Experience with Representing C++ Program Information in an Object-Oriented Database

Tamiya Onodera

IBM Research, Tokyo Research Laboratory
1623-14, Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242 Japan

Abstract

Two major issues related to storing program information in an OODB are sharing and clustering. The former is important since it prevents the database from consuming excessive disk space, while the latter is crucial, since it keeps clients running without thrashing. In our database, objects are shared across multiple programs' translation units, and are clustered by combining three techniques, namely, birth-order, death-order, and sharing-oriented clusterings. An initial experiment shows that, for a medium-size application, the database consumes 3.5 times less disk space than in a conventional environment, and that the invocation of a client is almost instantaneous.

1 Introduction

The growing size and complexity of software has intensified the need for advanced programming environments that can reduce the burden on software developers. Such an environment should integrate tools, including a compiler, a debugger, a source-code browser, and a builder. *Tool integration*, in this narrow context of lower CASE, means integration of presentation, control, and data [1]. To build such an integrated environment, it is believed, the correct strategy is to use a database management system.

OOPSLA 94- 10/94 Portland, Oregon USA

Our attempt to represent C++ program information by using an object-oriented database follows these lines, but at the same time we have aimed to alleviate two serious and pervasive problems in the development of programs: *bloated working storage* and *stalled invocation.* In the first, a tremendously large amount of disk space is used to store information for tools, and in the second, an intolerably long time is spent before a tool is ready to accept user interaction.

As an example, we present statistics on a Motif application that is being developed at our laboratory by using IBM C Set ++ for AIX/6000. The numbers given below were all obtained under AIX 3.2.4 on a RISC System/6000 Model 560. When compiled and linked without any specific options, the library, which consists of 300 source files and contains a total of 15,541 lines, consumes 3.03 megabytes of disk space, including source files, object files, and an executable. To use a debugger, we must build the executable with the debugger option specified. This requires 16.86 megabytes more disk space. When we invoked the dbx debugger against the executable, it took 479 seconds of real time for the debugger process to show its first prompt. To allow use of a browser, we must build the executable with the browser option specified. This, surprisingly, requires 76.92 megabytes more disk space. Moreover, the system configuration did not allow us to use the source code browser; the initialization of the browser process failed, because of the heavy demand on paging space.

The reason for bloated working storage is as fol-

lows. The use of a library in a C/C++ program results in one or more header files being included, and the inclusion of a header file contributes to increased consumption of disk space; information on types defined in the header file is generated if the program is compiled with the debugger option, while information on cross-references that originate in the header file is produced if the program is compiled with the browser option. When a header file is included in many source files, the same number of duplicates result, and are simply left to occupy disk space. What really aggravates the duplication is that modern software tends to or is encouraged to rely on more and more libraries, some of which are very large. For instance, the source file in the Motif application contains only 51.8 lines on average. However, after the file has been preprocessed, the average number of lines rises to as many as 9,153, since all the source files but one include `Xm.h` directly or indirectly. On the other hand, the reason for stalling invocation is simply that the `dbx` debugger and the source code browser eagerly attempt to initialize a large number of objects before they accept user interaction.

We can amend "bloated working storage" by enhancing the compiler to allow it to populate a database with objects, sharing as many objects as possible. When the compiler is about to store an object in a database, it first checks whether the database already contains any equivalent object and, if not, adds it to the database. On the other hand, we can solve "stalled invocation" by somehow realizing lazy evaluation. For instance, it may be promising to use memory mapping facilities if they are available; actually, there is a debugger that does not suffer from the problem. However, a much simpler and cleaner way of realizing lazy evaluation is to rely on a DBMS that allows objects to be fetched on demand from the persistent storage. Fortunately, the OODBMS used in our project, ObjectStore, is known for its notable use of virtual memory mapping architecture [2, 3]; when a client accesses a database object for the first time, the OODBMS brings the entire page containing the object into the client's virtual memory.

