Combining Optimization for Cache and Instruction-Level Parallelism*

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

Abstract

Current architectural trends in instruction-level parallelism (ILP) have significantly increased the computational power of microprocessors. As a result, the demands on the memory system have increased dramatically. Not only do compilers need to be concerned with finding ILP to utilize machine resources effectively, but they also need to be concerned with ensuring that the resulting code has a high degree of cache locality. Previous work has concentrated either on improving ILP in nested loops [3, 6, 7, 14, 16, 17] or on improving cache performance [9, 15, 18]. This paper presents a performance metric that can be used to guide the optimization of nested loops considering the combined effects of ILP, data reuse and latency hiding techniques. We have implemented the technique in a source-to-source transformation system called Memoria [5]. Preliminary experiments reveal that dramatic performance improvements for nested loops are obtainable (we regularly get at least a factor of 2 on kernels run on two different architectures).

1. Introduction

Modern microprocessors have increased computational power through faster cycle times and multiple instruction issuing. Unfortunately, compilers have not been able to fully utilize these advances. First, techniques like software pipelining [14, 16, 17] may not be able to take full advantage of a target architecture due to inner-loop recurrences or mismatches between the resource requirements of a loop and the resources provided by a machine [3, 6]. Second, memory speeds have not kept pace with microprocessor power. Although hardware designers have introduced multiple levels of cache to reduce latency, poor cache performance leaves many scientific applications bound by main-memory access time.

Previous work has separately addressed both the problem of improving the instruction-level parallelism available to scheduling [3, 6, 7] and the problem of restructuring code to improve cache performance [9, 15, 18]. This paper presents a loop performance metric that can be used to drive a transformation called unroll-and-jam to improve instruction-level parallelism and cache performance simultaneously. The method in this paper optimizes instruction-level parallelism (ILP), cache performance and cache-miss latency hiding requirements with one model and transformation. The method is implemented in a source-to-source transformation system and preliminary results validating its effectiveness are presented.

The work in this paper extends previous work on enhancing ILP and enhances previous work on improving cache performance through tiling. As in Carr and Kennedy [7], the method presented in this paper improves ILP by matching the resource requirements of a loop as closely as possible to the resources provided by a machine. However, we include the effects of cache. Our technique may also be useful in combination with tiling techniques designed for larger outer-level caches as a transformation targeted to small level-1 caches that support latency hiding. Unroll-and-jam creates small tiles that will easily fit in small caches and the tile size is chosen to make good use of the latency hiding capabilities of the cache.

The paper begins with a review of data dependence and background material related to measuring and improving the available ILP in innermost loops and measuring cache locality. Next, our performance metric is introduced, followed by a description of how the metric is used to guide unroll-and-jam. Finally, we present experimental evidence indicating the potential of our optimization strategy.

2. Background

We assume a pipelined architecture that allows asynchronous execution of memory accesses and floating-point operations (*e.g.*, HP PA-RISC or DEC Alpha). The target architecture must have an effective scalar optimizing compiler. In particular, the compiler should perform strength reduction, allocate registers globally (via a typical coloring scheme) and schedule the arithmetic pipelines. To estimate the utilization of ILP in loops under these assumptions, we use the notion of balance defined by Callahan, *et al.* [3].

2.1. 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.

2.2. Loop Balance

Just as machines have balance ratios, so do loops. We can define balance for a specific loop as $\beta_L = \frac{M_L}{F_L}$, where M_L and F_L are the number of memory operations and floating-point operations in the loop, respectively. We assume that references to array variables are actually references to memory, while references to scalar variables involve only registers. In this model, memory references are assigned a unit cost. Although pipeline interlock can effect the number of computation cycles in a loop, we will not deal with interlock here. Instead, we rely on other work that has already dealt with interlock [3].

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 a loop is said to be *memory bound*. 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. Floating-point operations often cannot be removed and arbitrarily increasing the number of memory operations will not likely improve performance. Finally, if $\beta_L = \beta_M$, the loop is balanced for the target machine.

