RC 21051 (11/7/97) Revised: 2/27/98
Computer Sciences/Mathematics

IBM Research Report

A Study of Dead Data Members in C++ Applications

Peter F. Sweeney, Frank Tip

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York
pfs@watson.ibm.com
tip@watson.ibm.com

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

A Study of Dead Data Members in C++ Applications

Peter F. Sweeney and Frank Tip

IBM Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598
{pfs,tip}@watson.ibm.com

Abstract

Object-oriented applications may contain data members that can be removed from the application without affecting program behavior. Such "dead" data members may occur due to unused functionality in class libraries, or due to the programmer losing track of member usage as the application changes over time. We present a simple and efficient algorithm for detecting dead data members in C++ applications. This algorithm has been implemented using a prototype version of the IBM VisualAge C++ compiler, and applied to a number of realistic benchmark programs ranging from 600 to 58,000 lines of code. For the non-trivial benchmarks, we found that up to 27.3% of the data members in the benchmarks are dead (average 12.5%), and that up to 11.6% of the object space of these applications may be occupied by dead data members at run-time (average 4.4%).

1 Introduction

Object-oriented applications may contain data members (instance variables) that can be removed from the application without affecting program behavior. Such "dead" data members may occur for several reasons:

- When an application uses a class library, it typically uses only part of the library's functionality. Certain members may be accessed only from the unused parts.
- The expected use of a class at design time may differ from the actual use of that class at coding time.
- Programmers may lose track of which members are used, due to the growing complexity of an application and its class hierarchy as the application changes over time.

This paper presents a simple and efficient algorithm that performs a whole-program analysis of a C++ application and identifies dead data members. The algorithm has been implemented in the context of the IBM VisualAge C++ compiler (version 4.0) [18], and has been applied to a set of medium-sized C++ applications ranging from 600 to 58,000 lines of code. For the non-trivial benchmarks, we found that up to 27.3% of the data members can be identified as dead, and that up to 11.6% of the object space is occupied by dead data members at run-time. On the average, we found that 12.5% of the data members are dead, and that 4.4% of object space is occupied by dead data members.

Elimination of unused data members is interesting from an optimization perspective because it reduces the amount of memory consumed by an application. An application's execution time may also be reduced, through reduced object creation/destruction time, and caching/paging effects. The detection of dead data members may also be useful in an integrated development environment, by providing feedback to the programmer, or by filtering out unimportant artifacts from an application.

The remainder of this paper is organized as follows. In Section 2, we define what it means for a data member to be live or dead. Section 3 presents our algorithm for detecting dead data members. Section 4 evaluates the algorithm on a set of medium size benchmarks. Section 5 presents related work and Section 6 presents our conclusions and future work.

2 Defining Liveness and Deadness

In the remainder of this paper, we will use the following intuitive definitions of "liveness" and "deadness" of data members. We call a data member m *live* if there is an object o in the program that contains m such that the value of m in o may affect the program's observable behavior (i.e., output or return value). If there is no such object o, we call m *dead*.

Note that, according to this definition, data members that are only accessed from unreachable code are classified as dead. More interestingly, data members that are assigned a value that is subsequently never used in the

```
class N {
public: int mn1;  /* live:  accessed and observable */
        int mn2;  /* dead:  not accessed */
};
class A {
public: virtual int f(){ return ma1; };
        int ma1;  /* live:  accessed and observable */
        int ma2;  /* dead:  not accessed */
        int ma3;  /* dead:  accessed but not observable */
};
class B : public A {
public: virtual int f(){ return mb1; };
        int mb1;  /* dead:  accessed from unreachable code */
        N mb2;    /* live:  accessed and observable */
        int mb3;  /* dead:  accessed, but not observable */
        int mb4;  /* live:  accessed and observable */
};
class C : public A {
public: virtual int f(){ return mc1; };
        int mc1;  /* dead:  accessed from unreachable code */
};

int foo(int *x){ return (*x) + 1; }
int main(){
        A a; B b; C c; A *ap;
        a.ma3 = b.mb3 + 1;
        int i = 10;
        if (i < 20){ ap = &a; } else { ap = &b; }
        return ap->f() + b.mb2.mn1 + foo(&b.mb4);
}
```

Figure 1: Example program.

program are classified as dead as well. We are particularly interested in detecting situations of this kind, since in real-life C++ programs, data members are typically initialized with a value in a constructor. Otherwise, the initialization of data members would lead to liveness, and very few dead data members would be dead.

Figure 1 shows an example C++ program. The following observations can be made about the data members in the class hierarchy of this program:

- `A::ma1` and `N::mn1` are *live* because the accesses to these data members affect the program's return value.

- `A::ma2` and `N::mn2` are *dead*, because there are no accesses to these data members in the program.

