Enabling Loop Fusion and Tiling for Cache Performance by Fixing Fusion-Preventing Data Dependences

Jingling Xue[†]and Qingguang Huang[†]
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia

Minyi Guo
School of Computer Science and Engineering
The University of Aizu
Fukushima 965-8580, Japan

Abstract

This paper presents a new approach to enabling loop fusion and tiling for arbitrary affine loop nests. Given a set of multiple loop nests, we present techniques that automatically eliminate all the fusion-preventing dependences by means of loop tiling and array copying. Applying our techniques iteratively to multiple loop nests yields a single loop nest that can be tiled for cache locality. Our approach handles LU, QR, Cholesky and Jacobi in a unified framework. Our experimental evaluation on an SGI Octane2 system shows that the benefit from the significantly reduced L1 and L2 cache misses has far more than offset the branching and loop control overhead introduced by our approach.

1 Introduction

Due to the ever-widening performance gap between processors and memories, loop tiling or blocking [6, 16] remains an important optimisation in improving cache performance. However, loop tiling is applicable to perfect loop nests only. Unfortunately, dense matrix kernels often contain multiple, imperfectly nested loops over the same data. Figure 1 shows the four important kernels, LU with partial pivoting, QR, Cholesky and Jacobi, which are among the frequently used in scientific applications.

To enable an imperfect loop nest to be tiled, the compiler may apply transformations such as loop fusion and loop distribution to turn parts of the nest into some perfect loop nests [15] and then apply loop tiling to tile the individual perfect loop nests. However, these transformations are mostly dependence-preserving and are thus frequently inapplicable since the data dependences in the program would otherwise be violated. Several recent attempts aim at providing a more systematic treatment to the problem of tiling imperfect loop nests. They have tackled the problem from different angles with varying degrees of success, resulting in the special-purpose techniques for array factorisations [2] and stencil computations [12] as well as the general-purpose techniques for arbitrary loop nests based on data shackling [8] and iteration space transformations [1].

This paper presents a new general-purpose approach to tiling imperfect loop nests from yet a different perspective. Our key motivation can be stated as follows. Many dense kernel programs include multiple loops iterating over the same data. Often the most computations in such a kernel are carried out in a set of loops, which cannot be tiled directly since they are imperfectly nested. In Figure 1, each kernel contains one such a nest as highlighted by $*$'s. But fusing the loops where the most computations are performed with the others and then tiling the fused loops will greatly improve cache performance. Unfortunately, the fused program is generally incorrect due to the existence of some fusion-preventing dependences. We observe that in *real* programs, the "trouble spots" that prevent loop fusion can be relatively few. We are motivated to fix these fusion-preventing dependences so that the fixed program is correct. We eliminate the fusion-preventing flow and output dependences by applying loop tiling (which subsumes loop unrolling and unroll-and-jam [17]) to the appropriate loop nests in the fused program. We eliminate all the fusion-preventing anti-dependences by inserting array copy operations wherever appropriate.

Our approach creates loop tiling candidate nests aggressively. As a result, more conditionals (due to code sinking) and loop control overhead (due to tiling) may be introduced into the transformed codes. Our experimental evaluation on an SGI Octane2 system shows that our approach generates simple tiled codes and achieves performance speedups across varying problem sizes on an SGI Octane2 computer system. The benefit from the significantly reduced L1 and L2 cache misses has far more than offset the branching and loop control overhead incurred.

The rest of this paper is organised as follows. Section 2 describes the class of programs considered by this work. Section 3 presents our methodology for tiling imperfect loop nests. In particular, Section 3.1 presents an algorithm for fusing multiple perfect loop nests. Section 3.2 discusses how to apply this algorithm iteratively to a program to yield larger perfect loop nests. Section 4 presents and analyses our experimental results for the four kernels shown in Figure 1. Section 5 compares with the related work. Section 6 concludes by discussing some future work.

[†]This work is supported by an ARC Grant DP0452623.

0190-3918/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

```
*do k=1, N
    temp=0
    m=k
    do i=k, N
      d= A(i,k)
      if (abs(d).GT.temp)
        temp=abs(d)
        m=i
    if (m.NE.k)
      do j=k, N
        temp=A(k,j)
        A(k,j)=A(m,j)
        A(m,j)=temp
    do i=k+1, N
      A(i,k)=A(i,k)/A(k,k)
*   do j=k+1, N
*     do i=k+1, N
        A(i,j)=A(i,j)-A(i,k)*A(k,j)
```

(a) LU (with pivoting)

```
*do i=1, N
    norm=0
    do j=i, N
      norm=norm+A(j,i)*A(j,i)
    norm2=sqrt (norm)
    asqr=A(i,i)*A(i,i)
    A(i,i)=dsqrt(norm-asqr+(A(i,i)-norm2)^2)
    do j=i+1, N
      A(j,i)=A(j,i)]/A(i,i)
    do j=i+1, N
       X(j,i)=0
       do k=i, N
         X(j,i)=X(j,i)+A(k,i)*A(k,j)
*   do j=i+1, N
*     do k=i+1, N
        A(k,j) = A(k,j)-A(k,i)*X(j,i)
```

(b) QR

```
*do k=1, N
    A(k,k)=dsqrt(A(k,k))
    do i=k+1, N
      A(i,k)=A(i,k)/(A(k,k)
*   do j=k+1, N
*     do i=j, N
        A(i,j)=A(i,j)-A(i,k)*A(j,k)
```

(c) Cholesky

```
*do t=0, M
*   do i=2, N-1
*     do j=2,N-1
        L(j,i)=(A(j,i-1)+A(j-1,i)
            + A(j+1,i)+A(j,i+1)*0.25
    do i=2, N-1
      do j=2, N-1
        A(j,i)=L(j,i)
```

(d) Jacobi

Figure 1. Four frequently used kernels in dense matrix computations.