2.3. 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 [1, 3, 7]. Additional computation can be introduced into an innermost loop body without a proportional increase in memory references. For example, the loop:

```
  DO 10 J = 1, 3*N
    DO 10 I = 1, M
10    A(I,J) = A(I,J-1) + A(I,J-2)
```

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

```
  DO 10 J = 1, 3*N, 3
    DO 10 I = 1, M
      A(I,J) = A(I,J-1) + A(I,J-2)
      A(I,J+1) = A(I,J) + A(I,J-1)
10    A(I,J+2) = A(I,J+1) + A(I,J)
```

and after scalar replacement to effect register allocation [2, 8] becomes:

```
  DO 10 J = 1, 3*N, 3
    DO 10 I = 1, M
      A0 = A(I,J-1)
      A1 = A0 + A(I,J-2)
      A(I,J) = A1
      A2 = A1 + A0
      A(I,J+1) = A2
10    A(I,J+2) = A2 + A1
```

The original loop has one floating-point operation and three memory references, giving a balance of 3. After applying unroll-and-jam to `J` by a factor of 2 and scalar replacement, we have three floating-point operations and five memory references. The two loads of `A(I,J)`, the load of `A(I,J+1)`, and the second load of `A(I,J-1)` are replaced with references to registers. This gives a balance of 1.67. On a machine with a balance close to 1.0, the second loop performs better.

Not only does unroll-and-jam help match balance ratios, but it also improves parallelism in the presence of inner-loop recurrences. Callahan, *et al.*, prove that unroll-and-jam will not move outer-loop recurrences inward and will create multiple copies of inner-loop recurrences [3]. These recurrence copies are independent of each other and their schedules can overlap to improve ILP.

Previous work has shown that using the following objectives to guide unroll-and-jam is effective at improving ILP in the innermost loop [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:

$$\textbf{objective function:} \quad \min |\beta_L - \beta_M|$$
$$\textbf{constraint:} \quad R_L \leq R_M$$

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. We have previously shown that, by matching the balance of a loop with that of a target architecture using unroll-and-jam, speedups of nearly 3 are attainable on kernels [7].

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 [3].

2.4. Data Reuse

The unit-cost assumption for memory references made by previous work is not accurate in practice. To compute a more realistic measure of memory costs, we use the concept of data reuse to predict the effect of cache misses. To present our model of data reuse, we first assume the reader is familiar with concept of data dependence [13, 12]. $\vec{\delta} = \{\delta_1 \ldots \delta_k\}$ is a hybrid distance/direction vector with the most precise information derivable. It represents a data dependence between two array references, corresponding left to right from the outermost loop to innermost loop enclosing the references. Data dependences are *loop-independent* if the accesses to the same memory location occur in the same loop iteration; they are *loop-carried* if the accesses occur on different loop iterations.

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 cache hierarchy. Temporal and spatial reuse may result from *self reuse* of a single array reference or *group reuse* attributed to multiple references. Without loss of generality, in this paper we assume column-major storage for arrays.

The reuse model used in this paper is identical to the one described by Carr, *et al.* [9].To simplify analysis, we concentrate on reuse that occurs between a small number of inner loop iterations. This memory model assumes there will be no conflict or capacity cache misses in one iteration of the innermost loop since the amount of data that is likely to be accessed in one loop iteration is relatively small. To compute cache reuse, we first apply algorithm RefGroup, shown below, to calculate group reuse. Two references are in the same reference group if they exhibit group-temporal or group-spatial reuse (i.e., they access the same cache line on the same or different iterations of an inner loop). This formulation is slightly more restrictive than uniformly generated references [11, 18].

RefGroup: Two references *Ref*$_1$ and *Ref*$_2$ belong to the same reference group with respect to loop l if at least one of the two following conditions holds:

1. $\exists$ *Ref*$_1$ $\vec{\delta}$ *Ref*$_2$, and
   (a) $\vec{\delta}$ is a loop-independent dependence, or
   (b) δ_l is a small constant d ($|d| \leq 2$) and all other entries are zero,
2. $\exists$ *Ref*$_1$ $\vec{\delta}$ *Ref*$_2$, and δ_f is less than the cache-line size and all other entries are zero. δ_f is the distance associated with the induction variable in the first subscript position.

Condition 1 accounts for group-temporal reuse and condition 2 detects some forms of group-spatial reuse.

To compute self-reuse properties, we consider a representative reference from each RefGroup separately. If the reference is invariant with respect to the innermost loop, it has self-temporal reuse. If the inner-loop induction variable appears only in the first subscript position of the reference, then the reference has self-spatial reuse.

Consider the following example:

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

`A(I-1,J)` has group-temporal reuse, `C(J-1,I)` has group-spatial reuse, `A(I,J)` and `A(I-1,J)` have self-spatial reuse, `B(J)` has self-temporal reuse and `C(J,I)` has no reuse.