- `A::ma3` and `B::mb3` are *dead* since the accesses to these data members do not affect the program's return value.

- `B::mb1` and `C::mc1` are *dead*, because these members are accessed in unreachable code.

- `B::mb2` is *live*, because there is an access to `B::mb2` from which another member, `N::mn1` is accessed, which affects the program's return value.

- `B::mb4` is *live* because its address is taken, and subsequently used in a way that affects the program's return value.

Since the definition of liveness/deadness relies on the fact whether or not a given member access operation can be executed or not, the problem of precisely determining all dead data members is undecidable. Therefore, our analysis must be a conservative one. Below, we present an algorithm that conservatively approximates the set of dead data members: each data member classified by the algorithm as dead is guaranteed to be dead.

3 An Algorithm for Detecting Dead Data Members

In essence, our algorithm classifies a data member m as live if the value of m is read, or the address of m is taken. Since the act of storing a value into a data member cannot affect the program's observable behavior (i.e., output or return value) by itself, we ignore all write-accesses[1]. This analysis is performed on all functions and methods that can be transitively called from `main()`, thus ig-

[1] As an exception, data members that are `volatile` are marked live if they are written to.

noring member accesses that occur within unreachable functions. Conservative assumptions must be made in the presence of unsafe type casts, unions, and situations where the source code is only partially available (e.g., due to library usage).

Figure 2 shows pseudo code for the algorithm, expressed as a main procedure *DetectUnusedDataMembers*, and two auxiliary procedures *ProcessStatement* and *MarkAllContainedMembers*. The algorithm begins by marking all data members in all classes[2] as "dead" (line 3). On line 4, each class C in the program is marked as "not visited" (the purpose of this action will be explained shortly). Then, a call graph of the program is computed on line 5. We make no assumptions about the particular method used to determine the call graph, however, the accuracy of the call graph may have an impact on the precision of the analysis [11]. On lines 6–8, procedure *ProcessStatement* is invoked for each statement that occurs in a function reachable from `main()` in the call graph.

For a given statement s, procedure *ProcessStatement* (lines 15–35) determines all data members that are read in s, and all data members whose address is taken in s, and marks them "live". For each expression e in statement s that is not a call to the `delete` or `free` system functions[3] (line 18), the following actions are performed:

- If e is a read-access of the form $e'.m$, function *Lookup* is invoked to determine which data member C::m is accessed, and C::m is marked "live" (lines 19–22). Read accesses of data members using the `->` operator are treated similarly. For "address-taken" expressions of the form &$e'.m$, we conservatively assume that the data member may be read through some pointer in the program if its address is taken—we do not attempt to trace the use of such addresses. We make no assumptions about the method used to implement *Lookup*; an efficient member lookup algorithm for C++ was recently presented in [16]. Qualified read-accesses of data members are handled in a similar manner.

- If e is an expression of the form &Z::m (i.e., the offset of member m within class Z is computed), function *Lookup* is invoked to determine data member C::m that is accessed, and this C::m is marked "live" (lines 26–28). Expressions of the form &Z::m are typically used in conjunction with *pointers to members*, a somewhat obscure C++ feature for indirectly accessing members from a specified object. We do not attempt to trace where a pointer-to-member expression is used, and simply assume that any member whose offset is computed may be accessed somewhere in the program.

- If e is an unsafe[4] type cast expression of the form $(T)(e')$, for some type T that is not necessarily a class (lines 29–32), the type S of subexpression e' is determined, and all members contained within type S are marked "live". We make this conservative assumption because a read access to type T implies a read-access to some member of S. The data members of S are marked "live" by calling procedure *MarkAllContainedMembers* for type S. *MarkAllContainedMembers* (lines 36–50) marks all directly or indirectly contained data members of a class C; see Figure 2 for details. In order to prevent duplication of work, a class C is marked "visited" if *MarkAllContainedMembers* is called for C, and all actions in *MarkAllContainedMembers* are only performed for classes that were not yet visited (see line 38). All classes are initially marked "not visited" on line 4.

After processing the statements, the `union` constructs in P are examined. If a `union` U contains at least one data member that is marked "live", all other members that are directly or indirectly[5] contained in U are marked "live" (lines 9–11) by calling *MarkAllContainedMembers*. This conservative assumption is required because the value of a live union data member might otherwise depend on a write-access to a dead union member.

3.1 Example

We will now study how our algorithm analyzes the example program of Figure 1. Initially, all data members of the program are marked "dead". If we assume that the algorithm of [5] is used to construct a call graph, the call graph consists of the methods `A::f`, `B::f`, and `C::f` in addition to `main`. Analysis of the statements in the functions in the call graph proceeds as follows:

[2] This includes all data members in `struct`s and `union`s, which are defined as special cases of `class`es in C++ [1].

