On Copy Avoidance in Single Assignment Languages

Saumya K. Debray

Department of Computer Science
University of Arizona
Tucson, AZ 85721, U.S.A.
debray@cs.arizona.edu

Abstract: Copy avoidance refers to the safe replacement, at compile time, of copying operations by destructive updates in single-assignment languages. Conceptually, the problem can be divided into two components: identifying memory cells that can safely be reused at a program point via destructive updating; and deciding how to actually reuse such cells. Most of the work on this problem, to date, has focused on the first component, typically via dataflow analyses to detect when memory cells become dead and may be safely reused. In this paper, we examine the second component of the problem. We give an abstract formulation of the memory reuse problem, show that optimal reuse is NP-complete in general, and give an efficient polynomial-time approximation algorithm based on graph-matching techniques that produces optimal solutions for most commonly encountered cases of memory reuse.

1 Introduction

Single assignment languages, such as pure functional and logic programming languages, do not have any notion of updatable variables: the value of a variable or structure, once defined, does not change during execution. Updates to the value of a variable or structure have to be effected, instead, by creating new copies. This can lead to an undesirable degradation of performance.

The problem can be overcome to some extent by using sophisticated compilers that analyze the program to determine when a structure can be safely updated in place, and avoid creating copies in such cases. There has been a considerable amount of work in this context, e.g., see [1, 8, 9, 10] in the context of functional programming languages, and [2, 3, 4, 11, 12, 15] in the context of logic programming languages. Most of this work is aimed at determining *when* an update operation can be safely performed destructively. For example, the work of Bruynooghe [2, 3], Foster and Winsborough [4], Hudak and Bloss [1, 8, 9], Mulkers *et al.* [12], and Sastry *et al.* [15] focus on compile-time reference counting schemes to determine when a data structure being updated has at most one reference to it, and can therefore be safely updated in place.

Conceptually, there are two components to compile-time memory reuse. First, it is necessary to determine when a particular structure can be safely reused—or, equivalently, which structures can be safely reused at any given program point. Then, given a structure that can be safely reused, it is necessary to determine how best to reuse the memory it occupies. Most of the work to date on compile-time memory reuse has focused on the first component: the underlying assumption seems to be that once the data structures that are updatable at a given program point have been identified, deciding how to reuse them in a "good" way is straightforward. This paper examines the second component of the memory reuse problem, assuming that the first component has been addressed via existing compile-time analysis techniques, so that we know which data structures are available for reuse at any given program point. Any such structure can, in general, be reused in different ways, with different runtime costs (either in space or time), and the compiler should try to make its reuse decisions in a manner that reduces this cost as far as possible. We give an abstract formulation of this problem, show that optimal memory reuse is NP-complete in general, and give a polynomial-time approximation algorithm based on graph matching techniques that produces optimal solutions for most commonly encountered cases of memory reuse.

We make few assumptions about the language under consideration, except that it is a single assignment language, i.e., the value of a variable cannot be modified once it has been defined. We assume that the language supports some reasonable set of base types, as well as *vectors*, i.e., objects occupying a sequence of contiguous memory cells. Thus, an n-element array can be thought of as a vector of n cells, while a cons cell can be considered to be a vector of 2 cells. A term $f(t_1, \ldots, t_n)$ with n arguments $t_1, \ldots, t_n$ in logic programming languages such as Prolog is typically implemented as a vector of $n+1$ cells: one cell for the constructor f, and n cells for (pointers to) the arguments $t_1, \ldots, t_n$. The notion of a "memory cell" in this context, may—but need not—coincide with the notion of a word of memory in the underlying implementation.

2 The Memory Reuse Problem

In this section, we give a general formulation of the memory reuse problem. To motivate this formulation, we consider some simple examples.

Example 2.1 Consider the following clause, which is taken from the assembler for SB-Prolog: it takes a WAM instruction in symbolic form and consults the symbol table to determine the "external form" for that instruction, which refers to a displacement in the (runtime) symbol table:

```
asm_p2(putstr(Functor,Arity,Reg), Instr, SymTab) :-
     Instr = put_structure(Disp, Reg),
     symtab_lookup( (Functor,Arity), SymTab, Disp).
```

Suppose we know that the first argument of `asm_p2` is dead after a call to that predicate, so that the memory that it occupies—in this case, if we assume the "usual" WAM representation of terms, four cells—may be reused. There are two terms constructed in the body: the output instruction `put_structure(Disp, Reg)`, which occupies 3 memory cells, and the structure `(Functor,Arity)`, which also occupies 3 cells.[1] Suppose an assignment operation has cost 1. Then, if the first argument is reused to construct the term `put_structure(Disp, Reg)`, we need three assignments, one each for the principal functor and the two arguments, and therefore incur a cost of 3. On the other hand, if it is used to construct the term `(Functor,Arity)`, we require only one assignment, for the principal functor, since

[1] Most Prolog implementations represent a term (t_1, t_2) as ','(t_1, t_2), i.e., using a binary function symbol ','.

`Functor` and `Arity` are already present in adjacent memory cells, and therefore incur a cost of 1. In this case, therefore, it is cheaper to reuse the memory from the first argument to construct the term `(Functor,Arity)`, by simply destructively updating the principal functor of this term from `putstr`/3 to '`,`'/2. □

This example illustrates two points: first, there may be a number of different choices regarding how the memory for a particular vector should be reused; and second, each such choice for a memory reuse decision may have a different cost. A good compiler should, therefore, take such costs into account, and select the cheapest alternative in its memory reuse decisions whenever possible.