The following sections are organized as follows. Section 2 overviews C++ program databases. Section 3 describes how and what objects are shared, primarily to solve "bloated working storage." Section 4 is on clustering, which is crucial to the database performance, even if objects are fetched on demand. Section 5 gives initial performance measurements, while Section 6 deals with related work. Finally, Section 7 presents our conclusions and discusses future work.

2 C++ Program Database

The program database we are attempting to build stores *static* information on C++ programs. This includes build relationships of programs and cross-references of symbols appearing in programs. An important feature is that a single database holds information on *multiple* programs. This is essential to eliminate the problem of bloated working storage; if we built a database for each program, it would prevent information originating in `Xm.h` from being shared among different Motif applications.

2.1 Basic Structure

Static information on C++ programs is represented as graphs of objects, which are grouped into four categories: *files*, *cross-references* (abbreviated below as xrefs), *symbols*, and *types*. A file object is constructed for each of the operating system's files that have participated in building C++ programs. File objects form graphs to represent build dependency, or relationships showing which files are linked into which file and which file includes which files.

A primary ingredient of the program database is an *xref graph*, namely an xref object pointing to a symbol object that may point to a type graph; the symbol points to a type graph only if it has a type. This xref graph denotes which symbol optionally having which type is defined or declared or used in which *location*; logically, the location is a triple of file object, line number, and column number, but physically, it is squeezed into a pair,

```
// a.C
int x;
class A { char x; unsigned y; };
double foo(float a){  return x+a; }
```

Figure 1: A Sample Program to Show the Basic Structure of the Program Database

as we will see in Section 3. These xref graphs are then organized according to the block structure of a C++ program.

Consider the following source file as an example. Its compilation results in the population of objects shown in Figure 2. The block structure of the translation unit is represented as a graph formed by three objects: a file object corresponding to `a.o`, an xref object corresponding to the class definition, and an xref object corresponding to the function definition. Each of these has a pointer member to an array of the pointers to the xref graphs. All the xrefs of the translation unit, including those for the class and function definitions, are organized into the arrays. Note that not all the objects are shown in the figure; the database is also populated with special cross-references, which are introduced below. Not all the pointers are drawn in the figure, either; backward pointers, which allow fast climbing-up traversal, are also present.

2.2 Intended Clients

The intended tools are initially a debugger, a source-code browser, a builder, which is similar to the **make** utility, and a profiler. Actually, the information we stored in a database reflects this. First, the database stores *code-static* information as used by a debugger. Symbol objects have members for relative addresses, offsets within stack frames, or offsets within data layouts of classes. Special xrefs are provided for storing information on breakpoints.[1] Second, the database does not contain parse trees, such as those generated by a compiler's front-end, since it is not intended for code generation and program transformation. Note, however, that enough information is stored for incremental compilation or program slicing. Finally, although the database contains sufficient information to instrument code for execution profiling, it is not supposed to contain the results of profiling *per se*. This does not necessarily exclude the population of such profiling results; we believe that static and dynamic program information can be separate islands in a database, and this paper simply focuses on the population of static information.

The only intended populator is a compiler. To facilitate turning a conventional compiler into a populating compiler, we define a *population interface*, or a collection of methods encapsulated into a class. Following the definition of the interface, calls to the methods are inserted at appropriate locations in an existing compiler's source code. Linked with an implementation of the interface class, it is made a populating compiler.

3 Sharing

One purpose of sharing is to alleviate the problem of "bloated working space," and xrefs from commonly included header files must be shared. Sharing xrefs implies sharing symbols pointed to by them. Putting it another way, sharing symbols is considered to facilitate the sharing of xrefs. Obviously, sharing symbols can and should be based on the C++ semantics of linkage. However, storing multiple programs' information in a single database presents a significant challenge to this, as we will discuss.

3.1 Objects Shared

The entities that can be shared among translation units in our program database are *global symbols*, *global types*, and *global xrefs*. A symbol is said to be global if it is not a member or local. For a global symbol to be shared, it must represent what is common across translation units. For comparison, assume that the data structure of a function

[1] Since an xref essentially carries information on its location, this scheme can accommodate breakpoints at intervals with a finer granularity than lines.