[3] A data member whose address is passed to the `delete` or `free` system functions does not have to be marked as live, because these functions do not affect the program's observable behavior. We are particularly interested in detecting such situations, since data members that are pointers to objects are typically passed to `delete` in the enclosing class's destructor. Other system functions (e.g., `strcpy`) that are known not to affect some of their parameters, could be treated as a special case as well.

[4] For the purposes of this paper, a type cast from type S to type T is considered "unsafe" if T is a derived class of S and the object being cast cannot be guaranteed to be a of type T at run-time. We have verified that all down-casts in our benchmarks are safe. In general, unsafe type casts are unlikely to occur, but this is something the user of the tool has to verify.

[5] A `union` construct may contain data members whose type is a class (although there are restrictions on such classes [1]), and these classes may contain data members, or have base classes that contain data members.

```
procedure DetectUnusedDataMembers(Program $P$)
begin
  mark all data members in $P$ initially as “dead”;
  mark all classes in $P$ initially as “not visited”;
  construct the call graph $G$ of program $P$;
  for each statement $s$ in each function $f$ in call graph $G$ do
    call ProcessStatement($s$);
  end for
  for each union construct $U$ in $P$ do
    if (at least one of the members of $U$ is marked “live”) then
      call MarkAllContainedMembers($U$);
    end if
  end for
end;

procedure ProcessStatement(Statement $s$)
begin
  for each expression $e$ in statement $s$ do
    if ($e$ is not a call to the system functions delete or free) then
      if ($e$ is an expression of the form $e'.m$ and is a read-access) or
        ($e$ is a expression of the form &$e'.m$) then
/* access to data member from expression. similar for -> expressions. */
        let $X$ be the type of $e'$;
        let $C$ = Lookup($X$, $m$);        /* $m$ may occur in a base class of $X$ */
        mark data member $C$::$m$ “live”;
      else if    ($e$ is an expression of the form $e'.Y$::$m$ and is a read-access) or
                 ($e$ is a expression of the form &$e'.Y$::$m$) then
/* access to data member from expression using ‘::’ operator. similar for -> expressions. */
        let $C$ = Lookup($Y$, $m$);        /* $m$ may occur in a base class of $Y$ */
        mark data member $C$::$m$ “live”;
      else if ($e$ is an expression of the form &$Z$::$m$) then
/* pointer-to-member expression. */
        let $C$ = Lookup($Z$, $m$);        /* $m$ may occur in a base class of $Z$ */
        mark data member $C$::$m$ “live”;
      else if ($e$ is an unsafe type cast expression of the form $(T)(e')$, for some type $T$) then
        let $S$ be the type of $e'$;
        call MarkAllContainedMembers($S$);
      end if
    end if
  end for
end;

procedure MarkAllContainedMembers(Class $C$);
begin
  if class $C$ was marked “not visited” then
    mark class $C$ as “visited”;
    for each data member $m$ of $C$ do
      mark data member $C$::$m$ “live”;
      if the type of data member $m$ is a class $N$ then
        call MarkAllContainedMembers($N$);
      end if
    end for
    for each direct base class $B$ of $C$ do
      call MarkAllContainedMembers($B$);
    end for
  end if
end;
```

Figure 2: Algorithm for detecting unused data members.

- A::ma1, B::mb1, and C::mc1 are marked live, because their value is is read in the return statements of A::f, B::f, and C::f, respectively.

- Since data member B::mb3 is read in main(), it is classified as "live",

- The expression b.mb2.mn1 in main's return statement reads the values of both B::mb2 and N::mn1. Therefore, both B::mb2 and N::mn1 are marked live.

- Since the address of data member B::mb4 in object b is taken in the return expression of main, B::mb4 is classified as live.

Note that, although A::ma3 is accessed in main(), it is written to, not read. Therefore, A::ma3 is *not* classified as live. Due to the conservativeness of the algorithm, three dead data members are marked "live". B::mb3 is classified as live because it is read, even though it does not affect the program's return value. B::mb1 and C::mc1 are marked live because methods B::f and C::f are identified as reachable functions.

However, if a more accurate call graph is used, we can achieve better results. For example, a simple alias/points-to analysis algorithm [7, 15, 20, 17] can determine that pointer ap never points to a C object. This fact can be used to exclude method C::f from the call graph, so that the reference to C::mc1 can be disregarded, and data member C::mc1 can be marked "dead".

An even more ambitious approach would be to eliminate dead code prior to running our algorithm. For example, constant propagation [24] can be used to determine that the else-branch of the if statement is unreachable, enabling us to remove method B::f from the call graph, which would result in B::mb1 being classified as "dead". Program slicing [25, 21] may also be used to remove useless code from an application prior to running the algorithm.

