More Iteration Space Tiling

Michael Wolfe

Oregon Graduate Institute
Department of Computer Science and Engineering
20000 NW Walker Rd
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 89-003
May, 1989

More Iteration Space Tiling

Michael Wolfe
Oregon Graduate Center
19600 NW von Neumann Drive
Beaverton, OR 97006

Abstract

Subdividing the iteration space of a loop into blocks or *tiles* with a fixed maximum size has several advantages. Tiles become a natural candidate as the unit of work for parallel task scheduling. Synchronization between processors can be done between tiles, reducing synchronization frequency (at some loss of potential parallelism). The shape and size of a tile can be optimized to take advantage of memory locality for memory hierarchy utilization. Vectorization and register locality naturally fits into the optimization within a tile, while parallelization and cache locality fits into optimization between tiles.

1. Introduction

Advanced compilers are capable of many program restructuring transformations, such as vectorization, concurrency detection and loop interchanging. In addition to utilization of multiple vector processors, program performance on large systems now depends on effective use of the memory hierarchy. The memory hierarchy may comprise a virtual address space residing in a large (long latency) shared main memory, with either a shared cache memory serving all the processors or with a private cache memory serving each processor. In addition, vector processors typically have vector registers, which should be used effectively.

It has long been known that program restructuring can dramatically reduce the load on a memory hierarchy subsystem [AbuS78, AbKL81, Wolf87]. A recent paper ([IrTr88]) describes a procedure to partition the iteration space of a tightly-nested loop into *supernodes,* where each supernode comprises a set of iterations that will be scheduled as an atomic task on a processor. That procedure works from a new data dependence abstraction, called the *dependence cone.* The dependence cone is used to find legal partitions and to find dependence constraints between supernodes.

This paper recasts and extends that work by using several elementary program restructuring transformations to optimize programs with the same goals. Our procedures are not restricted to tightly-nested loops and do not depend on a new dependence abstraction, although it will benefit from precise information.

The next section describes and reviews basic concepts such as the iteration space of a loop, data dependence and parallel loop execution. The three most popular data dependence abstractions are explained, along with a new abstraction inspired by [IrTr88]. Section 3 introduces the elementary loop restructuring transformations used in this paper: loop interchanging, loop skewing, strip mining and tiling. Section 4 defines the footprint of an array reference with respect to a loop; this is the basis for optimization for memory locality. Section 5 enumerates the optimization goals for a compiler, while Section 6 describes the optimization process, and includes several examples. The final section has some concluding remarks.

2. Iteration Space and Data Dependence

We say that nested iterative loops (such as Fortran DO or Pascal FOR loops) define an *iteration space,* comprising a finite discrete Cartesian space with dimensionality equal to the loop nest level. For example, the two loops in Program 1 traverse the two-dimensional iteration space shown in Figure 1. The semantics of a serial loop define how the iteration space is traversed (from left to right, top to bottom in our pictures).

Program 1:

```
do I = 1, 5
  do J = 1, 10
    A(I,J) = B(I,J) + C(I)*D(J)
  enddo
enddo
```

There is no reason that the iteration space need be rectangular; many popular algorithms have inner loops whose limits depend on the values of outer loop indices. The iteration space for Program 2 is triangular, as shown in Figure 2, suggesting the name triangular loop. Other interesting iteration space shapes can be defined by nested loops, such as trapezoids, rhomboids, and so on; we will show how some of these shapes can be generated from each other via loop restructuring.

Program 2:

```
do I = 1, 5
  do J = I, 5
    A(I,J) = B(I,J) + C(I)*D(J)
  enddo
enddo
```

Data Dependence: Many compilers available today for vector and parallel computers advertise the ability to detect vector or parallel operations from serial loops. Parallelism is detected by discovering the essential data flow (or data dependence) in the loop and allowing vector or parallel execution when data dependences are not violated. Loop restructuring transformations, such as loop interchanging, are often applied to enhance the available parallelism or otherwise optimize performance; data dependence information is needed to test whether restructuring transformations are legal (the program produces the same answer after restructuring as it did before).

There are three essential kinds of data dependence, though in this paper they will be treated identically. A *flow-dependence* relation occurs when the value assigned to a variable or array element in the execution of one *instance* of a statement is used (read, fetched) by the subsequent execution of an instance of (the same or another) statement. Program 3 has a flow dependence relation from statement S_1 to itself, since the value assigned to A(I+1) will be used on the next iteration of the loop. We usually write this $S_1 \ \delta \ S_1$.

Program 3:

```
     do I = 1, N-1
S1:    A(I+1) = A(I) + B(I)
     enddo
```

An *anti-dependence* relation occurs when the value read from a variable or array element in an instance of some statement is reassigned by a subsequent instance of some statement. In Program 4 there is an anti-dependence relation from S_1 to S_2, since B(I,J) is used in S_1 and subsequently reassigned by S_2 in the same iteration of the loop. We usually write this $S_1 \ \bar{\delta} \ S_2$.

Program 4:

```
     do I = 1, N
       do J = 1, M
S1:      A(I,J) = B(I,J) + 1
S2:      B(I,J) = C(I,J) -1
       enddo
     enddo
```

Finally, an *output dependence* relation occurs when some variable or array element is assigned in an instance of a statement and reassigned by a subsequent instance of some statement. An example of this is Program 5 where there is an potential output dependence relation

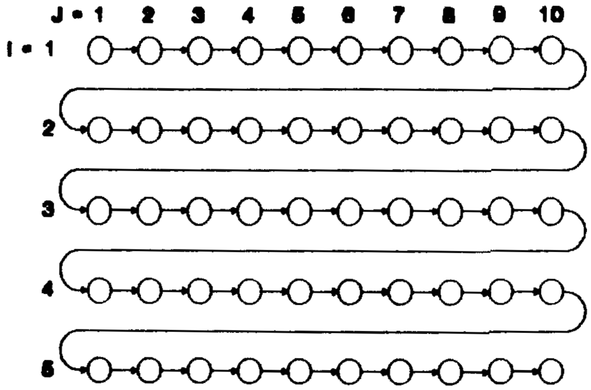

Figure 1.