2 Program Model

Our approach is applicable to all affine loop nests. In general, all loop bounds are required to be affine since the loops with non-affine bounds are difficult to fuse. However, some non-affine if-conditional or array subscript expressions may be allowed (as long as our algorithm given in Figure 2 can successfully compute the required dependences and eliminate all fusion-preventing dependences). For example, we can handle the LU program given in Figure 1, which contains a data-dependent test in line 6.

All programs are given in the FORTRAN-like syntax. In Figure 1, Jacobi is the classic program for solving PDEs by explicit method. The other three kernels are taken from [7] except that the scalar `m` in LU is not array-expanded here. Note that QR given here has been simplified in [7] with some inessential statements removed.

3 Approach

Given a program in the form of an imperfect loop nest, our approach applies loop fusion inside out to the program to create larger and larger perfect loop nests, and eventually one single perfect loop nest if desired. During this process, all the fusion-preventing dependences are eliminated automatically by means of loop tiling and array copying. The resulting perfect loop nests can be tiled in the normal manner. Section 3.1 gives an algorithm for fusing multiple perfect loop nests, which is the key to our approach. Section 3.2 describes how to apply this algorithm to a program iteratively to obtain increasingly larger perfect loop nests.

3.1 Fusing Multiple Perfect Loop Nests

The problem of fusing multiple perfect loop nests is solved in more or less the same way as the special case when two perfect loop nests are to be fused.

The fusion of two perfect loop nests is legal iff all dependences from the first (i.e., the lexically earlier) nest to the second nest are not reversed in the fused program [17, p. 315]. The dependences that are reversed are known as the *fusion-preventing dependences*. There are three kinds of fusion-preventing dependences: flow (i.e., write before read) dependences, output (i.e., write before write dependences) and anti- (i.e., read before write) dependences.

Suppose we are given two perfect loop nests that are to be fused in a certain way. The two nests may not have the same loop bounds in a common dimension or even the same number of loops. The violated dependences are fixed as follows. We first compute the fusion-preventing dependences between the two nests. We then eliminate the fusion-preventing flow and output dependences automatically by applying loop tiling to the first loop nest. Finally, we eliminate the fusion-preventing anti-dependences by inserting array copy operations inside the second loop nest. The resulting program is a perfect loop nest and can thus be tiled for locality improvement.

In the case of more than two nests, our algorithm first applies loop tiling bottom-up across all the loop nests to eliminate the fusion-preventing flow and output dependences. We then insert array copy operations to eliminate all the fusion-preventing anti-dependences.

Suppose there are K perfect loop nests:

$$
\begin{array}{ll}
\mathcal{L}_1: & \text{do } I_1 = L_{1,1}, U_{1,1} \\
& \quad \vdots \\
& \quad \text{do } I_{n_1} = L_{1,n_1}, U_{1,n_1} \\
& \quad\quad \mathit{BODY}_1(I_1, \ldots, I_{n_1}) \\
\vdots & \\
\mathcal{L}_K: & \text{do } I_K = L_{K,1}, U_{K,1} \\
& \quad \vdots \\
& \quad \text{do } I_{n_K} = L_{K,n_K}, U_{K,n_K} \\
& \quad\quad \mathit{BODY}_K(I_1, \ldots, I_{n_K})
\end{array} \tag{1}
$$

where the loop bounds of each loop nest are assumed to be affine. Two different loop nests may not have the same loop bounds in a common dimension or even the same number of loops.

Let IS_k be the n_k-dimensional iteration space of the k-th loop nest $\mathcal{L}_k$. Let $n = \max\{n_k \mid 1 \leqslant k \leqslant K\}$. If the dependences in the program (1) are ignored for the moment, it is always possible to fuse the K nests into one perfect loop nest whose n-dimensional iteration space is:

$$\mathit{IS} = \{(I_1, \ldots, I_n) \mid \forall\, 1 \leqslant i \leqslant n : L_i \leqslant I_i \leqslant U_i\} \tag{2}$$

The fusion transformation consists of finding an injective mapping from IS_k to IS for every loop nest $\mathcal{L}_k$:

$$F_k : \mathit{IS}_k \mapsto \mathit{IS} \tag{3}$$

The fused program becomes one single perfect loop nest:

$$
\begin{array}{l}
\text{do } I_1 = L_1, U_1 \\
\quad \vdots \\
\quad \text{do } I_n = L_n, U_n \\
\quad\quad \text{if } (I_1, \ldots, I_n) \in F_1(\mathit{IS}_k) \\
\quad\quad\quad \mathit{BODY}_1(F_1^{-1}(I_1, \ldots, I_n)) \\
\quad\quad\quad\quad \vdots \\
\quad\quad \text{if } (I_1, \ldots, I_n) \in F_K(\mathit{IS}_K) \\
\quad\quad\quad \mathit{BODY}_K(F_K^{-1}(I_1, \ldots, I_n))
\end{array} \tag{4}
$$

For many real dense matrix programs, IS is typically the same or slightly larger than the iteration space of the loop nest that carries out the most computations in a program (e.g., the loop nests highlighted by $*$'s in Figure 1). The iteration space of $\mathcal{L}_k$ such that $k < n$ is often embedded at a boundary of IS. Its exact placement may not be critical to our approach for two reasons. First, fusing a loop nest that carries out the most computations with the other loop nests enables this "important" loop nest to be tiled so that the overall cache performance of the program is improved. Second, whichever placement is used, our approach can always turn the fused loop nest into a correct program.