3.2 The sizeof operator

The sizeof operator, which returns the size of an object or type as a number of bytes, can be used in different ways, and may or may not affect the program's observable behavior. If program behavior is affected, all data members in the affected classes must be marked live, otherwise the use of sizeof can be ignored. Since the effects of sizeof cannot easily be determined automatically, our approach is that the user must specify which uses of sizeof can be ignored; by default, sizeof is treated conservatively.

In the current set of benchmarks, sizeof is only used for storage allocation purposes, and does not affect the program's observable behavior. Therefore, we do not mark any data members as live due to use of sizeof.

3.3 Dealing with Library Usage

Situations where the source code and class hierarchy for parts of the program are unavailable due to library usage require special care. In general, it is not possible to classify a data member in a library class as "dead" or "live" unless the complete source code for the library is available. In particular, it is not possible to classify the data members of a library class C for which header information and method bodies are available if we do not have access to all library source code in which C's data members may be accessed.

A data member in a user class derived from a library class can be classified, assuming that the execution of the library code cannot result in an access to that data member. This implies that conservative assumptions must be made during the construction of the call graph in the presence of libraries, since a library may make calls to virtual methods in the user's code; similar precautions must be taken if the library calls methods indirectly through function pointers (callbacks). Such situations can be dealt with conservatively as follows. If a virtual method f is defined in a library class, and f is overridden in a class C in the application code, we assume $C :: f$ to be reachable. In addition, if the address of a function f is taken in reachable code, we assume f to be reachable.

3.4 Complexity Analysis

Our algorithm requires the construction of a call graph, and relies on an algorithm for performing member lookup. Using the Rapid Type Analysis algorithm of [5], a call graph can be constructed in linear time in practice [11]; for more sophisticed call graph construction algorithms, we refer the reader to [11].

Using the member lookup algorithm of [16], all member lookups can be computed in time $O(M \times (C + I))$, where C is the number of classes in the hierarchy, I the number of inheritance relations among these classes, and M the number of distinct member names (assuming that the program contains no ambiguous member lookups).

Assuming that the call graph and all member lookups have been pre-computed, our algorithm requires a single traversal of the expressions that occur in reachable functions. All actions performed for each expression can be performed in unit time, with the exception of calls to procedure *MarkAllContainedMembers*. The total amount of time spent in all calls to this procedure is $O(C \times M)$, assuming that all members in all classes are eventually visited and marked. This implies that, excluding the cost of pre-computing member lookups and

Benchmark	Description	LOC	Classes	Data
			total (used)	Members
jikes	IBM Java to byte code translator	58,296	268 (246)	1052
idl	SunSoft IDL compiler + demo back end	30,288	85 (82)	118
npic	Framework for alias analysis	22,728	198 (184)	290
lcom	"L" hardware description language compiler	17,278	78 (73)	287
taldict	Taligent's synthetic dictionary benchmark	11,854	55 (13)	22
ixx	IDL specification to C++ code generator	11,157	101 (81)	343
simulate	Simulation class library + example	6,672	42 (27)	46
sched	RS/6000 instruction timing simulator	5,712	46 (44)	186
hotwire	Scriptable graphical presentation builder	5,355	37 (21)	166
deltablue	Incremental dataflow constraint solver	1,250	10 (8)	23
richards	Simple operating system simulator	606	12 (12)	28

Table 1: Benchmark programs used to evaluate the dead data member detection algorithm. The columns of this table show the name of the application, a brief description of the application, the size of the application (in lines of source code), the number of classes in the application's class hierarchy; the number between brackets is the number of "used" classes (i.e., classes for which a constructor is called in user code), and the number of data members that occurs in used classes, respectively.

construction of the call graph, the total cost of our algorithm is $O(N + (C \times M))$, where N is the number of expressions in the program.

4 Results

The algorithm of Section 3 has been implemented in the context of the IBM VisualAge C++ compiler (version 4.0) that is currently being developed jointly by IBM Research and IBM Toronto. We use a slightly modified version of the Program Virtual Call Graph (PVG) algorithm [4] to build a call graph of a C++ application. For resolving member lookups, we rely directly on the information provided by the compiler. Unfortunately, there is no linguistic means to detect whether or not a class occurs in a library. Therefore, we rely on the user to indicate which classes are library classes.

We applied the dead data member detection algorithm to a small set of medium-sized C++ benchmarks in order to answer the following questions:

1. What percentage of data members in an application can be determined to be dead?

2. What percentage of object space is occupied at runtime by dead data members?

The first question is answered directly by our algorithm, as will be discussed below in Section 4.2. The answer to the second question is obtained by analyzing the objects created during program execution, and measuring the amount of space in these objects occupied by dead data members; this is done by a combination of code instrumentation and analysis of a dynamic trace of the execution [14]. The dynamic measurements will be discussed in Section 4.3.

