Space-Time Trade-Off Optimization for a Class of Electronic Structure Calculations*

Technical Report OSU-CISRC-11/01-TR24

Daniel Cociorva[1] Gerald Baumgartner[2] Chi-Chung Lam[2] P. Sadayappan[2]
J. Ramanujam[3] Marcel Nooijin[4] David E. Bernholdt[5] Robert Harrison[6]

[1] Department of Physics
The Ohio State University
cociorva@pacific.mps.ohio-state.edu

[2] Department of Computer and Information Science
The Ohio State University
{gb,clam,saday}@cis.ohio-state.edu

[3] Department of Electrical and Computer Engineering
Louisiana State University
jxr@ece.lsu.edu

[4] Department of Chemistry
Princeton University
Nooijen@Princeton.edu

[5] Oak Ridge National Laboratory
bernholdtde@ornl.gov

[6] Pacific Northwest National Laboratory
Robert.Harrison@pnl.gov

November 30, 2001

Abstract

The accurate modeling of the electronic structure of atoms and molecules is very computationally intensive. Many models of electronic structure, such as the Coupled Cluster approach, involve collections of tensor contractions. There are usually a large number of alternative ways of implementing the tensor contractions, representing different trade-offs between the space required for temporary intermediates and the total number of arithmetic operations. In this paper, we present an algorithm that starts with an operation-minimal form of the computation and systematically explores the possible space-time trade-offs to identify the form with lowest cost that fits within a specified memory limit. Its utility is demonstrated by applying it to a computation representative of a component in the CCSD(T) formulation in the NWChem quantum chemistry suite from Pacific Northwest National Laboratory.

*We are grateful to the National Science Foundation for support of this work through the Information Technology Research Program

1 Introduction

The development of high-performance parallel programs for scientific applications is usually very time consuming. The time to develop an efficient parallel program for a computational model can be a primary limiting factor in the rate of progress of the science. 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 quantum chemistry. 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.

A critical issue in implementing electronic structure models, e.g. using coupled cluster methods, is the management of storage requirements for intermediates. Significant savings in computational cost can be achieved by computing and storing various intermediate array quantities, that are reused several times in the process of generating the final results. However, the space requirements for these intermediates is often extremely large, making it infeasible to store even on disk. In this case, there is no choice but to discard and recompute some of the intermediates. Therefore the following optimization problem is of great interest: given a set of computations expressed as a sequence of tensor contractions (explained later on), and a specified limit on the amount of available storage, re-structure the computation so as to minimize the amount of redundant recomputation required. In this paper, we present a framework that we have developed to address this problem. The space-time trade-off optimization we consider here is part of a planned synthesis system that incorporates a number of optimization modules.

The computational structures that we address in this paper arise in scientific application domains that are extremely compute-intensive and consume significant computer resources at national supercomputer centers. They are present in computational physics codes modeling electronic properties of semiconductors and metals [1, 7, 16], 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 [14, 15]. 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, the pertinent optimization issues and an overview of the overall synthesis system that is under development. Section 3 elaborates on the problem using a concrete example that is abstracted from a computationally intensive calculation in the NWChem [6] system. Section 4 provides a high-level description of the solution approach. Sections 5 and 6 present details of the approach to solve the space-time trade-off problem. Section 7 presents results from the application of the new algorithm to the example abstracted from NWChem. Conclusions are provided in Section 8.

2 The Computational Context

In the class of computations considered, the final result to be computed can be expressed in terms of tensor contractions, essentially 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:

$$S_{abij} = \sum_{cdefkl} A_{acik} \times B_{befl} \times C_{dfjk} \times D_{cdel}$$

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:

$$S_{abij} = \sum_{ck} \left(\sum_{df} \left(\sum_{el} B_{befl} \times D_{cdel} \right) \times C_{dfjk} \right) \times A_{acik}$$

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

$$
\begin{aligned}
T1_{bcdf} &= \sum_{el} B_{befl} \times D_{cdel} \\
T2_{bcjk} &= \sum_{df} T1_{bcdf} \times C_{dfjk} \\
S_{abij} &= \sum_{ck} T2_{bcjk} \times A_{acik}
\end{aligned}
$$

(a) Formula sequence

```
T1=0; T2=0; S=0
for b, c, d, e, f, l
  T1bcdf += Bbefl Dcdel
for b, c, d, f, j, k
  T2bcjk += T1bcdf Cdfjk
for a, b, c, i, j, k
  Sabij += T2bcjk Aacik
```

(b) Direct implementation (unfused code)

```
S = 0
for b, c
  T1f = 0; T2f = 0
  for d, f
    for e, l
      T1f += Bbefl Dcdel
    for j, k
      T2fjk += T1f Cdfjk
  for a, i, j, k
    Sabij += T2fjk Aacik
```

(c) Memory-reduced implementation (fused)

Figure 1: Example illustrating use of loop fusion for memory reduction.

arrays $T1$ and $T2$. Often, the space requirements for the temporary arrays poses a serious problem. For this example, abstracted from a quantum chemistry model, the array extents along indices $a - d$ are the largest, while the extents along indices $i - l$ are the smallest. Therefore, the size of temporary array $T1$ would dominate the total memory requirement.

The operation minimization problem encountered here is a generalization of the well known matrix-chain multiplication problem, where a linear chain of matrices to be multiplied is given, e.g. ABCD, and the optimal order of pair-wise multiplications is sought, i.e. ((AB)C)D versus (AB)(CD) etc. In contrast to this, for computations expressed as sets of matrix contractions, although the final realization of the computation is in terms of a sequence of matrix-matrix products, there is additional freedom in choosing the pair-wise products. For the above example, instead of forcing a single chain order, e.g. ABCD, other orders are possible, such as the BCDA order shown for the operation-reduced form above.

