JPS: A Distributed Persistent Java System

by

Chandrasekhar Boyapati

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1998

Author ....
Department of Electrical Engineering and Computer Science
September 14, 1998

Certified by ....
Barbara Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Committee on Graduate Students

JPS: A Distributed Persistent Java System

by

Chandrasekhar Boyapati

Submitted to the Department of Electrical Engineering and Computer Science
on September 14, 1998, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Distributed persistent object systems provide a convenient environment for applications that need to manage complex long-lived data. Since Java has no persistence model built into it, the tremendous growth in the popularity of Java has generated a lot of interest in systems that add persistence to Java. This thesis presents the design, implementation and performance evaluation of a *Java Persistent Store* called JPS. JPS is an efficient distributed persistent Java system built on top of the Thor object-oriented database.

JPS provides several advantages over other persistent Java systems. Unlike most of other systems, JPS is a distributed system that allows multiple clients to simultaneously access the object store. JPS is built to be used over a wide area network and scales well with large databases. JPS also provides a very reliable and highly available storage.

More importantly, JPS offers significantly better performance for many important types of workloads. The original Thor system that used Theta as the database programming language has been extensively compared to other OODBs and shown to consistently outperform other systems—sometimes by more than an order of magnitude—under a wide variety of commonly used workloads. Our performance results indicate that JPS performs almost as well as the Theta-based system.

Thesis Supervisor: Barbara Liskov
Title: Ford Professor of Engineering

Acknowledgments

I would like to thank Barbara Liskov, my thesis supervisor, for her patience, counsel and support. Her insightful comments have been most valuable for my thesis. The presentation of this thesis has been significantly improved because of her careful and critical proof-reading.

Atul Adya, Miguel Castro and Andrew Myers helped me understand the various aspects of Thor design and implementation. They have also been wonderful companions.

Jason Hunter has been a great office-mate and a good friend. Other members of the lab have also contributed to make the lab a pleasant environment. I would like to thank Arvind, Dina, Dorothy, Doug, Kavita, Liuba, Paul, Peter, Phil, Rahul, Sandeep, Shan-Ming, Umesh and Zheng for providing a friendly and relaxed work atmosphere.

Prasad and Shiva have been terrific house-mates and great sources of fun.

Finally, my family has been very supportive of my goals and efforts. I thank my parents for the many things they have done to make this possible.

Contents

List of Figures

Chapter 1

Introduction

Distributed persistent object systems provide a convenient environment for applications that need to manage complex long-lived data. These systems are drawing increasing interest from application designers whose data requirements are not met by traditional data management systems. Commonly cited application areas that use persistent object stores include computer aided software engineering (CASE) tools [LLOW91], computer aided design and computer aided manufacturing (CAD/CAM) [CDN93], geographical information systems (GIS) [DKL+94], office automation and document preparation systems [BM92].

Since Java [GJS96] has no persistence model built into it, the tremendous growth in the popularity of Java has generated a lot of interest in systems that add persistence to Java. This thesis describes the design and implementation of a *Java Persistent Store* called JPS. JPS is an efficient distributed persistent Java system built on top of the Thor [LAC+96, CALM97] object-oriented database.

There are many other research and commercial efforts underway to add persistence to Java. These include PJama [AJDS96], which is being built by Sun and the University of Glasgow, the Persistent Storage Engine for Java [PSE] developed by ObjectStore, and GemStone/J [Gem] developed by GemStone. Also, Oracle is planning to put Java into their next version of the Oracle8 database server, release 8.1. However, our approach offers two important advantages over these other systems.

Firstly, JPS is a more complete system. Unlike most of the systems mentioned above, it allows multiple clients to simultaneously access the object store without interfering with one another. Furthermore, JPS is built to be used over a wide area network and scales well with large databases. JPS also provides a very reliable and highly available storage for objects.

Secondly, JPS offers better performance for many important types of workloads. All the other above-mentioned systems use either an ordinary JVM [LY97], or a slightly modified JVM to run Java programs. Unfortunately, ordinary JVMs are not optimized for the kind of workloads that are common in distributed persistent object systems. For example, ordinary JVMs are not built to efficiently handle a large collection of objects that do not all fit in main memory, or a large number of clients simultaneously accessing the object store.

Good performance for a distributed object storage system requires good solutions for client cache management, storage management at servers, and concurrency control for transactions. Since JPS is built on top of Thor, it offers efficient solutions to the above problems. Various performance studies have shown that these solutions consistently outperform other techniques—sometimes by more than an order of magnitude – on a wide variety of common workloads [LAC+96, CALM97, AGLM95, Gru97, Ghe95].

1.1 Research Approach

The goal of this thesis is to design and implement persistence for Java using Thor as a basis. To achieve this goal, we need to solve two key problems.

The first problem is to define the programming model provided by JPS. A key issue here is understanding what it means to make Java objects persistent. For example, it is unclear what it means to store objects that represent, say, a thread or an open file descriptor persistently, or what it means to restore these objects when they are accessed later, potentially by a different application, and potentially on a different client machine. Therefore, we need to either provide an answer to this question, or we need to have a way of preventing such objects from being made persistent. And if we follow the latter approach, our solution must not impose any unnecessary restrictions on the natural way of programming in Java.

The second problem is to provide an implementation for JPS that has good performance just like the original Thor system. Since Thor's runtime system is optimized for good performance, we did not want to replace it with the runtime system of a standard Java interpreter. Consequently, we do not use a standard Java interpreter to run Java programs in JPS.

Our basic approach for running Java programs is as follows. We first translate

Java bytecodes into C code that interfaces appropriately with the Thor runtime environment. We then run an optimizing C compiler on the generated C code to produce native machine code.

Instead of writing a bytecodes-to-C translator from scratch, we made use of Toba [PTB+97], an existing Java bytecodes-to-C translator developed at the University of Arizona. We modified the Toba translator appropriately to make it work with our system.

The original Thor/Theta system has been extensively compared to other OODBs and is shown to consistently outperform other systems under a wide variety of commonly used workloads [LAC+96, CALM97, AGLM95, Gru97, Ghe95]. Our experimental results indicate that JPS performs almost as well as the Theta-based system.

1.2 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 provides some background on Thor. Chapter 3 describes the JPS programming model. Chapter 4 gives an overview of the JPS implementation. Chapter 5 presents the programming interface that JPS provides to users. Chapter 6 describes the performance evaluation of JPS. Chapter 7 describes related work on other persistent Java systems. Finally, chapter 8 summarizes this thesis and describes ways this work could be extended.

Chapter 2

A Sketch of Thor

This chapter presents an overview of the Thor object-oriented database (OODB). Section 2.1 describes the semantics of the Thor system and the interface presented by Thor to the outside world. Section 2.2 briefly describes the implementation of the Thor OODB, emphasizing the important techniques that Thor incorporates to enhance performance.

2.1 Thor Interface

2.1.1 Object Universe and Transactions

Thor provides a universe of persistent objects. It is intended to be used in a distributed system consisting of many clients and many servers. Objects are stored on server machines. They contain references to other objects, which may reside at any server. Thor maintains a set of persistent roots, one per server. All objects reachable from any persistent root are defined to be persistent—the rest of the objects are garbage.

Applications run on client machines and interact with Thor by calling methods on objects. These calls are made within atomic transactions. If a transaction commits, Thor guarantees that the transaction is serialized with respect to all other transactions, and that its modifications to the persistent universe are recorded reliably. If a transaction aborts (say, because of using stale data), any modifications it performed are undone at the client and it has no effect on the persistent state of Thor.

2.1.2 Theta: The Database Programming Language

To preserve the integrity of the database, Thor guarantees *type-safe* sharing of objects. Type-safe sharing requires that the database programming language have two properties: (1) *type correct calls*—every method call goes to an object with the called method and signature, and (2) *encapsulation*—only code that implements an object can manipulate its representation. To ensure type-safety, the original implementations of Thor required that the method code for implementing persistent objects be written in Theta [LCD+94], a statically typed programming language that enforces strict encapsulation.

But Java is also a type-safe object-oriented programming language and can be used equally well as the database programming language in Thor. To this end, we have modified Thor to replace Theta with Java.

2.1.3 Writing Applications in Unsafe Languages

Thor allows application code to be written non-*type-safe* languages like C++ without compromising the integrity of the database. Thor runs the unsafe code in a separate protection domain on the client machines. Every call to Thor from the application code is type-checked dynamically to ensure that the object being called does exist and has the called method, and that method invoked has a compatible signature. The application code refers to the database objects only via opaque *handles*, and communicates with the rest of the Thor code through a thin interface called the *veneer*. This architecture is shown in figure 2-1.

As illustrated, the interface is very simple. In addition to method calls and transaction commits and aborts, there is a way to obtain the root directory associated with a particular server. (Servers are named by their IP addresses, which can be obtained by looking them up in the DNS [TPRZ84].) The ability to access the root allows an application to shut down a session with Thor and then start a session later and retrieve its objects. Access to roots is the only way an application can retrieve objects later; handles are valid only for the duration of a session. This restriction is what allows us to do garbage collection, since it provides a set of well-defined roots (the persistent roots and the handles of active sessions).

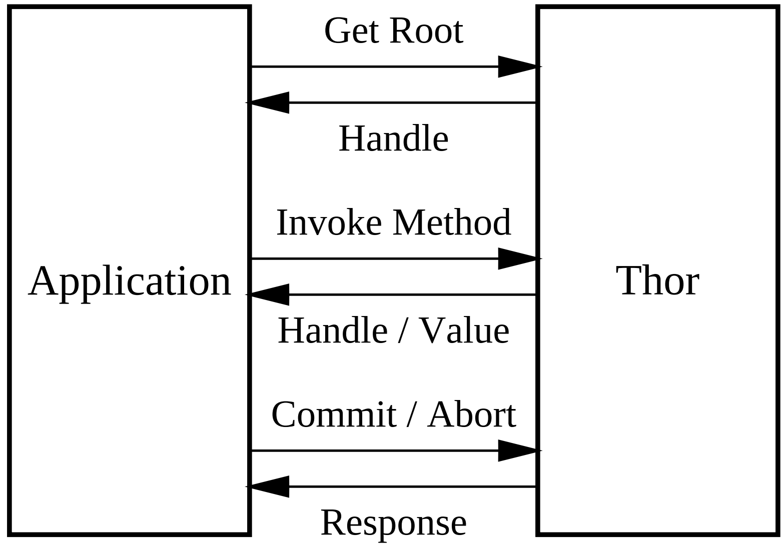

Figure 2-1: Application interface in Thor

2.2 Thor Implementation

2.2.1 Client-Server Architecture

Thor has a client-server architecture where applications run at the clients and perform all accesses on cached copies of database objects. All computation within a transaction takes place locally at the client. A portion of Thor called the front end (FE) runs at each client to maintain the client cache and to do concurrency control. Figure 2-2 provides an illustration. It shows a client with an FE that contains cached copies of two persistent objects.

Moving both data and computation to the clients has two main advantages. First, work is offloaded from the servers to the clients, improving the scalability of the system and allowing it to take advantage of the CPU and memory resources available at the clients. Second, many object accesses can be performed without contacting the servers, allowing efficient fine-grained interaction between an application and the OODB.

