Unroll-and-Jam Using Uniformly Generated Sets

Steve Carr
Department of Computer Science
Michigan Technological University
Houghton MI 49931-1295
carr@mtu.edu

Yiping Guan
Shafi Inc.
3637 Old US 23 Ste. 300
Brighton MI 48116
shafiinc@ismi.net

Abstract

Modern architectural trends in instruction-level parallelism (ILP) are to increase the computational power of microprocessors significantly. As a result, the demands on memory have increased. Unfortunately, memory systems have not kept pace. Even hierarchical cache structures are ineffective if programs do not exhibit cache locality. Because of this compilers need to be concerned not only with finding ILP to utilize machine resources effectively, but also with ensuring that the resulting code has a high degree of cache locality.

One compiler transformation that is essential for a compiler to meet the above objectives is unroll-and-jam, or outer-loop unrolling. Previous work either has used a dependence-based model [7] to compute unroll amounts, significantly increasing the size of the dependence graph, or has applied a more brute force technique [16]. In this paper, we present an algorithm that uses a linear-algebra-based technique to compute unroll amounts. This technique results in an 84% reduction over dependence-based techniques in the total number of dependences needed in our benchmark suite . Additionally, there is no loss in optimization performance over previous techniques and a more elegant solution is utilized.

1. Introduction

The power of the microprocessor has been dramatically improved through multiple instruction-issue in a single cycle and pipelined functional units. As a result, more operations can be performed per machine cycle. However, the speed of memory has not been increasing at the same rate, resulting in a memory bottleneck. Even with the use of memory hierarchy, poor cache performance, large memory latencies and limited bandwidth of memory systems are still causing idle computation cycles and empty pipeline stages.

One part of attacking these performance problems through compiler optimization is to match the ratio of memory operations to floating-point operations in a program loop (*loop balance*) to the optimum such ratio handled by a target machine (*machine balance*) with a transformation called unroll-and-jam (outer-loop unrolling) [8, 7]. Unroll-and-jam has been shown to be effective at lowering the difference between loop balance and machine balance. Speedups on the order of 20 are possible on nested loops while speedups on the order of 2 are frequent [7].

Previous work with unroll-and-jam has used the dependence graph to compute a formula by which loop balance can be predicted based upon unroll amounts [8, 7]. The problem with this approach is that it requires the computation and storage of input dependences to determine memory reuse [13]. Input dependences make up a large portion of the resulting dependence graph and are only needed for memory performance analysis. Therefore, time and space are wasted when the input dependences are not needed.

Wolf and Lam present a linear-algebra-based approach to memory-reuse analysis that obviates the need to compute and store input dependences [15]. In this paper, we show how to compute unroll amounts using their linear-algebra-based reuse analysis. This method will save a significant amount of dependence graph storage space and will eliminate the complicated special-case analysis of the dependence-based approach.

The rest of this paper begins with related work and background material. Then, we present our algorithm for computing loop balance and an experiment showing the savings in dependence graph space obtained by the linear algebra model. Finally, we present our conclusions and future work.

2. 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 [4]. Aiken and Nicolau discuss a transformation identical to unroll-and-

1072-4451/97 $10.00 © 1997 IEEE

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 unnecessarily restricts unroll amounts. Wolf and Lam present the framework for determining data locality that we use in this paper. They use loop interchange and tiling to improve locality [15]. They present unroll-and-jam in this context as register tiling, but they do not present a method to determine unroll amounts. In [8], a method that improves ILP by matching the resource requirements of a loop as closely as possible to the resources provided by a machine is presented. However, this work assumes that all memory references are cache hits. In [7], cache effects are added to the resource requirements of a loop, but dependence-analysis-based data-reuse analysis is used. Finally, Wolf, Maydan and Chen present a method similar to ours [16]. They include tiling and permutation in their method. We consider only unroll-and-jam in our work. However, Wolf, et al., unroll data structures, exhaustively trying each unroll amount and computing their performance metric for each potential new loop body. Instead, we directly precompute tables that do not require unrolling a data structure and give a more elegant solution to the problem. Additionally, Wolf, et al., do not include cache effects when performing unroll-and-jam. Our method can possibly be substituted for their technique of computing unroll amounts within their optimization framework.