The loop fusion used for obtaining the fused program (4) may be illegal. Figure 2 gives an algorithm for fixing all the fusion-preventing dependences so that the fixed program has the same input/output behaviour as the original program (1).

The following notations (with some from [20]) are used:

$I, I', I'', \ldots$	an iteration vector in IS
$\prec$	lexicographic "less than" operator
$[R]$	set of iteration vectors at which reference R is accessed
$R(I)$	subscript expression when the loop variables have the values specified by I
$R_1(I) \overset{sub}{=} R_2(I')$	subscripts of $R_1(I)$ and $R_2(I')$ are equal
$\mathit{Writes}_A(k)$	all write references of A in $\mathcal{L}_k$
$\mathit{Reads}_A(k)$	all read references of A in $\mathcal{L}_k$

Let A be an arbitrary but fixed array in the program (1). When we transform (1) to (4) by loop fusion, the fusion-preventing dependences, i.e., the dependences that are violated are characterised precisely by the following sets:

- $WW_A(k, k')$ gives the output dependences of A that prevent $\mathcal{L}_k$ and $\mathcal{L}_{k'}$ from being fused, where $k < k'$:

$$
\begin{aligned}
WW_A(k, k') = \{(I, I') \mid\; & R \in \mathit{Writes}_A(k) \wedge I \in [R] \\
& \wedge R' \in \mathit{Writes}_A(k') \wedge I' \in [R'] \\
& \wedge I' \prec I \wedge R(I) \overset{sub}{=} R'(I')\}
\end{aligned} \tag{5}
$$

- $WR_A(k, k')$ gives the flow dependences of A that prevent $\mathcal{L}_k$ and $\mathcal{L}_{k'}$ from being fused, where $k < k'$. This set is defined exactly as $WW_A(k, k')$ except that $\mathit{Writes}_A(k')$ in $WW_A(k, k')$ is replaced by $\mathit{Reads}_A(k')$.

- $RW_A(k, k')$ gives the anti-dependences of A that prevent $\mathcal{L}_k$ and $\mathcal{L}_{k'}$ from being fused, where $k < k'$:

$$
\begin{aligned}
RW_A(k, k') = \{(I, I', \alpha(R')) \mid\; & R \in \mathit{Reads}_A(k) \wedge I \in [R] \\
& \wedge R' \in \mathit{Writes}_A(k') \wedge I' \in [R'] \\
& \wedge I' \prec I \wedge R(I) \overset{sub}{=} R'(I')\}
\end{aligned} \tag{6}
$$

  where $\alpha(R')$ indicates that the reference R' is the LHS of the $\alpha(R')$-th assignment in the loop nest $\mathcal{L}_{k'}$. This component is useful in imposing an execution order for different writes executed at the same iteration in $\mathcal{L}_{k'}$.

IEEE COMPUTER SOCIETY

1 **ALGORITHM:** *FixDeps*
2 **INPUT:** The fused program (4), denoted P
3 **OUTPUT:** A program, P'', with the same input/output behaviour as the original program (1)
4 P' = *ElimWW_WR*(P)
5 P'' = *ElimRW*(P')
6 Insert more coping operations to simplify the if conditionals introduced in lines 46 – 48

7 **ALGORITHM:** *ElimWW_WR*(P)
8 Let $\mathcal{V}$ be the set of all variables in P
9 $P_1 := P$
10 for $k = K-1, 1$
11 // compute the WW and WR dependences
12 for every array A in $\mathcal{V}$
13 for $k' = k+1, K$
14 Compute $WW_A(k,k')$ and $WR_A(k,k')$ in P_{K-k}
15 $WW_A(k) := \bigcup_{k'=k+1}^{K} WW_A(k,k')$
16 $WR_A(k) := \bigcup_{k'=k+1}^{K} WR_A(k,k')$
17 $W(k) := \bigcup_{A \in \mathcal{V}} (WW_A(k) \cup WR_A(k))$
18 if $W(k) = \emptyset$ then GOTO line 34
19 // compute the tile size to tile $\mathcal{L}_k$
20 $D_1 := W(k)$
21 for $i = 1, n$
22 $d_i = \max\{I_i - I'_i \mid (I, I') \in D_i\}$ // $\max \emptyset =_{def} 0$
23 $D_{i+1} := D_i \setminus \{(I, I') \mid \forall\, (I, I') \in D_i : I_i - I'_i > 0\}$
24 Let m be the largest value such that $d_{m+1} = \cdots = d_n = 0$
25 Let $(T_1, \ldots, T_m)$ be a legal tile size for the outermost loops of $\mathcal{L}_K$ such that $\forall\, 1 \leqslant i \leqslant m : T_i > d_i$
26 Set $T_{m+1} = \cdots = T_n = 1$
27 // generate the tile code for $\mathcal{L}_k$
28 Let $(O_1, \ldots, O_n)$ be the lexicographic minimum of *IS*
29 Let $\mathcal{T}$ be the tiling transformation, $\mathcal{T} : F_k(IS_k) \mapsto \mathbb{Z}^{2n}$
$\mathcal{T}(I_1, \ldots, I_n) = (O_1 + \lfloor \frac{I_1 - O_1}{T_1} \rfloor, \ldots, (O_n + \lfloor \frac{I_n - O_n}{T_n} \rfloor, I_1, \ldots, I_n)$
30 Let $(I_1, \ldots, I_n, J_1, \ldots, J_n)$ be the loop variables of tiled $\mathcal{L}_k$
31 Let $P_t(I_1, \ldots, I_n) \leqslant p_t$ be the inequalities defining the tiles
32 Let $P_e(I_1, \ldots, I_n, J_1, \ldots, J_n) \leqslant p_e$ specify the points in a tile
33 Replace the following code for $\mathcal{L}_k$ in (4)

if $(I_1, \ldots, I_n) \in F_k(IS_k)$
  $BODY_1(F_k^{-1}(I_1, \ldots, I_n))$

by:

if $P_t(I_1, \ldots, I_n) \leqslant p_t$
  The $J_1, \ldots, J_n$ loops for enumerating
    points in $P_e(I_1, \ldots, I_n, J_1, \ldots, J_n) \leqslant p_e$ in $\prec$
    if $(J_1, \ldots, J_n) \in F_k(IS_k)$
      $BODY_k(F_k^{-1}(J_1, \ldots, J_n))$

34 $P_{K-k+1} := P_{K-k}$ (tiled if not from line 18)
35 return P_K

Figure 2. An algorithm for fixing all the fusion-preventing data dependences.