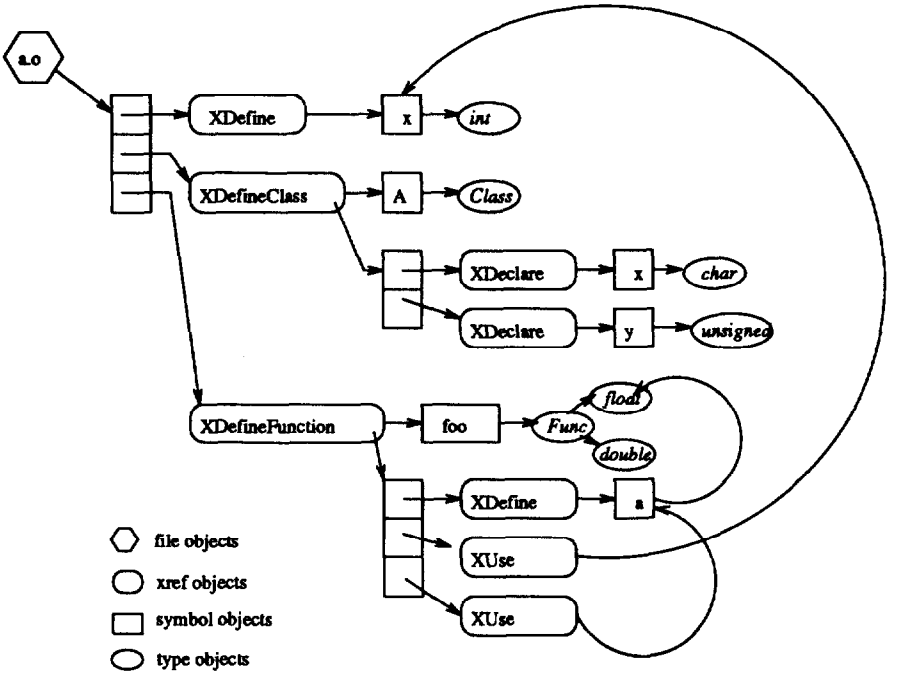

Figure 2: Basic Structure of the Program Database: only the relevant values are shown within the objects.

symbol is defined to have a member that points to a graph representing its definition. This is often the case in internal data structures of a conventional compiler, since at most one definition is legally given to a function symbol in a translation unit. However, this is no longer allowed in our database setting, and thus the function symbol shown in Figure 2 results. An alternative would be to have a function symbol's member point not to a definition but to a list of definitions. However, doing so might slightly degrade performance, since inserting a new definition results in the updating of an existing object, such as a list node; in general, modifying existing objects is more costly than simply constructing new objects.

A type is said to be global if it is not locally defined or a derived type constructed from a locally defined type. We attempt to share derived types as well as fundamental types. Thus, only a single representation exists in a database for each of `int`, `char*`, `double (float)`, `int (*[])(int, char*)`, and so forth.

An xref is said to be global if it occurs with file scope. Xrefs are our primary targets for sharing, since this eliminates duplicated objects due to commonly included header files. Though the preprocessor's conditional commands might cause the compiler to attempt to store different sets of objects for different inclusions of a header file, only objects corresponding to the delta are added. Note, however, that the granularity of sharing is a global xref; when a conditional compilation eliminates the text of a declaration of a member, the xref graph representing the definition of the member's class is newly stored in its entirety.

The compilations of two translation units in Figure 3 populate a database with objects, as in Figure 4. We assume that `a.o` and `b.o` are compiled in this order.

