# Efficient Synthesis of Out-of-Core Algorithms Using a Nonlinear Optimization Solver [⋆]

Sandhya Krishnan [a], Sriram Krishnamoorthy [a,∗], Gerald Baumgartner [b], Chi-Chung Lam [a], J. Ramanujam [c], P. Sadayappan [a], Venkatesh Choppella [d]

[a] *Dept. of Computer Science and Engineering, The Ohio State University, Columbus, OH 43210. USA*

[b] *Dept. of Computer Science, Louisiana State University, Baton Rouge, LA 70803. USA*

[c] *Dept. of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. USA*

[d] *Indian Institute of Information Technology and Management, Kerala Technopark, Thiruvananthapuram, Kerala 695 581. INDIA*

---

**Abstract**

We address the problem of efficient out-of-core code generation for a special class of imperfectly nested loops encoding tensor contractions arising in quantum chemistry computations. These loops operate on arrays too large to fit in physical memory. The problem involves determining optimal tiling of loops and placement of disk I/O statements. This entails a search in an explosively large parameter space. We formulate the problem as a nonlinear optimization problem and use a discrete constraint solver to generate optimized out-of-core code. The solution generated using the discrete constraint solver consistently outperforms other approaches by up to a factor of four. Measurements on sequential and parallel versions of the generated code demonstrate the effectiveness of the approach.

*Key words:* data locality optimization, out-of-core algorithms, program transformation, compiler optimization, discrete constrained search, tensor contractions

---

[⋆] Supported in part by the National Science Foundation through grants CHE-0121676, CHE-0121706, CCF-0073800 and EIA-9986052.

[∗] Corresponding Author; Address: 395, Dreese Laboratories, 2015 Neil Ave, Columbus, OH 43210-1277

*Email addresses:* `krishnas@cse.ohio-state.edu` (Sandhya Krishnan), `krishnsr@cse.ohio-state.edu` (Sriram Krishnamoorthy), `gb@csc.lsu.edu` (Gerald Baumgartner), `clam@cse.ohio-state.edu` (Chi-Chung Lam), `jxr@ece.lsu.edu` (J. Ramanujam), `saday@cse.ohio-state.edu` (P.

## 1 Introduction

Many scientific and engineering applications need to operate on data sets that are too large to fit in the physical memory of the machine. We focus on the domain of electronic structure calculations in quantum chemistry [16, 34, 38]. These calculations require contractions (generalized matrix multiplications) of multi-dimensional tensors that are often larger than available physical memory. In such situations, it is necessary to develop *out-of-core* algorithms that explicitly orchestrate the movement of blocks of data between main memory and secondary disk storage. We are developing a program synthesis tool called the Tensor Contraction Engine (TCE) [4, 3] to facilitate the development of parallel programs for this domain, by automatically transforming high-level tensor contraction expressions into efficient parallel programs.

In this paper, we address the following problem that arises in the context of the TCE system. We are given an imperfectly nested loop structure containing a collection of tensor contraction computations expressed in an "abstract" form, that is, without concern for whether the arrays can fit within available physical memory. The problem consists of generating a "concrete" form of the code by suitably tiling the loops and inserting the necessary disk I/O statements so as to minimize the total cost of disk I/O. In the case of code generation for a parallel system, the problem also involves distributing the workload among processors and inserting the required communication. The search space of possible placements of the disk I/O statements and possible combinations of tile sizes is explosively large. We formulate the problem as a non-linear optimization problem and use a general-purpose discrete constraint solver to generate optimized out-of-core code.

The paper is organized as follows: In Sec. 2, we explain the computational context for which the data locality optimization approach is developed. In Sec. 3, we review related work in the area. Sec. 4 describes the Discrete Constrained Search (DCS) solver [8, 49, 50, 51] and outlines the steps used to convert the abstract code specification into concrete code. Sec. 5 illustrates the code generation process using a representative example. Our experimental results in Sec. 6 demonstrate that the DCS-based approach to out-of-core code generation is efficient and effective.

## 2 The Computational Context

The optimization presented in this paper has been developed in the context of the Tensor Contraction Engine (TCE) [4, 12], a domain-specific compiler for ab initio quantum chemistry calculations. The TCE takes as input a high-level spec-

Sadayappan), `choppell@iiitmk.ac.in` (Venkatesh Choppella).

```
double T(V,N)

T(*,*) = 0
B(*,*) = 0
FOR i = 1, N
    FOR n = 1, V
        FOR j = 1, N
            T(n,i) += C2(n,j) * A(i,j)
END FOR j,n,i

FOR i = 1, N
    FOR n = 1, V
        FOR m = 1, V
            B(m,n) += C1(m,i) * T(n,i)
END FOR m,n,i
```

(a) Unfused code

```
double T(V,N)

T(*,*) = 0
B(*,*) = 0
FOR i, n, j
    T(n,i) += C2(n,j) * A(i,j)
END FOR j,n,i

FOR i, n, m
    B(m,n) += C1(m,i) * T(n,i)
END FOR m,n,i
```

(b) Compact notation

```
double T

B(*,*) = 0
FOR i, n
    T = 0
    FOR j
        T += C2(n,j) * A(i,j)
    END FOR j

    FOR m
        B(m,n) += C1(m,i) * T
    END FOR m
END FOR n,i
```

(c) Fused code

Fig. 1. Example of the use of loop fusion to reduce memory requirements. Loops $i$ and $n$ are fused to reduce $T$ to a scalar.

ification of a computation expressed as a set of tensor contraction expressions and transforms it into efficient parallel code. Several compile-time optimizations are incorporated into the TCE: algebraic transformations to minimize operation counts [31, 32], loop fusion to reduce memory requirements [28, 30, 29], space-time trade-off optimization [10], communication minimization [11], and data locality optimization [12, 13] of memory-to-cache traffic.

A tensor contraction expression is comprised of a collection of multi-dimensional summations of the product of several input arrays. As an example, consider the following contraction, used often in quantum chemistry calculations to transform a set of two-electron integrals from an atomic orbital (AO) basis to a molecular orbital (MO) basis:

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

This contraction is referred to as a four-index transform. Here, $A(p,q,r,s)$ is a four-dimensional input array initially stored on disk, and $B(a,b,c,d)$ is the transformed output array to be placed on disk at the end of the computation. The arrays $C1$ through $C4$ are called the transformation matrices. In practice, these four arrays are identical; we identify them by different names in order to be able to distinguish them in the text.

The indices $p$, $q$, $r$, and $s$ have the same range $N$, denoting the total number of orbitals, which is equal to $O + V$. $O$ denotes the number of occupied orbitals and $V$ denotes the number of unoccupied (virtual) orbitals. Likewise, the index ranges for $a$, $b$, $c$, and $d$ are the same, and equal to $V$. Typical values for $O$ range from 10 to 300; the number of virtual orbitals $V$ is usually between 50 and 1000.

The calculation of $B$ is done in the following four steps to reduce the number of floating point operations from $O(V^4N^4)$ in the initial formula (8 nested loops, for $p$, $q$, $r$, $s$, $a$, $b$, $c$, and $d$) to $O(VN^4)$:

$$B(a,b,c,d) = \sum_s C1(s,d) \times \left(\sum_r C2(r,c) \times \left(\sum_q C3(q,b) \times \left(\sum_p C4(p,a) \times A(p,q,r,s)\right)\right)\right)$$

This operation-minimization transformation results in the creation of three intermediate arrays:

$$\begin{aligned} T1(a,q,r,s) &= \sum_p C4(p,a) \times A(p,q,r,s) \\ T2(a,b,r,s) &= \sum_q C3(q,b) \times T1(a,q,r,s) \\ T3(a,b,c,s) &= \sum_r C2(r,c) \times T2(a,b,r,s) \end{aligned}$$

Assuming that the available memory less than $V^4$ (which for $V = 800$ and double precision arrays is about $3TB$), none of $A$, $T1$, $T2$, $T3$, and $B$ can fit entirely in memory. Therefore, the intermediates $T1$, $T2$, and $T3$ need to be written to disk on production, and read from disk before consumption in the next step. Since none of these arrays can be fully stored in memory, it may not be possible to perform all multiplication operations by reading each element of the input arrays from the disk only once. This could result in the amount of disk I/O volume being much larger than the total volume of the data on disk.