We have previously shown that the problem of determining the operator tree with minimal operation count is NP-complete, and have developed a pruning search procedure [12, 13] that is very efficient in practice. For the above example, although the latter form is far more economical in terms of the number of arithmetic operations, its implementation will require the use of temporary intermediate arrays to hold the partial results of the parenthesized array subexpressions. Sometimes, the sizes of intermediate arrays needed for the “operation-minimal” form are too large to even fit on disk.

A systematic way 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 which is consumed by another loop nest, fusing the two loop nests allows the dimension corresponding to the fused loop to be eliminated in the array. This results in a smaller intermediate array and thus reduces the memory requirements. For the example considered, the application of fusion is illustrated in Fig. 1(c). By use of loop fusion, for this example it can be seen that $T1$ can actually be reduced to a scalar and $T2$ to a 2-dimensional array, without changing the number of arithmetic operations.

For a computation comprising of a number of nested loops, there will generally be a number of fusion choices, that are not all mutually compatible. This is because different fusion choices could require different loops to be made the outermost. In prior work, we addressed the problem of finding the choice of fusions for a given operator tree that minimized the total space required for all arrays after fusion [9, 10, 11].

However, for many of the computational structures within the coupled cluster component of the NWChem software suite, we find instances where the minimal memory required after optimal loop fusion is still too large. In such situations, in order to create an executable implementation, it is essential to trade space for time, by only storing lower dimensional slices of the largest arrays, and recomputing the slices as needed. This is the compiler optimization problem we address in this paper. We extend the use of a previously proposed concept of a *fusion graph* and develop an algorithm that explores a space of alternative space-time trade-offs to determine the *best* set of lower-dimensional arrays that fit within a specified space limit, so that the additional recomputation cost is minimized.

The problem addressed in this paper is one of several optimization issues being addressed in the context of a larger effort to develop a tool for the automatic synthesis of high-performance parallel code from a high-level specification for a class of quantum chemistry calculations.

Figure 2: The Planned Synthesis System

Figure 2 shows the components of the system being developed. 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 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.

Memory Minimization: The operation-minimal computation sequence synthesized by the Algebraic Transformation module might require an excessive amount of memory due to the large temporary intermediate arrays involved. The Memory Minimization module attempts to perform 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 unless a successful space-time trade-off is performed. This is the issue we address in this paper. If no satisfactory 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.

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. These issues have been addressed elsewhere [3, 4].

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.

```
for a, e, c, f
    for i, j
        $X_{aecf}$ += $T_{ijae}$ $T_{ijcf}$
for c, e, b, k
    $T1_{cebk}$ = $f_1$(c, e, b, k)
for a, f, b, k
    $T2_{afbk}$ = $f_2$(a, f, b, k)
for c, e, a, f
    for b, k
        $Y_{ceaf}$ += $T1_{cebk}$ $T2_{afbk}$
for c, e, a, f
    E += $X_{aecf}$ $Y_{ceaf}$
```

array	space	time
X	V^4	V^4O^2
T1	V^3O	C_iV^3O
T2	V^3O	C_iV^3O
Y	V^4	V^5O
E	1	V^4

Figure 3: Unfused operation-minimal form.

In the next section we use an example from quantum chemistry to further elaborate on the space-time trade-off optimization addressed in this paper.

3 Elaboration of the Problem

One of the most computationally intensive components of many quantum chemistry packages is the CCSD(T) scheme. It is a coupled cluster approximation that includes all single and double excitations from the Hartree-Fock wavefunction plus a perturbative estimate for the *connected* triple excitations. For molecules well described by a Hartree-Fock wave function, the CCSD(T) method predicts bond energies, ionization potentials, and electron affinities to an accuracy of approximately ± 0.5 kcal/mol, bond lengths accurate to $\pm 0.0005\dot{A}$, and vibrational frequencies accurate to $\pm 5cm^{-1}$. This level of accuracy is adequate to answer many of the questions that arise in studies of complex chemical systems.

As a motivating example for the problem addressed, we discuss a component of the CCSD(T) calculation. The following representative equation arises in the Laplace factorized expression for linear triples perturbation correction.

$$A3A = \frac{1}{2}\left(X_{ce,af}Y_{ae,cf} + X_{a\bar{e},c\bar{f}}Y_{c\bar{e},a\bar{f}} + X_{a\bar{e},\bar{c}f}Y_{\bar{c}\bar{e},af} + X_{\bar{a}e,c\bar{f}}Y_{ce,\bar{a}\bar{f}} + X_{\bar{a}e,\bar{c}f}Y_{\bar{c}e,\bar{a}f} + X_{\bar{a}\bar{e},\bar{c}\bar{f}}Y_{\bar{c}\bar{e},\bar{a}\bar{f}}\right)$$

where the X and Y intermediates are of the form $X_{ae,cf} = t_{ij}^{ae}t_{ij}^{cf}$ and $Y_{ce,af} = \langle cb \parallel ek\rangle\langle ab \parallel fk\rangle$, respectively.