The next example illustrates the situation where a structure that is available for compile-time reuse is large enough to be used to construct more than one new data structure.

Example 2.2 Consider the following clause for a quicksort program in Janus [13][2], where `|E|` denotes the size of the array `E` and `++` denotes array concatenation:

```
qs(E, K, ^B) :- 0 < K, K < |E| ->
     qs(E[0..K], ^C), qs(E[(K+1)..(|E|-1)], ^D), B = C++D.
```

Here, the input array `E` is split into two parts: the first, `E[0..K]`, containing the first `K`+1 elements, and the second, `E[(K+1)..(|E|-1)]`, containing the remainder of the array. If we know that the input array `E` is dead at this point , the memory occupied by `E` can be reused to construct both of the two smaller arrays. □

In general, then, a compile-time memory reuse problem at any given program point is characterized by the following:

1. There is a set of *producers* that produce memory that can be reused.
2. There is a set of *consumers* that consume memory. These are structures that have to be constructed in memory for subsequent use.
3. A producer may be used to satisfy the demands of one or more consumers. In this case, we can distinguish between two situations:
   (a) *OR-allocation*, where a producer can satisfy the memory requirements of at most one of a set of choices, as in Exampl 2.1; and
   (b) *AND-allocation*, where a producer can satisfy the requirements of all of a set of choices, as in Example 2.2.
4. There may be a cost associated with choosing a particular producer to satisfy the requirements of a particular consumer.

This suggests the following definition of a memory reuse problem:

Definition 2.1 A memory reuse problem is a 5-tuple $\langle P, C, choices, cost, newcost\rangle$, where:

[2] The Janus syntax given here is a "flattened" version of that used in [13].

- P is a set of *producers*;
- C is a set of *consumers*;
- $choices : P \longrightarrow \wp(\wp(C))$ is a function that maps each producer to a set of sets of consumers. Intuitively, given a producer $u \in P$, the memory produced by u can be used for any one of the elements of $choices(u)$; each element of $choices(u)$ represents a set of consumers that may collectively be allocated using u. This function must satisfy the following: $\emptyset \in choices(u)$ for every $u \in P$, i.e., it is an option to not reuse a producer.
- $cost : P \times C \longrightarrow \mathcal{N} \cup \{\perp\}$ is a function such that for any producer u and consumer v, the value of $cost(u, v)$ is the cost of constructing v from u.[3] This function must satisfy the following: for any $u \in P$ and $v \in C$, $cost(u, v) \neq \perp$ if and only if v occurs in some element of $choices(u)$. The intuition is that $cost(u, v) = \perp$ if and only if v cannot be constructed from u.
- $newcost : C \longrightarrow \mathcal{N}$ is a function that gives, for each consumer c, the cost of constructing c from new memory.[4]

■

A solution to a memory reuse problem is simply a specification of which producers are to be used to construct which consumers. More formally:

Definition 2.2 Given a memory reuse problem $\langle P, C, choices, cost, newcost \rangle$, a function $alloc : P \longrightarrow \wp(C)$ is a *solution* to the problem if and only if the following hold:

1. $alloc(u) \in choices(u)$ for every $u \in P$, i.e., allocation decisions must be legal; and
2. for every $u, v \in P$, if $u \neq v$ then $alloc(u) \cap alloc(v) = \emptyset$, i.e., at most one producer can be used to construct any one consumer.

■

A point to note in this definition is that a solution need not reuse every producer: if $alloc(u) = \emptyset$ for some producer u, then u is not being reused in that solution.

[3] W. Winsborough has pointed out to us that in general, the cost of constructing a consumer v from a producer u depends also on which portion of u is used to construct v, and that this may be an issue if a producer is to be AND-allocated to multiple consumers [18]. This can be handled by generalizing $cost$ to be a 4-ary relation such that a tuple $\langle u, v, i, c \rangle \in cost$ if and only if the cost of constructing the consumer v at position i of the producer u is c. It has been conjectured that given a set of consumers to be AND-allocated from a producer, the problem of mapping the consumers to positions in the producer in order to minimize the cost of the AND-allocation is NP-complete [18].

[4] An alternative would be to consider "new memory" as a distinguished element of the set of producers. The problem with this is that since any subset of the set of consumers can be constructed from new memory, all these possibilities have to be accounted for in $choices$, resulting in characterizations of memory reuse problems that are exponentially larger than one intuitively expects them to be.

Indeed, if there are more producers than consumers, then it will not be possible to reuse every producer.

