Efficient Storage Reuse of Aggregates in Single Assignment Languages

Zhonghua Li and Chris C. Kirkham

Department of Computer Science, Manchester University, Manchester, M13 9PL, UK
{zhonghua,chris}@cs.man.ac.uk

Abstract. The storage reuse of aggregates is a key problem in implementing single assignment languages. In this paper, on the basis of a typical subset of the single assignment language SISAL, we analyze the inherent limits of storage reuse and define what the maximal storage reuse is. We propose an efficient method of achieving storage reuse, which is of polynomial complexity and linear in common cases. It achieves the maximal storage reuse for an extensive program class into which all common benchmark programs fall. We also show that no general method can guarantee the maximal storage reuse for programs outside the class. In this case, our method can choose the most likely operation to reuse the storage of an aggregate or a set of shared aggregates.

1 Introduction

The demand for higher speed from computational users continues to increase. Parallel processing has been considered as an important way to higher performance. Unfortunately, because of its extra complexity, software technology has yet to match architectural advances. For the development of parallel software, a brand-new methodology is needed. Language is an important aspect of the methodology. There are two possible choices, one is to take existent languages such as Fortran, and the other is to develop new languages. Although there has been a large production of software in existent languages and also programmers are familiar with them, these languages have been designed inherently for sequential machines, and are not suitable for parallel computations. The greatest hindrance to parallelization of conventional languages is that programs always overspecify, and impose too many unnecessary sequencing constraints. On the other hand, applicative languages, in particular, single assignment languages such as SISAL [6] have emerged as a promising approach to parallel programming. Because of referential transparency, only data dependencies define sequencing in applicative programs. Therefore, the identification of concurrency is trivial.

But, for applicative languages, aggregate structures such as arrays pose a problem for their efficient implementation. Because of their semantics, an update operation generally requires copying the entire array, and the update at appropriate indices is made in the new copy. The old copy needs to be kept intact because it may be referenced by other subcomputations in the program.

This simple implementation of the update operation leads to inefficient use of storage, and degrades the performance of an algorithm. In the context of scientific computing, where the manipulation of large aggregates is common-place, the copying can become intolerable. However, for most programs, the resulting copying operations are only inherent to language semantics and not the algorithms themselves. Storage reuse is essential to their efficient implementation, saving time by not copying values, and saving space by not re-allocating storage.

In this paper, we discuss the storage reuse of aggregates in a sequential implementation of first-order single assignment languages with nested aggregates. There are two reasons for us to restrict our attention to "sequential": (1) there has been no satisfactory solution to this problem and, (2) storage reuse in a sequential implementation is also an important part of parallel implementation since a partial sequentialization of programs is essential for an efficient parallel implementation. We analyze the inherent limits of storage reuse, define what a maximal storage reuse is, and give an efficient method of achieving storage reuse.

2 Background

2.1 The single assignment language

Single assignment languages are value-oriented. Their semantics deals with functions on values rather than destructive operations on data objects residing in memory. Even arrays are treated as values. In this paper, we consider the storage reuse for a subset of SISAL, called *SL*. We ignore the SISAL data types: ***stream***, ***union*** and ***record***. These simplifications make the resulting language *SL* a first-order strict functional language with nested aggregates, and enable us to focus on the storage reuse of arrays without need to consider unrelated aspects. In this section, we introduce the main features of interest.

An *SL* function computes one or more output values as a function of one or more input values. A function has no side effects and retains no state information from one invocation to another. Hence the values returned by any invocation of an *SL* function depend only on the arguments. Since *SL* is a side-effect free language, subexpressions may be evaluated in any order without affecting the results, provided all data dependencies are satisfied. Language constructs are provided for conditional and iterative expressions, which implicitly contain control dependencies. An iterative expression, called *while loop*, has a set of initial value definitions, a corresponding set of redefinitions used to define new values on every iteration, a loop control and a set of resulting values defined in terms of the values of the names during all the iterations or only in the last iteration. The keyword *old* is used to refer to the value of a name in the previous iteration. *SL* also offers a *for* construct for expressing parallel iterations with no cross iteration dependencies, called *forall loop*.

SL includes only one kind of data structure: *array*. An array has an integer index set and its components are of arbitrary but uniform type. Arrays can be nested to any depth. The operations on an array can be classified into three

categories: initialization, reference, and modification. *array_fill*(l, h, v) creates an array with indices ranging from l to h and each element being equal to v. $A[j]$ selects the jth element of A. A typical array modification operation is $A[j : v]$, which produces a new value identical to A except that the element at the index j is substituted by v. A naive applicative implementation of the modification operation $A[j : v]$ requires construction of a new array almost identical to A. The time and storage requirements of this operation are linear to A's size. This is very expensive. A loop can greatly magnify the effect of an array modification in it. As [2] pointed out, the insertion sort program in a single assignment language may be several thousand times slower than the FORTRAN counterpart when sorting an array of 1000 floating point numbers.

