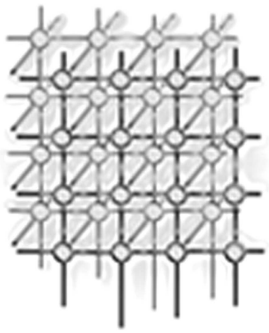
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2006; **00**:1–15 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Efficient Search-Space Pruning for Integrated Fusion and Tiling Transformations



Xiaoyang Gao[†], Sriram Krishnamoorthy[†], Swarup Kumar Sahoo[†], Chi-Chung Lam[†], Gerald Baumgartner[‡], J. Ramanujam[*], P. Sadayappan[†]

SUMMARY

Compile-time optimizations involve a number of transformations such as loop permutation, fusion, tiling, array contraction, etc. Determination of the choice of these transformations that minimizes the execution time is a challenging task. We address this problem in the context of tensor contraction expressions involving arrays too large to fit in main memory. Domain-specific features of the computation are exploited to develop an integrated framework that facilitates the exploration of the entire search space of optimizations. In this paper, we discuss the exploration of the space of loop fusion and tiling transformations in order to minimize the disk I/O cost. These two transformations are integrated and pruning strategies are presented that significantly reduce the number of loop structures to be evaluated for subsequent transformations. The evaluation of the framework using representative contraction expressions from quantum chemistry shows a dramatic reduction in the size of the search space using the strategies presented.

KEY WORDS: loop fusion, loop tiling, integrated loop transformations, out-of-core computations, pruning the search-space of optimizations, tensor contractions

1. Introduction

Optimizing compilers incorporate a number of loop transformations such as permutation, tiling, fusion, etc. Considerable work has addressed loop tiling for enhancement of data locality [4, 5, 9, 14, 19, 20, 23, 24, 25, 26]. Much work has also been done on improving locality and/or parallelism by loop fusion

[*]Correspondence to: J. Ramanujam, Department of Electrical and Computer Engineering and Center for Computation and Technology, Louisiana State University, Baton Rouge, LA 70803, USA. E-mail: jxr@ece.lsu.edu
[†]Department of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210, USA. E-mail: {gaox,krishnsr,sahoo,clam,saday}@cse.ohio-state.edu
[‡]Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA. E-mail: gb@csc.lsu.edu
Contract/grant sponsor: US National Science Foundation; contract/grant number: 0121676, 0121706, 0403342, 0508245, 0509442, and 0509467

[7, 8, 10, 11, 22]. Fusion can create imperfectly nested loops, which are more complex to tile effectively than perfectly nested loops. Several works have addressed the tiling of imperfectly nested loops [2, 23]. Although there has been much progress in developing unified frameworks for modeling a variety of loop transformations [1, 2, 16, 17, 26], their use has so far been restricted to optimization of indirect performance metrics such as reuse distance, degree of parallelism, etc.

The development of model-driven optimization strategies that target direct performance metrics remains a difficult task; in particular, the size of the search space of possible transformations is a major factor in this. In this paper, we consider the specific domain of tensor contractions (generalized matrix products) involving tensors too large to fit into physical memory. We use special properties of the computations in this domain to integrate the various transformations and investigate pruning strategies to reduce the search space to be explored.

The large sizes of the tensors involved require the development of *out-of-core* implementations that orchestrate the movement of data between disk and main memory. In this paper, we discuss the integration of loop fusion and tiling transformations with the objective of minimizing disk I/O cost. Loop fusion is used here in the context of fusing the loops involved in a set of tensor contractions. We first evaluate the set of all fusions to be explored. For each fusion structure, all loop permutations and I/O placements would be evaluated. A generalized tiling approach is presented that significantly reduces the number of loop structures to be explored. It also enables subsequent optimizations of I/O placements and loop permutations. This approach enables an exploration of the entire search space using a realistic performance model, without the need to resort to heuristics and search of a limited subspace of the search space to limit search time.

The rest of this paper is organized as follows. In the next section, we elaborate on the computational context of interest and introduce some preliminary concepts. Section 3 describes a tree partitioning algorithm. In Section 4, we propose a loop structure enumeration algorithm and prove its completeness. An overview of the program synthesis system, of which the presented framework is a part, is given in Section 5. The reductions in the space of loop structures to be explored is shown for representative computations in Section 6. Conclusions are provided in Section 7.

2. Computational Context

The work presented in this paper is being developed in the context of the Tensor Contraction Engine (TCE) program synthesis tool [3]. The TCE targets a class of electronic structure calculations involving many computationally intensive components expressed as tensor contraction expressions. In the context of optimizing tensor contraction expressions, loop permutation, tiling and fusion are the most important transformations for enhancing performance. There has been a considerable amount of published research on loop tiling [4, 5, 9, 14, 19, 20, 23, 24, 25, 26] and loop fusion [7, 8, 10, 11, 22] as optimizing transformations. While earlier work focused on perfectly nested loops or sequences of perfectly nested loops [14, 24, 25], several frameworks have recently been proposed to transform imperfectly nested loops [1, 2, 16, 17, 26]. However, none of the prior work has addressed the use of realistic and concrete performance models along with an integrated handling of loop fusion and loop tiling. The loop transformation framework being developed for the TCE uses realistic cost models for disk I/O, along with a pruning search strategy to explore a large space of alternative loop structures obtainable through application of loop fusion, tiling and permutation. In the rest of this section, we explain the

computational form of tensor contraction expressions through an example, and place the work of this paper in the larger TCE context. The TCE takes as input a high-level specification of a computation expressed as a set of tensor contraction expressions, and transforms it into efficient parallel code. The current prototype of the TCE incorporates several compile-time optimizations which are treated in a decoupled manner, with the transformations being performed in a pre-determined sequence. In [12], we presented an integrated approach to determine the tile sizes and I/O placements for a fixed structure of the computational loops after fusion and permutation. Techniques to prune the search space of possible I/O placements, orderings, loop permutations and tiling for given a choice of fusion of tensor contractions were presented in [21]. In this paper, we present a technique to enumerate the various fusion structures and develop an algorithm to significantly reduce the number of loop nests to be evaluated for each fusion structure.