The cost of a solution is the total the cost of reusing memory for those consumers that are constructed by reusing memory from producers, and of using new memory for those consumers that do not reuse memory from producers:

Definition 2.3 Given a memory reuse problem $\langle P, C, \mathit{choices}, \mathit{cost}, \mathit{newcost}\rangle$, the *cost* of a solution *alloc* to this problem is given by the following: let $\mathit{Reused} \subseteq C$, given by

$$\mathit{Reused} = \{v \mid \exists u \in P : v \in \mathit{alloc}(u)\}$$

denote the set of consumers constructed by reusing producers. Then, the cost of the solution *alloc* is given by the following, with $\sum \emptyset = 0$:

$$\begin{aligned} \mathit{cost}(\mathit{alloc}) = {} & \textstyle\sum\{\mathit{cost}(u, v) \mid u \in P \wedge v \in \mathit{Reused} \cap \mathit{alloc}(u)\} + \\ & \textstyle\sum\{\mathit{newcost}(v) \mid v \in C \setminus \mathit{Reused}\}. \end{aligned}$$

A solution for a memory reuse problem is said to be *optimal* if its cost is no greater than that of any other solution to that problem. ∎

In the remainder of the paper, we will assume, additionally, that the cost of constructing a structure from new memory is at least as much as the cost of constructing that structure by reusing a producer. In other words, given a memory reuse problem $\langle P, C, \mathit{choices}, \mathit{cost}, \mathit{newcost}\rangle$, for every $u \in P$ and $v \in C$, if $\mathit{cost}(u, v) \neq \bot$ then $\mathit{cost}(u, v) \leq \mathit{newcost}(v)$. This assumption seems reasonable, since the cost of constructing a structure v from new memory requires initializing the fields of v, together with some additional costs such as checking for availability of memory and the actual allocation of new memory, which would not be incurred if v were to reuse a producer.[5] It does not lose generality, since if there is any $u \in P$ and $v \in C$ such that $\mathit{cost}(u, v) \neq \bot$ and $\mathit{cost}(u, v) > \mathit{newcost}(v)$, then reusing u to construct v can never produce an optimal solution: we can always do better by constructing v from new memory. We can therefore delete v from each element of $\mathit{choices}(u)$ without affecting any optimal solution.

3 Space- and Time-Optimal Solutions

There are (at least) two reasonable approaches to measuring the cost of a solution: one can consider either (*i*) the time cost, such as might be measured by the number of instructions necessary to construct the consumers; or (*ii*) the space cost, such as might be measured by the amount of new memory needed by the consumers. The two are not equivalent: a solution to a memory reuse problem that is optimal with respect to the time cost may not be optimal with respect to the space cost, and vice versa. This is illustrated by the following example:

[5] However, it is conceivable that the cost of value trailing necessary for destructive updates in a logic programming language might outweigh the cost of allocating and initializing new storage [18].

Example 3.1 Consider the memory reuse problem $\langle P, C, \mathit{choices}, \mathit{cost}, \mathit{newcost} \rangle$, where $P = \{\mathrm{P1}, \mathrm{P2}\}$; $C = \{\mathrm{C1}, \mathrm{C2}, \mathrm{C3}\}$; and the functions *choices*, *cost*, and *newcost* are defined as follows:

- $\mathit{choices} = \{\mathrm{P1} \mapsto \{\emptyset, \{\mathrm{C1}\}, \{\mathrm{C2}\}\}; \mathrm{P2} \mapsto \{\emptyset, \{\mathrm{C2}\}, \{\mathrm{C3}\}\}\}$;
- $\mathit{cost} = \{(\mathrm{P1}, \mathrm{C1}) \mapsto 5; (\mathrm{P1}, \mathrm{C2}) \mapsto 2; (\mathrm{P2}, \mathrm{C2}) \mapsto 6, (\mathrm{P2}, \mathrm{C3}) \mapsto 1\}$;
- $\mathit{newcost} = \{\mathrm{C1} \mapsto 9; \mathrm{C2} \mapsto 8; \mathrm{C3} \mapsto 7\}$.

Suppose the functions *cost* and *newcost* specify the time cost of memory reuse, while the space cost of constructing a consumer x from new memory is given by $\mathit{newcost}(x) - 1$. Then, the optimal solution if we want to minimize time costs is $\{\mathrm{P1} \mapsto \{\mathrm{C2}\}, \mathrm{P2} \mapsto \{\mathrm{C3}\}\}$, with time cost $\mathit{cost}(\mathrm{P1}, \mathrm{C2}) + \mathit{cost}(\mathrm{P2}, \mathrm{C3}) + \mathit{newcost}(\mathrm{C1}) = 12$. However, the space cost of this solution is $\mathit{newcost}(\mathrm{C1}) - 1 = 8$. If we want to minimize space costs instead, the optimal solution is given by $\{\mathrm{P1} \mapsto \{\mathrm{C1}\}, \mathrm{P2} \mapsto \{\mathrm{C2}\}\}$. The space cost for this solution is $\mathit{newcost}(\mathrm{C3}) - 1 = 6$, but its time cost is $\mathit{cost}(\mathrm{P1}, \mathrm{C1}) + \mathit{cost}(\mathrm{P2}, \mathrm{C2}) + \mathit{newcost}(\mathrm{C3}) = 18$. $\square$

This example shows that a space-optimal solution to a memory reuse problem need not be a time-optimal one, and vice versa. However, when compiling a program we know whether time costs are to be given greater importance than space costs or vice versa, and can accordingly define the function *cost* in memory reuse problems encountered. Definition 2.3 therefore captures both approaches to measuring the cost of a solution.

