Software Configuration Management in an Object Oriented Database

Mick Jordan and Michael L. Van De Vanter
Sun Microsystems Laboratories
Mountain View, CA 94043 USA

Software Configuration Management in an Object Oriented Database

Mick Jordan
mjj@Eng.Sun.COM

Michael L. Van De Vanter
mlvdv@Eng.Sun.COM

Sun Microsystems Laboratories
2550 Garcia Avenue
Mountain View, CA 94043 USA

Abstract

The task of configuration management for software development environments is not well supported by conventional files, directories, and ad hoc persistence mechanisms. Typed, immutable objects combined with ubiquitous versioning provide a much more sound basis. A prototype configuration management system currently under development relies upon persistent objects provided by a commercial object-oriented database system. Mechanisms and policies developed for this prototype simplify programming with persistent objects; results include simplicity of both design and implementation, as well as flexibility, and extensibility. Initial measurements suggest that performance is acceptable.

1 Introduction

A practical software system consists of hundreds or thousands of separate but related components. A software development environment (SDE) must help manage this collection by providing *configuration management* services. Closely related is *version management*, often associated with tracking the evolution of individual components. However, configurations naturally exist as versions, and it is the effective management of such configurations that distinguishes a truly useful SDE for large, multi-user projects.

An SDE typically relies on the persistent storage facilities provided by the underlying operating system, usually in the form of a file system with two basic mechanisms: files and directories. Unfortunately the weak functionality of these primitives does not permit adequate modelling of configurations. This has led to the suggestion that an SDE rely instead on a database management system (DBMS) [1].

Experience with SDEs built using traditional DBMS technology has not been encouraging. Indeed such systems have acquired a reputation for being resource hungry, and they have not found their way into widespread use. However, in recent years there has been a surge of interest and development in object-oriented database (OODB) technology. While the relational database community might argue that OODBs are fated to repeat the perceived failure of the network databases of the 1970's, there is no doubt that the better impedance match with everyday programming has a great appeal to many developers. Indeed one can argue that OODBs are becoming successful because they are delivering, in part, the promise of persistent programming languages [2].

This paper describes our experience building a prototype configuration management system using ObjectStore® [3], a commercial database system produced by Object Design Inc.®

We begin by reviewing the advantages of an OODB as infrastructure for an SDE and then provide an overview of the essential facilities provided by ObjectStore. Next we briefly describe the structure and operation of our configuration management system and explain how objects help simplify both design and implementation. We then turn to the mechanisms and policies that we have developed to exploit ObjectStore successfully in our prototype, many of which have wide applicability. After presenting some preliminary performance measurements we describe related work and discuss our conclusions and future plans.

2 Why a Database?

Traditionally an SDE is built on top of a file system. The inertia caused by existing file-based tools makes it quite difficult to change this situation, and any database solution must provide a way to integrate legacy tools. Meanwhile progress in tools continues, and the limits of file-based systems are being pushed. In particular, it is increasingly common to find ad hoc "databases" being added to file based SDEs. Examples include data to support browsing, C++ [4] template instantiation, and build state for *make* [5]. Managing these ad hoc persistent stores, particularly in the face of concurrent access, is difficult and error prone. Yet, there is no shortage of object-based systems offering solutions, for example CORBA [6] and Microsoft OLE 2 [7]. However, although these systems provide varying degrees of support for persistent data, they currently do not scale to the task of supporting an SDE efficiently. In contrast, OODBs have matured to the point that we believe it is practical to deploy them as an alternative to file systems and ad hoc persistence mechanisms.

An OODB offers two main advantages over a file system as infrastructure for an SDE:

- A transactional store, and
- Precise modelling of application data.

2.1 Transactional Store

All updates to a database take place inside a transaction; they either complete in their entirety or not at all. Therefore the evolution of data in a database proceeds through a sequence of well-defined states. Furthermore, data modified in a transaction by one process is not visible to other processes until the transaction commits. Providing such a guarantee for a file-based system is substantially harder.[1] While the transactional property might be seen as a minor feature in a single user programming environment, it is extremely important in large, multi-user projects. Even in single user environments, the use of multiple processes can easily put data into an inconsistent state, for example should a process abort at an inopportune moment.

2.2 Precise Data Modelling

We believe that the capability to design objects that closely model the application domain is the more significant benefit of an OODB. File based systems are hampered by the performance and modelling discontinuities that occur at the file and directory boundaries. An OODB can provide a more consistent solution. In particular, there are substantial advantages to be gained from utilizing *fine-grain* objects: objects too small to be stored efficiently as individual files. That all objects are also *typed*, significantly improves the readability and reliability of programs that manipulate them. Traditional solutions to storing fine-grain objects in a file system involve *pickling* schemes [8] [9], whereby programming language objects are preserved in some way inside one or more files. Although this can work well for simple structures, it rapidly becomes unwieldy for the complex, linked structures that prevail in compilers and associated tools [10].

1. A transactional file system would provide such a guarantee.

3 ObjectStore

ObjectStore is a commercially available OODB that runs under UNIX® and Microsoft® Windows. It is primarily targeted to the C++ language, but support for Smalltalk [11] has been added recently. Our experience is with the C++ system for the Solaris® variant of UNIX®.

