CHAPTER 8

Compiler Memory-Hierarchy Management

8.1 Introduction

Increasingly, the performance of a modern computer system is determined by the performance of its memory hierarchy. While on-chip operation speeds have been dramatically improving, the performance of memory has not. Thus, as the processor cycle times come down the latencies to main memory have been increasing. These latencies must be ameliorated if the machine performance is to keep pace with the performance of the processor.

In this chapter, we address four ways that memory latencies can be ameliorated by a compiler:

1. by restructuring the code to increase the usage of data elements in scalar floating-point registers,

2. by restructuring to improve the reuse from cache memories,
3. by taking advantage of "prefetch" instructions on machines that offer those instructions and
4. by improving reuse from vector registers on vector machines.

As we shall show, these techniques can yield dramatic improvements in the performance of modern high-performance processors.

8.2 Scalar Register Allocation

In most compiler courses, we teach students that the problem of scalar register allocation has essentially been solved by the register coloring techniques pioneered by Chaitin and his colleagues at IBM research [Chaitin et al.] and refined by Chow and Hennessy at Stanford [Chow and Hennessy] and by Briggs, Cooper, Kennedy and Torczon at Rice [Briggs, Cooper, Kennedy and Torczon; Briggs, Cooper and Torczon, ...].

These techniques attempt to allocate all of the uses of each scalar variable in single "live range"—a program region in which the given variable is live throughout—to a register. To accomplish this, the compiler typically carries out the following steps:

1. it identifies the live ranges of variables and gives each a unique name,
2. it builds an interference graph to model which pairs of live ranges cannot be assigned to the same register,
3. using a fast heuristic coloring algorithm, it attempts to color the resulting interference graph with a number of colors equal to the number of registers available, and
4. if the coloring fails, it forces at least one live range out of registers and repeats the attempt to color beginning at step 3.

This approach is used in the C and Fortran compilers for every RISC processor that the authors are aware of. In practice, this method is very effective. In most small routines it produces a perfect allocation, in which every non-array variable is assigned to a register throughout every one of its live ranges. Thus, conventional wisdom says that nothing more need to be done to improve register usage on a uniprocessor.

However, conventional wisdom is wrong for the floating-point registers, which are primarily used to hold temporarily individual elements of array variables. You can think of a floating-point register as holding a very small window of information, equal to one word, in the array it represents. To achieve the highest performance, we must try to make as much use as possible of the data in the window before moving on to another element, because each reuse of the element in the window eliminates the need for a load instruction that might take tens or even hundreds of cycles.

To see the importance of these techniques, consider a simple example which can be viewed as a simplified version of matrix multiplication.

```
DO I = 1, N
   DO J= 1, M
      A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

Before this example was widely understoon, almost every industrial compiler would fail to recognize that `A(I)` can be left in a register throughout the inner loop, even though it is very easy to see that the address of `A(I)` does not not change in that loop. The reason for this failure is that uniprocessor compilers only do a good job allocating scalar variables to registers—an array access is treated as an expression to be evaluated on each iteration of the loop. Even though the compiler can use strength reduction to recognize that the address does not change, few compilers extended strength reduction to eliminate the load instruction.

Now consider a slight variation of the program above:

```
DO I = 1, N
   T = A(I)
   DO J= 1, M
      T = T + B(J)
   ENDDO
   A(I) = T
ENDDO
```

This version uses a scalar variable as a temporary resting place for the value of `A(I)` during the execution of the inner loop. All compilers that use a coloring scheme of any kind will properly assign this scalar variable to a register. Thus, the second loop will be much faster than the first, even though it differs only superficially.

This observation led to the development of an optimization, known as *scalar replacement* that converts array references to scalar reference to improve the effectiveness of the compiler's coloring-based register allocator. Although it is designed as a compiler optimization, scalar replacement can be understood and evaluated as a source-to-source transformation.

In the next few sections, we will present a number of such source-to-source transformations that improve the performance of a uniprocessor memory hierarchy, along with algorithms for implementing them.

8.2.1 Data Dependence

There are two applications of dependence in memory heirarchy transformations. The first is a familiar one—to determine the correctness of different transformations.

The second role is a new one—the determination of transformations to improve the performances of particular accesses. As we shall see, we will need to adjust the dependence graph to play this second role, which arises because dependence tells us when two memory references are to the same memory location. We begin by reviewing the different types of dependences and how they can affect memory hierarchy management.

First, consider *true* or *flow* dependence:

```
S1  A(I) =
    ...
S2      = A(I)
```

There is a true dependence from S_1 to S_2 because of the use of `A(I)`. If we can hold the value in a register between these two statements we can save a load instruction.

An *antidependence* does not have an application to improvement of register allocation, but can be useful for cache management:

```
S1      = A(I)
    ...
S2  A(I) =
```

Here, the use of `A(I)` in S_1 will cause the cache block that contains its value to be brought into cache. If the block is not cast out between the two references, it will avoid the cost of the cache load at S_2.

Output dependence can also be useful in cache management, but it has a special use in register usage improvement as well. If

```
S1  A(I) =
```

```
     ...
S2      = A(I)
     ...
S3   A(I) =
```

Here there is an output dependence between S_1 and S_3, along with a true dependence between S_1 and S_2.The true dependence allows us to eliminate a load in statement S_2 but the output dependence tells us that, after the use in S_2, the value of `A(I)` is dead and need not be stored.

For the purposes of memory hierarchy, we will use a fourth kind of dependence, called *input dependence*, in which the source and sink both use the same location.