For illustration purposes, we focus on the following contraction (a two-index transform):

$$B(m,n) = \sum_{i,j} C1(m,i) \times C2(n,j) \times A(i,j)$$

The operation minimal form of the two-index transform and the corresponding intermediate array are as follows:

$$\begin{aligned} B(m,n) &= \sum_i C1(m,i) \times (\sum_j C2(n,j) \times A(i,j)) \\ T(n,i) &= \sum_j C2(n,j) \times A(i,j) \end{aligned}$$

Fig. 1 shows the computation of the array $B$ and illustrates how memory requirements for the computation of $B$ may be reduced using loop fusion. Such an “abstract form" of the computation cannot be directly compiled and executed because it does not take into account the available physical memory (if the arrays fit within the virtual memory, execution of the compiled code is possible, but will exhibit extremely poor performance due to excessive overhead of paging to move virtual memory pages between disk and physical memory). The transformation of abstract forms into concrete forms that can be efficiently executed is addressed in Sec. 4. Concrete forms have explicit disk I/O statements to move data between disk-resident arrays and their in-memory counterparts. Fig. 1(a) shows the abstract form of the computation before loop fusion. The computation consists of two loop nests: a first loop that produces the intermediate $T(1:V,1:N)$, and a second loop that uses $T$ to produce the result $B(1:V,1:V)$.

In Fig. 1(b), each loop nest is abbreviated as a single "For" loop with a sequence of indices. Fig. 1(c) illustrates the result of loop fusion. Note that all loops in each of the two loop nests in Fig. 1(a) are fully permutable and there are no fusion-preventing dependences between the loops. Hence, the common loops $i$ and $n$, shown underlined, can be fused. After loop fusion, the storage requirements for $T$ can be reduced because there is no longer a need for an explicit dimension of $T$ corresponding to any loop indices that are fused between the producer of $T$ and the consumer of $T$ — storage elements can be reused over sequential iterations of fused loops. In this example, $T$ can be contracted to a scalar as shown in Fig. 1(c). Although the total number of arithmetic operations remains unchanged, the significant reduction in size of the intermediate array $T$ implies that it may be completely stored in memory, without the need for any disk I/O for it. In contrast, if $V \times N$ is larger than the available memory, the unfused version would result in $T$ being written out to disk after it is produced in the first loop, and then read in from disk for the second loop.

Given a fused *abstract* form of the computation, in the form of an imperfectly nested loop structure (i.e., a nested loop structure in which at least one loop other than the inner-most contains more than one statement, as in Fig. 1(c)) the out-of-core code generation process requires consideration of a number of issues. Each loop in the imperfectly nested loop structure is split into a tiling and an intra-tile loop. Given a tiled loop structure, there are a number of different candidate positions for placing disk I/O statements. Thus, a search space consisting of two dimensions, placement of disk read/write statements and the tile sizes, needs to be explored. A disk I/O statement transfers blocks of data between the disk resident array and its in-memory counterpart. The size of the in-memory buffer is a function of the tile sizes and placement of the corresponding disk I/O statement. The task of the out-of-core code generation algorithm is to tile the loops, determine the optimal placements of disk I/O statements, and tile sizes that minimize the disk I/O cost while satisfying the memory limit constraints.

## 3 Related Work

We have addressed various issues arising in the synthesis context described above focusing primarily on minimizing memory-to-cache data movement [13, 12]. In Cociorva et al. [13], we developed an integrated approach to fusion and tiling transformations for a restricted class of loops arising in the context of our program synthesis system; these assumed restrictions were subsequently removed [12]. We developed a tile size search procedure to estimate the total capacity miss cost for each of a large number of combinations of tile sizes for the various loops of an imperfectly nested loop set. After finding the best combination of tile sizes, we made adjustments to address spatial locality considerations — by adjusting the tile sizes for any loop indexing the fastest varying dimension of any array to be larger than the cache line size.

Krishnan [26] extended this approach to the disk-memory hierarchy using a greedy approach to disk read/write placement. For each set of tile sizes, Krishnan's algorithm places read/write statements immediately inside those loops at which the memory limit is just exceeded. In Krishnan et al. [27], we describe an algorithm to determine effective tile sizes. This algorithm explores the tile size search space using the set of candidate fusion structures with disk I/O placements as input. The search space was divided into feasible and infeasible solution spaces and their boundary was shown to contain the optimal solution. We developed an algorithm to locate the boundary efficiently and used steepest ascent hill-climbing to determine an efficient solution for the tile sizes.

There has been some work in the area of software techniques for optimizing disk I/O. These include parallel file systems, compile time [5, 6, 7, 20, 21, 22, 41, 44] and runtime libraries and optimizations [47, 9]. Several compiler techniques for optimizing out-of-core programs in High Performance Fortran are discussed in [5, 6]. Bordawekar et al. [7] develop a scheduling strategy to eliminate additional I/O arising from communication among processors. Solutions to choreographing disk I/O with computation are presented by Paleczny et al. [44]; they organize computations into groups that operate efficiently on data accessed in chunks. Compiler-directed pre-fetching is discussed by Mowry et al. [41], which is orthogonal to compiler transformations discussed in this paper. ViC* (Virtual C*) [15] is a preprocessor that transforms out-of-core C* programs into in-core programs with appropriate calls to the ViC* I/O library. Kandemir et al. [20, 21, 22] developed file layout and loop transformations for reducing I/O. None of these techniques address model-driven automatic tile size selection for optimizing I/O and all of them deal only with perfectly nested loops.

Considerable research on loop transformations for locality in nested loops has been reported in the literature [14, 35, 36, 39, 52]. Nevertheless, a performance-model driven approach to the integrated use of loop fusion and loop tiling for enhancing locality in imperfectly nested loops has not been addressed in these works. Loop tiling for enhancing data locality has been studied extensively [2, 14, 23, 24, 45, 46, 52, 53], and analytical models of the impact of tiling on locality in perfectly nested loops have been developed [19, 33, 40]. Mitchell et al. [40] provide analytical models for multi-level tiling of matrix-matrix multiplication. Ahmed et al. [1] have developed a framework that embeds an arbitrary collection of loops into an equivalent perfectly nested loop that can be tiled; this allows a cleaner treatment of imperfectly nested loops. Lim et al. [37] developed a framework based on affine partitioning and blocking to reduce synchronization and improve data locality. Specific issues of locality enhancement, I/O optimization and automatic tile size selection have not been addressed in the works that can handle imperfectly nested loops [1, 37, 46].

## 4 Proposed Approach

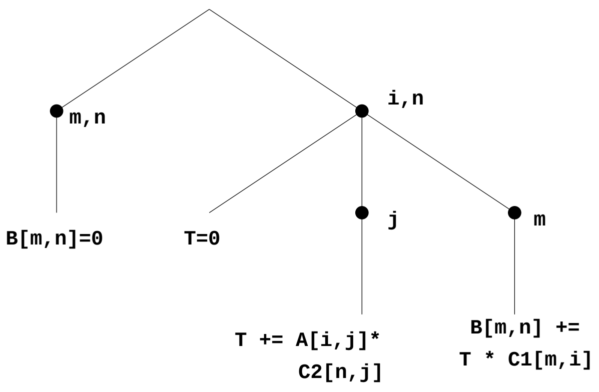
We use the Discrete Constrained Search solver to compute the best placement of disk I/O statements that would minimize the disk access cost while satisfying the memory limit constraints. Discrete Constrained Search (DCS) [8, 49, 50, 51] is a software package for determining the constrained global minima (CGM) in the discrete variable space of a single-objective, discrete, constrained non-linear programming problem (NLP). A web interface to the DCS solver is available [48]. It uses AMPL, *A Modeling Language for Mathematical Programming* [18], as the problem input format. Due to the limitations in AMPL in modeling arbitrary discrete variables, their current implementation can only solve problems with continuous variables by discretizing them.

The out-of-core code generation process translates the abstract code into concrete code by loop tiling and placement of disk I/O statements. We fully explore the search space of disk I/O placements and tile sizes by formulating the search as a non-linear constrained minimization problem where the objective function is the disk I/O cost. The solution to be determined is constrained by the memory limit and minimum I/O block size for efficient disk I/O. We input the formulated non-linear problem to the DCS system, which determines the optimal combination of placement of disk I/O statements and tile sizes.