The impedance match between ObjectStore and C++ is generally very good. Objects are allocated in a database by overloaded variants of the C++ `operator new`. In consequence all objects in the database can be referenced by C++ pointers. ObjectStore transparently maps portions of the database into the address space of the C++ process, using virtual memory mapping primitives of the underlying operating system. Access to objects in the database must occur inside a transaction, and any pointers that are mapped within that transaction are not valid after it terminates. Multiple processes may access the same database, and ObjectStore enforces a multiple reader-single writer locking policy.

Within these constraints, and inside a transaction, manipulation of database objects is no different from manipulating virtual memory (heap) objects, and it is this feature that makes use of the database transparent. However, to avoid *lock conflicts* with other processes it is necessary to limit the time spent in each transaction. A lock conflict occurs when one process holds a write lock and another process requests a read or write lock, or vice versa; this delays the requesting process until the lock is relinquished. Since locks are acquired only as data is accessed, *deadlock* can occur when access patterns produce cyclic lock dependencies. In case of deadlock one transaction is automatically aborted and subsequently retried up to a user-defined maximum number of times.

In order to retain access to an object between transactions, it is necessary to use an ObjectStore *reference*: a kind of heavyweight pointer. Unless some care is used to minimize the use of references, much of the transparency is lost. We will discuss this issue in more detail in section 5.4.

ObjectStore is based on a client-server model. Databases are managed by a server process on a designated server machine. Client processes, typically on separate machines, communicate with the server to read and write data. Client-side caching is heavily used to minimize communication overhead.

A process can access multiple ObjectStore databases within a transaction, and database objects can contain references to objects in other databases. An update transaction that modifies data in multiple databases managed by different servers is implemented by a two-

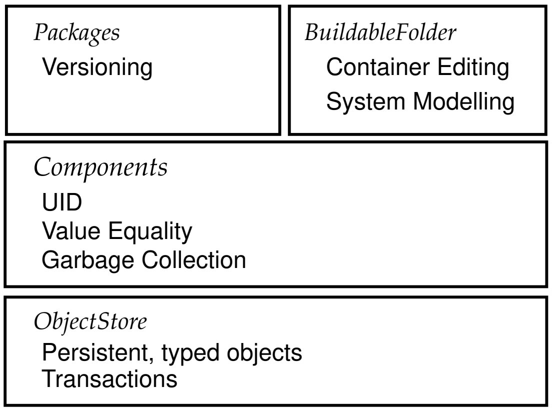

Figure 1. Architectural Layers in Forest

phase commit protocol.

We currently do not use ObjectStore's other features, such as querying, collections library, and generic version management. Our primary interest is in exploiting the shared, distributed, and transacted virtual memory model provided by the basic infrastructure.

4 Configuration Management in Forest

Our configuration management system is part of a larger project, Forest, that aims to provide an integrated, multi-user, software development environment for medium to large scale systems programming. A major goal of Forest is to discover whether current OODB technology is up to the task.

The goal of the configuration management system is to support the precise specification of a software system that may occur in many versions and variants. It must be capable of dealing with a spectrum of object types, from very small objects that might represent a fragment of a source program through to structured objects that might represent a complete system build containing thousands of objects. Intermediate in this spectrum are objects that represent object modules, programs, etc. In file-based systems the structure of these kinds of objects is typically defined by a *file format.* In object-based systems we expect such objects to be defined increasingly as abstract object types.

Figure 1 suggests how services are provided by different architectural layers of the prototype.

4.1 Components

Since the term object is rather overloaded, we use the term *component* to denote the objects that are manipulated by the Forest configuration management system. Although ObjectStore can store instances of any C++

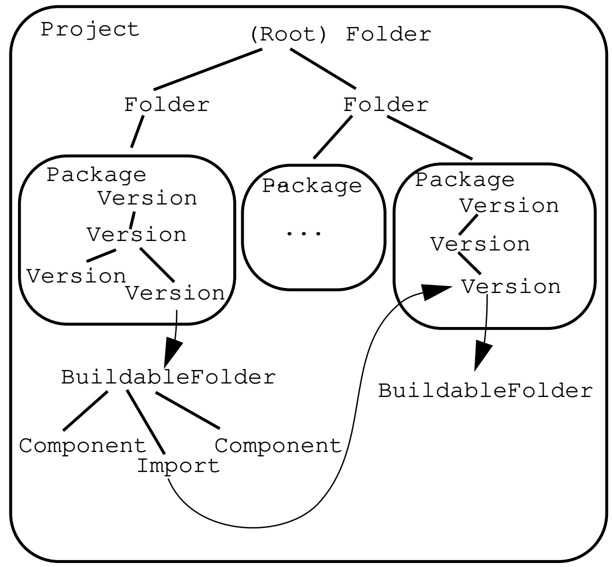

Figure 2. Abstract Example of a Forest Project

type, we restrict components to be a subclass of an abstract C++ class `Component`, which defines a common set of operations and attributes.

It is possible to construct arbitrarily complex collections of components, connected in an OODB, much more easily than one can with a file system. In practice, we enforce some familiar high level structure, partly to meet the goals of the configuration management system and partly for administrative and engineering reasons.

4.2 Projects

Components are allocated in and belong for their lifetime to a *project*. A project physically groups related components and, as the name implies, is intended to be shared by a group of collaborating programmers. A project is similar to a disk volume or mounted file system and is implemented as a single ObjectStore database.[1] Components in one project may refer to those in another, but only through special components called *links*.