In the class of computations considered, the final result to be computed can be expressed using a collection of multi-dimensional summations of the product of several input arrays. As an example, we consider a transformation often used in quantum chemistry codes to transform a set of two-electron integrals from an atomic orbital (AO) basis to a molecular orbital (MO) basis:

$$B(a,b,c,d) = \sum_{p,q,r,s} C1(d,s) \times C2(c,r) \times C3(b,q) \times C4(a,p) \times A(p,q,r,s)$$

Here, all arrays would be initially stored on disk. The indices p, q, r and s have the same range N. The indices a, b, c and d have the same range V. Typical values for N range from 60 to 1300; the value for V is usually between 50 and 1000.

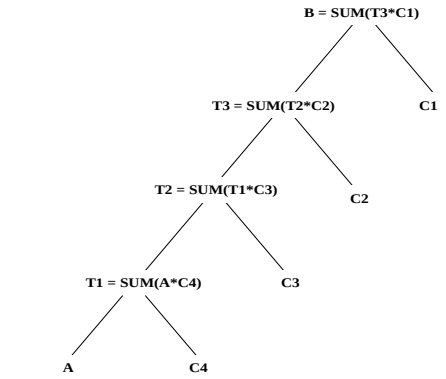
The calculation of B is done in four steps to reduce the number of floating point operations: $T1(a,q,r,s) = \sum_p C4(a,p) \times A(p,q,r,s)$; $T2(a,b,r,s) = \sum_q C3(b,q) \times T1(a,q,r,s)$; $T3(a,b,c,s) = \sum_r C2(c,r) \times T2(a,b,r,s)$; and $B(a,b,c,d) = \sum_s C1(d,s) \times T3(a,b,c,s)$.

The sequence of contractions in this form can be represented by an operation tree as shown in Fig. 1(a). The leaves of the operation tree correspond to the input arrays and the root to the output array. The interior nodes, which could be intermediate or output arrays, are produced by the tensor contraction of their immediate children. The edges in the operation tree represent the *producer-consumer* relationship between the different tensor contraction expressions. Note that an operation tree is a binary tree in which each node has either zero or two children.

Assuming that the available memory limit on the machine running this calculation is less than V^4 (which is 3TB for $V = 800$), any of the logical arrays A, $T1$, $T2$, $T3$ and B is too large to entirely fit in memory. Therefore, if the computation is implemented as a succession of four independent steps, the intermediates $T1$, $T2$ and $T3$ have to be written to disk after they are produced, and read from disk before they are used in the next step. Furthermore, the amount of disk access volume could be much larger than the total volume of the data on disk. Since none of these arrays can be fully stored in memory, it may not be possible to read each element only once from disk.

Suitable fusion of the common loops involved in the contractions that produce and consume an intermediate can reduce the size of the intermediate array, making it feasible to retain it in memory. An intermediate node is said to be fused if the loops involved in its production are fused with those involved in its consumption. Henceforth, the term intermediate node will be used to refer to both the intermediate array produced in the corresponding interior node of the operation tree, as well as the contraction that produces it. The reference shall be clear from the context.

There are many different ways to fuse the loops and they could result in different memory usage. Based on the computation context, there are no fusion-preventing dependences in tensor contraction



(a) Operation tree for the four-index transform

```
for a,r,q,s,p
[ t1_{a,q,r,s} += A_{p,q,r,s} * C4_{a,p}

for a,b,r,s,q
[ t2_{a,b,r,s} += t1_{a,q,r,s} * C3_{b,q}

for a,b,c,r,s
[ t3_{a,b,c,s} += t2_{a,b,r,s} * C2_{c,r}

for a,b,c,d,s
[ B_{a,b,c,d} += t3_{a,b,c,s} * C1_{d,s}
```

(b) Corresponding unfused code structure

Figure 1. Operation tree and unfused code structure for the four-index transform.

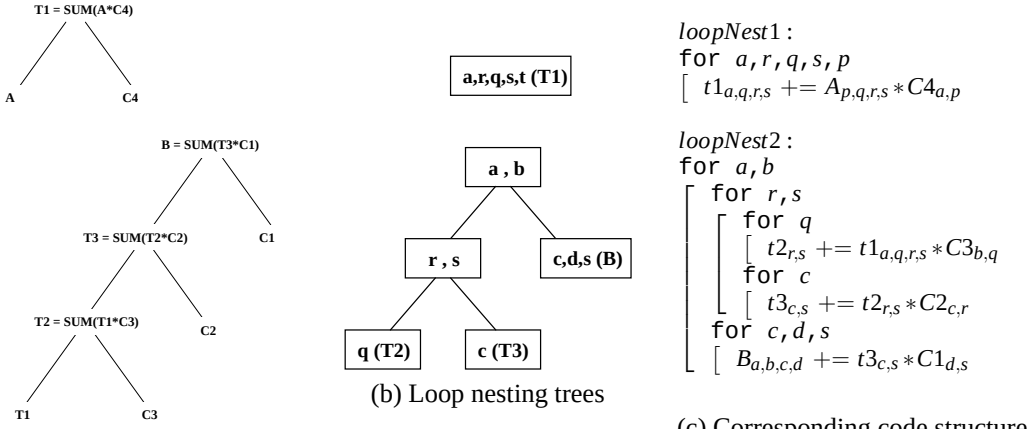
expressions [3, 13]. Given a choice of fusion, an intermediate node not fused with its parent divides the operation tree into two parts, both of which can be evaluated independently. Such an intermediate node that is not fused, is said to be a *cut-point* in the operation tree. A cut-point node is assumed to be written to disk on production and read back during its consumption. A connected operation tree without any interior cut-points is called a *fused sub-tree*. The divided operation tree for the four-index transform corresponding to $T1$ being a cut-point is shown in Fig. 2(a). The cut-point divides the operation tree into two fused sub-trees, one of which produces $T1$, and the other consumes it.

The *loop nesting tree* (LNT) represents the loop structure corresponding to a fused sub-tree. Each node in an LNT is labeled by the indices of a set of fully permutable loops that appear together at some level in the resulting overall imperfectly nested loop structure after applying loop fusion to the contraction computations in the sub-tree. The leaves represent the innermost loops, while the root represents the outermost loops. Fig. 2(b) shows possible LNTs corresponding to the two fused subtrees in Fig. 2(a). The corresponding code structure is shown in Fig. 2(c).