4.1 Benchmark Characteristics

Table 1 shows the set of benchmark programs that were used to evaluate the dead data member detection algorithm. The columns of the table show for each benchmark: the name of the application, a short description, the size (number of lines of source code), the total number of classes, and the number of "used" classes (i.e., classes for which a constructor call occurs in the application), and the number of data members that occur in used classes.

Several of these benchmarks have been studied previously in the literature for other purposes (e.g., experimentation with virtual function-call elimination algorithms) [5, 9, 8, 6, 12, 3]. The programs of Table 1 range from 606 to 58,296 lines of code, and contain between 10 to 268 classes, and between 22 and 1052 data members. Some benchmarks (e.g., `taldict`, `simulate`, and `hotwire`) use class libraries that have been developed independently from the application. Several other benchmarks (e.g., `idl`, `lcom`, `ixx`, and `sched`) use classes that were custom-built for the application. The code for all of these classes is available for analysis, and the results presented below only apply to application code for which the full source code is available. In addition, all benchmarks rely on low-level libraries (e.g., `iostream.h`), for which the source code is unavailable or only partially available. In the computation of the numbers below, classes and data members in such libraries are ignored.

Besides being of different sizes, the benchmark programs also cover a wide range of programming styles. The sched benchmark, for example, is not written in a very object-oriented style, and contains very little inheritance: most of the classes are structs. On the other hand, idl is a highly object-oriented application with a complex class hierarchy and heavy use of virtual functions and virtual inheritance.

4.2 Static Measurements

Figure 3 shows the percentage of dead data members in the used classes for the benchmark programs. The percentages shown in this figure are unweighted in the sense that they do not take into account the size of each data member. We believe that taking the size of data members into account for the static measurements is not meaningful, because there is no way to take into account statically how many times each class is instantiated. Data members in unused classes are ignored in the computation of the percentages, since eliminating such members does not affect the size of any objects that are created at run-time.

In the smallest two of the benchmarks, deltablue and richards, no dead data members were found. For the other benchmarks, the percentage of dead data members varies from 3.0% to 27.3%. Not surprisingly, the largest percentage of unused data members is found in the programs that use class libraries: taldict, simulate, and hotwire. However, our measurements indicate that even in applications with a custom-built class hierarchy, the amount of redundancy can be considerable.

4.3 Dynamic Measurements

Table 2 shows the relevant execution characteristics for each of the benchmark programs. The columns in the table show the amount of space occupied by objects throughout program execution[6], the amount of space occupied by dead data members in these objects, the maximum amount of space occupied by objects at a single point in time during execution (the "high water mark"), and the high water mark if dead data members are eliminated from objects. Note that, in general, these two high water marks may occur at different execution points.

Figure 4 shows the percentage of object space occupied by dead data members at run-time for each of the benchmarks. The figure shows two percentages for each benchmark:

- The leftmost (light grey) bar indicates the number of bytes in objects occupied by dead data members, as a percentage of the total number of bytes occupied by objects.

- The rightmost (dark grey) bar indicates a percentage of the reduction in size of the original high water mark, if all dead data members were to be eliminated.

Both figures take into account the size of each data member, as well as the number of times an object is created.

Interestingly enough, there is no strong correlation between a high percentage of dead data members in Figure 3, and a high percentage of object space occupied by those data members in Figure 4. Another point to note is that, for a number of benchmarks, the high water mark numbers are (nearly) identical to the numbers for total object space. This situation occurs when an application heap-allocates most objects, and does not deallocate them until the end of program execution.

4.4 Evaluation

Although the number of benchmarks we used is relatively small, some interesting observations can be made.

- The smallest two benchmarks, richards and deltablue, do not contain any dead data members. This is in line with our expectation that it is unlikely that many dead data members will occur in small programs.

- The benchmarks that use a class library not specifically built for the application, taldict, simulate, and hotwire, have the highest percentage of dead data members. This confirms our intuition that dead data members may arise due to unused library functionality.

- For some benchmarks with a high percentage of dead data members, the space occupied by these data members at run-time is relatively small. In such cases, classes with dead data members are instantiated infrequently.

- Even in applications with custom-built class hierarchies, the amount of dead data members is non-negligible.

- Unfortunately, we have limited data on the development history of our benchmarks. Nevertheless, we believe that applications that have a long maintenance history and/or have multiple successive or concurrent developers could accumulate many dead data members.

For the nine nontrivial benchmarks, the average percentage of dead data members is 12.5%, resulting in

[6] We assume that the heap allocator always allocates the exact number of bytes that is requested.