Integrals with two vertical bars have been antisymmetrized and may be expressed as: $(\langle pq \parallel rs\rangle = \langle pq \mid rs\rangle - \langle pq \mid sr\rangle)$, where integrals with one vertical bar are of the form $\langle \mu\nu \mid \omega\lambda\rangle = \int\int dr^3 ds^3 \phi_\mu(\underline{r})\phi_\nu(\underline{s})|\underline{r} - \underline{s}|^{-1}\phi_\omega(\underline{r})\phi_\lambda(\underline{s})$ and are quite expensive to compute (requiring on the order of 1000 arithmetic operations). Electrons may have either up or down (or alpha/beta) spin. Down spin is denoted here with an over bar. The indices i, j, k, l, m, n refer to occupied orbitals), of number O between 30 and 100. The indices a, b, c, d, e, f refer to unoccupied orbitals of number V between 1000 and 3000. The integrals are written in the molecular orbital basis, but must be computed in the underlying atom-centered Gaussian basis, and transformed to the molecular orbital basis. We omit these details in our discussion here.

A3A is one of many contributions to the energy, and among the most expensive, scaling as $O(OV^5)$. Here, we assume that we have already computed the amplitudes t_{ij}^{ae}, and they must be read as necessary, and contracted to form a block of X. The integrals $\langle cb \parallel ek\rangle$ must be recomputed as necessary, contracted to form a block of Y corresponding to X, and the two contracted to form the scalar contribution to the energy.

Figure 3 shows pseudo-code for the computation of one of the energy components E for $A3A$. Temporary arrays $T1$ and $T2$ are used to store the integrals of form $\langle ab \parallel ek\rangle$, where the functions $f1$ and $f2$ represent the integral calculations. The intermediate quantities X_{aecf} are computed by contracting over (i.e., summing over products of) input array T, while the intermediate quantities Y_{ceaf} are obtained by contracting over $T1$ and $T2$. The final result is a single scalar quantity E, that is obtained by adding together the $O(OV^3)$ pair-wise products $X_{aecf}Y_{ceaf}$.

The cost of computing each integral $f1$, $f2$ is represented by C_i, and in practice is of the order of hundreds or a few thousand arithmetic operations. The pseudo-code form shown in Fig. 3 is computationally very efficient in minimizing the number of expensive integral function evaluations $f1$ and $f2$, and maximizing the reuse of the stored

```
for a, e, c, f
    for i, j
        $X_{aecf}$ += $T_{ijae}$ $T_{ijcf}$
for a, f
    for c, e, b, k
        $T1_{cebk}$ = $f_1$(c,e,b,k)
for c, e
    for a, f, b, k
        $T2_{afbk}$ = $f_2$(a,f,b,k)
for c, e, a, f
    for b, k
        $Y_{ceaf}$ += $T1_{cebk}$ $T2_{afbk}$
for c, e, a, f
    E += $X_{aecf}$ $Y_{ceaf}$
```

⇒

```
for a, e, c, f
    for i, j
        X += $T_{ijae}$ $T_{ijcf}$
    for b, k
        T1 = $f_1$(c,e,b,k)
        T2 = $f_2$(a,f,b,k)
        Y += T1 T2
    E += X Y
```

array	space	time
X	1	V^4O^2
T1	1	C_iV^5O
T2	1	C_iV^5O
Y	1	V^5O
E	1	V^4

Figure 4: Use of redundant computation to allow full fusion.

```
for $a^t$, $e^t$, $c^t$, $f^t$
    for a, e, c, f
        for i, j
            $X_{aecf}$ += $T_{ijae}$ $T_{ijcf}$
    for b, k
        for c, e
            $T1_{ce}$ = $f_1$(c,e,b,k)
        for a, f
            $T2_{af}$ = $f_2$(a,f,b,k)
        for c, e, a, f
            $Y_{ceaf}$ += $T1_{ce}$ $T2_{af}$
    for c, e, a, f
        E += $X_{aecf}$ $Y_{ceaf}$
```

array	space	time
X	B^4	V^4O^2
T1	B^2	$C_i(\frac{V,B}{)}^2V^3O$
T2	B^2	$C_i(\frac{V,B}{)}^2V^3O$
Y	B^4	V^5O
E	1	V^4

Figure 5: Use of tiling and partial fusion to reduce recomputation cost.

integrals in $T1$ and $T2$ (each element of $T1$ and $T2$ is used $O(V^2)$ times). However, it is impractical due to the huge memory requirement. With $O = 100$ and $V = 5000$, the size of $T1$, $T2$ is $O(10^{14})$ bytes and the size of X, Y is $O(10^{15})$ bytes. By fusing together pairs of producer-consumer loops in the computation, reductions in the needed array sizes may be sought, since the fusion of a loop with common index in the pair of loops allows the elimination of that dimension of the intermediate array. It can be seen that the loop that produces X (with indices a, e, c, f), the loop that produces Y (with indices c, e, a, f) and the loop that consumes X and Y to produce E (with indices c, e, a, f) can all be fully fused together, permitting the elimination of all explicit indices in X and Y to reduce them to scalars. However, the loops producing $T1$ (with indices c, e, b, k) and $T2$ (with indices a, f, b, k) cannot also be directly fused with the other three loops because their indices do not match.

Figure 4 shows how reduction of space for $T1$ and $T2$ can be achieved by introduction of redundant loops around their producer loops — add loops with the missing indices a, f for $T1$ and c, e for $T2$. Now all five of the loops have common indices a, e, c, f that can be fused, permitting elimination of those indices from all temporaries. Further, by fusing together the producer loops for $T1$ and $T2$ with their consumer loop that produces Y, the b, k indices can also be eliminated from $T1$ and $T2$. Dramatic reduction of memory space is achieved, reducing all temporaries $T1, T2, X$ and Y to scalars. However, the space savings come at the price of significant increase in computation. Now, no reuse is achieved of the quantities derived from the expensive integral calculations $f1$ and $f2$. Since C_i is of the order of 1000 in practice, the integral calculations now dominate the total compute time, increasing the operation count by three orders of magnitude over the unfused form in Fig. 3.