Persistent objects are stored at servers called *object repositories* (ORs). Each OR stores a subset of the persistent objects. Thor provides a highly reliable and highly available storage for persistent objects. To achieve high availability, each OR is replicated at a number of server machines. Thor currently uses a primary/backup replication scheme [LGG$^+$91, Par98].

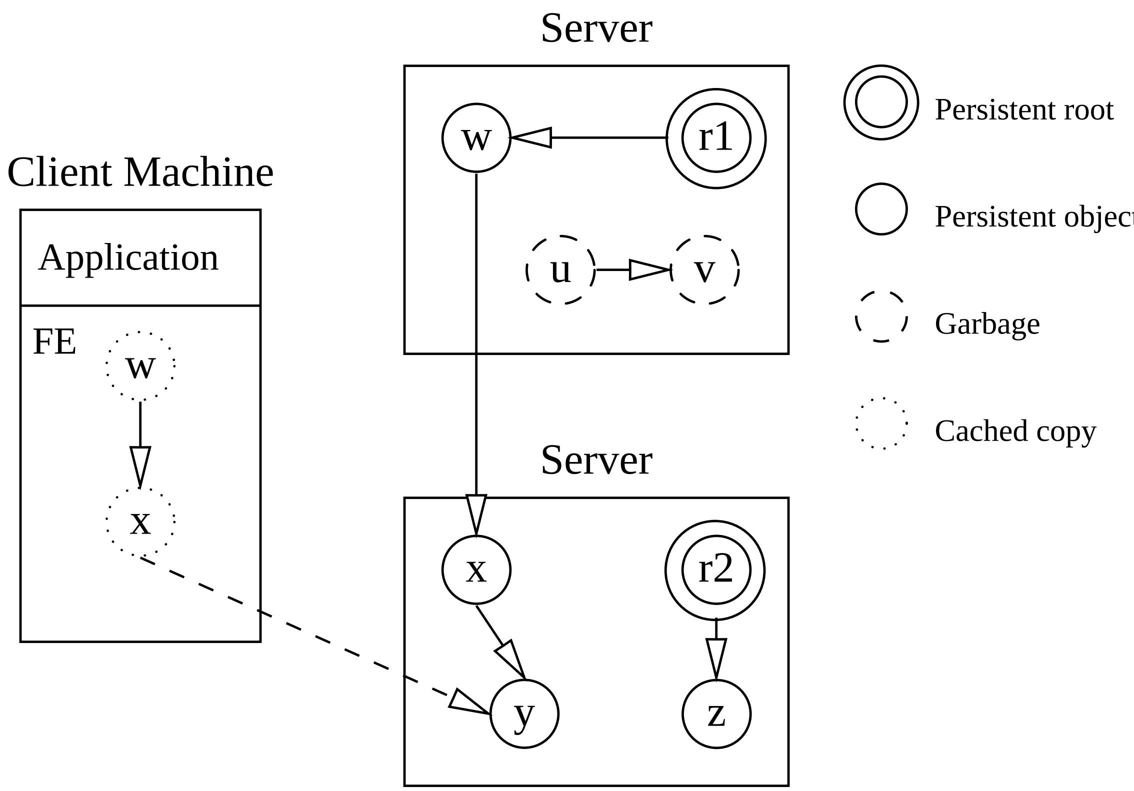

Figure 2-2: The Thor architecture

2.2.2 Storage Management at Servers

Servers store objects on disks in pages and maintain a page cache in main memory to speed up client fetch requests. When a transaction commits, its FE sends information about its modifications to the servers in the commit request. If the commit succeeds, the changes become visible to later transactions. Thor uses object-shipping and does not ship the containing pages to the servers; moreover, at commit time, it ships only new versions of the modified objects. This approach reduces communication costs if the size of an object is much smaller than the size of a page, which is what we expect in the common case.

Object shipping could lead to poor performance if the objects' pages were read immediately from the disk to install the objects in the database. Thor avoids this cost by using the *modified object buffer* (MOB) architecture [Ghe95]. The server maintains an in-memory MOB that holds the latest versions of objects modified by recent transactions. New versions are written to the MOB when transactions commit; when the MOB fills up, versions are written to their disk pages in the background.

The use of a MOB does not compromise the reliability of the system. This is because Thor maintains a stable transaction log in which it records information about committed transactions before sending the commit reply. This log is replicated on backup machines. Every OR sends the modifications it receives to backup ORs running on other machines at commit time.

Experimental results show that for typical object-oriented database access patterns, the MOB architecture out-performs the traditional page-based organization of server memory that is used in most databases.

2.2.3 Object References

Keeping objects small both at servers and clients is important because it has a large impact on performance [WD94, MBMS95]. Objects in Thor are small primarily because object references (or *orefs*) are only 32 bits. Orefs refer to objects at the same OR; objects point to objects at other ORs indirectly via *surrogates*. A surrogate is a small object that contains the identifier of the target object's OR and its oref within that OR [DLMM94]. Surrogates do not impose much penalty in either space or time because the database can usually be partitioned among servers so that inter-server references are rare and followed rarely.

It is not practical to represent object pointers as orefs in the client cache because each pointer dereference would require a table lookup to translate the name into the object's memory location. Therefore, many systems perform *pointer swizzling* [WD92, KK90]; they replace the orefs in objects' instance variables by virtual memory pointers to speed up pointer traversals. Thor uses indirect pointer swizzling [KK90]; the oref is translated to a pointer to an entry in an indirection table and the entry points to the target object. Indirection allows the Thor FE to move and evict objects from the client cache with low overhead.

In-cache pointers in Thor are 32 bits, which is the pointer size on most machines. Thor uses 32-bit pointers even on 64-bit machines—it simply ensures that the cache and the indirection table are located in the lower 2^{32} bytes of the address space. Consequently, every time an object reference is fetched from or stored into an object instance field, it has to be cast appropriately. Our initial implementation was done on machines with 64-bit pointers.

2.2.4 Client Cache Management

Thor uses *hybrid adaptive caching* (HAC) [CALM97] for managing its client cache. HAC is a combination of page caching and object caching that combines the virtues of both while avoiding their disadvantages. HAC partitions the client cache into page frames and fetches entire pages from the server. To make room for an incoming page, HAC does the following:

- selects some page frames for *compaction*,
- discards the cold objects in these frames,
- compacts the hot objects to free one of the frames.

HAC achieves the low miss penalties of a page-caching system, but is able to perform well even when locality is poor, since it can discard pages while retaining their hot objects. It realizes the potentially lower miss rates of object-caching systems, yet avoids their problems of fragmentation and high overheads. Furthermore, HAC is adaptive: when locality is good it behaves like a page-caching system, while if locality is poor it behaves like an object-caching system.

Experimental results indicate that HAC outperforms other object storage systems across a wide range of cache sizes and workloads; it performs substantially better—by more than an order of magnitude—on common workloads that have low to moderate locality.

2.2.5 Concurrency Control

Thor uses *adaptive optimistic concurrency control* (AOCC) [AGLM95, Gru97] to support transactions. This allows applications to run at clients without the need to communicate with servers for concurrency control purposes (eg, to obtain locks). AOCC provides both serializability and external consistency for committed transactions. AOCC uses loosely synchronized clocks to achieve global serialization.

Like all other optimistic schemes, AOCC synchronizes transactions at the commit point, aborting transactions when synchronization fails. When a transaction T commits, the servers also send *invalidation messages* to all clients that have old copies of objects installed by T in their local caches. These messages are not required for correctness, but they improve the efficiency of the system.

Results of simulation experiments [Gru97] show that AOCC outperforms *adaptive callback locking* (ACBL) [CFZ94], which was the best known concurrency control scheme in earlier studies, across a wide range of system and workload settings. In addition to having better performance, AOCC is simpler than ACBL, and scales better with respect to the number of clients using shared data.

A recent study describes a new technique called Asynchronous Avoidance-based Cache Consistency (AACC) [OVU98] that outperforms AOCC on some workloads. However, this study concentrates on a system configuration where part of the method code is run on the servers, which is different from the architecture of Thor.

Chapter 3

Programming Model

This chapter describes the system interface of JPS as seen by an application programmer, along with the motivation for our design choices. JPS is built on top of the Thor OODB described in chapter 2. Hence, it has similar semantics for transactions and persistence as in Thor. However, there are a number of new issues that arise as a result of using Java. This chapter identifies and discusses these issues.

The rest of this chapter is organized as follows. Section 3.1 explains why we think persistence by reachability is the right persistence model for a persistent object store. Section 3.2 describes the issues that arise because of some Java classes like *Threads* whose objects cannot automatically be made persistent. We refer to such classes as *transient* classes. Section 3.3 discusses ways to deal with transient classes in Java. Section 3.4 talks about issues relating to persistence of static variables in Java.

3.1 Persistence by Reachability

There are two fundamentally different ways for doing storage management in a persistent system:

1. *Explicit storage management*, where application programmers, based on their knowledge of object semantics and usage, explicitly store or delete objects from a persistent store.

2. *Persistence by reachability*, where the underlying runtime environment decides to store or delete objects based on their reachability from active applications and from a set of persistent roots. Such systems typically use *garbage collection* to reclaim space from objects that are no longer reachable.

Explicit storage management is used in many systems and programming languages. For example, in file systems users manage their own storage. This approach has less runtime overhead since these systems do not have to implement garbage collection. Programmers can exploit program semantics to decide exactly when an object is useful and when it is not.

However, explicit storage management is a burden on programmers and is prone to errors. For example, if a programmer forgets to store a live object or deletes a live object from the storage by mistake, an application may attempt to use the object and fail. On the other hand, if a programmer forgets to delete a garbage object, its storage will be lost.

Explicit storage management becomes even more problematic in persistent stores for the following reasons. Objects may be shared by many applications and there may be many different ways of accessing them. Hence, it is very difficult for a programmer to decide when to store or delete an object. Furthermore, objects are persistent, so the effects of deleting live objects or retaining garbage objects will last for ever. Therefore, persistence by reachability is the appropriate model for persistent object systems.

3.2 Transient Classes in Java

In the original design of Thor, the database programming language Theta was meant to be used to implement only those classes whose objects would be stored persistently in the database. Consequently, Theta did not provide language support for certain features like I/O and threads.

But in a persistent object system that uses Java as the programming language, new issues arise because of certain Java classes that support features like I/O and threads. This is because the objects of those classes store state that is part of the execution environment of the system. We will refer to such classes as *transient* classes.

It is unclear what persistence services should be provided to objects of transient classes. For example, it is unclear what it means to store objects that represent, say, a thread or an open file descriptor persistently, or what it means to restore these objects when they are accessed later, potentially by a different application, and potentially on a different client machine.

This section presents more details about the uses of transient classes in Java. Section 3.2.1 explains how the use of native methods in Java makes some Java classes transient and why it does not make sense to provide automatic persistence to objects of transient classes. Section 3.2.2 discusses ways of identifying transient classes. Though our system does not allow objects of transient classes to be made persistent, it is possible to design a system that would allow these objects to be stored persistently. Section 3.2.3 describes this alternate design and explains why we decided not to use it in our system.

3.2.1 Transient Classes and Native Methods in Java

Applications occasionally require access to underlying system features or devices for which using Java code is cumbersome at best, or impossible at worst. Because of this, Java allows programmers to write parts of their code in another language and call this code from Java. Such code is called *native* code. Programmers also use native code to improve the speed of certain performance-critical portions of their applications. There are numerous examples of uses of native methods in Java class libraries that form part of the Java core API specification.