Our algorithm *FixDeps* has two main procedures. The *ElimWW_WR* procedure applies loop tiling to the fused loop nests (4) so that all WW_A and WR_A sets are empty. The *ElimRW* procedure applies array copying so that all RW_A sets are empty. Both procedures are discussed below.

3.1.1 *ElimWW_WR*: Loop Tiling

The basic idea is to apply loop tiling iteratively bottom-up across the K loop nests starting from the second last loop nest. By tiling $\mathcal{L}_{K-1}$, we make sure that $\mathcal{L}_{K-1}$ and $\mathcal{L}_K$ can be fused without violating any flow and output dependences from $\mathcal{L}_{K-1}$ to $\mathcal{L}_K$. By tiling $\mathcal{L}_{K-2}$, we ensure that $\mathcal{L}_{K-1}$, $\mathcal{L}_{K-2}$ and $\mathcal{L}_K$ can be fused without violating any flow and output dependences from $\mathcal{L}_{K-2}$ to $\mathcal{L}_{K-1}$ and $\mathcal{L}_K$. This process is repeated until $\mathcal{L}_1$ is processed.

36 **ALGORITHM:** *ElimRW*(P)
37 Let $\mathcal{V}$ be the set of all variables in P
38 for every array A in $\mathcal{V}$
39 for $k = K-1, 1$
40 for $k' = k+1, K$
41 Compute $RW_A(k,k')$
42 $\overline{RW}_A(k) := \bigcup_{k'=k+1}^{K} \{(I', k', s') \mid (I, I', s') \in RW_A(k,k')\}$
43 Compute $\min_\prec \overline{RW}_A(k)$ (see Section 3.1.2)
44 Introduce a new copying array for A, $H_{A,k}$, whose size is specified by $|\min_\prec \overline{RW}_A(k)|$
45 Insert the following copy operations at the beginning of the loop body of $\mathcal{L}_{k+1}$

if $(I, k', s') \in \min_\prec \overline{RW}_A(k)$
  $H_{A,k}(f_{R'}(I)) = A(f_{R'}(I))$

where $A(f_{R'}(I))$ is the LHS of the s'-th assignment in $\mathcal{L}_{k'}$
46 for every read reference R of form $A(f_R(I))$ in $Reads_A(k)$
47 $C_R := I \in [R] \wedge\ k' > k \wedge R' \in Writes_A(k') \wedge I' \in [R']$
$\wedge\, I' \prec I \wedge R(I) \stackrel{sub}{=} R'(I')$
48 Replace $A(f_R(I))$ by:

if $I \in C_R$
  $H_{A,k}(f_R(I))$
else
  $A(f_R(I))$

Figure 2. An algorithm for fixing all the fusion-preventing data dependences (Cont'd).

As a loop invariant at the beginning of the loop in line 10, all the fusion-preventing flow and output dependences in the loop nests $\mathcal{L}_{k+1}, \ldots, \mathcal{L}_K$ of the fused program (4) have been eliminated. In lines 11 – 17, $W(k)$ is computed to be the set of all flow and output dependences from $\mathcal{L}_k$ to $\mathcal{L}_{k+1}, \ldots, \mathcal{L}_K$. These are the WW and WR dependences that are violated when $\mathcal{L}_k$ is fused with $\mathcal{L}_{k+1}, \ldots, \mathcal{L}_K$. In lines 19 – 24, we find the outermost m loops in $\mathcal{L}_k$ that carry all the dependences in $W(k)$. These are the loops to be tiled to eliminate all the violated dependences in $W(k)$. However, $(d_1+1, \ldots, d_m+1)$ computed in line 22 may not be a legal tile size if the dependences within $\mathcal{L}_k$ are also taken into account. Thus, in line 25, a legal tile size is found based also on the dependences within $\mathcal{L}_k$ [18, 19]. A legal tile size always exists. For example, $(N_1, \ldots, N_m)$ is always legal, where N_i is the maximum number of points in the i-th dimension of $F_k(IS_k)$. In lines 25 – 26, the tile size $(T_1, \ldots, T_n)$ for tiling $\mathcal{L}_k$ is selected. In lines 27 – 33, the tiled code for $\mathcal{L}_k$ is generated in the standard manner [18, 19]. Finally, in line 34, the tiled program becomes the program that will be used when $\mathcal{L}_{k-1}$ is tiled.