3.2 Computing the Hash Values of Graphs

Whenever the compiler is about to add an object, it looks for any structurally equivalent object. Obviously, the database maintains hash tables

```
// a.h
class A {
  char x; unsigned y;
};
// a.C
extern int x;
#include "a.h"
double foo(float);
// b.C
extern int x;
#include "a.h"
double hoo(float);
```

Figure 3: A Sample Program to Show Object Sharing

for these lookups. Precisely, the population of an object involves the following steps. The compiler first constructs an object in the transient heap. For each of the candidate persistent objects in the hash table, the compiler performs the structural equivalence test. If this succeeds for some candidate, the compiler subsequently uses the persistent object and discards the transient object. Otherwise, it makes a persistent copy of the transient object, inserts it into the hash table, and discards the transient object.

It might be interesting to compute a hash value for a graph. To facilitate this, the compiler maintains an invariant specifying that, when it attempts to store an object, only the root of that object is transient and its siblings or subgraphs have already been made persistent, or stored into a database. The hash value of a graph is computed by using one or more values of fundamental types in the root object and one or more direct siblings' persistent addresses.[2] The computation is very fast, since it does not involve traversal of the graph, but it still effectively takes into account all the objects contained in the entire graph.

[2]The persistent address of an object within a database is a pair of segment identifier and offset in ObjectStore. Since these values are stored in the protected members of a class ObjectStore supplies, we need a hacking method for deriving a class to breach the access control.

3.3 Overlinking

The object files resulting from the compilation of two translation units, shown in Figure 4, are not necessarily linked into the same program. If they are not, the symbol object for `int x` actually represents two different symbolic entities; this is inevitable, since it is not necessarily known in advance which executable a particular object file is eventually linked into, especially if it is a member of an archive. Similarly, even within a single program a single symbol for a class may be used to represent different C++ classic entities of the same name, since they may have internal linkages. In short, *overlinking* occurs.

Overlinking is much easier to cope with than "underlinking." For this purpose, the *coverage* of an object is defined to be a subset of database objects that can be reached from the object by tracing pointers. Though a symbol object does not necessarily uniquely identify a C++ symbolic entity, it really does if paired with an appropriate coverage. For instance, a pair of the symbol denoting `int x` and the coverage of the file object into which the file `a.o` is linked uniquely identifies the symbolic entity declared in `a.c`. This pairing effectively means that a database client limits its interest to the specified coverage when dealing with queries.

4 Clustering

Clustering[3] is a technique of populating together objects that are referenced together, thereby improving reference locality. The client of a poorly clustered database is very likely to cause *thrashing*, which effectively prevents it from running any further. Reference locality, however, totally depends on clients' access patterns. In particular, *burst* access made by clients is of great concern.

Obviously, we cannot list all the clients in ad-

[3]ObjectStore allows two levels of clustering, by what are called "segments" and "object clusters." What we have used for clustering the program database are "segments." However, in this paper we consistently use the term "cluster," which means a segment in ObjectStore terminology.

