Loop Optimizations for a Class of Memory-Constrained Computations*

D. Cociorva, J. W. Wilkins
Dept. of Physics
The Ohio State University
Columbus, OH 43210, USA
{cociorva,wilkins}@pacific.mps.ohio-state.edu

C. Lam, G. Baumgartner, P. Sadayappan
Dept. of Comp. & Info. Sci.
The Ohio State University
Columbus, OH 43210, USA
{clam,gb,saday}@cis.ohio-state.edu

J. Ramanujam
Dept. of Elec. & Comp. Engr.
Louisiana State University
Baton Rouge, LA 70803, USA
jxr@ee.lsu.edu

ABSTRACT

Compute-intensive multi-dimensional summations that involve products of several arrays arise in the modeling of electronic structure of materials. Sometimes several alternative formulations of a computation, representing different space-time trade-offs, are possible. By computing and storing some intermediate arrays, reduction of the number of arithmetic operations is possible, but the size of intermediate temporary arrays may be prohibitively large. Loop fusion can be applied to reduce memory requirements, but that could impede effective tiling to minimize memory access costs. This paper develops an integrated model combining loop tiling for enhancing data reuse, and loop fusion for reduction of memory for intermediate temporary arrays. An algorithm is presented that addresses the selection of tile sizes and choice of loops for fusion, with the objective of minimizing cache misses while keeping the total memory usage within a given limit. Experimental results are reported that demonstrate the effectiveness of the combined loop tiling and fusion transformations performed by using the developed framework.

1. INTRODUCTION

There has been tremendous progress in optimizing compiler technology, primarily focused on transformations that improve execution time. However there has been very little work that addresses memory space optimization and trade-off between memory requirements and execution time. In some scientific application domains, there are often various alternative implementations of the computation, that involve complex space-time trade-offs. For example, in some scientific domains (exemplified by *ab initio* methods for modeling the electronic properties of materials), the input data to the computation and the final results of the computation are relatively compact, but large intermediate arrays are used. The alternative computational forms differ in how some intermediate results are either stored and reused or recomputed due to space limitations. In previous work, we have investigated the application of algebraic transformations to minimize operation count [21, 23, 24, 25]. However these transformations can result in the use of huge temporary intermediate arrays that exceed the virtual memory limits on even large computer systems. In this paper, we address the integrated treatment of loop fusion transformations to reduce storage requirements, along with loop tiling transformations to minimize memory access time.

The optimizations addressed in this paper are relevant to some scientific application domains that are very compute-intensive and consume significant computer resources at national supercomputer centers. Many of these codes are limited in the size of the problem that they can currently solve because of memory and performance limitations. These computational structures arise in computational physics codes modeling electronic properties of semiconductors and metals [3, 15, 39], and in computational chemistry codes such as ACES II [44], GAMESS [40], Gaussian [10] and NWChem [13]. In particular, they comprise the bulk of the computation with the coupled cluster approach to the accurate description of the electronic structure of atoms and molecules [28, 36, 31, 4, 9].

The class of computations we consider in this paper involves multi-dimensional summations over products of arrays. There is considerable opportunity to reduce the total number of arithmetic operations by making use of the algebraic properties of commutativity, associativity, and distributivity. To make the overall optimization problem more tractable, we view it as two separable sub-problems:

1. Given a specification of the required computation as a multi-dimensional sum of the product of input arrays, determine an equivalent sequence of nested loop computations that uses a minimum number of arithmetic operations.

2. Given an operation-count-optimal form of the computation (from the solution to the above sub-problem), perform appropriate loop transformations to optimize its execution, subject to memory capacity limitations.

*Supported in part by the National Science Foundation under grants DMR-9520319, CCR-0073800, and NSF Young Investigator Award CCR-9457768.

ICS '01 Sorrento, Italy

The first sub-problem was previously addressed by us and reported in [25, 26] (a brief summary is provided in Section 2). A synthesis procedure was developed that transforms an input canonical representation of a multi-dimensional sum-of-products expression into an equivalent sequence of nested loops with a minimal arithmetic operation count. In this paper, we address the problem of minimizing memory access costs, subject to memory capacity limitations. We consider:

1. **Loop Tiling**: There is significant scope for temporal reuse of data in the loops that arise in this context. Since the arrays involved are often too large to fit into the cache, loop tiling has significant potential for reducing cache misses.

2. **Loop Fusion**: As mentioned above, the input to the performance optimization problem addressed in this paper is an operation-count-optimal sequence of nested loops generated using a synthesis procedure [25, 26]. The sequence of nested loops use temporary intermediate arrays, but the operation-optimal sequence may require intermediate arrays that are too large to fit in the available virtual memory. Loop fusion can be very effective in reducing the sizes of the intermediate arrays. By fusing a loop nest that produces an intermediate array with the loop nest that consumes it, the dimensionality of the intermediate array can be reduced, thereby reducing its memory requirement.

Loop tiling for enhancing data locality and parallelism has been extensively studied [2, 5, 8, 14, 37, 38, 45, 47, 43]. Loop fusion has also been studied [41, 42, 33, 32] as a means of improving data locality. There has been much less work investigating the use of loop fusion as a means of reducing memory requirements [11]. We have previously investigated the problem of finding optimal loop fusion transformations for minimization of intermediate arrays in the context of the class of loops considered here [22, 21]. However, as we elaborate in the next section, the decoupled application of known techniques for loop tiling and loop fusion transformations is unsatisfactory in this context.

In this paper, we address the use of tiling and fusion transformations in an integrated manner, and seek to minimize the number of cache misses under a constraint on the total amount of available main memory. The paper is organized as follows. Section 2 provides some background on the context in which the addressed optimization problem arises. Section 3 addresses the problem of selecting optimal tile sizes for single and multiple loop nests. A dynamic programming algorithm for finding the optimal fusion and tiling is developed in Section 4. Experimental results on an SGI Origin 2000 system are presented in Section 5. We discuss related work in Section 6. Section 7 provides conclusions.

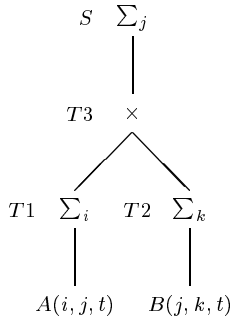
2. THE COMPUTATIONAL CONTEXT

In the class of computations considered, the final result to be computed can be expressed as multi-dimensional summations of the product of several input arrays. Due to commutativity, associativity, and distributivity, there are many different ways to obtain the same final result and they could differ widely in the number of floating point operations required. Consider the following example:

$$S(t) = \sum_{i,j,k} A(i,j,t) \times B(j,k,t) \qquad (1)$$

$$\begin{aligned} T1(j,t) &= \sum_i A(i,j,t) \\ T2(j,t) &= \sum_k B(j,k,t) \\ T3(j,t) &= T1(j,t) \times T2(j,t) \\ S(t) &= \sum_j T3(j,t) \end{aligned}$$

(a) Formula sequence



(b) Binary tree representation

Figure 1: A formula sequence and its binary tree representation.

If implemented directly as expressed above, the computation would require $2 \times N_i \times N_j \times N_k \times N_t$ arithmetic operations to compute. However, assuming associative reordering of the operations and use of distributive law of multiplication over addition is acceptable for the floating-point computations, the above computation can be rewritten in various ways. One equivalent form that only requires $N_i \times N_j \times N_t + N_j \times N_k \times N_t + 2 \times N_j \times N_t$ operations is as shown in Figure 1(a).

Generalizing from the above example, we can express integrals of products of several input arrays as a sequence of formulae. Each formula produces some intermediate array and the last formula gives the final result. A formula is either:

- a multiplication formula of the form:

$$Tr(\dots) = X(\dots) \times Y(\dots), \text{ or}$$

- a summation formula of the form:

$$Tr(\dots) = \sum_i X(\dots),$$

where the terms on the right hand side represent input arrays or intermediate arrays produced by a previously defined formula. Let IX, IY and ITr be the sets of indices in $X(\dots)$, $Y(\dots)$ and $Tr(\dots)$, respectively. For a formula to be well-formed, every index in $X(\dots)$ and $Y(\dots)$, except the summation index in the second form, must appear in $Tr(\dots)$. Thus $IX \cup IY \subseteq ITr$ for any multiplication formula, and $IX - \{i\} \subseteq ITr$ for any summation formula. Such a sequence of formulae fully specifies the multiplications and additions to be performed in computing the final result.

A sequence of formulae can be represented graphically as a binary tree to show the hierarchical structure of the compu-

tation more clearly. In the binary tree, the leaves are the input arrays and each internal node corresponds to a formula, with the last formula at the root. An internal node may either be a multiplication node or a summation node. A multiplication node corresponds to a multiplication formula and has two children which are the terms being multiplied together. A summation node corresponds to a summation formula and has only one child, representing the term on which summation is performed. As an example, the binary tree in Figure 1(b) represents the formula sequence shown in Figure 1(a). We have shown that the problem of determining the operator tree with minimal operation count is NP-complete, and have developed a pruning search procedure [25].

The operation-minimization procedure described above results in the creation of intermediate temporary arrays. Sometimes these intermediate arrays that help in reducing the computational cost create a problem with the memory capacity required.

Consider the sequence of formulae in Figure 2(a) instead of directly computing

$$S(k,l,m) = \sum_{i,j,n} A(i,j,k) \times B(i,l,m) \times C(j,l,m) \times D(j,m,n) \quad (2)$$

Let $N_i = N_j = N_j = N_k = N_l = N_m = N_n = N = 1000$ and all the arrays be of double precision. The use of the temporary intermediate arrays reduces the total number of arithmetic operations from $O(10^{18})$ to $O(10^{15})$.

The total size of the five arrays A, B, C, D and S is $5 \times N^3 = 5 \times 10^9$ elements or 40GB. However, the size of the intermediate array $T1$ is $N^4 = 10^{12}$ elements or 8TB, which likely exceeds the memory capacity of most computers and will impose a high overhead with disk I/O. Thus, the operation minimization through use of temporary intermediates has provided a benefit in reducing arithmetic operations, but has increased the memory requirements significantly.

In many situations, problems with memory requirements of the large intermediate arrays can be mitigated through loop fusion. Loop fusion merges loop nests with common outer loops into larger imperfectly nested loops. When one loop nest produces an intermediate array which is consumed by another loop nest, fusing the two loop nests allows the dimension corresponding to the fused loop to be eliminated in the array. This results in a smaller intermediate array and thus reduces the memory requirements. For the example considered, the effect of fusion is shown in Figure 2(b). The use of loop fusion can be seen to result in significant reduction to the total memory requirement. For a computation comprised of a number of nested loops, there will generally be a number of fusion choices, that are not all mutually compatible. This is because different fusion choices could require different loops to be made the outermost. In prior work, we addressed the problem of finding the choice of fusions for a given operator tree that minimized the space required for all intermediate arrays after fusion [22, 23].

Loop tiling is used to enhance temporal cache locality and thus reduce memory access costs in compute-intensive loops such as matrix-matrix product and other BLAS-3 routines. Although current commercial compiler technology is quite sophisticated regarding tiling of perfectly nested loops such as matrix-matrix multiplication, tiling of imperfectly nested loops poses a significant challenge. The state-of-the-art in shackling [20] does not allow a simple treatment of sequences of loops. Further, if loop fusion is first used to control the storage requirement, constraints are imposed on permutability of loops in loop nests, thereby constraining the amount of temporal reuse possible through tiling. This problem is illustrated through an example later in the paper.

Thus, although each of the two transformations considered — loop tiling and loop fusion — has been much studied, the independent application of these transformations is not appropriate in this context. By taking an integrated view of this optimization problem requiring both loop tiling and loop fusion, we develop a strategy that results in significant performance improvement.

3. TILE SIZE SELECTION

The purpose of this section is to provide a practical tiling procedure that minimizes the number of cache misses for computations involving large arrays. Consequently, certain approximations are made in our cost model, most of which derived from the assumption that any linear dimension of an array is significantly larger than 1.

3.1 Tiling, Data Reuse, and Memory Access Cost

There are two sources of data reuse: a) temporal reuse, with multiple references to the same memory location, and b) spatial reuse, with references to neighboring memory locations on the same cache line. To simplify the treatment in the rest of the paper, the cache line size is implicitly assumed to be one. In practice, tile sizes are determined under this assumption, and then the tile sizes corresponding to loop indices that index the fastest-varying dimension of any array are increased if necessary, to equal the cache line size. When this is done, other tiles may be sized down slightly so that the total cache capacity is not exceeded. Another practical consideration regarding spatial reuse that we do not explicitly address in this paper is that of alignment of arrays with cache-line boundaries. Where possible, it is advantageous to align array origins with cache-line boundaries. However, even if such alignment is not possible, copying of data tiles into contiguous temporary arrays is a solution to this problem.

We introduce a memory access cost model (*Cost*), a lower bound on the number of cache misses, as a function of tile sizes and loop bounds. In practice, the number of misses is higher than this lower bound due to self-interference and cross-interference between arrays. Our cost model would be exact in the theoretical case of a fully associative cache.

3.2 Single Node Tiling

First, we consider the most general form of a node in the operator-tree, that might arise in the considered context (Section 2):

$$C(i_1,...,i_n,k_1,...,k_m,l_1,...,l_p) = \sum_{j_1,...,j_q} A(i_1,...,i_q,j_1,...,j_n,l_1,...,l_p) \times B(j_1,...,j_q,k_1,...,k_m,l_1,...,l_p)$$

Here $j_1,...,j_q$ are the summation indices that appear in both A and B but not in C, $l_1,...,l_p$ are common to all 3 arrays, and $k_1,...,k_m$ and $i_1,...,i_n$ appear in one of the two input arrays and the result. To simplify the notation, we

$$T1(j,k,l,m) = \sum_i A(i,j,k) \times B(i,l,m)$$
$$T2(j,l,m) = \sum_n C(j,l,n) \times D(j,m,n)$$
$$S(k,l,m) = \sum_j T1(j,k,l,m) \times T2(j,l,m)$$

(a) Formula sequence

```
FOR j = 1, Nj
  FOR l = 1, Nl
    FOR m = 1, Nm
      T2 = 0
      FOR n = 1, Nn
        T2 = T2 + C(j,l,n) * D(j,m,n)
      FOR k = 1, Nk
        T1(k) = 0
        FOR i = 1, Ni
          T1(k) = T1(k)+A(i,j,k)*B(i,l,m)
        S(k,l,m) = S(k,l,m) + T1(k) * T2
```

(b) Memory-reduced version (fused)

Figure 2: A sequence of formulae and the fused code with reduced memory.

consider a logical view, where a collection of indices such as $i_1, ..., i_q$ is replaced by an index vector i:

$$C(i,k,l) = \sum_j A(i,j,l) \times B(j,k,l) \tag{3}$$

This notation divides the indices into four categories: three of them (i, j, and k) appear in two of the arrays but not in the third, while l appears in all of them. It is not possible for any index to appear in only one of the three arrays. An index that appears in A or B must also appear in C unless it is a summation index. Any summation indices must appear both in A and B; otherwise the operator tree would be non-optimal in the total number of arithmetic operations [25].

There is no possible reuse of data along the l index, as for each value of l different sections of the arrays A, B, and C are used. Hence l should be the outer-most loop, and the cost for the entire nest equals the cost for the ijk nest multiplied by the number of iterations over l.

Therefore, the general form of a node that we consider is:

$$C(i,k) = \sum_j A(i,j) \times B(j,k) \tag{4}$$

With the most general form of tiling, this computation can be expressed as:

```
FOR ii = 1, Ni, Ti
  FOR jj = 1, Nj, Tj
    FOR kk = 1, Nk, Tk
      FOR i = ii, ii + Ti - 1
        FOR j = jj, jj + Tj - 1
          FOR k = kk, kk + Tk - 1
            C(i,k) = C(i,k) + A(i,j) * B(j,k)
```

where T_i, T_j, and T_k are the tile sizes and N_i, N_j, and N_k are the extents of the arrays in the dimensions i, j, and k, respectively (Figure 3). We will refer to the loops over ii, jj, and kk as tiling loops, whereas the loops over i, j, and k will be referred to as intra-tile loops. In order to avoid some clutter, in the above example of a tiled loop nest (and elsewhere in the paper), we do not explicitly capture the proper handling of incomplete last tiles. Actual codes generated using the presented framework will of course have to use appropriate min/max functions for the loop bounds of intra-tile loops to correctly handle incomplete tiles.

If the sizes of all arrays are larger than the cache size C, it can be shown [35] that, *for this particular permutation of the tiling loops ii, jj, and kk*, the solution to minimize the cost is $T_k = 1$, $T_i = T_j = T = \sqrt{C+1} - 1$. Since C is typically much larger than 1, for all practical purposes we can approximate $T \approx \sqrt{C}$. Of course, the cache capacity constraint has to be rigorously satisfied when the solution is carried out. Also, T has to be rounded up/down to an integer, as $\sqrt{C}$ is not necessarily an integer. We choose to leave these actual implementation details aside in order to provide a simplified picture of the algorithm. Then the total cost is:

$$\textit{Cost} \approx N_i N_j + \frac{2N_i N_j N_k}{T} \tag{5}$$

In this equation, the first term represents the cost for the array A, whose elements get full temporal reuse in the kk loop, while the second term is the cost for the arrays B and C. It is important to note that the cost, as well as the optimal tile sizes, depend on the permutation of the tiling loops. The first term in Equation 5 can be replaced by $N_i N_k$ or $N_j N_k$ if different permutations are considered. The absolute minimum memory access cost for the single node problem is, therefore, equal to:

$$\textit{Cost} \approx \min(N_i N_j, N_j N_k, N_i N_k) + \frac{2N_i N_j N_k}{T} \tag{6}$$

3.3 Loop Fusion and Multiple Node Tiling

As discussed in Section 2, loop fusion is often necessary in order to reduce the sizes of the intermediate arrays and to keep the total memory requirements within the amount of physical memory available. In this section, we consider, as an example, the effects of loop fusion on the total memory requirements and the memory access cost of a three-node problem, and show that an integrated approach of tiling and fusion is needed.

Consider three equations, where the first two compute arrays C and D,

$$C(i,k) = \sum_j A(i,j) \times B(j,k), \tag{7}$$

$$D(k,m) = \sum_l E(l,m) \times F(k,m), \tag{8}$$

which are used in a subsequent computation for array G:

$$G(i,m) = \sum_k C(i,k) \times D(k,m) \tag{9}$$

Tiling of independent loop nests. If we had no constraints on the total storage required, Equations 7–9 could be simply implemented as a sequence of three disjoint sets of nested

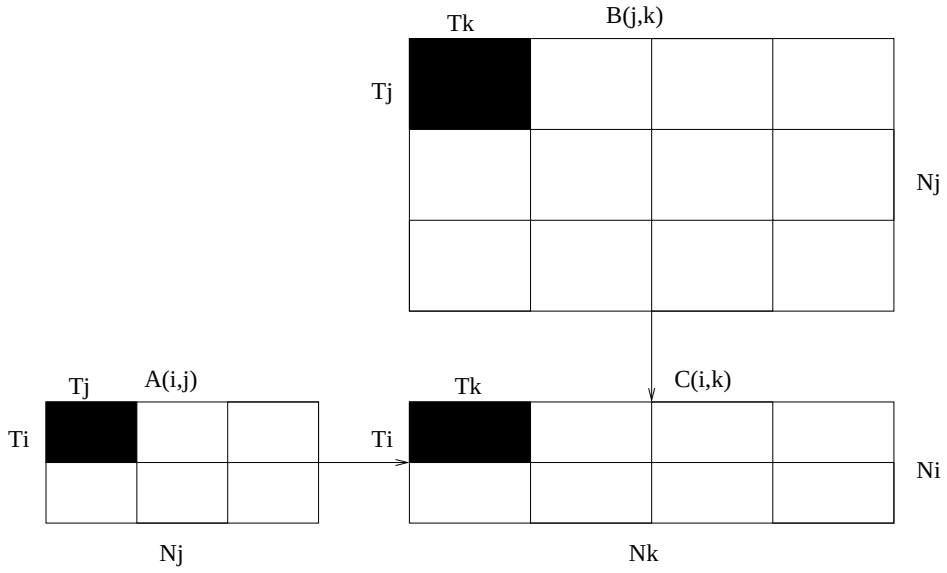


Figure 3: Example of single node tiling. Arrows point from input arrays A and B to the produced array C.

loops, which could be tiled according to Section 3.2. The memory access cost would be given by a sum of three terms, each one having the form of Equation 5. However, in practice for some applications involving *ab initio* calculations for electronic properties, it is often impossible or impractical to allocate storage for the intermediate arrays C and D.

Loop fusion. To minimize memory usage, the arrays C and D can be reduced in size by fusion of the k loop over all three nodes and by fusion of the m loop over the last two nodes. This would reduce C to a one-dimensional array and D to a scalar, thereby dramatically reducing the total storage required [22]:

```
FOR k = 1, Nk
  FOR i = 1, Ni
    C(i) = 0.0
    FOR j = 1, Nj
      C(i) = C(i) + A(i,j) * B(j,k)
  FOR m = 1, Nm
    D = 0.0
    FOR l = 1, Nl
      D = D + F(k,l) * E(l,m)
    FOR i = 1, Ni
      G(i,m) = G(i,m) + C(i) * D
```

However, this approach could result in a significant performance penalty in the form of decreased temporal reuse of cache elements. In this example, tiling along i is no longer possible, preventing any possible temporal reuse of elements of A. This is because every element of C and D would have to be fully used in computing G before it could be overwritten. Thus effective tiling of the resulting loops is constrained because of the loop fusion.

Integrated treatment of tiling and fusion. An alternative to these choices is to analyze the three sets of nested loops individually, find the loop tiling and permutation that minimizes the memory access cost for each individual node, and *then* consider the possibility of memory reduction by *fusion of the tiling loops only* (Figure 4). Since both memory access cost (Equation 5) and potential loop fusions are dependent on the permutation of the tiling loops at each node in the tree, a search algorithm is necessary to find the optimal loop permutations and loop fusions that minimize both memory access cost and memory space requirements. Such an algorithm is presented in Section 4.

Loop fusion causes a small reduction to the total number of cache misses in the cost formula 5: if an array that is produced in one loop and consumed in the following loop, is reduced by fusion (so that its extents are equal to the tile sizes along those directions), its elements will be reused at the parent node, resulting in zero cost for that array at the parent node. The implication of this observation is that when fusing the nested loops corresponding to a parent-child node pair in the operator tree, the number of cache misses for the optimally fused and tiled structure is very well approximated simply by the sum of the number of misses for the optimally tiled forms of the individual loops, counting the produced-consumed array's cost only once. In the following section, this observation is used in developing an efficient dynamic-programming algorithm that computes the optimally tiled and fused loop structure using a bottom-up traversal of the operator tree.

4. A TILING AND FUSION ALGORITHM

We now propose an efficient algorithm for the memory access cost and memory space minimization problem. Given an operator tree, our goal is to find:

- the tile sizes of individual nodes in the operator tree,
- the tiling loop permutations of individual nodes in the tree, and
- the loop fusions between adjacent nodes

such that the total memory access cost and memory space of the operator tree are optimal. Since both memory access cost and memory space cannot in general be minimized by the same solution, the goal of our algorithm is to find the set of candidate solutions defined as follows. A solution is a candidate solution if no other solution has both a lower memory access cost and a lower memory space requirement. This implies that once the set of candidate solutions is ordered by

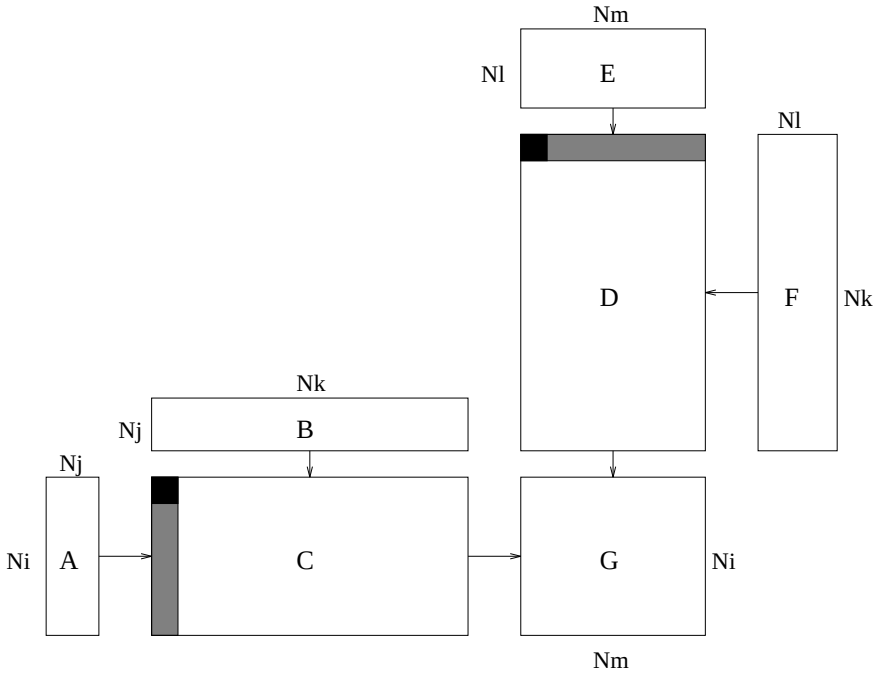


Figure 4: Example of tiling and fusion for a three-node operator tree. The arrays C and D are reduced by loop fusion to "bands," represented in the figure by shaded areas. Potential tiles for C and D are shown in black. The arrows point from consumed to produced arrays.

increasing memory access cost, it is also in decreasing memory space order. The optimal solution is the candidate solution that has the lowest memory access cost, its memory space requirement being at the same time lower than the total available memory.

We impose two constraints on the problem:

1. The operator tree is such that the formation of index vectors (as described in Section 3) from loop indices is consistent, i.e., if a pair of indices combine together in one way at one node of the operator tree, they will similarly combine at any other node where they both appear together. The case of a general operator tree (where this constraint is not satisfied) is still an unsolved open problem.

2. The arrays involved are large enough so that every linear dimension of an array is larger than the square root of the cache size. This condition is generally true in the computational domains that motivate the problem addressed by this paper.

These two assumptions permit the estimation of the memory access cost of each operator tree node (Equations 5 and 6), and the gain (in both access cost and space) from loop fusion. Therefore, for a given tiled and fused loop structure of the tree, our model completely describes its access cost and memory space.

The aim of the algorithm we present here is to efficiently enumerate the list of candidate solutions. One could imagine a naive search, in which all loop fusions would be considered, and the loop structures created by these fusions would be compared for access cost and space. The downside of this approach is that for large trees it becomes extremely time-consuming. It can be shown that the complexity of the problem increases exponentially with N, the number of nodes in the tree.

Another approach to solve this problem is to consider a dynamic programming bottom-up procedure. It may be observed that the access cost and space requirement of one tree region are unrelated to other tree regions. Hence, we can compute partial solutions at each node concerning the subtree below. This procedure has a cost that is linear in N and much faster than the exhaustive approach.

The advantage of the bottom-up approach is that inferior loop structures are pruned even at early stages of the search and the list of partial solutions at each node is relatively small. For each partial solution, we keep the following information:

- The memory access cost of the subtree below, including the root of the subtree.
- The sum of memory space requirements of all intermediate arrays in the subtree, including the array produced at the root of the subtree.
- A list of the possible fusions on the branch between the root of the subtree and its parent. This list is kept because a solution with higher memory access cost and space that allows more fusions may later improve by subsequent fusion.

Any solution that is inferior to another one in all three respects (higher memory access cost and space, and fewer allowable fusions) is discarded. Thus we keep track of only those candidate solutions for a subtree that may become part of an optimal solution. The procedure for computing the candidate solutions for a subtree rooted at node X whose two children are both intermediate arrays is given in Figure 5. If

Merge (X):
  foreach permutation p of $X.Indices$
    UpList = **PossibleFusions** $(p, X.Parent.Indices)$
    LeftList = **PossibleFusions** $(p, X.LeftChild.Indices)$
    RightList = **PossibleFusions** $(p, X.RightChild.Indices)$
    foreach solution $s1 \in X.LeftChild.Solutions$
      foreach solution $s2 \in X.RightChild.Solutions$
        foreach fusion $f1 \in s1.Fusions$
          foreach fusion $f2 \in s2.Fusions$
            if $f1 \in$ *LeftList* $\wedge$ $f2 \in$ *RightList* $\wedge$ **EitherPrefix**$(f1, f2)$ **then**
              create new solution s
              compute *s.Cost* and *s.Space* based on fusions $f1$ and $f2$
              $s.Fusions = \{f \in UpList \mid$ **EitherPrefix**$(f, f1) \wedge$ **EitherPrefix**$(f, f2)\}$
              InsertSolution $(s, X.Solutions)$

InsertSolution $(s, Solns)$:
  foreach $s' \in Solns$
    if Inferior (s, s') **then**
      return
    if Inferior (s', s) **then**
      $Solns = Solns - \{s'\}$
  $Solns = Solns \cup \{s\}$

PossibleFusions $(p, Indices) \equiv \{p' \mid p'$ is a prefix of $p \wedge p' \subseteq$ *Indices*$\}$

EitherPrefix $(f, f') \equiv f$ is a prefix of $f' \vee f'$ is a prefix of f

Inferior $(s1, s2) \equiv s1.Cost > s2.Cost \wedge s1.Space > s2.Space \wedge s1.Fusions \subset s2.Fusions$

Figure 5: Procedure for finding the solutions for a subtree rooted at X.

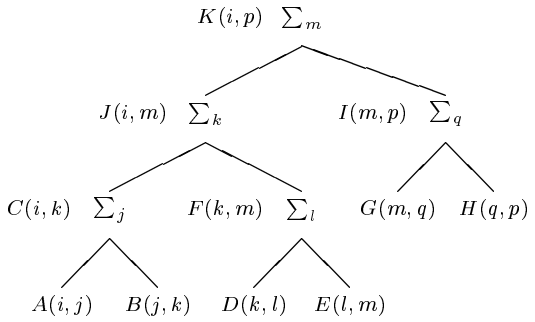


Figure 6: An example operator tree.

node X has one or more children that are input arrays, the procedure would be simpler.

As an example, consider the following multi-dimensional summation

$$K(i,p) = \sum_{j,l,k,q,m} D(k,l) \times E(l,m) \times A(i,j) \times B(j,k) \times G(m,q) \times H(q,p) \quad (10)$$

computed by a sequence of formulae represented by the operator tree shown in Figure 6. Let us consider a 32KB cache, and the sizes of the arrays to be $N_i = 64000$, $N_m = 6400$, $N_l = N_k = N_q = 640$ and $N_p = N_j = 64$ 8-byte elements. Let us also consider 64 8-byte elements to be a "memory unit", to simplify the algebra. The dimensions have been chosen to be very different, since this is characteristic of many computational problems in the domain of primary interest.

The algorithm starts with the leaf nodes and proceeds upwards. The first node in the tree could be the one creating the intermediate array $C(i,k)$ as a product of the input arrays $A(i,j)$ and $B(j,k)$. Each permutation of the three indices i, j and k is examined, according to Equation 5 (with the mention that N_i = 1,000 memory units, N_j = 1 memory unit and N_k = 10 memory units):

No.	Permutation	Possible fusions	Cost (units)	Usage (units)
1	(i,j,k)	$0, \{i\}$	2.1×10^4	10^4
2	(i,k,j)	$0, \{i\}, \{i,k\}$	3×10^4	10^4
3	(j,i,k)	0	2.1×10^4	10^4
4	(k,i,j)	$0, \{k\}, \{k,i\}$	3×10^4	10^4
5	(k,j,i)	$0, \{k\}$	2.001×10^4	10^4
6	(j,k,i)	0	2.001×10^4	10^4

Let us take a closer look at the way the table above is constructed. For each permutation, e.g., (i,j,k) (which means that i is the outer loop, and k is the inner loop), the possible fusions with the parent node may be determined by inspection (i.e., comparing common indices). The memory cost is derived according to Equation 5, while the memory usage of the intermediate array $C(i,k)$ is $N_i N_k = 10^4$, independent of the permutation. Even at this level, two out of the six partial solutions can be pruned out, because they are not optimal. Those solutions are permutations 3 and 6. Both have higher or equal memory cost and usage than solution 5, for example. Their lists of fusions are also very limited, because the summation index j at the outer loop prevents any fusion with the

parent node. The importance of fusion can be pointed out by looking at solutions 2 and 4. Although their present costs are high, their lists of fusions are large, therefore they could become starting points of an optimal solution in the future. The list of partial solutions at node C has four members, permutations 1, 2, 4 and 5. A similar list is created for node F and the merging procedure is applied to those two lists, creating a list of solutions for J, and so on.

The table below lists the total number of possible loop structures and the number of candidate solutions at each node of the tree.

Node	Number of subtree solutions	
	Dynamic programming	Exhaustive search
1. $C(i,k)$	4	6
2. $F(k,m)$	3	6
3. $J(i,m)$	8	348
4. $I(m,p)$	3	6
5. $K(i,p)$	2	19638

The exponential increase in the number of solutions is observable in the case of the exhaustive search. Nodes 1, 2 and 4, that have no children nodes, have 6 solutions. Node 3 has 348 solutions, while node 5 has 19638 solutions. In contrast, the dynamic programming approach presents a relatively constant number of subtree solutions per node.

Assuming a memory space threshold of 256MB and a cache size of 32KB, the universal tile size is $\mathrm{T} = 64$ elements and the optimal solution produced by the dynamic programming algorithm for this example is as follows.

```
FOR kk = 1, Nk, T
  FOR ll = 1, Nl, T
    FOR m = 1, Nm
      FOR k = kk, kk + T - 1
        FOR l = ll, ll + T - 1
          F(k,m) = F(k,m) + D(k,l) * E(l,m)
FOR pp = 1, Np, T
  FOR qq = 1, Nq, T
    FOR m = 1, Nm
      FOR p = pp, pp + T - 1
        FOR q = qq, qq + T - 1
          I(m,p) = I(m,p) + G(m,q) * H(q,p)
FOR ii = 1, Ni, T
  set array C(T,Nk) to 0
  FOR jj = 1, Nj, T
    FOR k = 1, Nk
      FOR i = ii, ii + T - 1
        FOR j = jj, jj + T - 1
          C(i-ii+1,k) = C(i-ii+1,k) +
                        A(i,j) * B(j,k)
  FOR mm = 1, Nm, T
    set array J(T,T) to 0
    FOR k = 1, Nk
      FOR i = ii, ii + T - 1
        FOR m = mm, mm + T - 1
          J(i-ii+1,m-mm+1) = J(i-ii+1,m-mm+1) +
                             C(i-ii+1,k) * F(k,m)
    FOR p = 1, Np
      FOR i = ii, ii + T - 1
        FOR m = mm, mm + T - 1
          K(i,p) = K(i,p) + J(i,m) * I(m,p)
```

There are three fusions that reduce the sizes of the arrays $C(N_i, N_k)$ and $J(N_i, N_m)$ to $C(T, N_k)$ and $J(T, T)$, respectively.

5. EXPERIMENTAL RESULTS

In this section, we present results of experimental performance evaluation of the effectiveness of the integrated approach to fusion and tiling developed in this paper. The algorithm from Section 4 was used to fuse and tile the code for the sequence of computations shown in Figure 4. Measurements were made on a single processor of a Silicon Graphics Origin 2000 system consisting of thirty-two 300MHz IP31 processors and 16GB of main memory. Each processor has a MIPS R12000 CPU and a MIPS R12010 floating point unit, as well as 64KB on-chip caches (32KB data cache and 32KB instruction cache), and a secondary, off-chip 8MB unified data / instruction cache. The tile size selection algorithm presented earlier assumes a single cache. It can be extended in a straightforward fashion to multi-level caches, by multi-level tiling. However, in order to simplify measurement and presentation of experimental data, we chose to apply the algorithm only to the secondary cache, in order to minimize the number of secondary cache misses. For each computation, we determined the number of misses using the hardware counters on the Origin 2000.

Three alternatives were compared:

- TFA: use of the integrated Tiling and Fusion Algorithm presented in Section 4
- FUS: use of fusion alone, for reducing memory requirement
- UNF: no explicit fusion or tiling

In order to study the performance effects in the context that motivated our study of this problem (where huge intermediate arrays pose problems with memory requirements), we chose array sizes so that one of the intermediate arrays (C) dominated the total memory requirements of the computation (Figure 4). Keeping N_i, N_j, N_l and N_m constant (N_i = 2048, N_j = 256, N_l = 256, N_m = 256), we ran performance tests for N_k increasing from 2048 to 524288 8-byte elements (Table 1). For very large N_k, array C is so large that the system's virtual memory limit for one process is exceeded, so that loop fusion is necessary to bring the total memory requirements under the limit.

The codes were all compiled with the highest optimization level of the SGI Origin 2000 FORTRAN compiler (`-O3`). To reduce cache conflicts, the array tiles were first copied into a contiguous work space whose size did not exceed that of the secondary cache, and the computations were performed in that work space. Therefore, potential cache conflicts were eliminated during the computation phase (order N^3 cost), and may have only occurred during copying (order N^2 cost). The performance data was generated over multiple runs, and average values are reported. Standard deviations were typically around 10MFLOPs. The experiments were run on a time-shared system; some interference with other processes running at the same time on the machine was inevitable, and its effects are especially pronounced for the tests with large N_k.

Table 1 presents the memory requirement, measured performance (in MFLOPs) and the number of secondary cache misses generated by the three alternatives. Both CPU and wall clock times were measured, and they were found to be essentially the same.

The main observations from the experiment are:

N_k	Memory requirement			Performance (MFLOPs)			Cache misses		
	TFA	FUS	UNF	TFA	FUS	UNF	TFA	FUS	UNF
2K	16MB	16MB	52MB	489	83	473	1.1×10^6	4.5×10^7	1.7×10^6
4K	24MB	24MB	96MB	491	72	450	2.1×10^6	7.2×10^7	2.1×10^6
8K	40MB	40MB	184MB	487	75	452	4.5×10^6	8.5×10^7	4.4×10^6
16K	72MB	72MB	360MB	485	67	448	1.0×10^7	2.4×10^8	9.5×10^6
32K	136MB	136MB	712MB	479	71	435	1.9×10^7	3.1×10^8	2.5×10^7
64K	264MB	264MB	1.4GB	483	68	420	4.8×10^7	7.5×10^8	6.6×10^7
128K	520MB	520MB	2.8GB	483	65	326	9.8×10^7	1.7×10^9	4.0×10^8
256K	1GB	1GB	5.5GB	484	66	N/A	1.7×10^8	3.5×10^9	N/A
512K	2GB	2GB	11GB	478	62	N/A	3.2×10^8	9.2×10^9	N/A

Table 1: Performance data for the integrated tiling and fusion algorithm (TFA), compared with performance data for fused alone (FUS), and unfused loops (UNF).

- The total memory requirement is minimized by complete fusion of the outer loops over k and m (FUS), which brings the intermediate arrays C and D down to a vector and a scalar, respectively. However, the memory requirement of the integrated tiling and fusion algorithm (TFA) is only slightly higher, since the tiling loops are fused, and the sizes of arrays C and D are reduced after fusion to bands of much smaller sizes. In Table 1, the difference in memory requirements between FUS and TFA cannot be seen due to the number of significant digits used. The UNF version has significantly higher memory requirements since no fusion has been applied to reduce temporary memory requirements. As N_k is increased, the UNF version requires more memory than the virtual memory limit on the system.

- The maximally fused version (FUS) has the lowest memory requirement, but incurs a high performance penalty due to the constraints imposed on the resulting loops that prevents effective tiling and exploitation of temporal reuse of some of the arrays, which leads to a higher number of cache misses, as shown in Table 1.

- The TFA and UNF versions show comparable performance for small N_k. The SGI compiler is quite effective in tiling perfectly nested loops such as the sequence of three matrix-matrix products present in the UNF version. The performance using the BLAS library routine DGEMM was found to be the same as that of the UNF version with a sequence of three nested loops corresponding to the three matrix products. For large N_k, the performance of UNF suffers significant deterioration, possibly due to secondary cache interference with other processes.

6. RELATED WORK

Much work has been done on improving locality and parallelism by loop fusion. Kennedy and McKinley [17] presented a new algorithm for fusing a collection of loops to minimize parallel loop synchronization and maximize parallelism. Two polynomial-time algorithms for improving locality were given. Recently, Kennedy [18] has developed a fast algorithm that allows accurate modeling of data sharing as well as the use of fusion enabling transformations.

Singhai and McKinley [41] examined the effects of loop fusion on data locality and parallelism in combination. They viewed the optimization problem as a problem of partitioning a weighted directed acyclic graph, in which the nodes represent loops and the weights on edges represent amount of locality and parallelism. Although the problem is NP-hard, they were able to find optimal solutions in restricted cases and heuristic solutions for the general case.

However, the work addressed in this paper considers a different use of loop fusion, which is to reduce array sizes and memory usage of automatically synthesized code containing nested loop structures. Traditional compiler research has not addressed this use of loop fusion because this problem does not arise with manually-produced programs.

Gao et al. [11] studied the contraction of arrays into scalars through loop fusion as a means to reduce array access overhead. They partitioned a collection of loop nests into fusible clusters using a max-flow min-cut algorithm, taking into account the data dependencies. However, their study is motivated by data locality enhancement and not memory reduction. Also, they only considered fusions of conformable loop nests, i.e., loop nests that contain exactly the same set of loops.

Recently, we investigated the problem of finding optimal loop fusion transformations for minimization of intermediate arrays in the context of the class of loops considered here [22]. To the best of our knowledge, the combination of loop tiling for data locality enhancement and loop fusion for memory reduction has not previously been considered.

Memory access cost can be reduced through loop transformations such as loop tiling, loop fusion, and loop reordering. Although considerable research on loop transformations for locality has been reported in the literature [8, 29, 30, 33, 45], issues concerning the need to use loop fusion and loop tiling in an integrated manner for locality and memory usage optimization have not been considered. Wolf et al. [46] consider the integrated treatment of fusion and tiling only from the point of view of enhancing locality and do not consider the impact of the amount of required memory; the memory requirement is a key issue for the problems considered in this paper. Loop tiling for enhancing data locality has been studied extensively [2, 8, 37, 38, 45, 43], and analytic models of the impact of tiling on locality have been developed [12, 27, 34]. Recently, a data-centric version of tiling called data shackling has been developed [19, 20] (together with more recent

work by Ahmed et al. [1]) which allows a cleaner treatment of locality enhancement in imperfectly nested loops. As mentioned earlier, loop fusion has also been used as a means of improving data locality [18, 41, 42, 33, 32].

7. CONCLUSION

In this paper, we have addressed the memory-access and space optimization of a class of nested loop computations that implement multi-dimensional summations of the product of several arrays. We have described a dynamic programming algorithm for finding the optimal fusion and tiling. Experimental results demonstrate the importance of the application of tiling together with fusion on such nested loops.

Acknowledgments

We would like to thank the National Center for Supercomputing Applications (NCSA) and the Ohio Supercomputer Center (OSC) for the use of their computing facilities.

8. REFERENCES

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loops. *Proc. ACM International Conference on Supercomputing*, Santa Fe, NM, 2000.

[2] J. M. Anderson, S. P. Amarasinghe and M. S. Lam. Data and Computation Transformations for Multiprocessors. *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing*, July, 1995.

[3] W. Aulbur. *Parallel Implementation of Quasiparticle Calculations of Semiconductors and Insulators*, Ph.D. Dissertation, Ohio State University, Columbus, OH, October 1996.

[4] K. L. Bak, P. Jorgensen, J. Olsen, W. Klopper. Accuracy of atomization energies and reaction enthalpies in standard and extrapolated electronic wave function / basis set calculations. *J. Chem. Phys.*, Vol. 112, pp. 9229–9242, 2000.

[5] L. Carter, J. Ferrante and S. F. Hummel. Efficient Parallelism via Hierarchical Tiling. *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, Philadelphia, PA, February 1995.

[6] S. Chatterjee, J. R. Gilbert, R. Schreiber, S.-H. Teng. Optimal Evaluation of Array Expressions on Massively Parallel Machines. *ACM TOPLAS*, 17 (1), pp. 123–156, Jan. 1995.

[7] S. Chatterjee, J. R. Gilbert, R. Schreiber, S.-H. Teng. Automatic Array Alignment in Data-Parallel Programs. *20th Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, New York, NY, pp. 16–28, January 1993.

[8] S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. *Proceedings of the SIGPLAN '95 Conference on Programming Languages Design and Implementation*, La Jolla, CA, June 1995.

[9] T. H. Dunning, Jr. A roadmap for the calculation of molecular binding energies. *J. Phys. Chem. A*, 2000 (in press).

[10] J. Foresman and A. Frisch. *Exploring Chemistry with Electronic Structure Methods: A Guide to Using Gaussian*, Second Edition. Gaussian, Inc., Pittsburgh, PA, 1996.

[11] G. Gao, R. Olsen, V. Sarkar and R. Thekkath. Collective Loop Fusion for Array Contraction. *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992, Springer-Verlag, Lecture Notes in Computer Science, pp. 281–295.

[12] S. Ghosh, M. Martonosi and S. Malik. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. *Proceedings of the 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1998.

[13] High Performance Computational Chemistry Group. NWChem, A computational chemistry package for parallel computers, Version 3.3, 1999. Pacific Northwest National Laboratory, Richland, WA 99352.

[14] K. Högstedt, L. Carter and J. Ferrante. Determining the Idle Time of a Tiling. *24th Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, New York, NY, pp. 160–173, January 1997.

[15] M. S. Hybertsen and S. G. Louie. Electronic Correlation in Semiconductors and Insulators: Band Gaps and Quasiparticle Energies. *Phys. Rev. B*, 34, 5390 (1986).

[16] K. Kennedy and K. S. McKinley. Optimizing for Parallelism and Data Locality. *1992 ACM Intl. Conf. on Supercomputing*, pp. 323–334, July 1992.

[17] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, pp. 301–320, Portland, OR, August 1993.

[18] K. Kennedy. Fast greedy weighted fusion. In *Proc. ACM International Conference on Supercomputing*, May 2000. Also available as Technical Report CRPC-TR-99789, Center for Research on Parallel Computation (CRPC), Rice University, Houston, TX, 1999.

[19] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, June 1997.

[20] I. Kodukula, K. Pingali, R. Cox, and D. Maydan. An experimental evaluation of tiling and shackling for memory hierarchy management. In *Proc. ACM International Conference on Supercomputing (ICS 99)*, Rhodes, Greece, June 1999.

[21] C. Lam. *Performance Optimization of a Class of Loops Implementing Multi-Dimensional Integrals*, Ph.D. Dissertation, The Ohio State University, Columbus, OH, August 1999.

[22] C. Lam, D. Cociorva, G. Baumgartner and P. Sadayappan. Optimization of Memory Usage and Communication Requirements for a Class of Loops Implementing Multi-Dimensional Integrals. *Proceedings of the 12th International Workshop on Languages and*

Compilers for Parallel Computing, San Diego, CA, August 1999.

[23] C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. In *International Conference on High Performance Computing*, Calcutta, India, December 1999.

[24] C. Lam, P. Sadayappan, and R. Wenger. Optimal reordering and mapping of a class of nested-loops for parallel execution. In *Languages and Compilers for Parallel Computing*, San Jose, August 1996, pp. 315–329.

[25] C. Lam, P. Sadayappan and R. Wenger. On Optimizing a Class of Multi-Dimensional Loops with Reductions for Parallel Execution. *Parallel Processing Letters*, Vol. 7 No. 2, pp. 157–168, 1997.

[26] C. Lam, P. Sadayappan and R. Wenger. Optimization of a Class of Multi-Dimensional Integrals on Parallel Machines. *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 1997.

[27] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 91.

[28] T. J. Lee and G. E. Scuseria. Achieving chemical accuracy with coupled cluster theory. In S. R. Langhoff (Ed.), *Quantum Mechanical Electronic Structure Calculations with Chemical Accuracy*, pp. 47–109, Kluwer Academic, 1997.

[29] W. Li. *Compiling for NUMA Parallel Machines*. PhD thesis, Cornell University, August 1993.

[30] W. Li. Compiler cache optimizations for banded matrix problems. In *International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[31] J. M. L. Martin. In P. v. R. Schleyer, P. R. Schreiner, N. L. Allinger, T. Clark, J. Gasteiger, P. Kollman, H. F. Schaefer III (Eds.), *Encyclopedia of Computational Chemistry*. Wiley & Sons, Berne (Switzerland). Vol. 1, pp. 115–128, 1998.

[32] K. S. McKinley. A Compiler Optimization Algorithm for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 9(8): 769–787, August, 1998.

[33] K. S. McKinley, S. Carr and C.-W. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[34] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6):641–670, June 1998.

[35] J. Navarro, A. Juan and T. Lang. MOB Forms: A Class of Multilevel Block Algorithms for Dense Linear Algebra Operations. *Proceedings of the ACM International Conference on Supercomputing*, 1994.

[36] K. A. Peterson and T. H. Dunning, Jr. (1997). The CO molecule: Role of basis set and correlation treatment in the calculation of molecular properties. *J. Molec. Struct. (Theochem)*, Vol. 400, pp. 93–117.

[37] G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, Montreal, Canada, June 1998.

[38] G. Rivera and C.-W. Tseng. Eliminating Conflict Misses for High Performance Architectures. *Proceedings of the 1998 International Conference on Supercomputing*, Melbourne, Australia, July 1998.

[39] H. N. Rojas, R. W. Godby and R. J. Needs. Space-Time Method for Ab-Initio Calculations of Self-Energies and Dielectric Response Functions of Solids. *Phys. Rev. Lett.*, 74, 1827, (1995).

[40] M. Schmidt, K. Baldridge, J. Boatz, S. Elbert, M. Gordon, J. Jensen, S. Koseki, N. Matsunaga, K. Nguyen, S. Su, T. Windus, M. Dupuis, and J. Montgomery. General Atomic and Molecular Electronic Structure System (GAMESS). *J. Comput. Chem.*, 14:1347–1363, 1993.

[41] S. Singhai and K. S. McKinley. Loop Fusion for Parallelism and Locality. *Mid-Atlantic States Student Workshop on Programming Languages and Systems, MASPLAS '96*, April 1996.

[42] S. Singhai and K. S. McKinley. A Parameterized Loop Fusion Algorithm for Improving Parallelism and Cache Locality. *The Computer Journal*, 40(6):340–355, 1997.

[43] Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. *SIGPLAN'99 Conf. on Programming Languages Design and Implementation*, pp. 215–228, May 1999.

[44] J. F. Stanton, J. Gauss, J. D. Watts, M. Nooijen, N. Oliphant, S. A. Perera, P. G. Szalay, W. J. Lauderdale, S. A. Kucharski, S. R. Gwaltney, S. Beck, A. Balková, D. E. Bernholdt, K. K. Baeck, P. Rozyczko, H. Sekino, C. Hober, and R. J. Bartlett. *ACES II*, a software product of the Quantum Theory Project, University of Florida. Integral packages included are VMOL (J. Almlöf and P. R. Taylor); VPROPS (P. Taylor) ABACUS; (T. Helgaker, H. J. Aa. Jensen, P. Jørgensen, J. Olsen, and P. R. Taylor).

[45] M. E. Wolf and M. S. Lam. A Data Locality Algorithm. *SIGPLAN'91 Conf. on Programming Languages Design and Implementation*, pp. 30–44, June 1991.

[46] M. E. Wolf, D. E. Maydan, and D. J. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 274–286, Paris, France, December 2–4, 1996.

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