Native methods sometimes need to store references to native data structures in objects. For example, in the *java.lang.Thread* class, native methods needs to store a reference to the current execution environment in the Thread object. But Java has no provision for this, since every instance variable in a Java object must either be of a primitive type (like *int*, *boolean*, etc.) or must be a reference to another Java object. Hence, native methods often use Java *int*s as place holders to store such references to native data structures. For example, the java.lang.Thread class has an instance variable named *eetop* of type int. Native methods in java.lang.Thread use *eetop* not as a Java int but as a pointer to *struct execenv*, a C struct that captures the current state of the execution environment of this thread. Figure 3-1 provides an illustration.

Obviously, it does not make sense to store *eetop* persistently as a Java int and restore it later when the object is accessed again. This is because there is a *hidden state* (in the form of a native data structure) associated with each Thread object that is not visible to a JVM that treats *eetop* as a Java int. (Other examples of classes in the Java API with hidden states include *java.io.FileDescriptor*, *java.awt.Color*, *java.awt.Font*, etc.) It is not possible for a JVM by looking at Java bytecodes to track all this hidden data. Hence, objects of such classes should not be allowed to be made persistent.

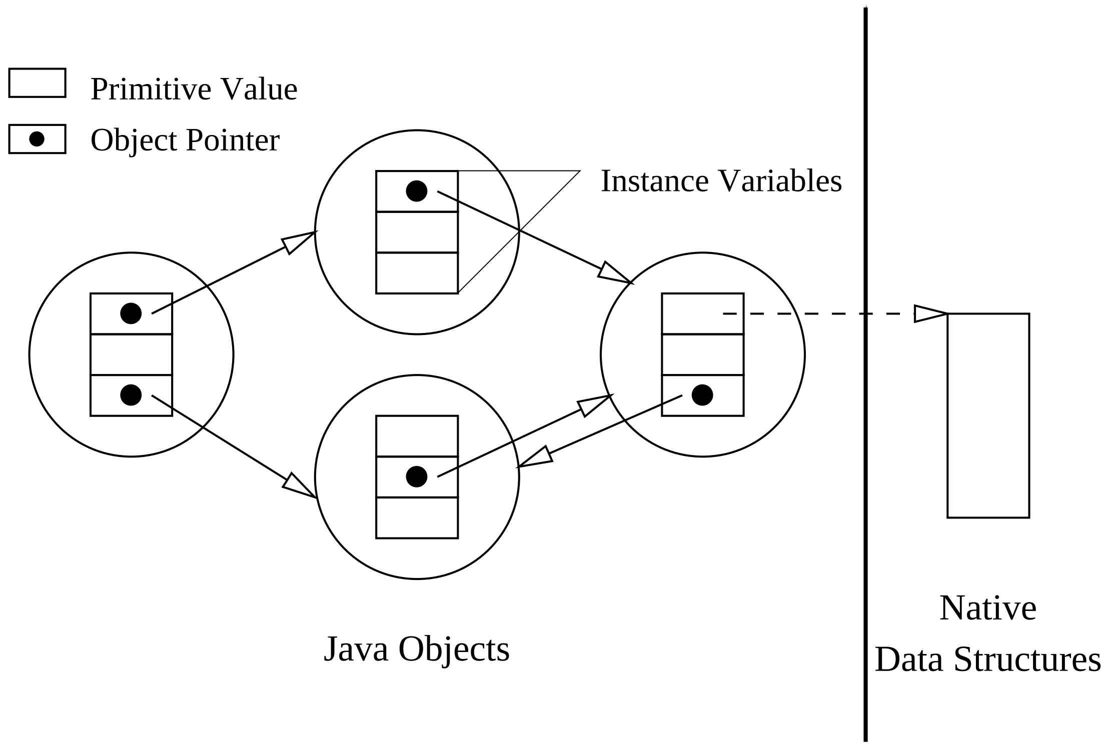

Figure 3-1: Hidden state maintained by native methods

3.2.2 Identifying Transient Classes

Any Java variable that is accessed by native code could potentially contain a reference to hidden data. There is no way for a JVM to automatically identify all the Java variables that are used in violation of their declared types by native methods. Hence, there has to be some way for programmers to convey this information explicitly to the underlying system.

One possible way would be to introduce a new data type in Java, say *native*. All variables that are used to store references to native data would be of type *native*.

Another way would be by using a keyword like *transient* as a class modifier. All transient classes would be explicitly declared to be transient by this keyword.

In fact, Java already has a keyword *transient* reserved for future use. According to the Java language specifications [GJS96], *variables may be marked transient to indicate that they are not part of the persistent state of an object*. In our case, all variables that are used to store references to native data could be marked transient. This solution is similar to the solution of introducing a new data type called native described above.

Incorporating any of the above solutions would involve changing a lot of existing

Java code, including code in Java class libraries that form part of the Java core API specifications. So our solution is to identify the transient classes using some extra-language mechanism. In JPS, we do this by listing all these classes in a file and passing this file to the underlying system.

3.2.3 A Design for Storing Transient Objects Persistently

It is possible to design a system that would enable objects of otherwise transient classes to be made persistent. For example, each of these classes could have two additional programmer-defined static methods called *persist* and *unpersist*. The signatures of these methods for a hypothetical class T are shown below. Class pT is a

```
class T {
    ...
    static pT persist(T obj);
    // Requires: pT is a class with no hidden state, ie, it has no
    //     references to native data structures.
    // Effects: Constructs and returns an object p_obj of class pT such that
    //     unpersist(p_obj) and obj are identical.

    static T unpersist(pT p_obj);
    // Requires: p_obj has no hidden state.
    // Effects: Constructs and returns an object obj of class T such that
    //     if persist(obj_1) and p_obj are identical, then obj and obj_1
    //     are also identical.
}
```

Figure 3-2: Specifications of possible *persist* and *unpersist* methods

non-transient version of class T, that captures both the visible state and hidden state represented by T.

The system automatically calls the *persist* method and stores the returned object in the persistent store instead of the original object. When a client tries to access this object later, the system fetches the pT object and calls the *unpersist* method with it. The *unpersist* method reconstructs the T object including its hidden state by exploiting the knowledge of the application semantics.

While such a mechanism can be made to work, it adds additional complexity to the system, and it is not clear that it will be very useful in practice. For one thing, it might not be possible to implement the persist and unpersist methods for certain classes. Besides, it is not clear why one would want to store objects like *Threads* or *FileDescriptors* persistently. Because of these reasons, we decided not to include this feature in our system.

3.3 Handling Objects of Transient Classes

JPS has a persistence model based on reachability. Consequently, any Java object can be made persistent in JPS by making it reachable from a persistent root. But as discussed in section 3.2, it does not make sense to store transient objects persistently. Therefore, we need a compile-time or runtime mechanism to prevent such objects from being stored persistently. This section discusses ways of addressing this issue.

One possible way to prevent transient objects from being stored persistently is to define a set of rules that would prevent these objects from ever being reachable from a persistent root. These rules would be enforced at compile time. In fact, this is exactly the programming model that was used in the original Thor/Theta system. We will refer to this model as the *static model*. Section 3.3.1 discusses how this model can be extended to a Java-based system like JPS. The static model however turns out to be overly restrictive in case of applications written wholly in Java. Section 3.3.2 then describes the *persistent programming model*, which provides a much more flexible programming environment. This is the model we adopted for JPS.

3.3.1 Static Model

The static model was used in the original version of Thor [LAC+96]. This model was designed to allow code written in a non-*type-safe* languages like C++ to use Thor without compromising the integrity of the database. The basic idea is as follows.

The method code for all objects that are to be stored in the database is written in a type-safe language called Theta. Application code that uses these database objects by calling methods on them is allowed to be written in any unsafe language. The application code refers to the database objects only via opaque handles. Thor runs the unsafe code in a separate protection domain on the client machines.

All the types defined and implemented in Theta form a closed type system. A Theta type can only be a subtype of other Theta types. Any variable used in a Theta class has to be of a Theta type. Furthermore, no type implemented in the application language can be a subtype of a Theta type.

An immediate consequence of using this model is that there is no way for a database object implemented in Theta to contain a pointer to an object implemented in the application language. As a result, since the persistent root itself is a Theta object, and since none of the Theta classes is transient, this model guarantees that none of the objects reachable from a persistent root belong to transient classes.

Problems with the Static Model

The static model is based on a set of static rules that prevent objects of transient classes from ever being made reachable from any persistent root. Though the static model works fine when the database types and the application are written in different languages, it leads to awkward situations when both the database types and the application code are written in Java. This is because there is no natural way to separate the Java type system into two independent sets.

The type system in the static model of Thor/Theta has the following characteristics.

1. The classes implemented in Theta form a closed type system. All object belonging to these classes can be made persistent. We will refer to these classes as persistent classes.

2. The classes implemented in the application language can use the persistent classes, but cannot be subtypes of persistent classes. All these classes are transient classes.

These rules were the result of a design whose goal was to keep objects of persistent and transient classes in separate address spaces and to not let objects of persistent classes have references to objects of transient classes. But extending these rules to a Java-based system directly leads to many problems.

Firstly, a strict interpretation of rule 1 above means that even local variables of methods in persistent classes cannot be of transient types. This rule was enforced in the original Thor system to ensure that transient classes are kept in a separate address space. Clearly, this restriction is unnecessary in our system.

Secondly, a programmer might want to instantiate a transient object of a persistent class, and store references to other transient objects in it. For example, *Vector* would be a persistent class, but one might want to instantiate a *Vector* of *Threads*. There is no good reason to disallow this as long as one does not attempt to store this *Vector* of *Threads* persistently. Some systems like O2 [D^+90] tried to solve this problem by allowing every class to have two versions—a transient version and a persistent version. For example, the transient version of Vector would be used to store transient objects, while the persistent version of Vector would store persistent objects. But by adopting this approach, one would soon end up with two versions of most classes. It is easy to image how awkward it would be to program in such an environment.

In addition, according to the Java language specifications, all Java classes, including transient classes, must be subtypes of *java.lang.Object*. But it is not possible for both persistent and transient classes to have a common root in the type hierarchy if neither persistent classes nor transient classes are allowed to be subtypes of each other.

Allowing persistent classes to be subtypes of transient classes seems to be bad idea since transient classes usually contain a hidden state and a persistent subclass of such a class would also inherit the hidden state. On the other hand, if we allow transient classes to be subtypes of persistent Java classes, then an object whose declared type is a persistent type, but whose real type is a transient type, could be made reachable from a persistent root. Compile-time checking alone cannot detect such an attempt and only runtime checking can prevent this from happening.

One can imagine designing other solutions to address this particular subtyping problem. For example, *Object* can be treated as a transient class even though it has no hidden state. Furthermore, as a special case, persistent classes would be allowed to be subtypes of *Object*. But then, given that Java does not have parametric polymorphism [MBL97], one would never be able to store a *Vector* persistently since the declared type of the elements of a *Vector* is *Object*.

These reasons lead us to the conclusion that runtime checks, to ensure that all objects being made persistent are indeed of persistent classes, are inevitable. And any additional restrictions could disallow otherwise perfectly legal and well-written programs.

The persistent programming language model described below was designed to address the above problems.