3. Top Sub-tree Enumeration

In this section, we discuss the procedure to enumerate the set of top sub-trees. An arbitrary operation tree with M intermediate nodes has at most $O(2^M)$ possible top sub-trees, but not all of the top sub-trees can be a fully fused operation tree. We can prune the set of possible top sub-trees by using the following two rules: (i) the fused intermediate array must be fit into memory; and (ii) the parent of two fused nodes can not be fused above.

The first rule is used to prune ineffective fusions. In general, fusing a loop between the producer of an intermediate array and its consumer eliminates the corresponding dimension of the array and reduces the array size. If the array fits in memory after fusion, no disk I/O is required for that array. On the other hand, if the array does not fit in the physical memory even after fusion, the disk I/O cost is not reduced and thus fusion does not result in any improvement. Therefore, we force the fusion of any



(a) Divided operation trees (b) Loop nesting trees (c) Corresponding code structure

Figure 2. Representations involved in generation of a fused code structure.

loops corresponding to an intermediate node to cause the resulting intermediate to reside in memory. We also assume that an intermediate array resides in disk if its producer is not fused with its consumer.

The second rule is derived from the characteristics of an operation tree. Consider an intermediate node t. If both its children are fused with it, then the loops corresponding to the summation indices in the given node must be the outermost loops; and it can not be fused with its parent anymore. Thus either t or one of its children must be a cut-point. Based on this rule, we can restrict the number of top sub-trees to $O(M^2)$.

From the first rule, it follows that contraction nodes form a chain. The second rule implies that two contraction chains may join at a root node, i.e., cut-point.

The function to enumerate the fused sub-trees rooted at a given node is shown in Algorithm 1. It is executed at each node of the operation tree in bottom-up manner and constructs the fused sub-trees rooted at a given node from those of its children. Given a node t, at first, we create a new sub-tree including only t and its direct children. Then we extend existing sub-trees from one of its children to include itself. These sub-trees can be further extended to include the parent of t; so we call them the *promising sub-trees*, which would be in a single chain form (each node has at most one fused child). We can also create sub-trees by merging two existing sub-trees from both its children. In this case, t must be a cut-point and this sub-tree cannot be extended anymore. In the algorithm, a top sub-tree Tr is identified by its $CutpointSet$, which includes cut-points in its leaves; note that input nodes are not cut-points. The field $t.PTreeSet$ represents the set of promising sub-trees and will be used to construct the fused sub-tree rooted at the parent of t. Note that we do not know whether a fused node can fit into memory at this step. This is ensured by the choice of loop structures.

4. Loop Structure Enumeration

In this section, we first present an algorithm that can generate the set of loop structures corresponding to a fused subtree. We then prove that for any loop structure S of the fused subtree, we can find

a corresponding loop structure S' in the generated set, such that S' can be transformed to S by an appropriate multi-level tiling strategy.

4.1. Enumeration Algorithm

In the previous section, we showed that a fused subtree must be in one of these two forms:

- The contraction nodes form a chain. We call it a *contraction chain*. For instance, Fig. 1(a) is such an operation tree in which the contraction chain is $T1, T2, T3, B$.

t_1 = the left child of t
t_2 = the right child of t
if t_1 is an input node, $b_1 = null$, else $b_1 = t1$
if t_2 is an input node, $b_2 = null$, else $b_2 = t2$
$TreeSet$ = empty
//Create a new sub-tree
Create a new Tree Tr with $Tr.CutpointSet = \{b_1, b_2\}$
Insert Tr into $TreeSet$
//Extending promising sub-trees from its left child
if b_1 is not $null$ **then**
  for each sub-tree st in $t1.PTreeSet$ **do**
    Create a new Tree Tr with $Tr.CutpointSet = st.CutpointSet + b2$
    Insert Tr into $TreeSet$
  end for
end if
//Extending promising sub-trees from its right child
if b_2 is not $null$ **then**
  for each sub-tree st in $t2.PTreeSet$ **do**
    Create a new Tree Tr with $Tr.CutpointSet = st.CutpointSet + b1$
    Insert Tr into $TreeSet$
  end for
end if
$t.PTreeSet = TreeSet$
//Merging sub-trees from both children, and extending the result
if both b_1 and b_2 are not $null$ **then**
  for each pair of sub-trees $st1$ in $childSet1$ and $st2$ in $childSet2$ **do**
    Create a new Tree Tr
    $Tr.CutpointSet = \{st1.CutpointSet, st2.CutpointSet\}$
    Insert Tr into $TreeSet$
  end for
end if
return $TreeSet$

Algorithm 1: EnumerateTopSubtrees(t: the root of a sub-tree) returns $TreeSet$

- The contraction nodes form two chains joining at the root node. In this case, the *contraction chain* is connected by these two chains. An example of such an operation tree is shown in Fig. 3, in which the contraction chain is $T1, T2, B, T3, T4$.

An operation tree with n contraction nodes $t_1, ..., t_n$ can be translated into a sequence of perfectly nested loops, one for each contraction. Each of the perfectly nested loops can be considered an independent loop nesting tree. The fusion of sub-trees producing and consuming an intermediate node creates an imperfectly nested loop nest, in which some of the common loops are merged. Many different choices exist in the ordering of the fusions within this sequence of perfectly nested loop nests. Choosing the best loop structure for a given fusion structure requires the determination of the tile sizes and disk I/O costs for each of the numerous possibilities, which is a very expensive operation. We tackle this problem by enumerating *maximally fused loop structures* from which all possible fusion structures can be derived by appropriate choice of loop tiling.

The process of enumeration of the fusion structure set corresponding to a fully fused operation tree can be modeled as a paranthesization problem. Consider the contraction chain $T1, T2, T3, B$ of the operation tree shown in Fig. 1(a), and one of its parenthesizations $(((T1\ T2)T3)B)$. According to the nesting of parentheses, the contraction producing T1 and consuming T1 are fused first, and the resulting loop nest is fused with the contractions producing T3 and B, in that order.