A desirable solution would be somewhere in between the unfused structure of Fig. 3 (with maximal memory requirement and maximal reuse) and the fully fused structure of Fig. 4 (with minimal memory requirement and minimal reuse). This is shown in Fig. 5, where tiling and partial fusion of the loops is employed. The loops with indices a, e, c, f

Figure 6: Fusion graph for unfused operation-minimal form of loop in Figure 3

are tiled by splitting each of those indices into a pair of indices. The indices with a superscript t represent the tiling loops and the unsuperscripted indices now stand for intra-tile loops with a range of B, the block size used for tiling. For each tile (a^t, e^t, c^t, f^t), blocks of $T1$ and $T2$ of size B^2 are computed and used to form B^4 product contributions to the appropriate components of Y, which are stored in an array of size B^4.

As the tile size B is increased, the cost of function computation for $f1, f2$ decreases by factor B^2, due to the reuse enabled. However, the size of the needed temporary array for Y increases as B^4 (the space needed for X can actually be reduced back to a scalar by fusing its producer loop with the loop producing E, but Y's space requirement cannot be decreased). When B^4 becomes larger than the size of physical memory, expensive paging in and out of disk will be required for Y. Further, there is diminishing returns on reuse of $T1$ and $T2$ after B^2 becomes comparable to C_i, since the loop producing Y now becomes the dominant one. So we can expect that as B is increased, performance will improve and then level off and then deteriorate. The optimum value of B will clearly depend on the cost of access at the various levels of the memory hierarchy.

The computation considered here is just one component of the $A3A$ term, which in turn is only one of very many terms that must be computed. Although developers of quantum chemistry codes naturally recognize and perform some of these optimizations, a collective analysis of all these computations to determine their optimal implementation is beyond the scope of manual effort. While recent developments in optimizing compiler research have resulted in significant strides in data locality optimization, we are unaware of any existing work that addresses the kind of space-time trade-off optimization required in the context we consider.

4 Solution Approach: The Fusion Graph

The operation-minimization procedure discussed above usually results in the creation of intermediate temporary arrays. Sometimes these intermediate arrays that help in reducing the number of arithmetic operations create a problem with the memory capacity required.

For a computation comprising of a number of nested loops, there will generally be a number of fusion choices, that are not all mutually compatible. This is because different fusion choices could require different loops to be made the outermost. A data structure that we call a $fusion\ graph$ can be used to facilitate enumeration of all possible compatible fusion configurations for a given computation tree.

Figure 6 shows the fusion graph for the unfused form of the computation from Fig. 3. Corresponding to each node in a computation tree, the fusion graph has a set of vertices corresponding to the loop indices of the node of the computation tree. In Fig. 6, we do not show the operator tree corresponding to the computation, but directly illustrate the fusion graph. The potential for fusion of a common loop among a producer-consumer pair of loop nests is indicated in the fusion graph through a dashed $potential\ fusion$ edge connecting the corresponding vertices. Leaf nodes in the fusion graph correspond to input arrays or primitive function evaluations and do not represent a loop nest. The edges from the leaves to their parents are shown as dotted edges and do not affect the fusion possibilities. If a pair of loop nests is fused using one or more common loops, it is captured in the fusion graph by changing the dashed potential-fusion edges to continuous fusion edges. If more than two loop nests are fused together, a chain of fusion edges results, called a *fusion chain*. The *scope of a fusion chain* is the set of nodes it spans. The fusion graph allows us to characterize the condition for feasibility of a particular combination of fusions: the scope of any two fusion chains

(a) Fully fused computation from Fig. 4.

(b) Partially fused computation from Fig. 5.

Figure 7: Fusion graphs showing redundant compution and tiling.

in a fusion graph must either be disjoint or a subset/superset of each other. Scopes of fusion chains do not partially overlap because loops do not (i.e., loops must be either separate or nested).

The fusion graph in Fig. 6 can be used to determine the fusion possibilities. On the left side of the graph, the edges corresponding to (a, e, c, f) can all be made fusion edges, suggesting that complete fusion is possible for the loop nests producing and consuming X, reducing it to a scalar. Similarly, on the right side of the graph, the edges corresponding to (c, e, a, f) can also be made fusion edges, reducing Y to a scalar. Further, by creating fusion edges for indices (c, e), the producer loop for $T1$ can be fully fused with the Y loop that consumes it. However, now the producer loop for $T2$ cannot be fused since the addition of any fusion edge (say for index a) will result in partially overlapping fusion chains for a and (c, e).

The fully fused version from Fig. 4 can be represented graphically as shown in Fig. 7(a). Additional vertices have been added for indices (c, e) and (a, f) respectively at the nodes corresponding to the producer loops for $T1$ and $T2$. Now, complete fusion chains can be created without any partial overlap in the scopes of the fusion chains. From the figure, it can be seen that in fact the redundant computation need only be added to one of $T1$ or $T2$ to achieve complete fusion — for example, removing the additional vertices for (a, f) at $T2$ does not violate the non-partial-overlap condition for fusion.

The fusion graph was used to develop an algorithm [10, 9] to determine the combination of fusions that minimizes the total storage required for all the temporary intermediate arrays. A bottom-up dynamic programming approach was used, that maintains a set of pareto-optimal fusion configurations at each node, merging solutions from children nodes to generate the optimal configurations at a parent. The two metrics used are the total memory required under the subtree rooted at the node, and the constraints imposed by a configuration on fusion further up the tree. A configuration is inferior to another if it is "*more or equally constraining*" with respect to further fusions than the other, and uses no less memory. At the root of the tree, the configuration with lowest memory requirement is chosen.