3.3.2 Persistent Programming Language Model

In contrast to the static model which forces a rigid separation of the type system, the persistent programming language model imposes no such restrictions on a Java programmer. Any Java class can be implemented using any other Java class, as in normal Java programs. The system only requires that the programmer should not attempt to make objects of transient classes persistent. The system ensures this by performing runtime checks.

Since JPS is a system based on transaction semantics, the commit point serves as a convenient time to do these runtime checks. Just before sending the commit request to the server, the JPS code running on a client machine (in the FE) checks whether all the objects that were newly made reachable from a persistent root by this transaction are instantiations of persistent classes. If not, the transaction is aborted and the system throws a Java runtime exception. For example, if a Vector object reachable from a root contained references to transient objects, this would be detected at commit time and the transaction would be aborted.

Interaction of Persistence Issues with Transaction Semantics

An important question that needs to be answered is how the persistence issues interact with the transaction semantics of JPS. In particular, if a transaction aborts, which are the objects whose states are restored and which are the ones whose states are not restored?

For objects of persistent classes, restoring the object state on an abort seems to be the only reasonable choice. Transient classes are a little complicated. This is because these classes contain hidden data that the JVM does not keep track of. Consequently, the system cannot restore the state of this hidden data. JPS does however restore the JVM-visible part of the data in these objects.

3.4 Persistence of Java Static Variables

Static variables in Java are per-class variables, unlike normal instance variables for which there is a copy of the variable for every object. Since static variables are not part of any object, they would not be stored persistently in the normal process of making all objects reachable from the root persistent.

Some systems including JPS create a special class object for every Java class and treat static variables as part of these class objects. But since these class objects are not exposed as first-class objects in the Java language, these objects will still never be reachable from any persistent root.

One way to handle static variables in a distributed Java system would be to store all the special class objects (and hence all the static variables) persistently, even though they are not reachable from any persistent root. In this case, there would be one copy of each static variable for the entire system.

Another way to handle static variables is to not store them persistently at all and initialize them separately on every client machine every time an application is run.

Whichever of the above options we pick, one can easily imagine usages of static variables where our choice would be the wrong one. We finally decided not to store static variables persistently because a programmer can always change his program slightly to store the information in the static variables persistently, if really he wanted to. This can be accomplished by storing the information meant to be persistent in instance variables of objects.

We do provide transaction semantics to static variables though.

Chapter 4

Implementation

This chapter gives an overview of the implementation of JPS. The full JPS implementation includes all of Thor, which is the result of many years of on-going research. This chapter mainly focuses on the issues relating to the implementation of a Java interface to Thor.

Our basic approach for running Java programs in JPS is as follows. We first translate Java bytecodes into C code that interfaces appropriately with the Thor runtime environment. We then compile the C code into native code. Section 4.1 explains why we choose this approach.

Section 4.2 describes the runtime system of JPS. One of the nice features of the JPS implementation is that for all object operations, there is a clean functional interface between the JPS runtime system and the code produced by the bytecodes-to-C translator. This section also gives a flavor of this interface.

Instead of writing a bytecodes-to-C translator from scratch, we made use of the Toba translator [PTB+97] and modified it for our purpose. Section 4.3 presents an overview of Toba and describes how Toba was modified to work with JPS. It also describes other Java-to-C translators.

Section 4.4 describes some extensions made to the runtime system of the original Thor/Theta system because of the differences between the object models of Java and Theta.

Section 4.5 describes the limitations of our JPS implementation.

4.1 Basic Approach

Building a persistent Java system on top of Thor primarily involves building a runtime system for Java that interfaces appropriately with Thor.

One of the main advantages of using Thor as the base for building a persistent Java system is that Thor achieves good performance in a distributed environment by using a combination of optimizing techniques. Consequently, we wanted to integrate Java into Thor in a way that is compatible with these techniques.

This ruled out the option of using a standard Java interpreter inside Thor because doing that would have meant replacing Thor's runtime system with the runtime system of a standard Java interpreter. One of the things that makes Thor very efficient is that it manages its client cache very effectively when the database does not entirely fit in main memory. Previous experimental studies have shown that the techniques used can improve performance over page-based system by more than an order of magnitude on memory-bound workloads [CALM97]. By replacing Thor's runtime system with the runtime system of a standard Java interpreter, we would instead have to depend on the paging mechanism of underlying operating system to manage the cache.

One possible approach to integrate Java into Thor was to appropriately modify a standard Java interpreter and incorporate it into the client side of Thor. This is similar to the approach taken by the PJama [AJDS96] project. However, since interpreted code runs one to two orders of magnitude slower than compiled code [PTB+97, MMBC97], we abandoned this path.

Another option was to build a Just-In-Time (JIT) compiler [JIT] into Thor's client side. Though this might have been the ideal solution in the long run keeping in mind the portability goals of Java, implementing an efficient JIT compiler inside Thor requires a tremendous amount of work.

The third option was to compile Java into native machine code off-line, perhaps by first converting it into some intermediate language like C. Doing an off-line compilation into native code has the advantage that aggressive compiler optimizations can be performed, unlike in JITs. This is the option we chose for our system.

Why we read in Bytecodes

An interesting characteristic of Java is that the bytecodes contain nearly the same amount of information as the Java source itself. This has even been demonstrated by constructing decompilers that decompile bytecodes and produce a Java source that is similar to the original program [Vli96, For96].

We chose to read in bytecodes rather than the Java source itself mainly because Java programs are usually distributed in the form of bytecodes. Writing a translator for bytecodes is also simpler than writing a translator for the Java source itself. Moreover, bytecodes form a convenient intermediate language and programs written in any source language can be compiled into bytecodes.

Why we produce C Code

The choice of C as the intermediate language has at least two advantages. It makes our system portable. Also, it permits reuse of the extensive compiler technology that is already deployed everywhere.

Instead of writing a bytecodes-to-C translator from scratch, we made use of Toba [PTB+97], an existing Java bytecodes-to-C translator developed at the University of Arizona.

4.2 JPS Runtime System

This section gives a brief overview of the runtime system of JPS. It mainly focuses on the part of the runtime system that is exposed to the Java-to-C translator.

Section 4.2.1 describes the object layout in JPS.

Since JPS is a persistent object system, the runtime system of JPS is different from the runtime system of standard Java implementations in many ways. For example, in a standard JVM, all the objects reside in the program's virtual address space. But in JPS, the persistent object store resides on servers. Objects are automatically fetched from the servers by the underlying runtime system when they are accessed for the first time. Section 4.2.2 describes some of these basic differences.

One of the nice features of the JPS implementation is that for all object operations, there is a clean functional interface between the JPS runtime system and the code produced by the bytecodes-to-C translator. Section 4.2.3 gives a flavor of this interface.

4.2.1 Object Layout

This section gives an overview of the object layout in JPS. Objects in JPS refer to each other via an indirection. This is because JPS manages its own client cache and this indirection is necessary to allow objects to be moved and evicted from the client cache with low overhead.

Every object that is installed in the client cache has an entry in the indirection table. An object's entry consists of a pointer to the dispatch vector of the object's class, a pointer to the fields of the object, and some header information consisting of the object's oref and some garbage collection information. Figure 4-1 provides an illustration.

4.2.2 Differences with a Standard Java Runtime System

Since JPS is a client-server persistent Java system, it has to do some additional book-keeping in the normal process of invoking methods and accessing instance variables. Below, we describe some of these book-keeping overheads.

Swizzling and Checking if an Object Exists in the Client Cache

Since JPS is a persistent object system, all its objects may not reside in the program's virtual address space. Objects are automatically fetched from the servers when they are accessed for the first time.

Whenever an object is accessed in JPS, the system has to first ensure that the reference to the object has been swizzled (that is, it points to a valid entry in the indirection table). Then, the system has to check if the object is present in the client cache. If not, the system has to fetch the object from the server and install it in the cache before proceeding with the object access.

Maintaining Object Usage Statistics

Since JPS maintains its own client cache, it has to be able to identify cold objects that can be evicted from the cache. JPS maintains usage statistics on a per-object basis. Whenever an object is accessed in JPS, these usage statistics have to be updated.

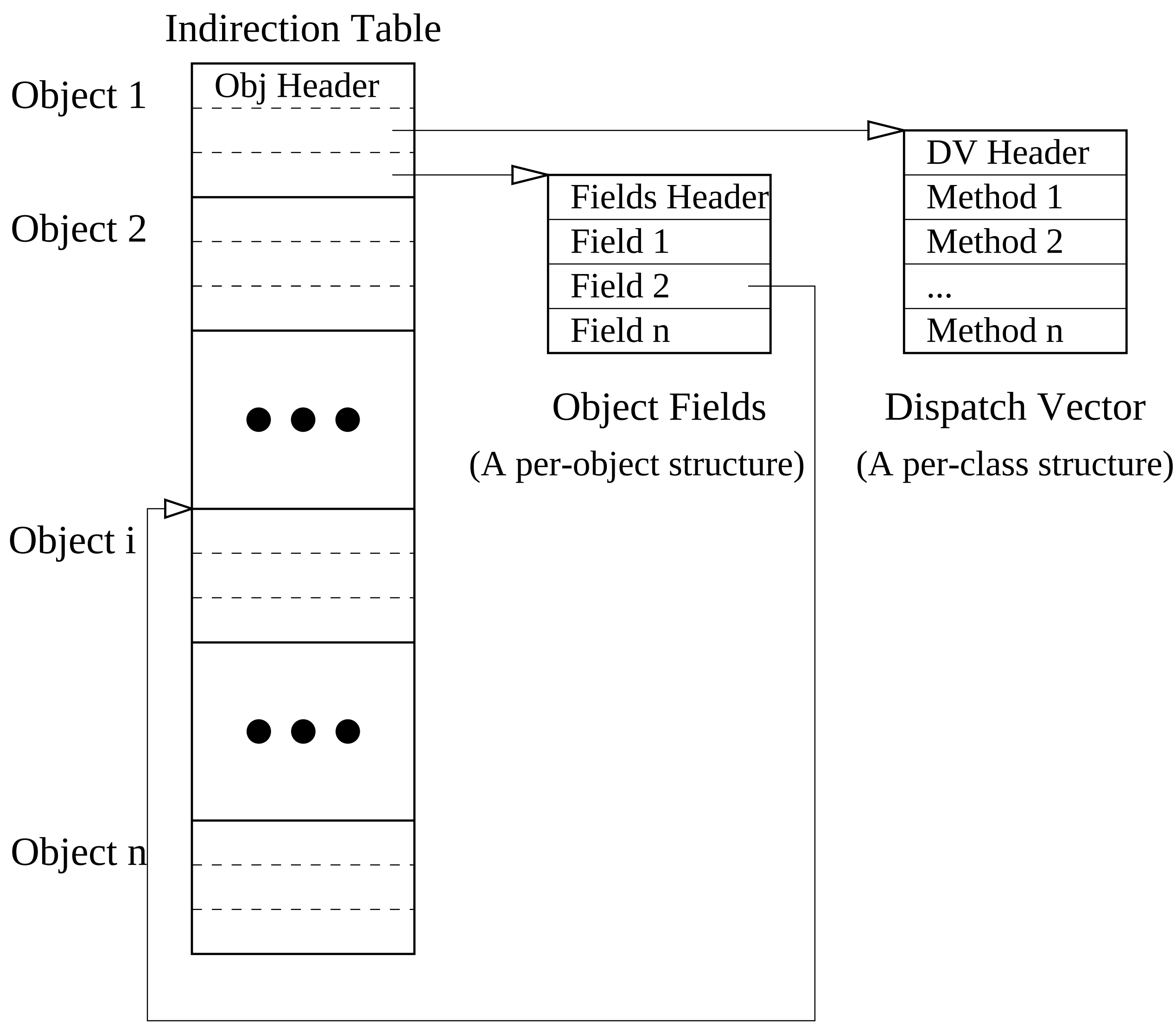

Figure 4-1: Object layout in JPS

Marking Objects as Read or Written for Concurrency Control

Whenever an instance variable of an object is read or written in JPS, the object has to be explicitly marked as read or written. JPS has to maintain the ROS (*read object set*) and MOS (*modified object set*) during a transaction. This information is used at the end of every transaction to determine whether the transaction is serializable with respect to all other committed transactions, and for sending changes made to the objects in the MOS to the server so that they can be recorded in the stable store.