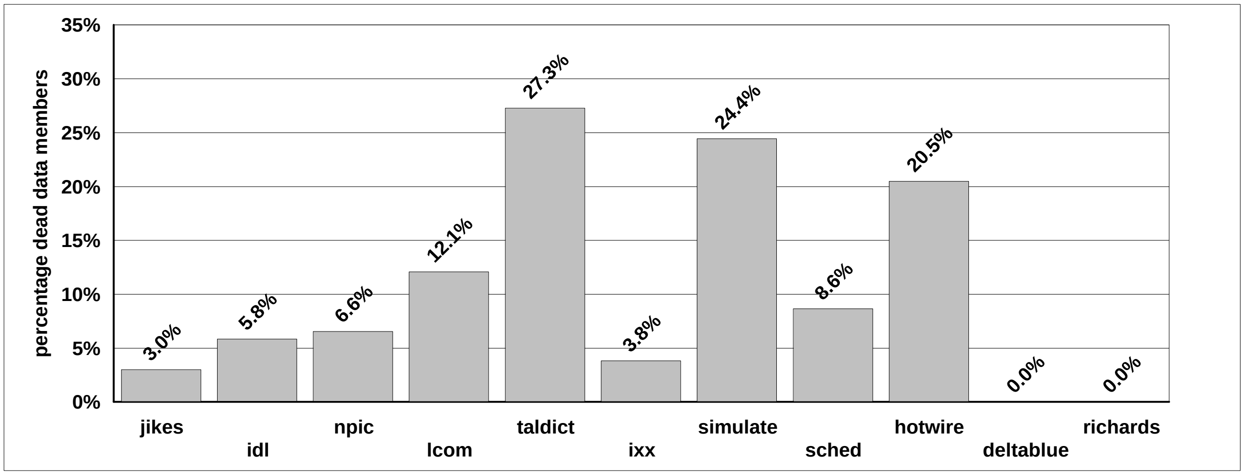

Figure 3: Percentage of dead data members detected in the benchmark programs of Table 1.

Benchmark	Object Space	Dead Data Member Space	High Water Mark	High Water Mark w/o dead data members
jikes	2,921,490	55,112	2,232,472	2,179,730
idl	708,249	15,388	701,273	685,885
npic	115,248	5,516	24,972	23,840
lcom	2,274,956	241,435	1,652,828	1,491,048
taldict	7,080	36	7,008	6,972
ixx	551,160	29,745	299,516	269,775
simulate	54,869	41	11,585	11,544
sched	9,032,676	1,049,148	9,032,676	7,983,528
hotwire	10,780	284	10,780	10,496
deltablue	276,364	0	196,212	196,212
richards	4,880	0	4,880	4,880

Table 2: Execution characteristics of the benchmark programs of Table 1. The table shows for each benchmark: the space occupied by objects created during execution, the space occupied by dead data members in objects created during execution, the high water mark (i.e., maximum amount of space occupied by objects at a single point in time during execution), and the high water mark if dead data members are eliminated from objects. All measurements are in bytes.

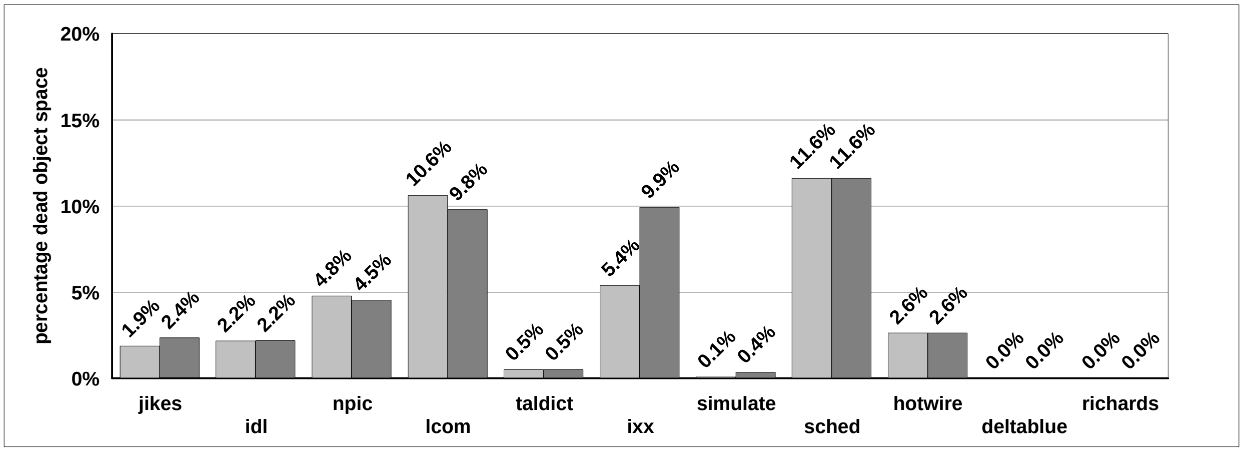

Figure 4: Percentage of object space occupied by dead data members for each of the programs of Table 1. Light grey bars indicate the percentage of space occupied by dead data members throughout program execution. Dark grey bars indicate the reduction of the maximum amount of space (high water mark) required at a single point in time by each program.