3. Adding Cache Effects

In Section 2.2, it is assumed that all references to memory are cache hits. However, this is not realistic in practice. In this section, we show how to remove the assumption of ideal cache performance from the computation of loop balance and replace it with the model described in Section 2.4.

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 [15]. Essentially, only those array references that are determined to be cache misses in the innermost loop by the model of data reuse 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:[1]

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

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.

4. Computing Loop Balance

Below, we give an overview of the method for computing loop balance in a compiler. We derive a function that, given a set of unroll amounts for the loops surrounding a computation, will compute the balance of a loop body after unroll-and-jam by those unroll amounts. A more detailed derivation of the formula (excluding cache effects) can be found elsewhere [7]. Using this formulation, we can find the unroll amounts that will best balance a loop on a particular architecture assuming a unit cost for memory accesses.

4.1. Computing M_L and F_L

Given the following definitions, we show how to compute a formula for loop balance from a dependence graph.

$V^C =$	references that have a loop-carried or loop-independent incoming consistent dependence,but are not invariant with respect to any loop
$V^I =$	references that are invariant with respect to some loop
$V^\emptyset =$	all other references
$\vec{X} =$	$\{X_1, X_2, \ldots, X_{n-1}\}$, where X_i is the number of times the i^{th} outermost loop in L is unrolled + 1 (X_i is the number of loop bodies created by unrolling loop i)

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

For the purposes of this derivation, we assume that each loop nest is perfectly nested, all references in V_C have at most one incoming or outgoing edge and have at most one induction variable in each subscript position. Note also that references that are invariant with respect to a loop are not discussed here due to space limitations. Finally, we assume an infinite register set, letting the constraint in the optimization problem control register pressure. These assumptions simplify the presentation and are not a limitation of the method [7].

The computation of M_L and F_L have been described in detail elsewhere [7]. We will briefly review their computation here. F_L is simply the original number of floating-point operations, f, multiplied by the number of loop bodies after unroll-and-jam

$$L(n, \vec{X}) = \prod_{1 \leq i < n} X_i.$$

$M_L = M^\emptyset + M^C + M^I$. $M^\emptyset$, M^C and M^I compute the number of new memory references created by unroll-and-jam for each reference in $V^\emptyset$, V^C and V^I, respectively. For elements of $V^\emptyset$, each new reference resulting from unroll-and-jam will be a reference to memory since there will be no incoming dependences for any of the new references. This gives

$$M^\emptyset = \sum_{v \in V^\emptyset} L(n, \vec{X}).$$

M^C computes the number of new memory references created by unroll-and-jam for each reference in V^C. This total is denoted as $C(n, v, \vec{X})$. Therefore,

$$M^C = \sum_{v \in V^C} C(n, v, \vec{X}).$$

$C(n, v, \vec{X})$ must take into account the fact that some of the newly created references will have their value held in registers after unroll-and-jam and scalar replacement (see the example in Section 2) [2, 8]. Thus,

$$C(n, v, \vec{X}) = L(n, \vec{X}) - D(n, v, \vec{X})$$

where $L(n, \vec{X})$ computes the number of memory references after unroll-and-jam, but before scalar replacement and $D(n, v, \vec{X})$ represents the number of references deleted by scalar replacement. In [7] this value is shown to be

$$D(n, v, \vec{X}) = \prod_{1 \leq i < n} (X_i - \delta_{i,v})^+.$$

Here $\delta_{i,v}$ is the i^{th} entry in the distance vector of the incoming dependence edge for reference v. The formula computes the number of references that, after unroll-and-jam, will have an incoming dependence that is loop independent or carried by the innermost loop (i.e., those that have temporal reuse).

The computation of M^I is omitted due to space. Please see Carr and Kennedy for details [7].

4.2. Computing Cache Effects

To compute the latency hiding requirements of a loop nest, we will use the method of Mowry, *et al.*, to decide what needs to be prefetched [15]. Those references with self-spatial reuse need to be prefetched every

$$S = \left\lfloor \frac{C_l}{\text{stride}} \right\rfloor$$

loop iterations where C_l is the cache-line size and "stride" is the distance between successive memory accesses by the same array reference. Those references possessing group-temporal, group-spatial or self-temporal reuse according to our reuse model do not need to be prefetched, and references without reuse need to be prefetch on every iteration.

