Resolving the Integrity/Performance Conflict

Andrew C. Myers
andru@lcs.mit.edu
MIT Laboratory for Computer Science*

Abstract

Future applications will require integrity of complex, persistent data in the face of hardware and program failures. Thor [6], a new object-oriented database, offers a computational model that ensures data integrity in a distributed system without sacrificing expressiveness or performance. Perhaps surprisingly, compiler technology is important to making this work well.

1 Traditional Approaches

Currently, writing applications that share complex, persistent data across a network is too hard, because the underlying software substrate is not available. No existing system combines high performance, complete data integrity, and a sufficiently powerful data model.

Allowing multiple applications to share complex persistent data creates problems because of the desire for protection. Increasingly, persistent data contains semantic information. It may only be accessed in particular ways that ensure its integrity is not violated. Data may be accessed by multiple users, or shared across a network which can experience failures. Failures of hardware, or of users to properly access objects, cannot be allowed to compromise persistent data integrity.

The conventional approaches to providing these services have been operating systems, databases, and persistent programming languages. Each of these systems solves part of the problem just outlined. Unfortunately, none can provide a complete solution. This paper presents an opinion about what each conventional paradigm contributes and then explains how, in this context, Thor solves the complete problem.

1.1 Operating Systems: Firewalls

Operating systems have provided the traditional approach for the integrity of persistent data in the presence of potentially invalid accesses. In the most common structure, the machine is separated into two pieces: *kernel space* and *user space*. Only a fixed and limited portion of the persistent data can be fully accessed from user space: a file's contents. Other persistent information, such as directory structures and links, can only be accessed through system calls, which provide a narrow interface for crossing the protection boundary from user space into kernel space.

The advantage of this design is that arbitrary operations may be performed in user space, and so user programs can be very fast as long as they are not accessing persistent data. This fact leads to a computational paradigm in which applications load data from a file when they start, then save on exit.

Unfortunately, a file's contents have no meaning of their own — the meaning is completely defined by each program that can access the file. Also limiting is the fact that the operations provided by the system calls are fixed and cannot be extended by the user. Users often encounter a semantic mismatch when they try to force the desired behavior of their data into the fixed metadata framework provided by the particular operating system in use.

The fact that the most successful operating systems (*e.g.* Unix) have not imposed a semantic

*545 Technology Square, Cambridge, MA 02139, USA. This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136 and in part by the National Science Foundation under Grant CCR-8822158.

framework on file data, merely suggests that it is better to have no built-in model at all than to impose a fixed, limited model.

1.2 Persistent Programming Languages: Encapsulation

A second aspect of handling complex, persistent data is guaranteeing data integrity while allowing the user to define new operations on this data. The traditional approach to this problem has been the use of *safe* programming languages, in which only specified operations can be performed on data objects. Allowing object types to define the allowed operations ensures integrity, but it may also be limiting. For this reason, persistent programming languages are often object-oriented, or add features for schema evolution, or have only runtime type-checking. These features provide extensibility, but cause safe, persistent programming languages to suffer from performance problems.

1.3 Distributed Databases: Transactions

In the database community, users have been concerned about maintaining data integrity in a distributed system. Data is stored in a number of separate locations, each of which may fail or lose contact with the others. The possibility of partial failure may allow an update to the system to be partially lost, leading to an inconsistent data state. Even without hardware failure, inconsistent states can also be generated by concurrent accesses by different users.

To avoid inconsistent states, *transactions* have been used to wrap up a set of complex operations into a single, *atomic* update. Atomicity ensures that the update is either performed completely, or else fails and has no effect — leaving the system in the previous consistent state.

The limitation of most current databases lies in another domain — the simplicity of the data model. In a relational database system, there is no satisfactory way to encapsulate objects and control the ways that they can be accessed or modified [1]. This means that object integrity cannot be provided at a sufficiently rich semantic level to allow arbitrary and fine-grained sharing of data with other users.