Although the complexity of the algorithm is exponential in the number of index variables and the number of solutions could in theory grow exponentially with the size of the expression tree, the number of index variables in practical applications is small enough and there is indication that the pruning is effective in keeping the size of the solution set at each node small.

The fusion graph framework addresses a memory minimization problem, without changing the operation count. If we applied it to the fusion graph of Fig. 3, the bottom-up dynamic programming algorithm would evaluate a number of potential fusion combinations and find that fusion could be used to reduce the sizes of arrays X and Y and convert them to scalars. It would also be able to reduce the size of one of the two temporary arrays $T1$ or $T2$, but would be unable to reduce the other at all. Although three of four temporary arrays would be dramatically reduced in size, the size of the single remaining temporary array (of size $O(V^3O)$) would make the problem unexecutable on most systems due to disk storage limits.

An enhancement of the model to capture a wider range of space-time trade-offs was already seen in Fig. 7(a), where additional vertices were added to the fusion graph to introduce redundant recomputation to the producer loops for $T1$ and $T2$ and thereby enable a greater degree of fusion. As discussed earlier, the fully fused version of the loops results in excellent memory savings but adds excessive recomputation costs. A combination of fusion and tiling is needed to achieve a good balance between recomputation and memory usage. Figure 7(b) shows how the possibility of tiling can be introduced into the fusion graph. For each loop of a loop nest that is to be tiled, the corresponding vertex in the fusion graph is replaced by a pair of vertices — one to represent the outer tiling loop and another to denote the intra-tile loop. By a choice of fusion configuration that only involves the tiling loops, a combination of fusion and tiling can be represented. This framework can be used to explore a range of space-time trade-offs. However, the search space is significantly larger than that for the memory minimization problem discussed in the previous sub-section, requiring that selective search strategies be developed.

In this paper, we develop a two-step search strategy for exploration of the space-time trade-off:

- Search among all possible ways of introducing redundant loop indices in the fusion graph to reduce memory requirements, and determine the optimal set of lower dimensional intermediate arrays for various total memory limits. In this step, the use of tiling for partial reduction of array extents is not considered. However, among all possible combinations of lower dimensional arrays for intermediates, the combination that minimizes recomputation cost is determined, for a specified memory limit. The range from zero to the actual memory limit is split into subranges within which the optimal combination of lower dimensional arrays remains the same.

- Because the first step only considers complete fusion of loops, each array dimension is either fully eliminated or left intact, i.e. partial reduction of array extents is not performed. The objective of the second step is to allow for such arrays. Starting from each of the optimal combinations of lower dimensional intermediate arrays derived in the first step, possible ways of using tiling to partially expand arrays along previously compressed dimensions are explored. The goal is to further reduce recomputation cost by partially expanding arrays to fully utilize the available memory

5 Dimension Reduction for Intermediate Arrays

In the first step of the space-time trade-off algorithm we search among all possible combinations of redundant computations and loop fusions. The search is structured as a dynamic programming algorithm with pruning.

The input to this algorithm is an expression tree representing the operation-minimal computational structure of the input formula. Expression tree nodes are of four types:

- array references a`[`i`]` with index vector i,

- function calls f`(`i`)` with argument vector i,

- summation quantifiers `sum(`i`,`t`)` with summation indices i and subtree t, and

- binary operators `bin(`o`,`l`,`r`)` with operator o (`+`, `-`, or `*`) and subtrees l and r.

For each tree node v, let *indices(v)* be the set of loop indices needed for evaluating v, and let *fusible(v)* be the set of indices that can be fused with the parent (indices other than summation indices). An index i is a redundant index for node v if i is not an index of v but of some ancestor node of v. E.g., in Fig. 6 indices a and f are redundant indices for $T1$. Let *redundant(v)* be the set of redundant indices for v.

Introducing a redundant loop index i to a node v can allow additional fusion between v and its parent, which reduces the dimension of the intermediate array holding the result of v, in exchange for recomputing v in every iteration of the i loop. The space-time trade-off algorithm computes for every combination of redundant indices the loop fusion structure that results in the least amount of total memory.

In a bottom-up traversal, we compute a set of solutions for each node v. Each solution consists of a *nesting* of the loops at v, the memory cost mc, the recomputation cost rc, and pointers to the solutions of the subtrees from which this solution was obtained. A nesting is a sequence of index sets that represents constraints on the loop structure for computing v. E.g., the nesting $\langle ij, k\rangle$ indicates that the loops i and j can be arbitrarily permuted, while k must be nested inside of i and j. A solution s' is inferior to solution s if its nesting is more constraining than that of s (e.g., $\langle i, j, k\rangle$ is more constraining than $\langle ij, k\rangle$), and if its memory cost and recomputation cost are both higher than those of s. The set of solutions for a node is recursively computed as follows:

- Suppose v is an array reference of the form a`[`i`]`. The set of possible loops around the array node is *fusible(t)* $\cup$ powerset(*redundant(t)*) with no constraints on the order of the loops. For the purpose of space-time trade-offs, we do not model the cost of reading arrays from disk. Therefore, we form a solution for each of these nestings with zero memory and recomputation costs.