```
S1      = A(I)
     ...
S2      = A(I)
```

The input dependenc from S_1 to S_2 clearly indicates the opportunity to eliminate a load at the second reference. It is straightforward to modify a standard dependence-testing procedure to compute input dependence. input dependence.

Earlier in this section, we indicated that the dependence graph would need to be modified for use in memory hierarchy management, at least for use in identifying transformation opportunities. To see this, consider the following example:

```
S1   A(I) =
     ...
S2      = A(I)
     ...
S3      = A(I)
```

There are three dependences for `A(I)` in this code fragment:

1. the true dependence from S_1 to S_2,
2. the true dependence from S_1 to S_3 and
3. the input dependence from S_2 to S_3.

Even though there are three dependences, these do not correspond to an opportunity to save three memory references. Both dependences 2 and 3 correspond to elimination of the load at S_3. To ensure that we do not overcount when evaluating optimization opportunities, the dependence graph should be pruned to eliminate dependences like dependence 2 in the example above. Pruning algorithms are well-understood [Carr-Kennedy??, ??]. However, since we will be recommending analternative strategy based on partial redundancy elimination, we will not discuss it further here.

A particularly important use of dependence information occurs in sum reduction loops similar to our earlier example:

```
DO I = 1, N
   DO J= 1, M
      A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

Here we find a true dependence and an output dependence from the statement to itself, both arising from the reference to A(I) and bothcarried by the J-loop. In addition, there is an antidependence from the statement to itself, also carried by the J-loop. Finally, there is an input dependence arising from the reference to B(J) carried by the J loop. These dependences are depicted in Figure 8.1.

FIGURE 8.1 Reduction example with dependences.

`DO I = 1, N`

`DO J= 1, M`

δ_J^{-1}

δ_J^o `A(I) = A(I) + B(J)` δ_I^i

`ENDDO` δ_J

`ENDDO`

In this example, the true dependence carried by the J-loop indicates that a load can be saved if `A(I)` is kept in a register until the next loop iteration, while the output and antidependence show that both the initial load and the store on each iteration may be moved out of the loop entirely.

The loop that results after this transformation would require only one load per iteration, for `B(J)`:

```
DO I = 1, N
   T = A(I)
   DO J= 1, M
      T = T + B(J)
   ENDDO
   A(I) = T
ENDDO
```

As indicated earlier, we use assignment of subscripted array variables to scalars to represent loads, because most scalar register allocation procedures would put such variables in registers, at least within a single subroutine.

8.2.2 Scalar Replacement

We now turn to the development of a general procedure for scalar replacement. We begin with a pruned dependence graph, developed as described in Section 8.2.1. In this graph, each true or input dependence represents an opportunity for eliminating exactly one load or store. If the dependence is loop-independent, there is typically no problem replacing the second reference with a reference to a generated scalar. For example, in the following code, the transformation is straightforward.

```
DO I = 1, N
   A(I) = B(I) + C
   X(I) = A(I)*Q
ENDDO
```

In the pruned dependence graph, there will be a loop-independent dependence edge from the first statement in the loop body to the second. In this case, scalar replacement will require capturing the result of the computation in the first statement in a scalar `t`, inserting a copy from the scalar to `A(I)`, and replacing the reference to `A(I)` in the second statement by a reference to `t`, to produce:

```
DO I = 1, N
   t = B(I) + C
   A(I) = t
   X(I) = t*Q
ENDDO
```

If `t` is assigned to a register in this loop, as we would expect with most compilers, there will be one less load on each iteration than in the original.

Here we insert a discussion of problems with conditional code.

To actually perform this calculation, we organize the work into three phases:

1. First, the collection of *name partitions* must be identified. Each name partition represents a collection of references that can be replaced by access to a single scalar variable, which would ideally be held in a register. Each name partition will have one or more *generators* which are the defining references for the partition. In other words, generators are references at which a copy from a subscripted variable to the scalar name used for this partition will be inserted.
2. Next, the name partitions for which scalar replacement will actually be performed are selected. This phase is necessary because of the poor performance of most scalar register allocators under severe register pressure. In this phase we limit the register pressure that will be generated by actually selecting a set of scalars which can fit into the available registers. The selection is based on the benefits of selection as measured by the number of loads and stores that are eliminated.
3. Finally, the code is modifed to perform replacement. In this phase transformations are performed to minimize the number of register-to-register copies that will occur in the final code.

These phases will be discussed in the next several sections.

8.2.2.1 Identification of Name Partitions

The first step in performing scalar replacement is to carry out an analysis of *available array expressions*, with the goal of determining if the value provided by the source of a dependence is generated on every path to the sink of the dependence. In general, we would like to handle lexically identical references to the same array as a group. However, this leads to difficulties because it does not take into account the modification of the loop induction variable on each iteration. The following example illustrates the problem:

```
      DO I = 1, N
         IF (B(I).LT.0.0) THEN
10          C(I) = A(I) + D(I)
         ELSE
20          C(I) = A(I-1) + D(I)
         ENDIF
30       E(I) = C(I) + A(I)
```

```
ENDDO
```

After the first iteration, the value of `A(I)` is available at the use in statement 20 because the reference is to the same location used in statement 30 on the previous iteration. However, it is not available in statement 10 and it is only partially available at statement 30, since the prior use of `A(I)` in statement 10 does not occur on every iteration.

To address the problem of determining which references may be eliminated in a group like the one above, Carr has developed a variant of availability analysis base on partial redundancy elimination.

(Do we want to include a more detailed treatment of availability??).

It is also useful to conduct *reachability analysis* on the array references under consideration. Taken together, availability and reachability analysis are used to determine which references provide a value for scalar replacement . Although reachability analysis is not needed for correctness, it prevents us from marking a reference as providing a value for scalar replacement when that value is redefined before it is used by a later identical reference. At the very least, this enhances readability of the code and it ensures that we do not miscount the savings to be achieved by a particular scalar replacement. The following example illustrates the issue:

```
DO I = 1, N
   A(I) = B(I-1) + B(I)
   B(I) = E(I) + C(I)
ENDDO
```

In this loop, the second definition of `B(I)` kills the first, so only the second definition reaches the reference to `B(I-1)`.