2.2 The intermediate dataflow graph

Our discussions are based on the dataflow representation of a program which is similar to $IF1$[8]. A program can be described as a set of dataflow graphs with each representing a function and one of them being the main function graph where the computation starts. A **dataflow graph** is a quadruple (IP, OP, N, E) where IP, OP, N and E are a set of input ports, a set of output ports, a set of internal nodes and a set of arcs respectively. Nodes represent computations and arcs represent dependencies between them. For example, if there is an arc from node N_1 to N_2, then N_2 is dependent on N_1, i.e. N_1 must be finished before N_2 starts. We simply denote this as $N_1 \prec N_2$.

A node receives data from its input ports, fulfills a specific computation and puts the results to its output ports. There are mainly two kinds of nodes: atomic nodes and compound nodes. An atomic node represents an indivisible sequential computation. A compound node can embody a control structure to encapsulate a complex computation. It consists of a set of input ports, a set of output ports and a set of subgraphs. Typically, there are three kinds of compound nodes for representing an SL program, *conditional*, *product loop* and *non-product loop*, which correspond to a *conditional*, a *forall loop* and a *while loop* expression respectively. For example, a *conditional* node includes three subgraphs, the *condition*, *then* and *else* subgraphs corresponding to the predicate, *then* part and *else* part of a *conditional* expression respectively. There are control and data dependencies between subgraphs of a compound node. As shown in figure 1(a), we represent a node by a rectangle, an arc by an arrow and an aggregate port by a circle. Usually, we omit representations of other ports. We also omit representations of dependencies within a compound node unless we represent them by dashed arrows when necessary. For a *conditional* node, the upper subnode corresponds to the *predicate*, the left lower one to the *then* part and the right lower one to the *else* part as C1 and C2 in this figure.

2.3 Aggregates

An aggregate is a compound value consisting of simpler values, which are called components of the aggregate. In this paper, by an aggregate we mean an ar-

ray. The aggregate operations fall into three categories, that is, *E-operations* which reference a whole aggregate returning the same aggregate as a result, *R-operations* which reference individual components of an aggregate, and *W-operations* [1] which change some components of an aggregate. We denote an *E-operation* on A by $E(A)$. We introduce a typical *R-operation* $read(A, i)$ and a typical *W-operation* $replace(A, i, a)$ to represent the SL expressions $A[i]$ and $A[i : v]$ respectively.

An aggregate, say A, may be a component of another bigger aggregate B. In this case, B is called a super-aggregate or nested aggregate and A is called a sub-aggregate of B. The *R-operation* $read(A, i)$ on a super-aggregate A can be taken as an *E-operation* on the sub-aggregate $A[i]$. The *W-operation* $replace(B, i, A)$ where A is an aggregate can be taken as an *E-operation* on A and $B[j]$ for $j \neq i$. A naive implementation of the *E-operation* $E(A)$ is to copy A into a new physical space to get a new value equal to A. Obviously, this is an inefficient way. In the following discussion, we assume the more efficient model: $E(A)$ represents the aggregate which is, not only logically but also physically, the same as A, that is, *E-operations* are implemented through sharing the same physical space. A related problem is the passing method of aggregate parameters of a function. Semantically, the call-by-value passing method is used for a strict functional language. This requires the copying of aggregate parameter values. Instead, we use the more efficient call-by-reference passing method for aggregate parameters in the implementation. In the following discussions, we use A, B, A', B', with or without subscripts to indicate aggregates.

3 Limits of Storage Reuse

3.1 Compilation granules

There can be more than one operation on an aggregate. A *W-operation* can reuse the storage of an aggregate only if it is the last operation on the aggregate. There are static and dynamic methods to achieve storage reuse. The former is to statically order the operations without violation of dependencies so that a *W-operation* can be done last on an aggregate to reuse its storage. The latter is to make decisions at run time, typically through reference counting. Because the static method involves less overheads, and the dynamic method can not achieve effective storage reuse without the static method, ordering the compilation granules is a basic part of any practical method of achieving storage reuse.

By a compilation granule, we mean a node whose execution is not interleaved with any node outside it, or a function call node. The size of a compilation granule (more exactly, the size and structure of a compilation granule, since a compilation granule can include compilation granules as its components) can

[1] There are two kinds of *W-operations* in SISAL, incremental construction and incremental modification. In this paper, we only introduce the latter. But our storage reuse method takes both of them into account, though an extra phase is needed for pre-allocating enough storage for in-place incremental construction operations.