At the outer level a project looks very like a conventional file system. A root *folder* (analogous to a file system directory) can contain other folders or arbitrary components. Two particular component types, *Package* and *BuildableFolder*, are the essence of the configuration management system; we will now describe these in more detail. Figure 2 suggests how these are related in practice.

1. We use the terms "project" and "database" interchangeably.

4.3 Packages and BuildableFolders

Forest follows the Vesta [12] approach to configuration management. A project contains a collection of heterogeneous *source* components, organized into families called *packages*, each of which is arranged into a tree of versions. Source components are so-called because they cannot be generated automatically by the system. Users typically create source components using tools such as editors. However, a binary program that is imported into the database from elsewhere is also a source component in this context. All other components, i.e. those that can be generated automatically by the system, are *derived*. Components, once created, are immutable. Tools used to construct derived components are themselves modelled as components and versioned in the same way. Systems are built by executing a functional program in a *system modelling* language. These properties provide the basis for reliable and repeatable system builds.

Unlike Vesta, in which the package structure is hardwired, a Forest package is just a particular subtype of the `Component` class. The operations available on a package are specified as ordinary C++ methods, and each executes as a transaction. A package version can bind *any* subtype of `Component`, although to date we have not exploited this flexibility. Variations of the package type could be defined (for example to implement different organizations of versions), as indeed could quite different structures, and all could coexist in the same database. This flexibility and extensibility is one of the clear benefits of the OODB approach.

Following Vesta, the component type that we currently bind to a package version is a special kind of container that we call a `BuildableFolder`. This is similar to a file system directory, in that it contains a set of heterogeneous (source) components. However, it also plays a key role in the system building process in that it is also *input* to the system builder. In this role a `BuildableFolder` acts as a module in the system modelling language. The source components act as constant values that are the inputs to the algorithm that computes (builds) the system. Large systems are built up by connecting package versions together using an *import* mechanism. The essential feature of the import mechanism is that it is not intensional; each `BuildableFolder` precisely and permanently identifies the other folders that it imports. The details of the system modelling process are beyond the scope of this paper; the mechanisms are similar to those described for Vesta, but the details are rather different.

Development typically proceeds by the user *checking out* the latest version of a package. This produces a new branch on the version tree and a new `BuildableFolder`, which initially shares the components contained in the folder that was the subject of the checkout operation. The package components, which can be of arbitrary type, are then evolved by appropriate editors. We use the term *evolved* rather than *modified* to emphasize that any editing must take place on copies. We expect editors to exploit the immutability provided by the system and share as much as possible of original components when constructing copies.[1]

Coordination of component editors is overseen by a special editor for a `BuildableFolder;` this editor ensures, amongst other things, that the entire folder is saved as a group, thus versioning the entire configuration. Once a version is saved, which involves committing any new components to the database, it can be built. Therefore, it is important that the group save operation be fast, in user time, and our experience with the database is positive in this regard. Eventually a version of this checkout branch, usually the last, is *checked in* by creating a new version on the main branch. This involves no copying or building, since it merely serves to associate a new version name with the same folder.

4.4 Fine-grain Components

Some components in a `BuildableFolder` have internal structure that is made visible through the interface to the component. Since this structure corresponds to the sub-file granularity in a file-based system, we refer to such components as *fine-grain*. Most current program editors are text-based, so typical source components are not structured, but we expect this to change as object-based systems and compound documents become more widespread. In contrast, it is somewhat easier to represent derived components, those created by tools during a system build, as structured objects by way of wrappers and similar techniques. In Forest, these are defined as appropriate subtypes of the `Component` class, tailored to model the abstract objects they represent.

For example, we specify the input to the parser for a programming language as a `TokenSequence` component that represents a sequence of tokens in that language. This component might be produced by a lexer from a text component or it might be manipulated directly by a language sensitive editor. Either way, the parser, as well as all downstream tools in the compilation pipeline, are only interested in the elements of the token sequence, and are unconcerned that it might be derived from or be part of another component. This pre-

1. This assumes that components are structured so that unmodified substructure is shareable.

cise modelling of the compilation pipeline provides opportunities to avoid certain phases. For example, a change to a comment or other annotation, such as a graphic, in the original source component, would only require re-lexing.

4.5 Uids and Abstract Values

Two particularly important attributes of a component are its *uid* and its *value*. Each component is assigned a globally unique identifier (uid) that denotes it for all time.[1] The uid acts much like a pointer or reference in a programming language. Forest uids are 128 bits wide, part of which encodes the database in which the component is allocated. The uid provides a guaranteed way to identify a component, in contrast to the facilities offered by the UNIX file system. The uid also provides a sure way to determine if two references are to the same object.

However, uid comparison sometimes draws too fine a distinction. Two components with different uids may in fact represent the same abstract value, for example the same sequence of tokens in the `TokenSequence` component of the previous section. The abstract value attribute is represented by a *fingerprint* [13]. A fingerprint is an opaque bit sequence, similar to a hash code, that encodes value probabilistically. Two components with the same value have the same fingerprint. Conversely, if two components have different values, then, with extremely high probability, they have different fingerprints.[2] Each component implementation can provide its own definition of the value attribute; the default, inherited, implementation uses the uid, which is safe but conservative. Fingerprints can be combined efficiently, which is how the abstract value of structured components is computed. Component implementations are free to cache the fingerprint or compute it on demand. The system builder deals with components through their abstract values because, in the system modelling language, object equality is defined as equality of abstract values.