3. Background

In this research, we assume a traditional highly optimizing scalar compiler for an instruction-level parallel (ILP) target machine (e.g., DEC Alpha). To estimate the utilization of available ILP in loops under this assumption, we use the notion of balance defined previously [4, 7].

3.1. 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, we define β_M 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. So, $\beta_M = \frac{M_M}{F_M}$. 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.

3.2. Loop Balance

Just as machines have a balance ratio, so do loops. *Loop balance* is defined by Callahan, et al., to be the ratio of the number of memory operations issued, M_L, to the number of floating-point operations issued, F_L[1]. Since M_L assumes that all loads are cache hits, we get a poor estimation of the memory requirements of the loop. To include cache misses, we add to the number of memory operations the cache miss penalty for each load that is determined to be a cache miss by reuse analysis [13, 15]. This allows us to charge for the delay slots introduced by the cache miss.

Since some architectures allow cache miss latency to be hidden either via non-blocking loads or software prefetching, our model is designed to handle the case where 0 or more cache miss penalties can be eliminated. To accomplish this we assume that an architecture has a prefetch-issue buffer size of $P_M \geq 0$ instructions and a prefetch latency of $L_M > 0$ cycles. This gives a prefetch issue bandwidth of $I_M = \frac{P_M}{L_M}$. Since an effective algorithm for software prefetching has already been developed by Mowry, *et al.*, we will use it to help model the prefetching bandwidth requirements of a loop [14]. Essentially, only those array references that are determined to be cache misses in the innermost loop by the data-reuse model will be prefetched. An innermost loop requires P_L prefetches every L_L cycles (where L_L is the number of cycles needed to execute one iteration of the loop) to hide main memory latency, giving an issue-bandwidth requirement of $I_L = \frac{P_L}{L_L}$. If $I_L \leq I_M$, then main memory latency can be hidden. However, if $I_L > I_M$, then $P_L - I_M L_L$ prefetches cannot be serviced. Assuming that prefetches are dropped if the prefetch buffer is full, then the prefetches that cannot be serviced will be cache misses. To factor this cost into the number of memory accesses in a loop, an unserviced prefetch will have the additional cost of the ratio of the cost of a cache miss, C_m, to the cost of a cache hit, C_h, memory accesses. This is expressed as follows:

$$\beta_L = \frac{M_L + (P_L - I_M L_L)^+ \times \frac{C_m}{C_h}}{F_L}$$

where

$$x^+ = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

If an architecture does not have a prefetch buffer, we can set $I_M = 0$ and the formulation will still incorporate cache misses into the computation of loop balance.

Comparing β_M to β_L can give us 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, as a result, idle computational cycles will exist. Such loops are called *memory bound*.

[1] In this work, we rely on the method in [4] to remove the effect of recurrences.

The performance of memory-bound loops can be improved by lowering β_L with unroll-and-jam and scalar replacement [8, 7]. In this paper, we only address the improvement of memory-bound loops.

3.3. Using Balance to Optimize Loops

Unroll-and-jam is a transformation that can be used to improve the performance of memory-bound loops by lowering loop balance [2, 4, 8, 7]. Additional computation can be introduced into an innermost loop body without a proportional increase in memory references. For example, the loop:

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

after unroll-and-jam of the J-loop becomes:

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

The original loop has one floating-point operation and one memory reference (A(J) can be held in a register), giving a balance of 1 (excluding cache effects). After applying unroll-and-jam, the loop has two floating-point operations and one memory reference (A(J), A(J+1), and the second load of B(I) can be held in registers). This gives a balance of 0.5. On a machine with $\beta_M = 0.5$, the second loop performs better. Previous work has shown that using the following objectives to guide unroll-and-jam is effective at improving ILP in the innermost loop [8, 7].

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

If these goals are expressed mathematically, the following integer optimization problem results:

objective function: $\min |\beta_L - \beta_M|$
constraint: $R_L \leq R_M$

where the decision variables in the problem are the unroll amounts for each of the loops in a loop nest and R_L and R_M are the number of registers required by the loop and provided by the machine, respectively. For each loop nest within a program, we model its possible transformation as a problem of this form. Solving it will give us the unroll amounts to balance the loop nest as much as possible.

For the purposes of this paper, we assume that the safety of unroll-and-jam is determined before we attempt to optimize loop balance. The amount of unroll-and-jam that is determined to be safe is used as an upper bound. A detailed description of how safety is determined and its effect on the limiting of unroll amounts can be found elsewhere [4].