Each partition of array references is handled separately when computing P_L. Therefore, $P_L = P^\emptyset + P^C + P^I$, where $P^\emptyset, P^C$ and P^I are the prefetch requirements for $V^\emptyset$, V^C and V^I, respectively. Each of these components is computed from the dependence graph using the reuse analysis presented in Section 2.4. Below we detail how to compute $P^\emptyset$ and P^C. We omit the computation of P^I due to space. This computation is similar to $P^\emptyset$ except that we use the computation of M^I rather than $M^\emptyset$ to determine to determine how many references have a particular reuse property.

4.2.1. Computing $P^\emptyset$

Elements of $V^\emptyset$ do not possess temporal reuse because they are not the sink of a consistent dependence. Therefore, only their spatial reuse properties need to be examined to determine their prefetching requirements. If a member of $V^\emptyset$ has self-spatial reuse, each of its copies will have self-spatial reuse. Thus, the original reference and its copies require

```
$P^\emptyset = \sum_{v \in V^\emptyset} \bar{p}^\emptyset(n, v, \vec{X})$ where

$\bar{p}^\emptyset(n, v, \vec{X}) \Leftarrow$
    if $v$ has self-spatial reuse wrt loop $n$
      return $\frac{L(n,\vec{X})}{S}$
    else if $\exists k, 1 \leq k < n | v$ is self-spatial wrt $k$ then
      return $\prod_{1 \leq i < n, i \neq k} X_i \times \left\lceil \frac{X_k}{S} \right\rceil$
    else
      return $L(n, \vec{X})$
```

Figure 2. Computation of $P^\emptyset$

a total of $\frac{L(n,\vec{X})}{S}$ prefetches per loop iteration after unroll-and-jam. For an example, see `B(J,I)` and `B(J,I+1)` in Figure 1.

If a reference does not posses self-spatial reuse and the loop corresponding to the induction variable in the first subscript position is unroll-and-jammed, group spatial reuse will result. In Figure 1, `A(I,J)` is self-spatial with respect to the `I`-loop. In the unroll-and-jammed loop, the newly created `A(I+1,J)` has group-spatial reuse. In this case a prefetch instruction will be needed for each reference that is created by unrolling a loop other than `I` and for each reference that brings in a new cache line when `I` is unrolled. This second value is $\lceil \frac{X_I}{S} \rceil$ prefetches per iteration. The formulation in Figure 2 summarizes the computation of the number of prefetches needed for references in $V^\emptyset$.

4.2.2. Computing P^C

For references in V^C we must consider both temporal and spatial reuse properties after unroll-and-jam. Given our model of cache locality, temporal reuse can only occur across the innermost loop. However, all references with temporal reuse across the innermost loop will have their values allocated to registers by scalar replacement. Therefore, we need only consider the self-spatial and group-spatial reuse properties of those references having no temporal reuse after unroll-and-jam. For each reference $v \in V^C$, this quantity is $C(n, v, \vec{X})$ as defined previously. If the original reference, v, has self-spatial reuse, then each new reference not removed by scalar replacement will have self-spatial reuse, requiring one prefetch every S loop iterations, or $\frac{C(n,v,\vec{X})}{S}$ prefetches per iteration.

For references not having self-spatial reuse, we divide group-spatial reuse into two types. Type 1 is described by condition 2 of our reuse model. Type 2 occurs between multiple references to the same cache line whose corresponding array references are not connected by a dependence (see

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

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

Figure 1. Group-Spatial Reuse Created by Unroll-and-Jam

A(I,J) and A(I+1,J) in Figure 1).

If a reference is self-spatial with respect to some outer loop, then some of the references left after scalar replacement may have group-spatial reuse. Recall that Type 1 group-spatial reuse occurs between two references connected by a dependence with a distance vector of the form $\{0,\ldots,0,\delta_k,0,\ldots,0\}$ where the k^{th} loop induction variable is in the first subscript position. If any distance vector entry other than k is non-zero, then there will be no group-spatial reuse. Since $C(n,v,\vec{X})$ computes the number of references that have at least one non-zero entry in the first $n-1$ entries of their incoming distance vector, the number of references that have a non-zero entry in the k^{th} position and some other position can be computed as in $C(n,v,\vec{X})$ by ignoring the k^{th} loop. We refer to this as $C(n,v,\vec{X},k)$.

