International Journal of Parallel Programming, Vol. 29, No. 5, October 2001 (© 2001)

Optimized Unrolling of Nested Loops

Vivek Sarkar[1]

Received December 2000; revised January 2001

Loop unrolling is a well known loop transformation that has been used in optimizing compilers for over three decades. In this paper, we address the problems of automatically selecting *unroll factors* for perfectly nested loops, and generating *compact code* for the selected unroll factors. Compared to past work, the contributions of our work include (i) a more detailed cost model that includes register locality, instruction-level parallelism and instruction-cache considerations; (ii) a new code generation algorithm that generates more compact code than the unroll-and-jam transformation; and (iii) a new algorithm for efficiently enumerating feasible unroll vectors. Our experimental results confirm the wide applicability of our approach by showing a $2.2\times$ speedup on matrix multiply, and an average $1.08\times$ speedup on seven of the SPEC95fp benchmarks (with a $1.2\times$ speedup for two benchmarks). Larger performance improvements can be expected on processors that have larger numbers of registers and larger degrees of instruction-level parallelism than the processor used for our measurements (PowerPC 604).

KEY WORDS: Loop transformations; loop unrolling; unroll-and-jam; unroll factors.

1. INTRODUCTION

Loop unrolling[1] is a well known program transformation that has been used in optimizing compilers for over three decades. In addition to its use in compilers, many software libraries for matrix computations contain loops that have been hand-unrolled for improved performance.[2] The original motivation for loop unrolling was to reduce the (amortized) increment-and-test overhead for loop iterations. For modern processors, the primary benefits of loop unrolling include increased instruction-level

[1] IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598. E-mail: vsarkar@us.ibm.com

0855-7485/01/1000-0545$19.50/0 © 2001 Plenum Publishing Corporation

parallelism (ILP), improved register locality ("register tiling"), and improved memory hierarchy locality.[3–5] Loop unrolling is also essential for effective exploitation of some newer hardware features, e.g., for uncovering opportunities for generating dual-load/dual-store instructions,[6] and for amortizing the overhead of a single prefetch instruction across multiple load or store instructions.[7, 8]

However, it has been observed that loop unrolling can also have a negative effect on a program's performance when it is not used judiciously. For example, excessive unrolling can lead to runtime performance degradation due to extra register spills when the register working set ("register pressure") of the unrolled loop body exceeds the number of available registers.[9] Another concern is with the code size of the unrolled loop, which can overflow a small first-level instruction-cache if loop unrolling is performed too aggressively.[10] Apart from creating a large unrolled loop body, additional loops have to be introduced to correctly handle cases where the unroll factor does not evenly divide the number of iterations. These remainder loops substantially increase the compile-time for the transformed code and the size of the final object code, even though only a small fraction of the program's execution time is spent in these remainder loops.

Most industry-strength compilers (including the optimizing back-end of the XL Fortran compiler, which is the baseline for our performance measurements) perform software pipelining and limited unrolling of *innermost loops*. However, unrolling of *perfectly nested loops* [as in the *unroll-and-jam* transformation[1, 11]] is performed less frequently (and with greater caution) because of its potential for increased overhead due to increases in runtime, compile-time or code size.

In this paper, we address the problems of automatically selecting *unroll factors* for a set of perfectly nested loops, and generating *compact code* for the selected unroll factors as as to make it a practical transformation for use by industry-strength compilers. Compared to past work, the contributions of our work include (i) a more detailed cost model that includes ILP and I-cache considerations; (ii) a new code generation algorithm for unrolling nested loops that generates more compact code (with fewer remainder loops) than the unroll-and-jam transformation; and (iii) a new algorithm for efficiently enumerating feasible unroll vectors.

The problem of automatically selecting unroll factors for nested loops has been addressed in past work by Carr and Kennedy[9] and more recently by Carr and Guan.[12] For loop kernels, their results are impressive and make a convincing case for leaving the task of selecting unroll factors to the compiler rather than the programmer. However, their results for full applications are less convincing—no results were reported by Carr and Guan[12] for applications, and for the 10 applications considered by Carr and Kennedy[9]

from the SPEC92, Perfect and RiCEPS benchmark suites, the average speedup obtained was 1.04× on an RS/6000 model 540.

The algorithm used by Carr and Kennedy[9] was based on the use of *input dependences*,[13] whereas the approach by Carr and Guan[12] was based on using the reuse model from Wolf and Lam[14] and its associated *linear algebra framework*. Our solution (which was developed independently[2] of these past approaches) has a different technical foundation based on using cost models that are both more detailed and more efficient to compute than the cost models used in previous work. Our current performance results on a PowerPC 604 processor show an average 1.08× speedup on seven of the SPEC95fp benchmarks (with a 1.2× speedup for two benchmarks). These speedups are significant because the baseline compiler used for comparison is the IBM XL Fortran product compiler which generates high quality code with unrolling and software pipelining of innermost loops enabled. The only benchmark common to Carr and Kennedy[9] and to our results is the SPEC benchmark, tomcatv. For tomcatv, the speedup due to unroll-and-jam (and scalar replacement) reported by Carr and Kennedy[9] was only 1.01×, whereas the speedup for tomcatv obtained using our approach was 1.23×.

The rest of the paper is organized as follows. Section 2 describes our approach to automatic selection of unroll factors for a set of perfectly nested loops. Section 3 describes how we generate code for a specified unroll vector; this algorithm generates code that is more compact than the code generated by the unroll-and-jam transformation. Section 4 contains our experimental results. Section 5 outlines extensions to our algorithm for selecting unroll factors to support other optimizations that interact with unrolling. Section 6 discusses related work, and Section 7 contains our conclusions. Appendix A contains an example to illustrate the compactness of the code generation obtained by our approach, compared to that of the unroll-and-jam transformation.

2. AUTOMATIC SELECTION OF UNROLL FACTORS

This section describes our approach to automatic selection of unroll factors for a set of perfectly nested loops. Section 2.1 reviews the unroll-and-jam transformation. Section 2.2 formalizes selection of unroll factors for multiple perfectly nested loops as an optimization problem. Section 2.3 introduces our cost function for estimating the cost of an unrolled loop nest for a given vector of unroll factors, and capacity cost functions to

[2] The origins of our work lie in the ASTI optimizer built during 1991–1993 for adding high-level transformations to the XL Fortran product compilers.[15]

model register set and I-cache constraints. Section 2.4 outlines our algorithm for efficiently enumerating feasible unroll vectors and selecting a feasible unroll vector that has lowest cost. Section 2.5 uses a matrix multiply computation as an example to illustrate our approach for automatically selecting unroll factors.

The program model assumed in our work is as follows. A loop is a candidate for unrolling if it is a counted loop with no premature exits, e.g., Fortran DO loops, or special cases of for loops in C and Java. Unlike some prior work on loop unrolling, we allow the lower bound, upper bound, and step expressions to have arbitrary (positive or negative) integer values that may be unknown at compile-time. We also permit general (structured or unstructured) acyclic control flow within a single iteration of the loop nest.

2.1. Unroll-and-Jam

Consider a perfect nest of two loops, i1 and i2, as shown in Fig. 1, and assume we wish to unroll only the outer loop by a factor of R. The first step in Fig. 1 shows the result of a mechanical unrolling of the outer i1 loop by an unroll factor of R. (For convenience, we use the standard Fortran *lower-bound, upper-bound, step* triple notation to describe loops that have nonunit step values.)

However, the output of the first step in Fig. 1 is not in a useful form for enabling code optimization because of the multiple copies of the inner i2 loop present after unrolling the i1 loop. The performance benefits due to unrolling are realized when the multiple copies of the i2 loop are fused together as shown in step 2 of Fig. 1 (the remainder loop is unaffected by this loop fusion step). As described in Section 3, this two-step unroll-and-jam sequence[1, 11] is performed as a single transformation in our framework.

Unlike unrolling a single loop, unrolling of multiple loops is not always legal. The first unroll step can always be performed, but data dependences may prevent the second fusion ("jam") step from being performed. Complex (nonlinear) loop bounds may also make it illegal to perform a loop unrolling transformation. In a classical unroll-and-jam transformation, it is the responsibility of the fusion step to recognize when an illegal unrolling transformation is being attempted on a loop nest. However, the legality condition for unrolling multiple loops is equivalent to that of tiling,[16] i.e., given a set of k perfectly nested loops $i_1, \ldots, i_k$, it is legal to unroll outer loop i_j if it is legal to permute loop i_j to the innermost position. In fact, unrolling of multiple loops can be viewed as dividing the iteration space into small tiles. However, the iterations in an unrolled "tile" execute *copies* of the loop body that have been expanded (unrolled) in place, rather than executing inner control loops as in tiling for cache locality.

```
STEP 1: Unroll the outer i1 loop
--------------------------------

! INPUT LOOP NEST                 ! UNROLLED LOOP
DO i1 = lo1,hi1                   DO i1 = lo1,hi1-(R-1),R
  DO i2 = lo2,hi2                   DO i2 = lo2,hi2
    BODY(i1,i2)                       BODY(i1,i2)
  END DO                            END DO
END DO                              . . .
                                    DO i2 = lo2,hi2
        Unroll outer                  BODY(i1+R-1,i2)
       -------------->              END DO
        loop R times              END DO

                                  ! REMAINDER LOOP
                                  DO i1 = i1,hi,1
                                    DO i2 = lo2,hi2
                                      BODY(i1,i2)
                                    END DO
                                   END DO


STEP 2: Fuse/jam multiple copies of inner i2 loop
-------------------------------------------------

! UNROLLED LOOP                     ! UNROLLED LOOP
DO i1 = lo1,hi1-(R-1),R             ! (AFTER FUSION)
  DO i2 = lo2,hi2                   DO i1 = lo1,hi1-(R-1),R
    BODY(i1,i2)                       DO i2 = lo2,hi2
  END DO          Fuse i2 loops         BODY(i1,i2)
  . . .          --------------->       . . .
  DO i2 = lo2,hi2                       BODY(i1+R-1,i2)
    BODY(i1+R-1,i2)                   END DO
  END DO                            END DO
END DO

! REMAINDER LOOP                 ! REMAINDER LOOP (UNCHANGED)
. . .                               . . .
```

Fig. 1. Unrolling of outer loop in a nest of two counted loops (unroll-and-jam).

```
                                        ! UNROLLED LOOP
                                        DO i1 = lo1,hi1-(u1-1),u1
    ! INPUT LOOP                          DO i2 = lo2,hi2-(u2-1),u2
    DO i1 = lo1,hi1                         . . .
      DO i2 = lo2,hi2                       BODY(i1,i2,...)
        . . .                               . . .
          BODY(i1,i2,...)      --->         BODY(i1+u1-1,i2,...)
        . . .                               BODY(i1,i2+1,...)
      END DO                                . . .
    END DO                                  BODY(i1+u1-1,i2+1,...)
                                            . . .
                                            . . .
                                          END DO
                                        END DO

                                        ! REMAINDER LOOPS
                                        . . .
```

Fig. 2. General unrolling of multiple nested loops.

The transformation in Fig. 1 demonstrates how unrolling can be performed on a doubly nested loop with unroll vector $(R, 1)$, i.e., an unroll factor of R for the outer loop and an unroll factor of 1 (no unrolling) for the inner loop. However, the framework presented in this paper can be used to generate code for any unrolling transformation specified by an arbitrary unroll vector for a set of perfectly nested loops.

2.2. Problem Statement

Consider a set of k perfectly nested loops with index variables, $i_1,..., i_k$. The perfect loop nest may have been written by a programmer or obtained as a result of compiler transformations such as loop distribution.[15, 16] An unrolling transformation can be specified by an *unroll vector*, $\mathbf{u} = (u_1, u_2,...)$, which identifies an unroll factor, u_j, for each loop j. The lexicographic ordering of unroll factors corresponds to the ordering of the loops from outermost to innermost.