For each parenthesization, a maximally fused loop structure represented in a loop nesting tree is created by a recursive construction procedure. We call it *maximally fused* since, in the construction procedure, each intermediate node will have its indices fused as much as possible with its parent. The construction procedure is shown in Algorithm 2. It takes a parenthesization P as input, and generate the corresponding LNT. Note that, in Algorithm 2, $t_i.indices$ denotes all loop indices surrounding the contraction node t_i. A parenthesization of a contraction chain with n nodes has $n-1$ pairs of parentheses. Each pair of parentheses includes two elements, a left and a right element. Each element is either a single contraction node or a parenthesization of a sub-chain within a pair of parentheses.

Consider a parenthesization $((T1(T2\ T3))B)$ of four-index transform. Fig. 4 shows the setp-by-step construction of the corresponding LNT. The final loop structure is shown in Fig. 6(b).

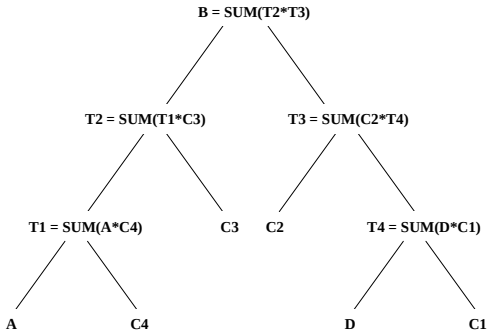


Figure 3. An operation tree with two chains

```
//Given a parenthesization, the algorithm map it to a maximally fused loop structure in LNT
l = P.left
r = P.right
if l is a parenthesization then
   lt = Construction(left)
else if l is a contraction then
   lt = Create a new LNT node
   lt.indices = l.indices
   lt.children = null
   lt.contraction = l {lt is a leaf, which includes a contraction node in it}
end if
if r is a parenthesization then
   rt = Construction(right)
else if r is a contraction then
   rt = Create a new LNT node
   rt.indices = r.indices
   rt.children = null
   rt.contraction = r {rt is a leaf, which includes a contraction node in it}
end if
comindices = lt.indices ∩ rt.indices
lt.indices = lt.indices − comindices
rt.indices = rt.indices − comindices
lnt = Create a new LNT node
lnt.indices = comindices
lnt.children = {lt, rt}
return lnt
```

Algorithm 2: Construction(P)

4.2. Completeness

In this section, we prove that the set of *maximally fused* loop structures generated by the enumeration algorithm (shown in Algorithm 2) can represent all loop structures of a fused subtree. The following definitions are provided to clarify the terms used in the proof.

Definition 1. Each leaf in an LNT includes a contraction node. The set of contraction nodes from all the leaves in an LNT is called the *leafcontractions* of the LNT.

Definition 2. Each node t in an LNT has exactly one path to the root. Let $t.upperindices$ denotes the union of all indices belonging to nodes on the path from t to the root. If a subtree *slnt* is rooted at t, we also define $slnt.upperindices$ to equal to $t.upperindices$.

Definition 3. Consider two leaves t_i and t_j in an LNT that belong to one subtree *slnt*. If there is no other subtree that contains both t_i and t_j and is a subtree of *slnt*, we say that *slnt* is the *minimal common subtree* of t_i and t_j, denoted as $MCS(t_i, t_j)$.

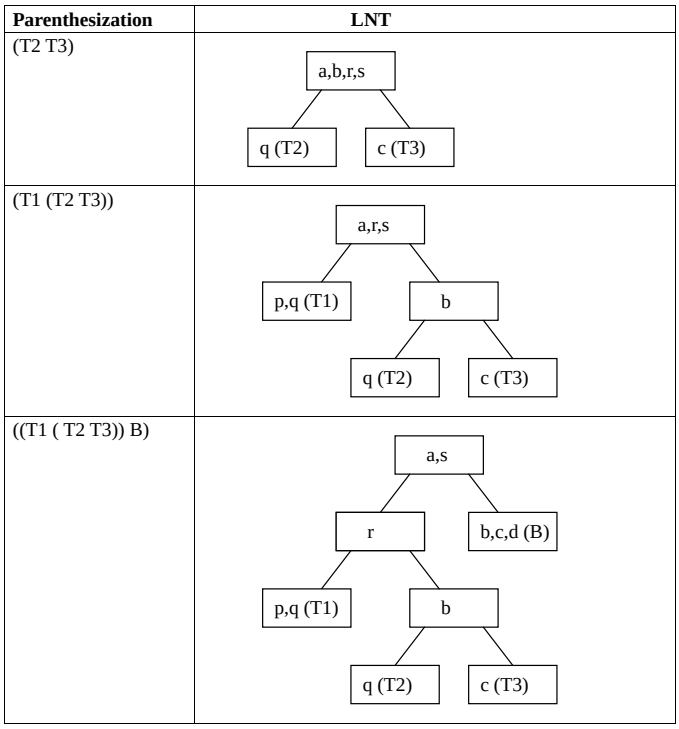


Figure 4. Construction of a maximally fused loop structure for a particular parenthesization of the four-index transform.

Given an arbitrary loop nesting tree lnt, we can map it to a maximal fused loop nesting tree lnt' that belongs to the set of *maximally fused* loop structures generated by the enumeration algorithm above, and can be translated to lnt with appropriate multi-level tiling. The mapping algorithm consists of two steps:

1. Generate a parenthesization P of the contraction chain corresponding to the given lnt using Algorithm 3.
2. Apply the construction procedure in Algorithm 2 on P to generate a maximally fused loop structure lnt'.

lnt' is obviously in the set of *maximally fused* loop structures generated by the enumeration algorithm. We now show that lnt' can be translated to lnt by sinking indices at upper levels down.

Lemma 1. For any pair of contraction nodes t_i and t_j, let $common(lnt, t_i, t_j)$ be the loops shared by t_i and t_j in lnt. We have $common(lnt, t_i, t_j) \subseteq common(lnt', t_i, t_j)$.