We continue with the two-index transform example introduced in Sec. 2 for transforming atomic orbitals into molecular orbitals. Fig. 2(a) shows an abstract code for the two-index transform. We assume that the arrays involved are too large to fit into the physical memory of the machine. The arrays involved in the loop structure fall into the following three categories: input arrays that initially reside on disk ($A$, $C1$ and $C2$), intermediate arrays produced and consumed within the computation and not required on completion ($T$), and output arrays that must finally be written to disk ($B$). Fig. 2(b) shows the parse tree corresponding to the abstract code

```
FOR m, n
  B[m,n] = 0
FOR i, n
  T = 0
  FOR j
    T += A[i,j] * C2[n,j]
  FOR m
    B[m,n] += T * C1[m,i]
```

(a) Abstract code for the 2-index transform

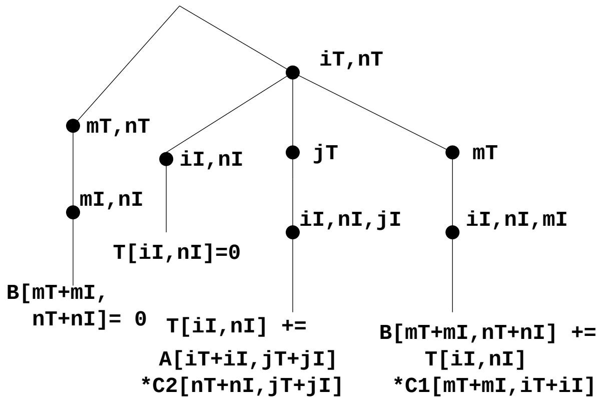


(b) Parse tree for the 2-index transform

Fig. 2. Example of abstract code and corresponding parse tree for 2-index transform.

```
FOR mT, nT
  FOR mI, nI
    B[mT+mI,nT+nI] = 0
FOR iT, nT
  FOR iI, nI
    T[iI,nI] = 0
  FOR jT, iI, nI, jI
    T[iI,nI] += A[iT+iI,jT+jI]
                * C2[nT+nI,jT+jI]
  FOR mT, iI, nI, mI
    B[mT+mI,nT+nI] += T[iI,nI]
                * C1[mT+mI,iT+iI]
```

(a) Abstract code for the tiled 2-index transform



iT,nT
mT,nT
iI,nI
jT
mT
mI,nI
iI,nI,jI
iI,nI,mI
T[iI,nI]=0
B[mT+mI, nT+nI]= 0
T[iI,nI] += A[iT+iI,jT+jI] *C2[nT+nI,jT+jI]
B[mT+mI,nT+nI] += T[iI,nI] *C1[mT+mI,iT+iI]

(b) Parse tree for the tiled 2-index transform

Fig. 3. Example of abstract code and corresponding parse tree for the tiled version of the 2-index transform.

in Fig. 2(a). To simplify the tree representation, each sequence of perfectly nested loops is represented by a single node labeled with the corresponding sequence of loop indices.

The input to the out-of-core code generation algorithm consists of the abstract code, the loop ranges and the memory limit of the machine. The algorithm consists of the following three steps:

(1) **Loop Tiling:** We split each loop into a tiling loop and an intra tile loop and propagate the intra tile loops down to the leaves. For example, as shown in Fig. 3, loop $i$ is split into tiling loop $iT$ and intra-tile loop $iI$. Fig. 3(b) shows the parse tree for the tiled abstract code in Fig. 3(a).
(2) **Candidate Placements:** For each array, we enumerate all feasible placements of disk read/write statements. Any placement surrounded by a loop index that is not involved in the I/O statement is ignored, as this I/O statement can be moved out of the loop reducing the I/O cost.
(3) **DCS Input Construction:** Given the enumeration from step $2$, we construct

```
Input Arrays: (Read Placements)
A:  iI, nT
C2: iI, jT
C1: iI, nT

Output Arrays: (Write Placements)
B:
  Write Placement: iI,  mT
  Read Required  : Yes, Yes

Intermediates: (Write & Read Placements)
T: In Memory
```

(a) Candidate I/O placements

```
FOR mT, nT
  FOR mI, nI
    B[mI,nI] = 0
  Write BDisk[mT:mT+Tm-1,nT:nT+Tn-1]
FOR iT, nT
  FOR iI, nI
    T[iI,nI] = 0
  FOR jT
    C2[1:Tn,1:Tj] =
      Read C2Disk[nT:nT+Tn-1,jT:jT+Tn-1]
    A[1:Ti,1:Tj] =
      Read ADisk[iT:iT+Ti-1,jT:jT+Tj-1]
    FOR iI, nI, jI
      T[iI,nI] += C2[nI,jI] * A[iI,jI]
  FOR mT
    B[1:Tm,1:Tn] =
      Read BDisk[mT:mT+Tm-1,nT:nT+Tn-1]
    C1[1:Tm,1:Tj] =
      Read C1Disk[mT:mT+Tm-1,jT:jT+Tj-1]
    FOR iI, nI, mI
      B[mI,nI] += T[iI,nI] * C1[mI,iI]
    Write BDisk[mT:mT+Tm-1,nT:nT+Tn-1]
```

(b) Final concrete code for 2-index transform

Fig. 4. Candidate I/O placements and final concrete code. $N_m = N_n = 35000$, $N_i = N_j = 40000$, memory limit = 1GB, double precision arrays.

non-linear equations for the objective function and constraints and provide them as input to the DCS solver. The DCS solver outputs the disk read/write placement for each array and the tile sizes that minimize the disk I/O cost and satisfy the memory limit constraint.

## *4.1 Candidate Placements*

Given a tiled imperfectly nested loop structure (Fig. 3), we consider various possible placements of reads for input arrays, reads and writes for intermediates, and writes (and reads, if required) for output arrays. In enumerating the candidate placements, there are some constraints that must be satisfied.

(1) **Input Array Constraints:** The read statement for an input array may only be placed to be executed before the statement where it is consumed. For example, in Fig. 3(a), the read for input array $A$ can be placed anywhere before line $7$.
(2) **Output Array Constraints:** The write for an output array may only be placed after the statement where it is produced. For example, in Fig. 3(a), the write for output array $B$ can be placed anywhere after line $9$.
(3) **Intermediate Array Constraints:** For intermediate arrays, we have two cases to consider: the array is either kept in memory or written to disk. If the array is kept in memory, there will be no disk I/O statements inserted for the array. On the other hand, if it is written to disk, there is a constraint imposed on its disk read/write placement. For example, in Fig. 3(a), intermediate $T$ is produced in

statement $7$ and consumed in statement $9$. If we consider these statements in the parse tree in Fig. 3(b), the lowest common ancestor for both the statements is loop $nT$. The write statement for the production and read statement for the consumption must be inside this $nT$ loop.

The approach to enumerating the placements for input, output and intermediate arrays is sketched below; details may be found in [26].

(1) **Input Arrays:** Each loop index surrounding the consumption of an input array is considered as a candidate position for placing the read. At any candidate position, if there exists a redundant loop immediately surrounding it, then we ignore that position and move further up. A redundant loop for a read statement is one that does not index the array being read. We also ignore those read placements that cause the in-memory version of the input array to be a scalar or a vector. This is because the resulting concrete code will involve in-memory matrix-matrix products using level-3 BLAS kernels [17], and scalar and vector operands will result in poor performance. Consider the abstract tiled code in Fig. 3. All loops surrounding statement 7 are candidate positions for placing the read for array $A$. Loops $jI$ and $nI$ are ignored so that the in-memory version of array $A$ is at least two-dimensional. Loop $jT$ is not considered because the surrounding loop $nT$ is redundant for array $A$. Another important check that needs to be made is that the in-memory version of the array fits in memory. For every candidate position, we compute the memory cost of the corresponding *local buffer* assuming a tile size of one. If the buffer does not fit in memory, we do not move further up.

(2) **Output Arrays:** The algorithm for enumerating write placements for an output array is exactly the same as that for input arrays, except that if any redundant loop surrounds the write statement, we need to insert a corresponding read for the array before the production. This is required as we will be re-accessing the disk array for every iteration of the redundant loop. For example, consider statement $9$ in Fig. 3(a), where the output array $B$ is produced. If the write for array $B$ is placed just after loop $mT$, an extra read will be required as the write will be surrounded by the redundant loop $iT$.

(3) **Intermediate Arrays:** If an intermediate array is written to disk, the algorithm for enumerating the disk read/write statements is exactly the same as for input/output arrays, except that the constraint specified earlier for intermediate arrays must be satisfied.