Figure 2 outlines the structure of the unrolled loop nest that would be obtained from a given unroll vector. (For simplicity, remainder loops are not shown in this code structure.) Note that the unrolled loop body contains $u_1 \times u_2 \times \cdots$ copies of the input loop body; each copy of *BODY* is instantiated for a different tuple of index value taken from the Cartesian product,

$$\{i_1,..., i_1 + u_1 - 1\} \times \cdots \times \{i_k,..., i_k + u_k - 1\}$$

The optimization problem that we are interested in solving is to find an unroll vector, $(u_1^{opt},\ldots, u_k^{opt})$, such that

1. Each unroll factor, u_i^{opt} is an integer in the range, $1\cdots u_i^{max}$, where $\mathbf{u}^{max} = (u_1^{max},\ldots, u_k^{max})$ is the *maximum unroll vector* for the loop nest,
2. The unroll vector identifies a *legal* unrolling transformation,
3. The amortized number of *register spills* per original iteration in the unrolled body does not exceed the number of register spills in the original loop body,
4. The unrolled loop body fits in the *instruction cache*, and
5. The estimated cost of the unroll configuration is minimized. (If multiple unroll vectors have the same estimated cost, then choose a vector with the smallest total unroll factor, $u_1 \times \cdots \times u_k$ as the solution.)

Conditions 1 and 2 are requirements imposed on a legal unrolling transformation. To enforce Condition 2, we identify noninnermost loops that cannot be *permuted* to the innermost position in the input loop nest due to dependence constraints or constraints on loop bounds.[17] For each such loop, i, we set $u_i^{max} = 1$ to ensure that loop i is not unrolled. For other loops, j, we set u_j^{max} = maximum number of iterations for loop j, using an estimated value when the number is unknown.

Conditions 3 and 4 are *capacity constraints*. Condition 3 ensures that loop unrolling does not cause extra register spills, and Condition 4 ensures that loop unrolling will (most likely) not lead to extra I-cache misses. Our experience is that Condition 3 is usually more tightly binding than Condition 4, i.e., ensuring no increase in register spills is usually sufficient to ensure that there is no increase in I-cache misses.

In general, enforcing Condition 3 requires detailed knowledge of the register allocation algorithm used by the back-end. For simplicity, our solution to modeling Condition 3 is to ensure that the maximum numbers of fixed-point and floating-pointing values in the unrolled loop that *may* be simultaneously live are bounded by the numbers of available fixed-point and floating-point registers respectively (see Section 2.3). This max computation is conservatively large—it assumes that two values may be simultaneously live if there exists some legal instruction reordering for which they would be simultaneously live (even if the values are not simultaneously live in the original instruction ordering). While this approximation may unnecessarily limit the amount of unrolling permitted, it ensures that any software pipelining or instruction scheduling performed by the back-end will not introduce additional spills.

Condition 5 is the *objective function* to be minimized and is defined next in Section 2.3.

2.3. Cost Function

In this section, we define an objective function $F(u_1,\dots, u_k)$ that evaluates the cost of a given unroll vector, $(u_1,\dots, u_k)$, for a perfect nest of k loops. [A simpler version of this cost function was presented by Sarkar.[15]] Having an explicit cost function simplifies the unrolling optimization and makes it convenient to retarget the optimization to different processor architectures or different models of the same processor architecture.

In our approach, the compiler builds the following *symbolic cost functions* based on the data references in the loop nest. All functions take unroll factors as arguments and return estimated values for the unrolled loop body that would be generated by a $u_1 \times \cdots \times u_k$ unroll transformation of the input loop nest:

- $IR(u_1,\dots, u_k)$ = number of distinct Integer Register (fixed-point) values in unrolled loop body.
- $FR(u_1,\dots, u_k)$ = number of distinct Floating-point Register values in unrolled loop body. IR and FR are computed by using the approach given by Sarkar[15] and Ferrante *et al.*[18] for estimating the number of distinct array elements accessed in a loop nest. This approach avoids the expense of computing input dependences or of using a linear algebra framework to perform the estimation.
- $LS(u_1,\dots, u_k)$ = estimated number of cycles spent on Load and Store instructions in unrolled loop body.
- $CP(u_1,\dots, u_k)$ = estimated Critical Path length of unrolled loop body (in cycles). We assume zero cost for *load/store* instructions when estimating CP, because they are already accounted for in LS.
- $TC_j(u_1,\dots, u_k)$ = estimated Total Cycles on class j of functional units required by unrolled loop body. We assume zero cost for load/store instructions when estimating TC_j, because they are already accounted for in LS. Let NF_j be the number of functional units of class j available in the machine.

The symbolic cost functions are represented as expression trees in the compiler with internal nodes that represent sum, product, reciprocal, min, max operators. A leaf of an expression tree can be an unroll factor, u_i, or a constant. This representation makes it convenient to evaluate a symbolic cost function for a given unroll vector.

The IR and FR cost functions are used to enforce register capacity constraints. In addition, an estimated code size for a single iteration is used to enforce the I-cache constraint.

The remaining cost functions contribute to the the objective function to be minimized, which is a *cost per iteration* defined as follows:

$$F(u_1,\dots,u_k) = \frac{\overbrace{LS(u_1,\dots,u_k)}^{\text{load/store term}}}{u_1 \times \cdots \times u_k} + \underbrace{\frac{\max\left[CP(u_1,\dots,u_k), \max_j\left(\left\{\frac{TC_j(u_1,\dots,u_k)}{NF_j}\right\}\right)\right]}{u_1 \times \cdots \times u_k}}_{\text{ILP term}}$$

The objective function is defined to be the sum of the *load/store term*, $LS(u_1,\dots,u_k)$ and the *ILP term*, which is a max function that provides an estimation of the parallel execution time of the unrolled loop body. Both terms are divided by the product of unroll factors, $u_1 \times \cdots \times u_k$ so as to obtain a cost function that is an *amortized* cost per original iteration of the input loop nest, thus making it possible to directly compare costs for different unroll vectors.

A key design principle behind this cost function is that its terms should be efficient to evaluate for different unroll vectors without actually having to perform the unrolling transformation for each candidate unroll vector. That is the main motivation for separating the load/store term (LS) from the ILP term in the max function. (Otherwise, we would have to use different CP and TC functions for different unroll vectors.)

It is instructive to compare this ILP term with the recurrence-constrained and resource-constrained minimum initiation intervals (*RecMII* and *ResMII*) that are used as lower bounds in modulo scheduling.[19,20] In fact, a computation similar to *RecMII* is used to obtain the CP value for a given unroll vector, and a computation similar to *ResMII* is used to obtain the TC_j values for a given unroll vector. The key difference is that software pipelining and modulo scheduling are only concerned with analyzing multiple iterations of the innermost loop, whereas this ILP term is used for analyzing the combined effect of unrolling multiple loops in a perfect nest. The notion of initiation interval does not apply to noninnermost loops, which is why we use the CP term instead. An interesting direction for future work would be to combine both approaches by using this ILP cost model for noninnermost loops, and the initiation interval cost model for the innermost loop.

Note that summing up the contributions of the load/store term and the ILP term goes beyond the "balancing" approach proposed by Carr and Guan.[12] Specifically, there are cases in which it might be beneficial to reduce only one of the two terms even if doing so causes an imbalance between the terms.

As a final note, we briefly mention the effect of control flow within a loop iteration on cost estimation: For the register capacity terms, IR and FR, we use the *worst-case* largest number of registers that might be needed for executing a single iteration. For the load/store and ILP terms, we instead do an *average-case* estimation of the individual cost functions.[21, 22]

2.4. Algorithm for Selection of Unroll Factors

Our algorithm for selecting an optimized unroll vector is driven by the cost functions introduced in Section 2.3. The basic idea is to enumerate a set of feasible and profitable unroll vectors, compute the objective function for each one, and select the one with smallest objective function as the optimized unroll vector $(u_1^{\text{opt}},\ldots, u_k^{\text{opt}})$.

For feasibility, we have to ensure that an unroll vector $(u_1,\ldots, u_k)$ is legal and also that it satisfies the following capacity constraints[3]:

$$IR(u_1,\ldots, u_k) \leqslant \text{\# available fixed-point regs}$$

$$FR(u_1,\ldots, u_k) \leqslant \text{\# available floating-point regs}$$

$$u_1 \times \cdots \times u_k \leqslant \frac{\text{(size of instruction cache)}}{\text{(code size of one iteration)}}$$

Given two unroll vectors, $\mathbf{u}$ and $\mathbf{v}$, we say that $\mathbf{u}$ *dominates* $\mathbf{v}$ (written as $\mathbf{u} \succcurlyeq \mathbf{v}$) if and only if $u_1 \geqslant v_1,\ldots, u_k \geqslant v_k$, i.e., each unroll factor in $\mathbf{u}$ is at least as large as the corresponding unroll factor in $\mathbf{v}$. An important observation used to prune the search space for feasible unroll vectors is that the capacity constraints are *monotonic*, i.e., if unroll vector $\mathbf{v}$ is infeasible because it violates a capacity constraint, then all unroll vectors $\mathbf{u}$ such that $\mathbf{u} \succcurlyeq \mathbf{v}$ must also be infeasible.

Figure 3 outlines the high-level structure of the algorithm for selecting an optimized unroll vector. Step 1 calls function EnumerateFeasibleVectors() to obtain a set of feasible unroll vectors, UV. Step 3 selects $\mathbf{u}^{\text{opt}}$, the unroll vector from UV that has the smallest cost per iteration as the optimized unroll vector for the input loop nest.

Figure 4 outlines the structure of function EnumerateFeasibleVectors(). The algorithm enumerates unroll vectors by moving from the innermost loop to the outermost loop of the nest. Step 2 enumerates the possible unroll factors, $1 \cdots u_i^{\max}$, for input loop i, and combines each value with the input unroll vector, $\mathbf{u}^{\text{cur}}$ (Step 2a). The for-loop in Step 2 is exited

[3] We assume two register classes (fixed and float) in this description, but the approach can be easily adapted to a different number of register classes.

Inputs:

1. Set of k perfectly nested loops with maximum unroll vector, $\vec{u}^{max} = (u_1^{max}, \ldots, u_k^{max})$, as defined in Section 2.2.

2. $F(u_1, \ldots, u_k)$, objective cost function for loop nest defined in Section 2.3.

Output: $\vec{u}^{opt} = (u_1^{opt}, \ldots, u_k^{opt})$, an optimized unroll vector for input loop nest.

Algorithm:

1. /* Call function EnumerateFeasibleVectors() in Figure 4, with unit vector $\vec{1} = (1, \ldots, 1)$ as input. */
   UV := EnumerateFeasibleVectors(k, $\vec{1}$)

2. Initialize $\vec{u}^{opt} := \vec{1}$

3. **for each** unroll vector $\vec{u} \in UV$ **do**

   if $F(\vec{u}) < F(\vec{u}^{opt})$ **or** $(F(\vec{u}) = F(\vec{u}^{opt})$ **and** $(u_1 \times \ldots \times u_k) < (u_1^{opt} \times \ldots \times u_k^{opt})$) **then**

   /* Better unroll vector found */ $\vec{u}^{opt} := \vec{u}$

   end if

   end for

4. return $\vec{u}^{opt}$

Fig. 3. Algorithm for selecting an optimized unroll vector.

the first time an unroll factor is encountered for loop i that causes a capacity constraint to be exceeded (Step 2b). Step 2c implements a pruning heuristic—the for-loop is exited if increasing the unroll factor for loop i from 1 to 2 shows no improvement in the objective function. If i is the outermost loop, the current unroll vector is inserted into the output set (Step 2d.i). Otherwise, function EnumerateFeasibleVectors() is invoked recursively to enumerate unroll factors for enclosing loops $i, i-1, \ldots, 1$. The resulting set, UV', is then merged with the output set, UV (Steps 2e.i and 2e.ii).