4.6 Copy or Share?

The combination of immutability and abstract value provides many benefits, not least the freedom to copy or share components at will. For example, a system that is built in a database at a developer site can be copied to a database at a client site. Provided that client and developer share the same environment with respect to the dependencies of the copied system, no rebuilding is necessary because, even though the copied components will have different uids, the abstract values of the components will not have changed. If, on the other hand, the client uses a different version of the compiler, then the affected components will be rebuilt transparently; the system takes full responsibility for the construction and location of derived components.

The ability to share components reliably has two major benefits. First, it permits components to be constructed as incremental modifications to existing components. This is similar to the way in which versioning systems such as RCS [14] use “diffs” to encode changes to text files, but it differs by being proactive rather than retroactive. In addition, arbitrary component types can be constructed incrementally in the Forest environment, leaving details to component implementations. The second benefit is the ability to share large derived components such as programs, libraries, or entire subsystems among many versions and many users. This can dramatically reduce overall storage requirements, making the system competitive or better than current file-based environments.

5 Storage Management and Locking

We turn now to the implementation of the configuration management system, and in particular to the impact ObjectStore has on the Forest/C++ programmer. This is felt in two main areas: storage management and data locking. We will describe the mechanisms and policies that we have developed to minimize this impact.

5.1 Allocation Mechanisms

As noted earlier, objects are allocated in an ObjectStore database using overloaded variants of the C++ `operator new`. The C++ language permits optional *placement* arguments to `new;` ObjectStore uses these to control where in the database an object is allocated, using two basic placement abstractions: *segments* and *clusters*.

Segments support large scale structuring of the database and provide hooks for various policies to improve performance. For example, whole segments can be transferred from the server in one request. Segments can be of unlimited size.

A segment can be divided into clusters, which are essentially abstract pages. A cluster has a minimum and a maximum size, the minimum being one page. ObjectStore’s locking is actually page-based, so objects in dif-

1. In principle the uid could change, provided the change appeared atomic to all clients, but the difficulty in locating all references makes this impractical.

2. The probability can be increased by adding more bits to the fingerprint representation; we use 64.

ferent clusters cannot cause lock conflicts. Clusters also provide for locality of reference for groups of small objects. Since clusters have a maximum size, a client must always be prepared for an allocation request to fail, requiring the cluster to be extended or a new one to be allocated. Objects larger than the maximum cluster size cannot be allocated in a cluster at all.

It would be unworkable for each allocation site to deal with all these issues, so in Forest we abstract placement data into a C++ class called `DBNewPlace`. Although this successfully encapsulates the ObjectStore placement information, a client must still acquire or generate an instance of this class in order to allocate any objects. Fortunately ObjectStore provides enquiry methods that return the cluster or segment in which an existing object is allocated. Provided such an object is at hand, it is easy to generate a `DBNewPlace` that will cause allocation with the same placement. In practice, almost all objects are associated with some container object, and allocation with the same placement as the container is usually the right thing to do. Since such allocations typically take place inside a method of the container object, the C++ `this` pointer can be used to generate the placement. In cases where a `this` pointer is not available, such as static member functions, a `DBNewPlace` instance must be passed as an explicit argument.

5.2 Allocation Policy

The `DBNewPlace` class provides the mechanism for abstracting the placement information, but who provides the policy? Our experience bears out Object Design's advice: unless this is carefully planned, lock conflicts, poor locality of reference, and consequent inadequate performance are inevitable.

The versioned, mostly immutable nature of a Forest database is of considerable help in establishing appropriate policy. By design, conflict can only occur on the mutable components that represent the leading edge of development. Recall that development by an individual user takes place on a package. Often two or more packages may be involved, for example a package that provides a library of code and an application package which uses the library. Different individuals will typically be working on different packages, ultimately integrating their work by way of packages that denote subsystems.[1]

Each Forest package is allocated in a separate segment, so all source components in all versions of the package are allocated together. Lock conflicts thus cannot arise from modifications to different packages. In addition, when a particular version of a package is built, all new derived components are allocated in a fresh segment. So concurrent builds of different versions of the same package also cannot conflict. Should two programmers be working on different branches (versions) of the same package, conflict can occur while creating new source components. However, this happens in user-time, and the lock is only held for the time it takes to write the new component. If such conflicts proved to be problematic, it would be relatively straightforward to allocate extra segments, for example, for each version branch. Only the code that creates new package versions would be affected by this change; editors and other mutators would simply inherit the appropriate placement.

Owing to the size constraints, we find clusters much less useful than segments; we use clusters only for components whose size we can predict with confidence.

5.3 Uid Generation

The most significant hot spot in the database is the data structure that deals with uids. At a minimum the *next-uid* value must be updated each time a component is created. The data structures that provide a mapping from a uid to the associated component are also potential hot spots. Evidently this data must be clustered separately, otherwise read-only navigation of the database will immediately conflict with mutating.

In fact, as the measurements in section 7 show, the conflicts that arise when multiple processes are mutating the database quickly cause serious performance problems. To address this problem we have made it possible for uid generation to be distributed amongst components in the database. Rather than hard-wire a particular distribution, any component type can opt to be a uid-generator by inheriting and implementing the `UidGenerator` interface. This interface defines a protocol that permits a hierarchy of components to collectively manage the uid space in one database.