2 Synthesis: Thor

Thor unifies these three approaches to persistent data. Like operating systems, it has a user space/kernel space partitioning that allows client applications to run quickly in an unsafe domain. However, in the protected "kernel space" runs a safe, persistent, object-oriented programming language called *Theta*. The presence of a powerful programming language on the safe side of the protection wall means that system calls are no longer limited to a fixed set of operations. Indeed, users of Thor can define their own object operations and even add new ones as the system is running. Finally, to provide support for distributed programming, Thor has transactions that allow programs to avoid the inconsistent object states that would arise from hardware failure or concurrent modification.

2.1 Replacing the Operating System

Thor can be viewed as a new kind of operating system. Instead of being restricted to a fixed set of safe operations, users can define new operations at will. Essentially, applications are allowed to throw new code over the protection firewall and have the kernel execute this code on their behalf. Because the code is written in a safe programming language, the kernel can execute it without concern for protection.

By contrast, ordinary operating systems only allow extension of the kernel by recompiling and rebooting the system — and no guarantees are made about the comprehensibility of the old persistent data in the new kernel. Spy [5], a system-monitoring facility, allowed code to be placed into the kernel, but for safety reasons the code could only observe and could not change any behavior.

Like operating systems, Thor has a protection boundary that is potentially expensive to cross. This expense means that users will want to cross it as little as possible. There are two obvious techniques for minimizing boundary crossings.

One technique is just the file-system approach of loading all data into user space, operating on it there, and putting it back into Thor when done. This approach is limiting, because the objects fetched into user space must be completely accessible and

modifiable by every user who can access them at all. Guaranteeing object integrity in the presence of unrestricted access is essentially impossible. This approach will work well only for simple object structures that have no integrity constraints, or immutable object structures where violation of object constraints can only occur in the user space, and cannot be observed by other users.

The other approach to minimizing protection-boundary crossings is to move much of the application's computation into the safe side — in other words, do computation in Theta. Operations that violate object integrity are made impossible simply by the nature of the safe programming language. For most purposes, this is the reasonable way to use Thor.

2.2 Don't Slow Down Computation

An immediate concern about using Theta for computation is that safe, object-oriented programming languages with schema evolution, transactions, and orthogonal persistence are thought to be too slow to use for real computation. Slowness kills: application programmers are notorious in their distaste for slow programming languages.

Making Theta fast is critical to this computational model. Speed will be achieved partly by using an extension of the *specialized compilation* techniques pioneered by SELF [3]. General-case code is compiled into fast, specialized versions that are selected dynamically, based on the state of the Theta runtime. Language features like object-oriented dispatch, transactions, persistent objects, and schema evolution exact a minimal penalty when they are not being used. Other work in progress will minimize the impact of these features when in use.

Thor is similar in goals to other object-oriented databases, and is perhaps most like GemStone in overall design [2]. However, Thor's internal language has a static (though flexible) type system that supports high-performance computation. Because the internal language is fast, the safe-computation model is workable. Object-oriented databases such as ObjectStore [4] sacrifice some data integrity and schema evolution flexibility to provide high performance and orthogonal persistence.

3 Summary

Future applications will require integrity of complex, persistent data in the face of hardware and program failures. Thor offers a computational model that ensures data integrity without sacrificing expressiveness or performance. As with conventional operating systems, a protection barrier provides protection against invalid data accesses. However, the protection barrier does not limit complexity or destroy encapsulation of data objects. As in databases, transactions provide reliability in the face of hardware failures and concurrent accesses. And Thor has a powerful object model that allows persistent data to evolve. In addition to the usual database considerations, the performance of this system is tied to compiler technology that mitigates the performance problems typically associated with persistent programming languages.

References

[1] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In J. Kim, J.M Nicholas, and S. Nishio, editors, *Proc. of the First International DOOD Conference*, Kyoto, Japan, December 1989.

[2] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[3] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented language. In *Proceedings of the SIGPLAN '89 Conference on Programming Languages and Implementation*, pages 146–160. ACM, July 1989.

[4] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[5] B.W. Lampson. Hints for computer system design. *IEEE Software*, January 1984.

[6] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in Thor. In M. Tamer Özsu, Umesh Dayal, and Patrick Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann, San Mateo, California, 1993.