# Memory-Optimal Evaluation of Expression Trees Involving Large Objects ⋆

Chi-Chung Lam[1], Daniel Cociorva[2], Gerald Baumgartner[1], and P. Sadayappan[1]

[1] Department of Computer and Information Science
The Ohio State University, Columbus, OH 43210
{clam,gb,saday}@cis.ohio-state.edu
[2] Department of Physics
The Ohio State University, Columbus, OH 43210
cociorva@pacific.mps.ohio-state.edu

**Abstract.** The need to evaluate expression trees involving large objects arises in scientific computing applications such as electronic structure calculations. Often, the tree node objects are very large that only a subset of them can fit in memory at a time. This paper addresses the problem of finding an evaluation order of nodes in a given expression tree that uses the least memory. We develop an efficient algorithm that finds an optimal evaluation order in $O(n^2)$ time for an $n$-node expression tree.

## 1 Introduction

This paper addresses the problem of finding an evaluation order of the nodes in a given expression tree that minimizes memory usage. The expression tree must be evaluated in some bottom-up order, i.e., the evaluation of a node cannot precede the evaluation of any of its children. The nodes of the expression tree are large data objects whose sizes are given. If the total size of the data objects is so large that they cannot all fit into memory at the same time, space for the data objects has to be allocated and deallocated dynamically. Due to the parent-child dependence relation, a data object cannot be deallocated until its parent node data object has been evaluated. The objective is to minimize the maximum memory usage during the evaluation of the entire expression tree.

This problem arises, for example, in optimizing a class of loop calculations implementing multi-dimensional integrals of the products of several large input arrays that computes the electronic properties of semiconductors and metals [2, 3, 9]. The multi-dimensional integral can be represented as an expression tree in which the leaf nodes are input arrays, the internal nodes are intermediate arrays, and the root is the final integral. In previous work, we have addressed the problems of 1) finding an expression tree with the minimal number of arithmetic operations and 2) the mapping of the computation on parallel computers to minimize the amount of inter-processor communication [4–6]. However, in practice, the input arrays and the intermediate arrays are often so large that they cannot all fit into available memory. There is a need to allocate array

⋆ Supported in part by the National Science Foundation under grant DMR-9520319.

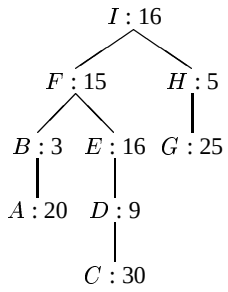



**Fig. 1.** An example expression tree

space dynamically and to evaluate the arrays in an order that uses the least memory. Solving this problem would help the automatic generation of code that computes the electronic properties. We believe that the solution we develop here may have potential applicability to other areas such as database query optimization and data mining.

As an example of the memory usage optimization problem, consider the expression tree shown in Fig. 1. The size of each data object is shown alongside the corresponding node label. Before evaluating a data object, space for it must be allocated. That space can be deallocated only after the evaluation of its parent is complete. There are many allowable evaluation orders of the nodes. One of them is the post-order traversal $\langle A, B, C, D, E, F, G, H, I\rangle$ of the expression tree. It has a maximum memory usage of 45 units. This occurs during the evaluation of $H$, when $F$, $G$, and $H$ are in memory. Other evaluation orders may use more memory or less memory. Finding the optimal order $\langle C, D, G, H, A, B, E, F, I\rangle$, which uses 39 units of memory, is not trivial.

A simpler problem related to the memory usage minimization problem is the register allocation problem in which the sizes of all nodes are unity. It has been addressed in [8, 10] and can be solved in $O(n)$ time, where $n$ is the number of nodes in the expression tree. But if the expression tree is replaced by a directed acyclic graph (in which all nodes are still of unit size), the problem becomes NP-complete [11]. The algorithm in [10] for expression trees of unit-sized nodes does not extend directly to expression trees having nodes of different sizes. Appel and Supowit [1] generalized the register allocation problem to higher degree expression trees of arbitrarily-sized nodes. However, the problem they addressed is slightly different from ours in that, in their problem, space for a node is not allocated during its evaluation. Also, they restricted their attention to solutions that evaluate subtrees contiguously, which is sub-optimal in some cases. We are not aware of any existing algorithm to the memory usage optimization problem considered in this paper.