4 Algorithms for Memory Reuse Problems

We first consider arbitrary memory reuse problems. The following result suggests that the existence of efficient algorithms for computing optimal solutions to such problems is unlikely:

Theorem 4.1 *The determination of an optimal solution to an arbitrary memory reuse problem is NP-complete. It remains NP-complete even if each producer has at most one nonempty alternative for OR-allocation, and all memory reuses have the same cost.*

Proof (sketch) Optimal memory reuse can be formulated as a decision problem as follows: "Given a memory reuse problem M and an integer $K \geq 0$, is there a solution to M with cost no greater than K?" The proof is by reduction from the One-in-Three Satisfiability problem where no clause contains a negated literal (see problem LO4 in [5]), which is known to be NP-complete [5, 16]. Given an instance I of the One-in-Three Satisfiability problem involving a set U of variables and a collection S of clauses, each containing 3 literals of which none are negated, the idea is to construct a memory reuse problem $M(I) \equiv \langle U, S, \mathit{choices}, \mathit{cost}, \mathit{newcost} \rangle$, where the functions *choices*, *cost*, and *newcost* are defined as follows:

- for each $u \in U$, let $S(u) = \{s \in S \mid u \in s\}$, then, $\mathit{choices}(u) = \{\emptyset, S(u)\}$.
- for any $u \in U$ and $v \in S$, $\mathit{cost}(u, v) = 1$ if u occurs in v, and is $\perp$ otherwise.

– $newcost(x) = |S| + 1$ for every $x \in S$.

It is not difficult to see that $M(I)$ can be constructed in time polynomial in the size of I. It can be shown that I has a solution τ if and only if the memory reuse problem $M(I)$ has a solution *alloc* defined as :

$$alloc(u) = \textbf{if } \tau(u) = true \textbf{ then } S(u) \textbf{ else } \emptyset.$$

with $cost(alloc) = |S|$. □

We therefore seek approximation algorithms for memory reuse problems that are efficient and achieve good solutions for common cases. Now in practice, it is often the case that each dead data structure that is reused is used to create just a single new structure. Memory reuse problems that satisfy this condition are said to be "simple":

Definition 4.1 A memory reuse problem $\langle P, C, choices, cost, newcost\rangle$ is *simple* if for every $u \in P$, $\forall x \in choices(u) : |x| \leq 1$. ■

It turns out that optimal solutions for simple memory reuse problems are efficiently computable:

Theorem 4.2 *Let $M \equiv \langle P, C, choices, cost, newcost\rangle$ be a simple memory reuse problem, then an optimal solution to M can be computed in time $O(mn \log n)$, where $n = |P| + |C|$ and $m = |\{(u, v) \mid cost(u, v) \neq \perp\}|$.*

Proof We transform the memory reuse problem M into a maximum-weight matching problem for a weighted bipartite graph $G = (V, E)$ defined as follows:

– $V = P \cup C$;

– there is an edge (u, v) in E if and only if $cost(u, v) \neq \perp$; and

– the weight of an edge (u, v) is $newcost(v) - cost(u, v)$.

Intuitively, the weight of an edge represents the savings realized by constructing v from u rather than from new memory. From the assumption that $newcost(v) \geq cost(u, v)$ for all u and v such that $cost(u, v) \neq \perp$, it follows that every edge in G has nonnegative weight.

A *matching* on G is a set of edges $E' \subseteq E$ such that no two edges in E' have a common vertex. A matching E' on G, corresponds to a solution *alloc* to the original memory reuse problem M: for any $u \in P$,

$$alloc(u) = \textbf{if } (u, v) \in E' \textbf{ then } \{v\} \textbf{ else } \emptyset.$$

By definition, $(u, v) \in E' \subseteq E$ implies that $cost(u, v) \neq \perp$, which can happen if and only if v occurs in some element of $choices(u)$. Since M is a simple memory allocation problem, it follows that every nonempty element of $choices(u)$ is a singleton, which means $\{v\} \in choices(u)$, and therefore $alloc(u) \in choices(u)$. Further, since no two edges in E' share a common endpoint, E' does not contain two edges (u, v) and (u', v) with $u \neq u'$. It is easy to show that this implies that for all $u, u' \in P$, $u \neq u'$ implies that $alloc(u) \cap alloc(u') = \emptyset$. This verifies that $alloc$ is a solution to M.

It is straightforward to show that such a solution is optimal if and only if the corresponding matching has maximum weight. It is known that the maximum-weight matching problem for a bipartite graph with n vertices and m edges can be solved, using network flow techniques, in time $O(mn \log_{2+m/n} n)$ [17].[6] The theorem follows. $\Box$

For simple memory reuse problems, therefore, we can determine optimal solutions efficiently using graph matching techniques. We next show how approximate solutions to arbitrary memory reuse problems can be obtained using similar techniques. The idea is straightforward: what makes the determination of an optimal solution to an arbitrary memory reuse problem I algorithmically difficult is the presence of AND-allocation constraints, so we simplify I by "lumping together" all AND-allocation constraints that are "related", i.e, that share a consumer. From this we can obtain a weighted bipartite graph for which a maximum-weight matching can be computed in polynomial time. This matching can then be translated into a solution to the original memory reuse problem I. It is possible, however, that simplifying I to lump together various AND-constraints can force us to omit certain allocation possibilities that are available in I. As a result, the solution computed may not reuse a producer even though it is possible and profitable to do so. To catch such possibilities, we "delete" from I the producers reused by the solution we have computed and the consumers that reused them. This yields another (smaller) memory reuse problem, and we can repeat the above procedure on the residual problem to obtain a solution that can be used to augment the earlier solution and compute a second residual problem, and so on. This procedure is repeated until no further memory reuse is possible. Details are given in Figure 1.