3.4. Data Reuse

To compute the cost of a memory operation, this paper uses the linear algebra model of Wolf and Lam [15]. This section describes the data reuse model that they have developed.

The two sources of data reuse are *temporal* reuse, multiple accesses to the same memory location, and *spatial* reuse, accesses to nearby memory locations that share a cache line or a block of memory at some level of the memory hierarchy. Temporal and spatial reuse may result from *self-reuse* from a single array reference or *group-reuse* from multiple references [15]. Without loss of generality, we assume Fortran's column-major storage.

In Wolf and Lam's model, a loop nest of depth n corresponds to a finite convex polyhedron Z^n, called an iteration space, bounded by the loop bounds. Each iteration in the loop corresponds to a node in the polyhedron, and is identified by its index vector $\vec{x} = (x_1, x_2, \ldots, x_n)$, where x_i is the loop index of the i^{th} loop in the nest, counting from the outermost to the innermost. The iterations that can exploit reuse are called the localized iteration space, L. The localized iteration space can be characterized as a localized vector space if we abstract away the loop bounds.

For example, in the following piece of code, if the localized vector space is *span*{(1,1)} , then data reuse for both A(I) and A(J) are exploited.

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

In Wolf and Lam's model, data reuse can only exist in *uniformly generated* references as defined below [10].

Definition 1 *Let n be the depth of a loop nest, and d be the dimensions of an array* A. *Two references* $A(f(\vec{x}))$ *and* $A(g(\vec{x}))$, *where f and g are indexing functions* $Z^n \rightarrow Z^d$, *are* uniformly generated *if*

$$f(\vec{x}) = H\vec{x} + \vec{c}_f \text{ and } g(\vec{x}) = H\vec{x} + \vec{c}_g$$

where H is a linear transformation and $\vec{c}_f$ and $\vec{c}_g$ are constant vectors.

For example, in the following loop,

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

the references can be written as $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} I \\ J \end{bmatrix}+\begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} I \\ J \end{bmatrix}+\begin{bmatrix} 0 \\ 1 \end{bmatrix}$, and $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\begin{bmatrix} I \\ J \end{bmatrix}+\begin{bmatrix} 0 \\ 2 \end{bmatrix}$. References in a loop nest are partitioned into different sets, each of which operates on the same array and has the same H. These sets are called *uniformly generated sets* (UGSs).

A reference is said to have *self-temporal* reuse if $\exists \vec{r} \in L$ such that $H\vec{r} = \vec{0}$. The solution(s) to this equation is called the self-temporal reuse vector space or R_{ST}. A reference has *self-spatial* reuse if $\exists \vec{r} \in L$ such that $H_S\vec{r} = \vec{0}$, where H_S is H with the first row set to $\vec{0}$. The solution(s) to this equation is called the self-spatial reuse vector space or R_{SS}. Two distinct references in a UGS, $\mathrm{A}(H\vec{x}+\vec{c}_1)$ and $\mathrm{A}(H\vec{x}+\vec{c}_2)$ have *group-temporal* reuse if $\exists \vec{r} \in L$ such that $H\vec{r} = \vec{c}_1 - \vec{c}_2$. And finally, two references have *group-spatial* reuse if $\exists \vec{r} \in L$ such that $H_S\vec{r} = \vec{c}_{1,S} - \vec{c}_{2,S}$.

Using the above equations we can partition the UGSs into sets whose members exhibit group-temporal reuse (GTSs) and group-spatial reuse (GSSs). The *leader* of one of these sets is the first reference in the set to access a particular data element or cache line. The number of GTSs is denoted by g_T and the number of GSSs is denoted by g_S. Wolf and Lam give the following formula for the number of memory accesses per iteration for a uniformly generated set, given a localized iteration space L and cache-line size l:

$$\frac{g_S + (g_T - g_S)/l}{l^e s^{\dim(R_{ST}\cap L)}} \tag{1}$$

where

$$e = \begin{cases} 0 & R_{ST}\cap L = R_{SS}\cap L \\ 1 & otherwise \end{cases}$$

The total number of memory accesses in L is the sum of the accesses for each uniformly generated set.