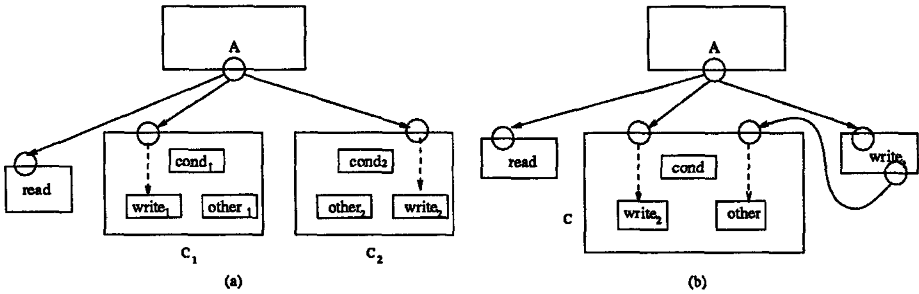

Fig. 1. Examples of conditional nodes

affect the degree to which the storage can be reused. In figure 1(a), C_1 and C_2 are two independent conditional nodes, and $write_1$ and $write_2$ are the *W-operations* on the aggregate A. $cond_1$ and $cond_2$ do not reference A and their values can vary with other input data. So, if a whole conditional node is a compilation granule, no method can guarantee that the storage of A can be reused reasonably efficiently. By "reasonably efficiently", we mean that the overheads from the reuse are less than the benefits from the reuse. On the other hand, if a conditional node can be decomposed into smaller compilation granules, e.g. the *condition* part, *then* part and *else* part, the storage of A can be reused through ordering these smaller compilation granules so that $cond_1$ and $cond_2$ are executed first and then making the further decision dynamically on the basis of their results as follows:

$cond_1$	$cond_2$	the decision
TRUE	TRUE	$write_1$ can reuse the storage
FALSE	FALSE	$write_2$ can reuse the storage
FALSE	TRUE	no *W-operation* is executed
TRUE	FALSE	both $write_1$ and $write_2$ will be executed the later one can reuse the storage

Generally, the smaller the compilation granules, the more opportunities for storage reuse through static ordering, but the compiler will be more complicated as the compilation granules become smaller. For simplicity of the compiler, the compound nodes, i.e. *conditional*, *product loop* and *non-product loop*, have been taken as compilation granules in some existing implementations[2]. In the rest of this paper, we make the same assumption. So, the control dependencies are encapsulated within compilation granules.

3.2 Aggregate operations and their degrees

We have described operations on an aggregate in §2.3. Generally, these operations are embedded in compilation granules. We are particularly interested in those outermost compilation granules including operations on an aggregate since

effectively ordering them is an important part of achieving storage reuse. It will be convenient to also call them operations on this aggregate. By contrast, we call those operations in §2.3 atomic operations. So, an aggregate operation can be atomic or compound. In the latter case, it can operate on more than one aggregate and can include atomic *R-*, *W-* and/or *E-operations*. Some operations are certain to operate on an aggregate, while others are not. For example, $A[1:2]$ definitely operates on A, and, *if* $x = 1$ *then* $A[1:2]$ *else B end if* can operate on aggregate A only when x is equal to 1. Generally, an operation can be arbitrarily complicated. In different operations, probabilities that atomic *R-*, *E-* or *W-operations* are to be executed are different. We can statically assign W-Degree(op, A) to operation *op* to represent the probability that an atomic *W-operation* can be executed later than any atomic *R-* or *E-operation* on *A*. When a node is executed, one of the following cases can occur for an aggregate:

1. An atomic *W-operation* on it is certain to be executed later than any possible atomic *R-operation* or *E-operation* on it.
2. An atomic *R-operation* or *E-operation* on it is certain to be executed later than any possible atomic *W-operation* on it.
3. No atomic *W-operation* is executed on it.
4. An atomic *W-operation* may be executed on it, and if this occurs, it is later than any *R-operation* or *E-operation* on it.
5. An atomic *W-operation* may be executed on it, and whether or not the operation will be later than any possible *R-operation* or *E-operation* on it can not be predicted.

In the first case, the W-Degree assigned is one. In the second and third cases, the W-Degree assigned is zero. In the last two cases, the W-Degree is evaluated to a value between zero and one. Similarly, we can define R-Degree(op, A) for an operation *op* on an input aggregate *A* to represent the probability that an atomic *R-operation* on A can be executed in the operation; define E-Degree(op, A, B) for an operation *op* on an input aggregate *A* to represent the probability that *A* can be directly used as an output *B* of the operation; and E-Degree(op, A)=$\sum_B$E-Degree(op, A, B).

More generally, we call an operation *op* an *R-operation* on *A* if *R-Degree(op, A)>0 and W-Degree(op, A)=E-Degree(op, A)=0*; call it a *W-operation* on A if *W-Degree(op, A)>0*; and call it an *E-operation* on A if there is an output aggregate *B*, $E\text{-}Degree(op, A, B) > 0$. If there is indeed an atomic *W-operation* that is executed later than any atomic *R-operation* or *E-operation* on *A* in an execution of the *W-operation*, we say that it *acts as a physical W-operation on A in the execution.* If *A* is indeed directly used as an output in an execution of the E-operation, we say that it *acts as a physical E-operation* on *A* in the execution.

3.3 Dependencies

Without sharing After the compilation granules are fixed as §3.1, the inherent limits of storage reuse depend on the data dependencies. For a *W-operation wop*

on A, if there is another operation op on A and $wop \prec op$, wop must be executed before op and can not reuse the storage of A. As illustrated in figure 1(b), *read*, $write_1$ and C are all the possible operations on aggregate A, and $write_1 \prec C$. $write_1$ can not reuse the storage of A, even if it is safe, e.g. when *cond* is FALSE, and $write_1$ is the only *W-operation* on A in the execution, since the safety of the reuse can not be justified in an execution generally. We call a *W-operation* a *reuse candidate* on an aggregate if no operation on the same aggregate is dependent on it. Only a *reuse candidate* of an aggregate can reuse the storage.