Example 4.1 Consider the memory reuse problem of Figure 2, where the producers are A and B and the consumers are a, b, c, d, e, f, g, h. A can be reused to construct either $\{a, b, c\}$ with a savings of 1 each, for a total savings of 3; or $\{c, d, e\}$ with savings of 1, 2, and 1 respectively, for a total savings of 4. B can be reused to construct either $\{e, f\}$, with savings of 3 and 2 respectively, for a total savings of 5; or $\{a, e, g\}$ with savings of 1, 3, and 2 respectively, for a total savings of 6; or $\{h\}$ with savings 1. The problem is illustrated pictorially in Figure $2(a)$: to reduce clutter, we have grouped various sets of AND-allocations together, with reuse decisions indicated by one thick line for each group of AND-allocations labelled by the total savings for that group.

This problem is first simplified by merging sets of consumers that are not mutually disjoint. The graph of the resulting problem is shown in Figure $2(b)$. The graph is simplified further by removing multiple edges between nodes to retain only the heaviest edge between any pair of nodes. This yields the bipartite graph of Figure $2(c)$. A maximum weight matching to this graph consists of the edge with weight 6: this translates to a (partial) solution to the original problem that reuses

[6] Note that labelling the edges of G with costs rather than savings, in order to compute a least-cost solution, yields a generalization of the *Minimum Maximal Matching* problem, which is NP-complete even for bipartite graphs [5].

B to construct $\{a, e, g\}$. The reused producers and consumers are then deleted from the original problem to obtain the residual reuse problem of Figure 2(d). Notice that because B has been deleted, there are no producers for f and h. The whole procedure is now repeated on this smaller problem: its solution, which is to reuse A to construct $\{c, d\}$ with total savings 3, is used to augment the partial solution computed earlier. The residual problem after this contains no producers and so yields an empty matching at the next iteration, whereupon the algorithm terminates.

The resulting solution reuses A to construct $\{c, d\}$ and B to construct $\{a, e, g\}$, with a total savings of 9. The remaining consumers, i.e., b, f and h, are constructed from new memory. □

Given an arbitrary memory reuse problem $M \equiv \langle P, C, \mathit{choices}, \mathit{cost}, \mathit{newcost}\rangle$, let $|P| + |C| = n$, $|\{(u, v) \mid \mathit{cost}(u, v) \neq \bot\}| = m$, and $|\cup_{u \in P} \mathit{choices}(u)| = k$. The worst-case complexity of this algorithm can be computed as follows:

1. Consider the cost of a single call to the function *simplify* with input M. To construct the graph G, we have to construct $\widehat{C}$, which can be done as follows: first, construct a graph H whose vertices are the elements of C, and where there is an edge from a vertex v_1 to a vertex v_2 if and only if there is some $u \in P$ such that v_1 and v_2 appear in some element of $\mathit{choices}(u)$; then, the elements of $\widehat{C}$ correspond precisely to the connected components of H. To compute the edges of H, we consider each nonempty element s appearing in the range of *choices*; and for each such s, arbitrarily pick an element $v \in s$ and add an edge from v to each element $w \in s$ such that $v \neq w$. Now there are $O(k)$ elements in the range of *choices*, each containing at most $O(n)$ elements and therefore contributing $O(n)$ edges to H. Thus, H contains n vertices and $O(nk)$ edges, and can be constructed in time $O(nk)$. The connected components in H can then be identified in time $O(nk)$ using depth-first search. Thus, the set $\widehat{C}$ can be constructed in time $O(nk)$.

   The graph G constructed after this has $O(n)$ vertices. Identifying its edges in a naive way would incur cost $O(n^2k^2)$, since each edge can be identified in $O(k)$ comparisons, each of cost $O(n)$, and there are at most $O(nk)$ edges. However, we can do better by maintaining some additional information while manipulating the graph H. Initially, every node in H is unlabelled. For each $u \in P$ and $v \in \mathit{choices}(u)$, we compute $c_{uv} = \sum_{w \in v} \mathit{cost}(u, w)$, and pick an arbitrary element $x \in v$ and label it with the pair (u, c_{uv}). During the depth-first search to find the connected components of H, whenever we reach a node x that has a label (u, c), we add an edge in the graph G with weight c between the vertex corresponding to u and that corresponding to the connected component of x. Using this approach, the edges of G can be identified in time proportional to the number of vertices in H, i.e., in time $O(nk)$.

   Thus, the total cost of a call to *simplify* is $O(nk)$.

2. Given a weighted bipartite graph with v vertices and e edges, the cost of computing a maximum-weight matching is $O(ve \log v)$ [17]. The graph returned by *simplify* has $O(n)$ vertices and $O(k)$ edges, so the cost of computing a matching is $O(nk \log n)$. Finally, the updating of the solution using the matching so computed, and of computing the residual reuse problem, can be done in time

Input: A memory reuse problem $I = \langle P, C, choices, cost, newcost \rangle$.

Output: A solution S to I.

