Implementing Orthogonally Persistent Java

Marquez, Blackburn, Mercer, Zigman

William Cook
September 18, 2003

Persistence for Java

- Persistence Identification
- Concurrency Control
- Portability

Persistence Identification

- Persistence by reachability
- Class variables are implicit roots
  - Truly transparent
  - No modification required for persistence
    - (assuming that class variables are used in a way that makes sense with persistence)

```
public class Simple {
        static int count = 0;
        public static void main(String argv[]) {
                System.out.print(count + " ");
                count++;
        }
}
```


Concurrency Control

- Combining
  - PersistenceIndependence
    - Programs look the same whether they manipulate short-term or long-term data (no *transactions* mentioned in program)
  - Strict ACID Transactions
    - Atomicity and Isolation
- Implies: Transaction = Process
  - Reduced scope for concurrency
- Only way out
  - Relax Persistence Independence
  - ACID is too important to give up

Relax Persistence Independence

- Let programs express transactions
- "Chain and Spawn"
  - Allow Commit/Begin checkpoints in program
  - One *active* transaction per process
  - Doesn't seem to support threads
- Multiple transactions in a process?
  - How are they isolated?
  - How are operations associated with transactions?
  - What about "leakage"
    - use of persistent object after transaction commit
  - How are the specified by programmer?

Portability

- Persistence operations
  - faulting: storage -> memory
  - updating: modified memory -> storage
  - tracking: which objects have been modified?
  - transformation: storage / memory format
- Note
  - Locking, durability done by underlying store
- Uniformly applied to all objects
  - Persistence is an aspect
- Novel solution:
  - *Semantic extension*

Semantic Extension

- Technique
  - Modify class at load time
    - Alter method definitions & byte-codes
  - Dynamic code generation & modification
    - Reflection
- Benefits
  - No change to VM
  - Used with any VM
  - No compiler modification
  - Works with any compiled class

Semantic Extension

- Invocation
  - java Class [args]
  - java PersistentClassLoader [pargs] Class [args]
- PersistentClassLoader
  - A wrapper that then loads specified class
  - Persistence store arguments (pargs)
    - specify store location

Semantic Extension III

- What if existing program uses…
- Reflection
  - Modifications to classes may be *visible*
  - Must transform the reflection interfaces
    - simulate the inverse transformation
- Class Loaders
  - When is persistence transform applied?
  - Extend base ClassLoader to apply transform

Implementation

- Read barriers
  - Ensure object is in memory on demand
- Write barriers
  - Ensure updated objects are written out
- Transformation
  - Initializing generic Java objects on load
- Note
  - Persistent store operations
    - First read -> read lock
    - First write -> write lock (can deadlock)

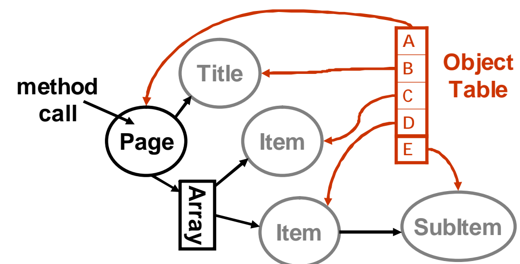
Swizzling

- Swizzling
  - Replacement of a ID by an object
- Strategies
  - Eager swizzling
    - object load à create all related objects
  - Lazy swizzling
    - object load à just hold the IDs of members
    - replace ID with reference on first use
  - No swizzling
    - lookup object every time member is used
  - Eager swizzling to handles
    - object load à create stub objects w/IDs

Are these really impossible?

Unfaulted Objects

- Unfaulted Objects
  - Objects that have been *mentioned* but not *used*
  - Mentioned = referenced by a loaded object
  - Used = method on object called



Shell

- Unfaulted object = empty object
- Semantic extensions
  - Add ID field
  - Add an *updated* field
  - Read barrier before all "getfield" bytecodes
- Problems
  - Consume as much memory as real object
  - Read barriers stay in the code

Façade

- Unfaulted object = stub (façade)
- Semantic extensions
  - For every loaded class C, create:
  - interface to class C_I
  - virtualized form of class C_V *implements* C_I
  - Façade version of class C_F *implements* C_I
    - Stores list of references
    - Fake methods
      - load real object
      - update references
- Benefits
  - Less storage, read barriers removed

Packing and Unpacking

- Implemented in C
  - Slow!
  - C code had to call back to Java
    - JNI is slow
    - Note: fixing this was a key goal of .NET
- Better:
  - At load time generate custom Java code
  - Limit reflection on class structure

Isolation

- Fine-grained locking on objects
  - Very complex to implement
- Instead, simply run isolated
  - Use ClassLoader to isolate transactions
  - Prevents objects from "leaking" out of transactional context

Isolation

- Could also be used within application?

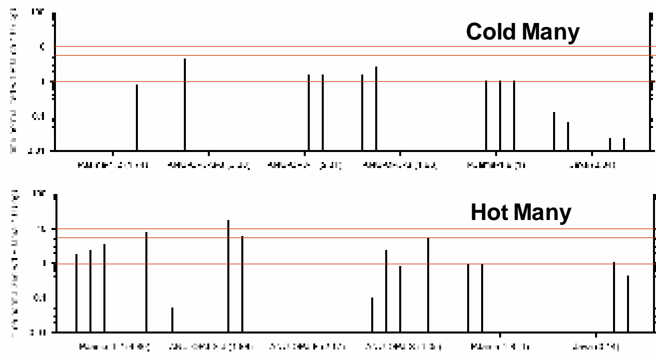
```
void runTransactions()
{
   runTransaction("TransactionClass1");
   runTransaction("TransactionClass2");
}

void runTransaction(string mainClass)
{
   ClassLoader l = new PersistentClassLoader();
   Object t = l.LoadClass(mainClass).newInstance();
   // initialize the transaction here
   new Thread((Runnable) t).start();
}
```


Isolation

- Issues
  - Each transaction must load all objects
  - Updates can only be made to loaded objects
    - No way to update an object without loading it
  - Composition of transactions
    - Define function that calls both, and load that with PersistentClassLoader
- Large read-only data sets
  - Read-only objects are loaded into each transaction
  - E.g. the product catalog in online ordering

007 Performance



Summary

- Orthogonal Persistence Works
  - ACID transactions
    - complete isolation of "programs"
    - Where program is either
      - process
      - subcomputation within a VM
      - can even use threads within a transaction
    - Composition of transactions
  - Persistence by reachability
    - can use static variables as roots
- Problems
  - optimization very hard
    - match performance of relational model?