Initially the only uid generator is at the root component, which also serves as the entry point to a database.[2] A component can register with the root and acquire a portion of the uid space that it then manages. Any component can be asked for its uid-generator. Since allocation and uid generation go together, we extend the `DBNewPlace` class to carry both the database placement and uid-generator instance. There are no constraints on how a component implements the `UidGenerator` interface; the protocol provides for the enumeration of the components under its control. In principle this hierarchy could be of any depth. In practice, we have only used a

1. Vesta coined the term *umbrella package* for this purpose.

2. It acts like "`/`" in the UNIX file system.

depth of two, by making each package be a uid generator that registers with the root component.

The contention caused by the uid structures could be avoided by binding our implementation more tightly to ObjectStore, since each persistent C++ object has a unique persistent address.[1] However, an initial goal of the project was to avoid too much dependence on implementation details of any one database, hence our own uid layer. Since the uid representation is hidden from Forest clients, we should be able to perform this optimization in the future. Nevertheless, the uid structure is just a particular example of an *index*, which occur repeatedly in database applications. A consequence of the distributed shared memory model is that such indices are certain hot spots if the update frequency is high.

5.4 Locking

Careful clustering and distributed uid generation does much to alleviate the potential for lock conflicts in Forest, but it does not permit the programmer to ignore the issue completely. In particular, any process that involves user-interaction or indeterminate delay cannot be undertaken inside a transaction without the risk of locking out another process. In Forest, this issue manifests itself most obviously in the user interface to the configuration management system. Much user activity involves browsing the package structure, using read-only transactions, with occasional bursts of editing and building that require update transactions.

In order to avoid holding a read lock that might prevent an update, user-interface code must use short transactions for database navigation. Pointers that represent navigational context must be converted to ObjectStore references, which were alluded to in section 3. ObjectStore provides a variety of reference types, with varying costs and functionality. All are represented in C++ as template types and behave like *smart pointers* to the real object type. For the most part this fiction succeeds, but the C++ type system sometimes falls short. For example, if `B` is a derived class of `A`, then a `B*` is assignable to an `A*`. However, there is no such relation between a `DBRef<B>` and a `DBRef<A>`, where `DBRef<T>` is a reference to a type `T`.

One particular trap for the unwary involves closing a transaction within a method of an object whose `this` pointer becomes invalid when the transaction ends. This is analogous to sawing off the branch on which one is sitting. The solution is to use functions or transient objects to scope transactions.

1. The public interface provides this as a text string that encodes the database, segment and offset.

5.5 How much to Store Persistently?

One reason that database-based environments have a reputation for storage excess is that the data structures associated with compilers and related tools can be very large. A good example is a program represented as an Abstract Syntax Tree (AST). Another is the debugging information in typical UNIX object files.[2]

Forest solves this problem with two distinct mechanisms:

- Separation of interface and implementation; and
- Aggressive sharing of substructure.

The second mechanism, discussed earlier, deserves further emphasis. The intrinsic immutability of Forest components enables common structure to be shared, confident in the knowledge that it cannot change.

The value of physically separating interface and implementation is well understood in the abstract data type programming communities. In the C++ community physical separation by way of abstract base classes is uncommon, perhaps because of efficiency concerns and poor language and compiler support.[3] In a database context, efficiency concerns are clearly secondary to the flexibility that abstract interfaces provide. Forest components are defined and accessed through interfaces defined as C++ abstract base classes. These abstract interfaces show no implementation detail: all methods are pure virtual, and no data members are permitted. The implementor enjoys complete freedom to represent the abstract state of the component in the most convenient way. The actual implementation takes the form of a class that is derived from the abstract class and contains the physical data members.

When coupled with immutability, this separation of interface and implementation becomes a powerful tool to control database size. Consider the AST example mentioned previously. An AST, when annotated with semantic information, is a large data structure that is not obviously worth storing persistently in its entirety. Fortunately, we can easily establish the minimum amount that must be stored: the source component from which the AST was generated, any ASTs that correspond to imported (included) source components, the tool responsible for the AST generation, and any environmental information, such as the target machine,[4] upon which the generation depended. Since, by definition,

2. This is particularly true for C++ code because language semantics makes it difficult to share information from include files.
3. C++ does not treat abstract base classes specially, and some implementations actually penalize their use.
4. Represented, of course, as a component.

these dependent components are immutable, references to these components are all that need be stored persistently. The full AST can be generated in transient memory on demand, and cached in a table keyed by its abstract value (fingerprint).

At the other extreme we might choose to store the entire AST persistently. Abstract base classes, combined with ObjectStore's pointer transparency, leave clients unaware of this transition, other than possible delays when the AST must be regenerated in transient memory. Evidently we have reduced this problem to a classic space-time trade-off. Indeed, it is possible to create both kinds of components in the same database and therefore measure the trade-off for a given processor/network/disk combination, all other factors being held equal. Given the continuing divergence in processor and disk speeds, generating infrequently used data on demand in transient memory seems a sound strategy. Note that in either case the same amount of virtual memory is required for the AST; either it is mapped from the database or allocated transiently by the process.

5.6 Garbage Collection

The high degree of sharing and the concurrent nature of data access makes manual storage management impractical in a Forest database. Fortunately the configuration management framework makes automatic garbage collection a practical possibility. First we must explain the different ways in which garbage can occur in a database. Although most components are immutable, there are exceptions.

