

USING UNFOLDING TO MINIMIZE INTER-ITERATION DEPENDENCIES

Timothy W. O'Neil
Computer Science Dept.
University of Akron
Akron, OH 44325-4002
email: toneil@cs.uakron.edu

Edwin H.-M. Sha
Computer Science Dept.
Univ. of Texas at Dallas
Richardson, TX 75083-0688
email: edsha@utdallas.edu

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. Previously 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 unfolding so as to parallelize code even further.

KEY WORDS

Data dependence analysis, redundant dependence analysis, compiler optimization, unfolding.

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 such one 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 al-

tered model then corresponds to a new version of the original program which performs the same task but does so more efficiently [1-3]. 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. Previously we have discussed the effect of retiming [6] on our elimination method [9]. In this paper, we will consider another of the most popular graph transformation techniques (unfolding) and outline how it can be used in conjunction with previous ideas to reduce required data hazards.

In this paper we review our original method from [8] 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 unfolding so as to parallelize code even further.

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 [5, 8, 9, 11].

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.

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.

Sometimes data dependencies are referred to as *true dependencies* while other 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 do not conflict, the instructions could be executed in parallel or reordered. Methods exist for reducing the occurrence of name dependencies [4]. For this reason we will focus primarily on true dependencies as we study maximizing 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 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 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.

```
DOALL i
  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 */
```

Figure 1. An example.

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 we assume

that this code is being executed serially, then we are also assuming an implicit dependence from line 4 to line 5. The 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 this code is parallelized, 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 our code much easier.

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 $j - i > 0$.

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 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 $\lambda_A = i - j$.

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 (Λ_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 dependen-

cies 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 → 1, 3) is subsumed by the combination of (4 → 3, 1) plus (3 → 2, 1) plus (2 → 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 (Δ_d, d) where

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

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 Δ_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.

The complexity of our solution varies with the program to which we apply our algorithm. 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 itera-

tion distance executes in $O(ND)$ time, while the **for** loop which eliminates redundancy across multiple distances executes in $O(D^2)$ time. Hence the first loop dominates if the program has many dependencies over a few distances, while the second dominates in cases where there are few dependencies spread over many distances. However, in any case, our algorithm is polynomial time.

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

/ 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 Unfolding

Unfolding [10] transforms a loop by scheduling multiple iterations simultaneously. We benefit in two ways which we will demonstrate. First, if we represent data dependencies by our notation, unfolding translates dependencies from the original program into new dependencies for the unfolded code in a consistent manner, allowing us to construct a mathematical formula for the translated dependencies. Second, we will see that sufficient unfolding minimizes the set of loop-carried dependencies in all cases. The cost is that the set of intra-iteration dependencies is increased. This is not a problem for us since we assumed in-order execution, but it is a large price to pay in a different environment.

As an example, consider our original example from Figure 1 unfolded twice. (In this case, we say that we have used an *unfolding factor* of 2.) This new code appears below as Figure 2. As we can see, we are simply merging two iterations into one and having our loop counter skip every

other number. We therefore have two copies of each instruction (with one carrying its original label and the other, the original label plus the length of the loop nest prior to unfolding) and two copies of each intra-iteration dependence. In the case of Figure 1, this length is 5, and so we derive the entries in the first three columns of Table 1.

```

DOALL i BY 2
  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 */
  x    =a[i+1]+b[i+1] /* 6 */
  a[i+2]=x            /* 7 */
  b[i+1]=b[i+1]+1    /* 8 */
  a[i+4]=a[i+2]-1    /* 9 */
  a[i+3]=a[i+2]+1    /* 10 */

```

Figure 2. Example 1 unfolded twice.

A similar analytical technique may be applied to the loop-carried dependencies. Because we have two copies of each instruction within our unfolded loop nest, we have two copies of each dependence to consider. As an example, reconsider the loop nest of Figure 2. We begin by considering those dependencies with distance 1. As represented in the first illustration of Figure 3(a), each copy of a dependence extends from an instruction in one iteration to an instruction in the next iteration. When we unfold our loop nest by a factor of 2, we remove every other barrier between iterations and renumber the instructions to the right of a removed barrier. The dependencies as represented by arrows remain the same, we have simply changed the names of the involved instructions and thus the way the dependencies are represented in our notation. A similar trend is evident with the distance 2 dependencies, as pictured in Figure 3(b). Here since every dependence skipped an iteration, all renumbered dependencies reach into the next iteration after unfolding, so all have distance 1. Proceeding in this fashion, we may complete the right-hand column of Table 1 and thus translate the complete list of our dependencies to those for the unfolded code.