The difficulties of reuse come from not only the dependencies between the operations on the same aggregate but also those on different aggregates. In the latter case, we call them interferences between these aggregates. In figure 2(a), due to the mutual interferences of A and B, the storage of only one of them can be reused. This is because, for a *W-operation* on an aggregate to reuse the storage, we need to introduce extra *temporal dependencies* between it and all the *R-operations* on the aggregate. We denote that op_1 *temporally precedes* op_2 by $op_1 \lhd op_2$ (represented as a dashed arrow in a diagram). These extra dependencies should not result in cycles. But, here, the dependency cycle, $w_A \prec r_B \lhd w_B \prec r_A \lhd w_A$, is formed.

Generally, suppose that $w_0, \ldots, w_{n-1}$ are reuse candidates on the aggregates $A_0, \ldots, A_{n-1}$ respectively. Let $k' = (k+1) mod\ n$. If there is a permutation $i_0, \ldots, i_{n-1}$ of $0, \ldots, n-1$ such that $\forall k\ (0 \leq k < n)$ either w_{i_k} is also an *R-operation* on $A_{i_{k'}}$ $(w_{i_k} \lhd w_{i_{k'}})$, or there is an *R-operation* r on $A_{i_{k'}}$ and $w_{i_k} \prec r\ (\lhd w_{i_{k'}})$, we say that they form an interference cycle. If all reuse candidates on these aggregates are in some cycle, then the storage of one of them can not be reused. We call such a cycle a *complete interference cycle.* An example is shown in figure 2(b) where op_1 is the *W-operation* on A and the *R-operation* on B and op_2 is the *W-operation* on B and the *R-operation* on A. Here, $op_1 op_2$ form a complete interference cycle. In the other cases, the interference cycle can be broken as follows without hindrance to storage reuse.

- There is a *reuse candidate* wop on A_{i_l} which is not in any interference cycle for some l, $0 \leq l < n$. In this case, the cycle can be broken from w_{i_l} and we can schedule the above *W-operations* in the following order: $w_{i_l} \ldots w_{i_{n-1}} w_{i_0} \ldots w_{i_{l-1}} wop$ so that the storage of $A_j (0 \leq j < n \wedge j \neq i_l)$ can be reused by w_j and the storage of A_{i_l} can be reused by wop. For example, in figure 2(c), $w_A r_B w_B r_A$ is an interference cycle and $w1_A$ is not in any interference cycle. This cycle can be broken through scheduling nodes in the following order, $w_A r_B w_B r_A w1_A$.
- The cycle is adjacent to another interference cycle which can be broken. We say that two cycles are adjacent if there is a *W-operation* w_1 in one cycle and a *W-operation* w_2 in the other and w_1 and w_2 operate on the same aggregate. If one can be broken, the other can be broken, e.g. from w_2.

With sharing Semantically, an atomic *E-operation* $E(A)$ on the aggregate A produces a new aggregate equal to A, say A', but A' is mapped into the same