2.5. Example

As an illustration, Fig. 5 compares the execution times of a 500×500 double-precision dense matrix multiply computation for different unroll factors. After tiling for cache locality, the inner tile of the matrix multiply kernel consists of three nested loops as follows:

```
do i1 = i1_lo, i1_hi
  do i2 = i2_lo, i2_hi
    do i3 = i3_lo, i3_hi
      a(i2,i1) = a(i2,i1) + b(i2,i3) * c(i3,i1)
    end do
  end do
end do
```

function EnumerateFeasibleVectors(i, $\vec{u}^{cur}$) **returns** UV

Inputs:

1. Index of current loop, i.
2. Current unroll vector, $\vec{u}^{cur}$, with unroll factors specified for loops in the range $i+1 \ldots k$. Unroll factors $u_1^{cur}, \ldots, u_i^{cur}$ are assumed to $= 1$.

Output: A *set of feasible unroll vectors*, UV, containing "expansions" of $\vec{u}^{cur}$. Each vector $\vec{u} \in UV$ satisfies Conditions 1–4 in Section 2.2, and also has the same unroll factors as $\vec{u}^{cur}$ in positions $i+1 \ldots k$ *i.e.,* only unroll factors in positions $1 \ldots i$ are enumerated in the expansion.

Algorithm:

1. Initialize UV := empty set of unroll vectors
2. **for** $n := 1$ **to** u_i^{max} **do**
   /* The u_i^{max} bound enforces Conditions 1 and 2 in Section 2.2 */
   (a) Update unroll factor for loop i, $u_i^{cur} := n$
   (b) **if** $\vec{u}^{cur}$ exceeds a capacity constraint (Condition 3 or Condition 4 in Section 2.2)
   then *break* /* exit for-loop */ **end if**
   (c) /* Pruning step — exit loop if no improvement is observed in the objective function by unrolling loop i */
   if $n > 1$ **and** $F(\vec{u}^{cur}) \geq F(\vec{u}^{prev})$
   then *break* /* exit for-loop */ **end if**
   (d) **if** $i = 1$ **then** /* i is the outermost loop */
      i. /* Insert $\vec{u}^{cur}$ into UV */
      $UV := UV \cup \{ \vec{u}^{cur} \}$
   (e) **else**
      i. /* Recursive call */
      $UV' :=$ EnumerateFeasibleVectors$(i-1, \vec{u}^{cur})$
      ii. /* Append UV' to UV */
      $UV := UV \cup UV'$
   end if
   end for
3. return UV

end function

Fig. 4. Function EnumerateFeasibleVectors().

Therefore, an unroll vector for the inner tile is specified by a (u_1, u_2, u_3) triple of unroll factors.

Figure 5 shows the execution time obtained for the matrix multiply kernel for different choices of unroll factors. To simplify the discussion in this section, we only consider two choices for each unroll factor value, $u_i = 1$ or $u_i = 4$, which leads to the eight possible values for the (u_1, u_2, u_3) triple enumerated along the horizontal axis. (Measurements of a larger set of unroll factors are presented in Fig. 8 in Section 4.) The (1, 1, 1) triple corresponds to the original loop nest because an unroll factor of one is an identity transformation. Other than unrolling of nested loops, all other optimization options are the same for the different unroll vectors shown in Fig. 5.

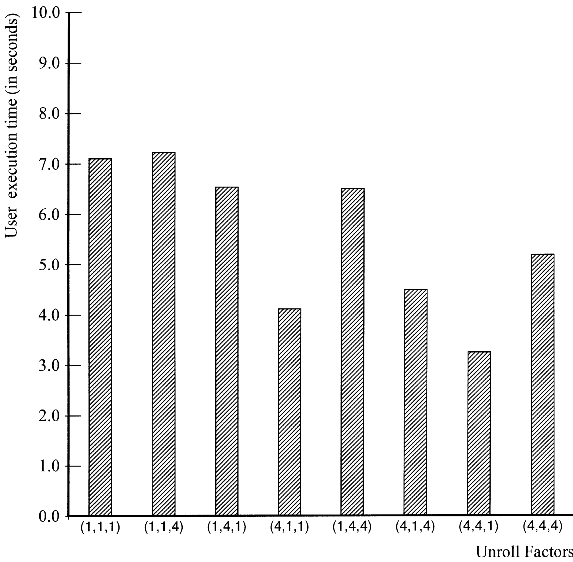

Fig. 5. Performance measurements on a 133 MHz PowerPC 604 processor for 500×500 matrix multiply example with different unroll factors.

For this example, we see that the performance obtained by unrolling nested loops varied significantly for different unroll vectors. The worst performance was obtained for $(u_1, u_2, u_3) = (1, 1, 4)$, which was slightly worse than that of the $(1, 1, 1)$ identity case. The best performance was obtained for $(u_1, u_2, u_3) = (4, 4, 1)$, which delivered a $2.2\times$ speedup.

We now describe how our approach can identify the $(4, 4, 1)$ unroll vector as the best candidate by using the cost functions and algorithm outlined in Sections 2.3 and 2.4. Note that a $(4, 4, 1)$ unroll vector is not likely to be obtained by commonly-used heuristics such as "unroll only the innermost loop" or "give all loops the same unroll factor."

Let (u_1, u_2, u_3) be a candidate unroll vector for the matrix multiply example. The most binding capacity constraint for this example is the number of floating-point registers, which is estimated by the compiler as $FR(u_1, u_2, u_3) = u_2 u_1 + (u_2 u_3 + u_3 u_1)$. This estimation follows directly from the presence of array references a(i,j), b(i,k), and c(k,j) (see Sarkar[15] and Ferrante *et al.*[18] for details). The $u_2 u_1$ term represents distinct unrolled copies of the loop-invariant references to array a, and the $(u_2 u_3 + u_3 u_1)$ term represents the number of registers required to hold distinct values of arrays b and c. Assuming that there are 30 registers available for used in the unrolled body, we need to ensure that $FR(u_1, u_2, u_3) \leqslant 30$ to satisfy the capacity constraints.

To estimate the objective function, $F(u_1, u_2, u_3)$, the compiler builds the following symbolic cost functions; we only show TC for the FPU (floating point unit), since the FPU is the critical resource for this example:

$$LS(u_1, u_2, u_3) = u_2 u_3 + u_3 u_1$$

$$CP(u_1, u_2, u_3) = 2u_3$$

$$TC_{FPU}(u_1, u_2, u_3) = 2u_1 u_2 u_3$$

$$NF_{FPU} = 1$$

$$\Rightarrow \quad F(u_1, u_2, u_3) = \frac{LS(u_1, u_2, u_3)}{u_1 \times u_2 \times u_3} + \frac{\max\left[CP(u_1, u_2, u_3), \dfrac{TC_{FPU}(u_1, u_2, u_3)}{NF_{FPU}}\right]}{u_1 \times u_2 \times u_3}$$

$$\Rightarrow \quad F(u_1, u_2, u_3) = \frac{(u_2 u_3 + u_3 u_1) + (2u_1 u_2 u_3)}{u_1 \times u_2 \times u_3}$$

$$\Rightarrow \quad F(u_1, u_2, u_3) = \frac{1}{u_1} + \frac{1}{u_2} + 2$$

Since $TC_{FPU}(u_1, u_2, u_3)/NF_{FPU} \geqslant CP(u_1, u_2, u_3)$, the ILP term for this example is resource bound rather than critical-path bound. However, if there were additional floating-point available (i.e., if $NF_{FPU} > 1$) then the ILP term may have been critical-path bound for some unroll vectors.

The algorithm selects values of u_1, u_2, u_3 so as to minimize $F(u_1, u_2, u_3) = 1/u_1 + 1/u_2 + 2$ subject to the constraint that $FR(u_1, u_2, u_3) = u_2 u_1 + u_2 u_3 + u_3 u_1$ is $\leqslant 30$. Note that the objective function for this example, $F(u_1, u_2, u_3)$, remains unchanged when u_3 is increased. Hence, the search space for optimal unroll vectors can be significantly reduced by restricting $u_3 = 1$ (see Step 2c in Fig. 4). Figure 6 illustrates how the algorithm for selection of unroll factors (outlined in Section 2.4) partitions the space of unroll vectors into feasible and infeasible regions for different values of u_1 and u_2, assuming a maximum unroll factor of 20 iterations in each dimension (a limit that may arise from the tile size used for cache tiling). All unroll vectors in the feasible region satisfy $FR \leqslant 30$. In the worst case, our algorithm will visit all 44 unroll vectors in the *feasible region* and the 10 unroll vectors along the *infeasible boundary*, but this is considerably less work than visiting all $20 \times 20 = 400$ possible values for $(u_1, u_2, 1)$ or all $20 \times 20 \times 20 = 8000$ possible values for (u_1, u_2, u_3).

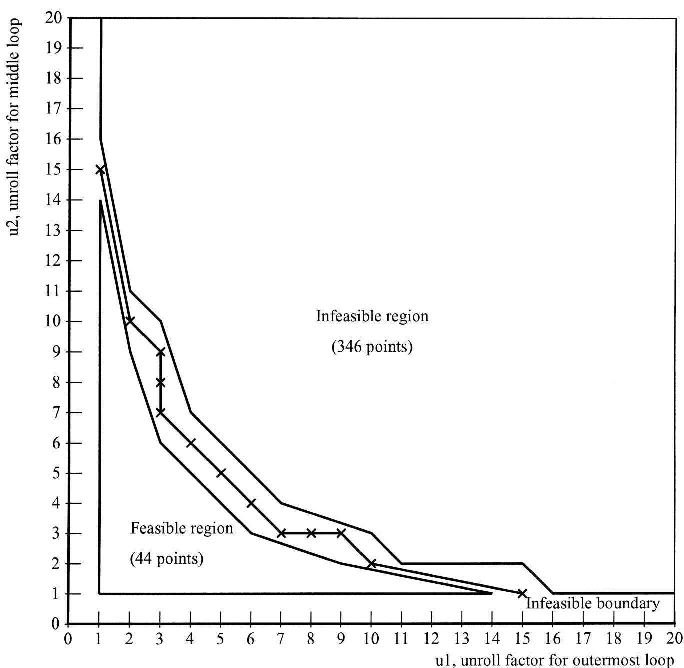

Fig. 6. Feasible and infeasible regions for enumeration of unroll vectors, assuming $u_3 = 1$.

Note that the simplified cost function discussed earlier, $F(u_1, u_2, u_3) = 1/u_1 + 1/u_2 + 2$ is a monotonically nonincreasing function of unroll factors. Given two unroll vectors **u** and **v**, if $\mathbf{u} \succcurlyeq \mathbf{v}$ then it must be the case that $F(\mathbf{u}) \leqslant F(\mathbf{v})$. This observation can be used to further prune our search by limiting our attention to the "largest" unroll vectors in the feasible region, i.e., feasible unroll vectors that are not dominated by another feasible unroll vector. The monotonicity of the cost function guarantees that since this set must contain at least one optimal solution. For the matrix multiply example, this pruning can reduce the search space from 44 vectors to 14 vectors. However, this pruning cannot be performed for more general non-monotonic cost functions that arise in practice due to ILP and other considerations.[4] In Section 5.1, we will see an example of a nonmonotonic cost

[4] In theory, the same consideration might argue against the heuristic in Step 2c of Fig. 4. However, we have never encountered a case in practice where it is profitable to unroll a loop with an unroll factor > 2, but the cost of using an unroll factor $= 2$ is worse than the no-unroll case.

function that is a result of considering the possibility of generating dual-word load/store instructions.

There are two optimal solutions to this constrained optimization problem, $(u_1, u_2, u_3) = (4, 5, 1)$ and $(u_1, u_2, u_3) = (5, 4, 1)$, both of which use a total of $FR = 29$ floating-point registers in the unrolled loop body. Increasing u_1 to 5 makes FR equal 35, which exceeds the limit. Of the eight unroll vectors measured in Fig. 5, our cost functions show that (4, 4, 1) should indeed be the best choice. (It is closest to the optimal (4, 5, 1) and (5, 4, 1) solutions.)