Proof: Given a subtree $slnt$, $slnt.upperindices$ represents all common loops shared by $slnt.leafcontractions$.

```
//Given an LNT, the algorithm map it to a corresponding parenthesization
if lnt.children ≠ null then
  P = null
  for each child c in lnt.children do
    P' = Parenthesize(c)
    if P is null then
      P = P'
    else
      P = new Parenthesization(P, P')
    end if
  end for
else
  P = c.contraction {c is a leaf and includes a contraction node}
end if
return P
```

Algorithm 3: Parenthesize(*lnt*)

There is an interesting property of *maximally fused* loop structures in the way they are constructed. For any subtree *slnt* in the LNT of a *maximally fused* loop structure, $slnt.upperindices$ includes all common loops among $slnt.leafcontractions$. In other words, it includes all possibly shared loops among $slnt.leafcontractions$. In addition, we can see from the mapping method that if *lnt* has a subtree *slnt*, then there exists a twin subtree $slnt'$ in lnt' that satisfies the following conditions:

$$\begin{aligned} slnt.leafcontractions &= slnt'.leafcontractions \\ slnt.upperindices &\subseteq slnt'.upperindices \end{aligned}$$

Given any pair of leaf nodes t_i and t_j, we define $mlnt = MCS(t_i,t_j)$ in *lnt*, where $mlnt.upperindices = common(lnt,t_i,t_j)$. Hence, we can find the corresponding subtree $mlnt'$ in lnt', where

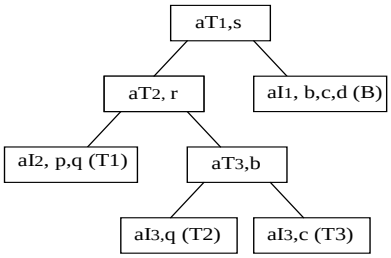
$$mlnt.upperindices \subseteq mlnt'.upperindices \subseteq common(lnt,t_i,t_j)$$

Thus, we have $common(lnt,t_i,t_j) \subseteq common(lnt',t_i,t_j)$. □

Lemma 2. If $common(lnt,t_i,t_j) \subset common(lnt',t_i,t_j)$, then we can transform lnt' to form lnt'' by sinking indices down, so that $common(lnt,t_i,t_j) = common(lnt'',t_i,t_j)$.

Proof: We define *mlnt* and $mlnt'$ as $MCS(t_i,t_j)$ in *lnt* and lnt' respectively. Any loop in $common(lnt',t_i,t_j)$ belongs to the root or an ancestor of $mlnt'$. Assuming loop *l* is in the difference of $common(lnt,t_i,t_j)$ and $common(lnt',t_i,t_j)$. We remove *l* from the original node *r*, and insert it to all children of *r*. After that, if *l* still belongs to the root or an ancestor of $mlnt'$, we repeat the sinking operation described above, until *l* is not in $mlnt'.upperIndices$ any more. The same method is applied for all indices in the difference of $common(lnt,t_i,t_j)$ and $common(lnt',t_i,t_j)$. The new LNT is denoted as lnt''. Then, we have $common(lnt,t_i,t_j) = common(lnt'',t_i,t_j)$. □

Applying the sinking operation in Lemma 2 for each pair of contraction nodes (t_i, t_j), we can transform lnt' to lnt'', which satisfies the condition: $\forall(t_i,t_j), common(lnt,t_i,t_j) = common(lnt'',t_i,t_j)$. After that, if a node *r* has no indices in *r.indices*, we remove *r* from lnt'', and put all children of *r* to its parent. Then, lnt'' is same as *lnt*.



(a) Multi-level tiling loop a

$$aT_1.range \times aI_1.range = a.range$$

$$aT_2.range \times aI_2.range = aI_1.range$$

$$aT_3.range \times aI_3.range = aI_2.range$$

(b) Range of different level tiles

Figure 5. An example of multi-level tiling in LNT.

Using *multi-level tiling strategy*, a maximally fused loop structure can be transformed into an arbitrarily fused loop structure by appropriate choice of tile sizes. *Multi-level tiling* can transform the LNT of a loop structure as follows. Each loop present in the root is split into two components, an inter-tile loop and an intra-tile loop. The intra-tile loop is placed on the child nodes of the root. Then the loops present in each of the child nodes including the intra-tile loops from the root are again split and intra-tile loops are placed on their respective child nodes. This process is performed recursively till the leaf nodes are encountered. The loop structure corresponding to the LNT can also be transformed accordingly. Fig. 5 shows the way to tile loop a in the LNT in Fig. 4 and the relationship between different tiles, where $a.range$ represents the range of loop a.

The sinking operation in an LNT can be modeled as *multi-level tiling* of the loop structure. Tiling a given fused loop structure with a tile size equal to its loop range leads to the same result as sinking the loop index from the original node to all its children. Let S and S' be loop structures represented by lnt and lnt' respectively. Since we can transform lnt' to lnt by sinking operations, we can also transform S' to S.

We illustrate the transformation procedure using an example. An arbitrary fully fused loop structure S of four-index transform is shown in Fig. 6(a), and the corresponding maximally fused loop structure S' is in Fig. 6(b). After we apply multi-level tiling, S' is translated to the format shown in Fig. 7(a). In addition, if we set ranges of inter-tile loops according to the following formulas: $aT_2 = aT_3 = sT_1 = sT_2 = sT_3 = rT_2 = qT_1 = 1$; $aT_1 = a.range$; and $rI_1 = r.range$. Now, if we remove all loops with $range = 1$, then S' can be rewritten in the format shown in Fig. 7(b), which is exactly the same as S. It should be noted that the indexing of the intermediate arrays has been shown in a more generic way.