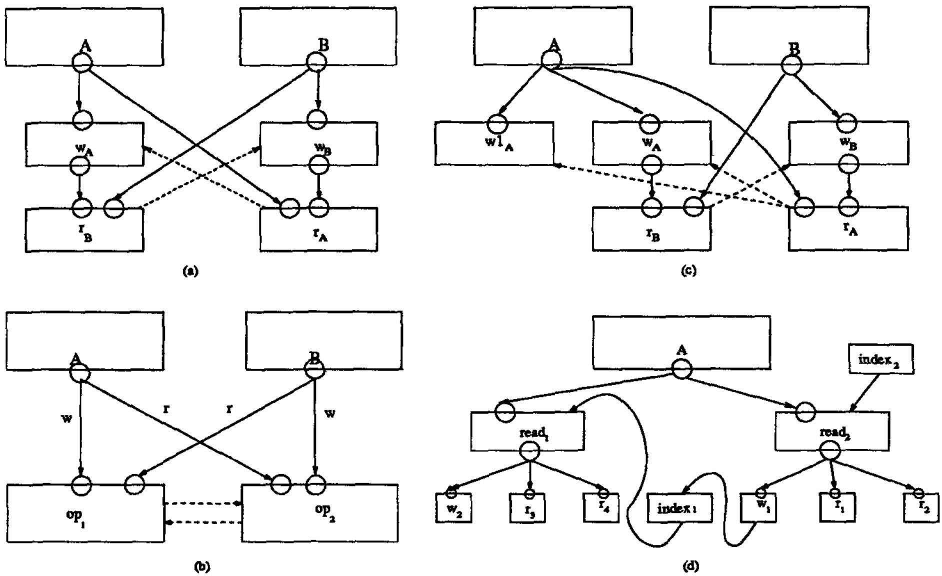

Fig. 2. Examples with dependencies

space as A instead of being allocated a new physical space. In this case, we call A and A' *inter-shared* or simply *shared.* Closely related to sharing is nesting of aggregates. We define the relation $\tilde{E}$ on $\mathcal{A}$, i.e. the set of all aggregates and sub-aggregates in a function, to describe the sharing among them. $\tilde{E}$ is the smallest equivalence relation satisfying the following condition:
For $A_1, A_2 \in \mathcal{A}$, if A_2 is generated through $E(A_1)$, $(A_1, A_2) \in \tilde{E}$.
$\tilde{E}$ is related to a specific execution of the program. For example, when *E-Degree(op, A, B)*$<$ 1, whether (A, B) $\in \tilde{E}$ depends on whether *op acts as a physical E-operation* on A in the execution. Each equivalence class on $\tilde{E}$ represents a set of inter-shared aggregates or sub-aggregates which are mapped to the same physical space. We call such a class an *aggregate cluster* or simply a *cluster.* Each non-shared aggregate comprises a cluster of only one element.

The sharing and nesting of aggregates make storage reuse even more complicated. We need to consider a kind of specific dependency, index dependency. As illustrated in figure 2(d), A is a nested aggregate; $read_2$ and $read_1$ are two *R-operations* which reference A by the inputs from $index_2$ and $index_1$ respectively; w_1, r_1 and r_2 are operations on the sub-aggregate referenced by $read_2$; w_2, r_3 and r_4 are operations on the sub-aggregate referenced by $read_1$; $index_1$ is dependent on w_1. Due to this dependency, when w_1 is executed, it is impossible to reasonably efficiently decide whether the sub-aggregates referenced by $read_1$ and $read_2$ belong to the same cluster, and whether there are more operations on the cluster. Therefore, it is impossible for w_1 to reasonably efficiently reuse

the storage of the sub-aggregate.

Generally, it is impossible for a *W-operation* *wop* on A_1 to reasonably efficiently reuse the storage of A_1 in the following cases:

- *wop* $\prec$ *op* where *op* is an operation on aggregate A_2 with $(A_1, A_2) \in \tilde{E}$; or
- *wop* $\prec$ *op.index* where *op* is an operation on the nested aggregate A_2 which includes a sub-aggregate (or a sub-aggregate of sub-aggregate, etc.) A_2' with $(A_1, A_2') \in \tilde{E}$, and *op.index* denotes the node whose output is used as the index of *op*. In this case, we say that *op* is *index-dependent* on *wop*.

Otherwise, we call *wop* a *reuse candidate* on the cluster which A_1 is in. Similarly, we can define the interferences and (complete) interference cycles between aggregate clusters, and a complete interference cycle can also affect the reuse of shared aggregates.

In general, in executing a function, there are three cases when the storage of an aggregate cluster AC can not be reused.

- There is no reuse candidate on AC.
- An aggregate of AC is an output of the function.
- An aggregate of AC is a parameter of the function, and this parameter can not be reused. (see §4.1)

We call the aggregate cluster AC non-reusable in these cases, otherwise reusable. Suppose for any function in a program, N_a is the total number of aggregate clusters, N_{nra} is the total number of non-reusable aggregate clusters and N_{cic} is the total number of complete interference cycles without any operation on non-reusable aggregate clusters. From the above analysis, we can conclude that, in executing the function, the *number* of aggregate clusters whose storage can be reasonably efficiently reused is not greater than $N_{mr} = N_a - N_{nra} - N_{cic}$. If the number of reused aggregate clusters is exactly equal to N_{mr}, we say that the maximal storage reuse is achieved.

4 The Method of Achieving Storage Reuse

In this section, we outline a storage reuse method. Statically, we order all nodes in each function so that the most likely *reuse candidate* is executed last on each aggregate. For shared aggregates, we can not, in general, statically determine whether such a *reuse candidate* is also the last one on the whole aggregate cluster. So, we introduce sharing counters for shared aggregates to dynamically capture the opportunities of storage reuse. In addition, we also need a dynamic mechanism to reuse the storage of an aggregate parameter of a function.

4.1 Dynamic mechanisms

1. A sharing counter is associated with the physical space of each aggregate cluster to reflect the number of aggregates mapped to it. An operation can

reuse the storage of the cluster only if (1) it is executed last on the current aggregate [2], and (2) the sharing count is 1.

2. The *E-operations* are implemented by incrementing the corresponding sharing-counter. After all the operations on an aggregate of this cluster are finished, the sharing counter is decremented.
3. In the static analysis, we assume that each aggregate parameter of a function is reusable. But, quite likely, there are multiple calls to the same function. Different calls have different environments, and so, for some calls, an aggregate parameter can be overwritten, while for the others, it can not be destructively written. Our strategy is: at compile time, an auxiliary parameter is generated for every aggregate parameter to indicate whether the storage for this aggregate can be reused in the current call.

4.2 Static methods

Definitions $\tilde{E}$ is related to a specific execution, and generally, it can not be calculated statically. In this section, we define the relation $\mathcal{R}$ to statically derive the possible sharing on all the aggregates and sub-aggregates in a function. We also introduce the concept of nesting degree.