Theorem 1 *All the fusion-preventing flow and output dependences in the original program (1) are eliminated in the program P' generated by* ElimWW_WR*.*

3.1.2 *ElimRW*: Array Copying

The basic idea is to make use of array copying to eliminate all the fusion-preventing anti-dependences. The order in which the loop nests are processed in line 39 is not significant. In lines 40 – 41, we compute all the fusion-preventing anti-dependences from $\mathcal{L}_k$ to $\mathcal{L}_{k+1}, \dots, \mathcal{L}_K$. To insert copy operations correctly, we must know the earliest iteration at which an anti-dependence is violated. In line 42, we identify each write access by not only the iteration at which the access is executed but also the loop nest that contains the write reference as well as the assignment whose LHS is that write reference. In line 43, we compute $\min_{\prec} \overline{RW}_A(k)$. Let the specifying constraint for this set be $\mathcal{P}(I, (I', k', s'))$. This set is defined as follows:

$$\begin{aligned}\min_{\prec} \overline{RW}_A(k) = \{(I', k', s') \mid \mathcal{P}(I, (I', k', s')) \\ \wedge \nexists (I'', k'', s'') \text{ s.t. } (I'', k'', s'') \prec (I', k', s') \\ \wedge \mathcal{P}(I, (I'', k'', s''))\}\end{aligned} \quad (7)$$

If all constraints in $\mathcal{P}(I, (I', k', s'))$ are affine, $\min_{\prec} \overline{RW}_A(k)$ can be computed parametrically (in terms of I) using the PIP [4] or Omega Calculator [11].

$\min_{\prec} \overline{RW}_A(k)$ contains the earliest writes at which some anti-dependences are violated in the program P' generated by *ElimWW_WR*. In line 48, we insert the copy statements to copy the old values of A at these iterations just before they are overwritten. In lines 43 – 45, we make sure that the copied values are used correctly only at the iterations defined by the predicate C_R in line 47.

The elimination of the fusion-preventing anti-dependences relies on the fact that all the fusion-preventing flow and output dependences have been eliminated.

Theorem 2 *The program P'' generated ElimRW has the same input/output behaviour as the original program (1).*

The number of copying arrays introduced for an existing array depends only on the number of fused loop nests. If array expansion [5] is used to eliminate output and anti-dependences, the amount of extra space introduced often depends on the problem size. For example, a 2-D array of size $N \times N$ is often expanded into a 3-D array of size $N \times N \times N$. In our case, the worst-case scenario is $N \times N \times L$, where L is the number of loop nests in the program. In addition, the following optimisations are often possible.

If all write references for an array A are located in one loop nest, then at least one copying array for A is required.

Theorem 3 *Let A be an array in the input program P' to ElimRW. If the write references of A are all contained in one loop nest only, then the copying arrays that may be introduced for A by ElimRW can be combined into one array.*

This theorem leads directly to the following result.

Theorem 4 *If the input program P' to ElimRW is free of output dependences between different write references, then the copying arrays $H_{A,2}, \dots, H_{A,K}$ that may be introduced for an existing array A can be combined into one array.*

3.2 Fusing Arbitrary Loop nests Program-wise

We can apply our algorithm *FixDeps* to a program to form increasingly larger perfect loop nests inside out. If desired, a single perfect loop nest can be eventually created.

Let us apply our algorithm to the four kernels given in Figure 1. By applying code sinking, we obtain the fused programs given in Figure 3. In each case, all perfect loop nests fused are numbered. Note that the last iteration of the `k` loop in LU has been peeled. Otherwise, loop peeling can be applied after our algorithm has been applied.

It is straightforward to fix the fusion-preventing dependences for these kernels to obtain the final programs given Figure 4. For LU, $WR_m(2,3) \neq \emptyset$. The final program is obtained by tiling the `i` loop with a tile size of `N`. The treatment of QR is similar. Since $WR_{norm}(2,3) \neq \emptyset$, the `k` loop for the loop nest 2 has been tiled by a tile size of `N`. The fused program for Cholesky is already legal. Finally, the anti-dependences in Jacobi are violated by loop fusion since $RW_A(1,2) \neq \emptyset$. These anti-dependences are fixed by array copying. We have applied some optimisations to copy more boundary values of `A` to simplify the if conditionals introduced in line 48 of *ElimRW*. The array L can be eliminated since $L(j,i)$ can be replaced by a scalar.

No extra memory space is introduced for these kernels.

4 Experiments

We evaluate this work using the four kernels given in Figure 1 on an SGI Octane2 with a 600MHz MIPS R14000A processor running IRIX64 6.5. Both L1 and L2 data caches are 2-way associative with LRU replacement. The L1 data cache has a size of 32KB with a line size of 32B while the L2 (unified) cache has a size of 2MB with a line size of 128B. Our SGI Octane has 3GB of RAM.

For each kernel, `seq` denotes its sequential program and `tiled` its tiled version. The tiled programs are obtained from the fused codes given in Figure 4 as follows. For LU and Cholesky, the outermost `k` loop is tiled. For QR, the outermost `i` and `j` loops are tiled. For Jacobi, we first apply $\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$ to skew the three loops in the fused code and then permute the time loop to the innermost position. Moving the time loop inside enables the temporal reuse carried by the loop to be exploited. Finally, all the three loops are tiled. Note that code sinking introduces some if conditionals into the fused programs given in Figure 3. In the tiled codes, the effect of code sinking is undone as much as possible.