4.3. Complexity

The total number of loop structures generated by the enumeration algorithm is the same as the number of parenthesizations of the contraction chain. For a contraction chain with n nodes, the number of all possible parenthesizations is given by the n^{th} *Catalan number*. It is exponential in the number of intermediate nodes n with an upper bound of $O(4^n/n^{3/2})$. In contrast, the number of possible loop structures is potentially exponential in the total number of distinct loop indices in the n intermediate

```
for a
  for r
    for q,s,p
      t1_{s,q} += A_{p,q,r,s} * C4_{a,p}
    for b,s,q
      t2_{b,r,s} += t1_{s,q} * C3_{b,q}
  for b,c,r,s
    t3_{b,c,s} += t2_{b,r,s} * C2_{c,r}
  for b,c,d,s
    B_{a,b,c,d} += t3_{b,c,s} * C1_{d,s}
```

(a) Arbitrary fused loop structure: S

```
for a,s
  for r
    for q
      for p
        t1 += A_{p,q,r,s} * C4_{a,p}
      for b
        t2_b += t1 * C3_{b,q}
    for b,c
      t3_{b,c} += t2_b * C2_{c,r}
  for b,c,d
    B_{a,b,c,d} += t3_{b,c} * C1_{d,s}
```

(b) Maximally fused loop structure: S'

Figure 6. An arbitrary loop structure and the corresponding maximally fused structure.

```
for aT_1,sT_1
  for rT_1,aT_2,sT_2
    for qT_1,rT_2,aT_3,sT_3
      'for p,qI_1,rI_2,aI_3,sI_3
        t1_{aI,qI,rI,sI} += A_{p,q,r,s} * C4_{a,p}
      for b,qI_1,rI_2,aI_3,sI_3
        t2_{aI,b,rI,sI} += t1_{aI,qI,rI,sI} * C3_{b,q}
    for b,c,rI_1,aI_2,sI_2
      t3_{aI,b,c,sI} += t2_{aI,b,rI,sI} * C2_{c,r}
  for aI_1,b,c,d,sI_1
    B_{a,b,c,d} += t3_{aI,b,c,sI} * C1_{d,s}
```

(a) After inserting intra-tile loops

```
for aT_1
  for rT_1
    for p,qI_1,sI_3
      t1_{aI,qI,rI,sI} += A_{p,q,r,s} * C4_{a,p}
    for b,qI_1,sI_3
      t2_{aI,b,rI,sI} += t1_{aI,qI,rI,sI} * C3_{b,q}
  for b,c,rI_1,aI_2,sI_2
    t3_{aI,b,c,sI} += t2_{aI,b,rI,sI} * C2_{c,r}
  for b,c,d,sI_1
    B_{a,b,c,d} += t3_{aI,b,c,sI} * C1_{d,s}
```

(b) After selecting proper tile counts

Figure 7. Translating S' to S by using a multi-level tiling strategy.

nodes, a considerably larger number. The fused operation tree is not very long for most representative computations. In most practical applications, a fused subtree usually has no more than five contractions in a single chain. Note that the n^{th} Catalan number is not very large when n is small. The first six Catalan numbers are listed here: $1, 1, 2, 5, 14, 42, \ldots$.

5. Integrated Framework

5.1. Optimization Process

In this section, we describe the overall program synthesis system that incorporates the steps described earlier in the paper. The program synthesis system takes an operation tree representing a set of tensor contractions as input, and generates an efficient loop structure with explicit disk I/O statements to implement the computation. The loop structure of an operation tree can be defined by

```
//Given a sub-tree rooted at t, the algorithm finds the optimal loop structure with minimal disk I/O
TopTree = EnumerateTopSubrees(t)
for each sub-tree ts_i in TopTree do
  tcs = ts_i.CutpointSet
  leafCost = 0
  for each cut-point ct in tcs do
    leafCost = LeafCost + ct.FS.Cost
  end for
  //Enumearte all fusion structures of fused sub-tree ts_i
  LoopSet = EnumerateLoop(ts_i)
  OptCost = ∞
  //Compute the minimal disk I/O cost of sub-tree ts_i
  for each loop structure ffs in LoopSet do
    mtfs = multiTiling(ffs)
    Cost = dataLocality(mtfs)
    if Cost < OptCost then
      OptCost = Cost
      OptFfs = ffs
    end if
  end for
  Cost = OptCost + leafCost
  if Cost < t.FS.Cost or t.FS = null then
    t.FS.Cost = Cost
    t.FS.TCS = TCS
    t.FS.FFS = OptFfs
  end if
end for
```

Algorithm 4: OptimalLoopStructure(t: the root of a sub-tree)

two factors: (1) the partitioning method to divide the operation tree into a set of fused sub-trees; and (2) the internal loop structure (fusion and tiling) of each fused sub-tree. The process to find the optimal loop structure may be viewed in terms of the following steps:

1. Operation Tree Partitioning: In this step, we enumerate all tree partitioning methods. A tree partitioning method divides the original operation tree into several fused sub-trees by identifying a set of cut-points. The optimal fusion structures for the sub-trees are independent of each other, and are determined separately.
2. Loop Structures Enumeration: For each fused sub-tree, we find a set of candidate fusion structures to be evaluated, as a set of LNTs. The optimal fusion structure would be included in the candidate set.
3. Intra-Tile Loop Placements: For a given LNT, we tile all loops at each node and propagate intra-tile loops to all the nodes below it.
4. Disk I/O Placements and Orderings: We then explore various possible placements and orderings of disk I/O statements for each disk array in a tiled loop structure with a pruning strategy to

determine the best placement and ordering.

5. Tile Size Selection: For each combination of loop transformations and I/O placements, the I/O cost is formulated as a non-linear optimization problem in terms of the tile sizes. The tile sizes that minimize the disk I/O cost are determined using a general-purpose non-linear optimization solver.
6. Code Generation: We calculate the disk access cost for each solution obtained, and generate code for the one with the minimal disk I/O cost.

5.2. Dynamic Programming Algorithm

A dynamic programming algorithm is employed to find the optimal loop structure of an operation tree. The algorithm calculates the minimal disk I/O cost and corresponding loop structure of each possible sub-tree of the original tree in bottom-up fashion. At a contraction node t, all sub-trees rooted at its interior nodes are evaluated before, whose minimal disk I/O cost and optimal loop structure stored in their roots. Thus, we will only evaluate these new fused sub-trees, which are rooted at t and will be referred as *top sub-trees* of t in later description. The optimal top sub-tree would be the one that minimizes the sum of the disk cost occurred in itself and other sub-trees rooted at its leaves. After traversing the entire operation tree, the optimal loop structure can be obtained by tracking back the optimal top sub-trees from root to leaves.

Algorithm 4 is employed to find the optimal loop structure for the operation tree rooted at a given node t. It will be executed at each node of the operation tree from bottom-up. For an arbitrary node t, let $t.FS$ denote its optimal top sub-tree, which includes three fields: TCS, FFS and $Cost$. TCS includes cut-points in its leaves; FFS represents its loop structure; and $Cost$ represents the disk I/O cost occurred in it.