3.5. SIV References

In this paper, we will concentrate on array references that have a single induction variable (SIV) in each subscript position. In addition, we require each subscript to be fully separable, *i.e.* each induction variable appears at most once in any array reference [11]. In terms of H, each row and column must have at most one non-zero value. These criteria may appear to be very restrictive. However, previous work has shown that on loops where unroll-and-jam is applicable nearly all array references fit these criteria [8]. Algorithms to handle more complicated subscripts can be found in [12].

4. Computing Unroll Amounts

In this section, we detail our computation of β_L and R_L using the reuse model of Wolf and Lam [15]. We will show how to pre-compute a matrix of coefficients that can be used to give β_L and R_L based upon unroll amounts for a set of loops.

4.1. Expressing Unroll Amounts

In this work, we express the unroll amounts for a set of loops as an *unroll vector* $\vec{u} = (u_1, u_2, \ldots, u_n)$ where u_i is the unroll amount for the i^{th} loop in a nest counting from outermost to innermost. Note that u_n will always be 0 as we do not consider unrolling the innermost loop. The set of all unroll vectors is called the *unroll space*, U. In this work, U is bounded by R_M in each dimension.

Given an array reference with a subscript of the form $H\vec{i} + c$, where $\vec{i}$ is the vector of induction variables, unroll-and-jam by $\vec{u}$ creates a number of new references with the subscript functions $H\vec{i} + H\vec{u'} + c$ for each $\vec{u'} \le \vec{u}$. Here $\vec{u'} \le \vec{u}$ implies that $u'_i \le u_i, 1 \le i \le n$. Given a localized vector space L, unroll-and-jam within L will not increase cache reuse. So, in the following discussion we will assume that $U \notin L$. To accomplish this, for each non-zero row in L, we set the corresponding rows to $\vec{0}$ in each of H and c.

4.2. Computing P_L

Assuming that we would like to prefetch every main memory access so that we have no cache misses, P_L is simply the result of Equation 1. To compute Equation 1 given an unroll vector $\vec{u}$, we need to know how many GSSs and GTSs will exist after unrolling by $\vec{u}$. We can pre-compute this value for each UGS and unroll vector and store that value in a table. Later we can use the table to determine the input to Equation 1 given an unroll vector. Figure 2 gives the algorithm for computing the table for the number of GTSs and Figure 3 give the algorithm for computing the table for the number of GSSs.

The key to the computation is determining when a GTS (or GSS) that is a created by unroll-and-jam merges with a previously existing GTS (or GSS) because there is locality between the two sets within the localized vector space. When computing the number of GTSs and GSSs after unroll-and-jam, we need only consider the merger of two leaders into the same group. The merger of two leaders will also indicate the merger of two entire groups. Each member of a copy of a group created by unroll-and-jam will have the same H and their respective constant vectors will be changed by the same ratio. So, the copies will all belong to the same group (see [12] for a proof.)

```
DO I = 1, N
  DO J = 1, N
    A(I,J) = A(I-2,J)
```

```
DO I = 1, N, 4
  DO J = 1, N
    A(I,J) = A(I-2,J)
    A(I+1,J) = A(I-1,J)
    A(I+2),J) = A(I,J)
    A(I+3,J) = A(I+1,J)
```

Figure 1. Example of Merging GTSs

The copies of two leaders in a UGS, f and g, with $c_f \leq c_g$ will belong to the same GTS for any unroll vector $\vec{u}$ such that $H\vec{u} \geq c_g - c_f$. For each $\vec{u}$ that satisfies this equation, each reference created from g, g', will have a corresponding reference created from f, f', such that $\exists\vec{r} \in L | H\vec{r} = c_{g'} - c_{f'}$. In the algorithms in Figures 2 and 3, we use this information to determine the point at which further unrolling will no longer introduce new GTSs and GSSs, respectively. Essentially, any $\vec{v} \geq \vec{u}$ will cause no increase in GTSs and GSSs.

For example, in Figure 1, before unrolling the I-loop there are two GTS leaders, A(I,J) and A(I-2,J), if we consider only the innermost loop as the localized vector space. Using the above formula we get $\vec{u} = (2,0)$. So, any unroll vector greater than or equal to (2,0) will not introduce a new GTS for copies of A(I-2,J). As can be seen in the example, the reference created with such unroll vectors, the reads from A(I,J) and A(I+1,J), belong to the same GTS as the original store to A(I,J) or one of its copies (in this case A(I+1,J)).