Once these analyses are done, we have determined which array references can provide values for scalar replacement. We call these references *potential generators* because they generate the value used at the sink of some outgoing dependence. The dependences leaving a generator are called *generating dependences*. Generators are still regarded as potential at this point because we have yet to determine whether scalar replacement will be profitable.

8.2.2.2 Elimination of Scalar Copies

However, the situation is more complicated if the dependence crosses one or more loop iterations:

```
DO I = 1, N
   A(I) = B(I-1) + B(I+1)
ENDDO
```

In this example, there is an input dependence from `B(I+1)` to `B(I-1)` because the same memory location referenced as `B(I+1)` on one iteration is referenced two iterations later as `B(I-1)`. Thus, the distance for the dependence is 2 iterations.

The problem with the dependences that span more than a single iteration is that eliminating the load will require more than one temporary variable. This is because the value computed on any particular iteration will need to be preserved in a register beyond the point where the correspinding value for the next iteration is first referenced. In the current example, which spans two iterations, we will use three different scalar temporaries defined as follows:

```
t1 = B(I-1)
t2 = B(I)
t3 = B(I+1)
```

These relations may be thought of as "loop invariants" which must hold on each iteration of the loop. Now we are ready to consider the code that should be generated to eliminate the load at the dependence sink.

```
t1 = B(0)
t2 = B(1)
DO I = 1, N
   t3 = B(I+1)
   A(I) = t1 + t3
   t1 = t2
   t2 = t3
ENDDO
```

While this code achieves the desired reduction in loads, it does so at the expense of introducing two scalar copy instructions which will be translated to register-register copies by the compiler. While such copies are fairly cheap, they do require instruction issue slots degrading performance unnecessarily. Since these copies implement a permutation of values on machine registers, we can eliminate the need for copies by unrolling to the cycle length of the permutation, which would produced the code in ??.

```
t1 = B(0)
t2 = B(1)
```

```
DO I = 1, MOD(N,3)
   t3 = B(I+1)
   A(I) = t1 + t3
   t1 = t2
   t2 = t3
ENDDO
DO I = 1, N, 3
   t3 = B(I+1)
   A(I) = t1 + t3
   t1 = B(I+2)
   A(I) = t2 + t1
   t2 = B(I+3)
   A(I) = t3 + t2
ENDDO
```

The first loop above, which is called a *preloop*, ensures that the number of iterations remaining when the main loop is entered is a multiple of 3, so that no special testing for loop ending conditions are needed in the main loop. The preloop is exactly the same as the naive solution discussed earlier. Now, however, the main loop is entered after at most two iterations of the preloop and requires no register-register copies while performing fewer loop-ending tests. It is fairly straightforward to prove that the final loop calculates the desired answer.

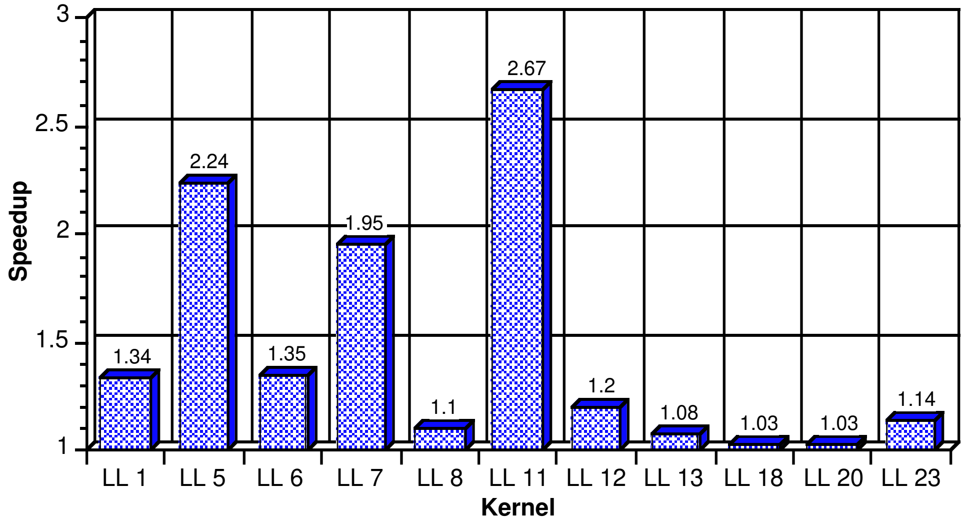
8.2.2.3 Experiments

To illustrate the effectiveness of scalar replacement, we will consider its application to a number of popular benchmark kernels and programs. Note that, unless the compiler on the target machine is exceedingly naive, scalar replacement can never result in a loss of performance.

The results shown here are from Carr's dissertation [Carr]. In his experiments, he ran two versions of the program—the original and the version produced by scalar replacemen applied to the original. Both versions were compiled and run on an IBM RS6000 Model 540. Speedup is computed by dividing the running time of the original by the running time of the version after scalar replaement.

The first chart, in Figure 8.2., shows the speedups achieved on the well-known Livermore loops. The chart shows only the loops for which a speedup occured—for all other loops, there was no change. We can see that the improvements range from a modest 1.03 to an astounding 2.67, all over a compiler with a very good register allocator.

FIGURE 8.2 Scalar Replacement on Livermore Loops.



Carr also tested scalar replaement on a number of well-known kernel programs, including LU decomposition as implemented in LAPACK [], several of the NAS kernels and some kernels he found locally at Rice. The results shown in Figure 8.3, include kernels for LU decomposition with and without pivoting and cover both the point algorithm and a blocked version. Each of these kernels contains invariant array references that require dependence analysis to detect, so these speedups would not be possible with a scalar-only compiler. The two exceptional performances, on Seval and Sor, were due to almost all of the running time being concentrated in a single computation-intensive loop, which scalar replacement successfully optimized.