$$C(n,v,\vec{X},k) = L(n,\vec{X},k) - D(n,v,\vec{X},k)$$

where

$$L(n,\vec{X},k) = \prod_{1\leq i<n, i\neq k} X_i$$

and

$$D(n,v,\vec{X},k) = \prod_{1\leq i<n, i\neq k} (X_i - \delta_{i,v})^+.$$

Essentially, all of the references handled by $C(n,v,\vec{X},k)$ will differ in some subscript position other than the innermost position, thus, having no group-spatial reuse (see the definition of A(I,J,K) and the use of A(I-1,J,K-1) in Figure 3). Unrolling k creates copies of these references that differ only in the innermost subscript position when compared against the references that existed before unrolling k (see the uses of A(I-1,J,K-1) and A(I,J,K-1) in Figure 3). These references have Type 2 group-spatial reuse. Therefore, we need

$$gs_1(n,v,\vec{X},k) = C(n,v,\vec{X},k) \times \left\lceil \frac{X_k}{S} \right\rceil$$

```
$P^C = \sum_{v\in V^C} \bar{p}^C(n,v,\vec{X})$ where

$\bar{p}^C(n,v,\vec{X}) \Leftarrow$
    if $v$ has self-spatial reuse wrt loop $n$
      return $\frac{L(n,\vec{X})}{S}$
    else if $\exists k, 1 \leq k < n | v$ is self-spatial wrt $k$ then
      if $\delta_n = 0$ then
        return $gs_1(n,v,\vec{X},k)$
      else
        return $gs_2(n,v,\vec{X},k)$
    else
      return $L(n,\vec{X})$
```

Figure 4. Computation of P^C

prefetches per loop iteration.

If two references are connected by a dependence with a distance vector of $\{0,\ldots,0,\delta_k,0,\ldots,0,\delta_n\}$, where $\delta_n > 0$, then no dependence can result in Type 1 group-spatial reuse. Therefore, unroll-and-jam can only create Type 2 group-spatial reuse between a reference and its copies. From previous work we know that temporal reuse is introduced in the innermost loop when $X_k > \delta_k$ [7]. We must stop considering group-spatial reuse at this point because we have already accounted for the group-temporal reuse in computing M_L. Note that in gs_1 this type of reference has Type 1 group-spatial reuse before unrolling k, but does not here (see the dependence between A(I,J,K) and A(I-1,J,K), and the dependence between B(I,J,K) and B(I-1,J-1,K) in Figure 3 for the contrast). The number of prefetches required per iteration for this type of reference is

$$\begin{aligned} gs_2(n,v,\vec{X},k) = \; & C(n,v,\vec{X},k) \times \left\lceil \frac{X_k}{S} \right\rceil + \\ & D(n,v,\vec{X},k) \times \left\lceil \frac{\min(X_k,\delta_{k,v})}{S} \right\rceil \end{aligned}$$

$C(n,v,\vec{X},k)$ is the same as in computing gs_1. However, we must also account for those references that have a dependence with $\delta_k > 0, k \neq n$ in the computation involving $D(n,v,\vec{X},k)$. These references will create Type 2 group-spatial reuse when k is unrolled (see B(I-1,J-1,K) and B(I,J-1,K) in Figure 3). In Figure 4, we summarize the computation of P^C.

```
      DO 10 I = 1,2*N
         DO 10 K = 1,2*N
           DO 10 J = 1,N
              A(I,J,K) = A(I-1,J,K-1)
10            B(I,J,K) = B(I-1,J,K-1)
```

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

Figure 3. Group-Spatial Reuse Before Group-Temporal Reuse

4.3. Computing L_L

To compute L_L, we chose the minimum of M_L and F_L. This assumes no stall cycles and represents a lower bound on the cycle time of the loop. By choosing the lower bound we are being conservative since we are allowing fewer cycles to hide prefetch latency. A more accurate estimation of L_L can use the the method of Callahan, *et al.*, to estimate pipeline interlock and include pipeline stall cycles in the loop [3].