The rest of this paper is organized as follows. In Section 2, we formally define the memory usage optimization problem and make some observations about it. Section 3 presents an efficient algorithm that solves the problem in $O(n^2)$ time for an $n$-node expression tree. Section 4 provides conclusions. Due to space constraints, the correctness proof of the algorithm is omitted from this paper but can be found in [7].

## 2 Problem Statement

The problem addressed is the optimization of memory usage in the evaluation of a given expression tree, whose nodes correspond to large data objects of various sizes. Each data object depends on all its children (if any), and thus can be evaluated only after all its children have been evaluated. The goal is to find an evaluation order of the nodes that uses the least amount of memory. Since an evaluation order is also a traversal of the nodes, we will use these two terms interchangeably. Space for data objects is dynamically allocated or deallocated under the following assumptions:

1. Each object is allocated or deallocated in its entirety.
2. Leaf node objects are created or read in as needed.
3. Internal node objects must be allocated before their evaluation begins.
4. Each object must remain in memory until the evaluation of its parent is completed.

We define the problem formally as follows:

> Given a tree $T$ and a size $v.\textit{size}$ for each node $v \in T$, find a computation of $T$ that uses the least memory, i.e., an ordering $P = \langle v_1, v_2, \ldots, v_n \rangle$ of the nodes in $T$, where $n$ is the number of nodes in $T$, such that
> 1. for all $v_i, v_j$, if $v_i$ is the parent of $v_j$, then $i > j$; and
> 2. $\max_{v_i \in P}\{\text{himem}(v_i, P)\}$ is minimized, where

$$\text{himem}(v_i, P) = \text{lomem}(v_{i-1}, P) + v_i.\textit{size}$$

$$\text{lomem}(v_i, P) = \begin{cases} \text{himem}(v_i, P) - \sum_{\{\text{child } v_j \text{ of } v_i\}} v_j.\textit{size} & \text{if } i > 0 \\ 0 & \text{if } i = 0 \end{cases}$$

Here, himem$(v_i, P)$ is the memory usage during the evaluation of $v_i$ in the traversal $P$, and lomem$(v_i, P)$ is the memory usage upon completion of the same evaluation. In general, we need to allocate space for $v_i$ before its evaluation. After $v_i$ is evaluated, the space allocated to all its children may be released. For instance, consider the post-order traversal $P = \langle A, B, C, D, E, F, G, H, I \rangle$ of the expression tree shown in Fig. 1. During and after the evaluation of $A$, $A$ is in memory. So, himem$(A, P) =$ lomem$(A, P) = A.\textit{size} = 20$. To evaluate $B$, we need to allocate space for $B$, thus himem$(B, P) =$ lomem$(A, P) + B.\textit{size} = 23$. After $B$ is obtained, $A$ can be deallocated, giving lomem$(B, P) =$ himem$(B, P) - A.\textit{size} = 3$. The memory usage for the rest of the nodes is determined similarly and shown in Fig. 2(a).

However, the post-order traversal of the given expression tree is not optimal in memory usage. For this example, none of the traversals that visit all nodes in one subtree before visiting another subtree is optimal. For the given expression tree, there are four such traversals. They are $\langle A, B, C, D, E, F, G, H, I \rangle$, $\langle C, D, E, A, B, F, G, H, I \rangle$, $\langle G, H, A, B, C, D, E, F, I \rangle$, and $\langle G, H, C, D, E, A, B, F, I \rangle$. If we follow the traditional wisdom of visiting the subtree that uses more memory first, we obtain the best of the four traversals, which is $\langle G, H, C, D, E, A, B, F, I \rangle$. Its overall memory usage is 44 units, as shown in Fig. 2(b), and is not optimal. The optimal traversal is $\langle C, D, G, H, A, B, E, F, I \rangle$, which uses 39 units of memory (see Fig. 2(c)). Notice that it 'jumps' back and forth between the subtrees. Therefore, any algorithm that only considers traversals that visit subtrees contiguously may not produce an optimal solution.