In the algorithm in Figure 2, we compute the number of new GTSs that will exist in an unrolled loop, given a specific unroll vector. The algorithm begins by ordering all of the leaders of GTSs from the first (earliest) one to access the set of values referenced by the UGS to the last. In function ComputeTable, we initialize each entry of the table to the original number of GTSs. This table then contains the number of new GTSs that are created due to a particular unroll vector if no merging of GTSs occurs. We then consider each pair of GTSs, starting with the earliest and comparing with those that occur later in loop execution, to determine at what point merging occurs, $\vec{v}_{i,j}$. We call the earliest leader under consideration for merging the *superleader*. If $\vec{v}_{i,j}$ is in the unroll space, we reduce the number of GTSs created for each point between the newly computed value and the point where this GTS merged with the previous superleader. When no previous superleader exists (*i.e.*, $i = 1$), we use the lexicographically largest unroll vector as the upper bound.

The algorithm in Figure 3 for computing the number of GSSs is similar to the algorithm in Figure 2. The only difference is the use of H_S rather than H.

```
function ComputeTable($H, Temp, S$)
  $Temp = |S|$
  for $i = 1$ to $|S|$ do
    for $j = i$ to $|S|$ do
      solve $H\vec{v}_{i,j} = c_i - c_j$
      if $\vec{v}_{i,j} \in U_L$ then
        foreach $\vec{v}_{i,j} \leq \vec{x} \leq \vec{v}_{i-1,j}$
          $Temp[\vec{x}]--$
end ComputeTable

function Sum($Temp$)
  $Table = \vec{0}$
  foreach $\vec{u} \in U$ do
    foreach $\vec{v} \leq \vec{u}$ do
      $Table[\vec{u}] += Temp[\vec{v}]$
  return $Table$
end Sum

function ComputeGTSTable($UGS$)
  foreach $u \in UGS$ do
    $GTSL$ =Order(GTS leaders)
    ComputeTable($H, Table, GTSL$)
  return Sum($Table$)
end ComputeGTSTable
```

Figure 2. Computing GTSTable

```
function ComputeGSSTable($UGS$)
  foreach $u \in UGS$ do
    $GSSL$ =Order(GSS leaders)
    ComputeTable($H_S, Table, GSSL$)
  return Sum($Table$)
end ComputeGSSTable
```

Figure 3. Computing GSSTable

4.3. Computing M_L

To compute M_L, we must compute how many array references will be removed by scalar replacement (held in a register) after unroll-and-jam. This is similar to computing the number of GTSs after unroll-and-jam if the localized vector space is the innermost loop only. However, it is possible that there is a definition in a GTS that precludes references from being scalar replaced [3, 9]. In the following code,

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

all of the references to A are in the same GTS. However, The definition of `A(I,J)` keeps `A(I+1,J)` from being utilized in scalar replacement.

To account for the above condition, we compute M_L on register-reuse sets (RRS). A register-reuse set is a set of references that uses the same set of values during the execution of the innermost loop. A GTS can be split up into multiple RRSs. Essentially, the GTS is ordered from the earliest reference to the latest as was done with leaders previously. A new RRS is created and references are added to that RRS until a definition is encountered. At that point a new RRS is created and the process continues. In the previous example, a new RRS is created after processing `A(I+1,J)` when the definition of `A(I,J)` is encountered. The algorithm for computing RRSs is listed in Figure 4.

```
function ComputeRRS($UGS$)
  foreach $u \in UGS$ do
    foreach $g \in GTS$ do
      $g$ = Order($g$)
      $R$ = new $RRS$
      $i = 0$
      while $++i < |g|$ do
        if $g[i]$ is a def then
          $R$ = new $RRS$
        $R \cup= g[i]$
  return $R$
end ComputeRRS
```

Figure 4. Computing Register Reuse Sets

Because the reuse of a value does not cross a definition, copies of two RRSs cannot be merged after unroll-and-jam unless the leader of the later RRS is not a definition (this can only happen between GTSs). So, the RRS leaders are split into mergeable register-reuse set (MRRS) leaders of size $\geq$ 1. The algorithm for M_L is then the same as the algorithm for GTSs except only leaders in the same MRRS can be merged. Figure 5 shows the algorithm.

```
function ComputeRRSTable($UGS$)
  foreach $u \in UGS$ do
    $RRSL$ =Order($RRS$ leaders)
    split $RRSL$ into $MRRSL$
    foreach $m \in MRRSL$
      ComputeTable($H$, $Table$, $m$)
  return Sum($Table$)
end ComputeRRSTable
```

