Improving the Ratio of Memory Operations to Floating-Point Operations in Loops

Steve Carr
Ken Kennedy

CRPC-TR92284
November 1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Improving the Ratio of Memory Operations to Floating-Point Operations in Loops

STEVE CARR
Michigan Technological University
KEN KENNEDY
Rice University

Over the past decade, microprocessor design strategies have focused on increasing the computational power on a single chip. Because computations often require more data from cache per floating-point operation than a machine can deliver and because operations are pipelined, idle computational cycles are common when scientific applications are run on modern microprocessors. To overcome these bottlenecks, programmers have learned to use a coding style that ensures a better balance between memory references and floating-point operations. In our view, this is a step in the wrong direction because it makes programs more machine-specific. A programmer should not be required to write a new program version for each new machine; instead, the task of specializing a program to a target machine should be left to the compiler.

But is our view practical? Can a sophisticated optimizing compiler obviate the need for the myriad of programming tricks that have found their way into practice to improve performance of memory hierarchy? In this paper we attempt to answer that question. To do so, we develop and evaluate techniques that automatically restructure program loops to achieve high performance on specific target architectures. These methods attempt to balance computation and memory accesses and seek to eliminate or reduce pipeline interlock. To do this, they statically estimate the performance of each loop in a particular program and use these estimates to determine whether to apply various loop transformations.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors – compilers, optimization

General Terms: Languages

Additional Keywords and Phrases: loop balance, machine balance

1 INTRODUCTION

Over the past decade, the computer industry has realized dramatic improvements in the power of microprocessors. These gains have been achieved both by cycle-time improvements and by architectural innovations like multiple instruction issue and pipelined functional units. As a result of these architecture improvements, today's microprocessors can perform many more operations per machine cycle than their predecessors.

But with these gains have come problems. Because the latency and bandwidth of memory systems have not kept pace with processor speed, computations are often delayed waiting for data from memory. As a result, processors see idle computational cycles and empty pipeline stages more frequently. In fact, memory and pipeline delays have become the principal performance bottleneck for high-performance microprocessors.

To overcome these performance problems, programmers have learned to employ a coding a style that achieves a good balance between memory references and floating-point operations. The goal of the coding methodology is to eliminate pipeline interlock and to match the ratio of memory operations to floating-point operations in program loops to the optimum such ratio that the target machine can deliver. This is usually done by unrolling outer loops and merging the resulting copies of inner loops. Unfortunately, this coding style almost always leads to very ugly code. In addition, this style is error-prone, tedious and time consuming.

Research supported by NSF Grant CCR-9120008 and by ARPA through ONR Grant N00014-91-J-1989.
Authors' addresses: Steve Carr, Department of Computer Science, Michigan Technological University, Houghton, MI 49931. Ken Kennedy, CITI, Rice University, Houston, TX 77251–1892

The thesis of this paper is that most of this type of hand optimization is unnecessary, because it can, and should, be performed automatically in a compiler. Having the compiler optimize loops is better for the programmer in two ways. First, it avoids a lot of work that is not necessary for the expression of a correct solution. Second, it makes it possible for the same program to be used on a variety of machine architectures.

To establish this thesis, the paper presents new techniques to automatically restructure program loops for high performance on specific target architectures. A method to statically estimate the performance of a given loop on a particular architecture is presented and used in an algorithm to determine how to transform a loop to best improve its performance. This approach extends previous work by showing how to automatically decide when and how to transform a loop . The transformation system is tailored to a specific target machine based on a few parameters of that machine, such as number of registers, floating-point operations per cycle and loads per cycle. Thus, the transformations are machine-independent in the sense that they can be retargeted to new processors by changing parameters. The method described in this paper has been implemented in an experimental compiler and experments to determine its effectiveness have been carried out. The paper concludes with a report on the performance improvements observed in these experiments.

2 BACKGROUND

This section lays the foundation for the application of reordering transformations that improve the memory performance of programs. It begins with a measure of machine and loop performance that will be used in the application of the transformations described in the next section. Then, a special form of dependence graph used by the transformation system to model memory usage is presented.

2.1 Performance Model

It is assumed that the target architecture is pipelined and allows asynchronous execution of memory accesses and floating-point operations (e.g. Intel i860 or IBM RS/6000). It is also assumed that the target machine has a typical optimizing compiler — one that performs scalar optimizations only. In particular, the compiler performs strength reduction, allocates registers globally (via a typical coloring scheme) and schedules both the instruction and the arithmetic pipelines. This makes it possible for the transformation system to restructure the loop nests, while leaving the details of optimizing the loop code to the compiler for the target machine. To measure the performance of program loops given the above assumptions, we use the notion of *balance* defined by Callahan, *et al.* [6].

2.1.1 Machine Balance

A computer is balanced when it can operate in a steady state manner with both memory accesses and floating-point operations being performed at peak speed. To quantify this relationship, β_M is defined as the rate at which data can be fetched from memory, M_M, compared to the rate at which floating-point operations can be performed, F_M:

$$\beta_M = \frac{\text{max words/cycle} = M_M}{\text{max flops/cycle} = F_M}$$

The values of M_M and F_M represent peak performance where the size of a word is the same as the precision of the floating-point operations. Every machine has at least one intrinsic β_M. For example, on the IBM RS/6000 $\beta_M = 1$.

2.1.2 Loop Balance

Just as machines have balance ratios, so do loops. Balance for a specific loop can be defined as

$$\beta_L = \frac{\text{number of memory references} = M}{\text{number of flops} = F}. \tag{1}$$

It is assumed that references to array variables are actually references to memory, while references to scalar variables involve only registers. Memory references are assigned a uniform cost under the assumption that loop interchange, tiling and cache copying can be performed to attain cache locality [16, 20, 24].

Comparing β_M to β_L gives a measure of the performance of a loop running on a particular architecture. If $\beta_L > \beta_M$, then the loop needs data at a higher rate than the machine can provide and idle computational cycles will exist. Such a loop is said to be *memory bound* and its performance can be improved by lowering β_L. If $\beta_L < \beta_M$, then data cannot be processed as fast as it is supplied to the processor and idle memory cycles will exist. Such a loop is said to be *compute bound.* Compute-bound loops run at the peak floating-point rate of a machine and need not be further balanced. This is because floating-point operations usually cannot be removed and arbitrarily increasing the number of memory operations is unlikely to be beneficial. Finally, if $\beta_L = \beta_M$, the loop is balanced for the target machine.

2.1.3 Pipeline Interlock

Previous work in loop balance has included a measure of pipeline interlock in the computation of balance. The complete formulation of balance is as follows

$$\beta_L = \frac{M}{F + \mathit{idle\ cycles}}. \tag{2}$$

In order for balance to more closely relate to performance, the idle cycles caused by pipeline interlock must be removed. Callahan, *et al.*, show how to use unroll-and-jam explicitly to remove idle cycles due to pipeline interlock using Equation 2 [6]. The new method described in Section 3 does not explicitly remove interlock. Instead, interlock is implicitly removed by unrolling to create balanced loops using Equation 1. It is assumed that unrolling to balance a loop will create enough copies of an inner-loop recurrence to remove all interlock. The experiments in Section 4 validate this assumption for short pipelines. However, if there is still interlock after unrolling, the Callahan, *et al.*, technique can be used to increase unroll amounts until the interlock is removed. Interlock is not ignored, but rather handled separately and only when necessary. Section 3.2.4 describes interlock removal in more detail.

2.2 Dependence Graph

A *dependence* exists between two references if there exists a control-flow path from the first reference to the second, and both references access the same memory location [17]. The dependence is

- a *true dependence* or *flow dependence* if the first reference writes to the location and the second reads from it,
- an *antidependence* if the first reference reads from the location and the second writes to it,
- an *output dependence* if both references write to the location, and
- an *input dependence* if both references read from the location.

If two references, v and w, are contained in n common loops, separate instances of the execution of the references can be described by an *iteration vector.* An iteration vector, denoted $\vec{i}$, is simply the values of

the loop control variables of the loops containing v and w. The set of iteration vectors corresponding to all iterations of the of the loop nest is called the ***iteration space***. Using iteration vectors, a ***distance vector***, $d = \langle d_1, d_2, \ldots, d_n \rangle$, can be defined for each dependence: if v accesses location Z on iteration $\vec{i_v}$ and w accesses location Z on iteration $\vec{i_w}$, the distance vector for this dependence is $\vec{i_w} - \vec{i_v}$. Under this definition, the kth component of the distance vector is equal to the number of iterations of the kth loop (numbered from outermost to innermost) between accesses to Z. The ***direction vector*** $D = \langle D_1, \ldots, D_n \rangle$ of a dependence is defined by the equation:

$$D_j = \begin{cases} < & \text{if } i_{v_j} < i_{w_j} \\ = & \text{if } i_{v_j} = i_{w_j} \\ > & \text{if } i_{v_j} > i_{w_j} \end{cases}$$

These symbols can be combined when more than one condition is true to give $\leq$ and $\geq$. If all direction vectors are possible for a dependence, then its direction is denoted as $*$. The elements are always displayed in order left to right, from the outermost to the innermost loop in the nest. As an example, consider Figure 1. The distance and direction vectors for the dependence between the definition and use of array A are $\langle 1, 0, -1 \rangle$ and $\langle <, =, > \rangle$, respectively. The direction vector for the dependence from C to itself is $\langle =, =, * \rangle$. There is no single distance vector for this dependence.

The loop associated with the outermost non-zero direction vector entry is said to be the *carrier* of the dependence. The ***threshold*** of a carried dependence is the value of the distance vector entry for the carrier. A minimum threshold equal to the minimum possible positive distance vector value is associated with those dependences having no single distance vector value in the outermost position. If all entries are 0 or $=$, the dependence is ***loop independent*** and its threshold is 0. Only those dependences that have a ***consistent threshold*** – that is, those dependences for which the threshold is constant throughout the execution of the loop – definitely carry memory reuse [6, 15]. For a dependence to have a consistent threshold, the location accessed by the dependence source on iteration i must be accessed by the sink on iteration $i + c$, where c does not vary with i. For the purposes of this paper, only those consistent dependences whose source and sink have only one induction variable in each subscript position are considered for memory reuse. Multiple-induction-variable subscripts are handled elsewhere [8].

3 METHOD

Conceptually, the transformation method is simple. For each loop nest, β_L is computed. Then, for each memory-bound loop, transformations are applied to try to make $\beta_L \leq \beta_M$. This section presents two transformations used to reduce β_L. The first transformation reduces the number of memory accesses in an innermost loop. The second introduces more floating-point operations into the innermost loop, keeping the loop-nest total constant, while reducing the overall memory requirements of the entire loop nest.