| Node | himem | lomem |
|---|---|---|
| $A$ | 20 | 20 |
| $B$ | 23 | 3 |
| $C$ | 33 | 33 |
| $D$ | 42 | 12 |
| $E$ | 28 | 19 |
| $F$ | 34 | 15 |
| $G$ | 40 | 40 |
| $H$ | 45 | 20 |
| $I$ | 36 | 16 |
| max | 45 | |

(a) Post-order traversal

| Node | himem | lomem |
|---|---|---|
| $G$ | 25 | 25 |
| $H$ | 30 | 5 |
| $C$ | 35 | 35 |
| $D$ | 44 | 14 |
| $E$ | 30 | 21 |
| $A$ | 41 | 41 |
| $B$ | 44 | 24 |
| $F$ | 39 | 20 |
| $I$ | 36 | 16 |
| max | 44 | |

(b) A better traversal

| Node | himem | lomem |
|---|---|---|
| $C$ | 30 | 30 |
| $D$ | 39 | 9 |
| $G$ | 34 | 34 |
| $H$ | 39 | 14 |
| $A$ | 34 | 34 |
| $B$ | 37 | 17 |
| $E$ | 33 | 24 |
| $F$ | 39 | 20 |
| $I$ | 36 | 16 |
| max | 39 | |

(c) The optimal traversal

**Fig. 2.** Memory usage of three different traversals of the expression tree in Fig. 1

One possible approach to the memory usage optimization problem is to apply dynamic programming on an expression tree as follows. Each traversal can be viewed as going through a sequence of configurations, each configuration being a set of nodes that have been evaluated (which can be represented more compactly as a smaller set of nodes in which none is an ancestor or descendant of another). In other words, the set of nodes in a prefix of a traversal forms a configuration. Common configurations in different traversals form overlapping subproblems. A configuration can be formed in many ways, corresponding to different orderings of the nodes. The optimal way to form a configuration $Z$ containing $k$ nodes can be obtained by minimizing over all valid configurations that are $k-1$-subsets of $Z$. By finding the optimal costs for all configurations in the order of increasing number of nodes, we get an optimal traversal of the expression tree. However, this approach is inefficient in that the number of configurations is exponential in the number of nodes.

The memory usage optimization problem has an interesting property: an expression tree or a subtree may have more than one optimal traversal. For example, for the subtree rooted at $F$, the traversals $\langle C, D, E, A, B, F\rangle$ and $\langle C, D, A, B, E, F\rangle$ both use the least memory space of 39 units. One might attempt to take two optimal subtree traversals, one from each child of a node $X$, merge them together optimally, and then append $X$ to form a traversal for $X$. But, this resulting traversal may not be optimal for $X$. Continuing with the above example, if we merge together $\langle C, D, E, A, B, F\rangle$ and $\langle G, H\rangle$ (which are optimal for the subtrees rooted at $F$ and $H$, respectively) and then append $I$, the best we can get is a sub-optimal traversal $\langle G, H, C, D, E, A, B, F, I\rangle$ that uses 44 units of memory (see Fig. 2(b)). However, the other optimal traversal $\langle C, D, A, B, E, F\rangle$ for the subtree rooted at $F$ can be merged with $\langle G, H\rangle$ to form $\langle C, D, G, H, A, B, E, F, I\rangle$ (with $I$ appended), which is an optimal traversal of the entire expression tree. Thus, locally optimal traversals may not be globally optimal. In the next section, we present an efficient algorithm that finds traversals which are not only locally optimal but also globally optimal.