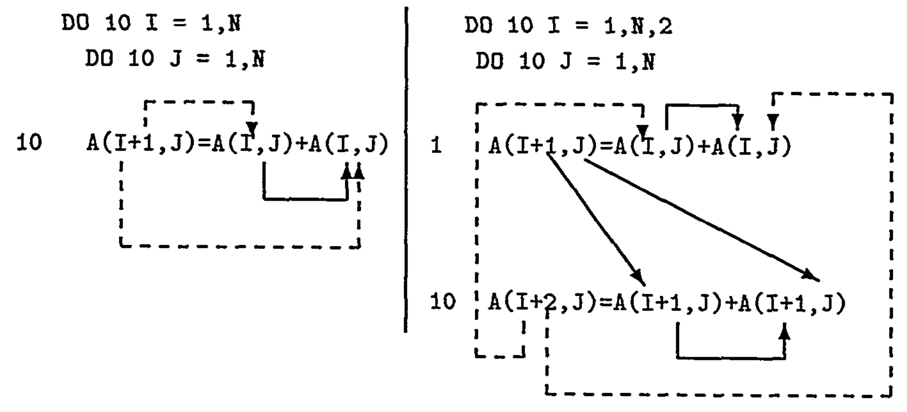
Figure 5. Computing RRSTable

4.4. Computing R_L

To compute R_L, we need to compute how many registers each RRS will need after unroll-and-jam. The algorithm for computing R_L is found in Figure 7. In this algorithm, we use each member of the RRSs, not just the leaders, to measure the number of registers needed. The superleader of a MRRS is the source of the value that flows through the set. However, the value from the superleader may cross outer loop iterations. Since scalar replacement is only amenable to innermost reuse, the value from the superleader may not provide the value for a scalar-replaced reference until enough unrolling has occurred to ensure that the reuse occurs only across the innermost loop.

Consider the example in Figure 6 (reuse within a loop iteration is denoted with solid arrows, reuse across loop iterations is denoted with dashed arrows). In the original loop, there are three references in the MRRS: `A(I+1,J)`, `A(I,J)` and `A(I,J)`. Before unrolling, the superleader of the MRRS, `A(I+1,J)`, does not provide the value to scalar replace the second reference to `A(I,J)` in the loop. In this case, the first reference to `A(I,J)` provides the value. However, after unrolling the `I`-loop by 1, the superleader provides the value for scalar replacement to the copies of `A(I,J)` – that is, the references to `A(I+1,J)` in statement 10 in the unrolled loop. Therefore, we must consider when a non-superleader provides a value for scalar replacement.

In the algorithm of Figure 7, this is accomplished by adding in register pressure only for unroll amounts in between where the current superleader introduces register pressure for another leader, $\vec{u}_{i,j}$, and the point where the previous superleader introduced register pressure, $\min(\vec{u}_{i-1,j}, \vec{u}_{i,j+1})$. Here, if the subscripts are out of the bounds of 1 to m, the value in each vector element is R_M.



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

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

Figure 6. Multiple Generators in One Partition

```
function ComputeRLTable($UGS$)
  foreach $u \in UGS$ do
    $RRS$ = Order($RRS$s)
    split $RRS$ into $MRRS$
    foreach $m \in MRRS$ do
      for $i = 1$ to $|m|$ do
        for $j = |m|$ downto $i$ do
          solve $H\vec{u}_{i,j} = c_i - c_j$
          if $\vec{u}_{i,j} \in U$ then
            foreach $\vec{u}_{i,j} \leq \vec{x} \leq \min(\vec{u}_{i-1,j}, \vec{u}_{i,j+1})$
              $Table[\vec{x}] \mathrel{+}= c_{i,n} - c_{j,n}$
  return Sum($Table$)
end ComputeRLTable
```

Figure 7. Computing RLTable

4.5. Choosing Unroll Amounts

In practice, we limit unroll-and-jam to at most 2 loops. So, we pick the two loops with the best locality as measured by Equation 1 to unroll and then construct the tables, as described above, for those loops. Next we search the entire solution space for the unroll amounts that give the best balance and satisfy the register constraint. Given that we bound the solution space by R_M, we can search for the solution, once we get the tables constructed, in $O(R_M^2)$.