- Derived components that result from building a package version can be discarded at will since, by definition, they can be recreated reliably from source components when needed.
- Building typically creates garbage in the form of intermediaries, for example components analogous to compiler intermediate files. These become garbage because they are not reachable from any package.
- Users can delete old package versions, causing some source components, modulo sharing, to become garbage. Since this is a potentially dangerous operation, and since derived components dominate storage costs, it is not clear that source deletion is really necessary.

Garbage collection is currently supported by reference counts on components. Reference counting of derived components is mostly handled automatically by the system builder. However, editors and other tools that manipulate structured components, such as the versioning system, are required to handle reference counting manually. The reference counting interface is part of the `Component` class, but the implementation is private and could be altered without affecting clients.

Since components may reference components in other databases, we have, in general, a distributed garbage collection problem. Each database occupies a distinct piece of the 128-bit wide address space, so there is no conceptual difficulty in garbage collecting a large enough portion of this address space to bound all inter-database references. Note that manipulation of reference counts and garbage collection are the only ways in which a distributed update transaction can occur in Forest. Given that the system builder is based on the abstract value of immutable components rather than uids, and that we can copy any package version from one database to another at will, it is not clear that inter-database references should be encouraged. We expect that a group of collaborating programmers will share a single database, and we postulate that sharing among larger groups might be served best by copying. However, this a matter for further research.

6 Safety

The shared memory model creates the potential for corruption by rogue applications of critical data structures, for example, those that underpin the folder structure and versioning system,. The solutions are either to abandon the shared memory model or to rely on the safety of the programming language in which the system is implemented. Cedar [15] is a classic example of the latter approach.

The potential for corruption is real, since C++ is unsafe, but ObjectStore provides some mechanisms to mitigate this. For example, a program fault is caught and translated into an exception that aborts the transaction, leaving the database unscathed. It is also possible to leverage the type information stored with the database to check the validity of pointers prior to committing a modified page.

In the long term we would prefer a safer implementation language. Compiler support for safe C++ [16], with garbage collection for transient objects, would be a viable alternative.

7 Performance Measurements

Although we have not yet put the prototype system into everyday use, early indications are encouraging. With no tuning at all, the prototype performs quite adequately. We intend to develop the prototype into a working system that we will use ourselves for everyday development.

Meanwhile, in order to estimate the performance of the environment in real use, we have developed a test program that simulates a user performing a set of editing operations. The test database, characteristic of a typical medium-sized project, contains approximately 2500 text components, mostly C++ source files, imported from a UNIX file system. These are distributed among 216 packages, each of which corresponds to a directory in the file system. The initial size of the database is approximately 18Mb.

The program simulates a user checking out a package at random, choosing a random number of components of that package to edit, editing them (by making copies), checking the package in, and repeating this process for a given number of iterations. Unlike real users, the test program performs edits in zero time and so provides a somewhat more stressful test than would occur in real use.

Each instance of the program simulates one user. By running the program on several machines, all accessing a shared database on the server, we can discover how the system scales as users are added. In particular we can measure the effect of making each package a uid generator. All times are wall clock since, in a distributed system, that is the only measurement of relevance to users.

The experiments were run on dual-processor SPARCstation™ 10 client machines with 64Mb of memory, connected by a 10 Megabit Ethernet® to a six-processor SPARCcenter™ 2000 running a beta version of the ObjectStore 4.0 database server. The database was stored in a UNIX file, a convenient feature of ObjectStore, but one that obviously limits performance in comparison to storing the database on a raw disk.[1] Client cache size, the amount of virtual memory available for mapping in database pages, was set to the default value of 8Mb. For this benchmark, which only accesses about 0.5Mb of data, the cache size is not an issue.

The results break down the run time into six activities: initial scan, checkout, checkin, analyzing and constructing a new `BuildableFolder`, reading the text component data, and writing it back in the new component. Each of these operations is implemented as an independent transaction to maximize the potential for concurrency. The initial scan walks the top-level folder structure and counts the number of packages. As a side-effect this partially warms each client cache. Table 1 shows the results when the database contains a single uid-generator. These clearly confirm that contention for write access to data is disastrous in the distributed shared memory model. For operations that require object creation, it is as if each new client adds its load to every other client machine. In other words, no benefit is accrued from distributing the processing among multiple client machines.

1. ObjectStore can also store databases on a raw file system.

Table 2 shows the effect of distributing the uid generation amongst the packages. These results are much more encouraging. In particular the sum of the checkout and write times only doubles as we go from one to eight clients, because of the much lower contention for locks. The consistent times for the scanning and reading phases indicates that the database server is affected only slightly by read-only clients and that the client side caching is effective. Clearly, as the number of clients increases, the probability that the same package will be chosen for editing by more than one client increases, and this does indeed occur during this experiment. This causes more communication between the server and clients in order to reclaim locks, and therefore increases the real-time delay for the affected client.

As noted earlier, the critical measure is the time taken to write back modified components and check in a new package version. In this experiment an average of five components are modified in each checkout/edit/checkin cycle. The time to commit the updates ranges from approximately five to twelve seconds. For the common case, a modification to one file, the results suggest a range of two to five seconds. While we would prefer an upper bound on the order of one second, these figures are acceptable given the prototype nature of the system.