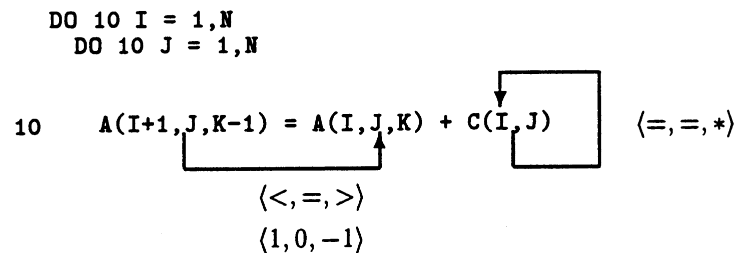

Figure 1 Example Dependence Graph

3.1 Scalar Replacement

The number of memory references in a loop can be lowered by replacing references to arrays with sequences of scalar variables. In the code shown below,

```
      DO 10 I = 2, N
10      A(I) = A(I-1) + B(I)
```

the value accessed by `A(I-1)` is defined on the previous iteration of the loop by `A(I)` on all but the first iteration. Using this knowledge, the flow of values between the references can be expressed with scalar temporaries as follows.

```
      T = A(1)
      DO 10 I = 2, N
        T = T + B(I)
10      A(I) = T
```

Since the values held in scalar quantities will probably be in registers, the load of `A(I-1)` has been removed, resulting in a reduction in the memory cycle requirements of the loop [11]. This transformation is called *scalar replacement* and is described in detail elsewhere [5, 6, 10, 13, 14, 21, 22]. Essentially, any consistent true or input dependence with 0's in the outer $n-1$ distance vector entries is amenable to scalar replacement. Additionally, any array reference that is invariant with respect to the innermost loop may be removed by scalar replacement.

3.2 Unroll-And-Jam

Unroll-and-jam is a transformation that can be used in conjunction with scalar replacement to improve the performance of memory-bound loops [1, 2, 6]. The transformation unrolls an outer loop and then jams the resulting inner loops back together.[1] Using unroll-and-jam, more computation can be introduced into an innermost loop body without a proportional increase in memory references. For example, the loop:

```
      DO 10 I = 1, 2*M
        DO 10 J = 1, N
10        A(I) = A(I) + B(J)
```

after unrolling becomes:

```
      DO 10 I = 1, 2*M, 2
        DO 20 J = 1, N
20        A(I) = A(I) + B(J)
        DO 10 J = 1, N
10        A(I+1) = A(I+1) + B(J)
```

and after jamming becomes:

```
      DO 10 I = 1, 2*M, 2
        DO 10 J = 1, N
          A(I) = A(I) + B(J)
10        A(I+1) = A(I+1) + B(J)
```

[1]Note that in scalar replacement the innermost loop is unrolled to remove any copies needed to capture loop-carried reuse. Unroll-and-jam is separate from any inner-loop unrolling that might be done to improve scheduling and scalar replacement.

In the original loop, one floating-point operation and one memory reference, B(J), are left after scalar replacement, giving a balance of 1. After applying unroll-and-jam, two floating-point operations and only one memory reference exist in the loop, giving a balance of 0.5. On a machine that can perform twice as many floating-point operations as memory accesses per clock cycle, the second loop would perform better.

Although unroll-and-jam has been studied extensively, it has not been shown how to tailor unroll-and-jam to specific loops run on specific architectures. In the past, unroll amounts have been determined experimentally and specified with a compile-time parameter [5]. However, the best choice for unroll amounts varies between loops and architectures. Section 3.2.3 derives a method to chose unroll amounts automatically in order to balance program loops with respect to a specific target architecture. Unroll-and-jam is tailored to a specific machine based on a few parameters of the architecture, such as effective number of registers and machine balance. The result is a machine-independent transformation system in the sense that it can be retargeted to new processors by changing parameters.

The rest of this section begins with an overview of the safety conditions for unroll-and-jam. Then, a method for updating the dependence graph so that scalar replacement can take advantage of the new opportunities created by unroll-and-jam is presented. Finally, an automatic method to balance program loops with respect to a particular architecture is derived.

3.2.1 Safety

With a semantics that is only defined on correct programs (as in Fortran), unroll-and-jam is safe if the flow of values is not changed. Dependences can prevent unroll-and-jam or limit the amount of unrolling that can be done and still allow jamming to be safe. Consider the following loop.

```
      DO 10 I = 1, 2*M
         DO 10 J = 1,N
10          A(I,J) = A(I-1,J+1)
```

A true dependence with a distance vector of $\langle 1, -1 \rangle$ goes from `A(I,J)` to `A(I-1,J+1)`. Unrolling the outer loop once creates a loop-independent true dependence that becomes an antidependence in the reverse direction after loop jamming. This dependence prevents jamming because the order of the store to and load from a location would be reversed as shown in the unrolled code below.

```
      DO 10 I = 1, 2*M,2
         DO 10 J = 1,N
            A(I,J) = A(I-1,J+1)
10          A(I+1,J) = A(I,J+1)
```

Location `A(3,2)` is defined on iteration $\langle 3, 2 \rangle$ and read on iteration $\langle 4, 1 \rangle$ in the original loop. In the transformed loop, the location is read on iteration $\langle 3, 1 \rangle$ and defined on iteration $\langle 3, 2 \rangle$, reversing the access order and changing the semantics of the loop. The following theorem from Callahan, *et al.*, establishes the conditions for unroll-and-jam safety and is stated without proof [6].

Theorem 1 *Let e be a true, output or antidependence carried at level k with distance vector*

$$d = \langle 0, \ldots, 0, d_k, 0, \ldots, 0, d_j, \ldots \rangle,$$

where $d_j < 0$ and all components between the kth and j^{th} are 0. Then the loop at level k can be unrolled at most $d_k - 1$ times before a dependence is generated that prevents fusion of the inner $n - k$ loops.

Essentially, this theorem states that unrolling loop k more than $d_k - 1$ will introduce a dependence with a negative entry in the outermost position after fusion. Since negative thresholds are undefined, the dependence

direction must be reversed. Dependence direction reversal changes the order in which values are referenced. To allow unroll-and-jam in some situations, loop skewing can be used to remove negative distance entries in preventing dependences [25]. It is assumed that any skewing done by a transformation system is done before the safety test for unroll-and-jam is applied. Additionally, when non-DO statements appear at levels other than the innermost level or multiple loops appear at inner levels, loop distribution must be safe in order for unroll-and-jam to be safe. No recurrence involving data or control dependences can be violated [3, 18].

3.2.2 Dependence Copying

To allow scalar replacement to take advantage of the new opportunities for reuse created by unroll-and-jam, the dependence graph must be updated to reflect these changes. In this section, we describe a method to update dependences for the two most prominent classes of variables. Both classes have only one induction variable in any given subscript expression.

Variant References. Below is a method to compute the updated dependence graph after loop unrolling for consistent dependences that contain only one loop induction variable in each subscript position and are variant with respect to the unrolled loop [6].

If the ith loop in a nest L is unrolled by a factor of k, then the updated dependence graph $G' = (V', E')$ for L' can be computed from the dependence graph $G = (V, E)$ for L by the following rules:

1. For each $v \in V$, there are $k+1$ nodes $v_0, \ldots, v_k$ in V'. These correspond to the original reference and its k copies.
2. For each edge $e = \langle v, w \rangle \in E$, with distance vector $d(e)$, there are $k+1$ edges $e_0, \ldots, e_k$ where $e_m = \langle v_m, w_n \rangle$, v_m is the mth copy of v, w_n is the nth copy of w and

$$n = (m + d_i(e)) \bmod (k+1) \tag{3}$$

$$d_i(e_m) = \begin{cases} \lfloor \frac{d_i(e)}{k+1} \rfloor & \text{if } n \geq m \\ \lfloor \frac{d_i(e)}{k+1} \rfloor + 1 & \text{if } n < m \end{cases} \tag{4}$$

To extend dependence copying to unroll-and-jam, we assume that loops are unrolled in an an outermost-to innermost-loop ordering. If loop k is unrolled and a new distance vector, $d(e) = \langle d_1, d_2, \ldots, d_n \rangle$ is created such that $d_k(e) = 0$ and $\forall i < k, d_i(e) = 0$ then the next inner non-zero entry, $d_j(e)$, becomes the threshold of e and j becomes the carrier of e. If $\forall i, d_i(e) = 0$, then e is loop independent. If $\exists i \leq k$ such that $d_i(e) > 0$ then e remains carried by loop i with the same threshold.

To demonstrate dependence copying, consider Figure 2. The original edge from `A(I,J)` to `A(I,J-2)` has a distance vector of $\langle 2, 0 \rangle$ before unroll-and-jam. After unroll-and-jam the edge from `A(I,J)` now goes to `A(I,J)` in statement `10` and has a vector of $\langle 0, 0 \rangle$. An edge from `A(I,J+1)` to `A(I,J-2)` with a vector of $\langle 1, 0 \rangle$ and an edge from `A(I,J+2)` to `A(I,J-1)` with a vector of $\langle 1, 0 \rangle$ have also been created. For the first edge, the first formula for $d_i(e_m)$ applies. For the latter two edges, the second formula applies.

Invariant References. To aid in the explanation of dependence copying for invariant references, the following terms are defined.

1. If any reference is invariant with respect to a particular loop, that loop is *reference invariant.*
2. If a specific reference, v, is invariant with respect to a loop, that loop is *v-invariant.*

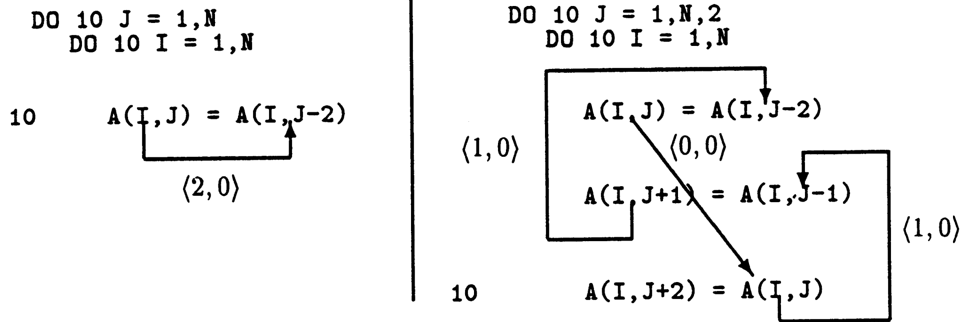

Figure 2 Distance Vectors Before and After Unroll-and-Jam

In the following example, both I and J are reference-invariant loops, I is B(J)-invariant and J is A(I)-invariant.

```
      DO 10 I = 1,N
        DO 10 J = 1,N
10        A(I) = A(I) + B(J)
```

The behavior of invariant references differs from that of variant references when unroll-and-jam is applied. Given an invariant reference v, the direction vector entry of an incoming edge corresponding to a v-invariant loop represents multiple values rather than just one value. Therefore, Equation 4 is insufficient. To accomadate the * in the outermost direction vector entry, the following steps are applied to copy an invariant dependence edge.

1. Equation 4 is applied to the minimum distance.
2. A copy of the original edge is created in each new loop body between the copies of the source and sink corresponding to the original references.
3. A copy of the dependence edge is inserted between every pair of identical references.
4. Loop-independent dependences are inserted from the references within a statement to all identical references following them.

In Figure 3, J is A(I,K)-invariant. When J is unrolled by 1, a loop-independent dependence is inserted between the two references to A(I,K) per step 1, a copy of the original dependence is made for the new copy of A(I,K) per step 2, and there are loop-carried dependences inserted between the two references per step 3. Step 4 is not necessary because step 1 has already inserted a loop-independent dependence.

The safety rules of unroll-and-jam prevent the exposure of a negative distance vector entry in the outermost non-zero position for true, output and antidependences. However, the same is not true for input dependences. The order in which reads are performed does not change the semantics of the loop. Therefore, if the outermost entry of an input dependence distance vector becomes negative during unroll-and-jam, the direction of the dependence is changed and the entries in the vector are negated.