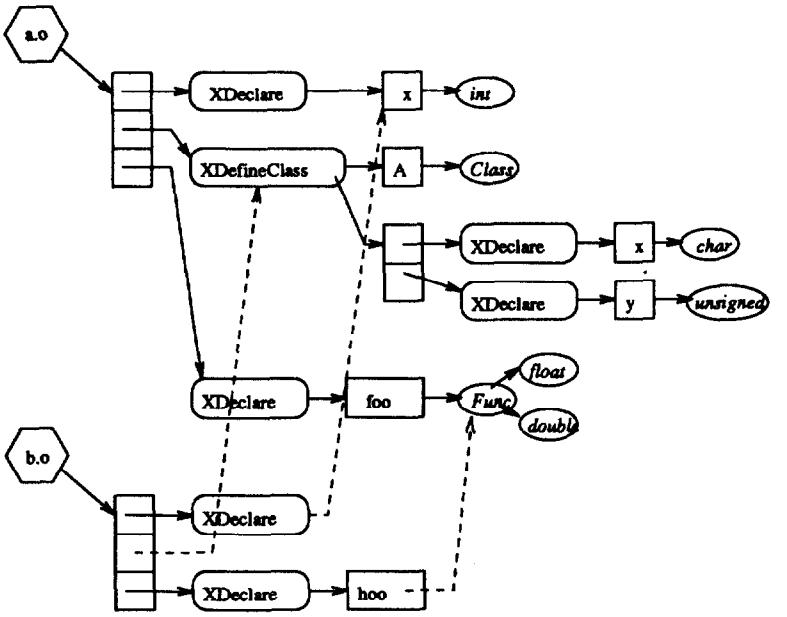

Figure 4: Sharing of Objects between Translation Units

vance, and access patterns of clients may impose conflicting demands on clustering. What we can do at best is to prepare for major, known clients. This section first considers burst access that expected clients are likely to make. It then summarizes three methods of clustering applicable to our program database, and describes the clustering actually used. Finally, it presents further effects of the clustering, one of which contributes to a substantial reduction in the size of a database.

4.1 Burst Access

Expected clients that heavily access a program database are a compiler, a scavenger, a source-code browser, and a debugger. Though a compiler is primarily considered as a populator, it is also a heavy accessor; it does a large number of lookups for sharing, which is likely to cause burst access.

A scavenger is a tool for reclaiming database objects that are no longer referenced; this tool is needed because ObjectStore leaves it up to users to garbage-collect database objects. Since the C++ convention is to call the `delete` operator for each unused object, burst access is likely to result.

A browser does not madly access a database as long as it is processing such requests as "show members of a class" and "show all the breakpointable locations of a function." However, burst access may happen when it attempts to show the entire inheritance graph, the entire call graph, all the cross-references of a symbol, or all the functions and global variables defined in a program.

A debugger is also a modest client as long as it handles such requests as "set a breakpoint here" and "print an automatic variable's value." Burst access may result for a request such as "show the call stack's contents" for a long stack and "show all the values of global variables."

Our observation is that coping with burst access by a compiler and a scavenger should have top priority. The reason is that both lookups and deletes may touch pages in the persistent storage scattered in the *database-wide*, while access by a browser or a debugger is limited to pages that contain objects representing a program.

4.2 Methods for Clustering

Three methods of clustering are considered promising for C++ program databases. First,

birth-order clustering simply stores objects in the order in which they are created. Since a populator can easily follow this order, it is used widely, even unconsciously. Not surprisingly, it can be observed that the birth order is respected by many access patterns; objects are never created in a random order, and the order reflects the logical structure of a program.

Death-order clustering stores together objects that die more or less at the same time. This method of clustering is very beneficial for a scavenger or, more generally, in terms of memory management, although we cannot always know the death times of objects. Furthermore, if a database management system allows us to delete a cluster, we can delete a bunch of database objects at the cost of a single function call, as is often done in implementing a customized memory management system for C and C++.

Finally, *sharing-oriented* clustering stores together objects that are shared among translation units, in order to prepare for lookup operations. As mentioned earlier, what are shared in a program database are global objects. If we simply follow the birth-order clustering, lookups may cause thrashing, since the method interleaves global objects with objects from inner blocks. In short, where there is sharing, there must be clustering.

4.3 Our Clustering

Objects are clustered in a C++ program database, as shown in Figure 5. First, we apply sharing-oriented clustering, following the prioritization mentioned earlier. Global symbols are clustered together with the hash table, as are global types.

As can be seen in the figure, objects that originate in the same file form a cluster, or what we call a *file cluster*. This *file-based* clustering is a combination of sharing-oriented and death-order clusterings. It is sharing-oriented because it clusters globals xrefs, which come from the same file, together with the hash table. It is not acceptable to put all the global xrefs into a single cluster, since the cluster would grow gigantic; dividing them among files prevents this. On the other hand, file-based clustering is also a death-order clustering, since objects from the same file tend to become unused simultaneously; this demographic fact reflects the current way of developing a C++ program, namely, on a file basis.

To add an object to a file cluster, the compiler must first look for the cluster corresponding to the operating system's file in which the object originates. It does so by using not only the file's name but also the file's timestamp. In this way, a file cluster is created for each version of an operating system's file.