In the algorithm, the function *EnumerateTopSubtree(t)* returns the set of all possible top sub-trees of t. After that, each of these sub-trees is evaluated in turn to determine the optimal loop structure. The initial cost of a sub-tree is the sum of the costs of its leaves. Then, for a fused sub-tree ts, the function *EnumerateLoop(ts)* will return the set of candidate fusion structures represented by loop nesting trees. Given a fully fused loop structure ffs, the function *multiTiling(ffs)* will insert multi-level intra-tile loops in it and return a tiled loop structure. For a tiled loop structure $mtfs$, the search space of disk I/O placements and orderings, loop permutations and tile sizes is modeled and pruned as a non-linear optimization problem, which is then solved to determine the minimal disk I/O cost. This process is encapsulated in the procedure *dataLocality(mtfs)*. The implementation details can be found in [21].

5.3. Example

For the operation tree in Fig. 1(a), we start at the lowest contraction node $T1$, which has only one top sub-tree $t11$ as showed in Fig. 8(a). Then we have $T1.FS = t11$. The optimal loop structure and minimal cost of $t11$ is calculated using the functions $EnumerateLoop(t11)$, $multiTiling(t11)$, and $dataLocality(t11)$. For simplicity, we do ignore the specifics of this function and assume that $t11.Cost = 100$.

The second node $T2$ has two top sub-trees $t21$ and $t22$ as shown in Fig. 8(b). Assuming the internal disk cost of $t21$ is 150 and of $t22$ is 200. But, since the leaf $T1$ of sub-tree $t21$ is the root of another

sub-tree $t11$, the total disk cost of $t21$ would be 250, higher than $t22$. So, we have $T2.FS = t22$, $T2.FS.TCS =$ and $T2.FS.Cost = 200$.

Fig. 8(c) represents three top sub-trees of node $T3$, where we assume their internal disk cost are 200, 250, and 300 respectively. Since $t22$ is rooted at a leaf of $t31$ and $t11$ is rooted at a leaf of $t32$, we get the total cost of these sub-tree as $t31.Cost = 400$, $t32.Cost = 350$, and $t33.Cost = 300$. $t33$ has the minimal total cost, then the optimal top sub-tree of $T3$ would be $t33$ with $Cost = 300$.

The four sub-trees identified at the root of the operation B, are shown in Fig. 8(d). The internal cost of these sub-trees are assumed to be 250, 300, 350, and 500. The optimal top sub-tree of B is determined to be $tB3$ with $Cost = 450$, which is also the minimal disk cost of the given operation tree in Fig. 1(a). The optimal tree partitioning method of the operation tree can be obtained by tracking back from the cut-points set of the root. $B.FS.TCS$ has one cut-point $T1$ and $T1.FS.TCS$ is empty. Hence the optimal tree partitioning method will divide the operation tree into two sub-trees at node $T1$. The optimal loop structures of these sub-trees can be found in $B.FS.FFS$ and $T1.FS.FFS$.

6. Results

The enumeration algorithm discussed in Section 4.1 generates a set of candidates loop structures to be considered for data locality optimization. Without this algorithm and generalized tiling, the set of loop structures to be evaluated might be too large, precluding their complete evaluation and necessitating the use of heuristics.

We evaluate the effectiveness of our approach using the following tensor contractions from representative computations from the quantum chemistry domain.

1. **Four-index transform (4index):** This is the sequence of contractions introduced in Section 2.
2. **CCSD:** The second and the third computations are from the class of Coupled Cluster (CC) equations [6, 15, 18] for ab initio electronic structure modeling. The sequence of tensor contraction expressions extracted from this computation is shown as follows:

$$S(j,i,b,a) = \sum_{l,k}(A(l,k,b,a) \times (\sum_{d}(\sum_{c}(B(d,c,l,k) \times C(i,c)) \times D(j,d)))$$

3. **CCSDT:** This is a more accurate CC model. A sub-expression from the CCSDT theory is:

$$\begin{aligned} S(h3,h4,p1,p2) = {} & \textstyle\sum_{p9,h6,h8}(y\_ooovvv(h8,h6,h4,p9,p1,p2) \times \\ & \textstyle\sum_{h10}(t\_vo(p9,h10) \times \sum_{p7}(t\_vo(p7,h8) \times \\ & \textstyle\sum_{p5}(t\_vo(p5,h6) \times v\_oovv(h10,h3,p7,p5))))) \end{aligned}$$

We evaluated the fused subtree corresponding to the entire operation tree without any cut-points. The number of all possible loop structures and the number of candidate loop structures enumerated by our approach are shown in Table I. It can be seen that a very large fraction of the set of possible loop structures, up to 98%, is pruned away using the approach developed in this paper.

7. Conclusions

In this paper, we discussed the exploration of the space of loop fusion and tiling transformations in order to minimize the disk access cost of tensor contraction evaluation. These two transformations were