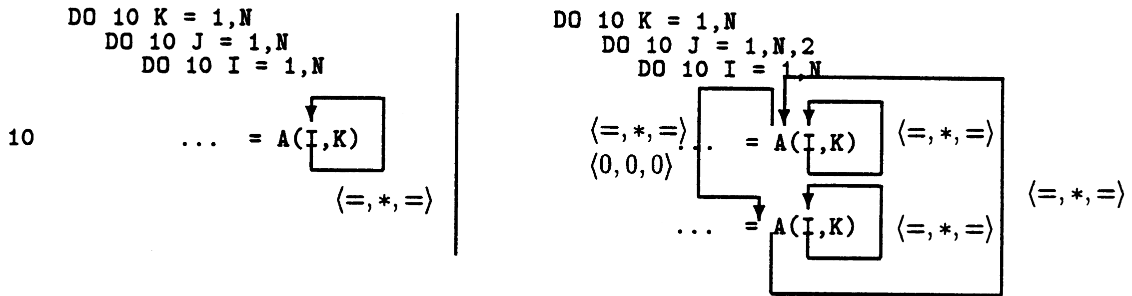

Figure 3 Invariant-Dependence Copying

3.2.3 Improving Balance with Unroll-and-Jam

As stated earlier, the goal is to convert loops that are bound by memory access time into loops that are balanced. Since unroll-and-jam can introduce more floating-point operations without a proportional increase in memory accesses, it has the potential to improve loop balance. In this section, a decision procedure to compute the balance ratio in unrolled loops is developed. As an example of how the decision procedure works, consider Figure 4. The original loop has three memory accesses per floating point operation, but it also has two dependences carried by the I-loop of which unroll-and-jam by a factor of 1 can take advantage. After unrolling the I-loop by 1, there are two dependences that become loop independent and expose opportunities for scalar replacement, leaving a balance of two memory operations per floating-point operation. The decision procedure for unroll-and-jam takes the dependence edges in the original loop and the unroll amount for the outer loop, and then computes the unrolled-loop balance as if the loop were unrolled by the specified amount. This procedure is applied to various unroll amounts in an efficient manner until the unroll amount that best balances a loop on a particular architecture is found. In Figure 4, the decision procedure would return $\beta_L = 2$ for an unroll amount of 1. Then it would continue to increase the unroll amount until the best balance is found.

Although unroll-and-jam improves balance, using it carelessly can be counterproductive. Transformed loops that spill floating-point registers or whose object code is larger than the instruction cache may suffer performance degradation. Since the size of the object code is compiler dependent, it is assumed that the instruction cache is large enough to hold the unrolled loop body. Thus, the transformation heuristic can remain relatively independent of the details of a particular software system. This leaves the following objectives for unroll-and-jam.

1. Balance a loop with a particular architecture.
2. Control register pressure.

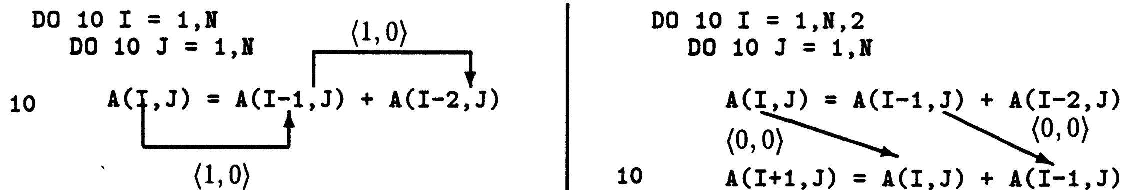

Figure 4 Objective for Unroll-and-Jam

The stated goals can be expressed as an non-linear integer optimization problem.

objective function: $\min |\beta_L - \beta_M|$
constraint: # floating-point registers required $\leq$ register-set size

The decision variables in the problem are the unroll amounts for each of the loops in a loop nest. The register-pressure constraint limits unrolling to the point before floating-point registers are spilled. For the solution to the objective function, a slightly negative difference is preferred over a slightly positive difference (it is better to have a slightly compute-bound loop rather than a slightly memory-bound one).

For each loop nest within a program, its possible transformation is modeled as a problem of the above form. Solving the optimization problem yields unroll amounts that balance the loop nest as much as possible. Using the following definitions throughout the derivation, the compile-time construction and efficient solution of a balance-optimization problem is detailed.

V = array references in an innermost loop
V_r = members of V that are memory reads
X_i = number of times the ith outermost loop in L is unrolled + 1
(This is the number of loop bodies created by unrolling loop i)
F = number of floating-point operations after unroll-and-jam
M = number of memory references after unroll-and-jam

For the purposes of this derivation, it is assumed that each loop nest is perfectly nested. Section 3.2.4 shows how to handle non-perfectly nested loops. Additionally, it is assumed that loops containing conditional-control flow in the innermost loop body are not candidates for unroll-and-jam. This is because inserting additional control flow within an innermost loop can have disastrous effects upon performance.[2] Finally, it is assumed that scalar replacement removes no stores. This is done for clarity of presentation and is not a limitation of the method [8].

Computing Transformed-Loop Balance. Since the value of β_m is a compile-time constant defined by the target architecture, only the coefficients in β_L need to be computed at compile time to have the objective function for a particular loop. To complete β_L for a particular loop, the number of floating-point operations, F, and memory references, M, is computed for one innermost-loop iteration. F is simply the number of floating-point operations in the original loop, f, multiplied by the number of loop bodies created by unroll-and-jam, giving

$$F = f \times \prod_{1 \leq i < n} X_i,$$

where n is the number of loops in a nest. M requires a detailed analysis of the dependence graph.

In computing F, it is assumed that floating-point common subexpressions are not removed from the inner loop. There are two reasons for this assumption. First, the removal of floating-point computation from memory-bound loops is not likely to improve performance. Since the loop is already bound by memory cycles, the removal of floating point operations will probably only increase the number of idle computation

[2]Experiments have shown this to be true on the IBM RS/6000.

cycles in the loop. Second, by assuming no common subexpression removal, the solution to the loop balance equation is an order of magnitude faster.

In computing M, an infinite register set is assumed. The register-pressure constraint in the optimization problem ensures that any assumption about a memory reference being removed is true. To simplify the derivation of the formulation for memory cycles, it is initially assumed that at most one incoming or outgoing edge is incident upon each $v \in V$. This restriction is removed at the end of Section 3.2.4.

The reference set of the loop can be partitioned into sets that exhibit different memory behavior when unroll-and-jam is applied. These sets are listed below.

1. $V^{\emptyset}$ = references without an incoming consistent dependence.
2. V_r^C = memory reads that have a loop-carried or loop-independent incoming consistent dependence, but are not invariant with respect to any loop.
3. V_r^I = memory reads that are invariant with respect to a loop.

Using this partitioning, the cost associated with each partition, $\{M^{\emptyset}, M_r^C, M_r^I\}$, is computed to get the total memory cost,

$$M = M^{\emptyset} + M_r^C + M_r^I.$$

For the first partition, $M^{\emptyset}$, each copy of the loop body produced by unroll-and-jam contains one memory reference associated with each original reference. This value can be expressed as follows.

$$M^{\emptyset} = \sum_{v \in V^{\emptyset}} \big(\prod_{1 \leq i < n} X_i \big).$$

For example, in Figure 5, unrolling the outer two loops by 1 creates 3 additional copies from the reference to `A(I,J,K)` that will be references to memory.

For each $v \in V_r^C$, unroll-and-jam may create dependences useful in scalar replacement for some of the new references created. In order for a reference, v, to be scalar replaced, its incoming dependence, e_v, must be converted into an innermost-loop-carried or a loop-independent dependence, which is called an *innermost* dependence edge. In terms of the distance vector associated with an edge, $d(e_v) = \langle d_1, d_2, \ldots, d_n \rangle$, the edge is made innermost when its distance vector becomes $d(e'_v) = \langle 0, 0, \ldots, 0, d_n \rangle$. Essentially, the subscript expression associated with each outer-loop induction variable must be identical in both the source and sink of the dependence in order for the reference at the sink of the dependence to be removed. After unroll-and-jam, only some of the references will be the sink of an innermost edge. The decision procedure for applying

```
      DO 10 K = 1,N
         DO 10 J = 1,N
            DO 10 I = 1,N
10             A(I,J,K) = ...
```

```
      DO 10 K = 1,N,2
         DO 10 J = 1,N,2
            DO 10 I = 1,N
               A(I,J,K) = ...
               A(I,J,K+1) = ...
               A(I,J+1,K) = ...
10             A(I,J+1,K+1) = ...
```

Figure 5 $V^{\emptyset}$ Before and After Unroll-and-Jam

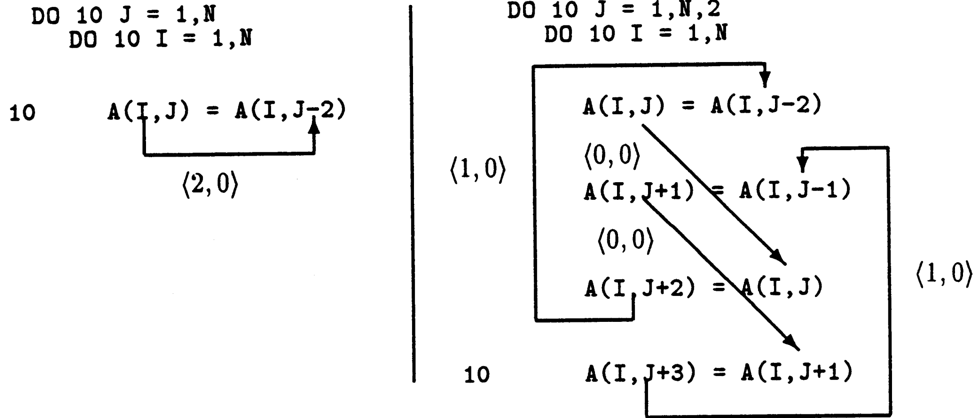

Figure 6 V_r^C Before and After Unroll-and-Jam

unroll-and-jam must quantify this value. For example, in Figure 6, the first copy of `A(I,J-2)`, the reference to `A(I,J-1)`, is the sink of a dependence from `A(I,J+3)` that is not innermost, but the second and third copies, `A(I,J)` and `A(I,J+1)`, are the sinks of innermost edges (in this case, ones that are loop independent) and will be removed by scalar replacement.

To compute the number of memory references inserted in the innermost loop by unroll-and-jam after scalar replacement, first the total number of references before scalar replacement is computed as in $M^\emptyset$,

$$\sum_{v \in V_r^C} (\prod_{1 \leq i < n} X_i).$$

Next, the number of new references that will be removed by scalar replacement is computed by quantifying the number of references that will be the sink of an innermost dependence after unroll-and-jam. To simplify the derivation, only the unrolling of loop i is considered, where $\forall e, d(e) = \langle 0, \ldots, 0, d_i, 0, \ldots, 0, d_n \rangle$. This limits the dependences to be carried by loop i or to be innermost already. Later, the formulation is extended to arbitrary loop nests and distance vectors.

Recall from Equation 4 in Section 3.2.2 that there must be some $v \in V_r^C$ for which $d_i(e_v) < X_i$ in order to create a distance vector with a 0 in the ith position, resulting in an innermost dependence edge (if $d_i(e) \geq X_i$, then not enough outer iterations are unrolled to create a reference with the subscript expressions containing the ith induction variable being equivalent in both the source and sink of a dependence). However, depending upon which of the two distance formulas is applied, it may be impossible to create an edge with a 0 in the ith position. In order for the formula that allows an entry to become 0, $d_i(e'_v) = \lfloor \frac{d_i(e_v)}{X_i} \rfloor$, to be applied, the edge created by unroll-and-jam, $e'_v = \langle w_m, v_n \rangle$, must have $n \geq m$. If $n < m$, then the applicable formula, $d_i(e'_v) = \lfloor \frac{d_i(e_v)}{X_i} \rfloor + 1$, prevents an entry from becoming 0. Below, Theorem 2 shows that $n \geq m$ is true for $X_i - d_i(e_v)$ of the new dependence edges if $d_i(e_v) < X_i$. This shows that $X_i - d_i(e_v)$ new references can be removed by scalar replacement.

