ECE 1754

Survey of Loop Transformation Techniques

Eric LaForest

March 19, 2010

Contents

1 Introduction

In this paper I present a survey of basic loop transformation techniques. These are general transformations, mostly at the source level, that do not stem from a common theoretical background except for the lexicographic order of the loop dependence vectors. Thus, I present them in a catalogue format grouped by general transformation types. The group types and examples are mostly taken from Sections 6 and 7 of Bacon, Graham, and Sharp's "*Compiler Transformations for High-Performance Computing*" [1], with my own explanations added. The examples are copied mostly verbatim because I could not improve upon their conciseness.

2 Data-Flow-Based Loop Transformations

2.1 Loop-Based Strength Reduction

Strength reduction replaces an expensive operation, usually in time, with a less expensive one. The canonical example is the replacement of a multiplication by 2^n with a $log_2(n)$ bit shift. In loops, the possibility of strength reduction manifests itself often when calculating memory location from array indices.

For example, the following code performs a constant multiplication with `i` every iteration. This operation can be quite expensive on a machine without a hardware multiplier. Even with one, a multiplication may take many more cycles than an addition.

```
do i = 1, n
 a[i] = a[i] + c*i
end do
```

The multiplication may be replaced by repeatedly adding the constant value `c` to an accumulator `T`, iteratively recreating the multiplication.

```
T = c
do i = 1, n
 a[i] = a[i] + T
 T = T + c
end do
```

This method is applicable to operations involving a constant which have a serial decomposition, such as exponentiation, or with a cheaper arithmetic equivalent, such as multiplication with a reciprocal instead of division (barring numerical precision concerns).

2.2 Induction Variable Elimination

The main induction variable in a loop is frequently used to calculate both memory addresses and loop exit conditions, as in the following example:

```
for(i = 0; i < n; i++){
 a[i] = a[i] + c;
}
```

Given a known final value of the main induction variable and the linear relations to derived values, the loop bounds can be expressed as a comparison test to the final derived values with only a single update calculation in the body of the loop,

completely eliminating the main induction variable and its calculations. The following code demonstrates this conversion on the previous example, eliminating index calculations:

```
A = &a;
T = &a + n;
while(A < T){
 *A = *A + c;
 A++;
}
```

2.3 Loop-Invariant Code Motion

In the following example, the index of `c[i]` does not change for the entire duration of the inner loop, despite being recalculated at every iteration, and is thus loop-invariant in the the inner loop.

```
do i = 1, n
 do j = 1, n
  a[i,j] = b[j,i] + c[i]
 end do
end do
```

The code may thus be moved to just before the loop, calculated once, its result stored in a register and reused throughout the inner loop.

```
do i = i, n
 T = C[i]
 do j = 1, n
  a[i,j] = b[j,i] + T
 end do
end do
```

Note that the moved code may need to be protected by a guard so as to execute only if the loop using its result will execute also, to preserve the meaning of any exceptions that may be raised. In this example, the transformation is identical to Scalar Replacement (Section 6.4).

2.4 Loop Unswitching

The following example shows a loop with a conditional execution path in its body. Having to perform a test and jump inside every iteration reduces the performance of the loop as it prevents the CPU, barring sophisticated mechanisms such as trace caches or speculative branching, from efficiently executing the body of the loop in a pipeline. The conditional also inhibits `do all` parallelization of the loop since any conditional statement must execute in order after the test.

```
do i = 2, n
 a[i] = a[i] + c
 if (x < 7) then
  b[i] = a[i] * c[i]
 else
  b[i] = a[i-1] * b[i-1]
 end if
end do
```

Similarly to Loop-Invariant Code Motion, if the loop-invariant expression is a conditional, then it can be moved to the outside of the loop, with each possible execution path replicated as independent loops in each branch. This multiplies the total code size, but reduces the running set of each possible branch, can expose parallelism in some of them, plays well with CPU pipelining, and eliminates the

repeated branch test calculations. Note that a guard may also be necessary to avoid branching to a loop that would never execute over a given range.

```
if (n > 2) then
 if (x < 7) then
  do all i = 2, n
   a[i] = a[i] + c
   b[i] = a[i] * c[i]
  end do
 else
  do i = 2, n
   a[i] = a[i] + c
   b[i] = a[i-1] * b[i-1]
  end do
 end if
end if
```

3 Loop Reordering

These transformations change the relative ordering and/or alignment of nested loops in order to expose parallelism by altering dependencies or to improve the locality of code to better fit the memory hierarchy.