3. GENERATION OF TRANSFORMED CODE

In this section, we outline how our compiler generates code for a specified unroll vector, $(u_1, \ldots, u_k)$. The algorithm processes loops by moving from the outermost loop to the innermost loop of the nest. Let i be the current loop with unroll factor u_i. First, the current unrolled loop body is expanded by the specified unroll factor u_i. Second, the loop header for the current loop is adjusted so that if the loop's iteration count, C_i, is known to be less than or equal to the unroll factor, u_i, then the loop is totally unrolled by simply replacing the loop header by an assignment of the index variable to the lower-bound expression; otherwise, the loop header is adjusted so that the unrolled loop's iteration count equals $\lfloor C_i/u_i \rfloor$. Third, a remainder loop nest is generated, if needed. The body of the remainder loop nest is a single copy of the input loop body. The remainder loop is not created if it is determined at compile time that the loop length C_i is a multiple of the unroll factor u_i.

In general, our algorithm produces $u_1 \times \cdots \times u_k$ copies of the code from the original loop body in the unrolled loop. In addition, the number of remainder loops produced by our algorithm is

$$\sum_{1 \leqslant i \leqslant j} \prod_{1 \leqslant h < j} u_h = (u_1 \times \cdots \times u_{j-1}) + (u_1 \times \cdots \times u_{j-2}) + \cdots + (u_1) + 1$$

where j is the largest loop index with a non-identity unroll factor, i.e., with $u_j > 1$. Each remainder loop contains a single copy of the code from the original loop body. In contrast, the unroll-and-jam transformation produces $(u_1 + mod(1, u_1)) \times \cdots \times (u_k + mod(1, u_k))$ copies of the code from the original loop body.[5]

Appendix A contains an example to highlight the difference between our code generation and the code generation obtained by the unroll-and-

[5] $mod(1, x)$ is a function that is $= 0$ if $x = 1$ and is $= 1$ otherwise (assuming that $x > 0$).

jam approach. For this example, our algorithm generated 21 remainder loops as opposed to 61 remainder loops generated by the unroll-and-jam approach. For the sake of completeness, a complete description of our algorithm for generating compact code when unrolling multiple nested loops is provided in Fig. 7.

Inputs:

1. $LOOP[1], \ldots, LOOP[k]$, a perfect nest of k loops, numbered from outermost to innermost. The index variable, lower bound, upper bound, and increment for $LOOP[j]$ are denoted by i_j, lb_j, ub_j, and inc_j.
2. $u[j] \geq 1$, an unroll factor for each $LOOP[j]$.
3. $C[j]$,= constant value or symbolic expression for number of iterations executed by $LOOP[j]$.

Output: Updated intermediate representation of the unrolled loops to reflect the loop unrolling transformation specified by unroll factors $u[1], \ldots, u[k]$.

Algorithm:

1. Initialize *nextParent* := parent of $LOOP[1]$ in intermediate representation
2. Detach subtree rooted at $LOOP[1]$ from *nextParent* // Use as source for generating additional copies
3. Initialize *unrolledBody* := copy of body of innermost loop, $LOOP[k]$
4. **for** $j := 1$ **to** k **do**
   (a) *currentParent* := *nextParent*
   (b) /* Expand *unrolledBody* by factor $u[j]$ for index i_j */
   Initialize *newUnrolledBody* := copy of *unrolledBody*
   for $u := 1$ **to** $u[j]$ **do**
      i. Initialize *oneCopy* = copy of *unrolledBody*, and replace all occurrences of "i_j" in *oneCopy* by "$i_j + inc_j*u$"
      ii. Append *oneCopy* to end of *newUnrolledBody*
   end for
   Delete old *unrolledBody*, and initialize *unrolledBody* := *newUnrolledBody*
   (c) /* Adjust header for unrolled loop j */
   Construct remainder expression $er_j = \mathrm{mod}(C[j], u[j])$
   if ($C[j]$ is constant and $C[j] = u[j]$) **then**
      - /* Loop j is to be completely unrolled */
      Construct the statement, "$i_j = lb_j$", call it *nextParent*, and make it the first child of *currentParent*
   else
      - Make a copy of the $LOOP[j]$ statement, call it *nextParent*, change it to
      "do $i_j = lb_j$, ub_j - er_j*inc_j, $u[j]*inc_j$", and make it a child of *currentParent*
   end if
   (d) /* Generate remainder loop sub-nest, if necessary */
   if ($er_j != 0$) **then**
      i. Set treeCopy := copy of subtree rooted at $LOOP[j]$, change the outermost statement in treeCopy to
      "do $i_j = ub_j$ - $(er_j\text{-}1)*inc_j$, ub_j, inc_j", and Make treeCopy a child of *currentParent*
   end if
5. Make *unrolledBody* a child of *nextParent*, and delete original subtree rooted at $LOOP[1]$

Fig. 7. Code generation algorithm.

4. EXPERIMENTAL RESULTS

In this section, we present experimental results to evaluate our approach for optimized unrolling of nested loops. The algorithm outlined in Section 2.4 has been implemented in the IBM XL Fortran product compiler. This loop unrolling phase is performed as a "high-order" transformation[15] so that back-end optimizations can exploit the code optimization opportunities created by loop unrolling. All runtime performance measurements were made on a 133 MHz PowerPC 604 processor. The performance measurements reported in this paper were obtained using version 5.0 of the IBM XL Fortran compiler with the following options turned on in all cases: `-O2` (optimization level 2), `-qarch=604`, `-qtune=604` (identifies target processor as PowerPC 604), **-qhot** (enables high-order transformations[15]). We will later refer to this set of options as "default optimization."

First, we present some detailed performance measurements for the matrix multiply example discussed in Section 2.5. Figure 8 shows the user execution times measured for 100 different unroll vectors of the form $(u_1, u_2, 1)$ for $1 \leqslant u_1, u_2 \leqslant 10$. Recall that u_1 and u_2 are the unroll factors for the the outer and middle loops respectively. We set the unroll factor for the innermost loop to $u_3 = 1$ for all the 100 data points because the cost function analysis in Section 2.5 revealed that unrolling the innermost loop would not deliver any performance benefit. (This was confirmed by the

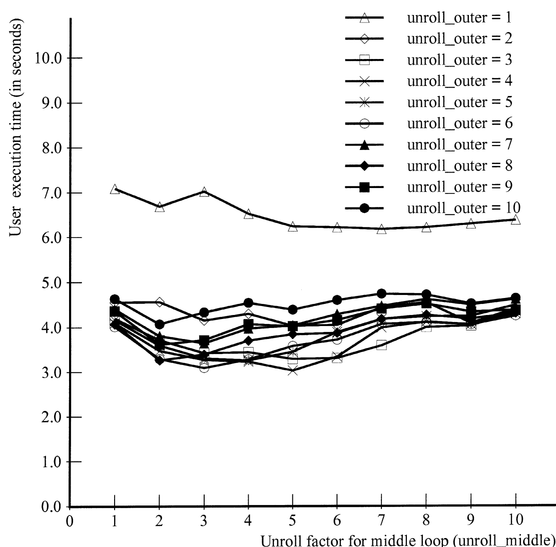

Fig. 8. Detailed performance measurements on a 133 MHz PowerPC 604 processor for 500×500 matrix multiply example with different unroll factors.

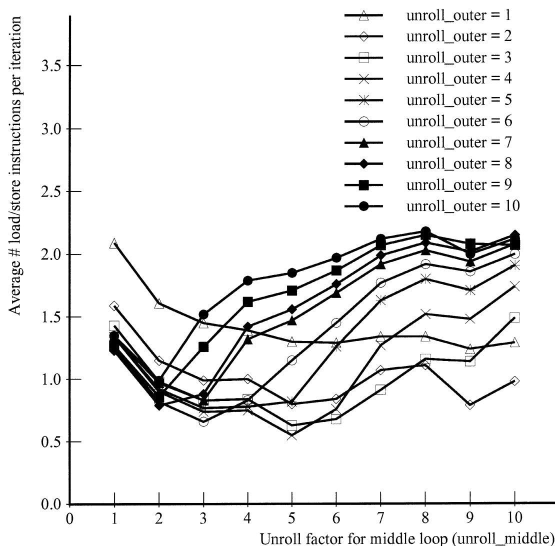

Fig. 9. Average number of loads and stores per original iteration for 500×500 matrix multiply example with different unroll factors.

results in Fig. 5 as well.) The unroll vector (4, 5, 1) that was identified in Section 2.5 as the optimal solution for this example indeed delivered the best performance in Fig. 8. Since register locality is the most significant performance issue for loop unrolling in this example, Fig. 9 shows the average number of loads and stores per original iteration for these 100 iterations. The average drops from 2.1 for the original loop identified by unroll vector (1, 1, 1) to 0.55 for unroll vector (4, 5, 1) represents a nearly $4\times$ reduction in the number of load/store instructions executed. These averages were obtained by using the hardware performance monitor to measuring the total number of load/store instructions executed and then dividing that number by the number of times the inner loop is executed ($500 \times 500 \times 500 = 1.25 \times 10^8$).

Table I summarizes the execution times obtained on seven SPEC95fp benchmark programs[23] for the following unroll configurations:

- NO-UNROLL—default optimization with unrolling suppressed (except for the $2\times$ unrolling performed by software pipelining in the back-end).
- (2, 2, 2)—default optimization with all loops in an innermost perfect loop nest assigned an unroll factor of two. (There was no innermost perfect loop nest encountered with > 3 loops in these benchmarks.)

Table I. SPEC95fp Benchmarks Compiled with Different Unroll Configurations[a]

Benchmark	NO-UNROLL	(2, 2, 2)	(3, 3, 3)	(4, 4, 4)	(5, 5, 5)	OPT-UNROLL
101.tomcatv	1317.0	1184.6	1256.8	1287.8	1375.3	1073.2
102.swim	2202.6	2127.4	2556.2	2928.7	3030.1	1836.4
103.su2cor	795.0	769.1	751.8	776.4	770.9	775.0
104.hydro2d	1581.3	1486.3	1496.7	1522.8	1469.8	1491.3
107.mgrid	1014.8	964.4	1024.5	1060.2	1407.1	1015.6
125.turb3d	1006.9	1028.1	1071.3	1207.9	1128.7	1007.3
145.fpppp	1181.1	1189.2	1216.5	1173.9	1469.8	1159.6

[a] Executed on a 133 MHz Power 604 (user execution times in seconds).

- (3, 3, 3)—default optimization with all loops in an innermost perfect loop nest assigned an unroll factor of three.
- (4, 4, 4)—default optimization with all loops in an innermost perfect loop nest assigned an unroll factor of four.
- (5, 5, 5)—default optimization with all loops in an innermost perfect loop nest assigned an unroll factor of five.
- OPT-UNROLL—default optimization with unrolling performed using the algorithm reported in this paper.

Table II shows the speedups obtained relative to NO-UNROLL, for the execution times reported in Table I. The average speedup of 1.08 × delivered by OPT-UNROLL outperformed that of the other unroll configurations measured. The maximum speedup delivered by OPT-UNROLL on a SPEC95fp benchmark was 1.2 ×, observed for two of the benchmarks

Table II. Speedups for SPEC95fp Benchmarks with Relative NO-UNROLL[a]

Benchmark	NO-UNROLL	(2, 2, 2)	(3, 3, 3)	(4, 4, 4)	(5, 5, 5)	OPT-UNROLL
101.tomcatv	1.00	1.11	1.05	1.02	0.96	1.23
102.swim	1.00	1.04	0.86	0.75	0.73	1.20
103.su2cor	1.00	1.03	1.06	1.02	1.03	1.03
104.hydro2d	1.00	1.06	1.06	1.04	1.08	1.06
107.mgrid	1.00	1.05	0.99	0.96	0.72	1.00
125.turb3d	1.00	0.98	0.94	0.83	0.89	1.00
145.fpppp	1.00	0.99	0.97	1.01	0.80	1.02
Average speedup	1.00	1.04	0.99	0.95	0.89	1.08

[a] Executed on a 133 MHz Power 604 (user execution times in seconds).