Applying our elimination algorithm, we reduce our set of inter-iteration dependences, first to $\{(6, 1), (8, 2)\}$, then to the singleton set $\{(6, 1)\}$ since $8 < 6 + 6 + 1$. Thus, when performing scheduling, there is only one restriction that must be observed, the minimum number of restrictions possible. We can formalize the pattern we have observed:

Theorem 3.1 *If a loop nest of length ℓ is unfolded f times, each dependence (λ, d) of the original loop nest is replaced by the f dependencies $(\lambda + \ell \cdot \left\lfloor \frac{d+k}{f} \right\rfloor \cdot f - d), \left\lfloor \frac{d+k}{f} \right\rfloor$ for $k = 0, 1, 2, \dots, f - 1$ in the unfolded loop nest.*

Proof: Let $(I_i \rightarrow I_j, d)$ be the dependence with $\lambda = i - j$. Thus we must complete instruction I_i of iteration zero be-

fore beginning instruction I_j of iteration d . Renumber all instructions so that each has a unique identifier and numbers are not repeated. Thus the instructions in iteration zero are numbered $1, 2, 3, \dots, \ell$; those in iteration one are numbered $\ell + 1, \ell + 2, \ell + 3, \dots, 2\ell$; those in the second iteration $2\ell + 1$ through 3ℓ ; and so on. Then our notation represents a dependence between instructions I_i and $I_{j+d\ell}$. Not only this, but since there is a copy of I_i every ℓ instructions, we really have a set of dependencies between instructions $I_{i+k\ell}$ and $I_{j+d\ell+k\ell}$ for $k \geq 0$.

Now, when we unfold our loop nest by a factor of f , we create one loop nest containing instructions $I_i, I_{i+\ell}, I_{i+2\ell}, \dots, I_{i+(f-1)\ell}$. Thus we must consider the dependencies between instructions $I_{i+k\ell}$ and $I_{j+(d+k)\ell}$ for $k = 0, 1, 2, \dots, f - 1$. In order to represent these dependencies in our notation, we must translate the subscripts on the copies of I_j so that they fall within our new unfolded loop nest. Thus these subscripts must fall between 1 and $f\ell$ inclusive. We perform this translation by replacing each subscript with its remainder modulo $f\ell$. This is equivalent to subtracting $m f\ell$ for some integer $m \geq 0$ from each subscript. It is clear that this m is counting the number of complete iterations of our unfolded loop nest which pass before we arrive at the iteration containing our desired instruction, and so will be the new distance for our transformed dependence. Since

$$(i + k\ell) - (j + (d + k)\ell - m f\ell) = (i - j) + \ell \cdot (m f - d)$$

our dependencies have the form $(\lambda + \ell \cdot (m f - d), m)$ for some values of m which are dependent on k and which we must now derive.

Since we want our new subscripts to lie between 1 and $f\ell$ inclusive, we must have $1 \leq j + (d + k)\ell - m f\ell \leq f\ell$, and since j is an integer with $1 \leq j \leq \ell$, this will certainly happen if $0 \leq (d + k)\ell - m f\ell \leq (f - 1)\ell$. Dividing all terms by $f\ell$ and simple algebraic manipulations yield $-1 < \frac{1-f}{f} \leq m - \frac{d+k}{f} \leq 0$, or $\frac{d+k}{f} - 1 < m \leq \frac{d+k}{f}$. Since our value of m is also integral, we see that $m = \left\lfloor \frac{d+k}{f} \right\rfloor$ by definition. \square

Based on this result, we can now deduce:

Corollary 3.1 *If a loop nest possesses loop-carried data dependencies, then unfolding cannot entirely eliminate loop-carried dependencies, but unfolding the loop nest by a minimum factor of $f = \max\{d : d \text{ is the distance of a dependence}\}$ is sufficient for reducing the set of loop-carried dependencies to a singleton set.*

Proof: If our loop nest possesses a loop-carried data dependence (λ, d) with $d \geq 1$, then unfolding will replace it by a set of f dependencies (λ_k, d_k) for $k = 0, 1, 2, \dots, f - 1$ where $d_k = \left\lfloor \frac{d+k}{f} \right\rfloor$ by our above result. Thus one of these new dependencies will have distance $d_{f-1} = \left\lfloor \frac{d+f-1}{f} \right\rfloor = \left\lfloor 1 + \frac{d-1}{f} \right\rfloor \geq 1$, and so we will always have at least one

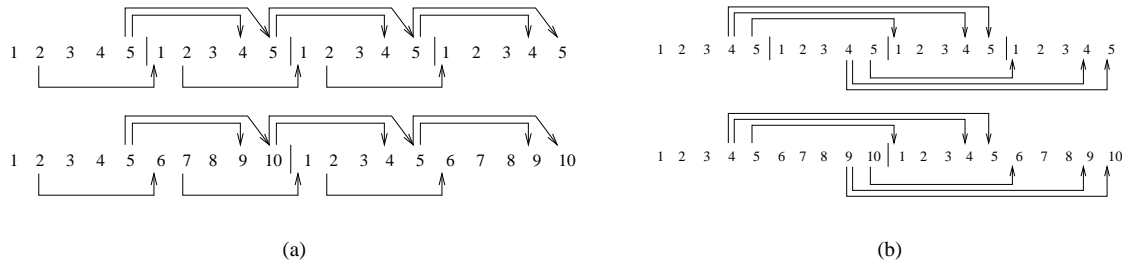


Figure 3. Translation of dependencies: (a) having distance 1; (b) having distance 2.