Method: return $S = solve(I)$;

function $solve(I)$ /* I is an arbitrary memory reuse problem */
begin
  let $I \equiv \langle P, C, choices, cost, newcost \rangle$;
  $S := \emptyset$;
  repeat
    $G := simplify(I)$;
    compute a maximum-weight matching M for G;
    if $M \neq \emptyset$ **then**
      for each edge $(u, v) \in M$ with weight n **do**
        – find a set $v' \in choices(u)$ such that $v' \subseteq v$ and $\sum_{w \in v'} cost(u, w) = n$;
        – augment the solution S with the memory reuse decision $u \mapsto v'$;
        – delete u from P;
        – delete each element of v' from C and from each element of $choices(u)$;
      od
    fi
  until $M = \emptyset$;
  return S;
end

function $simplify(I)$ /* I is an arbitrary memory reuse problem */
begin
  let $I \equiv \langle P, C, choices, cost, newcost \rangle$;
  $\widehat{C} := \cup\{choices(u) \mid u \in P\} \setminus \emptyset$;
  merge subsets of $\widehat{C}$ that are not pairwise disjoint;
  let $G = (V, E)$ be the weighted bipartite graph defined as follows:
    – $V = P \cup \widehat{C}$; and
    – for each $u \in P$ and $v \in \widehat{C}$, there is an edge $(u, v) \in E$ with cost n for every $v' \in choices(u)$ such that $v' \subseteq v$, where $n = \sum_{w \in v'} cost(u, w)$;
  for every $u \in P, v \in \widehat{C}$ **do**
    if there are multiple edges between u and v, delete all but the heaviest such edge;
  od
return G;
end

Figure 1: An Algorithm for Arbitrary Memory Reuse Problems

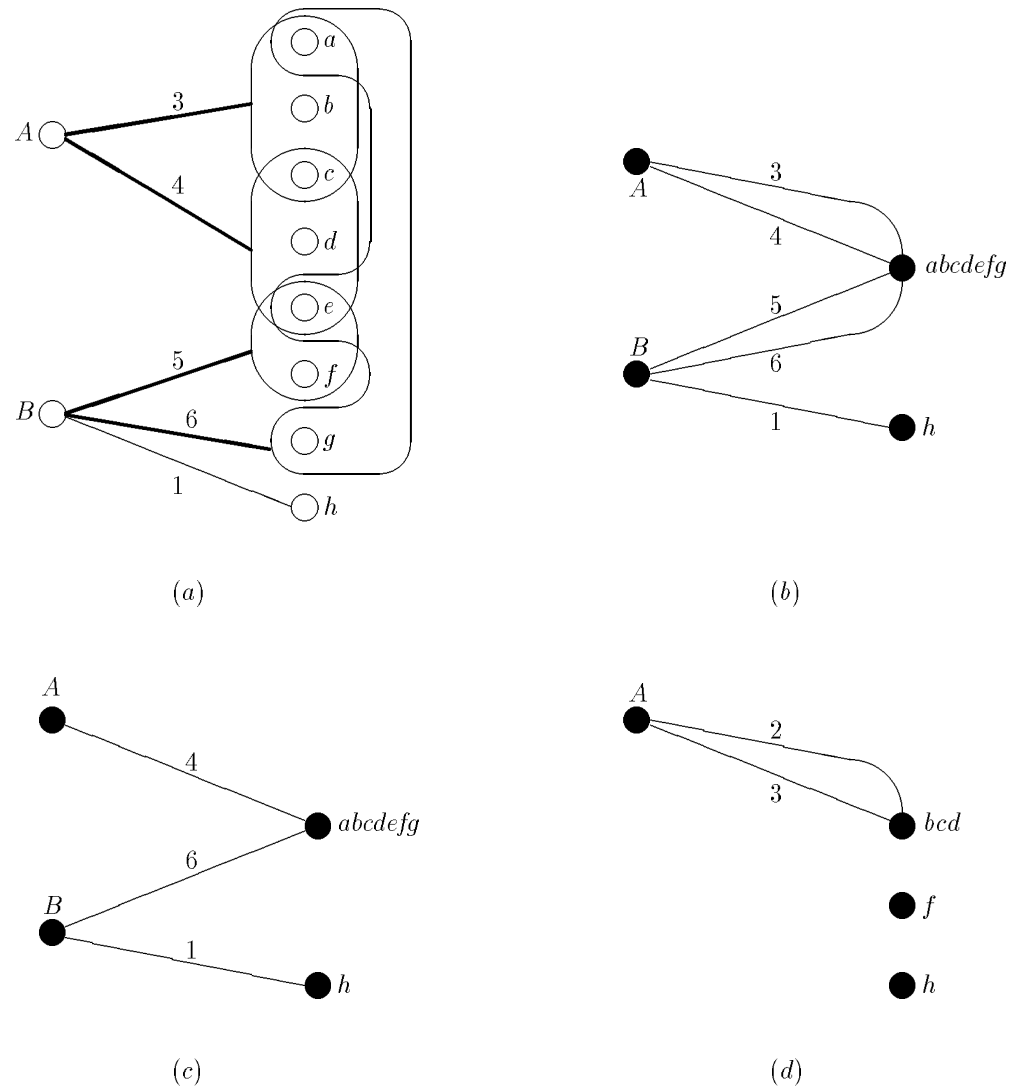

Figure 2: The Memory Reuse Problem of Example 4.1

Program	T_0 (ms)	T_1 (ms)	T_2 (ms)	δ_0 (%)	δ_1 (%)	Δ (%)
nrev	352.0	244.1	178.5	49.3	26.9	60.8
disj	437.8	413.3	399.7	8.7	3.3	55.5
qsort	616.0	510.0	470.7	23.6	7.7	37.1
prime	702.1	651.5	635.3	9.5	2.5	32.0
insert	2039.4	1954.8	1696.9	16.8	13.2	304.8
queen	4074.0	3932.5	3849.3	5.5	2.2	58.8