Fig. 4(a) shows the candidate read and write placements computed for each array in the code shown in Fig. 3(a). Fig. 4(b) shows the final concrete code for the two-index transform using the candidate read and write placements shown in Fig. 3(a). Note that in Fig. 4(b), for a loop index $x$, the index of the tiling loop is denoted $xT$ and the index of the intra-tile loop is denoted $xI$; in addition, the tile size for this loop index is denoted $Tx$.

## *4.2 DCS Input Construction*

If all possible combinations of disk I/O placements, shown in Fig. 4(a), are considered for all the arrays, a very large number of cases will have to be evaluated. Our approach to avoid explicit evaluation for each combination of I/O placements is to encode the placement into the formulation of a nonlinear optimization problem that is input to the DCS system, as explained below. DCS attempts to minimize an objective function subject to equality and inequality constraints. The input to DCS consists of *input parameters*, *variables*, *objective function*, and a set of *constraints*.

### *Input Parameters*

The input parameters for our problem are the memory limit of the machine and the ranges $Ni, Nj, \ldots$ of the loop indices $i, j, \ldots$.

### *Variables*

The variables in our case include tile sizes $Ti, Tj, \ldots$ for loops $i, j, \ldots$ where each tile size variable has a lower bound of $1$ and an upper bound of the full loop range. In addition to tile size variables, placement variables, $\lambda_i, i = 0, 1, 2, \ldots$, are introduced for each arrays that has more that one candidate placement. The placement variables corresponding to an array encode all the possible placements for the disk I/O placement for the array. The values chosen for these variables in the solution from the solver uniquely determine the disk I/O placement for that array.

### *Objective Function*

The objective function for our problem is the disk I/O cost. The disk I/O cost for an I/O statement is the product of the size of the array being read/written and the ranges of any redundant loops surrounding the statement. Consider the two possible read placements for input array $A$ shown in Fig. 4(a). For the first read placement above loop $iI$, the disk I/O cost will be:

$$D1_A = (Nn/Tn) \times \textit{Size}_A$$

where the total size of array $A$ is multiplied by the range of the redundant loop $nT$. The disk I/O cost for the second read placement (above loop $nT$), is $D2_A = \textit{Size}_A$. Since there are two possible placements for $A$, $\lceil log_2(2) \rceil = 1$ placement variable $\lambda_0$ is introduced as follows to express the disk I/O cost:

$$(\lambda_0 \times D1_A) + ((1 - \lambda_0) \times D2_A)$$

If $\lambda_0 = 1$, the first placement is selected, else if $\lambda_0 = 0$, the second one is selected. As explained later, the placement encoding variables are constrained to have a value of $0$ or $1$.

*Constraints*

The total space utilized for in-memory buffers is constrained to be within the memory limit. A static memory cost model is used, in which all the in-memory buffers are allocated memory at compile time. The total memory cost is the sum of the memory usage for all the individual in-memory buffers. The memory cost for an in-memory buffer is the product of the ranges of its indices. The memory cost expression for array $A$ can be constructed, along the same lines as the disk cost expression, as follows. For the read placement above loop $iI$, the in-memory buffer for input array $A$ will be $A[iI, jI]$, which makes the memory cost $M1_A = Ti \times Tj$. On the other hand, for the read placement above loop $nT$, the in-memory buffer is $A[iI, j]$, thus making the memory cost $M2_A = Ti \times Nj$. The memory limit constraint using placement variable $\lambda_0$ is:

$$(\lambda_0 \times M1_A) + ((1 - \lambda_0) \times M2_A) \leq \text{MemoryLimit}$$

The placement variables are constrained to take values $0$ or $1$ as follows:

$$\lambda_i \times (1 - \lambda_i) = 0, i = 0, 1, 2 \ldots$$

We also introduce constraints on the minimum size of the in-memory version of an array. The arrays are stored in a blocked fashion on disk. The block sizes of the arrays are equal to the size of their in-memory versions, determined by the out-of-core code generation algorithm. A block is the basic unit of I/O and is chosen to be large enough to make the disk seek time negligible compared to the block transfer time. Krishnamoorthy et al. [25] observed that the incremental improvement obtained in the ratio of transfer time to seek time became negligible, and approach the performance of sequential I/O, beyond a certain block size. The in-memory version of the array, and hence the block size, is constrained to be larger than this block size. For the system on which the experiments were conducted, and whose configuration is described in Sec. 6, the block size for reads must be at least $2$MB, while that for writes must be at least $1$MB.

In this manner, we can construct disk cost, memory cost and other constraint expressions for all arrays. Using these expressions, we build the input to DCS using the AMPL format [18]. DCS minimizes the objective function, that is, the disk I/O cost expression, while satisfying the memory limit, boundary, placement variable and buffer size constraints. DCS outputs values for the placement variables and tile sizes, thus providing the parameters for the concrete code.

The code generated for a multi-processor system uses the Global Arrays (GA) and

Disk Resident Arrays (DRA) libraries [43, 42]. GA provides a shared-memory programming model while encouraging locality of access. DRA extends the shared-memory model to secondary storage. GA/DRA provide an array abstraction in which the portion of data to be accessed is specified as a section of the array. In the generated code, the reads and writes from the disk are performed by the read and write routines in DRA. The in-memory computation is performed using kernel matrix multiplication libraries in GA. The I/O operations and the in-memory computations are collective operations.

## 5 Illustration

In this section, we illustrate the process of code generation using the 4-index transform. Consider the abstract code for the 4-index transform in Fig. 5. The 4-index transform involves four contractions, requiring three intermediate arrays. These are T1, T2 and T3 in the abstract code shown. Loops are fused so that two of the intermediates are significantly reduced in size, leaving only T1 to be a four-dimensional array. This fused abstract code was tiled and the intra-tile loops were moved to be innermost in the loop structure. The tiled abstract code for the 4-index transform is shown in Fig. 6.

Then, the possible placements for disk I/O statements are enumerated for all the arrays. The enumeration starts at the first tiling loop or the first redundant intra-tile loop and proceeds up the fusion graph. The list of candidate I/O placements is shown in Table 1. An I/O placement for an array denoted by a loop index specifies that the disk read(write) for that array is inserted above(below) the loop corresponding to that index, surrounding the use of the array. The loops surrounding the update of an array are considered for write placements, and those surrounding the read-only use of an array are considered for read placements. The read and write placements noted correspond to these loops. The read(write) placements are shown for the input(output) arrays. For intermediate arrays, the I/O cost for production and consumption of the array are enumerated. This is shown by the cross-product of the write and read placements. The input and output arrays are disk-resident. The intermediate arrays can potentially be in memory, which is also enumerated as a possible I/O placement. The placement variables allocated to each array are also shown in Table 1. Consider the I/O placements enumerated for array T2. Array T2 could be in memory, represented by the first "placement" possibility. If T2 is disk-resident, there are three possible write and read placements, forming nine placement pairs. Each placement pair uniquely determines the read and write placement for that array. The placement pair qT $\times$ cT results in the read and write at the same node in the fusion tree, and is equivalent to T2 residing in memory. Hence it is discarded, leaving eight placements pairs, as shown.

The I/O and memory costs are the sum total of I/O and memory costs for all the arrays. Each possible placement of I/O statement for an array has a potentially different contribution to the I/O and memory cost. The contributions to the I/O and memory cost by the array T1 for some of the possible placements are shown in Table 2. Every element in a disk-resident array needs to be accessed from disk at least once. The I/O cost shown is the factor of redundant I/O over the minimum access of Size$_{T1}$*8, where Size$_{T1}$ is the size of array T1 in the tiled code, and the size of each element is 8. When the placement variables have value 11111, T1 is in memory and has no I/O cost. T1 has 25 possible placements and any values of placement variables that are beyond 00110 do not correspond to any legal I/O placement. These values for the placement variables are pruned away by specifying the memory cost to be higher than the available memory. The table also shows that the read and write costs can potentially be different, as in the case of placement variables being equal to 01010. When determining the overall I/O costs, the read and write costs can be weighted by the average read and write times. Sequential access (read/write) time was found to be a good approximation of the actual access time, once the block size is larger than a threshold.