an average space savings of 4.4% at run-time if these members are removed (4.9% for the high water mark number). Given the simplicity of the algorithm, we believe that this optimization should be incorporated in any optimizing compiler.

5 Related Work

Agesen and Ungar [2] describe an algorithm for the Self language that eliminates unused slots from objects (a slot corresponds to either a data member or a method). This algorithm computes, for each message send (method call) that may be executed, a set of slots that is needed to preserve that send's behavior, and produces a source file in which redundant slots have been eliminated. In spirit, this work is very closely related to ours, although the details of the languages under consideration are very different. Self is a dynamically typed language without an explicit class hierarchy in which objects are obtained by cloning other objects. In statically typed languages such as C++, objects are created by instantiating classes. In addition, C++ is a much larger language than Self with a number of features that require special attention when determining dead instance variables.

In the context of C++, previous research has focused on the issue of determining and eliminating unused methods, and the usefulness of these optimizations has been demonstrated [5, 19].

The work described in the present paper was motivated in part by previous work for removing unused data members and inheritance relations from C++ class hierarchies [22, 23]. *Class hierarchy slicing* [22] is capable of eliminating unused inheritance relations in addition to classes, data members and methods. For the example program of Figure 1, class hierarchy slicing would be able to eliminate the unnecessary inheritance relation between class `C` and class `A`. This would result in the elimination of the `A`-subobject that contains data member `ma1` from object `c`. Class hierarchy slicing relies on alias/points-to information [7, 15, 20, 17] to resolve the potential receivers of virtual method calls. *Class hierarchy specialization* [23] is capable of making finer distinctions than class hierarchy slicing by constructing a new class hierarchy in which variables that previously had the same type X may obtain different types. As a result, data members may be excluded from certain X-objects while being retained in other X-objects. Like class hierarchy slicing, class hierarchy specialization requires alias/points-to information. Class hierarchy specialization is also capable of simplifying complex inheritance structures, in particular eliminating virtual inheritance. Virtual inheritance is typically implemented by using indirections in objects, which increase member access time, and which may increase object size, depending on the object model that is used. Unfortunately, neither class hierarchy slicing nor class hierarchy specialization have been implemented yet. It would be interesting to compare the results of these algorithms to the results presented in this paper.

Live variable analysis is a data flow analysis technique for determining if the value of a variable along any path is read before it is re-written [10]. This analysis is typically used to eliminate redundant writes: if a write to a variable is never read, then the write can be removed. The analysis described in this paper operates in a completely different domain, the removal of dead components from objects, and requires no flow-analysis.

In their study of abstract models of memory management, Morrisett et al. [13] provide a semantic definition of *reachable garbage* that is similar in spirit to our notion of liveness. Specifically, they observe that certain reachable heap-values cannot affect program behavior. Based on this observation, Morrisett et al. propose a type-inference algorithm that infers a type for each heap location; if an unconstrained type variable is inferred, that location can be replaced by an arbitrary value (i.e., "collected"). Our analysis for finding data members that are accessed (reachable) but dead is trivial: A data member is dead if it is only written to. We consider the combination of our algorithm with more advanced techniques for eliminating useless code (e.g., program slicing) a promising direction for future work.

6 Conclusions

We have presented a simple and efficient algorithm for detecting dead data members in C++ applications. This algorithm can be used as the basis for a space optimization performed by an optimizing compiler, or as a component of a program maintenance/understanding tool.

The algorithm has been evaluated using a set of realistic benchmark programs ranging from 600 to 58,000 lines of code. We found that in the nontrivial benchmarks, up to 27.3% of data members is dead, and that up to 11.6% of the object space of these applications may be occupied by dead data members at run-time. On the average, 12.5% of the data members are dead, and 4.4% of object space is occupied by dead data members. Evaluation of these measurements is in agreement with our belief that the use of selected parts of a general class library may give rise to redundant data members in objects.

Acknowledgements

We would like to thank David Bacon, Michael Burke, John Field, David Grove, Michael Karasick, G. Ramalingam, and Mark Wegman for many helpful comments and suggestions.