## 3 An Efficient Algorithm

We now present an efficient divide-and-conquer algorithm that, given an expression tree whose nodes are large data objects, finds an evaluation order of the tree that minimizes the memory usage. For each node in the expression tree, it computes an optimal traversal for the subtree rooted at that node. The optimal subtree traversal that it computes has a special property: it is not only locally optimal for the subtree, but also globally optimal in the sense that it can be merged together with globally optimal traversals for other subtrees to form an optimal traversal for a larger subtree which is also globally optimal. As we have seen in Section 2, not all locally optimal traversals for a subtree can be used to form an optimal traversal for a larger tree.

The algorithm stores a traversal not as an ordered list of nodes, but as an ordered list of indivisible units called elements. Each element contains an ordered list of nodes with the property that there necessarily exists some globally optimal traversal of the entire tree wherein this sequence appears undivided. Therefore, as we show later, inserting any node in between the nodes of an element does not lower the total memory usage. An element initially contains a single node. But as the algorithm goes up the tree merging traversals together and appending new nodes to them, elements may be appended together to form new elements containing a larger number of nodes. Moreover, the order of indivisible units in a traversal stays invariant, i.e., the indivisible units must appear in the same order in some optimal traversal of the entire expression tree. This means that indivisible units can be treated as a whole and we only need to consider the relative order of indivisible units from different subtrees.

Each element (or indivisible unit) in a traversal is a (*nodelist*, *hi*, *lo*) triple, where *nodelist* is an ordered list of nodes, *hi* is the maximum memory usage during the evaluation of the nodes in *nodelist*, and *lo* is the memory usage after those nodes are evaluated. Using the terminology from Section 2, *hi* is the highest himem among the nodes in *nodelist*, and *lo* is the lomem of the last node in *nodelist*. The algorithm always maintains the elements of a traversal in decreasing *hi* and increasing *lo* order, which implies in order of decreasing *hi*-*lo* difference.

Fig. 3 shows the algorithm. The input to the algorithm (the **MinMemTraversal** procedure) is an expression tree $T$, in which each node $v$ has a field $v.$*size* denoting the size of its data object. The procedure performs a bottom-up traversal of the tree and, for each node $v$, computes an optimal traversal $v.$*seq* for the subtree rooted at $v$. The optimal traversal $v.$*seq* is obtained by optimally merging together the optimal traversals $u.$*seq* from each child $u$ of $v$, and then appending $v$. At the end, the procedure returns a concatenation of all the *nodelists* in $T.$*root.seq* as the optimal traversal for the given expression tree. The memory usage of the optimal traversal is $T.$*root.seq*$[1].$*hi*.

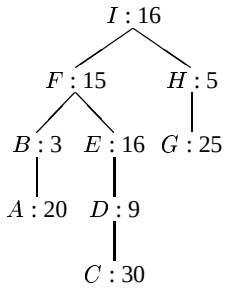
The **MergeSeq** procedure merges two given traversals $S1$ and $S2$ optimally and returns the merged result $S$. $S1$ and $S2$ are subtree traversals of two children nodes of the same parent. The optimal merge is performed in a fashion similar to merge-sort. Elements from $S1$ and $S2$ are scanned sequentially and appended into $S$ in the order of decreasing *hi*-*lo* difference. This order guarantees that the indivisible units are arranged to minimize memory usage. Since $S1$ and $S2$ are formed independently, the *hi*-*lo* values in the elements from $S1$ and $S2$ must be adjusted before they can be appended to $S$.