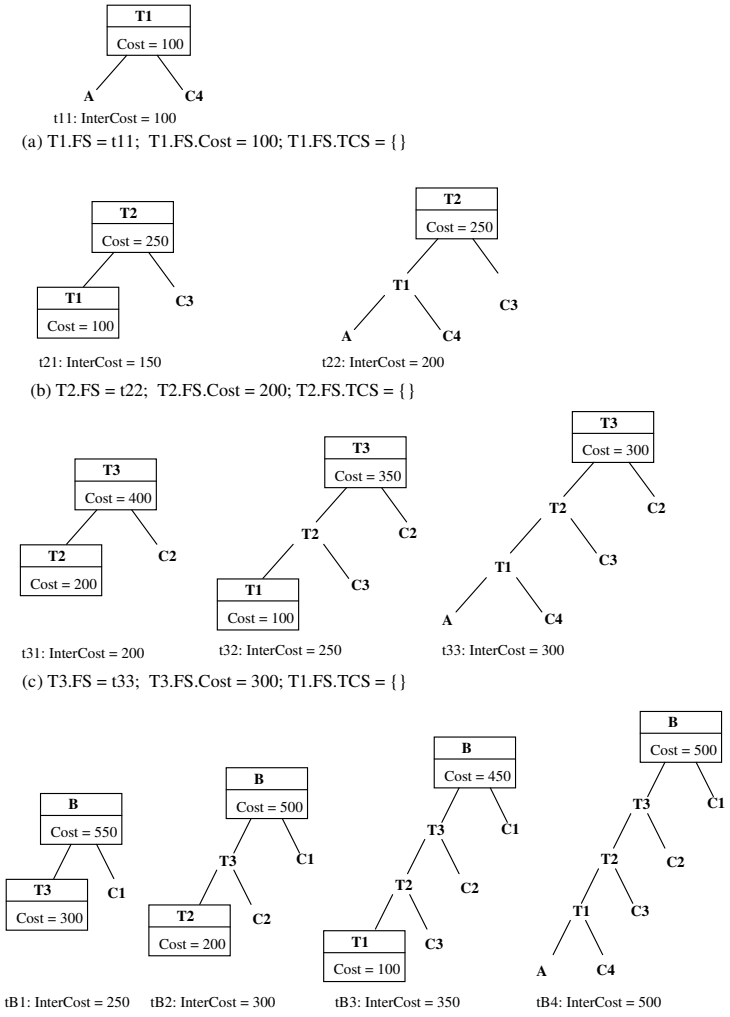


Figure 8. How to find the optimal loop structure of an operation tree by Algorithm 4

Table I. Effectiveness of pruning of loop structures.

	#Contractions	#Loop structures		Reduction
		Total	Pruned	
4index	4	241	5	98%
CCSD	3	69	2	97%
CCSDT	4	182	5	98%

integrated and pruning strategies are presented that significantly reduce the number of loop structures to be evaluated for subsequent transformations. We discussed approaches to partitioning the operation tree into fused sub-trees and generating a small set of "maximally-fused" loop structures that "cover" all possible imperfectly nested fused loop structures. The approach was evaluated on a set of computations representative of the targeted quantum chemistry domain and a significant reduction was demonstrated in the number of loop structures to be evaluated.

ACKNOWLEDGEMENTS

This work is supported in part by the National Science Foundation through awards 0121676, 0121706, 0403342, 0508245, 0509442, and 0509467.

REFERENCES

1. N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly nested loops. In *Proc. of ACM Intl. Conf. on Supercomputing*, 2000.
2. N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loops nests. In *Proc. of SC 2000*, 2000.
3. G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. *Proceedings of the IEEE*, 93(2):276–292, February 2005.
4. J. Chame and S. Moon. A tile selection algorithm for data locality and cache interference. In *Proc. of ACM Intl. Conf. on Supercomputing*, pages 492–499, 1999.
5. S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proc. of the SIGPLAN '95 Conference on Programming Languages Design and Implementation*, 1995.
6. T. Crawford and H. F. Schaefer III. An Introduction to Coupled Cluster Theory for Computational Chemists. In K. Lipkowitz and D. Boyd, editor, *Reviews in Computational Chemistry*, volume 14, pages 33–136. John Wiley, 2000.
7. C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *J. Parallel Distrib. Comput.*, 64(1):108–134, 2004.
8. G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective Loop Fusion for Array Contraction. In *Proc. of the Fifth LCPC Workshop*, 1992.
9. S. Ghosh, M. Martonosi, and S. Malik. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. In *Proc. of the Eighth ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1998.
10. K. Kennedy. Fast greedy weighted fusion. In *Proc. of ACM Intl. Conf. on Supercomputing*, 2000.
11. K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proc. of Languages and Compilers for Parallel Computing*, pages 301–320. Springer-Verlag, 1993.
12. S. Krishnan, S. Krishnamoorthy, G. Baumgartner, C. Lam, J. Ramanujam, P. Sadayappan, and V. Choppella. Efficient synthesis of out-of-core algorithms using a nonlinear optimization solver. *Journal of Parallel and Distributed Computing*, 66(5):659–673, May 2006.
13. C. Lam. *Performance Optimization of a Class of Loops Implementing Multi-Dimensional Integrals*. PhD thesis, The Ohio State University, Columbus, OH, August 1999.
14. M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. of Fourth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1991.
15. T. J. Lee and G. E. Scuseria. Achieving chemical accuracy with coupled cluster theory. In S. R. Langhoff, editor, *Quantum Mechanical Electronic Structure Calculations with Chemical Accuracy*, pages 47–109. Kluwer Academic, 1997.
16. A. W. Lim and M. S. Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Partitions. *Parallel Computing*, 24(3-4):445–475, May 1998.
17. A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using ane partitioning. In *Proc. of the Eigth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 103–112, 2001.

18. J. M. L. Martin. Benchmark Studies on Small Molecules. In P. v. R. Schleyer, P. R. Schreiner, N. L. Allinger, T. Clark, J. Gasteiger, P. Kollman, and H. F. Schaefer III, editors, *Encyclopedia of Computational Chemistry*, volume 4, pages 115–128. John Wiley, 1998.
19. G. Rivera and C.-W. Tseng. A Comparison of Compiler Tiling Algorithms. In *CC '99: Proc. 8th Intl. Conf. Compiler Construction*, pages 168–182. Springer-Verlag, 1999.
20. G. Rivera and C.-W. Tseng. Tiling optimizations for 3D scientific computations. In *Supercomputing '00: Proc. 2000 ACM/IEEE conference on Supercomputing (CDROM)*, 2000.
21. S. K. Sahoo, S. Krishnamoorthy, R. Panuganti, and P. Sadayappan. Integrated loop optimizations for data locality enhancement of tensor contraction expressions. In *Proc. of Supercomputing (SC 2005)*, 2005.
22. S. Singhai and K. S. McKinley. Loop Fusion for Parallelism and Locality. In *Proc. of Mid-Atlantic States Student Workshop on Programming Languages and Systems*, 1996.
23. Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. In *Proc. of ACM SIGPLAN PLDI*, 1999.
24. M. E. Wolf and M. S. Lam. A Data Locality Algorithm. In *Proc. of ACM SIGPLAN PLDI*, 1991.
25. M. E. Wolf, D. E. Maydan, and D. J. Chen. Combining loop transformations considering caches and scheduling. In *Proc. of the Twenty Ninth Annual International Symposium on Microarchitecture*, pages 274–286, 1996.
26. Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 169–181, 2000.