Theorem 2 *Given $0 \le m, d_i(e) < X_i$ and $n = (m + d_i(e)) \bmod X_i$ for each new edge $e'_v = \langle w_m, v_n \rangle$ created from $e_v = \langle w_0, v_0 \rangle$ by unroll-and-jam, then*

$$n \ge m \iff m < X_i - d_i(e).$$

Proof.

($\Rightarrow$)

Given $n \ge m$, assume

$m \ge X_i - d_i(e)$.

Since

$0 \le m, d_i(e) < X_i$

then

$n = (m + d_i(e)) \bmod X_i$

$= m + d_i(e) - X_i$

giving

$m + d_i(e) - X_i \ge m$.

Since

$d_i(e) < X_i$,

the inequality is violated, leaving

$m < X_i - d_i(e)$.

($\Leftarrow$)

Given

$m < X_i - d_i(e)$ and $m, d_i(e) \ge 0$,

then

$n = (m + d_i(e)) \bmod X_i$

$= m + d_i(e)$

Since

$m, d_i(e) \ge 0$

then

$m + d_i(e) \ge m$.

giving

$n \ge m$. □

In the case where $X_i \le d_i(e_v)$, no edge can be made innermost, resulting in a general formula of $(X_i - d_i(e_v))^+$ edges with $d_i(e'_v) = 0$. [3]

For an arbitrary distance vector, if any outer entry is left greater than 0 after unroll-and-jam, the new edge will not be innermost. If a dependence edge becomes innermost, additional unrolling of any loop creates a copy of that edge. For example, in Figure 6, the load from `A(I,J+1)` has an incoming innermost edge that is a copy of the incoming edge for the load from `A(I,J)`. The edge into `A(I,J+1)` was created by increasing the unroll amount of the `J`-loop after `A(I,J)`'s edge was created. Quantifying these ideas gives

$$\prod_{1 \le i < n} (X_i - d_i(e_v))^+$$

references that can be scalar replaced. Subtracting this value from the total number of references before

[3] $x^+ = \begin{cases} x & \text{if } x \ge 0 \\ 0 & \text{if } x < 0 \end{cases}$

scalar replacement and after unroll-and-jam gives the value for M_r^C. Therefore,

$$M_r^C = \sum_{v \in V_r^C} \Big(\prod_{1 \le i < n} X_i - \prod_{1 \le i < n} (X_i - d_i(e_v))^+ \Big).$$

In Figure 6, $X_1 = 4$ and $d_1(e) = 2$ for the reference to `A(I,J-2)`. The formula correctly predicts that 2 memory references will be removed. Note that if $X_1 \le 2$ were true, no memory references would have been removed because no distance vector would have a 0 in the first entry.

For references that are invariant with respect to a loop, memory behavior is different from variant references. If $v \in V_r^I$ is invariant with respect to loop L_i, $i < n$, unrolling L_i will not introduce more references to memory because each unrolled copy of v will access the same memory location as v. This is because the induction variable for L_i does not appear in the subscript expression of v. Unrolling any loop that is not v-invariant will introduce memory references. Finally, if v is invariant with respect to the innermost loop, no memory references will be left after scalar replacement no matter which outer loops are unrolled (scalar replacement removes all innermost-loop-invariant references). In Figure 7, unrolling the K-loop introduces memory references, while unrolling the J-loop does not. Using these properties gives

$$M_r^I = \sum_{v \in V_r^I} \Big(\prod_{1 \le i < n} \omega(e_v, i, n) \Big)$$

where

$\omega(e, i, n) \Leftarrow$ if e is invariant wrt loop n then
  return 0
 else if e is invariant wrt loop i then
  return 1
 else
  return X_i

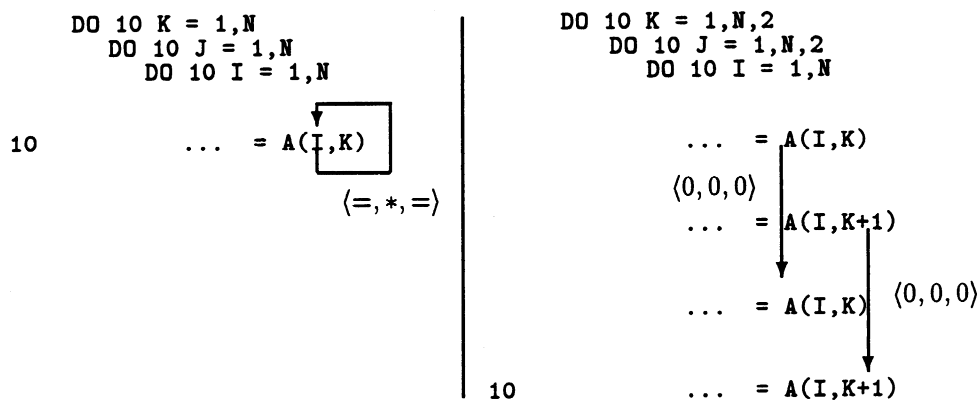

Figure 7 V_r^I Before and After Unroll-and-Jam

Now that the formulation for loop balance in relation to unroll-and-jam has been obtained , it can be used to prove that unroll-and-jam will never increase the measure of loop balance. Consider the following formula for β_L when unrolling only loop i.

$$\beta_L = \frac{\sum_{v \in V^{\emptyset}} X_i + \sum_{v \in V_r^C} X_i - (X_i - d_i(e_v))^+ + \sum_{v \in V_r^I} \omega(e_v, i, n)}{f \times X_i}$$

A simplified formula of M for a given X_i may be determined by examining each of vertices and edges in each of the above vertex sets, giving the following where $\forall j, a_j \geq 0$.

$$\begin{aligned} M^{\emptyset} &= a_0 X_i \\ M_r^C &= a_1 X_i + a_2 \\ M_r^I &= a_3 X_i + a_4 \end{aligned}$$

Combining these terms gives

$$\beta_L = \frac{(c_0 X_i + c_1)}{f \times X_i}$$

where $c_0, c_1 \geq 0$. Note that the values of c_0 and c_1 may vary with the value of X_i, but they are always non-negative. As X_i increases, c_0 may decrease and c_1 may increase because the value of $(X_i - d_i(e_v))^+$ may become positive for some $v \in V_r^C$. Theorem 3 below shows that if loop i is unrolled n times, there will be $n * f$ flops and possibly less than $n * m$ memory references, where m is the number of memory references in the original loop. Hence, unroll-and-jam does not increase the measure of unroll-and-jam.

Theorem 3 *The function for β_L is nonincreasing in the i dimension.*

Proof.

For the original loop, $X_i = 1$ and

$\beta_L = \frac{(c_0 + c_1)}{f}$

If unroll-and-jam is applied to the i-loop $n - 1$ times, $n > 1$, then $X_i = n$ giving

$\beta'_L = \frac{(c'_0 n + c_1 + c_2)}{fn}$

where c_2 is the sum of all $d_i(e_v)$ where $(X_i - d_i(e_v))^+$ became positive with the increase in X_i. Since c_0 will decrease or remain the same with the change in X_i, $c'_0 \leq c_0$. The difference between c_0 and c'_0 is the number of edges where $(X_i - d_i(e_v))^+$ becomes positive with the change in X_i. Given that $X_i = n$, the maximum value of any $d_i(e_v)$ added into c_2 is $n - 1$ (any greater value would not make $(X_i - d_i(e_v))^+$ positive). Therefore, the maximum value of β'_L is

$\beta'_L = \frac{(c'_0 n + c_1 + (c_0 - c'_0)(n-1))}{fn}$

To show that β_L is nonincreasing, $\beta_L \geq \beta'_L$ must be true.

$$\begin{aligned} \beta_L &\geq \beta'_L \\ \frac{n\beta_L}{n} &\geq \beta'_L \\ \frac{n(c_0+c_1)}{nf} &\geq \frac{(c'_0 n + (c_0 - c'_0)(n-1) + c_1)}{fn} \\ nc_0 + nc_1 &\geq nc'_0 + nc_0 - nc'_0 + c'_0 - c_0 + c_1 \\ (n-1)c_1 &\geq c'_0 - c_0 \end{aligned}$$

Since $n > 1, c_1 \geq 0$ and $c'_0 \leq c_0$, the inequality holds and β_L is nonincreasing. □

Theorem 4 *Unrolling multiple loop nests in an outermost- to innermost-loop order, does not increase β_L.*

Theorem 4 can be proved by simple induction on the number of loops unroll-and-jammed.

An Example. As an example of a formula for loop balance, consider the following loop.

```
      DO 10 J = 1,N
         DO 10 I = 1,N
10          A(I,J) = A(I,J-1) + B(I)
```

`A(I,J)` $\in V^{\emptyset}$, `A(I,J-1)` $\in V_r^C$ with an incoming distance vector of $\langle 1, 0 \rangle$, and `B(I)` $\in V_r^I$. This gives

$$\beta_L = \frac{X_1 + X_1 - (X_1 - 1)^+ + 1}{X_1}$$

Estimating Register Pressure. The next step in creating a balance-optimization problem for a particular loop is to derive a formula for register pressure. To compute the number of registers required by an unrolled loop, it must be determined which references can be removed by scalar replacement and how many registers each of those references needs. This quantity is called R. As in computing M, the partitioned sets of V, $\{V^{\emptyset}, V_r^C, V_r^I\}$, are considered. Associated with each of the partitions of V is the number of registers required by that set, $\{R^{\emptyset}, R_r^C, R_r^I\}$, giving

$$R = R^{\emptyset} + R_r^C + R_r^I.$$

Since each member of $V^{\emptyset}$ has no incoming dependence, registers cannot be used to capture reuse. However, members of $V^{\emptyset}$ still require a register to hold a value during expression evaluation. To compute the number of registers required for those memory references not scalar replaced and for those temporaries needed during the evaluation of expressions, the tree labeling technique of Sethi and Ullman is used [23]. Using this technique, the number of registers required for each expression in the original loop body is computed and the maximum over all expressions is taken as the value for $R^{\emptyset}$. This technique does not take into account the increase in register pressure due to instruction scheduling. This issue is addressed at the end of Section 3.2.4.