4.4. Evaluating β_L at Compile Time

Previous work has shown that the formula for β_L, not including cache costs, can be reduced to the following form for two loops, i and j, by examining the edges in the dependence graph [7].

$$\beta_L = \frac{c_0 X_i X_j + c_1 X_i + c_2 X_j + c_3}{f X_i X_j}$$

In practice, we limit unroll-and-jam to at most two loops because most loops have a nesting depth of 3 or less. Because the $^+$-function in computation of $D(n, v, \vec{X})$ causes the coefficients c_1, c_2, and c_3 to vary with the unroll amounts, we keep a set of coefficients that correspond to the most common distances. In practice, most distances are 0 or 1, allowing us to keep one set for each common distance and another set for all other distances. When adding in the computation of P_L, we have to consider the ceiling functions when computing the number of prefetches for references with spatial reuse. We must be able to remove the ceiling function in order to compute a coefficient from the dependence graph. We can use our assumption that all distances are 0 or 1 and simply count the number of ceiling operations that need to be performed. Given the formula for β_L and the fact that unroll amounts are limited by R_M, we can search a 2-d solution space for X_i and X_j in $O(R_M^2)$ time to find the best answer.

5. Experiment

We have implemented the previously described algorithm in Memoria, a Fortran source-to-source translator based upon the ParaScope programming environment [5, 4]. To test the effectiveness of our implementation, we have selected a set of Fortran loops from standard kernels and benchmark program suites. The loops are chosen from those within the suite that are not already balanced and those for which unroll-and-jam is legal (the percentage of loops that could not be unrolled is small). 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	RiCEPS/SIMPLE/CONDUCT
8	cond.9	RiCEPS/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

In this experiment, we target Memoria to two architectures: the DEC Alpha 21164 and the HP PA-RISC 712/80. The Alpha has an 8K direct-mapped level-1 cache and a

96K 3-way set-associative level-2 cache. The HP has a 256K direct-mapped cache. On each machine, we run our test loops trough Memoria and then use the the standard product compiler with full intraprocedural optimization (the -O option on the HP and the -O5 option on the DEC) to generate the final code. Figure 5 gives the experimental design.

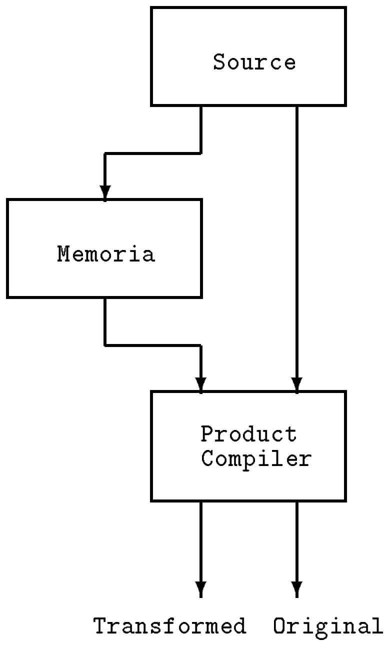

Figure 5. Experimental Design

Figures 6 and 7 show that 16 of the 19 test loops get an improvement on at least one of the DEC Alpha and HP PA-RISC (the other 3 loops show no improvement nor degradation). Improvements of a factor of 3 or more occur on 5 loops on the DEC Alpha (2,3,4,14,15) and 6 loops on the HP (2,3,4,8,13,15). To further analyze the reasons for these improvements we have performed cache simulations for each architecture. In addition, we measure the predicted improvement in balance as measured by our metric. This information is presented in Table 2.

As can be seen from Table 2, many of the loops have a large improvement in cache hit ratio on the DEC Alpha. The run-time improvements for loops 3, 4 and 8 are almost exclusively due to cache performance. Loops 13, 15 and 16 see large improvements in both cache performance and ILP. On the HP, almost all improvements are due to increases in ILP. Increases in both cache performance and ILP occur for loops 15 and 16. The reason that cache has less of an effect on the HP is due to its size (256K vs. 8K).

Previous work has suggested that it is most important to put the loop carrying the most data reuse in the innermost position to get the best performance [9]. Our experiments reveal two counterexamples to this heuristic. Both dmxpy0 and mmjki (loops 12 and 16) are ordered for the best cache performance while dmxpy1 and mmjik (loops 13 and 15) compute the same thing, respectively, but are ordered for better ILP. Before unroll-and-jam the cache ordering gives significantly better performance. However, after unroll-and-jam, the ILP ordering gives better raw performance on both architectures and loops. Unroll-and-jam significantly narrows the cache performance difference and also provides more ILP to the scheduler. This suggests that if unroll-and-jam can provide the cache performance, it might be better to order loops for ILP to get the better performance.