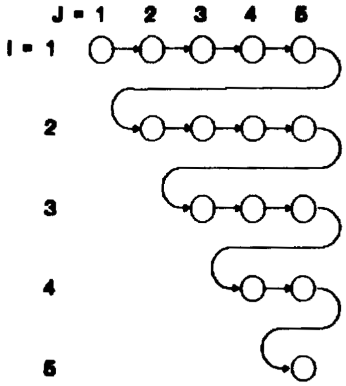

Figure 2.

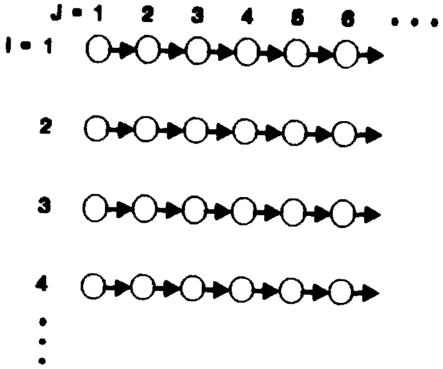

Figure 3.

from S_2 to S_1, since the variable `B(I+1)` assigned in S_2 may be reassigned in the next iteration of the loop by S_1. We usually write this $S_2 \ \delta^{o} \ S_1$. This also shows that the data dependence relations in a program must be approximated; since a compiler will not know the actual paths taken in the program, it must make conservative assumptions.

Program 5:

```
     do I = 1, N-1
S1:     if(A(I) > 0) B(I) = C(I)/A(I)
S2:     B(I+1) = C(I) / 2
     enddo
```

In order to apply a wide variety of loop transformations, data dependence relations are annotated with information showing how they are related to the enclosing loops. Three such annotations are popular today. Many dependence relations have a constant distance in each dimension of the iteration space. When this is the case, a ***distance vector*** can be built where each element is a constant integer representing the dependence distances in the corresponding loop. For example, in Program 6 there is a data dependence relation in the iteration space as shown in Figure 3; each iteration `(i,j)` depends on the value computed in iteration `(i,j-1)`. We say that the distances for this dependence relation are zero in the `I` loop and one in the `J` loop, and we write this $S_1 \ \delta_{(0,1)} \ S_1$.

Program 6:

```
     do I = 1, N
       do J = 2, M
S1:      A(I,J) = A(I,J-1) + B(I,J)
       enddo
     enddo
```

For many transformations, the actual distance in each loop may not be so important as just the sign of the distance in each loop; also, often the distance is not constant in the loop, even though it may always be positive (or negative).

Program 7:

```
     do I = 1, N
       do J = 1, N
S1:      X(I+1,2*J) = X(I,J) + B(I)
       enddo
     enddo
```

As an example, in Program 7 the assignment to `X(I+1,2*J)` is used in some subsequent iteration of the `I` and `J` loops by the `X(I,J)` reference. Let S_1`[I=1,J=1]` refer to the instance of statement S_1 executing for the iteration when the loop variables `I=1` and `J=1`. Then S_1`[I=1,J=1]` assigns `X(2,2)`, which is used by S_1`[I=2,J=2]`, for a dependence distance of `(1,1)`; however, S_1`[I=2,J=2]` assigns `X(3,4)`, which is used by S_1`[I=3,J=4]`, for a dependence distance of `(1,2)`. The distance for this dependence in the `J` loop is always positive, but is not a constant. A common method to represent this is to save a vector of the signs of the dependence distances, usually called a *direction vector*. Each direction vector element will beone of `{+,0,-}` [Bane88]; for historical reasons, these are usually written `{<,=,>}` [WoBa87, Wolf89]. In Program 7, we can associate the direction vector with the dependence relation by writing $S_1 \ \delta_{(<,<)} \ S_1$, where in Program 6, the dependence relation would be written $S_1 \ \delta_{(=,<)} \ S_1$.

Another (often more precise) annotation in inspired by [IrTr88]. Instead of saving only the sign of the distance (which loses a great deal of information about any actual distances), save a set of distance vectors from which all potential actual dependence distances can be formed by linear combination. In Program 7, for example, we would still save the distance vectors `+(1,1)` and `(0,1)`, since all actual dependence distances are linear combinations of these distances; the `+` in `+(1,1)` means that the linear combination must include a non-zero coefficient for this distance vector, while the other coefficients must be non-negative.

Another popular data dependence annotation saves only the nest level of the outermost loop with a positive distance. The dependence relation for Program 6 has a zero distance in the outer loop, but a positive distance in the inner loop, so we would write $S_1 \ \delta^2 \ S_1$. We also say that this dependence relation is *carried* by the inner J loop. Some dependence relations may not be carried by any loop, as in Program 8.

```
Program 8:
          do I = 1, N
            do J = 2, M
S1:           A(I,J) = B(I,J) + C(I,J)
S2:           D(I,J) = A(I,J) + 1
            enddo
          enddo
```

Here the references to A(I,J) produce a dependence relation from S_1 to S_2 with zero distance in both loops. We would thus say $S_1 \ \delta_{(0,0)} \ S_2$ or $S_1 \ \delta_{(=,=)} \ S_2$. Since it is carried by neither of the loops, we call it a *loop independent dependence*, represented $S_1 \ \delta^\infty \ S_2$.

