

7 Conclusion

In this paper we have constructed an original method for expressing and studying loop-carried dependencies. We have demonstrated that not all dependencies in a program require consideration. By modeling the problem formally and understanding its basic properties, we have been able to design an efficient polynomial-time algorithm which eliminates all such unnecessary restrictions. We have noted that reducing a program's dependence set to the bare minimum enhances parallelism, allowing us to make more efficient use of a system's resources. We have also noted that this reduction reduces scheduling complexity.

We have also pointed to several directions for future work. It remains for us to determine exactly how to optimally reorder code so as to maximize the inherent inter-iteration parallelism. A systematic method for studying a program and determining which instructions to group together for parallel execution also remains to be constructed. Once this is accomplished, it would be possible to combine our efforts here with other established methods so as to parallelize code even further.

References

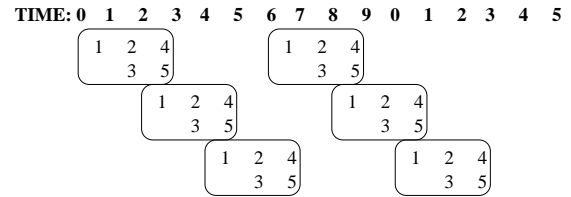
[1] W. Blume and R. Eigenmann. Nonlinear and symbolic data dependence testing. *IEEE Trans. Parallel & Distrib. Syst.*, 9:1180–1194, 1998.
 [2] B. Calder, G. Reinman, and D. Tullsen. Selective value prediction. In *Proc. 26th Int. Symp. Comput. Architect.*, pp. 64–74, 1999.

```
DOALL i
  x=a[i]+b[i] /* 1 */
  DOPAR
    a[i+1]=x /* 2 */
    b[i] =b[i]+1 /* 3 */
  ENDPAR
  DOPAR
    a[i+3]=a[i+1]-1 /* 4 */
    a[i+2]=a[i+1]+1 /* 5 */
  ENDPAR
```

(a)

Org. dep.	New dep.
$(2 \rightarrow 1, 1) = (1, 1)$	$(2 \rightarrow 1, 1) = (1, 1)$
$(5 \rightarrow 4, 1) = (1, 1)$	$(3 \rightarrow 3, 1) = (0, 1)$
$(5 \rightarrow 5, 1) = (0, 1)$	$(3 \rightarrow 3, 1) = (0, 1)$
$(5 \rightarrow 1, 2) = (4, 2)$	$(3 \rightarrow 1, 2) = (2, 2)$
$(4 \rightarrow 4, 2) = (0, 2)$	$(3 \rightarrow 3, 2) = (0, 2)$
$(4 \rightarrow 5, 2) = (-1, 2)$	$(3 \rightarrow 3, 2) = (0, 2)$
$(4 \rightarrow 1, 3) = (3, 3)$	$(3 \rightarrow 1, 3) = (2, 3)$

(b)



(c)

Figure 4: (a) Our original sample loop parallelized; (b) Translation of data dependencies from Figure 1(a) to 4(a); (c) Our sample loop's schedule.

[3] D.-K. Chen and P.-C. Yew. Redundant synchronization elimination for DOACROSS loops. *IEEE Trans. Parallel & Distrib. Syst.*, 10:459–470, 1999.
 [4] J. Hayes. *Computer Architecture and Organization*. WCB/McGraw-Hill, 1998.
 [5] C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.
 [6] Z. Li and W. Abu-Safah. On reducing data synchronization in multiprocessed loops. *IEEE Trans. Comput.*, C-36:105–109, 1987.
 [7] B. Massingill and K. Chandy. Parallel program archetypes. In *Proc. Int. Parallel Process. Symp.*, pp. 290–296, 1999.
 [8] T. Nakra, R. Gupta, and M. Soffa. Value prediction in VLIW machines. In *Proc. 26th Int. Symp. Comput. Architect.*, pp. 258–269, 1999.
 [9] K. Parhi and D. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Trans. Comput.*, 40:178–195, 1991.
 [10] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Parallel & Distrib. Syst.*, 10:160–180, 1999.
 [11] H. Rong and Z. Tang. Eliminate redundant loop-carried flow dependencies for VLIW architectures. *J. Softw.*, accepted for publication.
 [12] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.