Loops 1, 18, and 19 have no performance improvement due to unroll-and-jam for either architecture. On each of these loops cache performance does not change after unroll-and-jam and each loop is already balanced between cache accesses and computation.

In Table 2, we report the prefetch bandwidth required for each architecture in the “DEC I_L” and “HP I_L” columns. This data suggests that the loops in our test suite do not require a high degree of prefetching to eliminate much of the remaining cache misses. No unroll-and-jammed loop requires even a prefetch instruction per cycle and most require less than one every other cycle. Unroll-and-jam often decreases the prefetch bandwidth requirement of a loop by unrolling a loop that provides good spatial reuse.

Figures 6 and 7 show the performance difference between the cache-based model presented in this paper and the model that assumes perfect cache performance. Because we are dealing with a backend compiler over which we have no control we must handle the limits it imposes. Each of the HP and DEC compilers performs a form of scalar replacement. Therefore, we are unable to control which array values are put in registers and which are not when there are more values than registers. This forces us to limit unroll-and-jam even when larger unroll amounts might lead to better cache performance. Because of this limit unroll-and-jam is often constrained by register pressure in both models and we are not able to see the benefits of including cache effects. On the DEC Alpha 6 of the loops are unrolled differently using the cache-based model (loops 2, 3, 13, 14, 15 and 16) and performance gains of 2% to 55% are observed. On the HP, 9 loops are unrolled differently using the cache-based model (loops 2, 3, 7, 9, 12, 13, 14, 15 and 16) with performance gains of 2% to 64%.

On the HP, loops 13 and 15 experience the largest improvements due to the new model since their unrolling is not bound by register pressure and they have low original cache-hit ratios. On the Alpha, loops 2 and 3 experience the largest improvement. Here, cache performance is greatly improved for the small 8K cache. In those loops that are unrolled more due to the new model but little performance