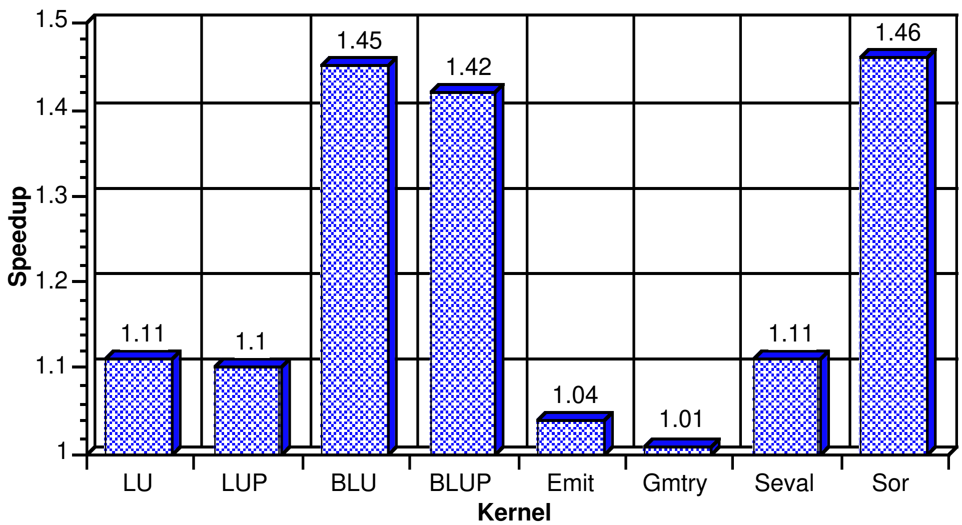
Lin Alg	MM	Matrix multiplication
	LU	LU decompositon
	LUP	LU decomposition with pivoting
	BLU	Block LU decompostion
	BLUP	Block LU Decomposition with pivoting
NAS	Vpenta	??
	Emit	??
	Gmtry	??

TABLE 6 Test kernels for memory transformations.

Geophysics	Fold	Convolution
	Afold	??Convolution
Local	Seval	B-spline evaluation
	Sor	successive over-relaxation

TABLE 6 Test kernels for memory transformations.

FIGURE 8.3 Scalar replacement on linear algebra kernels.



Carr also ran scalar replacement on a number of benchmark applications. These programs were taken from the well-known suites SPEC and Perfect, along with RiCEPS, a suite of compiler benchmarks collected at Rice University and three applications from Rice that are not part of RiCEPS.

Suite	Application	Description
SPEC	Matrix300	matrix multiplicaiton
	Tomcatv	mesh generation
Perfect	Adm	pseudospectral air pollution
	Arc2D	2D fluid-flow solver
	Flo52	transsonic inviscid flow
RiCEPS	Onedim	time-independent schrödinger equation

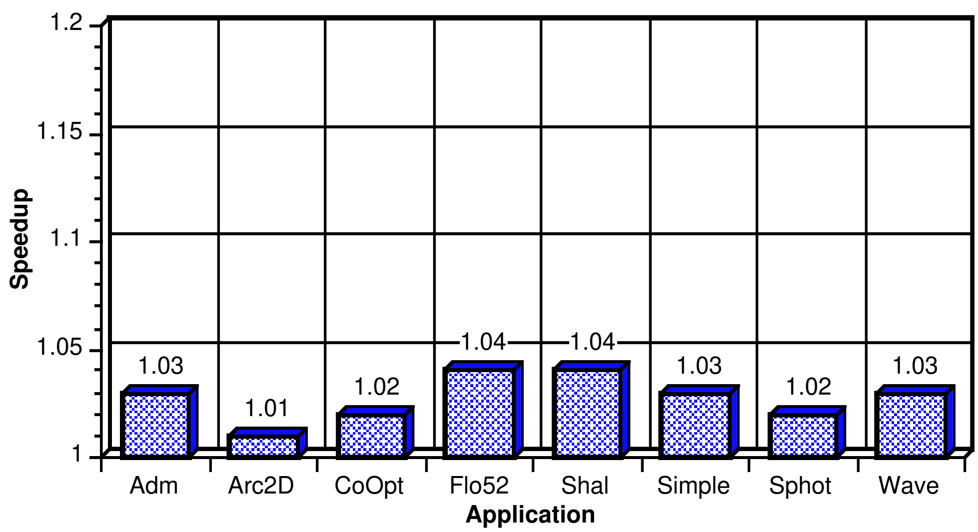
TABLE 7 Test applications memory transformations.

Suite	Application	Description
	Shal	weather prediction
	Simple	2D hydrodynamics
	Sphot	particle transport
	Wave	electromagnetic particle simulation
Local	CoOpt	oil exploration

TABLE 7 Test applications memory transformations.

The performance of these applications after scalar replacement is depicted in Figure 8.4. Once again, any application that is not shown received no improvement. Scalar replacement achieves only modest improvements on these applications, although individual loops within them achieved much more dramatic speedups.

FIGURE 8.4 Scalar Replacement on benchmark applications.



8.2.3 Unroll-and-Jam

We now return to a version of the example of Figure 8.1 on page 282 and ask whether we can achieve some reuse associated with the input dependence on array B.

```
DO I = 1, N*2
   DO J= 1, M
      A(I) = A(I) + B(J)
```

```
   ENDDO
ENDDO
```

As the loop is currently structured, it is unlikely that we can achieve any reuse, because a particular value of `B(J)` must stay in a register throughout the iteration of the `J`-loop until the next iteration of the `I`-loop. Since `N`—the number of iterations of the `J`-loop—is unknown, we must assume that any given machine will not have enough registers to get much reuse for `B`. However, if we make a simple transformation, called *unroll-and-jam,* we can bring pairs of iterations closer together:

```
DO I = 1, N*2, 2
   DO J= 1, M
      A(I) = A(I) + B(J)
      A(I+1) = A(I+1) + B(J)
   ENDDO
ENDDO
```

The essence of this transformation is to unroll the outer loop to multiple iterations and then to fuse the copies of the inner loop. By performing this transformation, we have brought two uses of `B(J)` together so that the new version of the loop performs only onel load of `B(J)` for each two uses. If we use two scalars to hold the two values `A(I)` and `A(I+1)` and one to hold the value of `B(J)`, we will need three registers throughout the inner loop:

```
DO I = 1, N*2, 2
   s0 = A(I)
   s1 = A(I+1)
   DO J= 1, M
      t = B(J)
      s0 = s0 + t
      s1 = s1 + t
   ENDDO
   A(I) = s0
   A(I+1) = s1
ENDDO
```

The innmer loop now requires only one load for every two floating point additions. Although the inner loop still requires a total of `M` loads, it is only executed half as often, so the overall cost of the computation measured in number of loads is only half as much as the original. Further savings can be achieved by unrolling to factors greater than 2.