4.2.3 The Translator-Runtime System Interface

JPS provides a clean functional interface between its runtime system and the code produced by the bytecodes-to-C translator. The translator-generated code uses this interface, while the JPS runtime system implements this interface with a set of C macros. Because of this interface, many details of the underlying runtime system are hidden from the generated code. For example, the generated code does not have to know how to check whether an object is resident in the cache or how to fetch the object from the server if it is not resident. All this is handled by the appropriate macros.

Java code	Operation	Generated C code
Vector v;		
v = new Vector();		
	Allocate new object	ALLOC_OBJ(v, Vector); *Call the default constructor*
x = v.elementCount;		
	Mark object as read *Read object field (Assuming v is swizzled, but not necessarily resident in cache)*	MARK_READ(v); GET_VAL_S(x, v, elementCount, Vector);
v.elementAt(i)		
	Invoke virtual method	DISPATCH(v, elementAt, Vector)(v, i)
x = v.elementCount;		
	Mark object as read *Read object field (Assuming v is resident in the cache)*	MARK_READ(v); GET_VAL_R(x, v, elementCount, Vector);

Figure 4-2: Examples illustrating the translator-runtime system interface

Figure 4-2 gives a flavor of this interface. For each object operation, it shows a Java code fragment that requires the operation and the part of the generated code that actually performs the operation. The symbol names in the generated code have been modified for ease of understanding.

The implementation of the GET_VAL_S macro above involves many things. First, the system has to check if the object is resident in the client cache. If not, it has to fetch the object from the server. Then the fields of the object have to be obtained from the object's entry in the indirection table. Finally, the required field of the object is read.

On the other hand, the implementation of the GET_VAL_R macro assumes that the object is resident in the cache. The translator can use this macro in the generated code if it knows that the corresponding object has already been fetched from the server and has been made resident in the client cache.

4.3 Bytecodes-to-C Translation

This section describes the bytecodes-to-C translation process in JPS.

As we mentioned before, we made use of the Toba translator [PTB+97] and modified it for our purpose. Section 4.3.1 presents an overview of Toba and describes how the Toba translator was modified to work with JPS.

There are many other freely available Java-to-C translators today. One of the main reasons we chose to use Toba over the other translators is that the Toba translator is written entirely in Java and its code is easy to understand and modify. Section 4.3.2 gives an overview of some of the other translators.

4.3.1 Toba

Toba [PTB+97] is a system for running Java programs. Like JPS, Toba first translates Java class files into C code, and then compiles C into machine code. The resulting object files are linked with the Toba runtime system to create traditional executable files.

The Toba system consists of two main components—a bytecodes-to-C translator, and a runtime system.

Since Toba is not a persistent object system, Toba's runtime system is not designed to be used in a system like JPS. Section 4.2.2 gives some reasons why this is so. As a result, we do not use Toba's runtime system in JPS. We only use a modified version of Toba's translator in JPS.

Toba Translator

The Java Virtual Machine (JVM) [LY97] defines a stack-based virtual machine that executes Java class files. Each Java class compiles into a separate class file containing information describing the class's inheritance, fields, methods, etc., as well as nearly all of the compile-time type information. The Java bytecodes form the machine's instruction set, and include simple arithmetic and control-flow operators along with instructions for manipulating objects like those for accessing static and instance variables, and those for invoking static, virtual, non-virtual and interface methods. The JVM also includes an exception mechanism for handling abnormal conditions that arise during execution.

Toba translates one class file at a time into a C file and a header file. To translate a class file, Toba requires the class files for all of the class's supertypes. To compile a class's resulting C file, header files are necessary from itself, its supertypes, and all imported classes.

To suppress one of the main sources of inefficiencies in Java bytecodes, Toba statically evaluates the stack by abstractly interpreting the bytecodes and replaces stack management with variables. Figure 4-3 shows an example. $i1$ and $i2$ refer to the first and second elements of the stack, while $iv1$, $iv2$, and $iv3$ refer to the first three JVM local variables.

Java code	**Bytecodes**	**Generated C code**
$a = b + c;$	iload_2	$i1 = iv2;$
	iload_3	$i2 = iv3;$
	iadd	$i1 = i1 + i2;$
	istore_1	$iv1 = i1;$

Figure 4-3: Replacing stack management with local variables

The C code produced by Toba is not very optimized. For example, as one can see in figure 4-3, Toba does not do copy propagation. The generated C code in the figure can be replaced by an optimized version that just does $iv1 = iv2 + iv3$. Toba assumes the existence of optimizing C compilers to do these optimizations.

Adopting the Toba Translator to work with JPS

The Toba translator can be thought of as consisting of two main components—a bytecodes parser, and a code generator. The code generator in turn is made up of two components—one that generates code for the simple arithmetic and control-flow operations, and the other that generates code for operations that manipulate objects like those invoking methods or accessing instance variables.

We used Toba's bytecodes parser and the code generator for simple operations almost unchanged in JPS.

The code generated for object operations has to be compatible with the underlying runtime system. Since the runtime system of JPS is different from the runtime system of Toba, we had to rewrite the portions of the Toba translator that generated code for object operations.

4.3.2 Other Java-to-C Translators

There are several systems available today that translate Java code to C code like Toba. Below, we discuss a few of the more prominent ones.

J2C

Like Toba, J2C [And] also translates Java bytecodes into C, which can then be compiled to native code. J2C is written in C and has absolutely no documentation; hence we decided not to use it for our purpose. J2C also has some limitations—for example, it does not support exceptions fully.

Jolt

Jolt [Sri] is another system that converts Java bytecodes to C code. But unlike Toba and J2C, it uses a very different approach.

The basic idea is as follows. The translator reads a Java class file and generates equivalent C code for methods in the Java class file. It also generates a new Java class file that marks all the converted code as native. Thus, Jolt still uses the standard Java interpreter, but runs native code through it. But since we wanted to use our own runtime system inside JPS instead of a standard JVM's runtime system, we decided not to use Jolt in JPS.

The Jolt approach also has many drawbacks. For example, overloaded methods cannot be translated. The native code interface, in its current implementation, does not handle methods that have the same name since the dynamic linker maps method names in class files to symbol names in shared libraries without using the argument information.

In addition, instance and class initializers apparently cannot be native methods (at least in the Sun implementation), so that is another set of methods that are not translated.

Harissa

Harissa [MMBC97] is another compiler that translates Java bytecodes to C. Harissa incorporates many the optimizations techniques to enhance performance. But it is distributed only in binary format. Hence we could not use it in our system.

JCC

JCC [Sha] is another Java-to-C converter. But unlike other systems described so far, JCC processes Java source code directly to generate C code. Also, JCC is available only for WIN32 platforms.

4.4 Extensions made to the Thor/Theta Runtime System

4.4.1 Interface Methods

In the original Thor/Theta system, any object with multiple supertypes had multiple entries in the indirection table for multiple dispatch vectors. Figure 4-4 shows an illustration. Every object reference pointed to one of the object's dispatch vectors, which was determined by the declared type of the object. If the object was cast to a different type, the object reference had to be changed to point to a different dispatch vector; the new object reference was computed by adding a compiler-generated offset to the old object reference.

The main advantage of this scheme was that the address of every method was always at a pre-determined offset in the object's current dispatch vector. Hence, invoking an interface method was identical to invoking a virtual method—there was very little overhead [Mye95].

One of the requirements for this scheme to work is that whenever an object is cast to a different type, the object reference had to be changed to point to a different dispatch vector; this is true even if the object is being cast into a type that is known

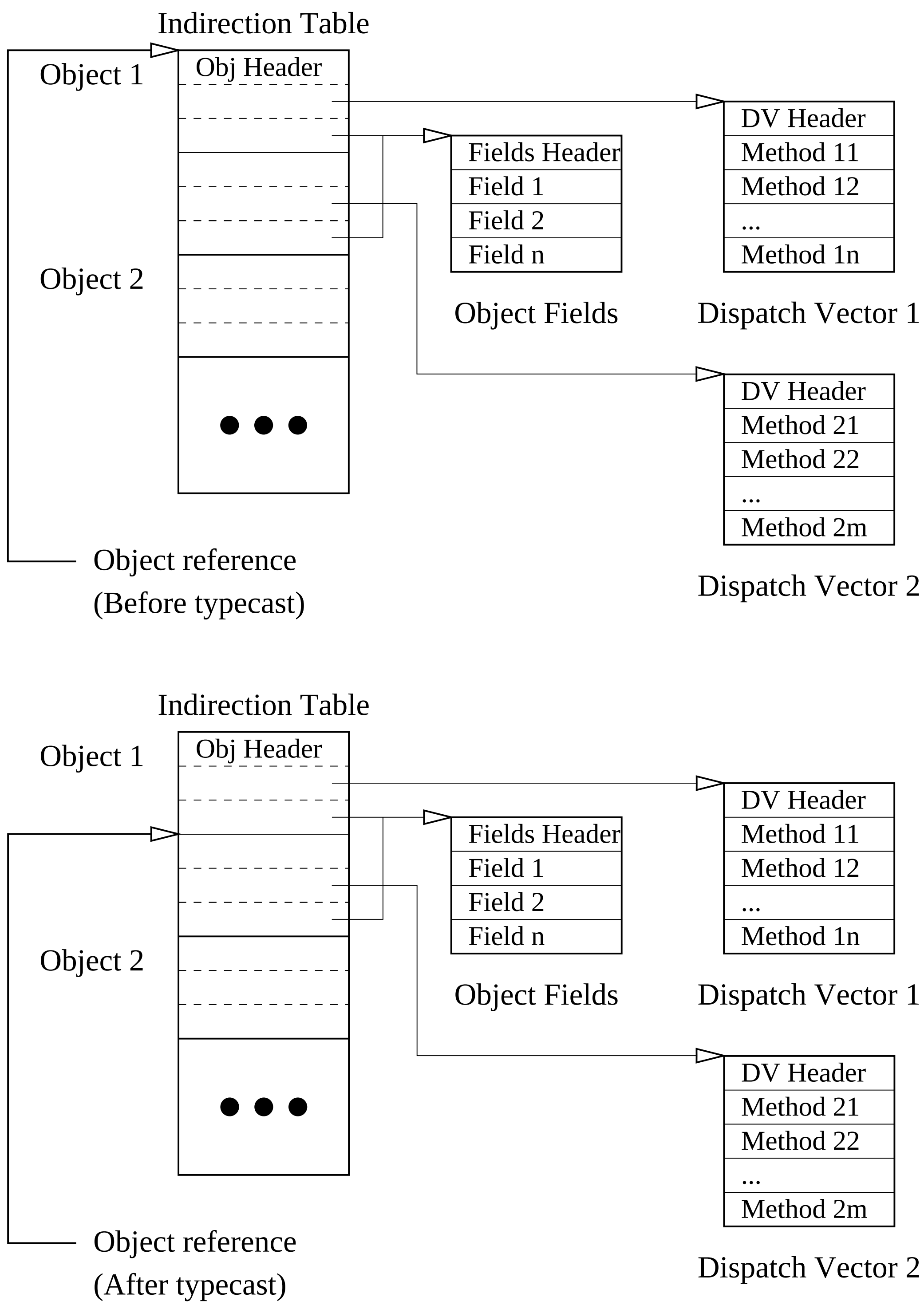

Figure 4-4: Reseting an object reference during a type cast

to be a super-type of the object's current declared type. However, the problem with Java bytecodes is that when an object is cast up in the Java code, this is not explicitly reflected in the bytecodes. Figure 4-5 illustrates this. In the figure Foo is a supertype of Bar. When a variable of type Bar is assigned to a variable of type Foo, there is no explicit type cast in the bytecodes.