(101.tomcatv and 102.swim). It is also important to note that, unlike all the other unrolling configurations, OPT-UNROLL never delivered a performance degradation.

Thus, the results in this section demonstrate the effectiveness of the approach presented in this paper for optimized unrolling of nested loops. We believe that larger performance improvements due to unrolling of nested loops can be expected on processors that have larger numbers of registers and larger degrees of instruction-level parallelism than the processor used for our measurements (a PowerPC 604).

5. EXTENSIONS

In this section, we outline extensions to our algorithm for selecting unroll factors to support other optimizations that interact with unrolling. Section 5.1 describes how the unrolling cost functions can be extended to model the use of *dual-word load/store instructions*. Section 5.2 discusses how unrolling can be performed on *reduction loops*. Section 5.3 describes how *statement interleaving* can be performed in conjunction with loop unrolling. Finally, Section 5.4 outlines how the unrolling cost functions can be extended to exploit *data-prefetch* instructions.

All but the last of these extensions have been implemented in the ASTI optimizer component of the IBM XL Fortran product compilers. (The prefetch extension has only been implemented in a prototype extension of the product compilers.) However, none of these extensions were enabled when obtaining the experimental results presented in Section 4. The extensions in Section 5.1 and 5.3 are only enabled when compiling for the IBM Power2 processor.[24] The extension in Section 5.2 is only enabled when using the `-qnostrict` option in the XL Fortran compiler.

5.1. Generation of Dual-Word Load/Store Instructions

Some processors support dual-word load/store operations that can load/store a pair of adjacent memory locations into adjacent registers in a single instruction.[6] Since a dual-word instruction moves two words in less time that two single-word instructions, the use of dual-word instructions can reduce the effective cost of load/store operations in a program. In the IBM Power2 processor,[24] a *quad-word* instruction is used to move two adjacent double-word locations into adjacent registers, thus displaying the same behavior for double-precision data that dual-word instructions exhibit for single-precision data.

In practice, loop unrolling is necessary to expose opportunities for generating dual-word instructions. Consider the following vector-sum loop as an example:

```
real*4 A(1:n), sum
do i = 1, n
  sum = sum + A(i)
end do
```

There is no opportunity to generate a dual-word instruction for this loop because the loop body contains only a single load operation, A(i). Assuming (for simplicity) that the CPU cost of the load instruction is 1 cycle, we obtain $LS = 1$ for the original loop with no unrolling which results in a load/store contribution of 1 cycle/iteration to the cost function defined in Section 2.3.

However, if the loop is unrolled by a factor of 2, the main unrolled loop becomes:

```
do i = 1, n-1, 2
  sum = sum + A(i)
  sum = sum + A(i+1)
end do
```

It now becomes possible to combine the reads of A(i) and A(i+1) into a dual-word load instruction, thus resulting in a cost of $LS = 1$ for an unroll factor of $u_i = 2$ (assuming for simplicity that the cost of a dual-word load instruction is the same as that of a single-word load instruction). This contributes 0.5 cycles/iteration to the load/store term in the cost function, thus resulting in a smaller cost than the no-unroll configuration.

In general, the load/store term for this example can be written as the following function of the unroll factor u_i:

$$\frac{LS(u_i)}{u_i} = \frac{\lceil u_i/2 \rceil}{u_i} \text{ cycles/iteration}$$

An interesting property of this cost function is that all even-valued unroll factors contribute the same cost (0.5), whereas odd values contribute larger costs that diminish and approach 0.5 as the unroll factor is increased. This is an example of a non-monotonic cost function alluded to in Section 2.5 that forces our algorithm to examine all feasible unroll vectors rather than only the largest ones.

5.2. Unrolling of Reduction Loops

In the previous section, we saw how unrolling can reduce the load/store cost of a vector-sum loop by using dual-word load instructions. However, straight unrolling did not contribute any reduction in the ILP term for this example. When we examine the unrolled loop in Section 5.1,

we see that there is a chain of dependences between the array load and the sum operations that causes the amortized critical path cost, $CP(u_i)/u_i$ to remain constant.

However, the knowledge that the computation being performed by the loop is a *sum reduction* can be used to transform the unrolled loop into a form that has a lower ILP cost. The idea is to use a "partial sums" approach and allocate a separate accumulator variable for each copy in the unrolled loop body, and then combine the partial sums at the end after exiting from the unrolled loop. The output of this transformation for the example in Section 5.1 is as follows:

```
! INITIALIZE ACCUMULATORS
sum1 = sum
sum2 = 0.0
! UNROLLED LOOP
do i = 1, n-1, 2
  sum1 = sum1 + A(i)
  sum2 = sum2 + A(i+1)
end do
! REMAINDER LOOP
sum = sum1 + sum2
do i = i, n
  sum = sum + A(i)
end do
```

It results in a decrease in the ILP cost since the two partial sum computations in the unrolled loop body are independent.

Note that this transformation is algebraically correct because addition is commutative and associative. However, the transformation is not guaranteed to compute results that are bitwise identical to that of the original loop. Therefore, it can only be performed under a user-controlled option that permits the compiler to reorder such operations (e.g., the -qnostrict option in the XL Fortran compiler).

If such an option is supplied, our extension to support unrolling of reduction loops can be summarized as follows. First, identify all loops in the loop nest that carry reductions. (Our current implementation only identifies sum and product computations as reductions, though this approach could also be used for min and max computations.) Next, update the CP cost function so that partial sum/product computations are considered to be independent when reduction loops are unrolled. Finally, modify the code generation algorithm so that partial accumulators are correctly initialized and combined in the transformed code.

5.3. Statement Interleaving

A practical issue that arises when performing high-level transformations such as loop unrolling is that an optimizing back-end is often unable to perform the same extent of code motion as a high-level optimizer because it lacks high-level information on memory accesses. Consider the following loop as an example:

```
do i = 1, n
  A(i,1) = A(i,1) + B(i)
  A(i,2) = A(i,1) + C(i)
end do
```

Note that the operations within a single iteration form a dependence chain and thus exhibit poor ILP.

After unrolling this loop by a factor of 2, the main unrolled loop becomes:

```
do i = 1, n-1, 2
  A(i,1) = A(i,1) + B(i)
  A(i,2) = A(i,1) + C(i)
  A(i+1,1) = A(i+1,1) + B(i+1)
  A(i+1,2) = A(i+1,1) + C(i+1)
end do
```

Since the i loop is parallel, the critical path length for two copies of the loop body is same as for one copy. Hence, unrolling by a factor of 2 effectively reduces the $CP(u_i)/u_i$ ratio in the ILP cost by a factor of 2.

However, many optimizing back-ends are unable to exploit the improved ILP in the unrolled loop above because of their inability to disambiguate among multiple stores to array A. (This is usually because they operate on an intermediate representation that converts array references to low-level load/store operations, and do not perform the analysis necessary to reconstruct the original subscript expressions.) To ensure that unrolling of parallel loops leads to effective exploitation of ILP, we perform an additional transformation that we call "statement interleaving." The idea behind this transformation is to interleave the statements generated for the unrolled loop body, so as to reduce the amount of code motion that the back-end would need to do to exploit ILP. The main legality constraint for statement interleaving is that it can only be performed when unrolling parallel loops.

The output obtained by performing this transformation on the above example is as follows:

```
do i = 1, n-1, 2
  A(i,1) = A(i,1) + B(i)
  A(i+1,1) = A(i+1,1) + B(i+1)
  A(i,2) = A(i,1) + C(i)
  A(i+1,2) = A(i+1,1) + C(i+1)
end do
```

ILP exploitation becomes easier because all (parallel) copies of a statement are made adjacent. Statement interleaving can also expose additional opportunities for generating dual-word load/store instructions.

5.4. Data-Cache Prefetching

Many processors support a *prefetch* instruction that can be used to initiate the fetching of a cache block prior to its use, so that the transfer delay can be overlapped with useful computation thus reducing the overhead of a cache miss. The problem of automatically inserting prefetch instructions for improved performance has received a lot of attention in past work, e.g., see Mowry.[7]

In our framework, a prefetch optimization phase is performed prior to loop unrolling, but after loop distribution, iteration-reordering transformations, and loop fusion.[15] [As discussed later, there is one prefetching parameter called the *iteration offset* that is computed *after* loop unrolling.] The output of this phase identifies a set of array references that have been selected for prefetching, because their *memory cost* exceeds a given threshold. Each reference selected for prefetching is annotated with a *replication vector*, *rep*[ ], indexed by loops in the loop nest. $rep[i]$ identifies the replication frequency for the array reference, when loop i is unrolled; if $rep[i] = r$, a single prefetch instruction is inserted after every r unrolled iterations of the i loop. The replication vector is computed by setting $rep[i] = \lceil B/s_i \rceil$ for each loop i, where B is the size of a data-cache block and s_i is the *stride* that the array reference would exhibit if loop i was the innermost loop.

For example, consider a data cache with block size, $B = 32$ bytes, and a reference $A(i)$ to a double-precision array A which results in a stride of $s_i = 8$ bytes. The replication frequency for loop i in array reference $A(i)$ is computed as $rep[i] = \lceil 32/8 \rceil = 4$, i.e., one prefetch instruction is inserted for every 4 unrolled iterations of loop i. Now consider a two-dimensional array reference $B(j, i)$ for which $s_i =$ (column size of array B), which can be arbitrarily large (say, 8000 bytes). The replication frequency for loop i in array reference $B(j, i)$ will be computed as $rep[i] = \lceil 32/8000 \rceil = 1$, i.e., one prefetch instruction is inserted for each unrolled iteration of loop i.

For simplicity, the only impact of prefetch instructions on the unrolling cost function modeled in our framework is on the load/store term in Section 2.3. Each prefetch instruction in the unrolled loop adds an extra cost to this term, P_C, which is the cost incurred by the CPU to execute the prefetch instruction and initiate the prefetch. Note that this cost does not include the actual *prefetch delay*, i.e., the time spent in the memory system to complete the prefetch.

In general, the number of cycles contributed by prefetch instructions to the *per-iteration* load/store cost is:

$$\frac{\prod_{1 \leqslant i \leqslant k} \left\lceil \frac{u_i}{rep[i]} \right\rceil \times P_C}{\prod_{1 \leqslant i \leqslant k} u_i}$$

Consider array reference $A(i)$ with $rep[i]=4$, as an example. If $P_C=1$, the average number of cycles contributed by prefetch instructions per iteration will equal 1 for $u_i=1$, $1/2$ for $u_i=2$, $1/3$ for $u_i=3$, $1/4$ for $u_i=4$, $2/5$ for $u_i=5$, and so on. Now consider array reference $B(j, i)$ with $rep[i]=1$. The number of cycles per iteration contributed by prefetch instructions for this array reference equals 1 for all unroll factors, u_i.

Thus, we see that the modeling of prefetch costs is nonmonotonic in unroll factors, similar to the modeling of dual-word load/store instructions. However, the prefetch costs can still be incorporated into the cost function from Section 2.3, so that it can be combined with other performance considerations for loop unrolling. This approach can also be extended to support unrolling in the presence of *flush* and *invalidate* instructions, when there are performance benefits or other reasons for inserting these instructions in optimized code.[8, 25]

We conclude this section with a brief mention of the *prefetch iteration offset*, O, which identifies the offset between the iteration containing a prefetch instruction and the iteration containing the memory access targeted by the prefetch instruction. The prefetch iteration offset is computed as

$$O = \left\lceil \frac{\text{(Prefetch delay)}}{\text{(Cost of single iteration of unrolled loop)}} \right\rceil$$

so as to ensure that O iterations of the innermost loop will be sufficient to overlap the delay of a single iteration. The cost of a single iteration of the unrolled loop is computed by evaluating the cost function in Section 2.3 for the final unroll configuration selected. (That is why the prefetch iteration offset can only be computed after unroll factors have been selected.) Note that the value of O is independent of the array reference. The same value is used for all prefetch instructions.