8 Related Work

The Vesta system [12] [17] [18] [19] clearly demonstrated that configuration management could be placed on a firm foundation through the use of immutable components and modular system modelling. The Vesta repository, however, remained essentially file-based, and was implemented on top of an existing file system. Tools (*bridges* in Vesta parlance) were still required to pickle language level objects into repository files and the caches for the builder were also stored in this way. Much of the implementation was concerned with implementing key transactional operations in terms of the underlying file-system. The OODB infrastructure provides this directly, while also supplying more flexibility and extensibility, and a consistent object model from the top-level structure of the repository to the fine-grained structure used by tools. To illustrate the leverage of the OODB, it is stated in [18] that the Vesta repository took less than one person year to implement. In contrast the Forest configuration management subsystem took less than one person month to build.

Table 1: Simulation Results with a Single UID Generator

	Number of Concurrent Users							
Operation	1	2	3	4	5	6	7	8
Initial Scan	36[a]	40	39	44	45	53	57	106
Checkout	21	39	94	114	155	157	195	226
Checkin	16	36	83	92	123	172	179	217
Folder Edit	24	60	92	141	152	153	180	242
Read	29	31	29	30	30	32	30	33
Write	105	162	372	526	682	814	940	1145

a. Measurements are in seconds of wall clock time for 25 iterations.

Table 2: Simulation Results with Multiple UID Generators

	Number of Concurrent Users							
Operation	1	2	3	4	5	6	7	8
Initial Scan	44[a]	43	42	44	48	52	60	65
Checkout	21	26	27	37	49	51	60	76
Checkin	16	16	21	24	22	26	28	31
Folder Edit	21	23	26	30	34	37	37	44
Read	30	30	30	31	32	34	33	37
Write	98	109	127	134	136	181	186	222

a. Measurements are in seconds of wall clock time for 25 iterations.

Vesta was itself influenced by the Cedar System Modeler [20]. The Cosmos [21] project also bases its configuration management on immutable objects.

PCTE [1] includes a database management system, OMS [22], which provides an entity-relationship model with object-oriented extensions, but it was not designed to support fine-grained components. The PACT project [23] developed configuration management services on top of OMS.

ClearCase® [24] and DSEE [25] both support a versioned file system with *views* to select particular versions in individual user contexts. Both support integrated system building. Clearcase is a hybrid system, storing some data in conventional files but storing versioning information and other attributes, some of which can be user-defined, in a database.

Magnusson et. al. [26] describe a revision control system for fine-grained, tree-structured, documents implemented with a client-server model and a specialized database. Their storage model for documents is very similar to our scheme for `BuildableFolders`, except that their documents are strictly tree-structured, whereas our import components produce graph-structured configurations. Because our versioning system is independent of component type, we believe that it can be re-used to represent arbitrary versioned documents. We share their view that a change to an internal node in a document requires the entire document to be revised.

ObjectStore also provides generic version management for arbitrary C++ objects, using a binary differencing mechanism for deltas between versions. References (pointers) to other objects *float* to an appropriate version as pages are mapped in, based on a selection mechanism that is similar to the view selection of ClearCase. This contrasts with the immutable configuration bindings that characterize the Vesta approach. In addition, checkout/checkin is tied to a hierarchy of workspaces, with the understanding that changes will eventually propagate from the leaves to the root of the hierarchy. This approach, often referred to as the *copy-modify-merge* model, is also taken by Teamware [27], which uses a combination of whole directory trees and SCCS [28] files to represent a configuration. In contrast the Vesta approach tries to minimize the need for copy-modify-merge by using modularization and interfaces between subsystems. We conjecture that both approaches have a role to play in configuration management, but with copy-modify-merge best restricted to fine-grained components for which modularization is not applicable.

Onodera [29] describes experience representing fine-grain C++ program information in a database, also using ObjectStore. The results show that a more global approach to storing information on multiple programs does reduce overall space requirements. It is also noted that pointer-based structures are inherently less space efficient than the tight encodings in current file formats. We believe that our approach of separating interface and implementation and the judicious use of transient objects, can maintain an interface that is convenient for programming yet space efficient in the database.

Baker [30] has argued that functional objects can simplify the construction of distributed and parallel computations, and his conclusions echo many of the ideas of the Vesta approach.

9 Conclusions

We have described the design and implementation of a prototype modern configuration management system. It follows the approach pioneered by Vesta, but is implemented on a commercially available object-oriented database.

Our experience suggests that transacted data, combined with typed, immutable components, provide an excellent basis for a software development environment. Many problems of scale that were experienced by earlier efforts can be solved by the pervasive application of incremental techniques. These are made possible by the combination of immutability and abstract interfaces to objects.

Our performance measurements suggest that it is no longer necessary to build a special purpose repository to support an SDE, and that the current generation of object-oriented databases can provide an adequate infrastructure, provided applications pay appropriate attention to object clustering.

10 Future Plans

Our future plans are mainly in two areas. First, we intend to evolve the prototype into a system that we can use for everyday programming, including the development of the system itself. Our current prototype supports an experimental evolution of C++ [31], which afforded us the luxury of designing the compiler to exploit the database directly. A working system for ANSI C++ will be somewhat more constrained, but we expect at least to replace the use of ad hoc file databases for derived information. For unmodifiable tools, Vesta's bridge techniques [17] are directly applicable.

The second area of research is to expand and exploit the use of fine-grained components in all areas of the software development environment. We plan to investigate the integration of structured document editors with the configuration management system. We are also interested in exploiting component abstract values to achieve build-avoidance at a finer grain, using the basic mechanisms of the system builder.