Key :

T_0 : execution time, no memory reuse

T_1 : execution time, "naive" memory reuse

T_2 : execution time, intelligent memory reuse

$\delta_0 = (T_0 - T_2)/T_0$: speed improvement due to intelligent memory reuse, relative to no reuse

$\delta_1 = (T_1 - T_2)/T_1$: speed improvement due to intelligent memory reuse, relative to naive reuse

$\Delta = (T_1 - T_2)/(T_0 - T_1)$: speed improvement due to intelligent reuse relative to naive reuse, compared to the speed improvement due to naive reuse compared to no reuse.

Table 1: Speed Improvements due to Intelligent Memory Reuse

$O(n + k)$. Thus, the total cost of each iteration in the body of the function *solve* takes time $O(nk + nk \log n + n + k) = O(nk \log n)$.

3. In the worst case, each iteration succeeds in matching one producer with one consumer, so the size of the problem decreases by 2 at each iteration. The total number of iterations is therefore $O(n)$, and the overall worst case cost of the algorithm is $O(n^2 k \log n)$.

Thus, given an arbitrary memory reuse problem with n producers and consumers and k choices for reuse, the algorithm described above runs in time $O(n^2 k \log n)$. Moreover, for simple memory reuse problems it runs in time $O(nk \log n) = O(nm \log n)$, and the solution computed is optimal.

5 Performance

To test the efficacy of our algorithm, we experimented with a number of small benchmarks on the `jc` system [6], running on a Sparcstation-2. The results are reported in Table 1. The benchmarks tested were the following:

`nrev` : the "naive reverse" program, run on a list of length 1000.

`disj` : a program to transform a propositional formula to its disjunctive normal form—in our experiment, the input formula contained a total of 62 connectives and variables, while the output contained 1319 variables and connectives altogether.

`qsort` : the quicksort program, run on an ordered list of 1000 integers (this causes the program to exhibit its worst-case $O(n^2)$ behavior).

`prime` : a program to compute a list of prime numbers upto 5000, using the Sieve of Eratosthenes.

`insert` : a program that creates a binary tree of 1000 integers.

`queen` : a program to determine all solutions to the 10-queens problem.

The programs `nrev`, `qsort`, `prime`, `insert`, and `queen` were obtained by a direct translation, from FGHC to Janus, of benchmarks used by Sundararajan, Sastry and Tick [15]; the `disj` benchmark was translated from a Prolog program due to J. Jaffar, discussed in [7]. These programs all gave rise to simple memory reuse problems, for which our algorithm computes optimal solutions in polynomial time.

The second, third, and fourth columns of Table 1 give the runtimes with, respectively, no memory reuse; naive memory reuse (where memory is reused but no attempt is made to avoid reinitializing cells that already have the desired value in place, as in [15]); and intelligent memory reuse using our algorithm. The fifth and sixth columns give the percentage speed improvement due to intelligent reuse, compared, respectively, to the cases with no reuse and with naive memory reuse. The numbers indicate that on typical programs, intelligent memory reuse leads to a speed improvement of roughly 2% to 13%, compared to “naive” memory reuse, and about 5% to 24% compared to no reuse at all (we do not consider `nrev` in these numbers, since it performs so little “interesting” computation that the speed gains from intelligent memory reuse are magnified unrealistically). The latter numbers are actually somewhat conservative, since they do not reflect the speed improvements due to reduced runtime garbage collection.

However, these numbers do not, of themselves, give an accurate estimate of the utility of intelligent memory reuse compared to naive reuse, since they do not distinguish between programs with many opportunities for memory reuse and programs with few such opportunities (the point is that in a program with comparatively few opportunities for local reuse, our algorithm will not produce very large speedups—but this is due to the very nature of the program, rather than a deficiency in our algorithm). It would be more reasonable, in our opinion, to compare the additional speed improvement is obtained by intelligent memory reuse (relative to naive reuse) with the speed improvement due to naive reuse (relative to no reuse). This figure is reported in the seventh column, and it shows that the additional speed improvements due to intelligent reuse, compared to the speed improvements due to naive reuse, are quite substantial. A point to note is that in all of the benchmarks tested except `insert`, the data structures being reused are cons cells, which means that each instance of intelligent reuse simply manages to avoid one assignment (by not having to update the head of the cons pair)—despite this, the incremental improvement due to intelligent reuse is quite significant, ranging from 32% to 58%. In `insert`, on the other hand, where each node of the binary tree being processed is a term with three arguments, intelligent reuse avoids two assignments, and the speed improvement due to intelligent reuse is a factor of 3 greater than the speed improvement due to naive reuse.