Unroll-and-jam also improves the efficiency of pipelined functional units. As an example, consider the following loop:

```
DO J = 1, M*2
   DO I = 1, N
      A(I,J) = A(I+1,J) + A(I-1,J)
   ENDDO
ENDDO
```

Here the value computed on one iteration of the inner I-loop is an input to the calculation on the next iteration. This provides good reuse, but causes a problem with the execution pipeline. If the functional unit pipeline is two stages long, the next iteration must wait for two cycles after the current one is started before it may begin. Thus the total time to execute this loop nest can be no better than `2*M*N` cycles. On the other hand, if we use unroll-and-jam on the same example, we get

```
    DO J = 1, M*2, 2
       DO I = 1, N
S1        A(I,J) = A(I+1,J) + A(I-1,J)
S2        A(I,J+1) = A(I+1,J+1) + A(I-1,J+1)
       ENDDO
    ENDDO
```

In this case, we have two independent recurrences sharing the same functional unit. Therefore, the functional unit can alternate work on these two recurrences, launching the addition for statement S_2 on the cycle after the execution of S_2. In this way, both recurrences can be done in the time required to compute one of them in the original loop. Thus, the loop nest takes half the time to execute, since the outer loop has only half as many iterations. After scalar replacement the code looks like this:

```
    DO J = 1, M*2, 2
       s0 = A(0)
       s1 = A(1)
       DO I = 1, N
S1        s0 = A(I+1,J) + s0
S2        s1 = A(I+1,J+1) + s1
          A(I,J) = s1
          A(I+1,J) = s0
       ENDDO
    ENDDO
```

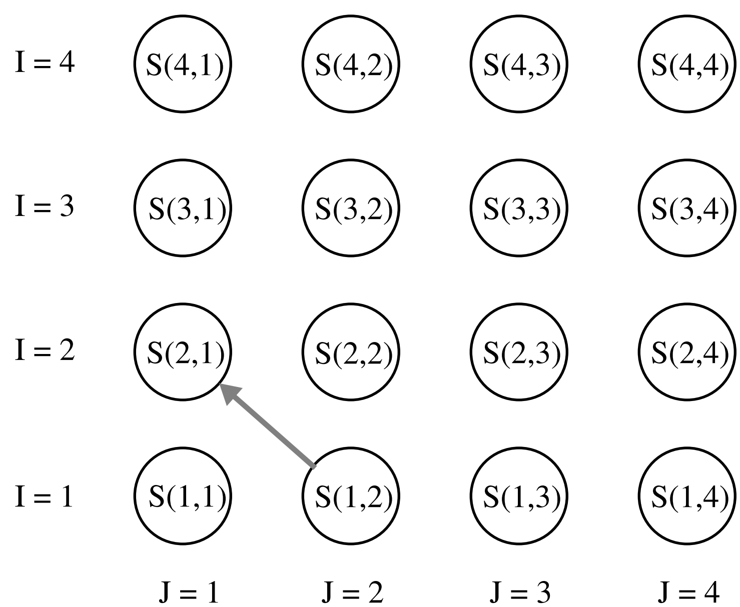
The inner loop has two loads and two stores for each pair of floating point operations.

We now turn to the question of the legality of unroll-and-jam. Clearly, the transformations above are legal. Is it ever illegal? Consider the following loop:

```
DO I = 1, N*2
   DO J = 1, M
      A(I+1,J-1) = A(I,J) + B(I,J)
   ENDDO
ENDDO
```

The dependence pattern for this loop is shown in Figure 8.5. Note that if the J loop is the inner loop, the instance of the statement for I=1 and J=2 is executed before the instance of the statement for I=2 and J=1 because all instances for I=1 are executed before any instance with I=2.

FIGURE 8.5 Legality of Loop Interchange.



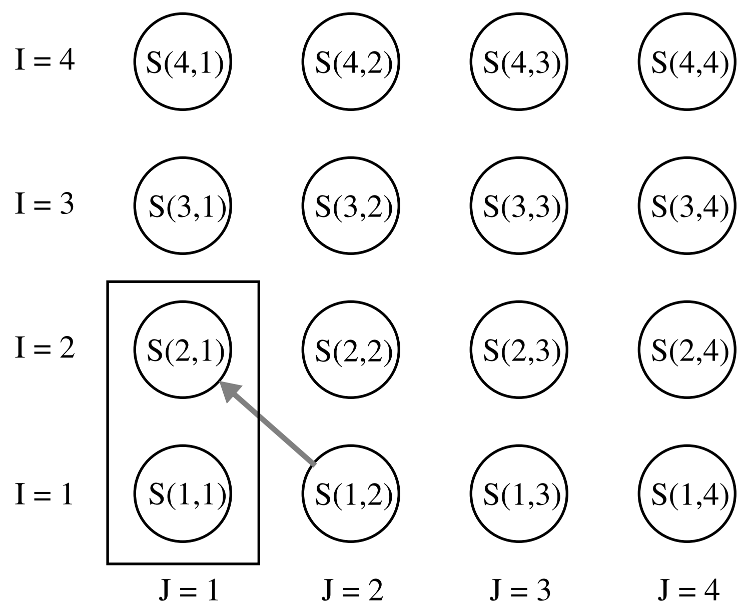
If we perform unroll-and-jam on this loop, unrolling to a factor of 2, we get:

```
DO I = 1, N*2, 2
```

```
      DO J = 1, M
         A(I+1,J-1) = A(I,J) + B(I,J)
         A(I+2,J-1) = A(I+1,J) + B(I+1,J)
      ENDDO
   ENDDO
```

In the transformed loop we are executing two iterations of the I-loop for each value of J. In the diagram in Figure 8.6, these two iterations are represented by the box around S(1,1) and S(2,1).