The tile sizes are limited to the valid range by the constraints

$$1 \leq Ta, Tb, Tc, Td \leq 190$$
$$1 \leq Tp, Tq, Tr, Ts \leq 180$$

The constraints on the placement variables to limit their values to either $0$ or $1$ is given by

$$\forall\, i \in \{0, \ldots, 26\} \lambda_i * (1 - \lambda_i) = 0.$$

The I/O sizes are constrained to be large enough for efficient I/O on the target system. The I/O sizes are just the size of the in-memory buffers and hence can be computed from the memory costs. The read and write constraints, respectively, for the array T1 are given by

$$8 * \sum_{i,j,k,l,m=0}^{1} \mathrm{Placement}(i,j,k,l,m) * \mathrm{MemCost}(i,j,k,l,m) \geq \mathrm{readbufsize}$$
$$8 * \sum_{i,j,k,l,m=0}^{1} \mathrm{Placement}(i,j,k,l,m) * \mathrm{MemCost}(i,j,k,l,m) \geq \mathrm{writebufsize}$$

where

$$\mathrm{Placement}(i,j,k,l,m) = \begin{cases} 1 \text{ if } (i = \lambda_0 \wedge j = \lambda_1 \wedge k = \lambda_2 \wedge l = \lambda_3 \wedge m = \lambda_4) \\ 0 \qquad\qquad\qquad\qquad \text{otherwise} \end{cases}$$

The parameters to complete the construction of the optimization problem are shown in Table 4. We determined the parameters for the Itanium-2 cluster at the Ohio

```
for a, q, r, s
  T1[a,q,r,s] = 0
for a, p, q, r, s
  T1[a,q,r,s] += C4[p,a] * A[p,q,r,s]
for a, b, c, d
  B[a,b,c,d] = 0
for a, b
  for c, s
    T3[c,s] = 0
  for r, s
    T2 = 0
    for q
      T2 += C3[q,b] * T1[a,q,r,s]
    for c
      T3[c,s] += C2[r,c] * T2
  for c, d, s
    B[a,b,c,d] += C1[s,d] * T3[c,s]
```

Fig. 5. Abstract code for the 4-index transform example.

```
for aT, qT, rT, sT, aI, qI, rI, sI
  T1[aT+aI,qT+qI,rT+rI,sT+sI] = 0
for aT, pT, qT, rT, sT, aI, pI, qI, rI, sI
  T1[aT+aI,qT+qI,rT+rI,sT+sI] += C4[pT+pI,aT+aI] *
                                  A[pT+pI,qT+qI,rT+rI,sT+sI]
for aT, bT, cT, dT, aI, bI, cI, dI
  B[aT+aI,bT+bI,cT+cI,dT+dI] = 0
for aT, bT
  for cT, sT, aI, bI, cI, sI
    T3[cT+cI,sT+sI,aI,bI] = 0
  for rT, sT
    for aI, bI, rI, sI
      T2[aI,bI,rI,sI] = 0
    for qT, aI, bI, rI, sI, qI
      T2[aI,bI,rI,sI] += C3[qT+qI,bT+bI] *
                          T1[aT+aI,qT+qI,rT+rI,sT+sI]
    for cT, aI, bI, rI, sI, cI
      T3[cT+cI,sT+sI,aI,bI] += C2[rT+rI,cT+cI] *
                                T2[aI,bI,rI,sI]
  for cT, dT, sT, aI, bI, cI, dI, sI
    B[aT+aI,bT+bI,cT+cI,dT+dI] += C1[sT+sI,dT+dI] * T3[cT+cI,sT+sI,aI,bI]
```

Fig. 6. Abstract code for the tiled 4-index transform.

Supercomputer Center, discussed in Sec. 6. The parameters correspond to a single-processor execution using local disks. The loop bounds Na,Nb,KC, and Nd are set to 190 and Np,Nq,NB, and Ns are set to 180. The array sizes (Size$_A$, Size$_{T1}$, ..) are also specified.

The optimization problem thus constructed is solved using the non-linear optimization solver. The result is interpreted to obtain the concrete code shown in Fig. 7. The tile sizes determined are shown in Table 3.

| Array | Possible placements | Placement Variables |
|---|---|---|
| T1 | In Memory + $\{$pI,aI,sT,rT,pT$\} \times \{$bI,aI,qT,sT,bT$\}$ | $\lambda_0 — \lambda_4$ |
| C4 | $\{$qT, pT, aT$\}$ | $\lambda_5 — \lambda_6$ |
| A | $\{$aI, sT, rT, qT$\}$ | $\lambda_7 — \lambda_8$ |
| T2 | In Memory + $\{$rI,bI$\} \times \{$rI,bI,cT$\}$ + $\{$qT$\} \times \{$rI,bI$\}$ | $\lambda_9 — \lambda_{12}$ |
| C3 | $\{$aI,rT,aT$\}$ | $\lambda_{13} — \lambda_{14}$ |
| T3 | In Memory + $\{$rI,bI,aI,cI$\} \times \{$cI,bI,aI,dT,cT$\}$ + $\{$rT$\} \times \{$cI,bI,aI,dT$\}$ | $\lambda_{15} — \lambda_{19}$ |
| C2 | $\{$aI,sT,aT$\}$ | $\lambda_{20} — \lambda_{21}$ |
| B | $\{$cI,bI,sT,dT,cT,bT$\}$ | $\lambda_{22} — \lambda_{24}$ |
| C1 | $\{$aI,sT,aT$\}$ | $\lambda_{25} — \lambda_{26}$ |

Table 1
Candidate disk I/O placements and placement variables for the arrays in the 4-index transform.

| Placement variables | | | | | I/O placement | | I/O cost (*Size$_{T1}$*8) | | Memory cost |
|---|---|---|---|---|---|---|---|---|---|
| $\lambda_0$ | $\lambda_1$ | $\lambda_2$ | $\lambda_3$ | $\lambda_4$ | Produce | Consume | Read | Write | |
| 1 | 1 | 1 | 1 | 1 | In Mem | In Mem | 0 | 0 | Na*Nq*Nr*Ns*8 |
| 1 | 1 | 1 | 1 | 0 | pI | bI | Np/Tp | Np/Tp | Tq*Tr*Ts*8 |
| 1 | 1 | 1 | 0 | 1 | pI | aI | Np/Tp | Np/Tp | Ta*Tq*Tr*Ts*8 |
| $\vdots$ | | | | | | | | | |
| 0 | 1 | 0 | 1 | 0 | pT | bI | 1 | Nb/Tb | Ta*Nq*Nr*Ns*8 |
| $\vdots$ | | | | | | | | | |
| 0 | 0 | 1 | 0 | 1 | - | - | - | - | 2*MemSize |
| $\vdots$ | | | | | | | | | |
| 0 | 0 | 0 | 0 | 0 | - | - | - | - | 2*MemSize |

Table 2
Contributions to disk I/O cost and memory cost by the array T1 in the 4-index transform.

| Ta | Tb | Tc | Td | Tp | Tq | Tr | Ts |
|---|---|---|---|---|---|---|---|
| 48 | 95 | 95 | 190 | 90 | 60 | 180 | 180 |

Table 3
Tile sizes for the 4-index transform example.

| Mem. limit | Min. read size | Min. write size | Read time | Write time |
|---|---|---|---|---|
| 2GB | 2MB | 1MB | 16ns/byte | 20ns/byte |