All the kernels are tiled for the L1 data cache only. We have experimented with two tile-size-selection algorithms to compute the tile sizes for different problem sizes, `LRW` from Wolf and Lam [13] and `PDAT` from Panda *et al.* [10]. `LRW` computes the largest square tile such that the number of self-interferencing cache misses for one array reference is minimised. `PDAT` uses the fixed tile size $\sqrt{\frac{K-1}{K}C}$, where C is the size of the L1 data cache and K is its associativity on the underlying machine. For each of the four kernels, the performance curves obtained using `LRW` and `PDAT` almost

IEEE COMPUTER SOCIETY

```
do k=1, N-1
   do j=k+1, N
     do i=k, N
   (1) if (i.EQ.k.AND.j.EQ.k+1)
         temp=0
         m=k
   (2) if (j.EQ.k+1)
         d=A(i,k)
         if (abs(d).GT. temp)
           temp=abs(d)
             m=i
   (3) if (j.EQ.k+1)
         if (m.NE.k)
           temp=A(k,i)
           A(k,i)=A(m,i)
           A(m,i)=temp
   (4) if (j.EQ.k+1)
         if (i.GE.k+1)
           A(i,k)=A(i,k)/A(k,k)
   (5) if (i.GT. k)
         A(i,j)=A(i,j)-A(i,k)*A(j,k)
temp=0
m=k
d= A(N, N)
   if (abs(d).GT. temp)
     temp=abs(d)
     m=N
```

(a) LU

```
do i=1, N
  do j=i, N
    do k=i, N
  (1) if (j.EQ.i.AND.k.EQ.i)
          norm=0
  (2) if (j.EQ.i)
            norm=norm+A(k,i)*A(k,i)
  (3) if (j.EQ.i.AND.k.EQ.i)
          norm2=sqrt(norm)
          asqr=A(i,i)*A(i,i)
          A(i,i)=dsqrt(norm-asqr+(A(i,i)-norm2)^2)
  (4) if (j.GE.i+1.AND.k.EQ.i)
          A(i,j)=A(i,j)/A(i,i)
  (5) if (j.GE.i+1.AND.k.EQ.i)
          X(j,i)=0
  (6) if (j.GE.i+1)
          X(j,i)=X(j,i)+A(i,k)*A(i,k)
  (7) if (j.GE.i+1.AND.k.GE.i+1)
         A(j,k)=A(j,k)-A(i,k)*X(j,i)
```

(b) QR

```
do k=1, N-1
  do j=k+1, N
     do i=j, N
   (1) if(i.EQ.j.AND.j.EQ .k+1)
         A(k,k)=sqrt(A(k,k))
   (2) if(j.EQ.k+1)
         A(i,k)=A(i,k)/A(k,k)
   (3) A(i,j)=A(i,j)-A(i,k)*A(j,k)
A(N,N)=sqrt(A(N,N))
```

(c) Cholesky

```
do t=0, M
   do i=2, N-1
     do j=2,N-1
    (1) L(j,i)=(A(j,i-1)+A(j-1,i)
               + A(j+1,i)+A(j,i+1)*0.25
    (2) A(j,i)=L(j,i)
```

(d) Jacobi

Figure 3. The fused versions of the four kernels given in Figure 1.

always coincide. Therefore, all the results presented here for tiled codes are obtained using PDAT only.

All the sequential and tiled programs are in ANSI C and compiled by the SGI MIPSpro compiler (version 7.4) at O3. For tiled codes, the compiler switch "LNO:blocking=off" is further used to disable loop tiling by the SGI compiler. (The sequential programs are equivalent to those in Figure 1 once the differences in storage order are considered.)

The Jacobi kernel has two problem size parameters, M and N, while each of the other three kernels has one problem size parameter, N. In our experiments, we have fixed $M = 500$ for Jacobi. For all the four kernels, we choose N from 200 to 2500 at multiples of 238. This captures some pathological cases about cache misses that might occur at some problem sizes [14]. All arrays are double arrays. So an array of size 512×512 fills up the 2MB L2 cache on our SGI Octane2 system. Therefore, we are able to investigate the impact of both L1 and L2 data caches on performance for all the four kernels. In our experiments, only one of the two processors in our SGI machine is used.

Figure 5 illustrates the performance improvements of the four kernels on the SGI Octane2 system. The speedups of LU range from 0.98 to 2.80, the speedups of QR range from 0.57 to 2.28, the speedups of Cholesky range from 1.11 to 4.27 and the speedups of JacobI range from 2.16 to 7.51. In all the cases, our tiled codes are simple and achieve good performance speedups consistently at all problem sizes.

Due to space limitations, we present only the performance analysis results obtained using the SGI *perfex* tool for Cholesky (abbreviated to CHOL). We measure the improved data reuse in terms of reduced L1 and L2 data cache misses, the branching overhead introduced by code sinking

```
do k=1, N-1
   do j=k+1, N
     do i=k, N
       if (i.EQ.k.AND.j.EQ.k+1)
         temp=0
         m=k
         do P=k, N
           d=A(P,k)
           if (abs(d).GT.temp)
             temp=abs(d)
             m=P
       if (j.EQ.k+1)
         if (m.NE.k)
           temp=A(k,i)
           A(k,i)=A(m,i)
           A(m,i)=temp
       if (j.EQ.k+1)
         if (i.GE.k+1)
           A(i,k)=A(i,k)/A(k,k)
       if (i.GT. k)
         A(i,j)=A(i,j)-A(i,k)*A(j,k)
 temp=0
 m=k
 d= A(N, N)
   if (abs(d).GT.temp)
     temp=abs(d)
     m=N
```

(a) LU

```
do i=1, N
   do j=i, N
     do k=i, N
       if (j.EQ.i.AND.k.EQ.i)
         norm=0
         do P=i,N
           norm=norm+A(i,P)*A(i,P)
       if (j.EQ.i.AND. k.EQ.i)
         norm2=sqrt(norm)
         asqr=A(i,i)*A(i,i)
         A(i,i)=dsqrt(norm-asqr+(A(i,i)-norm2)^2)
       if (j.GE.i+1.AND. k.EQ.i)
         A(i,j)=A(i,j)/A(i,i)
         X(j,i)=0
       if (j.GE.i+1)
         X(j,i)=X(j,i)+A(i,k)*A(i,k)
       if (j.GE.i+1.AND. k.GE.i+1)
        A(j,k)=A(j,k)-A(i,k)*X(j,i)
```