3.1 Loop Interchange

Loop interchange simply exchanges the position of two loops in a loop nest. One of the main uses is to improve the behaviour of accesses to an array. For example, given a column-major storage order[1], the following code accesses `a[]` with a

[1] Visually speaking, consecutive memory locations are stored in adjacent columns of a matrix, along rows, from top left to bottom right as in English reading order. The cache lines span consecutive sections of the rows. The array indices are arranged as `[column, row]`.

stride of `n`. This may interact very poorly with the cache, especially if the stride is larger than the length of a cache line or is a multiple of a power of two, causing collisions in set-associative caches.

```
do i = 1, n
 do j = 1, n
  b[i] = b[i] + a[i,j]
 end do
end do
```

Interchanging the the loops alters the access pattern to be along consecutive memory locations of `a[]`, greatly increasing the effectiveness of the cache.

```
do j = 1, n
 do i = 1, n
  b[i] = b[i] + a[i,j]
 end do
end do
```

However, loop interchange is only legal if the dependence vector of the loop nest remains lexicographically positive after the interchange, which alters the order of dependencies to match the new loop order. For example, the following loop nest cannot be interchanged since its dependency vector is $(1, -1)$. The interchanged loops would end up using future, uncomputed values in the array.

```
do i = 2, n
 do j = 1, n-1
  a[i,j] = a[i-1,j+1]
 end do
end do
```

Similarly, loop interchange can be used to control the granularity of the work in nested loops. For example, by moving a parallel loop outwards, the necessarily serial work is moved towards the inner loop, increasing the amount of work done per fork-join operation.

3.2 Loop Skewing

Loop skewing does exactly what it says: it skews the execution of an inner loop relative to an outer one. This is useful if the inner loop has a dependence on the outer loop which prevents it from running in parallel. For example, the following code has a dependency vector of $\{(1,0),(0,1)\}$[2]. Neither loop can be parallelized since they each carry a dependency. Simply interchanging the loops would merely interchange the indices holding the dependencies, accomplishing nothing.

```
do i = 2, n-1
 do j = 2, m-1
  a[i,j] =
  (a[a-1,j] + a[i,j-1] + a[i+1,j] + a[i,j+1]) / 4
 end do
end do
```

Loop skewing is implemented by adding the index of the outer loop, times some skewing factor f, to the bounds of the inner loop and subtracting the same value from all the uses of the inner loop index. The subtraction keeps the indices within the new loop bounds, preserving the correctness of the program. The effect on the

[2] Nested dependencies begin with the induction variable nesting order, $\{(i),(j)\}$, and nest again in the same order for the loop dependencies to each variable: $\{(outer,inner),(outer,inner)\}$.

inner loop iterations is to shift their position in the array forwards by f relative to the current outer loop, increasing the dependency distance to the outer loop in the same manner. In other words, given a dependency vector (a, b), skewing transforms it to $(a, fa + b)$. Since this transformation preserves the lexicographic order of the dependencies[3], it is always legal. Applying a skew factor of one to the above inner loop yields the following code:

```
do i = 2, n-1
 do j = 2+i, m-1+i
  a[i,j-i] =
  (a[a-1,j-i] + a[i,j-1-i] + a[i+1,j-i] + a[i,j+1-i]) / 4
 end do
end do
```

This new code executes in the same manner, but with dependencies of $\{(1, 1), (0, 1)\}$. Both loops still carry a dependency. However, interchanging the loops at this point yields a dependence vector $\{(1, 0), (1, 1)\}$, as shown in the following code:

```
do j = 4, m+n-2
 do i = max(2, j-m+1), min(n-1, j-2)
  a[i,j-i] =
  (a[a-1,j-i] + a[i,j-1-i] + a[i+1,j-i] + a[i,j+1-i]) / 4
 end do
end do
```

The inner loop can now be parallelized since it has now no loop-carried dependency on `j`, and the dependency to `i` is carried by the outer loop. Note that interchanging skewed loop bounds is no longer straightforward: each loop must take into account the upper and lower bounds of the other loop.

[3] If it was positive before, it will be positive after.