2.1. Parallel Loop Execution

We represent a parallel loop with a **doall** statement. We assume that a parallel loop can be executed in a multiprocessor by scheduling the iterations of the loop on different processors in any order, with no synchronization between the iterations. The iteration space of a **doall** is the same as that of sequential loop except that the traversal of a parallel loop is unordered. If we replace the outer loop of Program 1 by a **doall,** as in Program 9, the iteration space traversal would be as in Figure 4, with a single **fork,** followed by 5 parallel unordered iterations of I (each of which executed the 10 iterations of J), followed by a single **join.**

```
Program 9:
          doall I = 1, 5
            do J = 1, 10
              A(I,J) = B(I,J) + C(I)*D(J)
            enddo
          enddo
```

If we instead replace the inner loop by a **doall,** as in Program 10, the iteration space traversal would be as in Figure 5, where each iteration of the I loop contained a **fork,** followed by 10 parallel unordered iterations of J, followed by a **join.** It is obvious that in most cases, the best performance will result when the number of forks and joins is minimized, which occurs when **doalls** are at the outermost nest level.

```
Program 10:
          do I = 1, 5
            doall J = 1, 10
              A(I,J) = B(I,J) + C(I)*D(J)
            enddo
          enddo
```

It is also obvious that a sequential loop may be converted into a **doall** when it *carries* no dependence relations. For example, in Program 11 there is a flow-dependence relation $S_1 \ \delta_{(1,2)} \ S_1$ due to the assignment and subsequent use of A. Even though the distance in the J loop dimension is non-zero, it may be executed in parallel since the only dependence relation is *carried* by the outer I loop. The outer I loop can be executed in parallel only by the insertion of *synchronization* primitives.

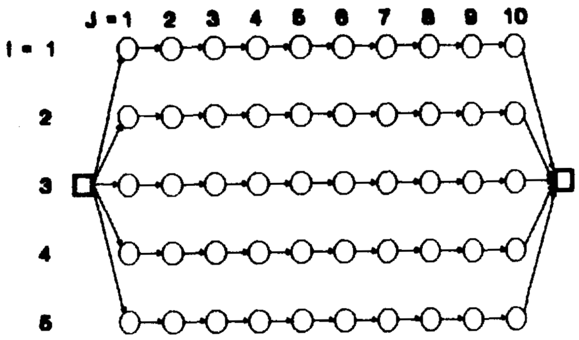

Figure 4.

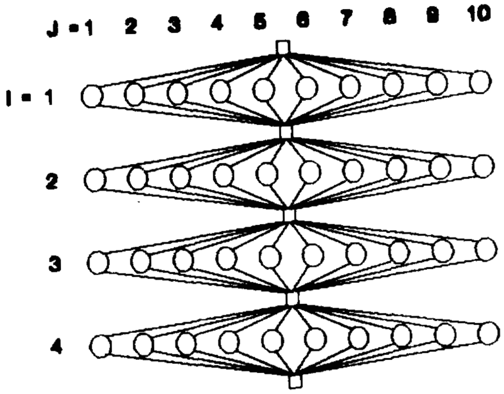

Figure 5.

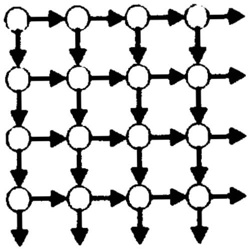

Figure 6.

Program 11:

```
     do I = 2, N
       do J = 3, M
S1:      A(I,J) = A(I-1,J-2) + C(I)*D(J)
       enddo
     enddo
```

3. Restructuring Transformations

The most powerful compilers and translators are capable of advanced program restructuring transformations to optimize performance on high speed parallel computers. Automatic conversion of sequential code to parallel code is one example of program restructuring. Associated with each restructuring transformation is a data dependence test which must be satisfied by each dependence relation in order to apply that transformation. As we have already seen, converting a sequential loop to a parallel **doall** requires that the loop carries no dependence. This *parallelization* has no effect on the data dependence graph, though we will see that other transformations do change data dependence relations somewhat.

Loop Interchanging: One of the most important restructuring transformations is loop interchanging. Interchanging two loops can be used with several different goals in mind. As shown above, the outer loop of Program 11 cannot be converted to a parallel **doall** without additional synchronization. However, the two loops can be interchanged, producing Program 12. Loop interchanging is legal if there are no dependence relations that are carried by the outer loop and have a negative distance in the inner loop (i.e., no (<,>) direction vectors [AlKe84, WoBa87]). The distance or direction vector for the data dependence relation in the interchanged loop has the corresponding elements interchanged, giving the dependence relation $S_1\ \delta_{(2,1)}\ S_1$. Since the outermost loop with a positive distance is the outer J loop, the J loop carries this dependence; now the I loop carries no dependences and can be executed in parallel. Loop interchanging thus enables parallel execution of other loops; this may be desirable if, for instance, it is known that M is very small (so parallel execution of the J loop would give little speedup) or if parallel access to the second dimension of A would produce memory conflicts.

Program 12:

```
     do J = 3, M
       do I = 2, N
S1:      A(I,J) = A(I-1,J-2) + C(I)*D(J)
       enddo
     enddo
```

Loop Skewing: Some nested loops have dependence relations carried by each loop, preventing parallel execution of any of the loops. An example of this is the relaxation algorithm shown in Program 13a. The data dependence relations in the iteration space of this loop are shown in Figure 6; the four dependence relations have distance vectors:

$S_1\ \delta_{(0,1)}\ S_1 \qquad S_1\ \delta_{(1,0)}\ S_1$

$S_1\ \bar{\delta}_{(0,1)}\ S_1 \qquad S_1\ \bar{\delta}_{(1,0)}\ S_1$

One way to extract parallelism from this loop is via the *wavefront* (or hyperplane) method [Mura71, Lamp74]. We show how to implement the wavefront method via loop skewing and loop interchanging [Wolf86].