1. $A \approx_{op} B$ denotes *E-Degree(op, A, B)*> 0.
2. $A \sqsubset_{op} B$ denotes that A may become a subaggregate of B through *op*; $A \sqsupset_{op} B$ denotes that B may be a subaggregate of A as a result of *op*, where A and B are input and output aggregates of *op* respectively.
   For example, for *op*: $B := A[i]$ where A is a nested aggregate, $A \sqsupset_{op} B$; for *op*: $B := C[i : A]$ where C and B are nested aggregates, $A \sqsubset_{op} B$; for *op*: $A :=$ *if cond then* $B[i]$ *else* $C[i]$ *end if* where C and B are nested aggregates, $B \sqsupset_{op} A$ and $C \sqsupset_{op} A$.
3. Let $\mathcal{A}$ be the set of all the aggregates and sub-aggregates in a function, then $\mathcal{R}$ is the smallest equivalence relation on $\mathcal{A}$ satisfying the following condition: For $A, B \in \mathcal{A}$, if there is an *op* so that $A \approx_{op} B$ or $A \sqsubset_{op} B$ or $A \sqsupset_{op} B$ holds, then $(A, B) \in \mathcal{R}$. $\mathcal{R}(A)$ includes all possible aggregates which may share with A, or whose sub-aggregates (or sub-aggregates of sub-aggregates, ...) or whose super-aggregates (or super-aggregates of super-aggregates, ...) may.
4. the nesting degree of an aggregate A:

$$\text{N-Degree(A)} = \begin{cases} 1 & \text{if } A \text{ is a flat aggregate} \\ \text{N-Degree(sub(A))} + 1 & \text{otherwise} \end{cases}$$

   where $sub(A)$ is any sub-aggregate of A.

[2] This is true if the aggregate is not a parameter of the current function. The more exact condition is reflected on the reusability tag. See §4.2

Priority rules The strategies of static ordering are embodied in the following priority rules. Suppose that op_1 and op_2 are two mutually independent operations and $(A, B) \in \mathcal{R}$.

1. **read-first**: If op_1 is an *R-operation* on A and op_2 is a *W-operation* on B, op_1 is executed before op_2.
2. **super-first**: If op_1 is a *W-operation* on A, op_2 is a *W-operation* on B, and *N-Degree(A)* $>$ *N-Degree(B)*, op_1 is executed before op_2.
3. **highest-degree-last**: If op_1 is a *W-operation* on A, op_2 is a *W-operation* on B, and *W-Degree*(op_1, A) $<$ *W-Degree*(op_2, B), op_1 is executed before op_2.

The *read-first* and *highest-degree-last* rules are motivated for the most likely reuse candidate to be executed last on each aggregate cluster. The *super-first* rule is motivated for super-aggregates to be able to dereference their sub-aggregates as early as possible in order to increase the opportunity for operations on sub-aggregates to be done in place.

But the above rules can not always be completely implemented since there can be conflicting requirements for the storage reuse of different aggregates clusters. For example, a conflict occurs in enforcing the *highest-degree-last* rule if there are operations op_1 and op_2 on both A and B with *W-Degree*(op_1, A)$>$ *W-Degree*(op_2, A) and *W-Degree*(op_1, B)$<$ *W-Degree*(op_2, B). To get around this difficulty, we replace this rule by the following simpler one:

> $3'$. *For two independent operations op_1 and op_2, op_2 is executed first if W-Degree(op_1) $>$ W-Degree(op_2), where W-Degree(op_i) $= \sum_A$ W-Degree(op_i, A)* $(i = 1, 2)$.

Similarly, there also can be conflicts in implementing the *read-first* and *super-first* rules. Figure 2(b) illustrates a conflict in implementing the read-first rule. Generally, these two rules can be completely implemented for a program only if the extra dependency arcs introduced to enforce them do not lead to dependency cycles. In these cases, we call this program *read-first-consistent* and *super-first-consistent* respectively.

Static steps

Phase 1: For each function,

1. Calculate the degrees of each aggregate operation;
2. Calculate the relation $\mathcal{R}$;

The process may be recursive due to recursive function calls, and the following analysis takes account of this. Suppose that n, m, k are the number of functions, the largest size of function (i.e. the maximal number of nodes and arcs within a function graph), and the maximal number of aggregate parameters to a function respectively. Because all the degrees of each aggregate are monotonically increasing in the calculation, by fixing the scale of degrees to a small constant s, the total time complexity of step 1 in the worst case is $O(s \cdot k \cdot n \cdot mn) = O(kmn^2)$, where skn is the maximal number of loops to calculate the fixed points, and mn represents the program size [5]. Similarly, the total time complexity of step

2 in the worst case is also $O(kmn^2)$. When k and n are quite small, they are basically linear to the program size. If there is no recursive function call, the time complexity of this phase is definitely linear.
Phase 2: For each function in the program, all nodes are ordered according to their dependencies and the above priority rules.