3.3 Loop Reversal

Loop reversal simply changes the direction of the iteration, inverting the sign of its position in the dependence vector. It is a legal transformation if the resulting dependence vector remains lexicographically positive. Although trivial, it is a useful optimization since it may enable others such as loop interchange and can reduce the loop exit condition to a single branch-not-equal-to-zero instruction. For example, the following code cannot be interchanged or have its inner loop parallelized because of $(1, -1)$ dependencies.

```
do i = 1, n
 do j = 1, n
 a[i,j] = a[i-1,j+1] + 1
 end do
end do
```

Reversing the inner loop yields $(1, 1)$ dependencies. The loops can now be interchanged and/or the inner loop made parallel.

```
do i = 1, n
 do j = n, 1, -1
  a[i,j] = a[i-1,j+1] + 1
 end do
end do
```

3.4 Strip Mining

Strip mining is used to control the granularity of the inner loop, usually to adjust to a vector length or to limit the working set at a given point. Strip mining itself does not alter dependencies, so it is always legal. Let's illustrate the process with the following simple loop:

```
do i = 1, n
 a[i] = a[i] + c
end do
```

Let's assume that strips of 64 array elements are desirable. The first new line computes the multiple of 64 closest to `n`. The outer loop iterates towards this multiple in increments of 64. A new inner loop performs the original loop on the current strip. Finally, a fixup loop may be required if `n` is not a multiple of 64. Note that this inner loop could also be converted into a `do all` loop.

```
TN = (n/64)*64
do i = 1, TN, 64
 do j = 1, 64
  a[i+j-1] = a[i+j-1] + c
 end do
end do
do i = TN+1, n
 a[i] = a[i] + c
end do
```

3.5 Cycle Shrinking

If a dependency prevent the parallelization of an inner loop, and the dependency distance is positive and constant, then strip mining the loop into an inner loop whose strip length is equal to the dependence distance can yield some fine-grained parallelism. For example, the following loop has flow dependencies of distance `k`.

```
do i = 1, n
 a[i+k] = b[i]
 b[i+k] = a[i] + c[i]
end do
```

All the steps of a strip of length `k` can execute in parallel since they will complete before the next strip begins, where the dependencies terminate.

```
do TI = 1, n, k
 do all i = TI, TI+k-1
  a[i+k] = b[i]
  b[i+k] = a[i] + c[i]
 end do all
end do
```

3.6 Loop Tiling

Strip mining is useful if the loop body performs calculations linearly through an array. But if the calculations do both row and column-wise accesses, strip mining will be of little benefit as the row-wise accesses orthogonal to the strip will require a different cache line each time. Eventually, this will conflict with the cache line in use by the strip and performance will suffer.

Loop tiling performs strip mining in multiple array dimensions, constraining the working set to fit within both the cache line length (for column-wise iteration) and the number of cache lines (for row-wise iteration), dividing the array into cache-size tiles. The following code shows this with a loop transpose example:

```
do i = 1, n
 do j = 1, n
  a[i,j] = b[j,i]
 end do
end do
```

By strip mining both loops at once the working set is limited to the cache size: 64 lines of 64 elements in this case. The transformation is legal if both loop can be interchanged, since the original outermost loop `i` is now the inner loop of another version of the inner loop `j`, as `TJ`.

```
do TI = 1, n, 64
 do TJ = 1, n, 64
  do i = TI, min(TI+63, n)
   do j = TJ, min(TJ+63, n)
    a[i,j] = b[j,i]
   end do
  end do
 end do
end do
```

4 Loop Restructuring

These transformations alter the form of the loop, but not the order or types of calculations. Thus, these transformations are virtually always legal (see Loop Fission (4.8), Fusion (4.9), and Node Splitting (4.10) for the exceptions).

4.1 Loop Unrolling

Loop unrolling is a simple transformation that instantiates f consecutive instances of loop iterations in the body and increases the loop step by the same factor. This divides the loop overhead by f, and also promotes reuse since identical and consecutive values appear multiple times in the unrolled loop body.

```
do i = 2, n-1
 a[i] = a[i] + a[i-1] * a[i+1]
end do
```

The following loop shows an unrolling of factor 2. The upper loop bound must be altered to stay in its original range and a small fixup conditional statement or loop may be needed afterwards to finish the last $n \, mod \, f$ statements.

```
do i = 1, n-2, 2
 a[i] = a[i] + a[i-1] * a[i+1]
 a[i+1] = a[i+1] + a[i] * a[i+2]
end do
if (mod(n-2,2) = 1) then
 a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

4.2 Software Pipelining

Software pipelining is similar to loop unrolling. The stages of the body of a loop are broken down into consecutive steps and scheduled on multiple execution units such that the next statement begins execution while the current one completes.

```
do i = 1, n
 a[i] = a[i] + c
end do
```

For example, given a machine with separate load/store and arithmetic units, the preceding loop can be unrolled once, and the steps of each statement interleaved together as described in the adjacent comments.