- Suppose v is of the form $\texttt{f(}i\texttt{)}$. Similar as for array references, we form a set of solutions for all possible nestings. For each nesting h, we initialize the memory cost to the storage needed for holding the result of $\texttt{f(}i\texttt{)}$ if all the indices in h are fused with the parent. The recomputation cost is initialized to the number of times f must be recomputed for all redundant indices in h times the cost of a function call.

- Suppose v is of the form $\texttt{sum(}i\texttt{,}t\texttt{)}$. For each solution s for subtree t, we initialize a solution s' for the summation node by adding one to the memory cost (for the scalar holding the result of the summation assuming full fusion with the parent) and by adding the recomputation cost for the summation node to that of the subtree. We then remove the summation indices i from the nesting in s'. All indices that are constrained to be nested inside the summation indices must be removed as well since they cannot be fused with the parent anymore. Removing a non-summation index j from the nesting results in an increase in memory since the j dimension of the resulting array must be stored. Finally, inferior solutions are pruned from the set of solutions for v.

- Suppose v is of the form $\texttt{bin(}o\texttt{,}l\texttt{,}r\texttt{)}$. Since the subtrees l and r might not have all the indices of v (*indices(v)* is the union of *indices(l)* and *indices(r)*), we first need to compute all the possible ways in which the solutions for l and r might be fused with v. For each solution s for a subtree, we compute the set of all prefixes of the nesting of s (e.g., for the nesting $\langle ij, k\rangle$, the prefix $\langle i\rangle$ represents the loop structure in which only i is fused with v). For all the nestings obtained in this way we construct new solutions for the subtrees by increasing the memory cost by the array dimensions that now need to be stored. Then, for all pairs of solutions s_l and s_r for l and r, respectively, we merge the constraints on the loop structure from the nestings of s_l and s_r. If s_l and s_r have compatible nestings, we obtain a merged nesting for v. E.g., for the nestings $\langle ij, kl\rangle$ and $\langle i, jk\rangle$ for the subtrees, we would obtain the nesting $\langle i, j, k, l\rangle$ for v. Finally, we construct solutions for v out of the merged nestings by adding the memory and recomputation costs for v to the costs for the subtrees and then prune inferior solutions.

The result of the above algorithm is a set of solution trees for the original expression tree. A solution tree contains a nesting and memory and recomputation costs for each tree node of the expression tree. For each node v, the nesting for v only reflects constraints on the loop structure for the subtree rooted at v. From a solution tree we compute a fusion tree by propagating constraints on loop nestings from the top of the tree down to the leaves. The resulting fusion tree is then translated into an abstract syntax tree by constructing a computation order for the tree nodes. A node v is computed after its subtrees. For a binary node, the subtree with the most loops fused is computed just before the parent. After the computation order is determined, the loops are inserted to form an abstract syntax tree representation of the code. For example, for the expression tree corresponding to the formula sequence in Fig. 1(a) this algorithm constructs the pseudo-code in Fig. 8 as the solution with the minimal recomputation cost that stays below 10^12 words.

6 Partial Expansion of Reduced Intermediates

Once a set of optimal solutions is determined by the first step of the space-time trade-off algorithm, we resort to array expansion for the second step. The second step operates on the abstract syntax tree generated by the first step of the algorithm. In this tree, an interior node represents a loop nest, while a leaf represents the computation of a node from the expression tree. A parent-child pair of nodes denotes an outer-inner loop pair, whereas nodes with the same parent represent adjacent loops. For an example, the abstract syntax tree corresponding to the pseudo-code in Fig. 8 is shown in Fig. 9.

The total number of operations needed to compute the final result is the sum over the number of operations for the leaves of the abstract syntax tree. For each leaf, the number of operations is obtained by multiplying the cost of the operation (one for multiplications or additions, a higher cost for function evaluations) by the loop ranges of all its ancestors in the abstract syntax tree. For example, the number of operations required to compute X in Fig. 9 is $2N_cN_aN_fN_eN_iN_j = 2O^2V^4$ operations (the factor of 2 comes from one multiplication and one addition). Likewise, the number of operations necessary to compute $T2$ is $1000N_cN_aN_fN_bN_k = 1000OV^4$ operations, assuming 1000 floating point operations are needed for each evaluation of $f2$. In the case of X the number of operations cannot be further reduced. There is no redundant cost in computing X. In contrast, $T2$ is repeatedly computed $N_c = V$ times, since c is a redundant loop for $T2$. For the pseudo-code presented in Fig. 8 and the corresponding abstract syntax tree in Fig. 9, the recomputation cost is $1000(N_c - 1)N_aN_fN_bN_k = 1000OV^3(V - 1)$, coming entirely from the evaluation of $T2$.

```
E = 0
for c
  for b,e,k
    T1[b,e,k] = f1(c,e,b,k)
  for a,f
    for e
      Y[e] = 0
    for b,k
      T2 = f2(a,f,b,k)
      for e
        Y[e] += T2 * T1[b,e,k]
    for e
      X = 0
      for i,j
        X += T[i,j,a,e] * T[i,j,c,f]
      E += Y[e] * X
return E
```

Figure 8: Pseudo-code for the solution with the lowest recomputation cost after the first step of the algorithm, subject to a memory limit of 10^{12} words. The array sizes are $N_i = N_j = N_k = O = 100$ and $N_a = N_b = N_c = N_e = N_f = V = 3000$. The redundant evaluation of $f2(a, f, b, k)$ is performed $N_c = V = 3000$ times.