Program 13a:

```
     do I = 2, N-1
       do J = 2, M-1
S1:      A(I,J) = 0.2*(A(I-1,J)+A(I,J-1)+A(I,J)
                      +A(I+1,J)+A(I,J+1))
       enddo
     enddo
```

Program 13b:

```
     do I = 2, N-1
       do J = I+2, I+M-1
S1:      A(I,J-I) = 0.2*(A(I-1,J-I)+A(I,J-I-1)+A(I,J-I)
                        +A(I+1,J-I)+A(I,J-I+1))
       enddo
     enddo
```

Program 13c:

```
     do J = 4, N+M-2
       do I = MAX(2,CEIL(J-M+1)), MIN(N-1,CEIL(J-2))
S1:      A(I,J-I) = 0.2*(A(I-1,J-I)+A(I,J-I-1)+A(I,J-I)
                        +A(I+1,J-I)+A(I,J-I+1))
       enddo
     enddo
```

Loop skewing changes the shape of the iteration space from a rectangle to a parallelogram. We can skew the J loop of Program 13a with respect to the I loop by adding I to the upper and lower limits of the J loop; this requires that we then subtract I from J within the loop. The skewed loop is shown in Program 13b and the skewed iteration space is shown in Figure 7. The direction vectors for the data dependence relations in the skewed loop will change from (d_1, d_2) to (d_1, d_1+d_2), so the modified dependence relations are:

$S_1\ \delta_{(0,1)}\ S_1 \qquad S_1\ \delta_{(1,1)}\ S_1$
$S_1\ \bar{\delta}_{(0,1)}\ S_1 \qquad S_1\ \bar{\delta}_{(1,1)}\ S_1$

Interchanging the skewed loops requires some clever modifications to the loop limits, as shown in Program 13c. As before, interchanging the two loops requires that we switch the corresponding elements in the direction vectors, giving:

$S_1\ \delta_{(1,0)}\ S_1 \qquad S_1\ \delta_{(1,1)}\ S_1$
$S_1\ \bar{\delta}_{(1,0)}\ S_1 \qquad S_1\ \bar{\delta}_{(1,1)}\ S_1$

Notice that in each case, the direction vector has a positive value in the first element, meaning that each dependence relation is carried by the outer loop (the J loop); thus, the skewed and interchanged I loop can be executed in parallel, which gives us the wavefront formulation.

Strip Mining: Vectorizing compilers often divide a single loop into a pair of loops, where the maximum trip count of the inner loop is equal to the maximum vector length of the machine. Thus, for a Cray vector computer, the loop in Program 14a will essentially be converted into the pair of loops in Program 14b. This process is called strip mining [Love77]. The original loop is divided into strips of some maximum size, the strip size; in Program 14b, the inner loop (or element loop) has a strip size of 64, which is the length of the Cray vector registers. The outer loop (the IS loop, for "strip loop") steps between the strips; on the Cray, the I loop corresponds to the vector instructions.

Program 14a:

```
     do I = 1, N
S1:    A(I) = A(I) + B(I)
S2:    C(I) = A(I-1) * 2
     enddo
```

Program 14b:

```
     do IS = 1, N, 64
       do I = IS, MIN(N,IS+63)
S1:      A(I) = A(I) + B(I)
S2:      C(I) = A(I-1) * 2
       enddo
     enddo
```

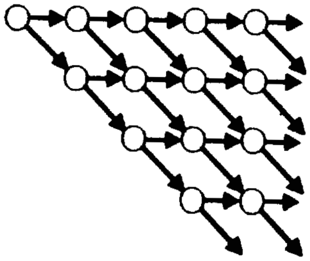

Figure 7.

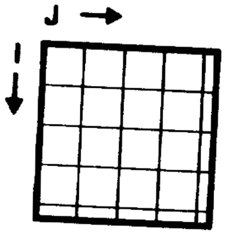

Figure 8.

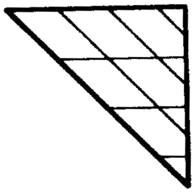

Figure 9a.

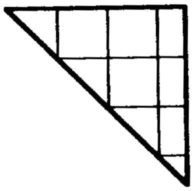

Figure 9b.

Strip mining is always legal; however it does have an effect on the data dependence relations in the loop. As strip mining adds a loop, it adds a dimension to the iteration space; thus it must also add an element to the distance or direction vector. When a loop is strip mined, a dependence relation with a (d) in the distance vector for that loop is changed into one or two dependence relations. If d is a multiple of the strip size ss, then a distance vector (d) is changed to the distance vector (d/ss,0). If d is not a multiple of ss, then a distance vector (d) generates two dependence relations, with distance vectors:

$$\left(\left\lfloor \frac{d}{ss} \right\rfloor, d \bmod ss\right) \quad \left(\left\lceil \frac{d}{ss} \right\rceil, -d \bmod ss\right)$$

In either case, if the original dependence distance is larger than (or equal to) the strip size, then after strip mining the strip loop will carry that dependence relation, allowing parallel execution of the element loop.

Iteration Space Tiling: When nested loops are strip mined and the strip loops are all interchanged to the outermost level, the result is a *tiling* of the iteration space. The double-nested loop in Program 15a can be tiled to become the four-nested loop in Program 15b. This corresponds to dividing the two dimensional iteration space for Program 15a into "tiles", as shown in Figure 8. Each tile corresponds to the inner two element loops, and the outer two "tile" loops step between the tiles.