```
do i = 1, n, 2       // Load/Store   : Arithmetic
 a[i] = a[i] + c     // load a[i]    : i = i + 1
 a[i+1] = a[i+1] + c // load a[i+1]  : a[i] + c
end do               // store a[i]   : a[i+1] + c
                     // store a[i+1] : i = i + 1
                     // jmp if i < n :
```

4.3 Loop Coalescing

Loop coalescing combines nested loops into a single loop, reducing to a single induction variable, and computing the indices from that variable. This reduces loop overhead (assuming the index calculations can be simplified) and can allow for better load balancing. For example, the following loop, although highly parallel, would load-balance poorly on P processors if `n` and `m` were slightly larger than P, leaving one processor to finish up a set of iterations and possibly doubling execution time.

```
do all i = 1, n
 do all j = 1, m
  a[i,j] = a[i,j] + c
 end do all
end do all
```

However, by using a single induction variable `T` which spans the entire range of the array in a linear manner, the row and column indices can be derived from `T` and placed in a single loop. The iterations of this loop can now be evenly divided amongst a number of chunks that is a multiple of P, yielding a perfect load balance[4].

```
do all T = 1, n*m
 i = ((T-1) / m) * m + 1
 j = mod(T-1, m) + 1
 a[i,j] = a[i,j] + c
end do all
```

[4] Memory hierarchy effects notwithstanding.

4.4 Loop Collapsing

Loop collapsing is similar to loop coalescing, but makes use of cases when the stride is constant. The difference is that collapsing reduces the dimensions of a loop to eliminate the overhead of calculating multiple indices for each array location. The following code shows this transformation for the previous loop coalescing example: The two-dimensional array a is cast as a linear array TA of the same size, requiring only one index.

```
real TA[n*m]
equivalence(TA,a)
do all T = 1, n*m
 TA[T] = TA[T] + c
end do all
```

4.5 Loop Peeling

Loop peeling extracts into a separate loop one or more iterations from the beginning or end of the loop iteration space. This transformation can eliminate dependencies and adjust loop bounds for later loop fusion. For example, the following loop cannot be made parallel since all iterations depend on the first.

```
do i = 2, n
 b[i] = b[i] + b[2]
end do
```

By peeling the first iteration out so as to calculate it before all the others, and adjusting the bounds of the loop, the dependency is eliminated. Since the peeled

iteration is in the same order as before, this transformation is always legal (see 4.8, Loop Fission).

```
if (2 <= n) then
 b[2] = b[2] + b[2]
end if
do all i = 3, n
 b[i] = b[i] + b[2]
end do all
```

4.6 Loop Normalization

This is a simple transformation which alters the loop bounds (and body to match) to iterate over `1` to `n` with a stride of one. This simplifies many analyses and enables other optimizations such as fusion.

```
do i = 2, n+1
 b[i] = a[i-1] * b[i]
end do
```

In this simple example, the code above had its bounds moved back by one, normalizing the loop, and its indices moved forward by the same amount to compensate. Multiple normalized loops can be trivially fused, assuming no backward dependencies are introduced.

```
do i = 1, n
 b[i+1] = a[i] * b[i+1]
end do
```

4.7 Loop Spreading

The following two loops cannot be combined with loop fusion due to unequal loop bounds and an introduced $S_2\delta_2^a S_1$ dependency on `a[]`.

```
    do i = 1, n/2
S1:  a[i+1] = a[i+1] + a[i]
    end do
    do i = 1, n-3
S2:  b[i+1] = b[i+1] +b[i] * a[i+3]
    end do
```

However, some instruction parallelism can be exposed by loop spreading, a limited for of loop fusion. The key is to delay the execution of S_2 by its dependence distance to S_1, plus one, to separate the dependent accesses within an iteration. To compensate, the indices of S_2 must be moved back by the same amount. This modified version of S_2 can then be run concurrently with each iteration of S_1, as denoted by the `COBEGIN` and `COEND` statements.

```
do i = 1, n/2
 COBEGIN
  a[i+1] = a[i+1] + a[i]
  if(i > 3) then
   b[i-2] = b[i-2] + b[i-3] * a[i]
  end if
 COEND
end do
```

In this case, the upper loop bound of the first loop is less than that of the second, so the remainder of the work of S_2 must be done serially, after shifting the lower

loop bound of the second loop to the end of the first, minus the original delay value since we're using the unmodified S_2.