Finally, birth-order clustering is applied within each cluster, and file objects form their own cluster.

4.4 Further Effects

File-based clustering also helps to reduce the size of a database. It allows us to squeeze the location of an xref object into a pair, instead of a triple of the source file, line number, and column number, by *cluster-tagging*, which is very similar to the page-tagging used in many Lisp interpreters. Given a virtual address of an xref, ObjectStore allows us to obtain the persistent address. In other words, we can find out which cluster (or segment in ObjectStore terminology) the xref is allocated to. If we build a table that maps clusters to files, we can eventually obtain the file in which the xref originates. The source file of an xref's location thus becomes a computable attribute. Notice that xref objects are by far the most dominant objects in a program database; squeezing a word from an xref leads to a substantial reduction in the database size.

Finally, we touch on the garbage collection of database objects. Owing to death-order clustering, individual objects do not have to be deleted; we simply delete a cluster. File clusters themselves are maintained by applying the reference-counting technique; we believe that reference counting on the granularity of clusters is quite acceptable. The clusters for shared symbols and types are never reclaimed by using reference counting. However, shared objects within the clusters are garbage-collected by using the

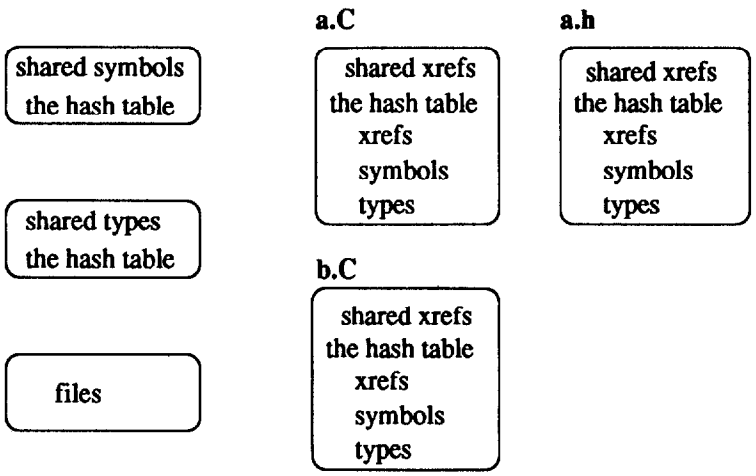

Figure 5: Clustering in the C++ Program Database

mark-and-sweep technique, which is much less frequently invoked. Note that the abovementioned arrangement of a file cluster for a version allows the reference count to be zero soon enough; if a file cluster contained objects from all the versions of an operating system's file, the reference count could not reach zero. However, it would be rather expensive to create a cluster each time a source file is modified and compiled; a good trade-off is to let a file cluster hold program information across a few versions of a source file.

5 Performance Measurements

We have implemented three important components, and built two clients with them. The first component, called pd, defines the scheme of the C++ program database, and must be linked into every client. It is also responsible for sharing and clustering, though such functions are only used by a populator. Another component, dop, defines the population methods mentioned in Section 2. We have built a populating version of the C++ compiler with dop and pd. Lastly, the pq component implements basic queries on top of pd; among them are collective queries such as "show all the classes" and navigational queries such as "find the definition of a function." We have built a simple source code browser with pq and pd. The pd, dop, and pq components are all written in C++, using ObjectStore Release 2.0, and contain 9053, 8575, and 1808 lines, respectively.

We have run two clients against three programs: these are a Lisp interpreter; a simple X application, which bounces balls in a window; and a GUI library, which is used for building GUI on Motif. The application mentioned in Section 1 was built by using the library. Table 1 summarizes the characteristics of these programs. We have performed measurements under AIX 3.2.4 on a RISC System/6000 Model 560 with 384 megabytes of main memory. We set the size of the client cache at 8 megabytes, which is one of the parameters for the client environment in ObjectStore.