6. RELATED WORK

As mentioned earlier, the loop unrolling and the unroll-and-jam transformations have been in use for over three decades.[1] However, little attention has been paid until recently to the problem of automatically selecting unroll factors to obtain the best performance from loop unrolling. For example, Wolf and Lam presented experimental results for register tiling in conjunction with cache tiling[14] using the SUIF compiler, but the register tiling in that work was implemented by hand.

The most closely related work to this paper is that of Carr and Kennedy[9] and by Carr and Guan.[12] Some of the key differences between our approach and their approaches[9, 12] have already been discussed in Section 1. Another difference that is worth mentioning is that the objective function in their approaches[9, 12] is to *balance* floating-point and memory-access instructions, whereas the objective function in our approach is to reduce execution time. These two objective functions are not necessarily equivalent. (As observed in Hailperin's review[26] of Carr and Kennedy,[5] "The most significant gap in the experimental results concerns the connection between floating-point/memory-access balance and performance.".) For example, the best results for the matrix multiply example discussed in this paper were obtained when the average number of loads is driven down to 0.5 loads per original iteration (see Figs. 8 and 9), even though each iteration has two floating-point operations. It is unclear from the descriptions by Carr and Kennedy[9] and Carr and Guan;[12] how a similar configuration would be obtained with their goal of balancing memory instructions and floating-point instructions.

Most of the other related work applies only to unrolling innermost loops rather than nested loops. Several industry compilers (including the baseline XL Fortran compiler used to obtain our experimental results) perform unrolling of (both counted and noncounted) innermost loops. The problem of combining loop unrolling with software pipelining has also received a lot of attention. Weiss and Smith[27] studied unrolling of a single innermost loop and compared it with software pipelining. Their conclusion was that loop unrolling can deliver greater speedup than software pipelining, but requires more hardware (more registers and a larger instruction buffer) to do so. Jones and Allan[28] suggested that loop unrolling be performed before software pipelining to effectively obtain a noninteger initiation interval. In their work, the unroll factor is determined by the desired initiation interval rather than by specific register and/or ILP cost considerations. Su *et al.*[29] proposed the URPR algorithm (unroll, pipeline, reroll) as a way of combining loop unrolling and instruction scheduling. Lavery and Hwu[30] evaluated the benefits of unrolling loops prior to modulo scheduling. In our approach, unrolling of nested loops is performed prior to software pipelining in the XL Fortran back end.

7. CONCLUSIONS

In this paper, we formalized selection of unroll factors for multiple perfectly nested loops as an optimization problem. We introduced an objective function to estimate the savings that will be obtained for a given vector of unroll factors, and capacity cost functions to model register set and I-cache constraints, and we specified the legality constraints for unrolling loops in a perfect nest. We outlined an algorithm for efficiently enumerating feasible unroll vectors (legal configurations that satisfy the capacity constraints) and selecting an unroll vector that delivers the best savings. We also addressed the problem of generating compact code for the remainder loops resulting from an unroll transformation on nested loops, and showed how our approach can generate fewer remainder loops than the classical unroll-and-jam approach. Our experimental results on seven SPEC95fp benchmarks using the XL Fortran compiler validated the robustness of our approach and demonstrated its effectiveness for use in industry-strength compilers. We expect to see larger performance improvements due to unrolling of nested loops on processors that have larger numbers of registers and larger degrees of instruction-level parallelism than the processor used for our measurements (PowerPC 604).

Possibilities for future work include extensions of the cost functions presented in this paper to handle new processor features such as multimedia extensions and combining our cost model with the initiation interval cost models used in software pipelining and modulo scheduling. An important extension in code generation support would be to enable unrolling of triangular/trapezoidal loops, similar to tiling of triangular/trapezoidal loops.[17]

ACKNOWLEDGMENTS

The author would like to thank Khoa Nguyen was his contribution to the algorithm for generating compact code when unrolling multiple nested loops, and Krishna Palem and Barbara Simons for their contributions to the algorithm for selection of unroll factors. The author would also like to thank members of the original ASTI optimizer group at IBM Santa Teresa Laboratory for their contributions to the design and initial implementation of the ASTI optimizer during 1991–1993, and members of the Parallel Development group in the IBM Toronto Laboratory for their ongoing work since 1994 on extending and shipping the ASTI optimizer as part of the IBM XL FORTRAN compiler products.

APPENDIX A

A.1. EXAMPLE OF GENERATING COMPACT CODE FOR UNROLLING MULTIPLE LOOPS

Consider generating code for unroll vector (4, 4, 4, 1) for the following example nest of four loops (such an unroll vector may be selected due to register locality considerations):

```
do l = 1, n-3, 4
   do k = 1, n-3, 4
      do j = 1, n-3, 4
         do i = 1, n
            sum = sum+a(i,j,k)+b(i,j,l)+c(i,k,l)
            sum = sum+a(i,j+1,k)+b(i,j+1,l)+c(i,k,l)
            sum = sum+a(i,j+2,k)+b(i,j+2,l)+c(i,k,l)
            sum = sum+a(i,j+3,k)+b(i,j+3,l)+c(i,k,l)
            sum = sum+a(i,j,k+1)+b(i,j,l)+c(i,k+1,l)
            sum = sum+a(i,j+1,k+1)+b(i,j+1,l)+c(i,k+1,l)
            sum = sum+a(i,j+2,k+1)+b(i,j+2,l)+c(i,k+1,l)
            sum = sum+a(i,j+3,k+1)+b(i,j+3,l)+c(i,k+1,l)
            sum = sum+a(i,j,k+2)+b(i,j,l)+c(i,k+2,l)
            sum = sum+a(i,j+1,k+2)+b(i,j+1,l)+c(i,k+2,l)
            sum = sum+a(i,j+2,k+2)+b(i,j+2,l)+c(i,k+2,l)
            sum = sum+a(i,j+3,k+2)+b(i,j+3,l)+c(i,k+2,l)
            sum = sum+a(i,j,k+3)+b(i,j,l)+c(i,k+3,l)
            sum = sum+a(i,j+1,k+3)+b(i,j+1,l)+c(i,k+3,l)
            sum = sum+a(i,j+2,k+3)+b(i,j+2,l)+c(i,k+3,l)
            sum = sum+a(i,j+3,k+3)+b(i,j+3,l)+c(i,k+3,l)
            sum = sum+a(i,j,k)+b(i,j,l+1)+c(i,k,l+1)
            sum = sum+a(i,j+1,k)+b(i,j+1,l+1)+c(i,k,l+1)
            sum = sum+a(i,j+2,k)+b(i,j+2,l+1)+c(i,k,l+1)
            sum = sum+a(i,j+3,k)+b(i,j+3,l+1)+c(i,k,l+1)
            sum = sum+a(i,j,k+1)+b(i,j,l+1)+c(i,k+1,l+1)
            sum = sum+a(i,j+1,k+1)+b(i,j+1,l+1)+c(i,k+1,l+1)
            sum = sum+a(i,j+2,k+1)+b(i,j+2,l+1)+c(i,k+1,l+1)
            sum = sum+a(i,j+3,k+1)+b(i,j+3,l+1)+c(i,k+1,l+1)
            sum = sum+a(i,j,k+2)+b(i,j,l+1)+c(i,k+2,l+1)
            sum = sum+a(i,j+1,k+2)+b(i,j+1,l+1)+c(i,k+2,l+1)
            sum = sum+a(i,j+2,k+2)+b(i,j+2,l+1)+c(i,k+2,l+1)
            sum = sum+a(i,j+3,k+2)+b(i,j+3,l+1)+c(i,k+2,l+1)
            sum = sum+a(i,j,k+3)+b(i,j,l+1)+c(i,k+3,l+1)
            sum = sum+a(i,j+1,k+3)+b(i,j+1,l+1)+c(i,k+3,l+1)
            sum = sum+a(i,j+2,k+3)+b(i,j+2,l+1)+c(i,k+3,l+1)
            sum = sum+a(i,j+3,k+3)+b(i,j+3,l+1)+c(i,k+3,l+1)
            sum = sum+a(i,j,k)+b(i,j,l+2)+c(i,k,l+2)
            sum = sum+a(i,j+1,k)+b(i,j+1,l+2)+c(i,k,l+2)
            sum = sum+a(i,j+2,k)+b(i,j+2,l+2)+c(i,k,l+2)
            sum = sum+a(i,j+3,k)+b(i,j+3,l+2)+c(i,k,l+2)
            sum = sum+a(i,j,k+1)+b(i,j,l+2)+c(i,k+1,l+2)
            sum = sum+a(i,j+1,k+1)+b(i,j+1,l+2)+c(i,k+1,l+2)
            sum = sum+a(i,j+2,k+1)+b(i,j+2,l+2)+c(i,k+1,l+2)
            sum = sum+a(i,j+3,k+1)+b(i,j+3,l+2)+c(i,k+1,l+2)
            sum = sum+a(i,j,k+2)+b(i,j,l+2)+c(i,k+2,l+2)
            sum = sum+a(i,j+1,k+2)+b(i,j+1,l+2)+c(i,k+2,l+2)
            sum = sum+a(i,j+2,k+2)+b(i,j+2,l+2)+c(i,k+2,l+2)
            sum = sum+a(i,j+3,k+2)+b(i,j+3,l+2)+c(i,k+2,l+2)
            sum = sum+a(i,j,k+3)+b(i,j,l+2)+c(i,k+3,l+2)
            sum = sum+a(i,j+1,k+3)+b(i,j+1,l+2)+c(i,k+3,l+2)
            sum = sum+a(i,j+2,k+3)+b(i,j+2,l+2)+c(i,k+3,l+2)
            sum = sum+a(i,j+3,k+3)+b(i,j+3,l+2)+c(i,k+3,l+2)
            sum = sum+a(i,j,k)+b(i,j,l+3)+c(i,k,l+3)
            sum = sum+a(i,j+1,k)+b(i,j+1,l+3)+c(i,k,l+3)
            sum = sum+a(i,j+2,k)+b(i,j+2,l+3)+c(i,k,l+3)
            sum = sum+a(i,j+3,k)+b(i,j+3,l+3)+c(i,k,l+3)
            sum = sum+a(i,j,k+1)+b(i,j,l+3)+c(i,k+1,l+3)
            sum = sum+a(i,j+1,k+1)+b(i,j+1,l+3)+c(i,k+1,l+3)
            sum = sum+a(i,j+2,k+1)+b(i,j+2,l+3)+c(i,k+1,l+3)
            sum = sum+a(i,j+3,k+1)+b(i,j+3,l+3)+c(i,k+1,l+3)
```