Program 15a:

```
          do I = 1, N
            do J = 1, N
S1:           A(I,J) = A(I,J) + B(I,J)
S2:           C(I,J) = A(I-1,J) * 2
            enddo
          enddo
```

Program 15b:

```
          do IT = 1, N, SS
            do JT = 1, N, SS
              do I = IT, MIN(N, IT+SS-1)
                do J = JT, MIN(N, JT+SS-1)
S1:               A(I,J) = A(I,J) + B(I,J)
S2:               C(I,J) = A(I-1,J) * 2
                enddo
              enddo
            enddo
          enddo
```

Tiling irregular iteration spaces is slightly more complicated than simply strip mining each loop. A triangular loop, such as Program 16a, when tiled by independently strip mining each loop, produces iteration space tiles as shown in Figure 9a. The desired tiling pattern is shown in Figure 9b; to get this pattern, the loops must be tiled as in Program 16b.

Program 16a:

```
          do I = 1, N
            do J = I, N
              . . .
```

Program 16b:

```
          do IT = 1, N, SS
            do JT = IT, N, SS
              do I = IT, MIN(N, IT+SS-1)
                do J = MAX(JT,I), MIN(N, JT+SS-1)
                  . . .
```

Triangular loop limits like this appear in many linear algebra algorithms, or may appear after other restructuring transformations, such as loop skewing.

4. Footprints in the Iteration Space

The portion of an array which is touched by an array reference within a loop is called the *footprint* of that array reference. For instance, in Program 17a, the footprint of the A(I,1) reference is the first column of A, while the footprint of B(2,I) is the second row of B. Since we do not know the upper limit of the loop, we must assume that the entire column or row may be touched; thus we assume that the footprint of C(I) is the entire vector C. When the loop limits are known, more knowledge can be extracted from the array references. For instance, if the loop in Program 17a were strip-mined, as in Program 17b, the footprint of each array in the *inner loop* would be only 16 elements.

Program 17a:

```
do I = 1, N
  A(I,1) = B(2,I) * C(I)
enddo
```

Program 17b:

```
do IS = 1, N, 16
  do I = IS, MIN(N, IS+15)
    A(I,1) = B(2,I) * C(I)
  enddo
enddo
```

The size of a footprint of an array reference is bounded by the product of the trip counts of the loops whose loop variables appear in the array subscripts (assuming all other induction variables are replaced by functions of the appropriate loop index variables). An array footprint can be calculated for each loop by setting the trip count for all outer loops to one. The total footprint of a loop is the sum of the footprints for all the arrays in the loop.

Thus, at each level of a nested loop we can find the footprint for each array reference in the loop. The footprint size is related to the amount of data that needs to be passed between levels of a memory hierarchy in order to execute that loop level. We would like the footprints for inner loops to be small, so that they can fit higher levels of the memory hierarchy. When loop limits are unknown, the size of a footprint may be unbounded; we use tiling to fix the size of footprints in the inner loops.

When the size of a footprint of some array in a loop is smaller than the iteration space of that loop, then there is some *reuse* of elements of the array in that loop. If that footprint fits into the highest level of the memory hierarchy, then it may already be available at that level the second and subsequent time it is needed, enabling faster execution of the program.

For instance, in Program 18a the footprint of C in the inner loop is only one element, but the footprint of B is M elements; we could interchange the loops, but that would lead to a footprint of N elements for C. Tiling the loops gives Program 18b; now within a tile the footprints of both B and C are only 32 elements, while the tile size itself is 1024 iterations. Thus, each element of B and C is used 32 times; if the 32 elements in the footprint of B and C can fit into the highest level of the memory hierarchy (cache, registers, local memory), then they need to be loaded only once (either automatically, as in a hardware-managed cache, or by additional software, as for registers).

Program 18a:

```
do I = 1, N
  do J = 1, M
    A(I,J) = B(J) * C(I)
  enddo
enddo
```

Program 18b:

```
    do I = 1, N, 32
      do J = 1, M, 32
        do I = IT, MIN(N, IT+31)
          do J = JT, MIN(M, JT+31)
            A(I,J) = B(J) * C(I)
          enddo
        enddo
      enddo
    enddo
```

We note here that there may be other problems with finding and optimizing for footprints. First, given a cache memory environment, a *cache line* may be more than one word wide. On the Sequent Symmetry, for example, a cache line is 2 words wide; thus, when a word is loaded into the cache memory, one of its neighbors is dragged along also, whether or not it is wanted. If a footprint comprised (say) 32 consecutive words, then at most 2 unneeded words would be dragged into the cache; if however the footprint comprised a row in an array stored by columns, then each word would drag another word into the cache. This could potentially double the amount of cache memory used for this footprint; wider cache lines exacerbate the problem. This (or other considerations) may induce a preferred ordering when processing tiles.

Second, for software managed memory hierarchies, we need to not only optimize the footprint size, but we need to be able to identify it. Usually this is no problem, as it will consist of a starting position, a stride and a length.

5. Optimization Goals

Given our toolkit of restructuring transformations, we wish to optimize nested loops for execution on multiprocessors with a memory hierarchy, where each processor may also have vector instructions. We tile the iteration space such that each tile will be a unit of work to be executed on a processor. Communication between processors will not be allowed during execution of a tile. Tiles will be optimized to provide locality and vector execution. The scheduling of tiles onto processors will be done to provide either locality across parallel tiles or not, depending on the memory hierarchy organization.

Atomic Tiles: Each tile is a unit of work to be scheduled on a processor. Once a tile is scheduled on a processor, it runs to completion without preemption. A tile will not be initiated until all dependence constraints for that tile are satisfied, so there will never be a reason that a tile, once started, should have to relinquish the processor.