5.1 Database Sizes

We first compiled these programs using the populating compiler, and built a database for each program; it is worth emphasizing that we were successful in populating a database even for the GUI library. The last column of Table 1 shows the sizes of the resulting databases. The levels of disk consumption are very much lower than in a conventional approach. Taking the GUI library as an example, the additional disk space needed to do browsing and debugging is 68.47 megabytes in the conventional system in Section 1, while the

Program	Number of Translation Units	Number of Lines				Size of Database (kilobytes)
		Before Preprocessing		After Preprocessing		
		Total	Average	Total	Average	
Interpreter	3	673	224.3	4,695	1,565	761
Bouncing Ball	46	4,018	87.3	361,924	7,868	5,881
GUI Library	294	15,071	51.3	2,686,647	9,138	32,498

Table 1: Characteristics of Measured Programs and Database Sizes

size of the database covering such information is much smaller, as shown in the table.

Though we constructed a database for each program in this measurement, our database can accommodate information on multiple programs, as mentioned earlier. This turns out to be a crucial advantage when, for instance, we are using the library to build multiple applications simultaneously; otherwise, we would have to create as many databases containing at least about 32.5 megabytes of objects as the number of the applications.

5.2 Browsing Times

Next, we invoked the simple browser against the above databases, and issued typical collective queries. The invocations were instantaneous, and it did not take long to retrieve necessary information from the databases. Table 2 lists the result in detail. It shows the amounts of time taken to retrieve objects necessary for building inheritance graphs and call graphs; the amounts of time measured do not include those spent on actually drawing graphs. We have measured them both in "cold" and "warm" ways. Since the times are proportional to the numbers of objects to be retrieved, the table also shows the numbers of nodes and arcs in the graphs.

In the measurements, we prepared the databases for these queries by building appropriate tables; this is a common tactic in database applications. Had we not done so, we would have had to traverse the entire database. In general, the time needed to traverse a database gives a good indication of the performance in processing a query for which the database is not prepared. The table therefore includes the amounts of time needed for hot and cold traversals, along with the numbers of objects to be visited.

Notice that the database size for the GUI library exceeds the cache size we set in the measurements; we suspect that this caused the times to increase irregularly for the GUI library.

5.3 Discussion

Though we found that our approach is promising in terms of disk consumption and client invocation, a severe problem, is the large amount of extra compilation time, that is, the time needed to construct a database. The populating compiler currently takes 4 to 5 times longer on average than the original compiler does when invoked with the debugger and browser options turned on. This is not due to database operations such as transactions and persistent allocations; the underlying database *per se* performs well enough as long as we create an appropriate clustering. At any rate, we are now working to resolve this problem.

In addition, we are not necessarily satisfied with the current sizes of program databases. Actually, more space can be squeezed. In many OODBMSs, including ObjectStore, objects are stored in persistent storage in almost the same formats as in virtual memory; they are still objects in persistent storage. However, objects are not a compact way of representing information. For instance, the x-coff format encodes the pointer type to the integer type as `*-1`, taking a mere three bytes, while the corresponding type graph in our scheme consists of two type objects, taking a total of 16 bytes.

Database	Inheritance Graph			Call Graph			Entire Traversal		
	Time (sec)		Size	Time (sec)		Size	Time (sec)		#Objects Visited (K)
	Cold	Warm	#Nodes + #Arcs	Cold	Warm	#Nodes + #Arcs	Cold	Warm	
Interpreter	0.0983	0.00773	105+41	0.100	0.0339	501+451	0.0310	0.0107	10.1
Bouncing Ball	1.39	0.0211	293+38	0.820	0.146	2,001+1,193	4.10	0.483	463
GUI Library	4.68	0.0613	517+165	13.5	0.749	7,600+6,988	51.4	37.8	3,492

Table 2: Performance of Browser's Typical Queries

Even the **compress** command in AIX attains 49% to 63% reductions for the databases in Table 1. More reduction can definitely be expected if we extensively use type information, which is also stored in the databases. Obviously, we should perform compression page-wise, thereby allowing on-demand compression. We are now designing the details of this procedure, which we call *type-driven compression.*

6 Related Work

It is well known that the C++ programming environment formerly named Cadillac [4] stores C++ program information in an object-oriented database. It too uses ObjectStore. However, as far as we know, no paper describes the database organization in enough detail for us to compare it with our work.