5. Experiment

We have implemented a simplified version of the previous algorithm [12] in Memoria [6], a source-to-source Fortran converter based upon the ParaScope programming environment [5]. The implementation differs from the algorithm in this paper only when multiple loops must be unrolled to get inner-loop reuse of a particular reference. This case did not appear in our testing, ensuring that our simplifications did not affect the results. The experiment consists of two parts: (1) an analysis of the savings in dependence graph size due to the lack of input dependences, and (2) an evaluation of the run-time improvement due to the algorithm.

5.1. Dependence Graph Savings

We ran 1187 routines from SPEC92, Perfect, NAS and local benchmark suites through Memoria and counted the number of input dependences and total dependences. Only 649 of those routines actually had dependences in them and we base our statistics on these. We found that in the loops that contained dependences a total of 84% of the 305,885 dependences were input dependences. On the average 55.7% (or 398) of the dependences in a routine were input dependences. The standard deviation for both of these numbers was quite large. The average percentage had a standard deviation of 33.6 and the average number of input dependences had a standard deviation of 3533.

Figure 8 shows the number of routines with a particular percentage of input dependences. In 74% of the routines at least one-third of the dependences were input, while in 53% of the routines at least 50% of the dependences were input. 25% of the routines have at least 90% input dependences.

While these statistics show large variance, they still illuminate the fact that input dependences often make up a significant portion of the dependence graph. Removing these dependences not only cuts down on the space required for a dependence graph but also reduces processing time when the dependence graph must be updated after loop transformations. We believe that this is significant enough to warrant using the linear algebra model.

5.2. Execution Time

We tested our algorithm on a set of loops found in the SPEC92, Perfect, NAS and local benchmark suites. The loops are chosen from those within the suite that are not already balanced and those on which unroll-and-jam is legal.

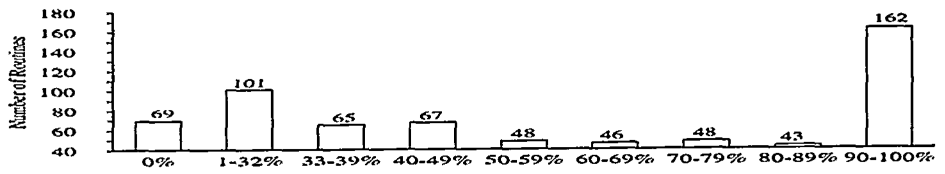


Figure 8. Percentage of Input Dependences

The test loops are listed in Table 1. The "Loop" column gives the name of the loop and an optional number. The number corresponds to the textual order of loops in the corresponding subroutine. The "Description" column gives the suite/benchmark/subroutine of the loop or a short description.

Loop Num	Loop	Description
1	jacobi	Compute Jacobian of a Matrix
2	afold	Adjoint Convolution
3	btrix.1	SPEC/NASA7/BTRIX
4	btrix.2	SPEC/NASA7/BTRIX
5	btrix.7	SPEC/NASA7/BTRIX
6	collc.2	Perfect/FLO52/COLLC
7	cond.7	local/SIMPLE/CONDUCT
8	cond.9	local/SIMPLE/CONDUCT
9	dflux.16	Perfect/FLO52/DFLUX
10	dflux.17	Perfect/FLO52/DFLUX
11	dflux.20	Perfect/FLO52/DFLUX
12	dmxpy0	Vector-Matrix Multiply
13	dmxpy1	Vector-Matrix Multiply
14	gmtry.3	SPEC/NASA7/GMTRY
15	mmjik	Matrix-Matrix Multiply
16	mmjki	Matrix-Matrix Multiply
17	vpenta.7	SPEC/NASA7/VPENTA
18	sor	Successive Over Relaxation
19	shal	Shallow Water Kernel

Table 1. Description of Test Loops

Our experiments showed that the uniformly generated set model presented in this paper gives the same performance improvement as the dependence based model [12]. We include the graphs from previous work in Figures 9 and 10 for inspection and refer the reader to that paper for a more detailed discussion[7]. This result shows that we can remove the storage of input dependences without a loss in optimization performance.