Figure 9: Abstract syntax tree for the fused loop structure shown in Fig. 8. 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). The leaves of the tree are multiplication and addition computations or function evaluations.

```
E = 0
for c_t
  for c_i
    c = c_i + c_t * NB
    for b,e,k
      T1[c_i,b,e,k] = f1(c,e,b,k)
  for a,f
    for e,c_i
      Y[c_i,e] = 0
    for b,k
      T2 = f2(a,f,b,k)
      for e,c_i
        Y[c_i,e] += T2 * T1[c_i,b,e,k]
    for c_i
      c = c_i + c_t * NB
      for e
        X[c_i] = 0
        for i,j
          X[c_i] += T[i,j,a,e] * T[i,j,c,f]
        E += Y[c_i,e] * X[c_i]
return E
```

Figure 10: Pseudo-code for the solution with the lowest recomputation cost after the second step of the algorithm, subject to a memory limit of 10^{12} words. The c loop is split into the tiling and intra-tile loops c_t and c_i. The ranges of these loops are N_B and B, respectively. B is the block size, and N_B is the number of blocks. Their product is equal to the original range $N_c = V = 3000$ of the c loop. The arrays $T1$, Y and X are partially expanded from size one to size B along the c dimension. The evaluation of $f2(a, f, b, k)$ is performed N_B times.

In practice, the intermediate arrays do not have to be fully down-sized to a lower number of dimensions. For example, the solution in Fig. 8 uses only 9.0×10^8 words, much less than the memory limit of 10^{12} words. We can therefore increase the sizes of some intermediate arrays in order to reduce the recomputation cost. In our algorithm, each redundant node in the tree is split into a parent-child pair, corresponding to a tiling loop node, and an intra-tile loop node. Figures 10 and 11 present the pseudo-code and abstract syntax tree for the same computation, this time performed with loop tiling. In this example, the root of the abstract syntax tree c is the only redundant loop, but in general the number of redundant loops could be as large as the number of nodes in the abstract syntax tree. Here the c loop is split into the tiling and intra-tile loops c_t and c_i. The ranges of these loops are N_B (the number of blocks) and B (the block size), respectively, such that their product is equal to the original range: $B \times N_B = N_c = V = 3000$. The arrays $T1$, Y and X are partially expanded from size one to size B along the c dimension. The redundant computation of $T2$ is now only performed N_B times instead of N_c times, resulting in a lower recomputation cost. The maximum value for the block size B is determined by the total amount of memory available in the system.

Our algorithm for determining the best choice for array expansion (the one that minimizes recomputation cost, and still stays within the total amount of memory available) proceeds as follows: for a given untiled abstract syntax tree generated in the first step (Fig. 9), all its redundant nodes are first split into tiling/intra-tile pairs. Subsequently, the resulting abstract syntax tree is transformed by intra-tile loop permutation and fission into an equivalent abstract syntax tree with the property that each intra-tile loop is either redundant or non-redundant with respect to all of its descendant leaves. At this point those intra-tile loops which are redundant with respect to their descendant leaves are removed.

Figures 10 and 11 show the pseudo-code and abstract syntax tree after such a transformation. The c_i loop is split into three loops along different branches of the tree. It is present as an ancestor of all the leaves except for the one that produces $T2$, where it has been removed to reduce the recomputation cost. After this tree transformation the algorithm proceeds by choosing numerical values for the tile sizes, thus fixing the loop ranges for all the nodes in the abstract syntax tree. If the original range of a loop is N_c, choosing a block size B for the intra-tile loop also fixes the range $N_B = N_c/B$ of the tiling loop.

We thus obtain a new abstract syntax tree with well-defined loop ranges. Using the loop ranges, we can determine the recomputation cost for the entire abstract syntax tree by adding the number of redundant operations for each leaf

Figure 11: Abstract syntax tree for the fused and tiled loop structure shown in Fig. 10. The c loop is redundant for the leaf evaluating $f2(a, f, b, k)$, resulting in a large recomputation cost. To improve upon that, the redundant loop c is split into a tiling/intra-tile pair of loops (c_t and c_i, respectively). The intra-tile loops c_i are then moved by fission and permutation operations toward the bottom of the tree. The c_i loop is finally discarded for the leaf computing $f2(a, f, b, k)$. The remaining c_i loops are indicated by empty circles.

of the tree. With this approach, we arrive at a total recomputation cost for the abstract syntax tree for given tile sizes. We repeat the calculation of the recomputation cost for different sets of tile sizes. We define our tile size search space in the following way: if N_i is the loop range of a recomputation loop, we use a tile size starting from $B = 1$ (no tiling), and successively increasing B by doubling it until it reaches N_i. This ensures a slow (logarithmic) growth of the search space for increasing values of N_i. If N_i is small enough, an exhaustive search can instead be performed.

This tiling procedure and search for the optimal tile sizes is repeated for all solutions produced by the first step of the algorithm. We finally choose the solution with the minimal recomputation cost.

7 Results

In this section we present the results of our two step space-time trade-off algorithm for the NWChem example introduced in Section 3. We choose input parameters relevant to the addressed problem: $N_i = N_j = N_k = O = 100$, $N_a = N_b = N_c = N_e = N_f = V = 3000$, function evaluation cost $C_i = 1000$ floating point operations and available memory of $M = 10^{12}$ words.

Figure 5 shows the pseudo-code for the solution that was manually optimized by a domain expert.[1] The a, c, e, and f loops are split into tiling and intra-tile loops of size N_B and B, respectively. They obey the constraint $B \times N_B = V$. The largest intermediate array is Y, which is a four-dimensional block of size B^4. The recomputation cost of this solution is $2C_iOV^3(V^2/B^2 - 1)$. Requiring that the total memory usage is less than $M = 10^{12}$ words, and using the values for O, V and C_i provided in the previous paragraph, we arrive at a recomputation cost of $\approx 5.1 \times 10^{16}$ operations. The recomputation cost is due to the redundant evaluation of the functions $f1$ and $f2$ N_B^2 times.