CIA++ [5] is a tool that stores a C++ program information in a database. The information covered is largely the same as in our scheme, but does not include code-static information. In addition, the database is relational. We are curious to learn how sharing and clustering are dealt with in relational setting.

Reprise [6] is a graph representation of C++ programs. It represents source-static information in much more detail than our scheme, but does not give any code-static information. Though the paper describes how such representations are generated, it does not say how or where they are stored.

Kendall and Allin [7] discuss how the size of a program database can be reduced. They propose eliding unused declarations and combining objects among translation units, which can be performed in two ways, by sharing and by linking. They also present detailed measurements of the degree of effectiveness of these methods in reducing the database size. One difference from our work is that their database is main-memory and contains only a single program. Databases accommodating multiple programs are very different and require the notion of overlinking. Another difference is that they attempt to reduce the database size by varying the requirements imposed on the program database; that is why elision becomes applicable, although the resulting database is not suitable for source code browsing. Our requirement is that the program database must be used for the browser, debugger, and builder.

7 Concluding Remarks

We have described how objects are shared and clustered in an object-oriented C++ program database. Sharing is important to prevent a program database from occupying too much secondary storage. Our initial experiment showed that the database occupies much less space than is required in a conventional file-based environment.

Clustering is crucial to keep clients running without thrashing. Though we have prepared for major expected clients, we have already found a client that has a conflicting demand on clustering. An *instance transformer*, which supports schema evolution, turns instances of updated classes into new versions, accessing all the instances of each

class. The best clustering for this is a *class-based* clustering, which allocates instances of a class together. However, it obviously conflicts with the sharing-oriented and death-order clustering. Dynamic reorganization of a database might be pursued in this case.

As we have seen, populating objects involves linking symbolic entities. We expect to be able to completely eliminate the phase of batch linking in the not-so-distant future; this will considerably accelerate the development cycle. In addition, it is believed that C++ program databases can automate a complicated process of template instantiations [8]. We believe that the evolution from separate compilation to populating compilation will have a significant impact on C++ program development.

Acknowledgments

We thank the members of the Compiler Tools Group at IBM Canada's Language Technology Centre and of the Programming Languages Group at IBM Japan's Tokyo Research Laboratory for their detailed discussions, which very much improved this work. Ed Merks and Kazu Yasuda defined the population interface, and Shoichi Ninomiya implemented the dop component.

Trademarks

ObjectStore is a trademark of Object Design, Inc. AIX, RISC System/6000, and C Set ++ are trademarks of IBM.

References

[1] Thomas, I. and B. A. Nejmeh. Definitions of Tool Integration for Environments. *IEEE Software* 9(2), 29-35 (1992).

[2] Lamb, C. et al. The ObjectStore Database System. *Communications of the ACM* 34(10), 50-63 (1991).

[3] ObjectStore User Guide, Object Design, Inc., 1992.

[4] Gabriel, R. P. et al. Foundation for a C++ Programming Environment. *Proceedings of C++ At Work* 1990, pp. 85-102.

[5] Grass, J. E. and Y. Chen. The C++ Information Abstractor. *Proceedings of the 1990 USENIX C++ Conference* 1990, pp. 265-277.

[6] Rosenblum, D. S. and A. L. Wolf. Representing Semantically Analyzed C++ Code with Reprise. *Proceedings of the 1991 USENIX C++ Conference* 1991, pp. 119-134.

[7] Kendall, S. C. and G. Allin. Sharing Between Translation Units in C++ Program Databases. *Proceedings of the 1994 USENIX C++ Conference* 1994, pp. 247-263.

[8] McCluskey, G. Template Instantiation For C++. *ACM SIGPLAN Notices* 37(12), 47-56 (1992).