5.3. Comparison with Wolf, *et al.*

Wolf, *et al.*, include unroll-and-jam in a set of transformations that they consider while optimizing for cache and ILP together [16]. They present a comparison with our work that contains a flaw. The loops that they used in their experiment were from a preliminary version of our implementation that did not limit register pressure [12]. If those loops had been optimized considering register pressure, our method would have faired much better. We do note, however, that they perform unroll-and-jam at the intermediate code level and have a better estimation of register pressure, likely giving them a performance edge.

6. Conclusion

In this paper, we have presented a method for computing unroll amounts for unroll-and-jam using uniformly generated sets to compute loop balance. This method saves in the storage of input dependences over previous dependence-based techniques [8, 7]. Our results show that our technique saves an average of 55.6% of the space needed for a dependence graph for a routine and a total of 84% of the space needed for all of the dependences in the suite. Not only is space reduced, but also the processing time of dependence graphs is reduced for transformations that update the dependence graph.

In the future, we will look into the effects of our optimization technique on architectures that support software prefetching since our performance model handles this. We will also examine the performance of unroll-and-jam on architectures with larger register sets so that the transformation is not as limited. We are currently developing compiler and architecture-simulation tools to allow us to perform this research.

Given that the trend in machine design is to have increasingly complex memory hierarchies to support increasing degrees of ILP, compilers will need to adopt more sophisticated memory-management and parallelism-enhancing transformations to generate fast and efficient code. The optimization method presented in this paper is a step in that direction.

Acknowledgments

This work has been supported by the National Science Foundation under grant CCR-9409341. Wei Li provided the Lambda library for our matrix algebra computations. Richard Schooler provided valuable insight into the linear-algebra-based reuse analysis.

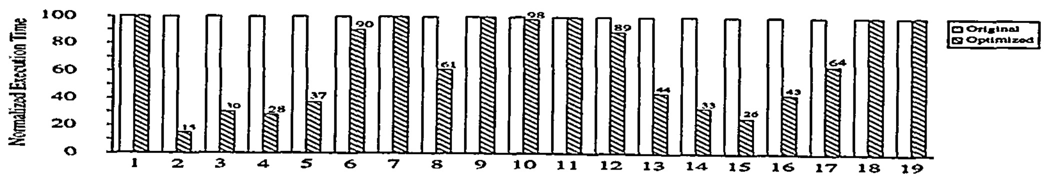


Figure 9. Performance of Test Loops on DEC Alpha

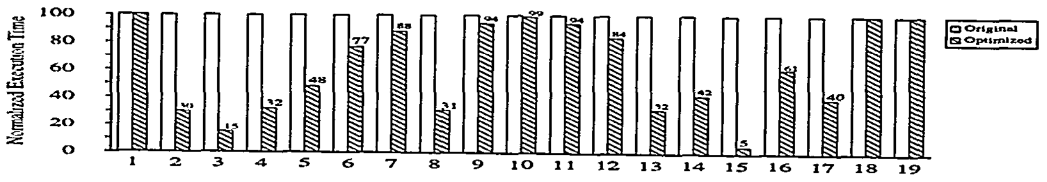


Figure 10. Performance of Test Loops on HP PA-RISC

References

[1] A. Aiken and A. Nicolau. Loop quantization: An analysis and algorithm. Technical Report 87-821, Cornell University, March 1987.

[2] F. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[3] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[4] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5:334–358, 1988.

[5] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*, Athens, Greece, June 1987.

[6] S. Carr. *Memory-Hierarchy Management*. PhD thesis, Rice University, Department of Computer Science, September 1992.

[7] S. Carr. Combining optimization for cache and instruction-level parallelism. In *Proceedings of the 1996 Conference on Parallel Architectures and Compiler Techniques*, pages 238–247, Boston, MA, October 1996.

[8] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[9] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software Practice and Experience*, 24(1):51–77, Jan. 1994.

[10] D. Gannon, W. Jalby, and K. Gallivan. 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.

[11] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. *SIGPLAN Notices*, 26(6):15–29, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[12] Y. Guan. Unroll-and-jam guided by a linear-algebra-based reuse model. Master's thesis, Michigan Technological University, Dec. 1995.

[13] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, 1996.

[14] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–75, Boston, Massachusetts, 1992.

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

[16] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Twenty-Ninth Annual Symposium on Micorarchitecture (MICRO-29)*, Dec. 1996.