**MinMemTraversal** $(T)$:
  **foreach** node $v$ in some bottom-up traversal of $T$
    $v.seq = \langle\rangle$  // an empty list
    **foreach** child $u$ of $v$
      $v.seq =$ **MergeSeq**$(v.seq, u.seq)$
    **if** $|v.seq| > 0$ **then**  // $|x|$ is the length of $x$
      $base = v.seq[|v.seq|].lo$
    **else**
      $base = 0$
    **AppendSeq** $(v.seq, \langle v\rangle, v.size + base, v.size)$
  $nodelist = \langle\rangle$
  **for** $i = 1$ to $|T.root.seq|$
    $nodelist = nodelist + T.root.seq[i].nodelist$  // $+$ is the concatenation operator
  **return** *nodelist*  // memory usage is $T.root.seq[1].hi$

**MergeSeq** $(S1, S2)$:
  $S = \langle\rangle$
  $i = j = 1$
  $base1 = base2 = 0$
  **while** $i \leq |S1|$ or $j \leq |S2|$
    **if** $j > |S2|$ or ($i \leq |S1|$ and $S1[i].hi - S1[i].lo > S2[j].hi - S2[j].lo$) **then**
      **AppendSeq** $(S, S1[i].nodelist, S1[i].hi + base1, S1[i].lo + base1)$
      $base2 = S1[i].lo$
      $i$++
    **else**
      **AppendSeq** $(S, S2[j].nodelist, S2[j].hi + base2, S2[j].lo + base2)$
      $base1 = S1[j].lo$
      $j$++
  **end while**
  **return** $S$

**AppendSeq** ($S$, *nodelist, hi, lo*):
  $E =$ (*nodelist, hi, lo*)  // new element to append to $S$
  $i = |S|$
  **while** $i \geq 1$ and ($E.hi \geq S[i].hi$ or $E.lo \leq S[i].lo$)
    $E = (S[i].nodelist + E.nodelist, \max(S[i].hi, E.hi), E.lo)$  // $S[i]$ is combined into $E$
    remove $S[i]$ from $S$
    $i$−−
  **end while**
  $S = S + E$  // $|S|$ is now $i + 1$

**Fig. 3.** Procedure for finding an memory-optimal traversal of an expression tree




(a) The expression tree in Fig. 1

| Node $v$ | Optimal traversal $v$.*seq* |
|---|---|
| $A$ | $\langle(A, 20, 20)\rangle$ |
| $B$ | $\langle(AB, 23, 3)\rangle$ |
| $C$ | $\langle(C, 30, 30)\rangle$ |
| $D$ | $\langle(CD, 39, 9)\rangle$ |
| $E$ | $\langle(CD, 39, 9), (E, 25, 16)\rangle$ |
| $F$ | $\langle(CD, 39, 9), (ABEF, 34, 15)\rangle$ |
| $G$ | $\langle(G, 25, 25)\rangle$ |
| $H$ | $\langle(GH, 30, 5)\rangle$ |
| $I$ | $\langle(CDGHABEFI, 39, 16)\rangle$ |

(b) Optimal traversals for subtrees

**Fig. 4.** Optimal traversals for the subtrees in the expression tree in Fig. 1

The amount of adjustment for an element from $S1$ ($S2$) equals the $lo$ value of the last merged element from $S2$ ($S1$), which is kept in *base1* (*base2*).

The **AppendSeq** procedure appends a new element specified by *nodelist*, *hi*, and *lo* to the given traversal $S$. Before the new element $E$ is appended to $S$, it is combined with elements at the end of $S$ whose *hi* is not higher than $E$.*hi* or whose *lo* is not lower than $E$.*lo*. The combined element has the concatenated *nodelist* and the highest *hi* but the original $E$.*lo*. Elements are combined to form larger indivisible units.