Table 4
Parameters used in the construction of the optimization problem for the 4-index transform example.

```
for aT, qT, rT, sT
  for aI, qI, rI, sI
    T1[aI,qI,rI,sI] = 0
  Write T1Disk[aT:aT+47,qT:qT+59,rT:rT+179,sT:sT+179]
for aT, pT
  C4[1:90,1:48] = Read C4Disk[pT:pT+89,aT:aT+47]
  for qT, rT
    A[1:90,1:60,1:180,1:Ns] = Read ADisk[pT:pT+89,qT:qT+59,rT:rT+179,1:Ns]
    for sT
      T1[1:48,1:60,1:180,1:180] =
                Read T1Disk[aT:aT+47,qT:qT+59,rT:rT+179,sT:sT+179]
      for aI, pI, qI, rI, sI
        T1[aI,qI,rI,sI] += C4[pI,aI] * A[pI,qI,rI,sT+sI]
      Write T1Disk[aT:aT+47,qT:qT+59,rT:rT+179,sT:sT+179]
C1[1:Ns,1:Nd] = Read C1Disk[1:Ns,1:Nd]
for aT, bT
  for cT, sT, aI, bI, cI, sI
    T3[cT+cI,sT+sI,aI,bI] = 0
  C3[1:Nq,1:95] = Read C3Disk[1:Nq,bT:bT+94]
  for rT
    C2[1:180,1:Nc] = Read C2Disk[rT:rT+179,1:Nc]
    for sT
      for aI, bI, rI, sI
        T2[aI,bI,rI,sI] = 0
      for qT
        T1[1:48,1:60,1:180,1:180] =
                Read T1Disk[aT:aT+47,qT:qT+59,rT:rT+179,sT:sT+179]
        for aI, bI, rI, sI, qI
          T2[aI,bI,rI,sI] += C3[qT+qI,bI] * T1[aI,qI,rI,sI]
        for cT, aI, bI, rI, sI, cI
          T3[cT+cI,sT+sI,aI,bI] += C2[rI,cT+cI] * T2[aI,bI,rI,sI]
  for cT, dT
    for aI, bI, cI, dI
      B[aI,bI,cI,dI] = 0
    for sT, aI, bI, cI, dI, sI
      B[aI,bI,cI,dI] += C1[sT+sI,dT+dI] * T3[cT+cI,sT+sI,aI,bI]
    Write BDisk[aT:aT+47,bT:bT+94,cT:cT+94,dT:dT+189]
```

Fig. 7. Concrete code for the 4-index transform example. $N_p = N_q = N_r = N_s = 190$, $N_a = N_b = N_c = N_d = 180$. Memory limit=2GB.

```
for a, i
  for b, j
    t_2[b,j] = 0
  for c, k
    t_1 = ((2.0 * v_ovvo[k,a,c,i]) + (-1.0 * v_ovov[k,a,i,c]))
    for b, j
      t_2[b,j] += (t[b,c,j,k] * t_1)
  for b, j
    t_3 = 0
    for c, k
      t_3 += (t[c,b,j,k] * v_ovvo[k,a,c,i])
    i0[a,b,i,j] = ((t_2[b,j] + (-1.0 * t_3)) + v_vvoo[a,b,i,j])
```

Fig. 8. Abstract code for the kernel from the CCD equation.

```
for a, b, i, j
  X[a,b,i,j] = 0
for b, d
  for j, l
    t_2[j,l] = 0
  for c, e
    t_1 = 0
    for f, k
      t_1 += (D[b,f,e,k] * N[d,c,f,k])
    for j, l
      t_2[j,l] += (S[c,j,e,l] * t_1)
  for a, i, j, l
    X[a,b,i,j] += (T[a,d,i,l] * t_2[j,l])
```

Fig. 9. Abstract code for the three-contraction example.

## 6 Experimental Results

We evaluated the developed approach, referred to as the DCS approach, by comparing it with two alternatives. The first, referred to as the equi-tile-size approach, is used in state-of-the-art quantum chemistry codes. In this approach, equal tile sizes are chosen for all loop indices. The tile sizes are made large enough to fully utilize the available memory. The placement of I/O statements is determined in a greedy fashion. For a given set of tile sizes, the I/O statements are placed at that position in the parse tree in which the total size of the data accessed in that subtree, rooted at that position, just fits in the available memory.

The second approach, referred to as the uniform sampling approach, was developed for locality optimization of the disk-memory hierarchy [26, 12]. A greedy approach to disk I/O placement is used, where for each set of tile sizes, the algorithm places read/write statements immediately inside those loops at which the memory limit is exceeded. The tile size search space is sampled uniformly in a logarithmic fashion along each dimension. This sampled search space is then explored using a brute force approach.

Performance was evaluated on an Itanium-2 Cluster at the Ohio Supercomputer Center. Each node in the cluster is a dual Itanium-2 900 MHz system running Linux 2.4.18. Each node has 4 GB of memory and an 80 GB SCSI hard disk. The generated code was compiled using the Intel Itanium Fortran Compiler for Linux (efc version 7.1). For code generation purposes, the physical memory available to the computation is specified as half the available memory to minimize any paging effects.

The performance of the concrete code generated by the three approaches was evaluated for three computations. The first is a three-contraction computation. It is a synthetic computation in which the outputs of two tensor contractions are contracted again. The loop structure of the computation is shown in Fig. 9. This is a prevalent subtree in tensor contraction operator trees, though it does not occur independently in many codes. The problem size is varied by increasing the range of the loop index $a$, with a corresponding increase in size of the $T$ array. The results are shown in Fig. 10. The size of the $T$ array is shown on the x-axis, with the predicted and measured disk I/O cost shown along the y-axis. It can be observed that the DCS approach is consistently better than the uniform sampling approach, which in turn is superior to using equal tile sizes. The experimentally measured performance matches prediction very closely.

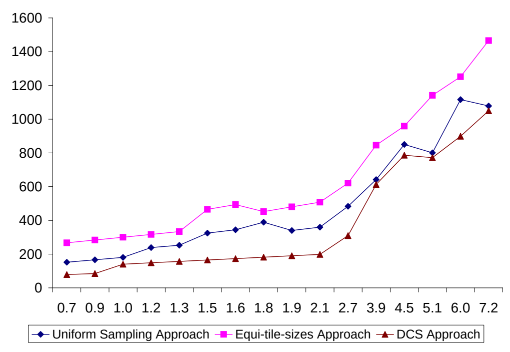
The second computation used for testing is the four-index transform, which was introduced earlier. The imperfectly nested loop structure shown in Fig. 5 was used. The number of virtual orbitals ($V$) was varied, to vary the problem size. Fig. 11 shows the predicted and measured disk I/O costs for the various problem sizes considered, shown in terms of the size of array A, the dominant input array affecting the memory requirement. For this example too, the DCS approach is superior to uniform sampling, which is better than the equi-tile-size approach. For the equi-tile-size approach, the experimentally measured data is a bit lower than prediction; we believe that this is a consequence of caching of produced-consumed files in physical memory.

The third computation considered is a sub-computation from the Coupled Cluster Doubles (CCD) equation [16, 34, 38] for ab initio electronic structure modeling. The computation is given by the loop structure shown in Fig. 8. The predicted and measured disk I/O cost for various virtual orbital ranges was evaluated. The results are shown in Fig: 12. For this computation, the DCS approach is again superior to the other two. However, surprisingly, we find that the uniform sampling approach is not consistently better than the equi-tile-size approach. We believe that this is due to sharp peaks and troughs in the disk-IO-cost as a function of tile size, causing uniform search to miss many local optima.

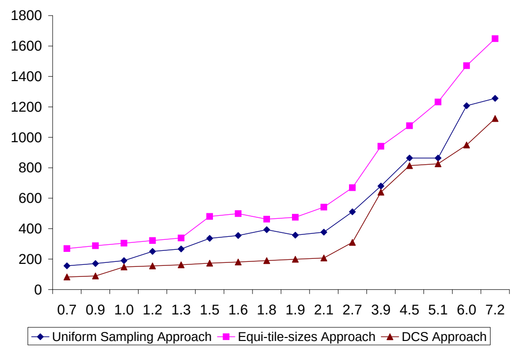
Overall, the graphs show that the predicted disk I/O closely matches the measured cost. The equal-tile-size approach generally performs worse then the other two approaches, while the DCS approach performs consistently better than the other two approaches. It is up to four times better than the equal-tile-size approach and up to two times better than the uniform sampling approach.

For the parallel context, we used the Global Arrays (GA) [43] and Disk-Resident Arrays (DRA) [42] framework. The GA model provides an abstraction of global shared multi-dimensional arrays, transparently implemented on systems with physically distributed memory. The DRA model extends the global shared abstraction to disk-resident multi-dimensional arrays, permitting an arbitrary multi-dimensional segment of a DRA to be moved into a memory-resident GA. The aggregate memory available on all the processors was used as the memory available for the computation, and tiling was done as in the sequential case.

The measured disk I/O times for the parallel code generated for the three-contraction example and the four-index transform are shown in Figs. 13 and 14 respectively.