References

[1] ACCREDITED STANDARDS COMMITTEE X3, I. P. S. Working paper for draft proposed international standard for information systems—programming language C++. Doc No X3J16/96-0219R1. Draft of 2 december 1996.

[2] AGESEN, O., AND UNGAR, D. Sifting out the gold. Delivering compact applications from an exploratory object-oriented programming environment. In *Proceedings of the 1994 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'94)* (Portland, OR, Oct. 1994), pp. 355–370.

[3] AIGNER, G., AND HÖLZLE, U. Eliminating virtual function calls in C++ programs. In *Proceedings of the Tenth European Conference on Object-Oriented Programming – ECOOP'96* (Linz, Austria, July 1996), vol. 1098 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 142–166.

[4] BACON, D. F. *Fast and Effective Optimziation of Statically Typed Object-Oriented Languages*. PhD thesis, University of California at Berkeley, Dec 1997. Forthcoming.

[5] BACON, D. F., AND SWEENEY, P. F. Fast static analysis of C++ virtual function calls. In *Proceedings of the 1996 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, Oct. 1996), pp. 324–341.

[6] CALDER, B., AND GRUNWALD, D. Reducing indirect function call overhead in C++ programs. In *Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages (POPL'94)* (Portland, Oregon, Jan. 1994), pp. 397–408.

[7] CARINI, P. R., HIND, M., AND SRINIVASAN, H. Flow-sensitive type analysis for C++. Tech. Rep. RC 20267, IBM T.J. Watson Research Center, 1995.

[8] DEAN, J., DEFOUW, G., GROVE, D., LITVINOV, V., AND CHAMBERS, C. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of the 1996 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, Oct. 1996), pp. 83–103.

[9] DRIESEN, K., AND HÖLZLE, U. The direct cost of virtual function calls in C++. In *Proceedings of the 1996 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, Oct. 1996), pp. 306–323.

[10] FISCHER, C. N., AND RICHARD J. LEBLANC, J. *Crafting A Compiler*. The Benjamin/Cummings Series in Computer Science. Benjamin/Cummings, Menlo Park, CA, 1988.

[11] GROVE, D., DEFOUW, G., DEAN, J., AND CHAMBERS, C. Call graph construction in object-oriented languages. In *Proceedings of the 1997 ACM Conference on Object Oriented Programming Systems, Languages, and Applications OOPSLA* (Oct. 1997), pp. 108–124.

[12] LEE, Y.-F., AND SERRANO, M. J. Dynamic measurements of C++ program characteristics. Tech. Rep. ADTI-1995-001, IBM Santa Teresa Laboratory, Jan. 1995.

[13] MORRISETT, G., FELLEISEN, M., AND HARPER, R. Abstract models of memory management. In *Functional Programming and Computer Architecture* (La Jolla, CA, June 1995), ACM, pp. 66–77.

[14] NAIR, R. Profiling IBM RS/6000 applications. *International Journal of Computer Simulation 6*, 1 (1996), 101–111.

[15] PANDE, H. D., AND RYDER, B. G. Static type determination and aliasing for C++. Report LCSR-TR-250-A, Rutgers University, October 1995.

[16] RAMALINGAM, G., AND SRINIVASAN, H. A member lookup algorithm for C++. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation* (Las Vegas, NV, 1997), pp. 18–30.

[17] SHAPIRO, M., AND HORWITZ, S. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the Twenty-Fourth ACM Symposium on Principles of Programming Languages* (Paris, France, 1997), pp. 1–14.

[18] SOROKER, D., KARASICK, M., BARTON, J., AND STREETER, D. Extension mechanisms in Montana. In *Proceedings of the 8th IEEE Israeli Conference on Software and Systems (Herzliya, Israel)* (June 1997), IEEE Computer Society, pp. 119–128.

[19] SRIVASTAVA, A. Unreachable procedures in object-oriented programming. *ACM Letters on Programming Languages and Systems 1*, 4 (December 1992), 355–364.

[20] STEENSGAARD, B. Points-to analysis in almost linear time. In *Proceedings of the Twenty-Third ACM Symposium on Principles of Programming Languages* (St. Petersburg, FL, January 1996), pp. 32–41.

[21] TIP, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (1995), 121–189.

[22] TIP, F., CHOI, J.-D., FIELD, J., AND RAMALINGAM, G. Slicing class hierarchies in C++. In *Proceedings of the 1996 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'96)* (San Jose, CA, Oct. 1996), pp. 179–197.

[23] TIP, F., AND SWEENEY, P. F. Class hierarchy specialization. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)* (Atlanta, GA, 1997), pp. 271–285. ACM SIGPLAN Notices 32(10).

[24] WEGMAN, M., AND ZADECK, F. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems 13*, 2 (1991), 181–210.

[25] WEISER, M. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, 1979.