Parallelism between Tiles: As much as possible, the tiles should be arranged in the iteration space to allow for as much parallelism between tiles as possible. If there is dependence in one dimension and not another, then the tile size may be adjusted so that each tile has a small size in the independent dimension to allow for more independent tiles along that dimension. Depending on how parallelism is implemented, the tile loops may need to be reordered and/or skewed to implement synchronization between tiles.

Vectors within Tiles: If the processors have vector instructions, then the innermost loop should be vectorized. This corresponds to ordering the element loops so that the innermost element loop is vector. This goal may be somewhat inconsistent with the next goal.

Locality within Tiles: The size of the tiles will be adjusted so as to provide good usage of the memory hierarchy. When no data reuse occurs, the ordering of the loops within a tile will not matter (there is no locality anyway); when data reuse does occur, the ordering of the loops will be optimized to take advantage of locality at least in the first and second levels.

Locality between Tiles: In the best case, all the data for a single tile will fit into the highest level of the memory hierarchy (cache, perhaps) allowing the optimizer to look for reuse between tiles. When adjacent tiles in the iteration space share much or all of the data, then the optimizer should try to schedule those tiles on the same processor. If multiple processors share a cache, then parallel tiles which share much of the same data should be scheduled onto those processors at the same time to take advantage of the shared cache. If multiple processors do not share a cache, then parallel tiles scheduled at the same time should be those which do not share data, to prevent memory interference.

6. Optimization Process

The tiling optimization process consists of several distinct steps, described below:

1) The iteration space may be reshaped, through loop skewing. This will give differently shaped tiles in the next step.

2) The iteration space is tiled. Tiling essentially consists of strip-mining each loop and interchanging the strip loops outwards to become the tile loops, though there are some slight complexities that should be handled properly for triangular loop limits. The tile size in each dimension is set in the next two steps.

3) The element loops are reordered and optimized. We can optimize for locality by reordering until the inner loops have the smallest total footprint. We may also optimize for vector instructions or memory strides in the inner loop. The iteration space of the tile may be reshaped via loop skewing and loop interchanging in this step also. Some limits on tile sizes may be set in this step to provide for locality within certain levels of the memory hierarchy (such as vector registers).

4) The tile loops are reordered and optimized. Again, this may involve reshaping the tile iteration space via loop skewing and interchanging. The optimization at this level will depend on the model of parallelism used by the system, and the dependence constraints between tiles. The method described in [IrTr88] has one outermost serial loop surrounding several inner parallel tile loops, using loop skewing (wavefronting) in the tile iteration space to satisfy any dependence relations. We also wish to take advantage of locality between tiles by giving each processor either a rule for which tile to execute next or at least a preference for which direction in the tile iteration space to process tiles to best take advantage of locality. The sizes of the tiles are also set at this time.

Let us show some simple examples to illustrate the optimization process.

Example 1: Given a simple nested sequential loop, such as Program 19a, let us see how tiling would optimize the loop for multiple vector processors with private caches. For a simple vector computer, we would be tempted to interchange and vectorize the I loop, because it gives a chained multiply-add vector operation and all the memory references are stride-1 (with Fortran column-major storage; the J loop would be used otherwise); this is shown in Program 19b. However, if the column size (N) was larger than the cache size, each pass through the K loop would have to reload the whole column of A and B into the cache.

Program 19a:

```
do I = 1, N
  do J = 1, M
    A(I,J) = 0.0
    do K = 1, L
      A(I,J) = A(I,J) + B(I,K)*C(K,J)
    enddo
  enddo
enddo
```

Program 19b:

```
do J = 1, M
  A(1:N,J) = 0.0
  do K = 1, L
    A(1:N,J) = A(1:N,J) + B(1:N,K)*C(K,J)
  enddo
 enddo
```

For a simple multiprocessor, we might be tempted to interchange the J loop outwards and parallelize it, as in Program 19c, so that each processor would operate on distinct columns of A and C. Each pass through the K loop would again have to reload the cache with a row of B and column of C if L is too large.

Program 19c:

```
doall J = 1, M
  do I = 1, N
    A(I,J) = 0.0
    do K = 1, L
      A(I,J) = A(I,J) + B(I,K)*C(K,J)
    enddo
  enddo
enddo
```

Instead, let us attempt to tile the entire iteration space. We will use symbolics for the tile size in each dimension, since determining the tile sizes will be done later. Tiling the iteration space can proceed even though the loops are not perfectly nested. Essentially, each loop is strip-mined, then the strip loops are interchanged outwards to become the tile loops. The tiled program is shown in Program 19d.

Program 19d:

```
do IT = 1, N, ITS
  do JT = 1, M, JTS
    do I = IT, MIN(N,IT+ITS-1)
      do J = 1, MIN(M,JT+JTS-1)
        A(I,J) = 0.0
      enddo
    enddo
    do KT = 1, L, KTS
      do I = IT, MIN(N,IT+ITS-1)
        do J = 1, MIN(M,JT+JTS-1)
          do K = 1, MIN(L,KT+KTS-1)
            A(I,J) = A(I,J) + B(I,K)*C(K,J)
          enddo
        enddo
      enddo
    enddo
  enddo
enddo
```

Each set of element loops is ordered to provide the kind of local performance the machine needs. The first set of element loops has no locality (the footprint of A is ITS×JTS, the same size as the iteration space), so we need only optimize for vector operations and perhaps memory stride; we do this by vectorizing the I loop.

```
IVL = MIN(N,IT+ITS-1)
do J = 1, MIN(M,JT+JTS-1)
  A(IT:IT+IVL,J) = 0.0
enddo
```