Input: a dataflow graph G;
Output: an ordered list of nodes L;
Algorithm **Order:**

1. Introduce a temporal dependency arc from op_1 to op_2 for each *R-operation* op_1 on A and each *W-operation* op_2 on B where $(A, B) \in \mathcal{R}$ as long as no dependency cycles are formed;
2. Introduce a temporal dependency arc from op_1 to op_2 for each *W-operation* op_1 on A and each *W-operation* op_2 on B where $(A, B) \in \mathcal{R}$ and *N-Degree(A)* >*N-Degree(B)* as long as no dependency cycles are formed;
3. Let T be an intermediate list. Initialize T to include the nodes which have no immediate predecessors and order them according to their *W-Degrees* so that node n_1 precedes node n_2 if *W-Degree*(n_1)< *W-Degree*(n_2);
4. If T is empty, output L and terminate; otherwise, $n = dequeue(T)$. If n is the last processed reuse candidate on A, then
   - if A is generated in the current graph, tag the corresponding input port of n "reusable"
   - otherwise (i.e. A is a parameter of the current function, or an input into the current compound node or subgraph node), copy the reusability tag of A into the corresponding input port of n (if any).

   If n is an atomic node, $L = L + n$; otherwise $L = L + Order_c(n)$. Delete n and all the arcs adjacent to it from G and insert the nodes whose sets of predecessors become empty into T according to their *W-Degrees*.
5. goto 4 ;

Here, $dequeue(T)$ deletes and outputs the first element of T; $L + m$ concatenates L and m where m is an element or a list. Step 1 and step 2 are to enforce the *read-first* and *super-first* priority rules respectively. Step 4 orders all nodes and identifies reusable input aggregates into each node. $Order_c$ orders all subgraphs in a compound node, identifies reusable input aggregates into a subgraph and recursively applies the algorithm $Order$ to each subgraph. $Order_c$ is similar to $Order$ and the only difference is that the former also uses the control dependencies encapsulated in the compound node. The worst case time complexity of this phase is polynomial since both step 1 and step 2 can be executed in $O(n^2 + e)$, and the worst-case complexity of steps 4 to 5 is also $O(n^2 + e)$ where n and e are the number of nodes and the number of arcs in the graph respectively. In common cases when the number of extra temporal dependency arcs is less than the size of the program, it is linear to the size of the program.

In sum, the static worst-case time complexity is polynomial to the size of the program. In common cases, it is linear.

4.3 Performance

Theorem 1. *The above method can achieve the maximal storage reuse under the following conditions for a* well-optimized *program* [3]*:*

1. *The program is read-first-consistent and super-first-consistent, and*
2. *For each operation op on an aggregate A, one of the following holds:*
   - *R-Degree(op, A)>0, W-Degree(op, A)=0, E-Degree(op, A)=0*
   - *W-Degree(op, A)+E-Degree(op, A)=1*

Proof: Due to the space limit, we only give an outline of the proof here. We consider a reusable aggregate cluster AC which includes nested aggregates (otherwise, the proof is simple). There must be reuse candidates on AC. Suppose that *wop* on $A \in AC$ is the latest executed one of them. Since the program is *read-first-consistent,* all the *R-operations* on AC must have been executed before *wop* by the *read-first* rule. Let us assume that *wop* fails in reusing the storage of AC. Then one of the following two cases must be true. **case 1:** *wop* does not *act as a physical W-operation* on A. By *condition 2*, it must *act as a physical E-operation* on A and output an aggregate $B \in AC$. But there is no operation on B since *wop* is the latest one on AC. So, B must be used as an output of the current function call (since the program is well-optimized). This contradicts the reusability of AC. **case 2:** The *sharing count* of AC is greater than 1. Since *wop* is the last operation on AC, there must be at least one super-aggregate which is still referencing an aggregate of AC. But this contradicts the premise that the program is *super-first-consistent,* by which the *super-first* rule can be completely implemented, and all the super-aggregates must have dereferenced aggregates of AC since no operation is *index-dependent* on *wop*. Thus, we conclude that the storage of any reusable aggregate cluster can be reused. □

We find that the program class described by Theorem 1 includes all the common benchmark programs: gaussian elimination, matrix transpose, matrix multiplication, LU-decomposition, quick-sort and insertion sort, where multiple-dimensional arrays are represented as nested one-dimensional arrays. The maximal storage reuse can be achieved for these programs. In fact, we can even loosen the read-first-consistency requirement to allow the existence of breakable cycles as in §3.3. In this case, we can modify the static ordering algorithm so that the nodes are ordered according to the reverse order of their dependencies to break all the cycles more easily. In other cases, static ordering is not sufficient for the maximal storage reuse and dynamically ordering operations can increase the opportunity of storage reuse. For the example illustrated in figure 3, $A_i \in \mathcal{R}(A)$ $(i = 1, 2)$ and $B_i \in \mathcal{R}(B)$ $(i = 1, 2)$, and according to the *read-first* rule, op_1 and op_3 must be executed next after *iop*. The other operations can be executed in the order $op_2 w_1 r_2 op_4 w_2 r_1$ for the maximal storage reuse if op_1 *acts as a physical W-operation* on A, since it has become clear that r_1 and op_2 do not operate on the same aggregate and r_1 need not precede op_2; otherwise,

[3] We call a program well-optimized if any value produced in a function call is always referenced later or used as an output of the function call.

in the order $op_4 w_2 r_1 op_2 w_1 r_2$ if op_3 *acts as a W-operation* on B. In the other cases when all the operations are in an interference cycle, no execution ordering, whether static or dynamic, can guarantee the maximal storage reuse. Whether the maximal storage reuse can be achieved can not be predicted, and on the basis of *degrees*, our method of static ordering tries to choose the most likely reuse candidate to operate last on each aggregate cluster to reuse its storage. But theoretical analysis of actual effects remains to be done.

