Unroll-and-Jam Using Uniformly Generated Sets*

Steve Carr† Yiping Guan‡

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 has either used a dependence based model [1] to compute unroll amounts, significantly increasing the size of the dependence graph, or has applied brute force techniques [2]. In this paper, we present an algorithm that uses linear-algebra-based techniques to compute unroll amounts that save 84% of the dependence-graph space needed by dependence-based techniques, while retaining performance, and that gives a more elegant solution.

1 Introduction

The power of the microprocessor has been dramatically improved through multiple instructions issued in a single cycle, and pipelined functional units. As a result, more operations can be performed per machine cycle than previously. In the mean time, 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) [3, 1]. 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 loops while speedups on the order of 2 are common [1].

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 [3, 1]. The problem with this

*This work was supported by the National Science Foundation under grant CCR-9409341.
†Department of Computer Science, Michigan Technological University, Houghton MI 49931-1295, carr@mtu.edu.
‡Shafi Inc., 3637 Old US 23 Ste. 300, Brighton MI 48116, shafiinc@isma.net.
approach is that it requires the computation and storage of input dependences to determine memory reuse [4]. 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 [5]. In this paper, we show how to compute unroll amounts using the 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 [6]. Aiken and Nicolau discuss a transformation identical to unroll-and-jam called loop quantization [7]. 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 do not control register pressure. 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 [5]. They present unroll-and-jam in this context as register tiling, but they do not present a method to determine unroll amounts. In [3], 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 [3]. In [1], 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 [2]. They include tiling and permutation in their method. We just consider unroll-and-jam in our work. However, they unroll data structures, exhaustively trying each unroll amount and computing their performance metric for each potential new loop body. 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 be substituted for their brute force 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 these assumptions, we use the notion of balance defined previously [6, 1].
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 $\beta_M$ 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 $\beta_M$.

3.2 Loop Balance

Just as machines have a balance ratio, so do loops. Loop balance is defined to be the ratio of the number of memory operations issued, $M_L$, to the number of floating-point operations issued, $F_L$. To account for cache misses, we add to the number of memory operations the cache miss penalty. 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. For an architecture with a prefetch-issue bandwidth of $I_M$ and a loop that executes for $L_L$ cycles and needs to issue $P_L$ prefetches, $(P_L - I_M L_L)^+$ of these prefetches will not be serviced, assuming we drop the prefetches that cannot be issued. An unserviced prefetch will be a cache miss having the cost of the ratio of a cache miss penalty, $C_m$, to the cache access cost, $C_h$. Thus, $\beta_L$ is expressed as follows:

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

Comparing $\beta_M$ to $\beta_L$ 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. Performance can be improved by lowering $\beta_L$ with unroll-and-jam and scalar replacement [3, 1]. In this paper, we will only address the improvement of loops that fit this criterion.

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 [8, 6, 3, 1]. Additional computation can be introduced into an innermost loop body without a proportional increase in memory references. For example, the loop:

\[G\]
DO 10 J = 1, 2*N
  DO 10 I = 1, M
10   A(J) = A(J) + B(I)

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

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

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. 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 [3, 1].

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 [6].

### 3.4 Data Reuse

To compute the cost of a memory operation, this paper uses the linear algebra model of Wolf and Lam [5]. This section describes the data reuse model 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 [5]. 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 \( \bar{x} = (x_1, x_2, \ldots, x_n) \), 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 \( \text{span}\{(1,1)\} \), then data reuse for both \( A(I) \) and \( A(J) \) are exploited.

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

Data reuse may exist on memory references that are called uniformly generated [9].

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

\[
f(\bar{x}) = H\bar{x} + \bar{c}_f \quad \text{and} \quad g(\bar{x}) = H\bar{x} + \bar{c}_g
\]

where \( H \) is a linear transformation and \( \bar{c}_f \) and \( \bar{c}_g \) are constant vectors.

For example, in the following loop,

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

the references can be written as

\[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
I \\
J
\end{bmatrix} + \begin{bmatrix}
0 \\
1
\end{bmatrix}
\begin{bmatrix}
I \\
J
\end{bmatrix} + \begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
I \\
J
\end{bmatrix} + \begin{bmatrix}
0 \\
2
\end{bmatrix}.
\]

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

Using the above equations we can partition the UGSs into sets whose members exhibit group-temporal reuse (GTSs) and group-spatial reuse (GSSs). The number for 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 \( l \):
\[ \frac{g_S + (g_T - g_S)/l}{l \in \text{dim}(R_{ST} \cap L)} \]  

where
\[
\ell = \begin{cases} 
0 & R_{ST} \cap L = R_{SS} \cap L \\
1 & \text{otherwise} 
\end{cases}
\]

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 [10]. 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 [3]. Algorithms to handle more complicated subscripts can be found in [11].

### 4 Computing Unroll Amounts

In this section, we detail our computation of \( \beta_L \) and \( R_L \) using the reuse model of Wolf and Lam [5]. We will show how to pre-compute a matrix of coefficients that can be used to give \( \beta_L \) 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 of 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 for new references with the subscript functions \( H \vec{i} + H \vec{u} \dot{\vec{u}} + c \) for each \( \vec{u} \leq \vec{u} \). Here \( \vec{u} \leq \vec{u} \) implies that \( u'_i \leq u_i, 1 \leq i \leq n \). Given a localized iteration space \( L \), unroll-and-jam within \( L \) will not increase cache reuse. So, in the following discussion we will assume that \( U \not\subseteq L \). To accomplish this, for each non-zero row in \( L \), we set the corresponding rows to \( \vec{u} \) 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 \( \vec{u} \),
Figure 1: Example of Merging GTSs

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 iteration 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 [11] for a proof.)