The second set of element loops can be reordered 6 ways; the JKI ordering gives stride-1 vector operations in the inner loop, and one level of locality for A in the second inner loop (the footprint of A in the K loop is ITS while the iteration space is ITS×KTS. Furthermore, if ITS is the size of a vector register, the footprint of A fits into a vector register during that loop, meaning that the vector register load and store of A can be floated out of the K loop entirely.

```
          IVL = MIN(N,IT+ITS-1)
          do J = 1, MIN(M,JT+JTS-1)
            do K = 1, MIN(L,KT+KTS-1)
              A(IT:IT+IVL,J) = A(IT:IT+IVL,J) + B(IT:IT+IVL,K)*C(K,J)
            enddo
          enddo
```

Since there are no dependence constraints between tiles along the IT and JT dimensions, those two loops can be executed in parallel. The method suggested in [IrTr88] will 'wavefront' the tile iteration space by having one sequential outermost loop surrounding parallel **doalls**; thus, the final program would as in Program 19e. Note that the tile loops had to be distributed (their formulation only dealt with tightly nested loops); also, the nested **doalls** inside the KT loop will generate KTS **fork/join** operations. If processors are randomly assigned to iterations of the **doalls** (and thus to tiles), the system will not be able to take advantage of locality between tiles.

Program 19e:

```
    doall IT = 1, N, ITS
      doall JT = 1, M, JTS
        IVL = MIN(N,IT+ITS-1)
        do J = 1, MIN(M,JT+JTS-1)
          A(IT:IT+IVL,J) = 0.0
        enddo
      enddo
    enddo
    do KT = 1, L, KTS
      doall IT = 1, N, ITS
        doall JT = 1, M, JTS
          IVL = MIN(N,IT+ITS-1)
          do J = 1, MIN(M,JT+JTS-1)
            do K = 1, MIN(L,KT+KTS-1)
              A(IT:IT+IVL,J) = A(IT:IT+IVL,J) + B(IT:IT+IVL,K)*C(K,J)
            enddo
          enddo
        enddo
      enddo
    enddo
```

Another obvious method is to leave the parallel **doalls** outermost, as in Program 19f. This generates a single **fork/join** operation, but the size of the parallel task is much larger, meaning there is less opportunity for load balancing. However, a single parallel task now comprises all KTS tiles along the KT dimension. Each iteration of the KT loop uses the same the footprint of A, so scheduling all iterations on the same processor takes advantage of that locality between tiles.

Program 19f:

```
doall IT = 1, N, ITS
  doall JT = 1, M, JTS
    IVL = MIN(N,IT+ITS-1)
    do J = 1, MIN(M,JT+JTS-1)
      A(IT:IT+IVL,J) = 0.0
    enddo
    do KT = 1, L, KTS
      IVL = MIN(N,IT+ITS-1)
      do J = 1, MIN(M,JT+JTS-1)
        do K = 1, MIN(L,KT+KTS-1)
          A(IT:IT+IVL,J) = A(IT:IT+IVL,J)  + B(IT:IT+IVL,K)*C(K,J)
        enddo
      enddo
    enddo
  enddo
enddo
```

Example 2: The example used in [IrTr88] is a five point difference equation, as was shown in Program 13. We will show how our methods can derive the two partitionings shown in their paper.

The first partition (Figure 2 of [IrTr88]) starts by skewing the iteration space before tiling, as in Figure 10a; each tile is executed with vector instructions along the I dimension. To satisfy data dependence relations between vertically adjacent tiles, the tile iteration space is then skewed again, as in Figure 10b; in this figure, vertically aligned tiles can be executed concurrently on different processors. This could be implemented by a wavefront method (sequential loop surrounding a **doall**), or by assigning tiles long the J dimension to the same processor and synchronizing between tiles along the I dimension.

The second partition (Figure 6 of [IrTr88]) tiles the iteration space first, as in Figure 11a, then skews each tile to get vector operations, as in Figure 11b. Finally, the tile iteration space is skewed to satisfy dependences between vertically adjacent tiles, resulting in Figure 11c; again, processors can be assigned to rows with time flowing to the right.

7. Summary

We have described several elementary program restructuring transformations which can be combined with parallelism detection to optimize programs for execution on parallel processors with memory hierarchies. These techniques are similar to those described in [IrTr88], but are more general and simpler to apply in the setting of a compiler or other program restructuring tool. Before any of these techniques are implemented in a compiler we need to understand the complexity of the optimization process. Given the data dependence information, it is simple to discover whether and how a loop can be tiled. The difficulty is trying to find the optimal loop ordering. This can be a `O(d!)` problem, where `d` is the loop depth, since we may have to consider each possible loop ordering. This is then complicated by the possibility of skewing the iteration space before tiling or skewing each tile individually. The procedure used here does have the advantage of decoupling the optimization of the code within a tile from optimization between tiles, reducing the complexity from `O((2d)!)` to just `O(d!)`. For loops that are not very deeply nested, the actual computation at each step in the optimization process is relatively small (computation of the footprints and dependences between iterations), so an exhaustive search of the loop orderings may be reasonable.

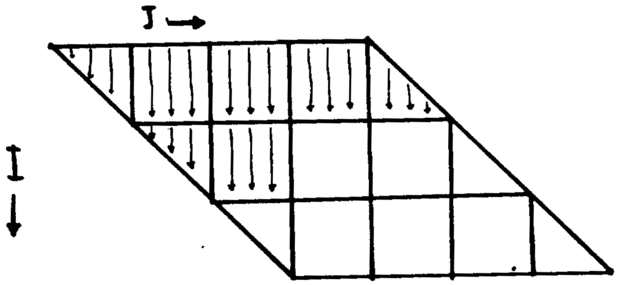

Figure 10a.