Java code	**Bytecodes**	**Generated C code**
interface Foo {...} class Bar implements Foo {...} Foo f; Bar b; ...		
f = b;	aload_2	a1 = av2;
	astore_1	av1 = a1;

Figure 4-5: Casting up is implicit in Java bytecodes

Because of this, we could not use the multiple dispatch vectors scheme in JPS. Instead, we had to employ the technique used by the Toba translator, which involves doing a runtime lookup to find the address of an interface method. Consequently, interface calls (method calls made when the declared type of an object is an interface) are slower in JPS compared to virtual method calls.

4.4.2 Exception Handling

Toba translates Java exceptions into C as follows. For every Java method that might catch an exception, Toba creates a small prologue in the C code that calls *setjmp* to initialize a per-thread *jmpbuf*. The prologue saves the previous jmpbuf in a local structure; the epilogue code restores the old value of the jmpbuf before the function returns.

Toba translates exception throwing into *longjmp* calls that use the jmpbuf. Such calls transfer control to the prologue of the nearest function that might handle the exception. This prologue code simply checks a table to determine if, given the type of the exception and the currently active program counter, whether this method can handle the exception. If so, the execution transfers to the exception handling code. If not, the prologue restores the previous jmpbuf, and executes a longjmp with this jmpbuf.

The main problem with this approach is that setjmp is an expensive operation. For example, on a DEC 3000/400 workstation running Digital Unix 4.0, a setjmp operation saves 84 *long* values into the jmpbuf. As a result, this might tend to discourage programmers from using exceptions.

The exception model in Theta is different from that in Java. All the exceptions in Theta are declared exceptions—a method is required to declare the exceptions it throws in its throws clause. This is unlike the case in Java, where subclasses of *java.lang.RuntimeException* and *java.lang.Error* need not be declared.

The way a Theta compiler translates exceptions into C code is as follows. On encountering an exception, a method sets a per-thread exception flag, stores the exception in a per-thread structure, and returns. On the caller side of the code, the exception flag is checked after returning from every method call. If it is not set, it implies that the method returned normally. If it is set, it implies that the method returned by throwing an exception. (Theta also has a special *failure* exception. It is an undeclared exception and it translated into C using setjmp and longjmp, similar to the way Toba translates all exceptions.)

The advantage with Theta is that the exception flag had to be checked only in case of methods that declared that they could throw exceptions. But since Java has undeclared exceptions, any Java method could potentially throw an exception; hence, adapting this technique to Java means that the exception flag has to be checked upon returning from every method call.

Though the second approach adds a small overhead for *every* method call, we chose to use it in our system. This is primarily because we did not want to discourage the use of exceptions; we strongly believe that the use of exceptions leads to cleaner programs.

4.4.3 Static Variables and Methods

The Theta language did not have a notion of static variables and static methods. JPS handles them in the following way.

JPS creates a special class object for every Java class and treats the static variables of that Java class as instance variables of the class object. Also, all the static methods of the Java class are treated as methods of the class object.

Since these objects are not exposed as first-class objects in the Java language, no other object can contain a reference to them. Hence, to prevent these objects from being garbage-collected, JPS *pins* these objects in the client cache.

JPS does not store these class objects persistently.

4.5 Limitations

This section discusses the limitations of our current JPS implementation.

4.5.1 Threads

JPS does not support multi-threaded Java programs yet. This is because Thor was not designed to allow multiple threads running simultaneously in the FE. However, with some careful re-writing of code, it would be possible to make the FE thread-safe.

4.5.2 Native Code in Class Libraries

There are a number of class libraries that are part of the Java core API specifications. Though most of the methods in these class libraries are implemented in Java itself, there are quite a few native methods in them too. For example, figure 4-6 shows the native methods in *java.lang.Object*.

```
public final native Class getClass();
public native int hashCode();
protected native Object clone() throws CloneNotSupportedException;
public final native void notify();
public final native void notifyAll();
public final native void wait(long timeout) throws InterruptedException;
```

Figure 4-6: Native methods in *java.lang.Object*

Usually, these native methods are implemented by the JVM. But since JPS essentially replaces a JVM, these methods have to be provided by JPS. Unfortunately, there is an ever-growing set of class libraries and Java 1.1.6 already has more than 500 native methods—implementing all of them requires a tremendous amount of mostly routine work. Our current implementation supports only a small subset of the Java class libraries. Chapter 5 describes how this limitation can be overcome.

Chapter 5

Programming Interface

This chapter describes the programming interface that JPS provides to users.

5.1 JPS API

Adding reachability-based persistence to a programming language only requires the following additions to the language API: a function to access the persistent roots, and a function to commit changes to the persistent store. There should also be a directory class to represent the roots. We provide all this functionality in a separate *jps* package.

Accessing Persistent Roots

The persistent store in JPS is partitioned into a number of servers called ORs (*object repositories*). Each OR has a persistent root. An OR can be identified by its unique id. Hence, accessing any persistent root just involves calling the static method *getRoot* with the appropriate OR id. Figure 5-1 shows the specifications of this method.

Committing Transactions

Committing the changes made in the current transaction simply involves calling the static method *commit*. The system keeps track of which objects are read and which objects are modified during the transaction. When the transaction finishes, this information is used to check whether all the reachable objects are of persistent classes, and whether the transaction can be serialized with respect to all other committed transactions. If so, the transaction commits successfully and the modifications to the persistent store are recorded reliably. If not, the transaction aborts, and the JVM-visible state of all objects in the local heap is restored to whatever it was before the beginning of this transaction. JPS also provides an *abort* method to abort the current transaction, which also rolls back the JVM-visible states of all objects.

Figure 5-1 shows the specifications of the *commit* and *abort* methods. JPS automatically starts a new transaction at the end of every transaction.

Adding, Looking-Up and Deleting Objects from a Persistent Root

A persistent root is essentially a directory of objects. Objects stored in such a directory are given *String* names to conveniently identify them. We provide a simple directory abstraction to represent a root directory. Its specifications are shown in figure 5-2.

5.2 Use of RMI in JPS

As discussed in section 4.5.2, we do not support all the class libraries to run inside JPS yet. JPS provides a way to overcome this limitation by the use of Java RMI (*Remote Method Invocation*). Part of the Java code that uses class libraries not supported by JPS can run outside JPS and communicate with the rest of the Java code running inside JPS via RMI. This approach is similar to the original Thor design where part of the client application runs outside Thor and communicates with the Thor code through a *veneer*.

```
package jps;

public class JPS {

// Overview: This class provides procedural abstractions to access
// persistent roots and to commit transactions.

    public native static Directory getRoot(int OR_id) throws NoSuchORException;
    // Effects: Returns the persistent root of the OR with id OR_id.
    //     Throws NoSuchORException if no such OR exists.

    public native static boolean commit() throws TransientObjectException;
    // Modifies: The local object heap and the persistent store.
    // Effects: If any object reachable from a persistent root is
    //         of a transient class, aborts the transaction and throws
    //         TransientObjectException.
    //     Else, if the transaction cannot be serialized with respect to all
    //         other committed transactions, aborts the transaction and
    //         returns false.
    //     Else, commits the transaction and returns true.
    //     In either case, starts a new transaction when the current transaction ends

    public native static boolean abort();
    // Modifies: The local object heap
    // Effects: Aborts the current transaction and starts a new transaction

}

public class NoSuchORException extends RuntimeException {...}

public class TransientObjectException extends RuntimeException {...}
```

Figure 5-1: Specifications of the *jps.JPS* class

```
package jps;

public class Directory {

// Overview: A Directory is a mutable data abstraction that maps
// String names to Objects.

    public Directory();
    // Create an empty Directory

    public void insert(String name, Object obj) throws DuplicateException;
    // Modifies: this
    // Effects: Inserts the object obj and its specified name into this.
    //     If another object with the same name already exists in this,
    //             throws DuplicateException.

    public Object lookup(String name) throws NoSuchElementException;
    // Effects: Returns the object with the specified name in this.  If no
    //     such object is found, throws NoSuchElementException.

    public Object remove(String name) throws NoSuchElementException;
    // Modifies: this
    // Effects: Removes the specified name and its associated object from this.
    //     If no object in this has the specified name, throws NoSuchElementException.

    public Enumeration names();
    // Effects: Returns an Enumeration object that will produce all the
    //             names in this.
    // Requires: this is not modified while the enumerator is in use

    public Enumeration objects();
    // Effects: Returns an Enumeration object that will produce all the
    //             objects in this.
    // Requires: this is not modified while the enumerator is in use

}

public class DuplicateException extends RuntimeException {...}
```

Figure 5-2: Specifications of the *jps.Directory* class

Chapter 6

Performance Evaluation

This chapter describes the performance evaluation of JPS.

The original Thor/Theta system has been extensively compared to other OODBs and is shown to consistently outperform other systems—sometimes by more than an order of magnitude—under a wide variety of commonly used workloads [LAC+96, CALM97, AGLM95, Gru97, Ghe95]. In this chapter, we will compare JPS with the Thor/Theta system. We will show that JPS performs almost as well as the Theta-based system.

The rest of this chapter is organized as follows. Section 6.1 describes our experimental setup. Section 6.2 describes the OO7 benchmark [CDN93], which is the workload we use in our experiments. This benchmark is intended to match the characteristics of many different CAD/CAM/CASE applications, but it does not model any specific application. Section 6.3 presents the results of our experiments that measure the performance of JPS relative to the performance of the Thor/Theta system.

6.1 Experimental Setup

The experiments ran both the servers and the clients on a DEC 3000/400 Alpha workstation, each with a 133 MHz Alpha 21064 processor, 160 MB of memory and OSF/1 version 4.0B. The machines were connected by a 10Mbit/s switched Ethernet and had DEC LANCE Ethernet interfaces. The switch was a DEC EtherWORKS Switch 8T/TX. The database was stored by a server on a Seagate ST-32171N disk with a peak transfer rate of 15.2 MB/s, an average seek time of 9.4 ms, and an average rotational latency of 4.17 ms [Sea97].