Finally, it should be noted that while the intelligent memory reuse scheme described here can be expected to produce performance improvements compared to naive reuse, it may or may not be faster than a system that does no memory reuse at all, depending on the details of the cost model used. The problem is similar to that of common subexpression elimination during compilation: in order to reuse the memory for a data structure, it is necessary to keep track of its whereabouts, which requires tying up a register or incurring some memory reads and writes. If the cost model does not take such low level costs into account, the expense of keeping track of a dead data structure until it can be reused may outweigh the savings incurred from its reuse, and produce code that runs slower (this is a phenomenon that we observed while experimenting with the `disj` benchmark mentioned above—disregarding such low level costs led, on some inputs, to a slowdown of about 20% compared to code with no memory reuse). It should be noted, however, that this problem is inherent in compile-time memory reuse—that is, a memory reuse scheme that is not careful about low level costs and tradeoffs may produce code slower than that without reuse, regardless of what reuse scheme is being used. However, our model for intelligent memory reuse is flexible enough to allow such low level costs to be taken into account in an implementation.

6 Conclusions

Conceptually, there are two components to compile time memory reuse: it is necessary to determine, first, which memory cells may be safely reused at a particular program point; and second, how they are to be "best" reused. Most of the research on compile-time memory reuse has, to date, concentrated on the first component, typically via dataflow analysis: the underlying assumption seems to be that once this has been solved, dealing with the second component is straightforward. In this paper, we focus on the second component of the memory reuse problem. We give an abstract characterization of the problem, show that determining an optimal solution is NP-complete, and give an efficient polynomial-time heuristic using graph-matching techniques. Our algorithm produces optimal solutions for most commonly encountered memory reuse problems.

Acknowledgements: This paper has benefited greatly from discussions with A. Prasad and Will Winsborough. Comments by Evan Tick helped improve the presentation of the paper greatly. I am also grateful to Evan Tick and Nevin Heintze for providing the code for the benchmarks. This work was supported in part by the National Science Foundation under grant CCR-9123520.

References

[1] A. Bloss, *Path Analysis and Optimization of Non-strict Functional Languages*, PhD Dissertation, Dept. of Computer Science, Yale University, 1989.

[2] M. Bruynooghe, "Compile-Time Garbage Collection", *Proc. IFIP Working Conference on Program Transformation and Verification*, Elsevier, 1986.

[3] M. Bruynooghe, A. Mulkers, and K. Musumbu, "Compile-Time Garbage Collection for Prolog", Draft report, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium, 1988.

[4] I. Foster and W. Winsborough, "Copy Avoidance through Compile-Time Analysis and Local Reuse", *Proc. 1991 International Symposium on Logic Programming*, San Diego, Nov. 1991, pp. 455–469. MIT Press, Cambridge.

[5] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[6] D. Gudeman, K. De Bosschere, and S.K. Debray, "`jc`: An Efficient and Portable Sequential Implementation of Janus", *Proc. Joint International Conference and Symposium on Logic Programming*, Washington DC, Nov. 1992, pp. 399–413. MIT Press.

[7] N. Heintze, "Practical Aspects of Set Based Analysis", *Proc. Joint International Conference and Symposium on Logic Programming*, Washington DC, Nov. 1992, pp. 765–779. MIT Press.

[8] P. Hudak, "A Semantic Model for Reference Counting and its Abstraction", in *Abstract Interpretation of Declarative Languages*, ed. S. Abramsky and C. Hankin, Ellis Horwood, 1987, pp. 45–62.

[9] P. Hudak and A. Bloss, "The Aggregate Update Problem in Functional Languages", *Proc. Twelfth ACM Symposium on Principles of Programming Languages*, 1985, pp. 300–314.

[10] S. B. Jones and D. Le Metayer, "Compile-time Garbage Collection by Sharing Analysis", *Proc. Conference on Functional Programming Languages and Computer Architecture*, ACM Press, New York, 1989, pp. 54–74.

[11] F. Kluźniak, "Compile-time Garbage Collection for Ground Prolog", Draft Report, Institute of Informatics, Warsaw University, 1987.

[12] A. Mulkers, W. Winsborough, and M. Bruynooghe, "Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs", *Proc. Seventh International Conference on Logic Programming*, Jerusalem, June 1990, pp. 747–762. MIT Press, Cambridge.

[13] V. Saraswat, K. Kahn, and J. Levy, "Janus: A step towards distributed constraint programming", in *Proc. 1990 North American Conference on Logic Programming*, Austin, TX, Oct. 1990, MIT Press, Cambridge, pp. 431-446.

[14] A. V. S. Sastry and W. Clinger, "Order-of-Evaluation Analysis for Destructive Updates in Strict Functional Languages with Flat Aggregates", Technical Report CIS-TR-92-14, Dept. of Computer and Information Science, University of Oregon, July 1992.

[15] R. Sundararajan, A. V. S. Sastry, and E. Tick, "A Compile-Time Memory Reuse Scheme for Concurrent Logic Programs", *Proc. Joint International Conference and Symposium on Logic Programming*, Washington, D.C., Nov. 1992. MIT Press.

[16] T. J. Schaefer, "The Complexity of Satisfiability Problems", *Proc. Tenth Annual ACM Symposium on Theory of Computing*, ACM, New York, 1978, pp. 216–226.

[17] R. E. Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia, 1983.

[18] W. Winsborough, personal communication, Feb. 1992.