```
do i = (n/2)-3, n-3
 b[i+1] = b[i+1] + b[i] * a[i+3]
end do
```

Had the first loop a higher upper bound than the second, then the `(i > 3)` conditional would have been augmented with a guard to the upper loop bound of the second loop and the second loop would have been omitted entirely.

4.8 Loop Fission[5]

Loop fission is a simple transformation that is very useful for simplifying loop bodies, often reducing the memory and registers required during the execution of the loop. It can also remove simple flow dependencies between loop statements, so long as the dependent statements execute in the same relative order afterwards. The following code shows this by having a distance zero flow dependence from the first to second statement.

```
do i = 1, n
 a[i] = a[i] + c
 x[i+1] = x[i]*7 + x[i+1] + a[i]
end do
```

The first statement can be moved to its own copy of the loop, where it can execute in parallel. The writes of `a[]` must still occur before the reads in the second

[5] I moved Fission and Fusion from Loop Reordering transformations to Loop Restructuring since they really do the latter and don't alter execution order.

statement, so the new loop must precede the second one. Consequently, if circular dependencies exist between two statements, they cannot be separated by fission (but see 4.10, Node Splitting, for a workaround).

```
do all i = 1, n
 a[i] = a[i] + c
end do all
do i = 1, n
 x[i+1] = x[i]*7 + x[i+1] + a[i]
end do
```

This transformation would also not be possible if the dependence distance was non-zero. For example, if the first statement was `a[i+1] = a[i] + c`, then the second statement, after loop fission, would not be able to access the original value of `a[i]`.

4.9 Loop Fusion

Loop fusion is simply the inverse of fission. It's benefits can be similar to loop fission, depending on the machine, and it always reduces the loop overhead. To be legal, both loops to be fused must have the same loop bounds, and the statements in the fused loop must not exhibit any backward dependencies (e.g.: $S_2\delta^a S_1$).

4.10 Node Splitting[6]

Cyclic dependencies in a loop prevent loop fission (Section 4.8). For example, the following loop has a flow dependence $S_1\delta_0^f S_2$ and an antidependence $S_2\delta_1^a S_1$ both

[6] This example was adapted from Padua & Wolfe [2].

on `a[]`, forming a cycle.

```
    do i = 1, n
S1:  a[i] = b[i] + c[i]
S2:  d[i] = (a[i] + a[i+1]) / 2
    end do
```

However, antidependencies can be eliminated by copying to a new name. Thus we can create a shifted copy of the original contents of `a[]` as `T[]` to replace the `a[i+1]` reference in S_2. This changes the dependencies to $S_3\delta_1^a S_1$, $S_1\delta_0^f S_2$, and $S_3\delta_0^f S_2$, breaking the cycle.

```
    do i = 1, n
S3:  T[i] = a[i+1]
S1:  a[i] = b[i] + c[i]
S2:  d[i] = (a[i] + T[i]) / 2
    end do
```

Without a cycle and since S_2 has no loop-carried dependencies to itself, it can be fissioned-off into its own loop (placed after the first loop to honour $S_1\delta_0^f S_2$) and made parallel. The same could be done to S_1 and S_3 if desired.

```
    do i = 1, n
S3:  T[i] = a[i+1]
S1:  a[i] = b[i] + c[i]
    end do
    do all i = 1, n
S2:  d[i] = (a[i] + T[i]) / 2
    end do
```

5 Loop Replacement

These are more radical transformations that rely on recognizing some common loop purposes and completely altering the loop to better implement them.

5.1 Reduction Recognition

A reduction is a common operation that reduces an array or list of values to a single scalar. Common examples are finding the min, max, or sum of a number of items. By default, reduction cannot be made fully parallel because of loop-carried dependence on the scalar.

```
do i = 1, n
 s = s + a[i]
end do
```

However, reductions can be parallelized into a binary tree of operations, or some partially collapsed version thereof. For example, if the machine has some vector or SIMD hardware it can be used to cluster the reduction steps. The following code adds a 64-entry array and initially reduces `a[]` in 64-entry vector chunks. The individual elements of the vector are then summed serially. This reduces the number of operations from `n` to `(n/64)+64`.

```
real TS[64]
TS[1:64] = 0.0
do TI = 1, n, 64
 TS[1:64] = TS[1:64] + a[TI: TI+63]
end do
```

```
do TI = 1, 64
 s = s + TS[TI]
end do
```

This transformation has the caveat that it is only guaranteed correct if the reduction operation is fully associative. Otherwise, the transformed loop may yield different results which may not match the programmer's intent.

5.2 Array Statement Scalarization

If a machine has no vector hardware, what can a compiler do with array operations expressed as vectors?