Org.	Unfolded		Org.	Unfolded	
$(1 \rightarrow 2, 0)$	$(1 \rightarrow 2, 0)$	$(6 \rightarrow 7, 0)$	$(2 \rightarrow 1, 1)$	$(2 \rightarrow 6, 0)$	$(7 \rightarrow 1, 1)$
$(2 \rightarrow 4, 0)$	$(2 \rightarrow 4, 0)$	$(7 \rightarrow 9, 0)$	$(5 \rightarrow 4, 1)$	$(5 \rightarrow 9, 0)$	$(10 \rightarrow 4, 1)$
$(2 \rightarrow 5, 0)$	$(2 \rightarrow 5, 0)$	$(7 \rightarrow 10, 0)$	$(5 \rightarrow 5, 1)$	$(5 \rightarrow 10, 0)$	$(10 \rightarrow 5, 1)$
$(3 \rightarrow 3, 0)$	$(3 \rightarrow 3, 0)$	$(8 \rightarrow 8, 0)$	$(5 \rightarrow 1, 2)$	$(5 \rightarrow 1, 1)$	$(10 \rightarrow 6, 1)$
$(1 \rightarrow 3, 0)$	$(1 \rightarrow 3, 0)$	$(6 \rightarrow 8, 0)$	$(4 \rightarrow 4, 2)$	$(4 \rightarrow 4, 1)$	$(9 \rightarrow 9, 1)$
			$(4 \rightarrow 5, 2)$	$(4 \rightarrow 5, 1)$	$(9 \rightarrow 10, 1)$
			$(4 \rightarrow 1, 3)$	$(4 \rightarrow 6, 1)$	$(9 \rightarrow 1, 2)$

Table 1. Dependencies of Figure 1 translated to Figure 2.

loop-carried dependence. Thus the minimal set we can hope to achieve would be a singleton set.

It is clear that the largest distance which can result among the f dependencies derived from (λ, d) as a result of unfolding by a factor of f is $\left\lfloor 1 + \frac{d-1}{f} \right\rfloor$. If we select $f \geq d$ for all such d , this maximum distance becomes 1 in all cases. Therefore, unfolding by a factor larger than all distances leaves us with loop-carried dependencies whose distances are all one. Our previous elimination algorithm removes all but the one having largest characteristic value, leaving us with only one loop-carried dependence. Thus unfolding by a factor larger than all distances and applying elimination results in a singleton set of dependencies, and the smallest factor which does this is the maximum among our distances. \square

Returning to our sample code in Figure 1, we found that, once redundant dependencies were eliminated, the largest distance possessed by any remaining dependence was 2. Therefore, unfolding this code twice is sufficient for minimizing the number of dependencies for this code, which we have seen. It is our conjecture that these steps (elimination and unfolding) may be performed in any order and yield the same result, but it remains an open problem to formally prove this.

4 Conclusion

In this paper we have reviewed our original method from [8] for expressing and studying loop-carried dependencies. 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 unfolding so as to parallelize code even further. In the process, we constructed a closed formula for an unfolded dependence, as well as a formal method for determining an unfolding factor which optimizes code and maximizes parallelism.

Acknowledgement

This work was partially supported by NSF grants MIP-9501006 and MIP-9704276; 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, NSF grants ETA-0103709 and CCR-0309461, Texas ARP grant 009741-0028-2001 and the TI University program.

References

- [1] L.-F. Chao and E. H.-M. Sha. Retiming and unfolding data-flow graphs. In *Proc. Int. Conf. Parallel Process.*, pages II 33–40, 1992.

- [2] L.-F. Chao and E. H.-M. Sha. Static scheduling for synthesis of DSP algorithms on various models. *J. VLSI Signal Process.*, 10:207–223, 1995.
- [3] 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.
- [4] R. Cytron *et al.* Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13:451–490, 1991.
- [5] J.P. Hayes. *Computer Architecture and Organization*. WCB/McGraw-Hill, 1998.
- [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] T.W. O’Neil and E. H.-M. Sha. Using retiming to minimize inter-iteration dependencies. In *Proc. ISCA 15th Int. Conf. Parallel & Distributed Computing Syst.*, pages 482–487, 2002.
- [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.