Fig. 10. Generated code using unroll-and-jam transformation (Part 1 of 4).

```
            sum = sum+a(i,j,k+2)+b(i,j,l+3)+c(i,k+2,l+3)
            sum = sum+a(i,j+1,k+2)+b(i,j+1,l+3)+c(i,k+2,l+3)
            sum = sum+a(i,j+2,k+2)+b(i,j+2,l+3)+c(i,k+2,l+3)
            sum = sum+a(i,j+3,k+2)+b(i,j+3,l+3)+c(i,k+2,l+3)
            sum = sum+a(i,j,k+3)+b(i,j,l+3)+c(i,k+3,l+3)
            sum = sum+a(i,j+1,k+3)+b(i,j+1,l+3)+c(i,k+3,l+3)
            sum = sum+a(i,j+2,k+3)+b(i,j+2,l+3)+c(i,k+3,l+3)
            sum = sum+a(i,j+3,k+3)+b(i,j+3,l+3)+c(i,k+3,l+3)
         end do
      end do
      do j = j, n
         do i = 1, n
            sum = sum+a(i,j,k)+b(i,j,l)+c(i,k,l)
            sum = sum+a(i,j,k+1)+b(i,j,l)+c(i,k+1,l)
            sum = sum+a(i,j,k+2)+b(i,j,l)+c(i,k+2,l)
            sum = sum+a(i,j,k+3)+b(i,j,l)+c(i,k+3,l)
         end do
      end do
   end do
   do k = k, n
      do j = 1, n-3, 4
         do i = 1, n
            sum = sum+a(i,j,k)+b(i,j,l)+c(i,k,l)
            sum = sum+a(i,j+1,k)+b(i,j+1,l)+c(i,k,l)
            sum = sum+a(i,j+2,k)+b(i,j+2,l)+c(i,k,l)
            sum = sum+a(i,j+3,k)+b(i,j+3,l)+c(i,k,l)
         end do
      end do
      do j = j, n
         do i = 1, n
            sum = sum+a(i,j,k)+b(i,j,l)+c(i,k,l)
         end do
      end do
   end do
   do k = 1, n-3, 4
      do j = j, n
         do i = 1, n
            sum = sum+a(i,j,k)+b(i,j,l+1)+c(i,k,l+1)
            sum = sum+a(i,j,k+1)+b(i,j,l+1)+c(i,k+1,l+1)
            sum = sum+a(i,j,k+2)+b(i,j,l+1)+c(i,k+2,l+1)
            sum = sum+a(i,j,k+3)+b(i,j,l+1)+c(i,k+3,l+1)
         end do
      end do
   end do
   do k = k, n
      do j = 1, n-3, 4
         do i = 1, n
            sum = sum+a(i,j,k)+b(i,j,l+1)+c(i,k,l+1)
            sum = sum+a(i,j+1,k)+b(i,j+1,l+1)+c(i,k,l+1)
            sum = sum+a(i,j+2,k)+b(i,j+2,l+1)+c(i,k,l+1)
            sum = sum+a(i,j+3,k)+b(i,j+3,l+1)+c(i,k,l+1)
         end do
      end do
      do j = j, n
         do i = 1, n
            sum = sum+a(i,j,k)+b(i,j,l+1)+c(i,k,l+1)
         end do
      end do
   end do
```

Fig. 11. Generated code using unroll-and-jam transformation (Part 2 of 4).

```
      do k = 1, n-3, 4
         do j = j, n
            do i = 1, n
               sum = sum+a(i,j,k)+b(i,j,l+2)+c(i,k,l+2)
               sum = sum+a(i,j,k+1)+b(i,j,l+2)+c(i,k+1,l+2)
               sum = sum+a(i,j,k+2)+b(i,j,l+2)+c(i,k+2,l+2)
               sum = sum+a(i,j,k+3)+b(i,j,l+2)+c(i,k+3,l+2)
            end do
         end do
      end do
      do k = k, n
         do j = 1, n-3, 4
            do i = 1, n
               sum = sum+a(i,j,k)+b(i,j,l+2)+c(i,k,l+2)
               sum = sum+a(i,j+1,k)+b(i,j+1,l+2)+c(i,k,l+2)
               sum = sum+a(i,j+2,k)+b(i,j+2,l+2)+c(i,k,l+2)
               sum = sum+a(i,j+3,k)+b(i,j+3,l+2)+c(i,k,l+2)
            end do
         end do
         do j = j, n
            do i = 1, n
               sum = sum+a(i,j,k)+b(i,j,l+2)+c(i,k,l+2)
            end do
         end do
      end do
      do k = 1, n-3, 4
         do j = j, n
            do i = 1, n
               sum = sum+a(i,j,k)+b(i,j,l+3)+c(i,k,l+3)
               sum = sum+a(i,j,k+1)+b(i,j,l+3)+c(i,k+1,l+3)
               sum = sum+a(i,j,k+2)+b(i,j,l+3)+c(i,k+2,l+3)
               sum = sum+a(i,j,k+3)+b(i,j,l+3)+c(i,k+3,l+3)
            end do
         end do
      end do
      do k = k, n
         do j = 1, n-3, 4
            do i = 1, n
               sum = sum+a(i,j,k)+b(i,j,l+3)+c(i,k,l+3)
               sum = sum+a(i,j+1,k)+b(i,j+1,l+3)+c(i,k,l+3)
               sum = sum+a(i,j+2,k)+b(i,j+2,l+3)+c(i,k,l+3)
               sum = sum+a(i,j+3,k)+b(i,j+3,l+3)+c(i,k,l+3)
            end do
         end do
         do j = j, n
            do i = 1, n
               sum = sum+a(i,j,k)+b(i,j,l+3)+c(i,k,l+3)
            end do
         end do
      end do
   end do
   do l = l, n
      do k = 1, n-3, 4
         do j = 1, n-3, 4
            do i = 1, n
               sum = sum+a(i,j,k)+b(i,j,l)+c(i,k,l)
               sum = sum+a(i,j+1,k)+b(i,j+1,l)+c(i,k,l)
               sum = sum+a(i,j+2,k)+b(i,j+2,l)+c(i,k,l)
               sum = sum+a(i,j+3,k)+b(i,j+3,l)+c(i,k,l)
```

Fig. 12. Generated code using unroll-and-jam transformation (Part 3 of 4).

```
            sum = sum+a(i,j,k+1)+b(i,j,l)+c(i,k+1,l)
            sum = sum+a(i,j+1,k+1)+b(i,j+1,l)+c(i,k+1,l)
            sum = sum+a(i,j+2,k+1)+b(i,j+2,l)+c(i,k+1,l)
            sum = sum+a(i,j+3,k+1)+b(i,j+3,l)+c(i,k+1,l)
            sum = sum+a(i,j,k+2)+b(i,j,l)+c(i,k+2,l)
            sum = sum+a(i,j+1,k+2)+b(i,j+1,l)+c(i,k+2,l)
            sum = sum+a(i,j+2,k+2)+b(i,j+2,l)+c(i,k+2,l)
            sum = sum+a(i,j+3,k+2)+b(i,j+3,l)+c(i,k+2,l)
            sum = sum+a(i,j,k+3)+b(i,j,l)+c(i,k+3,l)
            sum = sum+a(i,j+1,k+3)+b(i,j+1,l)+c(i,k+3,l)
            sum = sum+a(i,j+2,k+3)+b(i,j+2,l)+c(i,k+3,l)
            sum = sum+a(i,j+3,k+3)+b(i,j+3,l)+c(i,k+3,l)
         end do
      end do
      do j = j, n
         do i = 1, n
            sum = sum+a(i,j,k)+b(i,j,l)+c(i,k,l)
            sum = sum+a(i,j,k+1)+b(i,j,l)+c(i,k+1,l)
            sum = sum+a(i,j,k+2)+b(i,j,l)+c(i,k+2,l)
            sum = sum+a(i,j,k+3)+b(i,j,l)+c(i,k+3,l)
         end do
      end do
   end do
   do k = k, n
      do j = 1, n-3, 4
         do i = 1, n
            sum = sum+a(i,j,k)+b(i,j,l)+c(i,k,l)
            sum = sum+a(i,j+1,k)+b(i,j+1,l)+c(i,k,l)
            sum = sum+a(i,j+2,k)+b(i,j+2,l)+c(i,k,l)
            sum = sum+a(i,j+3,k)+b(i,j+3,l)+c(i,k,l)
         end do
      end do
      do j = j, n
         do i = 1, n
            sum = sum+a(i,j,k)+b(i,j,l)+c(i,k,l)
         end do
      end do
   end do
end do
```

Fig. 13. Generated code using unroll-and-jam transformation (Part 4 of 4).

```
do l=1,n-3,4
  do k=1,n-3,4
    do j=1,n-3,4
       do i=1,n,1
        sum = sum+a(i,j,k)+b(i,j,l)+c(i,k,l)
        sum = sum+a(i,j,k)+b(i,j,l+1)+c(i,k,l+1)
        sum = sum+a(i,j,k)+b(i,j,l+2)+c(i,k,l+2)
        sum = sum+a(i,j,k)+b(i,j,l+3)+c(i,k,l+3)
        sum = sum+a(i,j,k+1)+b(i,j,l)+c(i,k+1,l)
        sum = sum+a(i,j,k+1)+b(i,j,l+1)+c(i,k+1,l+1)
        sum = sum+a(i,j,k+1)+b(i,j,l+2)+c(i,k+1,l+2)
        sum = sum+a(i,j,k+1)+b(i,j,l+3)+c(i,k+1,l+3)
        sum = sum+a(i,j,k+2)+b(i,j,l)+c(i,k+2,l)
        sum = sum+a(i,j,k+2)+b(i,j,l+1)+c(i,k+2,l+1)
        sum = sum+a(i,j,k+2)+b(i,j,l+2)+c(i,k+2,l+2)
        sum = sum+a(i,j,k+2)+b(i,j,l+3)+c(i,k+2,l+3)
        sum = sum+a(i,j,k+3)+b(i,j,l)+c(i,k+3,l)
        sum = sum+a(i,j,k+3)+b(i,j,l+1)+c(i,k+3,l+1)
        sum = sum+a(i,j,k+3)+b(i,j,l+2)+c(i,k+3,l+2)
        sum = sum+a(i,j,k+3)+b(i,j,l+3)+c(i,k+3,l+3)
        sum = sum+a(i,j+1,k)+b(i,j+1,l)+c(i,k,l)
        sum = sum+a(i,j+1,k)+b(i,j+1,l+1)+c(i,k,l+1)
        sum = sum+a(i,j+1,k)+b(i,j+1,l+2)+c(i,k,l+2)
        sum = sum+a(i,j+1,k)+b(i,j+1,l+3)+c(i,k,l+3)
        sum = sum+a(i,j+1,k+1)+b(i,j+1,l)+c(i,k+1,l)
        sum = sum+a(i,j+1,k+1)+b(i,j+1,l+1)+c(i,k+1,l+1)
        sum = sum+a(i,j+1,k+1)+b(i,j+1,l+2)+c(i,k+1,l+2)
        sum = sum+a(i,j+1,k+1)+b(i,j+1,l+3)+c(i,k+1,l+3)
        sum = sum+a(i,j+1,k+2)+b(i,j+1,l)+c(i,k+2,l)
        sum = sum+a(i,j+1,k+2)+b(i,j+1,l+1)+c(i,k+2,l+1)
        sum = sum+a(i,j+1,k+2)+b(i,j+1,l+2)+c(i,k+2,l+2)
        sum = sum+a(i,j+1,k+2)+b(i,j+1,l+3)+c(i,k+2,l+3)
        sum = sum+a(i,j+1,k+3)+b(i,j+1,l)+c(i,k+3,l)
        sum = sum+a(i,j+1,k+3)+b(i,j+1,l+1)+c(i,k+3,l+1)
        sum = sum+a(i,j+1,k+3)+b(i,j+1,l+2)+c(i,k+3,l+2)
        sum = sum+a(i,j+1,k+3)+b(i,j+1,l+3)+c(i,k+3,l+3)
        sum = sum+a(i,j+2,k)+b(i,j+2,l)+c(i,k,l)
        sum = sum+a(i,j+2,k)+b(i,j+2,l+1)+c(i,k,l+1)
        sum = sum+a(i,j+2,k)+b(i,j+2,l+2)+c(i,k,l+2)
        sum = sum+a(i,j+2,k)+b(i,j+2,l+3)+c(i,k,l+3)
        sum = sum+a(i,j+2,k+1)+b(i,j+2,l)+c(i,k+1,l)
        sum = sum+a(i,j+2,k+1)+b(i,j+2,l+1)+c(i,k+1,l+1)
        sum = sum+a(i,j+2,k+1)+b(i,j+2,l+2)+c(i,k+1,l+2)
        sum = sum+a(i,j+2,k+1)+b(i,j+2,l+3)+c(i,k+1,l+3)
        sum = sum+a(i,j+2,k+2)+b(i,j+2,l)+c(i,k+2,l)
        sum = sum+a(i,j+2,k+2)+b(i,j+2,l+1)+c(i,k+2,l+1)
        sum = sum+a(i,j+2,k+2)+b(i,j+2,l+2)+c(i,k+2,l+2)
        sum = sum+a(i,j+2,k+2)+b(i,j+2,l+3)+c(i,k+2,l+3)
        sum = sum+a(i,j+2,k+3)+b(i,j+2,l)+c(i,k+3,l)
        sum = sum+a(i,j+2,k+3)+b(i,j+2,l+1)+c(i,k+3,l+1)
        sum = sum+a(i,j+2,k+3)+b(i,j+2,l+2)+c(i,k+3,l+2)
        sum = sum+a(i,j+2,k+3)+b(i,j+2,l+3)+c(i,k+3,l+3)
        sum = sum+a(i,j+3,k)+b(i,j+3,l)+c(i,k,l)
        sum = sum+a(i,j+3,k)+b(i,j+3,l+1)+c(i,k,l+1)
        sum = sum+a(i,j+3,k)+b(i,j+3,l+2)+c(i,k,l+2)
        sum = sum+a(i,j+3,k)+b(i,j+3,l+3)+c(i,k,l+3)
        sum = sum+a(i,j+3,k+1)+b(i,j+3,l)+c(i,k+1,l)
        sum = sum+a(i,j+3,k+1)+b(i,j+3,l+1)+c(i,k+1,l+1)
        sum = sum+a(i,j+3,k+1)+b(i,j+3,l+2)+c(i,k+1,l+2)
        sum = sum+a(i,j+3,k+1)+b(i,j+3,l+3)+c(i,k+1,l+3)
        sum = sum+a(i,j+3,k+2)+b(i,j+3,l)+c(i,k+2,l)
        sum = sum+a(i,j+3,k+2)+b(i,j+3,l+1)+c(i,k+2,l+1)
```