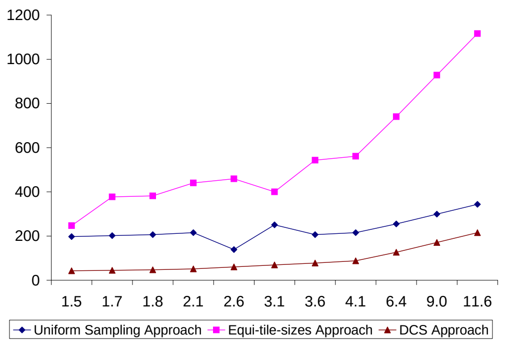


(a) Predicted Disk I/O cost (seconds)

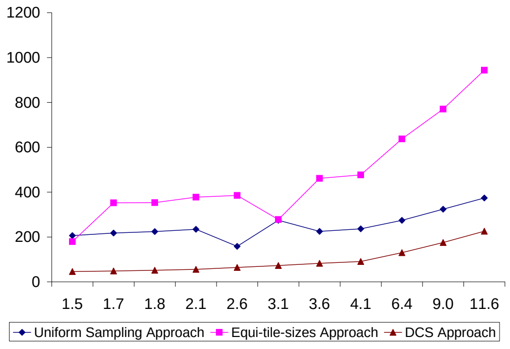


(b) Measured Disk I/O cost (seconds)

Fig. 10. The predicted and measured disk I/O cost for the three-contraction example. The size of $T$ array is shown along x-axis, in gigabytes.

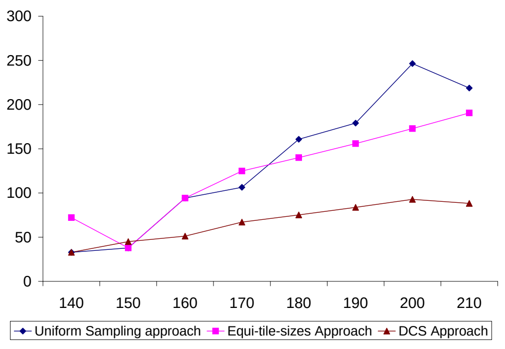


(a) Predicted Disk I/O cost (seconds)

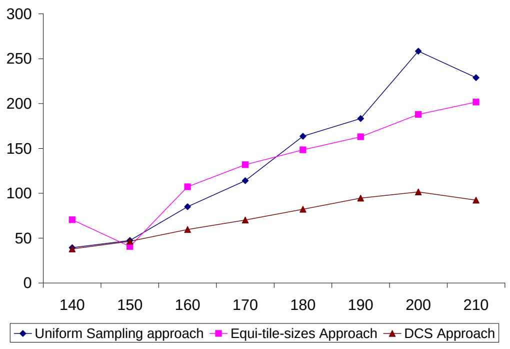


(b) Measured Disk I/O cost (seconds)

Fig. 11. The predicted and measured disk I/O cost for the four-index transform. The size of $A$ array is shown along x-axis, in gigabytes.



(a) Predicted Disk I/O cost (seconds)



(b) Measured Disk I/O cost (seconds)

Fig. 12. The predicted and measured disk I/O cost for the computation from the CCD equation. The virtual orbital ranges ($V$) are shown along the x-axis.

The DCS approach performs significantly better than the other approaches for the different processor counts considered. In particular, considerable improvement in performance can be observed for smaller processor counts. This shows the improved resource-utilization by the DCS approach, potentially enabling larger problems to be computed on a given machine more efficiently.

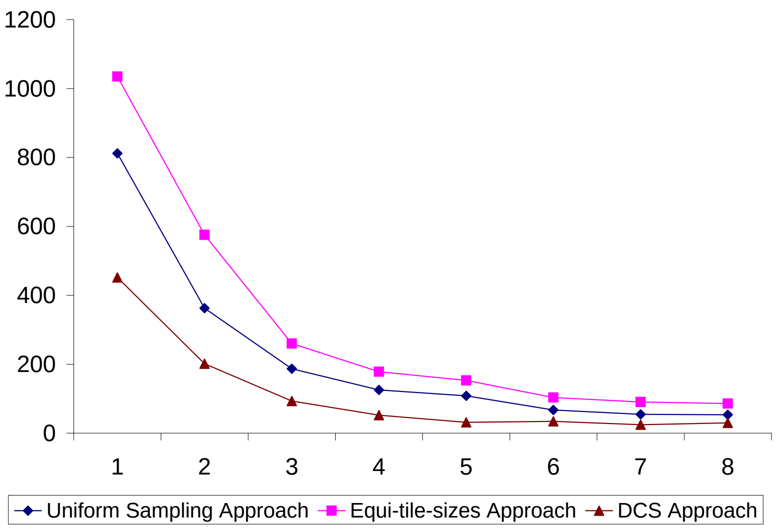


Fig. 13. The measured disk I/O cost for the generated parallel code for the three-contraction example. The number of processors is shown along the x-axis.

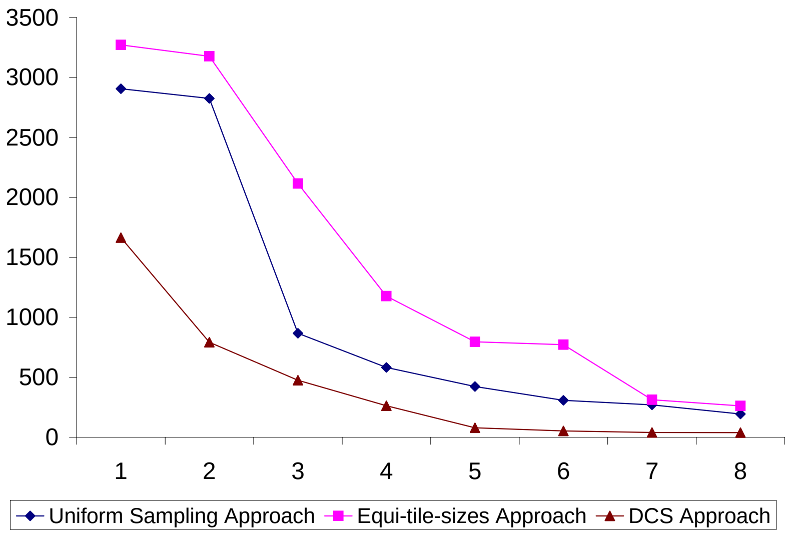


Fig. 14. The measured disk I/O cost, in seconds, for the generated parallel code for the four-index transform. The number of processors is shown along the x-axis.

### *6.1 Computational complexity of the three evaluated approaches*

The three approaches represent varying degrees of complexity in the code generation process, in terms of the determination of I/O placements and tile sizes. With the equal-tiles approach, the tile sizes and the I/O placements are determined in a straightforward manner based solely on the available memory. This approach vastly simplifies code generation, while not necessarily resulting in quality code. The choice of equal tile sizes for all the arrays ignores various aspects of the program such as the reuse characteristics of the different arrays, thus resulting in suboptimal code.

The uniform sampling approach takes a greedy approach to placement of I/O statements, and searches the sample space of possible tile sizes to choose the set of tile sizes that minimize the I/O cost. The greedy I/O placement strategy, while simplifying code generation complexity, is not always the best strategy. It also decouples the two phases of the code generation problem, thus potentially affecting the running time. Also, the sampling nature of the tile size search does not always produce optimal code, while significantly affecting code generation time. The dimensionality of the search space is linearly proportional to the number of loop indices.

The DCS approach produces a composite cost function combining the effects of I/O placements and tile sizes on the disk I/O cost. The search over this composite space uses heuristics established in the solver. The encoding of the I/O placements is such that the search space to be explored for the I/O placements increases logarithmically with the number of loop indices. Hence the complexity of the sample space to be explored is still linear in the number of loop indices, while generally generating a more globally optimal solution.

## 7 Conclusion

We have described an approach to the synthesis of out-of-core algorithms for a class of imperfectly nested loops. The approach was developed for the implementation in a component of a program synthesis system targeted at the quantum chemistry domain. The determination of optimal placements of disk I/O statements and choice of tile sizes requires search in a very large search space. By formulating it as a nonlinear constrained optimization problem, and use of a general-purpose constrained optimization solver, code was generated that outperforms other approaches by up to a factor of four.

## Acknowledgments

We would like to sincerely thank Prof. Benjamin Wah and Yi Xin Chen of the University of Illinois for their significant help with using the Discrete Constrained Search (DCS) Solver. We wish to express our appreciation to the reviewers of the paper for their suggestions. We are grateful to the Ohio Supercomputer Center (OSC) for the use of their computing facilities.

## References

[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly nested loops. In *Proc. ACM Intl. Conf. on Supercomputing*, pages 141–152, 2000.