(b) QR

```
do k=1, N-1
  do j=k+1, N
     do i=j, N
       if(i.EQ.j.AND. j.EQ .k+1)
         A(k,k)=sqrt(A(k,k))
       if(j.EQ.k+1)
         A(i,k)=A(i,k)/A(k,k)
       A(i,j)=A(i,j)-A(i,k)*A(j,k)
A(N,N)=sqrt(A(N,N))
```

(c) Cholesky

```
do k=2,N-1
  H(k,1)=A(k,1)
  H(1,k)=A(1,k)
  H(k,N)=A(k,N)
  H(N,k)=A(N,k)
do t=0, M
  do i=2, N-1
    do j=2,N-1
      L(j,i)=(H(j,i-1)+H(j-1,i)
             + A(j+1,i)+A(j,i+1)*0.25
      H(j,i)=A(j,i)
      A(j,i)=L(j,i)  // L(j,i) to be replaced by a scalar
```

(d) Jacobi

Figure 4. The fused codes given in Figure 3 with all the fusion-preventing data dependences fixed.

in terms of increased branches resolved and mispredicted, and loop control overhead (due to mainly tiling) in terms of extra instructions introduced.

Figures 6 – 8 present our measurements obtained by *perfex*. The typical L1 and L2 cache miss cycles for both sequential and tiled programs are compared in Figure 6. Note that the vertical axis is drawn in the log scale. The minimum and maximum costs for an L1 data cache miss are 10.00 cycles and 14.00 cycles, respectively. Due to the out-of-order execution, the *typical cost* is 9.92 cycles. The typical, minimum and maximum costs for an L2 data cache miss are 162.55, 166.41 and 196.51 cycles, respectively. By enabling loop fusion and tiling, our approach has reduced significantly L1 and L2 cache misses (as a whole). Relative to the reduced cache miss cycles shown in Figure 6, the branching overhead is small as illustrated in Figure 7. Each "resolved" curve represents the number of resolved conditionals, which also represents the number of cycles typically consumed since it takes one cycle to resolve a conditional branch. Each "mispredicted" curve represents the number of cycles typically consumed by mispredicted branches; it is five times as many as the number of mispredicted branches since the typical cost for one misprediction is 5 cycles. Finally, Figure 8 illustrates the total instruction increases in the tiled codes due to code sinking, fusion and tiling. The relatively large increases in dynamic instruction counts are

IEEE COMPUTER SOCIETY

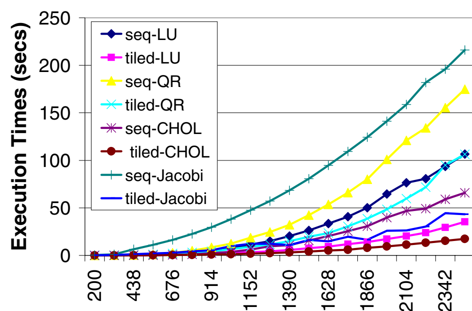

Figure 5. Performance improvements.

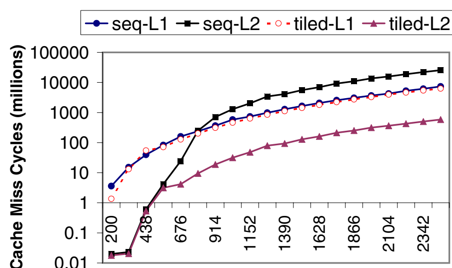

Figure 6. The typical miss cycles caused by L1 and L2 data cache misses for CHOL.

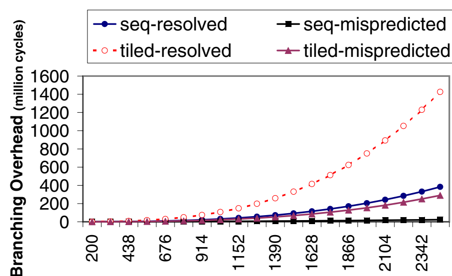

Figure 7. The typical cycles caused by branch resolutions and mispredictions for CHOL.

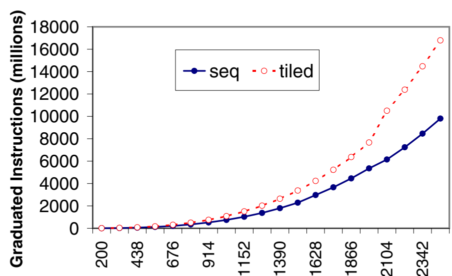

Figure 8. Graduated instructions for CHOL.

observed at all problem sizes. However, the extra instructions introduced are all integer operations (e.g., loads, stores and conditionals), which each take usually one cycle to execute. On the other hand, eliminating one single L2 data cache miss saves typically at least (typical L2 miss penalty - typical L1 miss penalty) = 162.55 – 9.92 = 152.63 cycles. By comparing Figure 6 with Figures 7 and 8, we see clearly that the benefit from the improved data reuse has far outweighed the cost due to branching and loop control overheads, resulting in performance speedups for Cholesky at all the problem sizes.

Figures 6 shows that our method may impact L1 and L2 cache misses differently for different kernels. While being almost equally effective for reducing both L1 and L2 misses in QR and Jacobi (not shown here), our method is far more effective in reducing L2 misses for LU and Cholesky.

Unlike the other three kernels, the tiled code for Jacobi is more efficient than the sequential version. Fusing the two loop nests in the sequential code results in one single perfect loop nest. In the tiled code, no extra conditionals are introduced. By fusing the two loop nests in the sequential code, we have also reduced the number of array loads in the tiled code by an average of 40.9%. The net effect is an average of 3.4% reduction in the the number of instructions executed in the tiled code. The speedups of JacobI are the mostly impressive, ranging from 2.16 to 7.51. Based on the performance analysis for Jacobi, we may be able to achieve better speedups for the other kernels if we can reduce the performance overheads as illustrated in Figure 8.