11 Acknowledgments

Jon Gibbons designed and implemented a prototype version of the Forest component system. Ted Goldstein designed the `DBNewPlace` interface. Roy Levin provided helpful insight and discussions on Vesta.

12 Trademarks

13 References

[1] An Overview of PCTE and PCTE+, Gerard Boudier, Ferdinando Gallo, Regis Minot and Ian Thomas, *Proceedings of the ACM/SIGSOFT Software Engineering Symposium on Practical Software Development Environments,* Boston, Massachusetts, November 1988, 248-257.

[2] Types and Persistence in Database Programming

Languages, Malcom P. Atkinson and O. Peter Buneman, *ACM Computing Surveys* 19,2 (June 1987) 105-190.

[3] The ObjectStore Database System, Charles Lamb. Jack Orenstein and Dan Weinreb, *Communications of the ACM* 4,10 (October 1991) 50-63.

[4] *The Annotated C++ Reference Manual*, Margaret A. Ellis and Bjarne Stroustrop, Addison-Wesley, Reading, Massachusetts, 1990.

[5] Make—A Program for Maintaining Computer Programs, Stuart I. Feldman, *Software--Practice & Experience* 9,3 (March 1979) 255-265.

[6] T*he Common Object Request Broker: Architecture and Specification*, Object Management Group, Document No. 91.12.1, 1991.

[7] *Inside OLE 2*, Kraig Brockschmidt, Microsoft Press, ISBN 1-55615-618-9, 1994.

[8] A Simple and Efficient Implementation for Small Databases, Birrel et al., *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, August 1987.

[9] A Study of Pickling Emphasizing C++, Daniel Craft, Olivetti Software Technology Laboratory Technical Report STL-89-2, September 1989.

[10] An Extensible Programming Environment for Modula-3, Mick Jordan, *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, Irvine, California, December 1990, 66-76.

[11] *Smalltalk-80, The Language and its Implementation*, Adele Goldberg and David Robson, Addison-Wesley, Reading, Massachusetts, 1983.

[12] The Vesta Approach to Configuration Management, Roy Levin and Paul McJones, DEC Systems Research Center TR 105, June 1993.

[13] Some applications of Rabin's fingerprinting method, Capocelli et al. (ed), *Sequences II: Methods in Communication, Security and Computer Science*, Springer-Verlag, New York, 1991.

[14] Design, Implementation, and Evaluation of a Revision Control System, Walter F. Tichy, P*roceedings 6th International Conference on Software Engineering*, Tokyo, Japan, September 1982, 58-67.

[15] A Structural View of the Cedar Programming Environment, Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach and Robert B. Hagmann, *ACM Transactions on Programming Languages and Systems* 8,4 (October 1986) 419-490.

[16] Safe, Efficient Garbage Collection for C++, John R. Ellis and David L. Detlefs, *USENIX C++ Conference Proceedings*, Cambridge Massachusetts, April 1994, 143-177.

[17] Bridges: Tools to Extend the Vesta Configuration Management System, Mark R. Brown and John R. Ellis, DEC Systems Research Center TR 108, June 1993.

[18] The Vesta Repository: A File System Extension for Software Development, Sheng-Yang Chin and Roy Levin, DEC Systems Research Center TR 106, June 1993.

[19] The Vesta Language for Configuration Management, Christine B. Hanna and Roy Levin, DEC Systems Research Center TR 107, June 1993.

[20] Practical Use of a Polymorphic Applicative Language, Butler W. Lampson, and Eric E. Schmidt, *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 1983, 237-255.

[21] A Unifying Model for Consistent Distributed Software Development Environments, J. Walpole, G. S. Blair, J. Malik and J. R. Nicol, P*roceedings of the ACM/SIGSOFT Software Engineering Symposium on Practical Software Development Environments*, Boston, Massachusetts, November 1988, 183-190.

[22] The Object Management System of PCTE as a Software Engineering Database Management System, Ferdinando Gallo, Regis Minot and Ian Thomas, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Palo Alto, California, December 1986, 12-15.

[23] Configuration Management in the PACT Software Engineering Environment, Ian Simmonds, *Proceedings of the 2nd International Workshop on Software Configuration Management*, Princeton, New Jersey, October 1989, 118-121.

[24] *ClearCase Concepts Manual*, Atria Software, 1992.

[25] Computer-Aided Software Engineering in a Distributed Workstation Environment, David B. Leblang and Robert P. Chase, Jr., *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development*

Environments, Pittsburgh, Pennsylvania, April 1984, 104-112.

[26] Fine-grained Revision Control for Collaborative Software Development. B. Magnusson, U. Asklund, S. Minör, Lund Institute of Technology, Department of Computer Science, LU-CS-TR:93-112.

[27] *CodeManager User's Guide*, Sun Microsystems Inc., Part No. 801-2169-11.

[28] The Source Code Control System, Marc J. Rochkind, *IEEE Transactions on Software Engineering*, SE-1,4 (December 1975) 364-370.

[29] Experience with Representing C++ Program Information in an Object-Oriented Database, T. Onodera, *Proceedings Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, October 1994, 403-413.

[30] Equal Rights for Functional Objects or, The More Things Change, The More They Are the Same, Henry G. Baker, *ACM OOPS Messenger* 4,4 (October 1993) 2-27.

[31] The Clarity Language Definition, Mick Jordan et. al. (ed) Sun Microsystems Laboratories, Technical Report: *in preparation*.