Fig. 14. Generated code using compact code generation (Part 1 of 2).

```
          sum = sum+a(i,j+3,k+2)+b(i,j+3,l+2)+c(i,k+2,l+2)
          sum = sum+a(i,j+3,k+2)+b(i,j+3,l+3)+c(i,k+2,l+3)
          sum = sum+a(i,j+3,k+3)+b(i,j+3,l)+c(i,k+3,l)
          sum = sum+a(i,j+3,k+3)+b(i,j+3,l+1)+c(i,k+3,l+1)
          sum = sum+a(i,j+3,k+3)+b(i,j+3,l+2)+c(i,k+3,l+2)
          sum = sum+a(i,j+3,k+3)+b(i,j+3,l+3)+c(i,k+3,l+3)
        end do
      end do
      do j=j,n,1
         do i=1,n,1
          sum = sum+a(i,j,k)+b(i,j,l)+c(i,k,l)
          sum = sum+a(i,j,k)+b(i,j,l+1)+c(i,k,l+1)
          sum = sum+a(i,j,k)+b(i,j,l+2)+c(i,k,l+2)
          sum = sum+a(i,j,k)+b(i,j,l+3)+c(i,k,l+3)
          sum = sum+a(i,j,k+1)+b(i,j,l)+c(i,k+1,l)
          sum = sum+a(i,j,k+1)+b(i,j,l+1)+c(i,k+1,l+1)
          sum = sum+a(i,j,k+1)+b(i,j,l+2)+c(i,k+1,l+2)
          sum = sum+a(i,j,k+1)+b(i,j,l+3)+c(i,k+1,l+3)
          sum = sum+a(i,j,k+2)+b(i,j,l)+c(i,k+2,l)
          sum = sum+a(i,j,k+2)+b(i,j,l+1)+c(i,k+2,l+1)
          sum = sum+a(i,j,k+2)+b(i,j,l+2)+c(i,k+2,l+2)
          sum = sum+a(i,j,k+2)+b(i,j,l+3)+c(i,k+2,l+3)
          sum = sum+a(i,j,k+3)+b(i,j,l)+c(i,k+3,l)
          sum = sum+a(i,j,k+3)+b(i,j,l+1)+c(i,k+3,l+1)
          sum = sum+a(i,j,k+3)+b(i,j,l+2)+c(i,k+3,l+2)
          sum = sum+a(i,j,k+3)+b(i,j,l+3)+c(i,k+3,l+3)
        end do
      end do
    end do

    do k=k,n,1
       do j=1,n,1
        do i=1,n,1
          sum = sum+a(i,j,k)+b(i,j,l)+c(i,k,l)
          sum = sum+a(i,j,k)+b(i,j,l+1)+c(i,k,l+1)
          sum = sum+a(i,j,k)+b(i,j,l+2)+c(i,k,l+2)
          sum = sum+a(i,j,k)+b(i,j,l+3)+c(i,k,l+3)
        end do
      end do
    end do
  end do

  do l=l,n,1
    do k=1,n,1
      do j=1,n,1
        do i=1,n,1
          sum = sum+a(i,j,k)+b(i,j,l)+c(i,k,l)
        end do
      end do
    end do
  end do
```

Fig. 15. Generated code using compact code generation (Part 2 of 2).

```
do l = 1, n
  do k = 1, n
    do j = 1, n
      do i = 1, n
        sum = sum + a(i,j,k) + b(i,j,l) + c(i,k,l)
      end do
    end do
  end do
end do
```

The transformed code generated for this example obtained by using the unroll-and-jam approach is shown in Figs. 10–13. Figures 14 and 15 show the transformed code obtained by using the code generation algorithm presented in this paper. Both approaches generated an unrolled loop body containing $4 \times 4 \times 4 = 64$ copies of the original loop body. However, our algorithm generated $4 \times 4 + 4 + 1 = 21$ remainder loops for this example as opposed to $5 \times 5 \times 5 - 64 = 61$ remainder loops generated by the unroll-and-jam approach. The number of remainder loops generated by the unroll-and-jam approach can potentially be reduced by first "rerolling" all unrolled remainder loops and then performing an "index set merging" transformation on remainder loops (i.e., the inverse of the "index set splitting" transformation[16]). However, we are not aware of any compiler that performs loop rerolling and index set merging of loops after applying an unroll-and-jam transformation.

REFERENCES

1. F. E. Allen and J. Cocke, A catalogue of optimizing transformations, in *Design and Optimization of Compilers*, Prentice-Hall, pp. 1–30 (1972).
2. J. J. Dongarra and A. R. Hinds, Unrolling Loops in Fortran, *Software—Practice and Experience* **9**(3):219–226 (March 1979).
3. J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau, Parallel Processing: A Smart Compiler and a Dumb Machine, *Proc. ACM Symp. Compiler Construction*, pp. 37–47 (June 1984).
4. D. F. Bacon, S. L. Graham, and O. J. Sharp, Compiler Transformations for High-Performance Computing, *ACM Computing Surveys* **26**(4):345–420 (December 1994).
5. Steve Carr and Ken Kennedy, Scalar Replacement in the Presence of Conditional Control Flow, *Software—Practice and Experience* (1):51–77 (January 1994).
6. Michael J. Alexander, Mark W. Bailey, Bruce R. Childers, Jack W. Davidson, and Sanjay Jinturkar, Memory bandwidth optimizations for wide-bus machines, *Proc. 26th Hawaii Int'l. Conf. Syst. Sci.*, Wailea, Hawaii, pp. 466–475 (January 1993).
7. T. C. Mowry, *Tolerating Latency Through Software-Controlled Data Prefetching*, Ph.D. thesis, Stanford University (March 1994).
8. Mauricio Breternitz, Michael Lai, Vivek Sarkar, and Barbara Simons, Compiler Solutions for the Stale-Data and False-Sharing Problems, Technical report, TR 03.466, IBM Santa Teresa Laboratory (April 1993).

9. Steve Carr and Ken Kennedy, Improving the Ratio of Memory Operations to Floating-Point Operations in Loops, *ACM TOPLAS* **16**(4) (November 1994).
10. Jack W. Davidson and Sanjay Jinturkar, Aggressive Loop Unrolling in a Retargetable, Optimizing Compiler, In *Compiler Construction, Proc. Sixth Int'l. Conf. Linkoping, Sweden*, Vol. 1060, *Lecture Notes in Computer Science*, Springer-Verlag, New York (April 1996).
11. David Callahan, Steve Carr, and Ken Kennedy, Improving Register Allocation for Subscripted Variables, *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implementation*, White Plains, New York, pp. 53–65 (June 1990).
12. S. Carr and Y. Guan, Unroll-and-Jam Using Uniformly Generated Sets, *Proc. MICRO-30*, pp. 349–357 (December 1997).
13. Allan K. Porterfield, Software Methods for Improvement of Cache Performance on Supercomputer Applications, Ph.D. thesis, Rice University, Rice COMP TR89-93 (May 1989).
14. Michael E. Wolf and Monica S. Lam, A Data Locality Optimization Algorithm, *Proc. ACM SIGPLAN Symp. Progr. Lang. Design and Implementation*, pp. 30–44 (June 1991).
15. Vivek Sarkar, Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers. *IBM J. Res. Dev.* **41**(3) (May 1997).
16. Michael J. Wolfe, *Optimizing Supercompilers for Supercomputers*, Pitman, London and The MIT Press, Cambridge, Massachusetts (1989). In the series, Research Monographs in Parallel and Distributed Computing.
17. Vivek Sarkar and Radhika Thekkath, A General Framework for Iteration-Reordering Loop Transformations, *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implementation*, pp. 175–187 (June 1992).
18. Jeanne Ferrante, Vivek Sarkar, and Wendy Thrash, On Estimating and Enhancing Cache Effectiveness, *Lecture Notes in Computer Science* (589):328–343 (1991). *Proc. Fourth Int'l. Workshop Lang. Compilers for Parallel Computing*, Santa Clara, California (August 1991).
19. B. Ramakrishna Rau, Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops, *Proc. 27th Ann. Int'l. Symp. Microarchitecture*, San Jose, California, pp. 63–74 (November 1994).
20. Vivek Sarkar and Barbara Simons, Don't Waste Those Cycles: An In-Depth Look at Scheduling Instructions in Basic Blocks and Loops, Video Lecture in University Video Communication's Distinguished Lecture Series IX (August 1994).
21. Vivek Sarkar, Determining Average Program Execution Times and their Variance, *Proc. SIGPLAN Conf. Prog. Lang. Design and Implementation* **24**(7):298–312 (July 1989).
22. Vivek Sarkar, Automatic Partitioning of a Program Dependence Graph into Parallel Tasks, *IBM J. Res. Dev* **35**(5/6) (1991).
23. The Standard Performance Evaluation Corporation, SPEC CPU95 Benchmarks, http://open.specbench.org/osg/cpu95/ (1997).
24. IBM Corporation, POWER2 and PowerPC, Special issue of *IBM J. Res. Dev.* **38**(5): 489–648 (September 1994).
25. Barbara Simons, Vivek Sarkar, Jr. Mauricio Breternitz, and Michael Lai, An Optimal Asynchronous Scheduling Algorithm for Software Cache Consistency, *Proc. Hawaii Int'l. Conf. Syst. Sci.* (January 1994).
26. Max Hailperin, Improving the Ratio of Memory Operations to Floating-Point operations in loops, *Computing Reviews*. Copy of review can be found in the ACM digital library at http://www.acm.org/pubs/citations/journals/toplas/1994-16-6/p1768-carr/.
27. S. Weiss and J. E. Smith, A Study of Scalar Compilation Techniques for Pipelined Supercomputers, *Proc. Second Int'l Conf. Architectural Support Progr. Lang. Oper. Syst. (ASPLOS)*, pp. 105–109 (October 1987).

28. Reese B. Jones and Vicki H. Allan, Software Pipelining: An Evaluation of Enhanced Pipelining, *Proc. 24th Ann. Int'l. symp. Microarchitecture*, pp. 82–92 (December 1990).
29. Bogong Su, Shiyuan Ding, Jian Wang, and Jinshi Xia, GURPR—A Method for Global Software Piplining; *Proc. 20th Ann. Int'l. Symp. Microarchitecture*, pp. 88–96 (December 1986).
30. Daniel M. Lavery and Wen-Mei W.Hwu, Unrolling-Based Optmizations for Modulo Sheduling, *Proc. MICRO-28*, pp. 327–337 (December 1995).

Printed in Belgium