The optimal solution is obtained using the two step space-time trade-off algorithm presented in Sections 5 and 6. The first step produces six solutions. All other possible loop fusion structures have both higher memory usage and higher recomputation cost than one or more of these solutions. Figure 12 shows the six solutions ranging in memory usage from three words to 2.7×10^{15} words, and in recomputation cost from zero operations to 4.9×10^{22} operations. The memory limit in our example is marked by the solid horizontal line. Solution number 1 is trivial, and represents the memory optimal solution with no redundant computation. Such a solution always exists for any operator tree. If its memory usage is below the memory limit, then the second step of the algorithm is no longer necessary, and this becomes the optimal final solution. Otherwise, it is discarded, along with all the other solutions that are above the memory limit (in this case, only number 1). The rest of the solutions (2 through 6 in this example) are then passed through the second step of the algorithm. Figure 8 shows the pseudo-code for solution 2, which has the lowest recomputation cost ($\approx 8.1 \times 10^{18}$ floating point operations) after the first step of the algorithm.

[1] The NWChem code also contains code for transforming integrals from the atomic basis into the molecular basis. This transformation is encapsulated in the function calls.

Figure 12: Relationship between memory usage and recomputation cost. Solid triangles represent the 6 different solutions produced by the first step of the space-time trade-off algorithm. The horizontal line shows the hard memory limit of $M = 10^{12}$ words used for this example. Except for solution 1, which uses more memory than the 10^{12} words limit, all the other solutions are analyzed by the second step of the algorithm.

The array expansion step brings significant further reduction of the recomputation cost for all the remaining 5 solutions. Their recomputation costs, ranging from 8.1×10^{18} to 4.9×10^{22} operations after step 1, are reduced to between 5.4×10^{15} and 5.1×10^{16} operations. The pseudo-code for the final optimal solution is presented in Fig. 10. It happens to be the tiled form of solution 2, which was the best solution before the array expansion step. However, this is just a coincidence, due in part to the very small operator tree considered for this example, which in turn generates a very limited number of solutions. In general, any of the solutions obtained in step one could become the optimal solution after tiling.

We note that the final solution is not trivial, in fact it has a rather complex structure. We also observe that, although their cost is similar, all the solutions (the tiled versions of 2 through 6) have abstract syntax trees that are quite different. Indeed, even for a relatively simple formula, like the one used in this example, the collection of solutions is rather rich and non-trivial. Manual optimization is unlikely to find and test all possibilities, especially for larger trees. It is also interesting to note that one of the solutions produced by the algorithm (the tiled version of 6) is identical to the manually optimized pseudo-code presented in Fig. 5. Its recomputation cost of 5.1×10^{16} operations is roughly one order of magnitude higher than the cost of the optimal solution.

We investigate the recomputation cost of the optimal code in comparison with that of the manually generated code for various values of the input parameters O, V, and M, consistent with their physical meaning. We find, as expected, that the structure of the optimal code may change from one set of input parameters to another. The improvement factor over the manual code presented in Fig. 5 ranges from 1 (when the manual code is optimal) to 20, depending on O, V, and M.

8 Conclusion

This paper describes 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 provides an overview of the planned synthesis system and presents a new optimization approach that can serve as the basis for a key component of the system for performing space-time trade-off optimizations.

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 [19]. 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 [8], the ATLAS [18] project for deriving efficient implementation of BLAS routines, and the PHIPAC [2] and TUNE [17] projects.

The paper describes a two step algorithm for the space-time trade-off optimization problem, and presents the results of applying it to a test case from the quantum chemistry code NWChem. The solution derived using our implementation of the algorithm reduces the recomputation cost of the coupled cluster calculation by a factor of 10 for typical problem sizes.

References

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

[2] 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.

[3] D. Cociorva, J. Wilkins, C.-C. Lam, G. Baumgartner, P. Sadayappan, and J. Ramanujam. Loop optimization for a class of memory-constrained computations. *Proc. 15th ACM International Conference on Supercomputing (ICS'01),* Sorrento, Italy, June 2001.

[4] 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. To appear in the *Proceedings of the International Conference on High Performance Computing*, December 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] High Performance Computational Chemistry Group. NWChem, A computational chemistry package for parallel computers, Version 3.3, 1999. Pacific Northwest National Laboratory, Richland, WA 99352.

[7] M. S. Hybertsen and S. G. Louie. Electronic correlation in semiconductors and insulators: band gaps and quasiparticle energies. *Phys. Rev. B*, 34:5390, 1986.

[8] Ken 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.

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

[10] C. Lam, D. Cociorva, G. Baumgartner and P. Sadayappan. Optimization of Memory Usage and Communication Requirements for a Class of Loops Implementing Multi-Dimensional Integrals. *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA, August 1999.

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

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

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

[14] 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.

[15] 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.

[16] 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.

[17] M. Thottethodi, S. Chatterjee, and A. Lebeck. Tuning Strassen's matrix multiplication for memory hierarchies. In *Proc. Supercomputing '98*, Nov. 1998.

[18] R. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software (ATLAS). In *Proceedings SC '98* (Electronic Publication), IEEE Publication, 1998.

[19] J. Xiong, D. Padua, and J. Johnson. SPL: A language and compiler for DSP algorithms. *ACM SIGPLAN PLDI*, June 2001.