```
a[2:n-1] = a[2:n-1] + a[1:n-2]
```

The apparent solution is to simply iterate over the vector range. However, this is incorrect as it breaks the simultaneity of the vector assignments on the left-hand side. Each iteration accesses the previous array element, which was incorrectly modified by the previous iteration.

```
do i = 2, n-1
 a[i] = a[i] + a[i-1]
end do
```

To avoid a destructive partial update of the array, a temporary must be used which is then copied back after all the iterations have run.

```
do i = 2, n-1
 T[i] = a[i] + a[i-1]
end do
do i = 2, n-1
 a[i] = T[i]
end do
```

However, a simpler solution is to reverse the loop direction, transforming the dependence on the previous element to one on the next element, still unmodified. This solution only works if the dependencies are all in the same direction (see 3.3, Loop Reversal).

```
do i = n-1, 2, -1
 a[i] = a[i] + a[a-1]
end do
```

6 Memory Access

While some previously described transformations improved memory access behaviour by altering the iterations of a loop, these transformations alter the memory layout of the data in order to better work with the memory hierarchy.

6.1 Array Padding

If a machine routes consecutive memory accesses to different memory banks in order to improve throughput, then a stride that is a multiple of the number of banks can defeat this improvement, as each successive request will be routed to the same bank, serializing the memory accesses. The same phenomenon can occur with cache and TLB sets.

For example, given a machine with eight memory banks and assuming column-major array order, the following code will create an array where each row will end up stored in a single bank. Thus the following code will repeatedly access the first bank only.

```
real a[8,512]
do i = 1, 512
 a[1,i] = a[1,i] + c
end do
```

The simple solution, at the cost of wasted space, is to introduce one or more dummy columns in the array such that the new number of columns has only 1 as common divisor with the number of banks. This ensures that each row element ends up staggered into consecutive banks.

```
real a[9,512]
do i = 1, 512
 a[1,i] = a[1,i] + c
end do
```

6.2 Scalar Expansion

Scalars introduce an $S_2 \delta^a S_1$ dependence in loops. They can manifest as compiler-generated temporaries.

```
do i = 1, n
 c = b[i]
 a[i] = a[i] + c
end do
```

This dependence can be eliminated by expanding the scalar into an array, effectively giving each iteration a private copy and enabling `do all` parallelism. The size of the new array can be controlled by first strip-mining the loop.

```
real T[n]
do all i = 1, n
 T[i] = b[i]
 a[i] = a[i] + T[i]
end do all
```

6.3 Array Contraction

In the following loop nest, the value of index `i` for `T[]` is constant throughout the inner loop since the outer loop is not parallel (no two values of `i` ever get used at once to address `T`). There are also no loop-carried dependencies on i, nor is `T[]`

used after the inner loop before i changes again. The net effect of this is that a given location `T[i,j]` is addressed only once over the loop nest.

```
real T[n,n]
do i = 1, n
 do all j = 1, n
  T[i,j] = a[i,j]*3
  b[i,j[ = T[i,j] + b[i,j]/T[i,j]
 end do all
end do
```

Since the elements of `T[]` along the `i` dimension are never reused, they can be simply eliminated. This will reduce storage space and improve the locality of `T[]`.

```
real T[n]
do i = 1, n
 do all j = 1, n
  T[j] = a[i,j]*3
  b[i,j[ = T[j] + b[i,j]/T[j]
 end do all
end do
```

6.4 Scalar Replacement

Another simple transformation deals with the frequent reuse of a fixed array element in an inner loop. In the following code, `total[i]` is repeatedly read and written in the inner loop.

```
do i = 1, n
 do j = 1, n
  total[i] = total[i] + a[i,j]
 end do
end do
```

Instead, the fixed array element can be assigned to a scalar before the inner loop, and if modified, stored back into the original element afterwards. This saves index calculations and reduces the total number of accesses to the array element, reducing memory traffic.

```
do i = 1, n
 T = total[i]
 do j = 1, n
  T = T + a[i,j]
 end do
 total[i] = T
end do
```

References

[1] BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. Compiler transformations for high-performance computing. *ACM Comput. Surv. 26*, 4 (1994), 345–420.

[2] PADUA, D. A., AND WOLFE, M. J. Advanced compiler optimizations for supercomputers. *Commun. ACM 29*, 12 (1986), 1184–1201.