FIGURE 8.6 Legality of Loop Interchange.



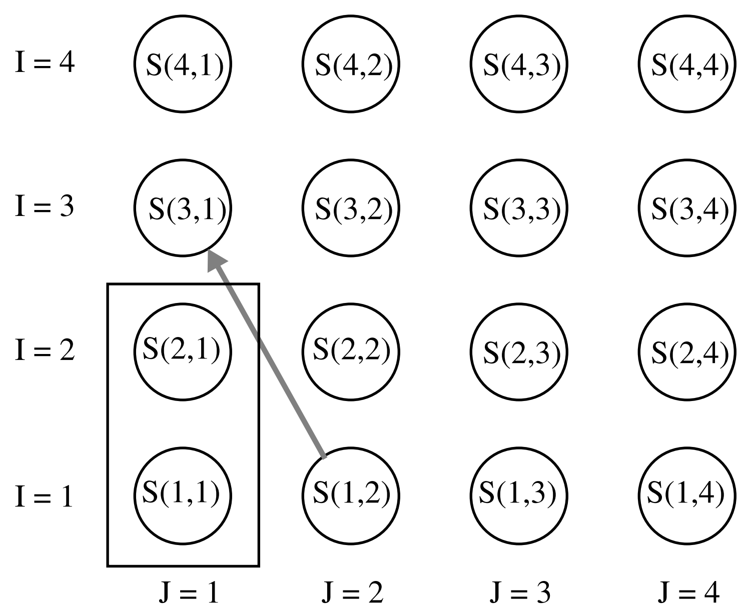
These two iterations are executed *before* the iteration containing S(2,1) and S(2,2). Hence, the dependence originates in a statement that executes after the statement at which it terminates. This is clearly illegal, and means that the unroll-and-jam does not preserve the original meaning of the program.

You might notice that the dependence in this loop has direction vector (<,>), which makes loop interchange illegal. It is not too surprising therefore that unroll-and-jam is illegal, because you can think of it as a loop interchange, followed by unrolling the inner loop (always legal), followed by another loop interchange. Should we then assume that loop unroll-and-jam is illegal whenever loop interchange is illegal? Consider the following variation on our example loop:

```
DO I = 1, N*2
   DO J = 1, M
      A(I+2,J-1) = A(I,J) + B(I,J)
   ENDDO
ENDDO
```

The sink of the dependence is now *two* iterations of the I-loop away, so the dependence pattern is as shown in Figure 8.7.

FIGURE 8.7 Legality of Loop Interchange.



If we unroll once to get two copies of the statement in the inner loop, we are still guaranteed that the source of each dependence will be executed before the sink. On the other hand, unrolling to three copies, would reverse the dependence making that transformation illegal.

We are now ready to state the conditions under which unroll-and-jam is illegal.

> **Definition 8.1.** An unroll-and-jam to *factor n* consists of unrolling the outer loop n-1 times to create n copies of the inner loop and fusing those copies together.
>
> **Theorem 8.1.** An unroll-and-jam to factor n if and only if there exists a dependence with direction vector $(<,>)$ such that the distance for the outer loop is $< n$.

Proof. From the discussion above, the transformation is clearly illegal if such a dependence exists. What if no such dependence exists. Then there are two cases. If no dependence with direction vector $(<,>)$ exists, then loop interchange is legal and unroll-and-jam must be as well. On the other hand, if such a dependence exists it must have distance n or greater. Then, the dependence has its origin in one iteration of the original loop and end in an iteration that is at least n iterations later. These two iterations cannot be in the same iteration group after unroll-and-jam because the factor is n—two iterations in the same group can be at most n iterations apart. □

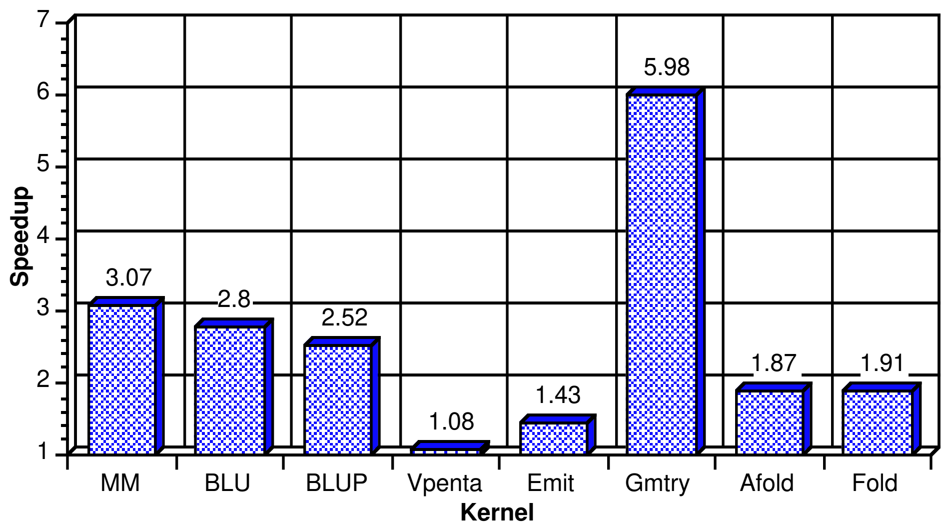
Note here that when we are testing for correctness of unroll-and-jam, we must use the full dependence graph and not the pruned version we use for scalar replacement. This is because, we need to be conservative in preserving the correctness of the program, while we wish to find only the dependences that might correspond to systematic reuse when considering scalar replacement.