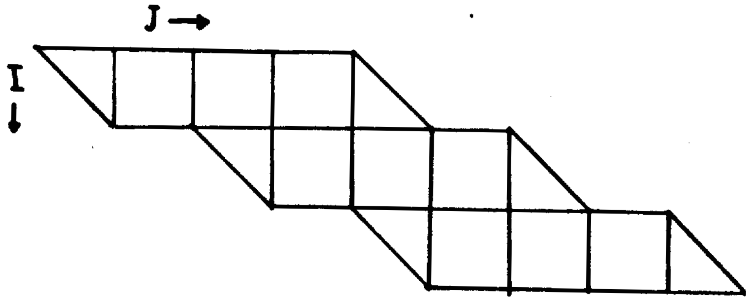

Figure 10b.

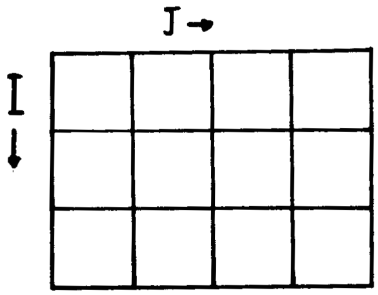

Figure 11a.

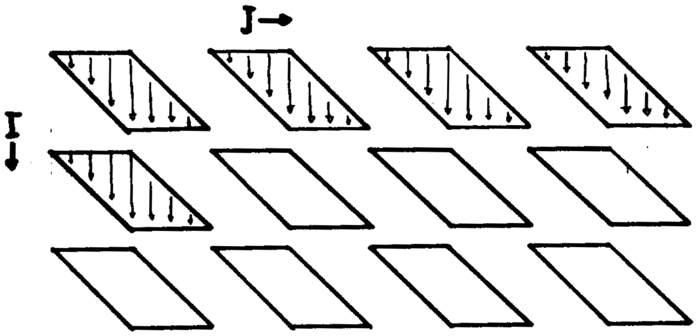

Figure 11b.

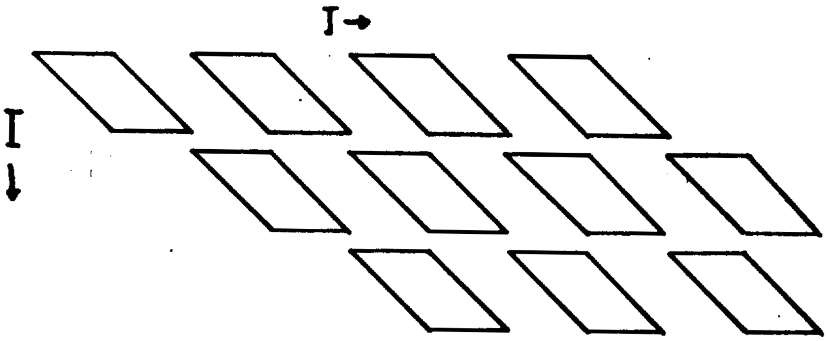

Figure 11c.

References

[AbuS78]
Walid Abdul-Karim Abu-Sufah, *Improving the Performance of Virtual Memory Computers,* Ph.D. Thesis, Dept. of Comp. Sci. Rpt. No. 78-945, Univ. of Illinois, Urbana, IL, Nov., 1978; available as document 79-15307 from University Microfilms, Ann Arbor, MI.

[AbKL81]
W. A. Abu-Sufah, D. J. Kuck and D. H. Lawrie, "On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations," *IEEE Trans. on Computers,* Vol. C-30, No. 5, pp. 341-356, May, 1981.

[AlKe84]
John R. Allen and Ken Kennedy, "Automatic Loop Interchange," *Proc. of the ACM SIGPLAN '84 Symposium on Compiler Construction,* Montreal, Canada, June 17-22, 1984, SIGPLAN Notices Vol. 19, No. 6, pp. 233-246, June, 1984.

[Bane88]
Utpal Banerjee, *Dependence Analyss for Supercomputing,* Kluwer Academic Publishers, Norwell, MA, 1988.

[IrTr88]
R. Irigoin and R. Triolet, "Supernode Partitioning," *Conf. Record of the 15th Annual ACM Symp. on Principles of Programming Languages,* pp. 319-329, Jan. 13-15, San Diego, CA, ACM Press, New York, 1988.

[Lamp74]
Leslie Lamport, "The Parallel Execution of DO Loops," *Comm. of the ACM,* Vol. 17, No. 2, pp. 83-93, Feb., 1974.

[Love77]
D. Loveman, "Program Improvement by Source-to-Source Transformation," *J. of the ACM,* Vol. 20, No. 1, pp. 121-145, Jan. 1977.

[Mura71]
Yoichi Muraoka, *Parallelism Exposure and Exploitation in Programs,* Ph.D. Thesis, Dept. of Comp. Sci. Rpt. No. 71-424, Univ. of Illinois, Urbana, IL, Feb., 1971.

[Wolf86]
Michael Wolfe, "Loop Skewing: The Wavefront Method Revisited," *Int'l Journal of Parallel Programming,* Vol. 15, No. 4, pp. 279-294, Aug. 1986.

[WoBa87]
Michael Wolfe and Utpal Banerjee, "Data Dependence and Its Application to Parallel Processing," *Int'l Journal of Parallel Programming,* Vol. 16, No. 2, pp. 137-177, April, 1987.

[Wolf87]
Michael Wolfe, "Iteration Space Tiling for Memory Hierarchies," *Proc. of the 3rd SIAM Conf. on Parallel Processing for Scientific Computing,* Garry Rodrigue (ed), Society for Industrial and Applied Mathematics, Philadelphia, PA, pp. 357-361, 1987.

[Wolf89]
Michael Wolfe, *Optimizing Supercompilers for Supercomputers,* MIT Press, Boston, 1989.