For variant references whose incoming edge is carried by some loop, $v \in V_r^C$, it must be determined how many references will be removed by scalar replacement after unroll-and-jam and how many registers are required for those references. The former value has previously been derived in the computation of M_r^C and is shown below.

$$\sum_{v \in V_r^C} \Big(\prod_{1 \le i < n} (X_i - d_i(e_v))^+ \Big)$$

The latter value, $d_n(e_v) + 1$, comes from scalar replacement, where it is assumed that all registers interfere [10]. All registers are assumed to interfere because any value that flows across a loop-carried dependence interferes with all other values and predicting the effects of scheduling before optimization to determine interference for loop-independent dependences is compiler dependent. To avoid reliance upon a particular scheduling algorithm, registers are assumed not to be reused, giving

$$R_r^C = \sum_{v \in V_r^C} \Big(\prod_{1 \le i < n} (X_i - d_i(e_v))^+ \Big) \times (d_n(e_v) + 1).$$

References that are invariant with respect to an outer loop require registers only if a corresponding reference-invariant loop is unrolled. Unrolling a loop that is not v-invariant does not increase register pressure, but rather creates opportunities for scalar replacement that may be utilized if a v-invariant loop is unrolled. No matter which loops are unrolled, references that are invariant with respect to the innermost loop always require registers. In Figure 7, the reference to `A(I,K)` requires no registers unless the `J`-loop is unrolled and

since K is unrolled once, two registers are required. Formulating this behavior gives

$$R_r^I = \sum_{v \in V_r^I} (\prod_{1 \le i < n} \alpha(e_v, i, n))$$

where

$\alpha(e,i,n) \Leftarrow$ if e is invariant wrt loop i then
    if $(\exists X_j | X_j > 1 \wedge e$ is invariant wrt loop $j)$ or
        (e is invariant wrt loop n) then
      return 1
    else
      return 0
  else
    return X_i

An Example. As an example of a formula for computing register pressure, consider the following loop.

```
      DO 10 J = 1,N
         DO 10 I = 1,N
 10         A(I,J) = A(I,J-1) + B(J)
```

A(I,J) $\in V^{\emptyset}$, A(I,J-1) $\in V_r^C$ with an incoming distance vector of $\langle 1,0 \rangle$ and B(J) $\in V_r^I$. This gives

$$R = 1 + (X_1 - 1)^+ + X_1.$$

3.2.4 Applying Unroll-and-Jam in a Compiler

This section discusses how to take the previous optimization problem and solve it practically for real program loop nests on real machines. First, it is shown how to choose the loops to which unroll-and-jam will be applied and then it is shown how to choose the unroll amounts for those loops. Next, interlock removal is presented, followed by the effects of multiple incident edges on balance computation. Then, non-perfectly nested loops are handled and finally, machine- and compiler-dependent issues that may have an effect on loop balance are presented.

Picking Loops to Unroll. Applying unroll-and-jam to all of the loops in an arbitrary loop nest can make the solution to the optimization problem extremely complex. To simplify the solution procedure, unroll-and-jam is only applied to a subset of the loop nest. Since experience suggests that most loop nests have a nesting depth of 3 or less, unroll-and-jam can be limited to 1 or 2 loops in a given nest. Even though tiling for cache may increase the depth of the loop, it is assumed that unroll-and-jam is only performed on the tiled iteration space that contains the cache reuse [24]. Unrolling outer loops that iterate by the tile size would increase the amount of data required to be held in cache and thus, defeat the purpose of tiling. For example, in the tiled loop

```
     DO I = 1,N,IS
       DO J = 1,N
         DO II = II, II+IS-1
 10        A(II) = A(II) + B(J)
```

unroll-and-jam might be applied to the J-loop, but applying it to I would probably not be profitable.

The heuristic for determining the subset of loops for unroll-and-jam is to pick those loops whose unrolling is likely to result in the fewest memory references in the unrolled loop. In particular, the loops that carry the most dependences that can be innermost after unroll-and-jam is applied should be unrolled. To find these loops, each distance vector is examined to determine if it can be made innermost for each pair of loops in the nest. Given a distance vector $d(e) = \langle d_1, \ldots, d_i, \ldots, d_j, \ldots, d_n \rangle$ where $d_i, d_j \geq 0$, unrolling loops i and j can make e innermost if $\forall k, k \neq i, j, n \; d_k(e) = 0$. If only one loop, i, is unrolled, then e can be made innermost if $\forall k, k \neq i, n \; d_k(e) = 0$.

Picking Unroll Amounts. In the following discussion of the problem solution, unrolling only one loop is considered in order to bring clarity to the solution procedure. Later, we discuss how to extend the solution to the problem for two loops. Given this restriction, the optimization formula can be reduced to the following

$$\min \left| \frac{\sum_{v \in V^\bullet} X_i + \sum_{v \in V_r^C} (X_i - \delta) + \sum_{v \in V_r^I} \omega(e_v, i, n)}{f \times X_i} - \beta_m \right|$$

$$R^\emptyset + X_i \times \sum_{v \in V_r^C} (\delta \times (d_n(e_v) + 1)) + \sum_{v \in V_r^I} \alpha(e_v, i, n) \leq \varrho$$

$$X_i \geq 1$$

where $\delta = (X_i - d_i(e))^+$ and ϱ is the effective size of the floating-point register set for the target architecture. The solution procedure begins by examining the vertices and edges of the dependence graph to accumulate the coefficient for X_i and any constant term as was shown earlier. Essentially, the summations are solved to get a linear function of X_i. The $^+$-function requires us to keep an array of coefficients and constants, one set for each value of d_i, that will be used depending upon the value of X_i. In practice, most distances are 0, 1 or 2, allowing us to keep 4 sets of values: one for each of the common distances and one for all remaining distances. The value of X_i determines which coefficient set is used. If $d_i > 2$ for some edge, the results may be imprecise if $3 \leq X_i < d_i$. However, experimentation discovered no instances of this situation.

Given a set of coefficients, the solution space can be searched, while checking register pressure, to determine the best unroll factor. Since most dependence distances are 0, 1 or 2, unrolling more than ϱ will probably increase the register pressure beyond the physical limits of the target machine. Therefore, the size of the solution space can be bounded by ϱ. By Theorem 3, the solution space is sorted in descending order, giving a time bound of $O(\log \varrho)$ time to search the solution space for the best answer.

To extend unroll-and-jam to two loops, a simplified optimization problem loop can be created from the general formula. The coefficients and constant terms in β_L for two particular loops, i and j, can be constructed by keeping 4 sets of coefficient values for each loop. By Theorem 3, if either X_i or X_j is held constant, the function for β_L is nonincreasing, which gives a two dimensional solution space with each row and column sorted in decreasing order, lending itself to an $O(\varrho)$ intelligent search. In general, a solution for unroll-and-jam of k loops, $k > 1$, can be found in $O(\varrho^{k-1})$ time. Appendix A shows matrix multiply before and after the application of unroll-and-jam using the decision procedure for two loops.

Removing Interlock. After unroll-and-jam, a loop may contain pipeline interlock, leaving idle computational cycles. To remove these cycles, the technique of Callahan, *et al.*, can be used to estimate the amount of interlock. If interlock exists, one loop can be unrolled to create more parallelism by introducing enough

copies of the inner loop recurrence [6]. Given the pipeline length, l_p, and the length of the longest recurrence carried by the innermost loop, $\rho(L_n)$, unroll-and-jam can be applied to an outer loop until $F > \rho(L_n) \times l_p$. Essentially, enough floating-point computation is moved into the innermost loop to fill the delay slots caused by interlock. This is possible because copies of the inner-loop recurrence will be parallel with each other and unroll-and-jam will not move outer-loop recurrences inward [6].

Experimental evidence in Section 4 suggests that it is reasonable to ignore interlock detection completely and only optimize for balance when the length of the pipeline is short. Because of the short pipeline, little unrolling will probably be necessary to remove interlock and the balance optimization problem already discussed will most likely take care of that interlock. Applying only one heuristic will speed up compilation. However, with a long pipeline, it may be advantageous to remove interlock first because the unrolling determined by the optimization problem will be less likely to be enough to remove interlock. Additionally, removing interlock first rather than second allows the optimization problem to keep register pressure within the resources of the machine.

Multiple Edges. In general, it cannot be assumed that there will be only one edge entering or leaving each node in the dependence graph since multiple incident edges are possible and likely. One possible solution is to treat each edge separately as if it were the only incident edge. Unfortunately, this is inadequate because many edges may correspond to the flow of the same set of values. In the loop,

```
      DO 10 J = 1,N
         DO 10 I = 1,N
 10         B(I,J) = A(I-1,J) + A(I-1,J) + A(I,J)
```

there is a loop-carried input dependence from `A(I,J)` to each `A(I-1,J)` with distance vector $\langle 0,1\rangle$ and a loop-independent dependence between the `A(I-1,J)`'s. Considering each edge separately would give a register pressure estimation of 5 registers per unrolled iteration of `J` rather than the actual value of 2, since both values provided come from the same source and require use of the same registers. Although the original estimation is conservative, it is more conservative than necessary. To improve the estimation, register sharing is considered.

To relate all references that access the same values, the scalar replacement algorithm considers the oldest reference in a dependence chain as the reference that provides the value for the entire chain [10]. Using this technique alone to capture sharing within the context of unroll-and-jam, however, can cause us to underestimate register pressure. Consider the following example.

```
      DO 10 I = 1,N
         DO 10 J = 1,N
 10         C(I,J) = A(I+1,J) + A(I,J) + A(I,J)
```

The second reference to `A(I,J)` has two incoming input dependences: one with a distance vector of $\langle 1,0\rangle$ from `A(I+1,J)` and one with a distance vector of $\langle 0,0\rangle$ from `A(I,J)`. Unrolling the `I`-loop by 1 gives

```
      DO 10 I = 1,N,2
         DO 10 J = 1,N
 1          C(I,J)   = A(I+1,J) + A(I,J)   + A(I,J)
 10         C(I+1,J) = A(I+2,J) + A(I+1,J) + A(I+1,J)
```

Now, the reference `A(I+1,J)` in statement `1` provides the value used in statement `10` by both references to `A(I+1,J)`, and the first reference to `A(I,J)` in statement `1` provides the value used in the second reference to `A(I,J)` in statement `1`. Before unrolling, it cannot be determined which of the two references to `A` will ultimately provide the value for copies of the second reference to `A(I,J)` because they both provide the value

to a copy at different points in the loop. Therefore, the edge from the oldest reference, `A(I+1,J)`, cannot necessarily be picked to calculate its register pressure. Instead, a stepwise approximation can be used, where each possible oldest reference within a dependence chain is considered.

The first step is to partition the dependence graph so that all references using the same registers after unrolling will be put in the same partition. The algorithm in Figure 8 accomplishes this task. Once the references have been partitioned, the algorithm in Figure 9 is applied to calculate the coefficients used in R. First, in the procedure *OldestValue*, the node in each register partition that is first to reference the value that flows throughout the partition is determined. The reference, v, that has no incoming dependence from another member of its partition or has only loop-carried incoming dependences that are carried by v-invariant loops is found. Only those edges that can create reuse in the unrolled loop as described in Section 3.2.4 are considered. After unroll-and-jam, the oldest reference will provide the value for scalar replacement for the entire partition and is called the *generator* of the partition.

Next, in procedure *SummarizeEdges*, the distance vector that will encompass all of the outgoing edges (one to each node in the partition) from the generator of each partition is computed. To be conservative, for $i = 1, \ldots, n-1$ the minimum d_i for all of the generator's outgoing edges is picked. This guarantees that the innermost-loop reuse within a partition will be measured as soon as possible in the unrolled loop body. For d_n, the maximum value is picked to ensure that the number of registers used is not underestimated. After computing a summarized distance vector, it is used to accumulate the coefficients to the equation for register pressure.

At this point in the algorithm, a distance vector and formula have been created to partially estimate the register requirements of a partition given a set of unroll values. In addition, the intermediate points where the generator of a partition does not provide the reuse at the innermost-loop level for some of the references within the partition must be determined (see `A(I-1,J)` and `A(I,J)` in the example in this section). Register requirements will be underestimated if these situations are not handled.