8.2.4 Effectiveness

In this section, we present some of Carr's results on the effectiveness of unroll-and-jam and scalar replacement [Carr dissertation, Carrand Kennedy]. In each of the tests below, the original Fortran was run through the experimental loop restructurer developed by Carr at Rice to produce a transformed version. For the purposes of the experiments quoted in this section, the restructurer used only the unroll-and-jam and scalar replacement transformations. Then both the original and transformed versions were compiled and run on an IBM RS/6000, model 540. Speedups were obtained by dividing the running time of the original by that of the transformed version.

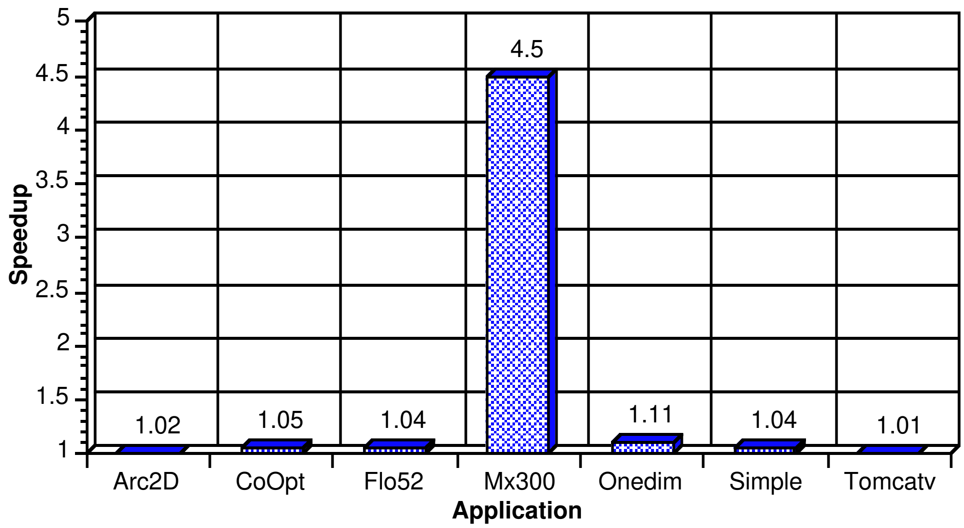
The matrix multiplication kernel (MM in the figure below) was a standard implementation with the sum reduction in the innermost loop. It was applied to a matrix of size 500 by 500. The two LAPACK kernels, BLU (block LU) and BLUP (block LU with pivoting) achieved the stated speedup on 500 by 500 matrices with a blocksize of 32 elements. The three NAS kernels, Vpenta, Emit, and Gmtry, all showed speedups ranging from modest for Vpenta to spectacular for Gmtry. Gmtry and Emit both involved key computational loop nests with outer-loop reductions that were subject to unroll-and-jam. Finally, two geophisical kernels, Afold and Fold, involved the computation of convolutions, which are also nice subjects for unroll-and-jam.WHERE DOES THE DISCUSSION OF TRIANG GO?

FIGURE 8.8 Scalar Replacement plus Unroll-and-jamon various kernels.



Carr also tested the methods on a number of applications from the collection of benchmarks described in Section 8.2.2.3 on page 287. The applications in that collection that are not shown in Figure 8.9 contained no opportunities for unroll-and-jam.

FIGURE 8.9 Scalar replacement plus unroll-and-jam on applications.



Performance improvements for entire applications are relatively modest in general because the effect on any single loop are counterbalanced by code in which the optimization is not applicable. The dramatic improvement for Matrix300, which was reproduced by the Kuck and Associates' front end for the IBM RS/6000 compiler using a similar algorithm, was one of the primary reasons for dropping Matrix300 from the SPEC benchmark suite.

One obvious question is: What is the marginal value of unroll-and-jam over scalar replacement?

FIGURE 8.10 Marginal value of unroll and jam over scalar replacement on kernels.

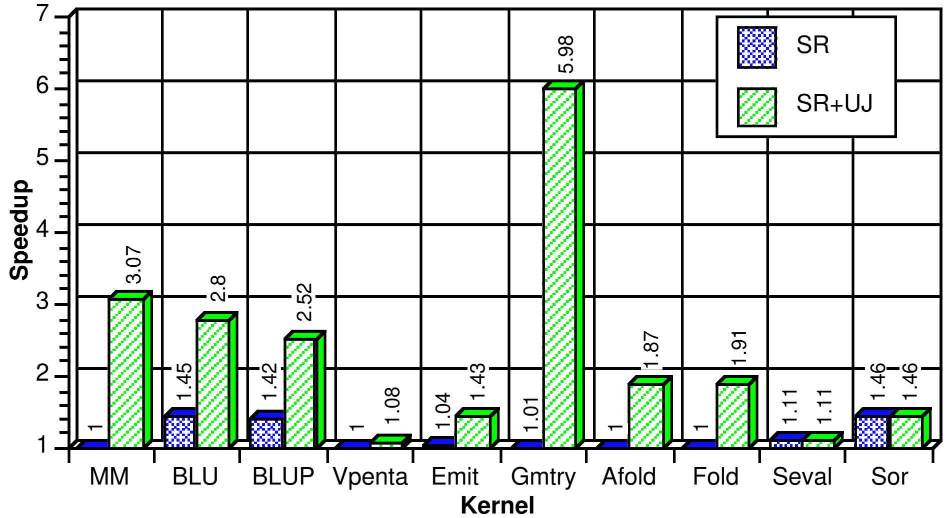
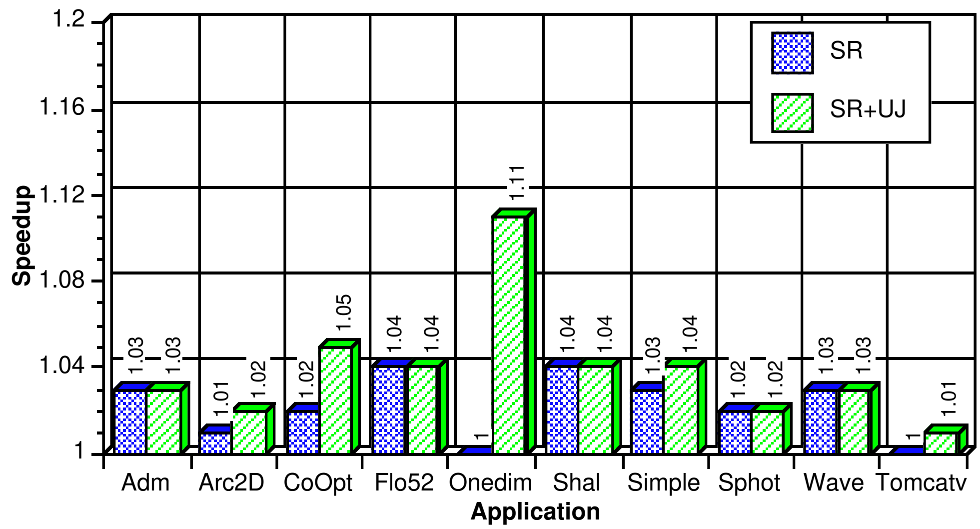