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 \( \exists \vec{r} \in L | H\vec{r} = c_{g'} - c_{g'} \). 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 iteration 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 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 in lexicographically increasing order. 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 lexicographically smallest and comparing with those greater than or equal to it, to determine at what point merging occurs, \( v_{i,j} \). We call the lexicographically smallest leader under consideration for merging the superleader. If \( v_{i,j} \) is in the unroll space, we reduce the number of GTSs created for each point between the newly computed value and the point where
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
    end ComputeTable
end ComputeTable

function Sum($Temp$)

$Table = 0$

foreach $\vec{u} \in U$ do
    foreach $\vec{v} \leq \vec{u}$ do
        $Table[\vec{u}] += Temp[\vec{v}]$
    end foreach
end foreach

return $Table$
end Sum

function ComputeGTSTable($UGS$)

foreach $u \in UGS$ do
    $GTS_L = \text{Lex(GTS leaders)}$
    $Table = ComputeTable(H, Table, GTS_L)$
end foreach

return Sum($Table$)
end ComputeGTSTable

Figure 2: Computing GTSTable
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.

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

Figure 3: Computing GSSTable

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 $H$.

4.3 Computing $M_L$

To compute $M_L$, we must compute how many array references will be removed by scalar replacement after unroll-and-jam. This is similar to computing the number of GTSs after unroll-and-jam if the localized iteration space is the innermost loop only. However, it is possible that there is more than one definition in a GTS and that scalar replacement will not remove definitions to the same array location [12, 13].

To account for this 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 in lexicographically increasing order. 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. The algorithm can be found in Figure 4.

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 lexicographically greater 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 $\geq 1$ and in computing the table we only consider merging copies of RRSs in the same MRRS. The algorithm for computing the table of values for $M_L$ based upon unroll vectors is in Figure 5.

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
function ComputeRRS(UGS)
    foreach u ∈ UGS do
        foreach g ∈ GTS do
            g = Lex(g)
            R = new RRS
            i = 0
            while ++ i < |g| do
                if g[i] is a def then
                    R = new RRS
                    R ∪= g[i]
            return R
        end
    end
end ComputeRRS

Figure 4: Computing Register Reuse Sets

function ComputeRRSTable(UGS)
    foreach u ∈ UGS do
        RRS L =Lex(RRS leaders)
        split RRS L into MRRSL
        foreach m ∈ MRRSL
            ComputTable(H, Table, m)
        return Sum(Table)
    end
end ComputeRRSTable

Figure 5: Computing RRSTable
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.

```plaintext
function ComputeRLTable(UGS)
    foreach u ∈ UGS do
        RRS = Lex(RRSs)
        split RRS into MRRS
        foreach m ∈ 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}] = \(c_{i,n} - c_{j,n}\)
                    end
            end
        end
    end
    return Sum(Table)
end ComputeRLTable
```

Figure 7: Computing RLTable

In the algorithm in Figure 7, this is accomplished by only adding in register pressure for unroll
Table 1: Percentage of Input Dependences

<table>
<thead>
<tr>
<th>Range</th>
<th>Number of Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td>69</td>
</tr>
<tr>
<td>1%-32%</td>
<td>101</td>
</tr>
<tr>
<td>33%-39%</td>
<td>65</td>
</tr>
<tr>
<td>40%-49%</td>
<td>67</td>
</tr>
<tr>
<td>50%-59%</td>
<td>48</td>
</tr>
<tr>
<td>60%-69%</td>
<td>46</td>
</tr>
<tr>
<td>70%-79%</td>
<td>48</td>
</tr>
<tr>
<td>80%-89%</td>
<td>43</td>
</tr>
<tr>
<td>90%-100%</td>
<td>162</td>
</tr>
</tbody>
</table>

amounts in between where the current leader being considered as superleader introduces register pressure for another leader, $u_{i,j}$ in the algorithm, and the point where the next oldest leader introduced register pressure, $\min(u_{i-1,j}, u_{i,j+1})$ in the algorithm. Here, if the subscripts are out of the bounds of 1 to $m$, the value in each vector element is $R_M$.

### 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 [11]. The implementation only differs from the algorithm in this paper 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.

Table 1 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 vary by large amounts, they still illuminate the fact that a good percentage of the time input dependences 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 significant enough to warrant not using input dependences.

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. The test loops are listed in Table 2. 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.

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

Table 2: 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. We include the graphs from
previous work in Figures 8 and 9 for inspection and refer the reader to that paper for a more
detailed discussion[1]. Note that the “No Cache” performance data uses the model described in [3]
that assumes everything is a cache hit.

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 [2]. 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 [11]. If those loops had been optimized considering register pressure,
we would have fared much better. Unfortunately, we no longer have an SGI and compiler on
which we can retest the algorithms. We do note, however, that they perform unroll-and-jam at the
intermediate code level and have a better estimation of register pressure, which is a critical issue.

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 is potentially more accurate than
an equivalent method that uses dependence-based reuse analysis [1]. Additionally it saves in the
storage of input dependences. 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 examine the performance of unroll-and-jam and software pipelining on machines that have large register files and high degrees of ILP.

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.

References