5 Related Work

Carr and Lehoucq [2] introduce some special-purpose techniques for tiling matrix factorisations: LU with and without pivoting, Cholesky and QR. They conclude that LU with partial pivoting and QR given in Figure 1 are not blockable based on dependence information alone. They show how the compiler can generate tiled codes for these two kernels by applying pattern matching for LU with pivoting or starting with a different algorithm for QR. We can generate tiled codes for these two kernels starting from the programs given in Figure 1 based on the dependence information only.

Song and Li [12] describe special-purpose techniques for tiling stencil computations. They transform Jacobi into a perfect loop nest by applying odd-even array copying to remove anti- and output dependences. Our approach eliminates flow and output dependences by loop tiling and anti-dependences by array copying for any affine loop nests. In the case of Jacobi, we obtain a single loop nest by applying array copying only. The tiled code is simple and achieves good performance speedups on three different architectures.

Kodukula *et al.* [8] describe a data shackling approach that is applicable to both perfect and imperfect loop nests. As they mention, their approach cannot handle stencil codes such as Jacobi and Gauss-Seidel. Ahmed *et al.* [1] present an approach to improving data reuse in any affine loop nests by means of iteration space transformations. They embed the iteration spaces of different loop nests in a program into a common iteration space such that the resulting program exhibits better data reuse. The embedding process must ensure that all dependences in the original program are pre-

IEEE COMPUTER SOCIETY

Method	LU	QR	Cholesky	Jacobi
Matrix Factorisations [2]	√	√	√	×
Stencil Computations [12]	×	×	×	√
Data Shackling [8]	√	√	√	×
Iteration Space Transforms [8]	×	×	√	√
This Work	√	√	√	√

Table 1. A comparison of five methods in terms of their ability in handling the four kernels given in Figure 1.

served. As a result, the dimensionality of the common iteration space may be larger than that of any existing iteration space in the original program. In this case, the effectiveness of tiling the resulting program becomes problematic. As they mention, their approach cannot handle LU with pivoting and QR. Recently, Menon *et al.* [9] discuss exclusively how to carry out automatic restructuring of codes such as LU with pivoting by symbolic analysis. In contrast, we attempt to solve the problem of tiling imperfect loop nests from a different angle. We allow arbitrary loop nests to be fused but we remove all the fusion-preventing dependences by applying loop tiling and array copying automatically.

Table 1 provides a comparison of our work with the previous research efforts in terms of their capability in handling the four important kernels in scientific computations. Our approach is the only one that can handle both matrix factorisations and stencil computations in a unified framework.

6 Conclusion

This paper presents a new approach to improving data reuse in imperfect loop nests. The basic idea is to fuse the loop nests in a program into larger perfect loop nests and then eliminate all the fusion-preventing dependences so that the resulting program is correct. The perfect loop nests formed can be tiled for improved cache performance. We eliminate the fusion-preventing flow and output dependences by means of automatic loop tiling and the fusion-preventing anti-dependences by means of automatic array copying. Our approach is applicable to affine loop nests as well as some non-affine ones including LU with pivoting as an important example. In comparison with some existing techniques as shown in Table 1, our approach can handle the four important kernels shown in Figure 1 for both matrix factorisations and stencil computations in a unified framework. Our approach is simple and generates simple tiled code for these four kernels. The tiled codes achieve consistently good performance improvements as validated by our experiments on an SGI Octane2 high-performance computer system. The benefit from the significantly reduced L1 and L2 cache misses has far more than offset the branching and loop control overhead introduced by our

One future work is to generalise loop distribution (which is the inverse of loop fusion) to make it more widely applicable. Another is to to develop a cost model for guiding our and other transformations for locality enhancement in whole programs.

References

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *14th international conference on Supercomputing*, pages 141–152. ACM Press, 2000.

[2] S. Carr and R. B. Lehoucq. Compiler blockability of dense matrix factorizations. *ACM Trans. Math. Softw.*, 23(3):336–361, 1997.

[3] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *15th International Parallel & Distributed Processing Symposium*, page 38. IEEE Computer Society, 2001.

[4] P. Feautrier. Parametric integer programming. *Operations Research*, 22:243–268, 1988.

[5] P. Feautrier. Dataflow analysis for array and scalar references. *Int. J. of Parallel Programming*, 20(1):23–53, Feb. 1991.

[6] F. Irigoin and R. Triolet. Supernode partitioning. In *15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, California., Jan. 1988.

[7] I. Kodukula. *Data-centric Compilation*. PhD thesis, Cornell University, 1998.

[8] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multilevel blocking. In *ACM SIGPLAN '97 Conference on Programming Language Design*, pages 346–357, 1996.

[9] V. Menon, K. Pingali, and N. Mateev. Fractal symbolic analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):776–813, 2003.

[10] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Transactions on Computers*, 48(2):142–149, 1999.

[11] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Comm. ACM*, 35(8):102–114, Aug. 1992.

[12] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 215–228, May 1999.

[13] M. Wolf and M. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI'91)*, pages 30–44, Jun. 1991.

[14] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation*, Jun. 1991.

[15] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.

[16] M. J. Wolfe. More iteration space tiling. In *Supercomputing '88*, pages 655–664, Nov. 1989.

[17] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[18] J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

[19] J. Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, Boston, 2000.

[20] H. Zima. *Supercompilers for Parallel and Vector Computers*. Frontier Series. Addison-Wesley (ACM Press), 1990.

0190-3918/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY