Towards Automatic Synthesis of High-Performance Codes for Electronic Structure Calculations: Data Locality Optimization

D. Cociorva[1], J. Wilkins[1], G. Baumgartner[2], P. Sadayappan[2], J. Ramanujam[3], M. Nooijen[4], D. Bernholdt[5], and R. Harrison[6]

[1] Physics Dept., Ohio State Univ., USA. {cociorva,wilkins}@pacific.mps.ohio-state.edu
[2] CIS Department, Ohio State University, USA. {gb,saday}@cis.ohio-state.edu
[3] ECE Department, Louisiana State University, USA. jxr@ee.lsu.edu
[4] Chemistry Department, Princeton University, USA. Nooijen@Princeton.edu
[5] Oak Ridge National Laboratory, USA. bernholdte@ornl.gov
[6] Pacific Northwest National Laboratory, USA. Robert.Harrison@pnl.gov

Abstract. The goal of our project is the development of a program synthesis system to facilitate the development of high-performance parallel programs for a class of computations encountered in computational chemistry and computational physics. These computations are expressible as a set of tensor contractions and arise in electronic structure calculations. This paper provides an overview of a planned synthesis system that will take as input a high-level specification of the computation and generate high-performance parallel code for a number of target architectures. We focus on an approach to performing data locality optimization in this context. Preliminary experimental results on an SGI Origin 2000 are encouraging and demonstrate that the approach is effective.

1 Introduction

The development of high-performance parallel programs for scientific applications is usually very time consuming. Often, the time to develop an efficient parallel program for a computational model is the primary limiting factor in the rate of progress of the science. Therefore, approaches to automated synthesis of high-performance parallel programs are very attractive. In general, automatic synthesis of parallel programs is not feasible. However, for specific domains, a synthesis approach is feasible, as is being demonstrated, e.g., by the SPIRAL project [35] for the domain of signal processing.

Our long term goal is to develop a program synthesis system to facilitate the development of high-performance parallel programs for a class of scientific computations encountered in computational chemistry and computational physics. The domain of our focus is electronic structure calculations, as exemplified by coupled cluster methods, where many computationally intensive components are expressible as a set of tensor contractions. We plan to develop a synthesis system that can generate efficient parallel code for a number of target architectures from an input specification expressed in a high-level notation. In this paper, we provide an overview of the planned synthesis system, and focus on an optimization approach for one of the components of the synthesis system that addresses data locality optimization.

The computational structures that we address arise in scientific application domains that are extremely compute-intensive and consume significant computer resources at national supercomputer centers. These computational forms arise in in some computational physics codes modeling electronic properties of semiconductors and metals [2, 8, 28], and in computational chemistry codes such as ACES II, GAMESS, Gaussian, NWChem, PSI, and MOLPRO. 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 [21, 23]. Computational approaches to modeling the structure and interactions of molecules, the electronic and optical properties of molecules, the heats and rates of chemical reactions, etc., are crucial to the understanding of chemical processes in real-world systems.

The paper is organized as follows. In the next section, we elaborate on the computational context of interest and the pertinent optimization issues. Sec. 3 provides an overview of the synthesis system, identifying its components. Sec. 4 focuses on data locality optimization and presents a new approach and algorithm for effective tiling in this context. Sec. 5 presents experimental performance data on the application of the new algorithm. Related work is covered in Sec. 6, and conclusions are provided in Sec. 7.

2 The Computational Context

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. Due to commutativity, associativity, and distributivity, there are many different ways to compute the final result, and they could differ widely in the number of floating point operations required. Consider the following expression:

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

If this expression is directly translated to code (with ten nested loops, for indices $a-l$), the total number of arithmetic operations required will be $4 \times N^{10}$ if the range of each index $a-l$ is N. Instead, the same expression can be rewritten by use of associative and distributive laws as the following:

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

This corresponds to the formula sequence shown in Fig. 1(a) and can be directly translated into code as shown in Fig. 1(b). This form only requires $6 \times N^6$ operations. However, additional space is required to store temporary arrays $T1$ and $T2$.

Generalizing from the above example, we can express multi-dimensional 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: **(i)** a multiplication formula of the form: $Tr(\ldots) = X(\ldots) \times Y(\ldots)$, or **(ii)** a summation formula of the form: $Tr(\ldots) = \sum_i X(\ldots)$, 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 $X(\ldots)$, $Y(\ldots)$ and $Tr(\ldots)$, respectively. For a formula to be well-formed, every index in $X(\ldots)$ and $Y(\ldots)$, except the summation index in the second form, must appear in $Tr(\ldots)$. 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.

The problem of determining an operation-optimal sequence that is equivalent to the original expression has been previously addressed by us. We have shown that the problem is NP-complete and have developed a pruning search procedure that is very efficient in practice [17, 19, 18].

An issue of great significance for this computational context is the management of memory required to store the elements of the various arrays involved. Often, some of

$$T1\ (\ b,c,d,f) = \sum_{e,l} B(b,e,f,l) \times D(c,d,e,l)$$

$$T2\ (\ b,c,j,k) = \sum_{d,f} T1(b,c,d,f) \times C(d,f,j,k)$$

$$S\ (\ a,b,i,j) = \sum_{c,k} T2(b,c,j,k) \times A(a,c,i,k)$$

(a) Formula sequence

```
read B(*,*,*,*)
read D(*,*,*,*)
T1(*,*,*,*) = 0
FOR b = 1, Nb
 FOR c = 1, Nc
  FOR d = 1, Nd
   FOR e = 1, Ne
    FOR f = 1, Nf
     FOR l = 1, Nl
      T1(b,c,d,f)
        += B(b,e,f,l) * D(c,d,e,l)
END FOR l, f, e, d, c, b
read C(*,*,*,*)
T2(*,*,*,*) = 0
FOR b = 1, Nb
 FOR c = 1, Nc
  FOR d = 1, Nd
   FOR f = 1, Nf
    FOR j = 1, Nj
     FOR k = 1, Nk
      T2(b,c,j,k)
        += T1(b,c,d,f) * C(d,f,j,k)
END FOR k, j, f, d, c, b
read A(*,*,*,*)
S(*,*,*,*) = 0
FOR a = 1, Na
 FOR b = 1, Nb
  FOR c = 1, Nc
   FOR i = 1, Ni
    FOR j = 1, Nj
     FOR k = 1, Nk
      S(a,b,i,j)
        += T2(b,c,j,k) * A(a,c,i,k)
END FOR k, j, i, c, b, a
write S(*,*,*,*)
```

(b) Direct implementation (unfused code)

Fig. 1. A sequence of formulae and the corresponding unfused code.

the input, output, and intermediate temporary arrays are too large to fit into the available physical memory. Therefore, the computation must be structured to operate on memory resident blocks of the arrays that are suitably moved between disk and main memory as needed. Similarly, effective use of cache requires that appropriate blocking or tiling of the computation is performed, whereby data reuse in cache is facilitated by operating on the arrays in blocks.

If all arrays were sufficiently small, the computation could be simply expressed as shown in Fig. 1(b). Here, all elements of the input arrays A, B, C, and D are first read in, the three sets of perfectly nested loops perform the needed computations, and the result array S is output. However, if any of the arrays is too large to fit in memory, the computation must be restructured to process the arrays in blocks or "slices." If, instead of fully creating an intermediate array (like T1) before using it, a portion of the array could be created and used before other elements of the array are created, the space required for the array could be significantly reduced. Similarly, even for input arrays, instead of reading in the entire array, blocks of the array could be read in and used before other blocks are brought in. A systematic approach to explore ways of reducing the memory requirement for the computation is to view it in terms of potential loop fusions. Loop fusion merges loop nests with common outer loops into larger imperfectly nested loops. When one loop nest produces an intermediate array that 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 application of fusion is illustrated in Fig. 2(a). In this case, most arrays can be reduced to scalars without changing the number of arithmetic operations.

For a computation consisting 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

```
FOR b = 1, Nb
 Sf(*,*,*) = 0
 FOR c = 1, Nc
  T2f(*,*) = 0
  FOR d = 1, Nd
   FOR f = 1, Nf
    T1f = 0
    FOR e = 1, Ne
     FOR l = 1, Nl
      Bf = read B(b,e,f,l)
      Df = read D(c,d,e,l)
      T1f += Bf * Df
    END FOR l, e
    FOR j = 1, Nj
     FOR k = 1, Nk
      Cf = read C(d,f,j,k)
      T2f(j,k) += T1f * Cf
  END FOR k, j, f, d
  FOR a = 1, Na
   FOR i = 1, Ni
    FOR j = 1, Nj
     FOR k = 1, Nk
      Af = read A(a,c,i,k)
      Sf(a,i,j) += T2f(j,k) * Af
 END FOR k, j, i, a, c
 write S(a,b,i,j) = Sf(a,i,j)
END FOR b
```

(a) Memory-reduced (fused) version

```
FOR b = 1, Nb
 Sf(*,*,*) = 0
 FOR cT = 1, Nc, Tc
  T2f(*,*,*) = 0
  FOR d = 1, Nd
   FOR f = 1, Nf
    T1f(*) = 0
    FOR e = 1, Ne
     FOR l = 1, Nl
      Bf = read B(b,e,f,l)
      Df(*) = read_tile D(cT,d,e,l)
      FOR cI = 1, Tc
       T1f(cI) += Bf * Df(cI)
    END FOR cI, l, e
    FOR j = 1, Nj
     FOR k = 1, Nk
      Cf = read C(d,f,j,k)
      FOR cI = 1, Tc
       T2f(cI,j,k) += T1f(cI) * Cf
  END FOR cI, k, j, f, d
  FOR a = 1, Na
   FOR i = 1, Ni
    FOR j = 1, Nj
     FOR k = 1, Nk
      Af(*) = read_tile A(a,cT,i,k)
      FOR cI = 1, Tc
       Sf(a,i,j) += T2f(cI,j,k) * Af(cI)
 END FOR cI, k, j, i, a, cT
 write S(a,b,i,j) = Sf(a,i,j)
END FOR b
```

(b) Tiling of the memory-reduced version

Fig. 2. Pseudocodes for (a) the memory-reduced (fused) solution, and (b) a tiled example of the same solution. In (b), the loop over c is split up in a tiling loop c_T, and an intra-tile loop c_I. The "tile size" is denoted by T_c. The procedure read_tile reads T_c elements from a four-dimensional array on the disk (A or D) into a one-dimensional memory array of size T_c (A_f or D_f).

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 formula sequence that minimized the total space required for all arrays after fusion [15, 16, 14].

Having provided information about the computational context and some of the optimization issues in this context, we now provide an overview of the overall synthesis framework before focusing on the specific data locality optimization problem that we address in this paper.

3 Overview of the Synthesis System

Fig. 3 shows the components of the planned synthesis system. A brief description of the components follows:

Algebraic Transformations: It takes high-level input from the user in the form of tensor expressions (essentially sum-of-products array expressions) and synthesizes an output computation sequence. The input is expressed in terms of multidimensional summations of the product of terms, where each term is either an array, an elemental function with each arguments being an expression index, or an expression involving only element-wise operations on compatible arrays. The Algebraic Transformations module uses the properties of commutativity and associativity of addition and multiplication and the distributivity of multiplication over addition. It searches for all possible ways of applying these properties to an input sum-of-products expression, and determines a combination that results in an equivalent form of the computation with minimal operation cost. The application of the distributive law to factor a term out of a summation implies the need to use a temporary array to hold an intermediate result.

Memory Minimization: The operation-minimal computation sequence synthesized by the Algebraic Transformation module might require an excessive amount of memory due to the large arrays involved. The Memory Minimization module attempts to perform

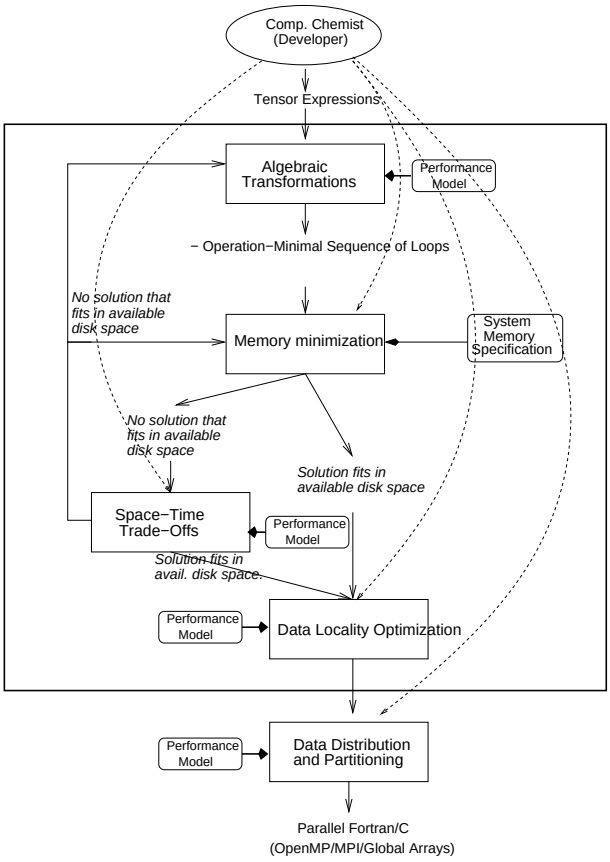


Fig. 3. The Planned Synthesis System

loop fusion transformations to reduce the memory requirements. This is done without any change to the number of arithmetic operations.

Space-Time Transformation: If the Memory Minimization module is unable to reduce memory requirements of the computation sequence below the available disk capacity on the system, the computation will be infeasible. This module seeks a trade-off that reduces memory requirements to acceptable levels while minimizing the computational penalty. If no such transformation is found, feedback is provided to the Memory Minimization module, causing it to seek a different solution. If the Space-Time Transformation module is successful in bringing down the memory requirement below the disk capacity, the Data Locality Optimization module is invoked. A framework for modeling space-time trade-offs and deriving transformations is currently under study.

Data Locality Optimization: If the space requirement exceeds physical memory capacity, portions of the arrays must be moved between disk and main memory as needed, in a way that maximizes reuse of elements in memory. The same considerations are involved in effectively minimizing cache misses — blocks of data must be moved between physical memory and the limited space available in the cache. In this paper, we focus on this step of the synthesis process. Given an imperfectly nested loop generated

by the Memory Minimization module, the Data Locality Optimization module is responsible for generating an appropriately blocked form of the loops to maximize data reuse in the different levels of the memory hierarchy.

Data Distribution and Partitioning: The final step is to determine how best to partition the arrays among the processors of a parallel system. We assume a data-parallel model, where each operation in the operation sequence is distributed across the entire parallel machine. The arrays are to be disjointly partitioned between the physical memories of the processors. This model allows us to decouple (or loosely couple) the parallelization considerations from the operation minimization and memory considerations. The output of this module will be parallel code in Fortran or C. Different target programming paradigms can be easily supported, including message-passing with MPI and paradigms with global shared-memory abstractions, such as Global Arrays and OpenMP. Even with the shared-space paradigms, in order to achieve good scalability on highly parallel computer systems, careful attention to data distribution issues is essential. Thus the underlying abstraction used in determining good data partitioning decisions remains the same, independent of the programming paradigm used for the final code.

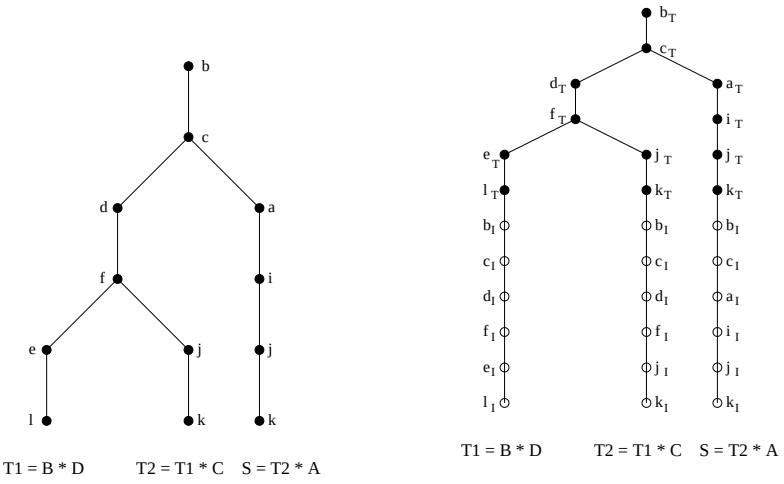
4 Data Locality Optimization Algorithm

We now address the data locality optimization problem that arises in this synthesis context. Given a memory-reduced (fused) version of the code, the goal of the algorithm is to find the appropriate blocking of the loops in order to maximize data reuse. The algorithm can be applied at different levels of the memory hierarchy, for example, to minimize data transfer between main memory and disk (I/O minimization), or to minimize data transfer between main memory and the cache (cache misses). In the rest of the paper, we focus mostly on the cache management problem. For the I/O optimization problem, the same approach could be used, replacing the cache size *CS* by the physical memory size *MS*.

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. In addition, other tiles may be sized down slightly so that the total cache capacity is not exceeded.

We introduce a memory access cost model (*Cost*), an estimate on the number of cache misses, as a function of tile sizes and loop bounds. For each loop — each node in the parse tree representation — we count the number *Accesses* of distinct array elements accessed in its scope. If this number is smaller than the number of elements that fit into the cache, then *Cost* = *Accesses*. Otherwise, it means that the elements in the cache are not reused from one loop iteration to the next, and the cost is obtained by multiplying the cost of the inner loop(s) — child node(s) in the parse tree — by the loop range.

To illustrate the cost model, we consider the fused code presented in Figure 2(a). The corresponding parse tree is shown in Figure 4(a) . For the subtree rooted at node l, the number of elements accessed in the l loop is N_l for the arrays B and D, and 1 for the array $T1$, for a total of $2N_l + 1$ accesses. The computation of the number of accesses at the next node in the parse tree e (next loop nesting level) depends on the relative size of $2N_l + 1$ with respect to the cache size *CS*. If $2N_l + 1 > CS$, there can be no reuse of the array elements from one iteration of e to the next. Therefore, the number of



(a): No tiling: A node in the tree represents a loop nest; a parent-child pair represents an outer loop (parent node), and an inner loop (child node).

(b): With tiling: Each loop (node) in (a) is split into a tiling/intra-tile pair of loops (nodes). The intra-tile loops are then moved by fission and permutation operations toward the bottom of the tree.The tiling loops are represented by black nodes in the figure, whilethe intra-tile loops are represented by empty nodes.

Fig. 4. Parse trees for the fused loop structure shown in Figure 2(a).

accesses for the subtree rooted at e is $N_e(2N_l + 1)$. However, if $2N_l + 1 < CS$, there is the possibility of data reuse in the e loop. The number of elements accessed in the e loop is $N_l * N_e$ for the arrays B and D, and 1 for the array $T1$ (the same element of $T1$ is repeatedly accessed), for a total of $2 * N_l * N_e + 1$ accesses. The new cost is again compared to *CS*, and then the next node in the parse tree is considered.

In practice, the problem has two additional aspects: the parse tree has branches (a parent node with multiple children nodes, corresponding to an outer loop with several adjacent inner loops), and each node in the tree is split into a parent-child pair, corresponding to a tiling loop node, and an intra-tile loop node. Figures 4(a) and 4(b) present the parse trees for the same computation, performed without and with loop tiling, respectively. For a given N-node untiled parse tree, the data locality optimization algorithm proceeds as follows: first, each node is split into a tiling/intra-tile pair. Subsequently, the resulting $2N$-node parse tree is transformed by loop permutation and intra-tile loop fission into an equivalent parse tree with the property that any tiling loop is exterior to any intra-tile loop (Figure 4(b)). Then, we pick starting values for the N tile sizes, thus fixing the loop ranges for the $2N$ nodes in the parse tree (if the original range of a loop is N_i, choosing a tile size T_i for the intra-tile loop also fixes the range N_i/T_i of the tiling loop).

We thus obtain a $2N$-node parse tree with well-defined loop ranges. Using a recursive top-down procedure, we compute the memory access cost of the parse tree (Figure 5). Each node is associated with a loop index *LoopIndex*, a boolean value *Exceed* that keeps track of the number of distinct accesses in the loop scope in relation to the cache size *CS*, a list of arrays *ArrayList* accessed in its scope, and a memory access cost *Cost*. The key of the algorithm is the procedure **ComputeCost**, presented in a pseudo-code format in Figure 5. **ComputeCost** (Node X) computes the memory access cost

```
Node: {
    Index LoopIndex
    boolean Exceed
    int Cost
    Node[] Children
    Array[] ArrayList
}

Array: {
    Index[] Indices
    int Accesses
}

boolean ContainIndex (Array A, Index LoopIndex) ≡
{T if LoopIndex ∈ A.Indices; F otherwise}

InsertArrayList (Node X, Node Y)
  foreach Array A ∈ Y.ArrayList
    if (A ∈ X.ArrayList) then
      X.ArrayList = X.ArrayList ∪ A
```

```
ComputeCost (Node X):
  X.ArrayList = NULL
  X.Cost = 0
  X.Exceed = F
  foreach Node Y ∈ X.Children
    ComputeCost (Y)
    InsertArrayList (X, Y)
    if (Y.Exceed) then X.Exceed = T
  if (X.Exceed) then
    foreach Node Y ∈ X.Children
      X.Cost += Y.Cost * X.Index.Range
    foreach Array A ∈ X.ArrayList
      A.Accesses = A.Accesses * X.Index.Range
  else
    int NewCost = 0
    foreach Array A ∈ X.ArrayList
      if (ContainIndex (A, X.LoopIndex)) then
        NewCost += A.Accesses * X.Index.Range
      else NewCost += A.Accesses
    if (NewCost < CacheSize) then
      X.Exceed = F
      X.Cost = NewCost
      foreach Array A ∈ X.ArrayList
        if (ContainIndex (A, X.LoopIndex)) then
          A.Accesses *= X.Index.Range
    else
      X.Exceed = T
      foreach Array A ∈ X.ArrayList
        A.Accesses *= X.Index.Range
        X.Cost += A.Accesses
```

Fig. 5. Procedure **ComputeCost** for computing the memory access cost by a top-down recursive traversal of the parse tree.

of a sub-tree rooted at X, using the cost model based on the counting of distinct array elements accessed in the scope of the loop. The procedure outlines the computation of $X.Cost$ for the general case of a branched sub-tree with any number of children nodes.

Using this cost model, we arrive at a total memory access cost for the $2N$-node parse tree for given tile sizes. The procedure is then repeated for different sets of tile sizes, and new costs computed. In the end the lowest cost is chosen, thus determining the optimal tile sizes for the parse tree. We define our tile size search space in the following way: if N_i is a loop range, we use a tile size starting from $T_i = 1$ (no tiling), and successively increasing T_i by doubling it until it reaches N_i. This ensures a slow (logarithmic) growth of the search space with increasing array dimension for large N_i. If N_i is small enough, an exhaustive search can instead be performed.

5 Experimental Results

In this section, we present results of an experimental performance evaluation of the effectiveness of the data locality optimization algorithm developed in this paper. The algorithm from Section 4 was used to tile the code shown in Figure 1(b). Measurements were made on a single processor of a Silicon Graphics Origin 2000 system consisting of 32 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:

– UNF: no explicit fusion or tiling

x	Memory requirement			Performance (MFLOPs)			Cache misses		
	TA	FUS	UNF	TA	FUS	UNF	TA	FUS	UNF
0.2	8MB	30KB	32MB	484	88	475	7.2×10^5	5.2×10^6	3.2×10^6
0.4	8MB	0.2MB	0.5GB	470	101	451	1.4×10^7	8.3×10^7	5.3×10^7
0.5	8MB	0.5MB	1.2GB	491	80	460	2.9×10^7	1.8×10^9	1.4×10^8
0.6	8MB	0.8MB	2.6GB	477	97	407	6.8×10^7	3.3×10^9	4.2×10^8
0.7	8MB	1.3MB	4.8GB	482	95	N/A	1.0×10^8	7.0×10^9	N/A
0.8	8MB	1.9MB	8.2GB	466	83	N/A	1.9×10^8	7.9×10^9	N/A
1	16MB	3.7MB	20GB	481	92	N/A	4.8×10^8	2.9×10^{10}	N/A
1.2	48MB	6.5MB	41GB	472	98	N/A	1.1×10^9	4.8×10^{10}	N/A
1.4	80MB	10MB	77GB	486	85	N/A	1.8×10^9	8.5×10^{10}	N/A
1.6	120MB	15MB	131GB	483	82	N/A	3.5×10^9	1.6×10^{11}	N/A
1.8	192MB	22MB	211GB	465	77	N/A	5.1×10^9	3.2×10^{11}	N/A
2	240MB	29MB	318GB	476	79	N/A	8.1×10^9	5.0×10^{11}	N/A

Table 1. Performance data for fusion, plus tiling algorithm (TA), compared with performance data for fused alone (FUS), and unfused loops (UNF).

- FUS: use of fusion alone, to reduce memory requirements
- TA: use of fusion, followed by tiling using algorithm presented in Section 4

We chose various array sizes for the problem to test the algorithm for a range of calculations typical in size for computational chemistry codes: $N_i = N_j = N_k = N_l = 40x$, $N_a = N_b = N_c = N_d = 300x$, and $N_e = N_f = 70x$, for x running from 0.2 to 2 (Table 1). For larger x, some arrays (e.g., $T1$) are so large that the the system's virtual memory limit 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`). The performance data was generated over multiple runs, and average values are reported. Standard deviations are typically around 10MFLOPs. The experiments were run on a time-shared system; so some interference with other processes running at the same time on the machine was inevitable, and its effects are especially pronounced for the larger x tests.

Table 1 shows the memory requirement, measured performance, and the number of secondary cache misses generated by the three alternatives. The main observations from the experiment are:

- The total memory requirement is minimized by fusion of the loops over b, c, d, and f (FUS), bringing all four-dimensional arrays (e.g., $T2$) down to at most two explicit dimensions. However, the memory requirement of fusion plus the tiling algorithm (TA) is not much higher, since the tiling loops are fused, and the arrays are reduced to much smaller "tile" sizes. The UNF version has significantly higher memory requirements since no fusion has been applied to reduce temporary memory requirements. As x is increased, the UNF version requires more memory than the per-process virtual memory limit on the system.
- The maximally fused version (FUS) has the lowest memory requirement, but incurs a severe 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 TA and UNF versions show comparable performance for smaller x. 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.

6 Related Work

Much work has been done on improving locality and parallelism through loop fusion. Kennedy and co-workers [11] have developed algorithms for modeling the degree of data sharing and for fusing a collection of loops to improve locality and parallelism. Singhai and McKinley [29] examined the effects of loop fusion on data locality and parallelism together. Although this problem is NP-hard, they were able to find optimal solutions in restricted cases and heuristic solutions for the general case. Gao et al. [6] studied the contraction of arrays into scalars through loop fusion as a means to reduce array access overhead. 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.

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. 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 [15]. 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 [22, 24, 33], 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. [34] 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 [27, 33, 30], and analytic models of the impact of tiling on locality have been developed [7, 20, 25]. Recently, a data-centric version of tiling called data shackling has been developed [12, 13] (together with more recent work by Ahmed et al. [1]) which allows a cleaner treatment of locality enhancement in imperfectly nested loops.

The approach undertaken in this project bears similarities to some projects in other domains, such as the SPIRAL project which is aimed at the design of a system to generate efficient libraries for digital signal processing algorithms [35]. SPIRAL generates efficient implementations of algorithms expressed in a domain-specific language called SPL by a systematic search through the space of possible implementations. Several factors such as the lack of a need to perform space-time trade-offs renders the task faced by efforts such as SPIRAL and FFTW [5] less complex than what computational chemists face. Other efforts in automatically generating efficient implementations of programs include the telescoping languages project [10], the ATLAS [32] project for deriving efficient implementation of BLAS routines, and the PHIPAC [3] and TUNE [31] projects.

Recently, using a very different approach, we considered the data locality optimization problem arising in this synthesis context [4]. In that work, we developed an inte-

grated approach to fusion and tiling transformations for the class of loops addressed. However, that algorithm was only applicable when the sum-of-products expression satisfied certain constraints on the relationship between the array indices in the expression. The algorithm developed in this paper does not impose any of the restrictions assumed in [4]. It takes a very different approach to effective tiling — first perform fusion to minimize memory requirements, followed by a combination of loop fission, tiling and array expansion transformations to maximize data reuse.

7 Conclusion

This paper has described a project on developing a program synthesis system to facilitate the development of high-performance parallel programs for a class of computations encountered in computational chemistry and computational physics. These computations are expressible as a set of tensor contractions and arise in electronic structure calculations. The paper has provided an overview of the planned synthesis system and has presented a new optimization approach that can serve as the basis for a key component of the system for performing data locality optimizations. Preliminary results are very encouraging and show that the approach is effective.

Acknowledgments

We would like to thank the Ohio Supercomputer Center (OSC) for the use of their computing facilities, and the National Science Foundation for partial support through grants DMR-9520319, CCR-0073800, and NSF Young Investigator Award CCR-9457768.

References

1. N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loops. *ACM Intl. Conf. on Supercomputing,* 2000.
2. W. Aulbur. *Parallel Implementation of Quasiparticle Calculations of Semiconductors and Insulators*, Ph.D. Dissertation, Ohio State University, Columbus, OH, October 1996.
3. J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC. In *Proc. ACM International Conference on Supercomputing,* pp. 340–347, 1997.
4. D. Cociorva, J. Wilkins, C.-C. Lam, G. Baumgartner, P. Sadayappan, and J. Ramanujam. Loop optimization for a class of memory-constrained computations. In *Proc. 15th ACM International Conference on Supercomputing,* pp. 500–509, Sorrento, Italy, June 2001.
5. M. Frigo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. ICASSP 98,* Volume 3, pages 1381–1384, 1998, `http://www.fftw.org.`
6. G. Gao, R. Olsen, V. Sarkar and R. Thekkath. Collective Loop Fusion for Array Contraction. *Proc. 5th LCPC Workshop* New Haven, CT, Aug. 1992.
7. S. Ghosh, M. Martonosi and S. Malik. Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity. *8th ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.
8. M. S. Hybertsen and S. G. Louie. Electronic correlation in semiconductors and insulators: band gaps and quasiparticle energies. *Phys. Rev. B*, 34:5390, 1986.
9. J. Johnson, R. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *Circuits, Systems and Signal Processing,* 9(4):449–500, 1990.
10. K. Kennedy et. al., Telescoping Languages: A Strategy for Automatic Generation of Scientific Problem-Solving Systems from Annotated Libraries. To appear in *Journal of Parallel and Distributed Computing,* 2001.
11. K. Kennedy. Fast greedy weighted fusion. *ACM Intl. Conf. on Supercomputing,* May 2000.

12. I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, June 1997.
13. 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.
14. C. Lam. *Performance Optimization of a Class of Loops Implementing Multi-Dimensional Integrals*, Ph.D. Dissertation, The Ohio State University, Columbus, OH, August 1999.
15. 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. *Proc. 12th LCPC Workshop* San Diego, CA, Aug. 1999.
16. C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. In *Proc. Intl. Conf. on High Perf. Comp.*, Dec. 1999.
17. C. Lam, P. Sadayappan, and R. Wenger. Optimal reordering and mapping of a class of nested-loops for parallel execution. In *9th LCPC Workshop*, San Jose, Aug. 1996.
18. C. Lam, P. Sadayappan and R. Wenger. On Optimizing a Class of Multi-Dimensional Loops with Reductions for Parallel Execution. *Par. Proc. Lett.*, (7) 2, pp. 157–168, 1997.
19. C. Lam, P. Sadayappan and R. Wenger. Optimization of a Class of Multi-Dimensional Integrals on Parallel Machines. *Proc. of Eighth SIAM Conf. on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 1997.
20. 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*, April 1991.
21. 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.
22. W. Li. Compiler cache optimizations for banded matrix problems. In *International Conference on Supercomputing*, Barcelona, Spain, July 1995.
23. 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.
24. K. S. McKinley, S. Carr and C.-W. Tseng. Improving Data Locality with Loop Transformations. *ACM TOPLAS*, 18(4):424–453, July 1996.
25. N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *Intl. Journal of Parallel Programming*, 26(6):641–670, June 1998.
26. G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. *ACM SIGPLAN PLDI*, June 1998.
27. G. Rivera and C.-W. Tseng. Eliminating Conflict Misses for High Performance Architectures. *Proc. of 1998 Intl. Conf. on Supercomputing*, July 1998.
28. 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.
29. 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.
30. Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. *ACM SIGPLAN PLDI*, May 1999.
31. M. Thottethodi, S. Chatterjee, and A. Lebeck. Tuning Strassen's matrix multiplication for memory hierarchies. In *Proc. Supercomputing '98*, Nov. 1998.
32. R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proc. Supercomputing '98*, Nov. 1998.
33. M. E. Wolf and M. S. Lam. A Data Locality Algorithm. *ACM SIGPLAN PLDI*, June 1991.
34. 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.
35. J. Xiong, D. Padua, and J. Johnson. SPL: A language and compiler for DSP algorithms. *ACM SIGPLAN PLDI*, June 2001.