First, applying procedure *Prune* to each partition, we remove all nodes for which there are no intermediate points where a reference other than the current generator provides the values for scalar replacement. Any reference that occurs on the same iteration of loops 1 to $n-1$ as the current generator of the partition will have no such points. Their value will always be provided by the current generator. Next, the number of

```
Procedure PartitionNodes(G, j, k)

Input: G = (V, E), the dependence graph
          j, k = loops to be unrolled

    put each v ∈ V in its own partition, P(v)
    while ∃ an unmarked node do
        let v be an arbitrary unmarked node
        mark v
        forall w ∈ V|((∃e = (w, v) ∨ e = (v, w))∧
          (e is input or true and is consistent) ∧
          (for i = 1 ... n − 1, i ≠ j, i ≠ k, d_i(e) = 0)) do
            mark w
            merge P(w) and P(v)
        enddo
    enddo
end
```

Figure 8 PartitionNodes

```
Procedure ComputeR(P, G, R)

Input: P = partition of memory references
       G = (V, E), the dependence graph
       R = register pressure formula

  foreach p ∈ P do
    v = OldestValue(p)
    d = SummarizeEdges(v)
    update R using d
    Prune(p,v)
    while size(p) > 1 do
      u = OldestValue(p)
      d = SummarizeEdges(u)
      update R using R_d − R_{d+d(ê)}, ê = (v, u)
      Prune(p,u)
      v = u
    enddo
  enddo
end

Procedure OldestValue(p)
  foreach v ∈ p
    if ∀e = (u, v), v is invariant wrt the loop at level(e)∨
                P(u) ≠ P(v)) then return v
end

Procedure SummarizeEdges(v)
  for i = 1 to n − 1
    foreach e = (v, w) ∈ E, w is a memory read
      d_i^s = min(d_i^s, d_i(e))
  foreach e ∈ E, w is a memory read
    d_n^s = max(d_n^s, d_n(e))
  return d^s
end

Procedure Prune(p,v)
  remove v from p
  foreach e = (v, w) ∈ E
    if d(e) = (=, =, ..., =, *) then remove w
end
```

Figure 9 Compute Coefficients for Optimization Problem

additional registers needed by the remaining nodes in the partition is computed by calculating the coefficients of R for an intermediate generator of the modified partition. The only difference from the computation for the original generator is that the coefficients of R derived from the summarized distance vector of the modified partition are modified by accounting for the registers that have already counted with the previous generator. This is done by computing the coefficients for R as if an intermediate generator were the generator in an original partition. Then, we subtract from R the number of references that have already been handled by the previous generator. The distance vector of the edge, $\hat{e}$, from the previous generator, v, to the new intermediate generator, u, is used to compute the latter value. Given the current summarized distance vector, d^s, and $d(\hat{e})$ for the edge between the current and previous generators, compute $R_{d^s} - R_{(d^s+d(\hat{e}))}$. The value $d^s + d(\hat{e})$ represents the summarized distance vector if the previous generator were the generator of the modified partition. In the example presented in this example, $d^s = \langle 0,0 \rangle$ and $d(\hat{e}) = \langle 1,0 \rangle$ giving $(X_1 - 0)^+ - (X_1 - 1)^+$ additional references removed. These steps for intermediate points are repeated until the partition has 1 or less members.

Now that we have shown how to handle multiple edges for variant references, we discuss the differences for references that are invariant with respect to some loop. Since multiple edges are allowed, references may be invariant with respect to one of the unrolled loops and variant with respect to the other and have dependences carried by both loops. This is not the case if only one edge can be incident upon a reference. If the previous situation occurs, the reference is treated as a member of V^I with respect to the first loop and V^C with respect to the second.

Finally, multiple edges affect the computation of M similarly. However, partitioning is not necessary to handle multiple edges because sharing does not occur. Instead, each reference is considered separately, using a summarized vector that consists of the minimum d_i over all incoming edges of v for $i = 1, \ldots, n-1$. Although this is a slightly different approximation than used for R, it is more accurate and will result in a better estimate of β_L.

Non-Perfectly Nested Loops It may be the case that the loop nest on which unroll-and-jam is to be performed is not perfectly nested. Consider the following loop.

```
      DO 10 I = 1,N
         DO 20 J = 1,N
20          A(I,J) = B + E
         DO 10 J = 1,N
10          C(J) = C(J) + D(I)
```

On the RS/6000, where $\beta_M = 1$, the I-loop would not be unrolled for the loop surrounding statement 20, but would be unrolled for the loop surrounding statement 10. To handle this situation, the I-loop can be distributed around the J-loops as follows

```
      DO 20 I = 1,N
         DO 20 J = 1,N
20          A(I,J) = B + E
      DO 10 I = 1,N
         DO 10 J = 1,N
10          C(J) = C(J) + D(I)
```

and each loop unrolled independently.

In general, unroll amounts are computed with a balance-optimization formula for each innermost loop body as if it were within a perfectly nested loop body. A vector of unroll amounts that includes a value for each of a loop's surrounding loops and a value for the loop itself is kept for each loop within the nest. When determining if distribution is necessary, if a loop that has multiple inner loops is encountered at the next

level, each of the unroll vectors for each of the inner loops is compared to determine if any unroll amounts differ. If they differ, each of the loops from the current loop to the loop at the outermost level where the unroll amounts differ is distributed. Then each of the newly created loops is examined for additional distribution at inner levels. If any loop that must be distributed cannot be distributed due to a recurrence, the unroll vector values in each vector are set to the smallest unroll amount in all of the vectors for each level (note that a loop that cannot be distributed will have an unroll value of 0 to ensure safety) [3]. The complete algorithm for distributing loops for unroll-and-jam is given in Figure 10.

Machine- and Compiler-Dependent Issues. This paper has described a solution to the loop balance and register-pressure problem, but there are other factors that may need to be addressed on some architecture-compiler combinations.

- **Addressing.** Unroll-and-jam will probably increase the number of address registers required for the array references in the inner loop body. Address-register pressure is a compiler-dependent issue that will affect balance if there is register spilling. In addition to address registers, the updating of their contents must also be considered. On architectures with an auto-increment addressing mode, there will be no effect on balance. However, if explicit instructions are necessary to increment address registers, then these instructions must be included. Most likely they will count as additional memory instructions since memory references are often serviced by an integer pipeline.

```
Procedure DistributeLoops(L, i)

Input: L = loop to check for distribution
       i = nesting level of L

  if L has multiple inner loops at level i + 1 then
    l = outermost level where the unroll amounts differ for the
        loop nests.
    if l > 0 then
      if loops at levels l, ..., i can be distributed then
        𝓛 = L
        while level(𝓛) ≤ l do
          distribute 𝓛 around its inner loops
          𝓛 = parent(𝓛)
        enddo
      else
        for j = 1 to i do
          m = minimum of all unroll vectors for the inner loops of
              L at level j
          assign to those vectors at level j the value m
        enddo
      endif
  endif
  N = the first inner loop of L at level i + 1
  if N exists then
    DistributeLoops(N, i + 1)
  N = the next loop at level i following L
  while N exists do
    DistributeLoops(N, i)
    N = the next loop at level i following N
  enddo
end
```

Figure 10 Distribute Loops for Unroll-and-jam

- **Instruction Cache.** In this paper, it is assumed that the instruction cache is large enough to hold an unrolled loop body. However, some architectures (*e.g.* Intel i860) have very small on-chip instruction caches. In these cases, it would be necessary to understand how the back-end compiler generates instructions so that some estimate of code size could be performed in order to limit unrolling.
- **Scheduling.** Instruction scheduling can increase register pressure. The heuristic presented does not handle this directly since intimate knowledge of the back-end compiler's scheduling algorithm would be necessary to give an effective estimate. One method for handling scheduling is to reduce the number of available registers artificially to allow room for the scheduler to require additional resources. The number of effective registers can be determined experimentally.
- **Data Cache.** The design of the data cache can have a great effect on the ability of the cache to retain data as assumed by the heuristic. Direct-mapped caches often suffer from interference between references [19]. However, this issue is beyond the scope of this paper and is left to the cache-optimization phase of the compiler [19, 20, 24].
- **Special Instructions.** Many advanced microprocessors have a floating-point multiply-add instruction that allows the two arithmetic instructions to be completed in time less than it would take to perform each instruction separately. This, in turn, affects the peak machine balance by increasing the number of flops that can be performed per cycle. However, reducing machine balance can give a false impression of β_M if the instruction is not used and, as a result, unnecessary unrolling may be performed. Instead, the multiply-add can be counted as one operation rather than two since the decision procedure actually counts floating-point instruction issues.

4 EXPERIMENT

A source-to-source translator, called Memoria, has been implemented in the ParaScope programming environment, using the dependence analyzer from PFC [3, 7, 8]. The experimental design is illustrated in Figure 11. In this scheme, Memoria serves as a preprocessor, rewriting Fortran programs to improve loop balance using both unroll-and-jam and scalar replacement as described. Both the original and transformed versions of the program are then compiled and run using the standard product compiler for the target machine.

For the test machine, the IBM RS/6000 model 540 was chosen because it has a good compiler and a large number of floating-point registers (32). In addition, the RS/6000 has a number of hardware features that make loop balance a natural estimate of performance:

- **Register renaming.** This allows the heuristic to ignore the effects of pipelining across iterations on register pressure and to ignore stalls related to antidependences on registers.

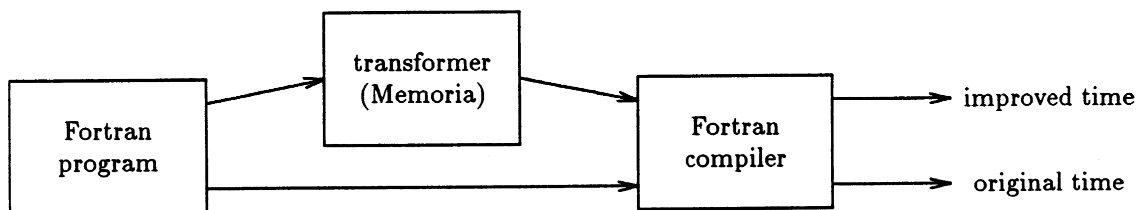

Figure 11 Experimental design.

- **Zero-cost branches.** Backward branches for loops are optimized by fetching instructions across the loop boundary and beginning operations on the next iteration before the current iteration finishes. This helps simulate a software pipelined loop and helps achieve the parallelism assumed to exist in a loop.

- **Auto-increment addressing mode.** Auto-increment allows the heuristic to ignore address instructions in the computation of balance.

- **Large instruction cache (64K).** The size of the cache allows the heuristic to ignore loop size.

- **4-way set-associative data cache.** Large associativity tends to effectively reduce cache interference.

As discussed at the end of section 3.2.4, address-register pressure needs to be controlled. For the RS/6000, any expression having no incoming innermost-loop-carried or loop-independent dependence requires an address register. Although considering address-register pressure removes the independence from a particular compiler, it could not be avoided for performance reasons. In addition, the number of registers available was artificially reduced to 26 to allow for scheduling and to prevent additional register spilling. This number was determined by experimentation with scalar replacement [8].

4.1 Performance Results

Figures 12 and 13 show performance results on the IBM RS/6000 after applying unroll-and-jam and scalar replacement to a number of kernels and complete applications. Full optimization was used on both the transformed and original code using the IBM XLF compiler version 2.2. The results are reported in normalized execution times where the performance of the transformed code is normalized against the performance of the original code. In Figures 12 and 13, (O) denotes the original code, (S) reports the results after Memoria applied scalar replacement and (US) reports the results after Memoria applied unroll-and-jam and scalar replacement. The base execution time for the original code is 100.