Both the generated C code and the system C code were compiled using GNU's gcc with the -O3 option. To reduce noise caused by conflict misses, we used *cord* and *ftoc*, two OSF/1 utilities that reorder procedures in an executable to reduce conflict misses.

In all our experiments, the database was accessed by a single client. The client and the server were both run with 64 MB cache, which is sufficient to hold the entire database.

6.2 OO7 Benchmark

In this section, we describe the OO7 benchmark [CDN93] on which our workloads are based. OO7 is intended to match the characteristics of many different CAD/CAM/-CASE applications, but does not model any specific application.

The OO7 database contains a tree of *assembly* objects, with leaves pointing to three *composite parts* chosen randomly from among 500 such objects. Each composite part contains a graph of *atomic parts* linked by *connection* objects. Each atomic part has 3 outgoing connections. The *small* database has 20 atomic parts per composite part, while the *medium* has 200. In our implementation, the small database takes up 4.2 MB and the medium database takes up 37.8 MB.

Traversals

Our experiments ran several database traversals that the OO7 benchmark defines. These traversals perform a depth-first traversal of the assembly tree and execute an operation on the composite parts referenced by the leaves of this tree.

Traversals T1 and T6 are read-only. T1 performs a depth-first traversal of the entire composite part graph, while T6 reads only its *root atomic part*. Traversals T2a and T2b are identical to T1 except that T2a modifies the root atomic part of the graph, while T2b modifies all the atomic parts. Note that T6 accesses many fewer objects than the other traversals.

6.3 Comparisions with the Thor/Theta System

6.3.1 Overheads in JPS

Section 4.4 discussed some of the overheads introduced in JPS over the original Thor/Theta system. In this section, we discuss additional overheads introduced because of using Java bytecodes.

Structured Loops

As mentioned in section 4.3.1, the C code produced by JPS is not very optimized. JPS assumes the existence of optimizing C compilers to do the optimizations. This assumption mostly holds true, though some simple tests performed by us revealed some problems. When converting from Java source code to bytecodes, there is a loss of information concerning structured loops, which are replaced by goto statements. This is reflected in the C code generated by JPS. As a result, certain optimizations like loop-unrolling do not work very well with the generated C code.

Consider the example shown in figure 6-1. We will compare the performance of a structured loop written in C directly, with that of one written in Java that is compiled into bytecodes and then translated into C. Again, the symbol names in the generated code in the example have been edited for ease of understanding.

Original Java/C code	Generated C code
j = 1; k = 2;	L1: i1=j; i2=k; i1=i1+i2; l=i1;
for (i=0; i< 1000; i++) {	i1=l; i2=j; i1=i1+i2; k=i1;
l = j+k;	i += 1;
k = l+j;	L2: i1=i; i2=1000;
}	if (i1 < i2) goto L1;

Figure 6-1: Translating structured loops

In figure 6-2, we give the instruction counts when both the original C code above, and the generated code are compiled and run. We give the data for the cases with and without loop unrolling enabled. The compiler used was DEC's CC on a DEC 3000/400 workstation. Loop unrolling was enabled by using the -O2 option. Only the instructions that are part of the loop are shown.

As one can see, the loop is executed only 250 times in the case of original C code because of loop unrolling, thus reducing the dynamic instruction count by almost four times. But in the case of generated C code, loop unrolling does not have much effect.

Without loop unrolling		With loop unrolling			
Both		**Original C code**		**Generated C code**	
Instruction Counts		Instruction Counts		Instruction Counts	
addl v0, 0x1, v0	1000	addl v0, 0x4, v0	250	addl t0, 0x1, t0	1000
ldq ...	1000	ldq ...	250	ldq ...	1000
addl s0, 0x1, s1	1000	addl s0, 0x7, s1	250	cmplt t0, t1, t2	1000
cmplt v0, t0, t1	1000	cmplt v0, t0, t1	250	addl s0, 0x2, s0	1000
addl s1, 0x1, s0	1000	addl s1, 0x1, s0	250	bne t2, ...	1000
bne t1, ...	1000	bne t1, ...	250		

Figure 6-2: Instruction counts from running structured loops

6.3.2 Results

As discussed in section 6.3.1, there are overheads introduced in JPS over the Thor-/Theta system. In this section, we will try to quantify the overheads by presenting the results of running our experiments.

The only interesting experiments are those that involve hot traversals. This is because, in the case of cold traversals, the execution time is dominated by the cost of fetching the objects from the server. This cost is same for both JPS and the original Thor system, since the code for doing this is the same in both the systems.

Figures 6-3 and 6-4 present the results for a small database, while figures 6-5 and 6-6 present the results for a medium database. The graphs show the elapsed time in running the traversals. As the results indicate, JPS performs between 1.1 and 1.25 times slower than Thor for hot traversals.

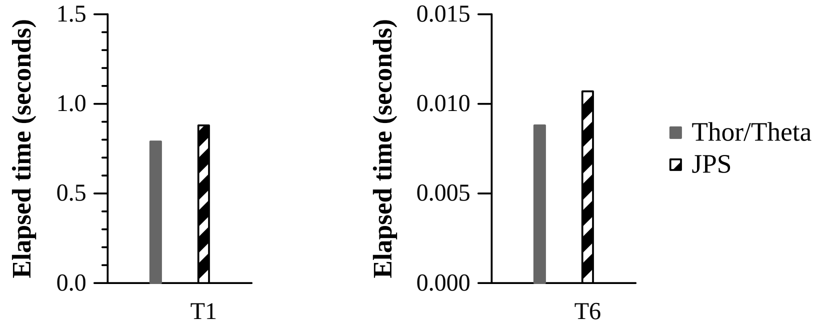

Figure 6-3: Hot traversals, small database

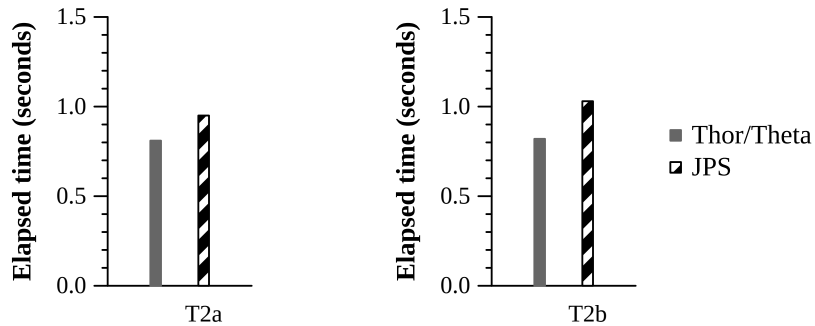

Figure 6-4: Hot traversals, small database

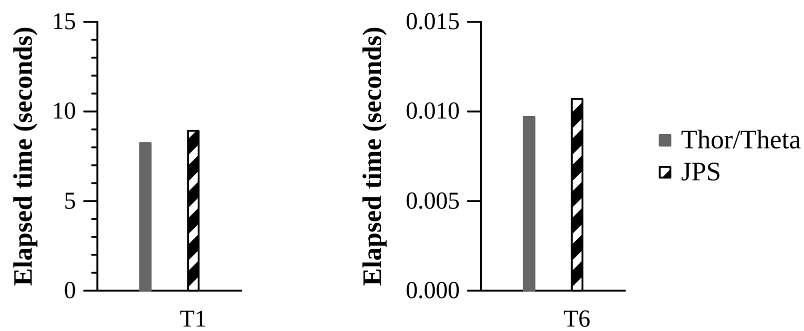

Figure 6-5: Hot traversals, medium database

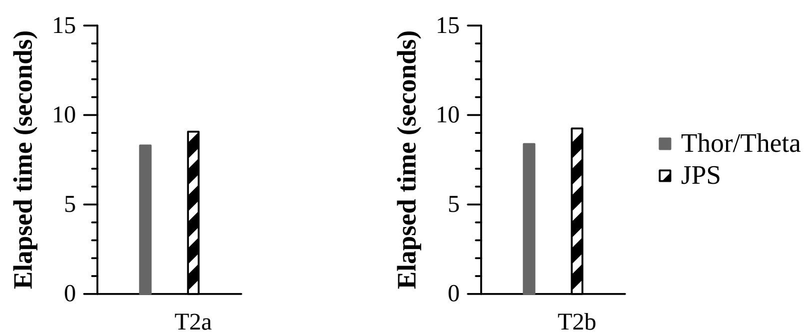

Figure 6-6: Hot traversals, medium database

Chapter 7

Other Persistent Java Systems

Since Java has no persistence model built into it yet, numerous efforts are underway to incorporate persistence into Java. This chapter provides an overview of some of the more prominent projects. For each of the systems, we will discuss their implementation approach, their persistence model, and their support for concurrency control.

7.1 PJama

PJama [AJDS96, ADJ+96] is a persistent Java system being built by people at Sun and the University of Glasgow.

Implementation Approach

PJama uses a modified Java interpreter to run Java programs. Since interpreted code runs much slower than compiled code, PJama's performance will be much worse compared to JPS.

Moreover, PJama is an object-caching system [Ont92, LAC+96]. But as described in section 2.2.4, JPS uses the hybrid adaptive caching technique which has been shown to improve the performance of an OODB significantly.

Persistence Model

PJama supports persistence by reachability. It also supports garbage collection of persistent objects.

The PJama papers do mention the problem of providing automatic persistence to *transient* classes, which is discussed in chapter 3 of this thesis. However, they talk about it only in the context of *Thread* objects. They ignore the fact that there could be other *transient* classes and do not discuss any general solutions for dealing with such classes.

Concurrency Control

PJama has transactions to support atomic updates to the stable store. But it does not support multiple clients accessing the object store simultaneously.

7.2 Persistence Storage Engine for Java

The Persistent Storage Engine for Java (PSEJ) [PSE] is being developed by ObjectStore.

Implementation Approach

PSEJ has a novel implementation. PSEJ runs a post-processor on Java bytecodes. This results in bytecodes that provide persistence and that can be run on any JVM.

Persistence Model

PSEJ also supports persistence by reachability. However, PSEJ does not support garbage collection of persistent objects. Persistent objects are supposed to be explicitly deleted, which can lead to serious problems as discussed in section 3.1.

PSEJ classifies Java classes into three types depending on how they are post-processed: *persistence capable* classes, *persistence aware* classes, and default classes. Persistence capable classes are the ones whose objects can be stored persistently. Persistence aware classes use persistence capable classes, but are themselves not persistence capable. The rest of the classes, which do not need any post-processing, are default classes.

During a transaction commit, all objects of persistence capable classes that are reachable from the root are made persistent. If any object of a class that is not persistence capable is found to be reachable from the root, the system throws an exception. This design is similar to that of JPS.

Concurrency Control

Concurrency control in PSEJ is weak enough to render it essentially useless. One can either lock the entire database, or run with no concurrency control at all.

PSEJ does support atomic transactions for single users. But if a transaction aborts, only the objects that were persistent at the time of the last commit are rolled back. The rest of the objects retain their current state.

7.3 GemStone/J

GemStone/J [Gem] is developed by GemStone.

Implementation Approach

Like PJama, GemStone/J uses a specially modified version of Sun's JVM. It is not clear if their modified JVM supports JIT compilation.

Persistence Model

Like the other persistent Java systems, GemStone/J provides persistence by reachability. It also supports garbage collection of persistent objects.

But it is not clear how GemStone/J deals with objects of Java classes like *Threads* that cannot be made persistent automatically.