To illustrate how the algorithm works, consider the expression tree shown in Fig. 1 and reproduced in Fig. 4(a). We visit the nodes in a bottom-up order. Since $A$ has no children, $A$.*seq* $= \langle(A, 20, 20)\rangle$ (for clarity, we write *nodelists* in a sequence as strings). To form $B$.*seq*, we take $A$.*seq* and append a new element $(B, 3 + 20, 3)$ to it. The **AppendSeq** procedure combines the two elements into one, leaving $B$.*seq* $= \langle(AB, 23, 3)\rangle$. Here, $A$ and $B$ form an indivisible unit, implying that $B$ must follow $A$ in some optimal traversal of the entire expression tree. Similarly, we get $E$.*seq* $= \langle(CD, 39, 9), (E, 25, 16)\rangle$. For node $F$, which has two children $B$ and $E$, we merge $B$.*seq* and $E$.*seq* by the order of decreasing *hi*-*lo* difference. So, the elements merged are first $(CD, 39, 9)$, then $(AB, 23 + 9, 3 + 9)$, and finally $(E, 25 + 3, 16 + 3)$ with the adjustments shown. They are the three elements in $F$.*seq* after the merge as no elements are combined so far. Then, we append to $F$.*seq* a new element $(F, 15 + 19, 15)$ for the root of the subtree. The new element is combined with the last two elements in $F$.*seq*. Hence, the final content of $F$.*seq* is $\langle(CD, 39, 9), (ABEF, 34, 15)\rangle$, which consists of two indivisible units. The optimal traversals for the other nodes are computed in the same way and are shown in Fig. 4(b). At the end, the algorithm returns the optimal traversal $\langle C, D, G, H, A, B, E, F, I\rangle$ for the entire expression tree (see Fig. 2(c)).

The time complexity of this algorithm is $O(n^2)$ for an $n$-node expression tree because the processing for each node $v$ takes $O(m)$ time, where $m$ is the number of nodes in the subtree rooted at $v$. Another feature of this algorithm is that the traversal it finds for a subtree $T'$ is not only optimal for $T'$ but must also appear as a subsequence in some optimal traversal for any larger tree that contains $T'$ as a subtree. For example, $E$.*seq* is a subsequence in $F$.*seq*, which is in turn a subsequence in $I$.*seq* (see Fig. 4(b)).

## 4 Conclusion

In this paper, we have considered the memory usage optimization problem in the evaluation of expression trees involving large objects of different sizes. This problem can be found in many practical applications such as scientific calculations, database query, and data mining, for which the data objects can be so large that it is impossible to keep all of them in memory at the same time. Hence, it is necessary to allocate and deallocate space for the data objects dynamically and to find an evaluation order that uses the least memory. We have proposed an efficient algorithm that finds an optimal evaluation in $O(n^2)$ time for an expression tree containing $n$ nodes. Also, we have described some interesting properties of the problem and the algorithm.

## References


1. A. W. Appel and K. J. Supowit. *Generalizations of the Sethi-Ullman algorithm for register allocation*, Software—Practice and Experience, 17 (6), pp. 417–421, June 1987.
2. W. Aulbur, *Parallel implementation of quasiparticle calculations of semiconductors and insulators*, Ph.D. Dissertation, Ohio State University, Columbus, October 1996.
3. M. S. Hybertsen and S. G. Louie, *Electronic correlation in semiconductors and insulators: band gaps and quasiparticle energies*, Phys. Rev. B, 34 (1986), pp .5390.
4. 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.
5. C. Lam, P. Sadayappan, and R. Wenger, *Optimization of a class of multi-dimensional integrals on parallel machines*, Eighth SIAM Conference on Parallel Processing for Scientific Computing, March 1997.
6. C. Lam, P. Sadayappan, D. Cociorva, M. Alouani, and J. Wilkins, *Performance optimization of a class of loops involving sums of products of sparse arrays*, Ninth SIAM Conference on Parallel Processing for Scientific Computing, March 1999.
7. C. Lam, *Performance optimization of a class of loops implementing multi-dimensional integrals*, Technical report no. OSU-CISRC-8/99-TR22, Dept. of Computer and Information Science, The Ohio State University, Columbus, August 1999.
8. I. Nakata, *On compiling algorithms for arithmetic expressions*, Comm. ACM, 10 (1967), pp. 492–494.
9. 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 (1995), pp. 1827.
10. R. Sethi, J. D. Ullman, *The generation of optimal code for arithmetic expressions*, J. ACM, 17(1), October 1970, pp. 715–728.
11. R. Sethi, *Complete register allocation problems*, SIAM J. Computing, 4(3), September 1975, pp. 226–248.