It should be noted that the IBM XLF compiler performs a limited form of scalar replacement for loop-independent dependences and loop-invariant references. Therefore, the results with scalar replacement show the benefit attained by using more sophisticated dependence analysis.

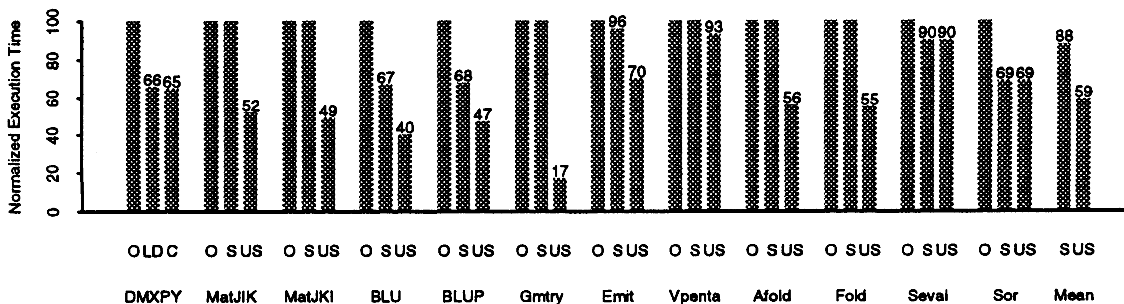

Figure 12 Kernel Performance on IBM RS/6000 (**O** = original kernel, **S** = after scalar replacement, **US** = after unroll-and-jam and scalar replacement).

Below we discuss the improvement attained by Memoria on each of the kernels and applications in our suite. First the results of each kernel displayed in Figure 12 is presented. Then, a discussion of the application performance shown in Figure 13 is presented.

DMXPY. DMXPY is a hand-optimized version of a vector-matrix multiply that is machine-specific (not for the RS/6000)) and difficult to understand. Unroll-and-jam and scalar replacement are applied to the unoptimized version (O) and compared with the performance with the LINPACKD version (LD). Memoria (C) is able to attain slightly better performance than the hand-optimized code, while allowing the kernel to be expressed in a machine-independent fashion. This is an example of how hand optimization is not necessarily best. It is better to express the kernel in a machine-independent form and allow the compiler to handle the machine details. This allows good performance across a variety of architectures without programmer effort.

Matrix Multiply. The results for two versions of matrix multiply on arrays of size 50x50 are displayed. Memoria achieves integer factor speedups on small matrices where most memory references are from cache. Not only does memory reuse in the loop improve, but also available instruction-level parallelism.

Linear Algebra Kernels. Experimentation with the block versions of LU decomposition with and without partial pivoting (LU and LUP, respectively) is shown. Memoria attains a factor of over 2 improvement in run-time. Each of the kernels contained an inner-loop reduction that is amenable to unroll-and-jam. In these instances, unroll-and-jam introduces multiple parallel copies of the inner-loop recurrence to improve balance.

NAS Kernels. Three of the NAS kernels (Gmtry, Emit and Vpenta) from the SPEC benchmark suite are examined. Gmtry and Emit observe larger improvements because they contain outer-loop reductions to which unroll-and-jam can be applied.

Geophysics Kernels. Included in the experiment are two geophysics kernels: one that computes the adjoint convolution of two time series (Afold) and another that computes the convolution (Fold). Each of these loops requires the optimization of trapezoidal, rhomboidal and triangular iteration spaces using techniques that have been developed in other works [5, 9]. Again, these kernels contain inner-loop reductions.

Seval and Sor. Seval evaluates a B-spline and has only singly nested loops (unroll-and-jam is not applicable). The reported improvement is obtained by applying scalar replacement to the main computational loop. Sor computes the successive-over-relaxation of a grid. Unroll-and-jam is applied only to remove pipeline interlock, but no improvement is noted.

Applications. We originally chose 27 Fortran applications from SPEC, Perfect, RiCEPS and local sources to be a part of our study. Of those 27, 5 failed to be analyzed successfully by PFC, 1 failed to compile on the RS6000 and 10 contained no reuse opportunities for the loop-balance optimization algorithms. Most of those programs that contained no reuse opportunities had loop-independent or loop-invariant reuse that was captured by the IBM XLF compiler.

The table below contains a short description of each the remaining applications that can be transformed by Memoria.

Suite	Application	Description
SPEC	Matrix300	Matrix Multiply
	Tomcatv	Mesh Generation
Perfect	Adm	Pseudospectral Air Pollution
	Arc2d	2d Fluid-Flow Solver
	Flo52	Transonic Inviscid Flow
RiCEPS	Onedim	Time-Independent Schroedinger Equation
	Shal	Weather Prediction
	Simple	2d Hydrodynamics
	Sphot	Particle Transport
	Wave	Electromagnetic Particle Simulation
Local	CoOpt	Oil Exploration

The results of performing scalar replacement and unroll-and-jam on the application suite are shown in Figure 13. The results for most applications are modest with the exception of Matrix300 and Onedim. The two applications spend considerable time computing reductions with Matrix300 being completely dominated by reductions. As shown in the kernel results, reductions present the best opportunity for improvement. For Tomcatv, Arc2d, Flo52, Simple and CoOpt, improvements from 15% to 50% are achieved on the individual loops to which unroll-and-jam is applied. Unfortunately, these loops do not dominate execution time. For Adm, Shal and Sphot, unroll-and-jam is never applied. A more detailed examination of the performance of unroll-and-jam on the applications is given in Section 4.2.

The study suggests that unroll-and-jam is most effective in the presence of reductions. Reductions often carry a large amount of reuse and unrolling a reduction greatly improves instruction-level parallelism. Unroll-and-jam is least effective in the presence of a floating-point divide because of its high computational cost. Since a divide takes 19 cycles on the RS/6000, its cost dominates loop performance enough to make memory references a minimal factor.

4.2 Effectiveness of the Loop Balance Estimate

The next portion of the experiments deals with the accuracy of the unroll-and-jam decision procedure in predicting loop balance. The ParaScope dependence analyzer is used for this portion of the experiment

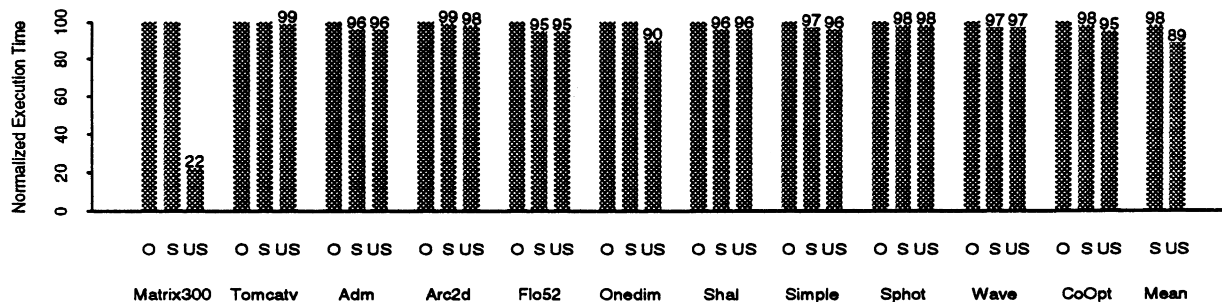

Figure 13 Application Performance on IBM RS/6000 (**O** = original application, **S** = after scalar replacement, **US** = after unroll-and-jam and scalar replacement).

rather than the PFC analyzer because the PFC analyzer became unsupported during the implementation of the statistics gathering and the ParaScope analyzer was upgraded to be of PFC quality.

In Table 1, the accuracy of the computation of balance by Memoria on the aforementioned kernels is detailed. "Predicted" numbers are obtained from the optimization problem created from the dependence graph for each loop. "Observed" numbers are estimates obtained by examining the source code with Memoria after unroll-and-jam and scalar replacement are applied. "Compiler" numbers are obtained by examining the assembler generated by the IBM XLF compiler with full optimization. "Optimal" represents the best balance given a fixed number of floating-point registers (using the IBM XLF register allocator). For the "Optimal" numbers, unroll amounts are increased until all floating-point registers are used or a minimum balance is reached. For each set of numbers, "IB" denotes balance before Memoria applied unroll-and-jam and scalar replacement, "FB" denotes the balance after the transformations and "FP" denotes the register pressure after the transformations. In each column, all references to memory are counted as one memory operation (cache miss penalties are not included) and each addition, subtraction, multiplication and multiply-add is counted as one floating-point operation. Division is counted as 19 flops since the base operations on the RS/6000 take one cycle and division takes 19 cycles.

Table 1 indicates that the predicted balance for this set of kernels is identical to the balance produced by the compiler. Slight inaccuracies in balance prediction are likely to occur when references have multiple incoming dependences with different distances at multiple loop levels. These inaccuracies do not emerge on this set of kernels. It should also be noted that these inaccuracies will not occur between the observed balance and the compiler balance. No results are reported for "Seval" because unroll-and-jam is not performed on the kernel.

For the kernels in this study, Memoria achieves a loop balance that is not appreciably different from the optimal balance. The only significant difference between the optimal balance and the balance achieved by Memoria occurred on Vpenta. Here, loop-independent dependences with limited live ranges contain reuse. Not considering register reuse causes Memoria to overestimate register pressure and unnecessarily restrict unrolling.

Kernel	Predicted			Observed		Compiler			Optimal	
	IB	FB	FP	FB	FP	IB	FB	FP	FB	FP
DMXPY	3.00	1.09	26	1.09	25	3.00	1.09	26	1.07	32
MatJIK	2.00	1.00	10	1.00	10	2.00	1.00	7	1.00	7
MatJKI	3.00	1.00	17	1.00	16	3.00	1.00	14	1.00	14
BLU	3.00	2.04	26	2.04	25	3.00	2.04	25	2.03	32
	2.00	1.00	10	1.00	10	2.00	1.00	7	1.00	7
BLUP	3.00	2.04	26	2.04	25	3.00	2.04	25	2.03	32
	2.00	1.00	10	1.00	10	2.00	1.00	7	1.00	7
Gmtry	3.00	2.04	26	2.04	25	3.00	2.04	25	2.03	32
Emit	3.00	1.09	26	1.09	25	3.00	1.09	26	1.07	32
	3.00	1.09	26	1.09	25	3.00	1.09	26	1.07	32
Vpenta	2.33	1.58	23	1.58	22	2.33	1.58	12	1.01	12
Afold	2.00	1.00	5	1.00	5	2.00	1.00	4	1.00	4
Fold	2.00	1.00	5	1.00	5	2.00	1.00	4	1.00	4
Sor	0.80	0.53	9	0.53	8	0.80	0.53	9	0.53	9

Table 1 Balance Prediction Accuracy

The register pressure results in Table 1 indicate that Memoria overestimates register pressure in most instances. This would argue for Memoria to include some heuristic for approximating scheduling to allow register reuse. However, it does not argue for the elimination of reserving registers as a buffer against the register allocator of the target compiler because Memoria only considers register pressure on a per loop basis, not over a whole routine as in global register allocation [11, 12]. Previous experiments have shown that register spilling will occur in loops with very low register pressure due to scalar replacement because register spills are done for an entire live range rather than with an understanding of program structure [4, 8].

Table 2 presents information concerning the unrolling of the loops in each kernel. "Unrolled" shows how many loops are unrolled due to balance and pipeline interlock. Of those that contained interlock, only SOR requires additional unrolling after balancing the loop without considering interlock. Interlock is removed in all other loops just by unrolling to improve the ratio of memory to floating-point operations. "Not Unrolled" shows how many loops are not unrolled due to balance and lack of potential improvement ("None"), and transformation legality ("Safety"). "Nests" reports the number of perfectly nested loop structures that are considered for unroll-and-jam. Imperfect loop nests have multiple perfect loop structures that are considered for unroll-and-jam.