Concurrency Control

GemStone/J supports full-fledged concurrency control and transactions. However, unlike JPS which ships objects to client machines, all objects in GemStone/J reside only on their server machines. All the method code is executed only on the server machines. But as discussed in section 2.2.1, the performance of such a system will not scale well if a large number of clients simultaneously try to run methods on database objects.

Chapter 8

Conclusions

In this thesis, we have presented the design, implementation and performance evaluation of a distributed persistent Java system called JPS. This chapter provides a summary of the thesis. It also suggests directions for future work.

8.1 Summary

JPS is a persistent Java system built on top of the Thor object-oriented database (OODB). JPS provides several advantages over other persistent Java systems. Unlike most of other systems, JPS is a distributed system that allows multiple clients to simultaneously access the object store. JPS is built to be used over a wide area network and scales well with large databases. JPS also provides a very reliable and highly available storage. In addition, JPS offers better performance for many important types of workloads.

JPS Interface

Chapter 3 describes the programming model provided by JPS. One of the key issues that arises in designing a persistent Java system is to decide what persistence services should be provided to objects of certain Java classes like threads. It is unclear what it means to store such objects persistently, or what it means to restore them when they are accessed later, potentially by a different application, and potentially on a different client machine. Therefore, we need to either provide an answer to this question, or we need to have a way of preventing such objects from being made persistent. We chose the later approach. However, we provide a programming model that does not impose any unnecessary restrictions on the natural way of writing Java programs. Instead, we do dynamic checking to prevent such objects from being made persistent.

Another interesting issue here is the persistence of static variables. Since these variables are not part of any object, they would not be stored persistently in the normal process of making objects reachable from the root persistent. JPS does not store static variables persistently. It initialize them separately on every client machine every time an application is run.

The complete specifications for the JPS API are presented in chapter 5.

JPS Implementation

Chapter 4 discusses the implementation of JPS. Our basic approach for running Java programs in JPS is as follows. We first translate Java bytecodes into C code that interfaces appropriately with the JPS runtime environment. We then run an optimizing C compiler on the generated C code to produce native machine code.

Instead of writing a bytecodes-to-C translator from scratch, we made use of Toba [PTB+97], an existing Java bytecodes-to-C translator developed at the University of Arizona. We modified the Toba translator appropriately to make it work with our system.

There are a number of differences between the Java object model and the Theta object model. For example, Theta does not have static variables. The exception model in the two languages is different. Because of the differences, the implementation of JPS also required making some changes to the runtime system of the original Thor/Theta system.

Performance Evaluation of JPS

Chapter 6 describes the performance evaluation of JPS. We compare JPS with the original Thor system which has been extensively compared with other systems in literature. We use the OO7 benchmark as our workload. Our results indicate that for hot traversals, JPS performs between 1.1 to 1.25 times slower than the original Thor system. This is due to a number of overheads introduced in JPS because of using Java bytecodes. For cold traversals, the results are almost identical.

8.2 Future Work

8.2.1 Supporting all the Java Class Libraries

There are a number of class libraries that are part of the Java core API specifications. Though most of the methods in these class libraries are implemented in Java itself, there are quite a few native methods in them too. Usually, these native methods are implemented by the JVM. But since JPS essentially replaces a JVM, these methods have to be provided by JPS.

Unfortunately, there is an ever-growing set of class libraries and Java 1.1.6 already has more than 500 native methods. Our current implementation supports only a small subset of the Java class libraries. Supporting all the Java class libraries inside JPS is left for future work.

8.2.2 Dynamic Class Loading

Currently JPS does not store code along with objects in the repository. Also, JPS assumes a complete knowledge of the code for all the types of objects stored in the repository. The JPS executable is created by linking the code for all such types of objects into a single executable.

But in realistic situations, new types would be defined and added to the repository over time. When the JPS code running on a client machine encounters such an object, it should be able to fetch the code for that object and dynamically load it and execute it. It will be useful to extend JPS to support this feature.

8.2.3 Translator Optimizations

The current bytecodes-to-C translator in JPS does a straight-forward translation of bytecodes. However, it is possible to improve the translator to generate more optimized code. Vortex [DGC96] and Harissa [MMBC97] papers describe many such optimizations. These optimizations can be incorporated into JPS. It would be especially interesting to study how these optimizations apply in context of persistent programming systems.

Bibliography

[ADJ+96] M. P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent java. In *ACM SiGMOD Record*, pages 68–75, Dec 1996. Available at http://www.sunlabs.com/research/forest/COM.Sun-.Labs.Forest.doc.external_www.papers.html.

[AGLM95] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *ACM SIGMOD Int. Conf. on Management of Data*, 1995. Available at ftp://ftp.pmg.lcs.mit.edu/pub/thor/occ.ps.

[AJDS96] M. P. Atkinson, M. J. Jordan, L. Daynes, and S. Spence. Design issues for Persistent Java: A type-safe, object-oriented, orthogonally persistent system. In *Seventh International Workshop on Persistent Object Systems (POS7)*, Cape May, New Jersey, USA, 1996. Available at http://www.sunlabs.com/research/forest/COM.Sun.Labs.Forest.doc-.external_www.papers.html.

[And] Yukio Andoh. J2C. Available at ftp://ftp.webcity.co.jp/pub/andoh/java.

[BM92] A. L. Brown and R. Morrison. A generic persistent object store. In *Software Engineering Journal 7(2)*, pages 161–168, 1992.

[CALM97] Miguel Castro, Atul Adya, Barbara Liskov, and Andrew C. Myers. HAC: Hybrid adaptive caching for distributed storage systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, Saint Malo, France, 1997. Available at ftp://ftp.pmg.lcs.mit.edu/pub/thor/hac-sosp97.ps.gz.

[CDN93] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 12–21, Washington, DC, 1993. Available at ftp://ftp.cs.wisc.edu/OO7.

[CFZ94] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-grained sharing in a page server OODBMS. In *ACM SIGMOD Int. Conf. on Management of Data*, Minneapolis, MN, May 1994.

[D+90] O Deux et al. The story of O2. *TKDE*, 2(1):91–108, March 1990.

[DGC96] J. Dean, D. Grove, and C. Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of OOPSLA*, San Jose (CA), USA, 1996. Available at www.cs.washington.edu/research/projects/cecil-/www/Papers/papers.html.

[DKL+94] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J. B. Yu. Client-server paradise. In *20th Intl. Conf. on Very Large Data Bases (VLDB)*, Santiago, Chile, 1994.

[DLMM94] M. Day, B. Liskov, U. Maheshwari, and A. C. Myers. References to remote mobile objects in Thor. In *ACM Letters on Programming Languages and Systems (LOPLAS)*, pages 115–126, May 1994.

[For96] D. Ford. Jive: A Java decompiler. Technical report RJ 10022, IBM Research Center, Yorktown Heights, May 1996.

[Gem] GemStone/J. Available at http://www.gemstone.com/products/j/main-.html.

[Ghe95] Sanjay Ghemawat. The modified object buffer: A storage management technique for object-oriented databases. PhD thesis, MIT Lab. for Computer Science, Cambridge, MA, 1995. Available at http://www.research-.digital.com/SRC/personal/Sanjay_Ghemawat/pubs/phd-tr.ps.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996. Available at http://java.sun.com/docs-/books/jls/html/index.html.

[Gru97] Robert Gruber. Optimism vs. Locking: A study of concurrency control for client-server object-oriented databases. PhD thesis, MIT Lab. for Computer Science, Cambridge, MA, 1997. Available at ftp://ftp.pmg-.lcs.mit.edu/pub/gruber/mit-lcs-tr-708.ps.gz.

[JIT] The JIT Compiler Interface Specification. Available at http://java.sun-.com/docs/jit_interface.html.

[KK90] T. Kaehler and G. Krasner. *LOOM – Large Object-Oriented Memory for Smalltalk-80 Systems.* Morgan Kaufmann, 1990.

[LAC+96] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 318–329, Montreal, Canada, 1996. Available at ftp://ftp.pmg.lcs.mit.edu/pub/thor/thor.ps.

[LCD+94] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A. C. Myers. Theta reference manual. Programming Methodology Group Memo 88, MIT Lab. for Computer Science, Cambridge, MA, 1994. Available at http://www.pmg.lcs.mit.edu/papers/thetaref.

[LGG+91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *13th ACM Symposium on Operating System Principles (SOSP)*, pages 226–238, Pacific Grove, CA, October 1991.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. In *Communications of the ACM 34(10)*, October 1991.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997. Available at http://www.javasoft.com/docs-/books/vmspec/html/VMSpecTOC.doc.html.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 132–145, January 1997. Available at ftp://ftp.pmg.lcs.mit.edu/pub/thor/popl97/popl97.html.

[MBMS95] J. C. Mogul, J. F. Barlett, R. N. Mayo, and A. Srivastava. Performance implications of multiple pointer sizes. In *USENIX 1995 Tech. Conf. on UNIX and Advanced Computing Systems*, pages 187–200, New Orleans, LA, 1995.

[MMBC97] Gilles Muller, Barbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Conference on Object-Oriented Technologies and Systems (COOTS)*, 1997. Available at http://www.irisa.fr/compose/harissa-/coots_web.ps.gz.

[Mye95] Andrew C. Myers. Bidirectional object layout for separate compilation. In *Proceedings of OOPSLA*, pages 124–139, 1995. Available at ftp://ftp.pmg.lcs.mit.edu/pub/thor/bidirectional/paper.html.

[Ont92] Ontos Inc., Lowell, MA. Ontos Reference Manual, 1992.

[OVU98] M. Tamer Ozsu, Kaladhar Voruganti, and Ronald C. Unrau. An asynchronous avoidance-based cache consistency algorithm for client caching dbmss. In *24th Intl. Conf. on Very Large Data Bases (VLDB)*, New York, USA, 1998.

[Par98] Arvind Parthasarathi. The NetLog: An efficient, highly available, stable storage abstraction. SM thesis, MIT Lab. for Computer Science, Cambridge, MA, 1998.

[PSE] ObjectStore PSE for Java. Available at http://www.odi.com/products-/psej.html.

[PTB+97] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. Toba: Java for applications, a Way Ahead of Time (WAT) compiler, 1997. Available at http://www.cs.arizona.edu/sumatra-/toba.

[Sea97] Seagate Technology, Inc., 1997. Available at http://www.seagate.com.

[Sha] Nik Shaylor. JCC - a Java to C converter. Available at http://www-.geocities.com/CapeCanaveral/Hangar/4040/jcc.html.

[Sri] K. B. Sriram. Jolt. Available at http://www.sbktech.org/jolt.html.

[TPRZ84] Douglas B. Terry, Mark Painter, David W. Riggle, and Songnian Zhou. The berkeley internet name domain server. In *USENIX Association [and] Software Tools Users Group Summer Conference*, pages 23–31, Salt Lake City, June 1984.

[Vli96] H. P. Van Vliet. Mocha – Java decompiler, 1996. Available at http://-www.oriondevel.com/robraud/classinfo/mocha.html.

[WD92] S. J. White and D. J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *18th Intl. Conf. on Very Large Data Bases (VLDB)*, pages 419–431, Vancouver, British Columbia, Aug 1992.

[WD94] S. J. White and D. J. DeWitt. QuickStore: A high performance mapped object store. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 395–406, Minneapolis, MN, May 1994.