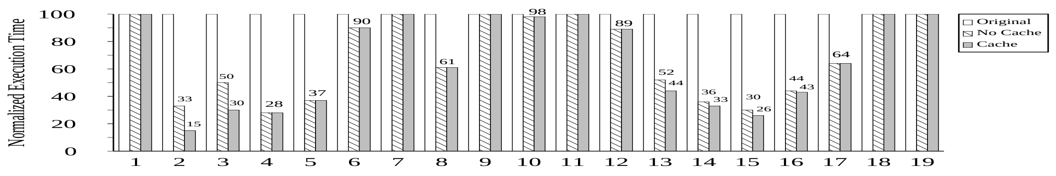

Figure 6. Performance of Test Loops on DEC Alpha

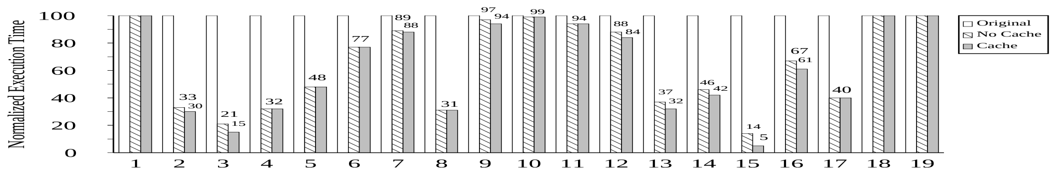

Figure 7. Performance of Test Loops on HP PA-RISC

Loop	DEC Hit Ratio		HP Hit Ratio		DEC Balance		HP Balance		DEC I_L		HP I_L	
	Orig	Opt	Orig	Opt	Orig	Opt	Orig	Opt	Orig	Opt	Orig	Opt
1	82.4	82.4	84.9	84.9	2.2	2.2	5.8	3.2	0.2	0.2	0.2	0.11
2	88.3	98.7	99.9	99.9	5.5	0.92	14.5	0.97	0.25	0.08	0.25	0.03
3	35.3	81.7	100	100	13.95	5.87	43.65	16.85	1.65	0.13	1.65	0.13
4	11.5	70.0	100	100	13.95	3.5	43.65	22.53	1.65	0.29	1.65	0.83
5	17.8	76.6	82.8	84.3	13.95	3.5	43.65	22.53	1.65	0.29	1.65	0.83
6	75.0	81.2	75.0	81.2	8.25	5.73	21.75	15.95	0.25	0.52	0.25	0.55
7	75.1	75.1	81.2	81.2	11	9.63	29	24.17	0.25	0.88	0.25	0.83
8	18.8	76.1	81.2	81.2	25	8.29	79	21.79	0.75	0.75	0.75	0.75
9	87.3	87.3	87.4	87.4	5.5	4.24	14.5	11.6	0.5	0.39	0.5	0.4
10	83.3	83.3	83.2	83.2	4.13	3.21	10.88	8.46	0.38	0.29	0.38	0.29
11	87.4	87.4	87.2	87.2	5.5	4.58	14.5	12.08	0.5	0.42	0.5	0.42
12	87.2	88.4	91.5	91.5	8.25	2.98	21.75	8.7	0.25	0.27	0.25	0.3
13	72.8	88.2	91.2	91.6	10.75	2.89	33.25	7.61	0.63	0.26	0.63	0.26
14	51.6	91.1	92.4	93.7	18.75	5.64	59.25	14.86	0.75	0.51	0.75	0.51
15	68.6	93.7	93.0	98.1	10.75	1.38	33.25	3.63	0.63	0.13	0.63	0.13
16	89.9	95.1	93.6	97.7	8.25	2.02	21.75	5.32	0.25	0.18	0.25	0.18
17	63.9	86.0	93.0	93.0	11.67	3.92	35.67	35.67	1.33	0.33	1.33	0.33
18	93.7	93.7	93.7	93.7	2.75	1.5	7.25	3.95	0.25	0.14	0.25	0.14
19	92	92	93	93	2.3	2.0	7.25	5.03	0.21	0.18	0.25	0.17

Table 2. Statistics for Test Loops

gain is observed, like loop 7 on the HP, additional unrolling does not improve cache performance. In those loops that are not unrolled more with the cache model, like loop 17, unrolling is bounded by register pressure and the new model can provide no benefit.

Finally, we mention the interaction of unroll-and-jam and software pipelining. While unroll-and-jam can dramatically improve the schedules produced by software pipelining, it can also greatly exacerbate register pressure [6, 10]. In this experiment, we artificially reduce the number of registers allowed to limit unroll-and-jam's potential negative effect on software pipelining. An important future research topic is to predict register pressure before software pipelining based upon the loop body and unroll amounts.

6. Summary And Future Work

In this paper, we have presented an extension to the computation of loop balance that incorporates the effects of latency hiding and cache misses by determining the number of misses whose latency cannot be hidden in an unrolled loop body. The extension allows us to optimize for both cache and ILP using one transformation. Unroll-and-jam guided by the new metric can yield large performance gains – a factor of 20 on one loop. Not only can loop balance be used to optimize a loop for a particular architecture, but with the addition of the cache-effect metric we can determine if a machine has enough latency-hiding bandwidth to hide the memory latency in loops.

This work can be seen as complimentary to previous work on tiling for cache [11, 18]. The previous tiling work can be targeted for larger high-level caches while our method is useful for small level-1 caches. Our optimization strategy produces small tile sizes and improves both ILP and cache performance.

In the future we plan to compute β_L using the memory reuse analysis designed by Wolf and Lam [18]. We will investigate if this reuse method will give us more accurate results since it computes all forms of group-spatial reuse and can consider reuse across more than the innermost loop. In addition, we will look into the effects of our optimization technique on architectures that support software prefetching. Finally, we will 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 positive step in that direction.

Acknowledgments

This research was supported by the National Science Foundation under grant CCR-9409341 and by Hewlett-Packard Company. The DEC Alpha was provided by a Digital Equipment Corporation External Research Program grant.

References

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

[2] 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.

[3] 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.

[4] 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.

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

[6] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, Maui, HI, January 1996.

[7] 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.

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

[9] S. Carr, K. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, Santa Clara, California, 1994.

[10] C. Ding. Improving software pipelining with unroll-and-jam and memory-reuse analysis. Master's thesis, Michigan Technological University, June 1996.

[11] 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.

[12] 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*.

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

[14] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Notices*, 23(7):318–328, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[15] 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.

[16] B. Rau. Iterative modulo scheduling. In *Proceedings of the 27th International Symposium on Microarchitecture (MICRO-27)*, pages 63–74, San Jose, CA, December 1994.

[17] N. Warter, G. Haub, and J. Bockhaus. Enhanced modulo scheduling for loops with conditional branches. In *Proceedings of the 25th International Symposium on Microarchitecture (MICRO-25)*, pages 170–179, Portland, OR, December 1992.

[18] 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*.