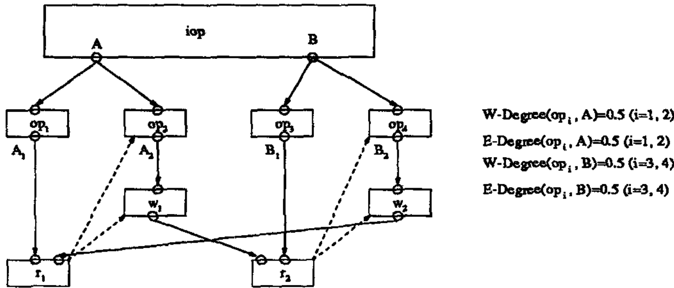

Fig. 3. A limiting case

5 Related Work

Storage optimization for applicative languages has been investigated by other researchers. But, almost all the results have been restricted to first-order functional languages without nested aggregates. Also, almost all the existing algorithms are potentially exponential except that in [7]. At Colorado, the SISAL group has developed a compiler for SISAL [2, 3]. They considered general iteration and function call boundaries with the assumption that there is no recursive function call. Although touching on nested aggregates, they did not propose a general solution and only attacked some special cases. Bloss considered update analysis to first-order lazy functional languages[1]. She defined path semantics to check whether an update can be performed destructively. But computing the path semantics of a program is at least exponentially complicated. So we do not consider it a practical method. In addition, in her work, the order of evaluation of arguments of primitive operations is statically fixed. This may decrease the opportunity of updates being done destructively. Gopinath proposed an approach to eliminating copies in divide and conquer problems through computing the target of an expression[4]. It also assumed a fixed pre-ordering of primitive operators. Also, his algorithm has very high time complexity, which makes it impractical. More recently, Sastry, Clinger and Ariola at University of Oregon claimed that their algorithm for strict functional languages with flat aggregates is the first practical one with a polynomial time complexity to solve the problem

[7]. But their algorithm is still quite conservative. For example, a function can be called from different sites with different environments. Parameters may be updated destructively under one environment but not under another one. However, according to their algorithm, if a parameter of the function can not be updated destructively in one site, then it is not allowed to be updated destructively in any environment. Obviously, this is over restrictive. None of this previous work has touched on the inherent limits of storage reuse and what the maximal storage reuse is.

6 Conclusions

The storage reuse of aggregates is essential for efficiently implementing single assignment languages. We have analyzed the inherent limits of storage reuse and defined what the maximal storage reuse is. We have also given an efficient storage reuse method. It is of polynomial complexity and linear in common cases and can achieve the maximal storage reuse for an extensive program class into which all common benchmark programs fall. We also show that no general method can guarantee the maximal reuse for programs outside the class, and in this case, our method can choose the most likely operation to reuse the storage of an aggregate or a set of shared aggregates.

Storage reuse and parallelization have conflicting requirements, e.g. although more pre-ordering can bring about more opportunities for reusing storage, it imposes more restrictions on exploitation of parallelism. Theoretical and experimental work has to be done to find their relationship.

References

1. A. Bloss. Update Analysis and Efficient Implementation of Function Aggregates. In *The 4th Int. Conf. on Functional Programming Language. and Computer Architecture*, pages 26–38, 1989.
2. D. C. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Computer Science Department, Colorado State University, 1989.
3. D.C. Cann. Retire Fortran? A Debate Rekindled. *CACM*, 35(8):81–89, Aug 1992.
4. K. Gopinath. *Copy elimination in single assignment languages.* PhD thesis, Computer System Laboratory, Stanford University, 1989.
5. Z. Li and C. Kirkham. Efficient Implementation of Aggregates in United Functions and Objects. In *33rd ACM Southeast Conference*, pages 73–82, Mar. 95.
6. J. R. McGraw, S .K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. C. Kirkham, W. Noyce, and R. Thomas. *SISAL: Streams and Iteration in a Single Assignment Language.* Lawrence Livermore National Laboratory, reference manual 1.2, manual m-146, rev. first edition, 1985.
7. A. V. Sastry, W. Clinger, and Z. Ariola. Order-of-evaluation Analysis for Destructive Updates in Strict Functional Languages with Flat Aggregates. In *the 6th Int. Conf. on Functional Programming Languages and Computer Architectures*, 1993.
8. S. K. Skedzielewski and M. L. Welcome. Dataflow Graph Optimization in IF1. In J-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 17–34. Springer-Verlag, NY, November 1985.