[2] J. Anderson, S. Amarasinghe, and M. Lam. Data and Computation Transformations for Multiprocessors. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 166–178, July 1995.

[3] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of High-Performance Parallel Programs for a Class of Ab Initio Quantum Chemistry Models. *Proceedings of the IEEE*, 93(2):276–292, 2005.

[4] G. Baumgartner, D. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan. A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry. In *Proc. Supercomputing 2002*, November 2002.

[5] R. Bordawekar. *Techniques for Compiling I/O Intensive Parallel Programs*. PhD thesis, Dept. of Electrical and Computer Eng., Syracuse University, April 1996.

[6] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A Model and Compilation Strategy for Out-of-Core Data-Parallel Programs. In *Proc. 5th ACM Symposium on Principles and Practice of Parallel Programming*, pages 1–10, 1995.

[7] R. Bordawekar, A. Choudhary, and J. Ramanujam. Automatic Optimization of Communication in Out-of-Core Stencil Codes. In *Proc. 10th ACM International Conference on Supercomputing*, pages 366–373, 1996.

[8] Y. Chen. Optimal Anytime Search for Constrained Nonlinear Programming. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, IL, May 2001.

[9] Y. Chen, M. Winslett, Y. Cho, and S. Kuo. Automatic parallel I/O performance optimization in Panda. In *Proc. 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 108–118, 1998.

[10] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison. Space-Time Trade-Off Optimization for a Class of Electronic Structure Calculations. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 177–186, 2002.

[11] D. Cociorva, X. Gao, S. Krishnan, G. Baumgartner, C. Lam, P. Sadayappan, and J. Ramanujam. Global Communication Optimization for Tensor Contraction Expressions under Memory Constraints. In *Proc. 17th International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[12] D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison. Towards Automatic Synthesis

of High-Performance Codes for Electronic Structure Calculations: Data Locality Optimization. In *Proc. Intl. Conf. on High Performance Computing*, volume 2228, pages 237–248. Springer-Verlag, 2001.

[13] D. Cociorva, J. Wilkins, 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 (ICS'01)*, pages 500–509, 2001.

[14] S. Coleman and K. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *Proc. SIGPLAN '95 Conference on Programming Languages Design and Implementation*, pages 279–290, 1995.

[15] T. Cormen and A. Colvin. ViC*: A Preprocessor for Virtual-Memory C*. Technical Report PCS-TR94-243, Dartmouth College, November 1994.

[16] T. Crawford and H. Schaefer III. An Introduction to Coupled Cluster Theory for Computational Chemists. In K. Lipkowitz and D. Boyd, editor, *Reviews in Computational Chemistry*, volume 14, pages 33–136. John Wiley & Sons, Ltd., 2000.

[17] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level-3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[18] R. Fourer, D. Gay, and B. Kernighan. AMPL: A Modeling Language for Mathematical Programming, 2002.

[19] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Trans. Program. Lang. Syst.*, 21(4):703–746, 1999.

[20] M. Kandemir, A. Choudhary, and J. Ramanujam. An I/O conscious tiling strategy for disk-resident data sets. *The Journal of Supercomputing*, 21(3):257–284, 2002.

[21] M. Kandemir, A. Choudhary, J. Ramanujam, and R. Bordawekar. Compilation techniques for out-of-core parallel computations. *Parallel Computing*, 24(3-4):597–628, June 1998.

[22] M. Kandemir, A. Choudhary, J. Ramanujam, and M. Kandaswamy. A unified framework for optimizing locality, parallelism, and communication in out-of-core computations. *IEEE Transactions of Parallel and Distributed Systems*, 11(7):648–668, July 2000.

[23] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, pages 346–357, 1997.

[24] 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)*, pages 482–491, 1999.

[25] S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam, and P. Sadayappan. On Efficient Out-of-core Matrix Transposition. Technical Report OSU-CIRSC-9/03-T52, The Ohio State University, Columbus, OH, September 2003.

[26] S. Krishnan. Data Locality Optimization for Synthesis of Out-of-Core Pro-

grams. Master's thesis, The Ohio State University, Columbus, OH, September 2003.

[27] S. Krishnan, S. Krishnamoorthy, G. Baumgartner, D. Cociorva, C. Lam, P. Sadayappan, J. Ramanujam, D. Bernholdt, and V. Choppella. Data Locality Optimization for Synthesis of Efficient Out-of-Core Algorithms. In *Proc. Intl. Conf. on High Performance Computing*, 2003.

[28] C. Lam. *Performance Optimization of a Class of Loops Implementing Multi-Dimensional Integrals*. PhD thesis, The Ohio State University, Columbus, OH, August 1999.

[29] 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.*, pages 103–110. Springer Verlag, 1999.

[30] 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. In *Proc. 12th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 350–364. Springer Verlag, 1999.

[31] C. Lam, P. Sadayappan, and R. Wenger. On Optimizing a Class of Multi-Dimensional Loops with Reductions for Parallel Execution. *Parallel Processing Letters*, 7(2):157–168, 1997.

[32] C. Lam, P. Sadayappan, and R. Wenger. Optimization of a Class of Multi-Dimensional Integrals on Parallel Machines. In *Proc. 8th SIAM Conf. on Parallel Processing for Scientific Computing*, 1997.

[33] M. Lam, E. Rothberg, and M. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[34] T. Lee and G. Scuseria. Achieving chemical accuracy with coupled cluster theory. In S. Langhoff (Ed.). *Quantum Mechanical Electronic Structure Calculations with Chemical Accuracy*, pages 47–109, 1997.

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

[36] W. Li. Compiler cache optimizations for banded matrix problems. In *Proc. International Conference on Supercomputing*, pages 21–30, 1995.

[37] A. Lim, S. Liao, and M. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proc. 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 103–112. ACM Press, 2001.

[38] J. Martin. Benchmark Studies on Small Molecules. In P. Schleyer, P. Schreiner, N. Allinger, T. Clark, J. Gasteiger, P. Kollman, and H. Schaefer III, editors, *Encyclopedia of Computational Chemistry*, volume 1, pages 115–128. John Wiley & Sons, Ltd., 1998.

[39] K. McKinley, S. Carr, and C. Tseng. Improving Data Locality with Loop Transformations. *ACM TOPLAS*, 18(4):424–453, July 1996.

[40] N. Mitchell, K. Hogstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *Intl. Journal of Parallel Programming*,

26(6):641–670, June 1998.

[41] T. Mowry, A. Demke, and O. Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *Proc. 2nd Symposium on Operating Systems Design and Implementations*, pages 3–17, 1996.

[42] J. Nieplocha and I. Foster. Disk Resident Arrays: An Array-Oriented I/O Library for Out-Of-Core Computations. In *Proc. 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 196–204, 1996.

[43] J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers. *The Journal of Supercomputing*, 10:197–220, 1996.

[44] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler Support for Out-of-Core Arrays on Parallel Machines. Technical Report 94509-S, Rice University, Houston, TX, December 1994.

[45] G. Rivera and C. Tseng. Eliminating Conflict Misses for High Performance Architectures. In *Proc. Intl. Conf. on Supercomputing*, pages 353–360, 1998.

[46] Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 215–228, 1999.

[47] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. PASSION Runtime Library for Parallel I/O. In *Proc. Scalable Parallel Libraries Conference*, pages 119–128, 1994.

[48] B. Wah and Y. Chen. Web Interface for Discrete Constrained Search Solver.

[49] B. Wah and Y. Chen. Constrained Genetic Algorithms and their Applications in Nonlinear Constrained Optimization. In *Proc. 11th IEEE Int'l Conf. on Tools with Artificial Intelligence*, pages 286–293, 2000.

[50] B. Wah and Y. Chen. Optimal Anytime Constrained Simulated Annealing for Constrained Global Optimization. In *Proc. ACM Symposium on Principles and Practice of Constraint Programming*, pages 425–439. Springer-Verlag, 2000.

[51] B. Wah and Y. Chen. Hybrid Constrained Simulated Annealing and Genetic Algorithms for Nonlinar Constrained Optimization. In *Proc. IEEE Congress on Evolutionary Computation*, pages 925–932, 2001.

[52] M. Wolf and M. Lam. A Data Locality Algorithm. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 30–44, 1991.

[53] M. Wolf, D. Maydan, and D. Chen. Combining loop transformations considering caches and scheduling. In *Proc. 29th Annual International Symposium on Microarchitecture*, pages 274–286, 1996.