FIGURE 8.11 Marginal value of unroll-and-jam on applications.



Clearly, unroll-and-jam, when it is applicable, can be a powerful tool for optimizing the peformance of scalar floating-point calculations on today's uniprocessors.

8.3 Cache Management

We now turn to the topic of enhancing the performance of cache memory. There are two types of reuse we wish to exploit through program transformations. *Temporal reuse* occurs when you reference exactly the same location in memory a second time. If the cache block containing that location is still in cache, the second access will be much cheaper than a load from memory.

The second type of reuse occurs because a cache block usually contains more than a single data item. Since there are several data items in the same block, when one of these items is referenced , all the others are loaded into cache at the same time. If the accessess to other items in the same block occur a short time after the first, then they are relatively cheap. This kind of reuse—of other items in the same cache block—is called *spatial reuse*.

These two types of reuse give rise to two strategies for improvement. Spatial reuse is highest when iterations of a loop are accessing memory sequentially—that is, when each iteration of a loop accesses a memory location that is adjacent to location used in the previous iteration. Thus, having the right loop within a nest as the inner loop is critical to good cache performance for machines with long cache blocks. For example, in the Fortran loop below

```
DO I = 1, N
   DO J = 1, M
      A(I,J) = A(I,J) + B(I,J)
   ENDDO
ENDDO
```

the inner loop iterates over rows. In Fortran, arrays are stored in column-major order, so successive accesses to `A(I,J)` for `J` fixed and `I` incremented by 1on each step will be to adjacent memory locations. Therefore, performance on machines with multiword cache blocks should be improved by interchanging the two loops:

```
DO J= 1, M
   DO I = 1, N
      A(I,J) = A(I,J) + B(I,J)
   ENDDO
ENDDO
```

Usually, the correct choice for an inner loop is not so clearcut. In the next section, we will discuss strategies for loop interchange to enhance spatial reuse.

A somewhat different problem arises in the case of temporal reuse. Let us assume for the moment that we are dealing with a machine on which each data item occupies a single cache line. On such a machine, temporal reuse is the only type of reuse. Now consider again our standard example loop:

```
DO I= 1, N
   DO J = 1, M
      A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

If we concentrate on cache effects we can see that references to `A(I)` cause a single cache miss—on the first access—for each distinct value of `I`. On the other hand, for `M` large enough, *every* access to `B(J)` will cause a miss, because amost every cache uses a "least recently used" replacement strategy. The implication is that if `M` is larger than the number of words in the entire cache, then `A(1)` is likely to be evicted from cache by accesses that occur in late iterations of the loop. In this case, the total number of misses caused by the loop nest is `N*M`.

Performance can be improved by strip mining the inner loop so that a single strip of length `S` that fits in cache and then moving the loop that iterates over the strips to the outermost position. The result is:

```
DO J = 1, M, S
   DO I = 1, N
      DO jj = J, MIN(M, J+S-1)
         A(I) = A(I) + B(J)
      ENDDO
   ENDDO
ENDDO
```

If `S` is small enough, the inner loop above will give rise to no misses on accesses to `B(J)` except for the first, for a total of `M` misses. However, we now have additional misses for `A(I)`—one miss for each element of `A` for each iteration of the `J`-loop, for a total of `N*M/S` misses, which is far fewer than the number of misses in the original loop for any reasonable value of `S`.

This transformation is known as *strip-mine-and-interchange*, and the general approach is called *cache blocking*. Section 8.3.2 on page 302 discusses methods for cache blocking.

8.3.1 Loop Interchange

Consider a perfectly-nested loop nest with at least two loops. A central issue is: Which loop in the nest should be innermost? This is important because the innermost loop determines which array dimension is accessed sequentially.

8.3.2 Blocking

8.3.3 Effectiveness

8.3.4 Complex Loop Nests

8.3.4.1 Loops with If Statements

Describe Steve Carr's method that uses partial redundancy elimination.

8.3.4.2 Trapezoidal Loops

Here we put trapezoidal unroll-and-jam and loop blocking.

8.3.4.3 Special Purpose Transformations

Here describe the transformation for Gaussian elimination.

8.3.5 Software Prefetching

Material from the Callahan, Kennedy Porterfield paper and Lam et al.

8.4 Vector Register Allocation

Here we insert material from the vector register paper. (Randy, this is for you).

8.4.1 Effectiveness

8.5 Summary

8.6 Historical Notes