Table 3 reports the accuracy of Memoria in predicting loop balance in the application suite. "Nests" reports the number of perfectly nested loops considered for unroll-and-jam, "Unrolled" reports results for those nests that are unrolled and "Not Unrolled" for those that are not unrolled. "N" in the "Unrolled" section presents the number of perfect nests in that application that are unrolled and "I" reports the number that are unrolled due to interlock. "IB", "FB" and "FP" mean the same as in Table 2. The balance numbers are reported as an average of all loops in an application to which unroll-and-jam is applied. "Predicted" numbers report the values predicted by the unroll-and-jam decision procedure and "Observed" numbers report the values calculated by examining the transformed source code. No numbers from the compiler-generated code are reported because isolating innermost unrolled loops in the thousands of lines of assembler would have been tedious and time consuming. It is expected that the compiler numbers for balance would be identical to the observed numbers as previously reported for the kernels because the output of the XLF

Kernel	Unrolled		Not Unrolled		Nests
	Balance	Interlock	Safety	None	
DMXPY	1	0	0	0	1
MatJIK	1	0	0	0	1
MatJKI	1	0	0	1	2
BLU	2	0	1	0	3
BLUP	2	0	3	0	5
Gmtry	1	0	1	5	6
Emit	2	0	0	5	7
Vpenta	1	0	0	1	2
Afold	1	0	0	0	1
Fold	1	0	0	0	1
Seval	0	0	0	0	0
Sor	0	1	0	0	1

Table 2 Kernel Unrolling Information

Application	Nests	Unrolled							Not Unrolled		
				Predicted			Observed				
		N	I	IB	FB	FP	FB	FP	N	NI	S
Matrix300	2	2	0	2.00	1.04	26	1.04	25	0	0	0
Tomcatv	5	1	0	2.00	1.58	24	1.58	24	4	4	0
Adm	106	0	0	-	-	-	-	-	106	72	34
Arc2d	75	15	0	2.70	1.55	15.9	1.55	15.1	60	46	14
Flo52	76	14	0	2.13	1.53	15.2	1.57	12.8	62	62	0
Onedim	10	6	0	2.00	1.00	10	1.00	10	4	4	0
Shal	7	0	0	-	-	-	-	-	7	7	0
Sphot	10	0	0	-	-	-	-	-	10	7	3
Simple	22	2	0	4.00	3.08	17.5	3.08	16.5	20	18	2
Wave	10	0	0	-	-	-	-	-	10	7	3
CoOpt	139	15	0	2.5	1.04	21.2	1.04	17.2	124	124	0

Table 3 Application Unrolling Information

compiler is very predictable. "N" in the "Not Unrolled" section denotes the number of perfect nests in the application that are not unrolled, "NI" reports the number of those in "N" that could not be improved by unroll-and-jam and "S" reports the number of those in "N" where unroll-and-jam is not safe.

Memoria predicts the observed numbers in every application except Flo52. This routine contains references with multiple incoming dependences that required summarization as described in Section 3.2.4. It is pleasing to note how accurately the decision procedure performs across a variety of applications and kernels.

5 RELATED WORK

Callahan, Cocke and Kennedy describe unroll-and-jam in the context of loop balance, but they do not present a method to compute unroll amounts automatically [6]. Aiken and Nicolau discuss a transformation identical to unroll-and-jam called loop quantization [1]. To ensure parallelism, they perform a strict quantization where each loop is unrolled until iterations are no longer data independent. However, with software or hardware pipelining true dependences between the unrolled iterations do not prohibit low-level parallelism. Thus, their method misses register usage benefits from true dependences and unnecessarily restricts unroll amounts. They also does not control register pressure. Wolf and Lam present a framework for determining the data locality for a loop nest and use loop interchange and tiling to improve locality [24]. They present unroll-and-jam in this context as register tiling, but they do not present a method to determine unroll amounts.

6 SUMMARY

We have presented a design for a transformation system that applies scalar replacement and unroll-and-jam to improve the balance of loops. These transformations are machine independent in the sense that they take as input only a few parameters of the target machine, such as machine balance and number of registers.

A major achievement of this work is the development of an optimization problem that minimizes the distance between machine balance and loop balance, bringing the balance of the loop as close as possible to the balance of the machine. This work has also shown how to use a few simplifying assumptions to make the solution to the optimization problem fast enough for inclusion in real compilers.

These transformations have been implemented in a Fortran source-to-source preprocessor and applied to a substantial collection of kernels and whole programs. The results show that, over whole programs, modest improvements are usually achieved, with spectacular results occurring on a few programs. The methods are particularly successful on kernels from linear algebra. These results are achieved on an IBM RS/6000, which has an extremely effective optimizing compiler and a small load penalty (1 cycle). We would expect such methods to produce larger improvements on machines with greater load penalties.

The effectiveness of scalar replacement and unroll-and-jam, as demonstrated by the experiments, argues that they should be included in every highly optimizing scalar compiler . Since it has been established that they can be carried out by a machine-independent source-to-source system that uses a few machine parameters to drive the optimization algorithm, it should be possible to develop a preprocessor that can be quickly retargeted to new processors as they emerge.

ACKNOWLEDGMENTS

Robert Reynolds provided valuable suggestions during the preparation of this document. In addition, Preston Briggs, Keith Cooper, and Uli Kremer made helpful suggestions on document form. John Cocke, Peter Markstein and David Callahan gave us the inspiration for this work.

References

[1] AIKEN, A., AND NICOLAU, A. Loop quantization: An analysis and algorithm. Tech. Rep. 87-821, Cornell University, March 1987.

[2] ALLEN, F., AND COCKE, J. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*. Prentice-Hall, 1972, pp. 1–30.

[3] ALLEN, J., AND KENNEDY, K. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems 9*, 4 (Oct. 1987), 491–542.

[4] BRIGGS, P. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, Department of Computer Science, 1992.

[5] CALLAHAN, D., CARR, S., AND KENNEDY, K. Improving register allocation for subscripted variables. *SIGPLAN Notices 25*, 6 (June 1990), 53–65. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation.*

[6] CALLAHAN, D., COCKE, J., AND KENNEDY, K. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing 5* (1988), 334–358.

[7] CALLAHAN, D., COOPER, K., HOOD, R., KENNEDY, K., AND TORCZON, L. ParaScope: A parallel programming environment. In *Proceedings of the First International Conference on Supercomputing* (Athens, Greece, June 1987).

[8] CARR, S. *Memory-Hierarchy Management.* PhD thesis, Rice University, Department of Computer Science, 1992.

[9] CARR, S., AND KENNEDY, K. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing '92* (Minneapolis, MN, November 1992).

[10] CARR, S., AND KENNEDY, K. Scalar replacement in the presence of conditional control flow. Tech. Rep. TR92283, Rice University,CRPC, Nov. 1992. To appear in *Software - Practice & Experience.*

[11] CHAITIN, G., AUSLANDER, M., CHANDRA, A., COCKE, J., HOPKINS, M., AND MARKSTEIN, P. Register allocation via coloring. *Computer Languages 6* (Jan. 1981), 45–57.

[12] CHOW, F., AND HENNESSY, J. Register allocation by priority-based coloring. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices Vol. 19, No. 6* (June 1984).

[13] DEHNERT, J. C., HSU, P. Y.-T., AND BRATT, J. P. Overlapped loop support in the Cydra 5. In Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (Boston, Massachusetts, 1989), pp. 26–38.

[14] DUESTERWALD, E., GUPTA, R., AND SOFFA, M. L. A practical data flow framework for array reference analysis and its use in optimizations. *SIGPLAN Notices 28*, 6 (June 1993), 68–77. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[15] GANNON, D., JALBY, W., AND GALLIVAN, K. Strategies for cache and local memory management by global program transformations. In *Proceedings of the First International Conference on Supercomputing.* Springer-Verlag, Athens, Greece, 1987.

[16] KENNEDY, K., AND MCKINLEY, K. Optimizing for parallelism and memory hierarchy. In *Proceedings of the 1992 International Conference on Supercomputing* (Washington, DC, July 1992), pp. 323–334.

[17] KUCK, D. *The Structure of Computers and Computations Volume 1.* John Wiley and Sons, New York, 1978.

[18] KUCK, D., KUHN, R., LEASURE, B., AND WOLFE, M. The structure of an advanced retargetable vectorizer. In *Proceedings of COMPSAC 80, the 4th International Computer Software amd applications conference* (Chicago, IL, Oct. 1980), pp. 709–715.

[19] LAM, M. S., ROTHBERG, E. E., AND WOLF, M. E. The cache performance and optimizations of blocked algorithms. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (Santa Clara, California, 1991), pp. 63–74.

[20] MEADOWS, L., NAKAMOTO, S., AND SCHUSTER, V. A vectorizing, software pipelining compiler for LIW and superscalar architectures. In *Proceedings of RISC '92* (Feb. 1992).

[21] RAU, B. Data-flow and dependence analysis for instruction-level parallelism. *Lecture Notes in Computer Science 589* (1991), 236–250. Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing.

[22] RAU, B. R., LEE, M., TIRUMALAI, P. P., AND SCHLANSKER, M. S. Register allocation for software pipelined loops. *SIGPLAN Notices 27*, 7 (July 1992), 283–299. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[23] SETHI, R., AND ULLMAN, J. The generation of optimal code for arithmetic expressions. *Journal of the ACM 17*, 4 (October 1970), 715–728.

[24] WOLF, M. E., AND LAM, M. S. A data locality optimizing algorithm. *SIGPLAN Notices 26*, 6 (June 1991), 30–44. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[25] WOLFE, M. Loop skewing: The wavefront method revisited. *Journal of Parallel Programming* (1986).

A Matrix Multiply – Before and After

A.1 Original Matrix Multiply

```
do j = 1, n
  do i = 1,n
    do k = 1,n
      c(i,j) = c(i,j)+a(i,k)*b(k,j)
    enddo
  enddo
enddo
```

A.2 Balanced Matrix Multiply

```
c
c   .. pre-loop removed
c
        do j = mod(n, 2)+1, n, 2
c
c   .. pre-loop removed
c
          do i = mod(n, 2)+1, n, 2
            if (1 .gt. n) goto 10003
            a$1$0 = a(i, 1)
            b$2$0 = b(1, j)
            c$0$0 = c(i, j)+a$1$0*b$2$0
            b$4$0 = b(1, j+1)
            c$3$0 = c(i, j+1)+a$1$0*b$4$0
            a$6$0 = a(i+1, 1)
            c$5$0 = c(i+1, j)+a$6$0*b$2$0
            c$7$0 = c(i+1, j+1)+a$6$0*b$4$0
            do k = 2, n
              a$1$0 = a(i, k)
              b$2$0 = b(k, j)
              c$0$0 = c$0$0+a$1$0*b$2$0
              b$4$0 = b(k, j+1)
              c$3$0 = c$3$0+a$1$0*b$4$0
              a$6$0 = a(i+1, k)
              c$5$0 = c$5$0+a$6$0*b$2$0
              c$7$0 = c$7$0+a$6$0*b$4$0
            enddo
            c(i+1, j+1) = c$7$0
            c(i+1, j) = c$5$0
            c(i, j+1) = c$3$0
            c(i, j) = c$0$0
10003       continue
          enddo
        enddo
```