Lazy Type Changes in Object-oriented Database

Shan Ming Woo and Barbara Liskov

MIT Lab. for Computer Science

December 1999

Background

- Behavior of OODB apps compose of behavior of persistent obj
- Behavior of objects governed their types
- *Type changes* needed to update OODB apps
- *How to execute type change?*

Requirements

- A type change may affect other types
- An *upgrade* consists of a set of re type changes
- Upgrades are ordered
- Execution of an upgrade have to atomic w.r.t. app transactions to prevent type errors

Na ve Execution

- Step 1: shut down the database
- Step 2: transform objects
- Step 3: restart the database
- Drawback: *database availability suffers*
- Not suitable for large databases and mission-critical databases
- Solution: *lazy type changes*

Lazy Type Changes

- Objects transformed lazily, i.e. just before use
- Database availability not affected
- Workload of an upgrade:
  1. Spread effectively over time
  2. Distributed fairly among apps
- Suitable for all databases, including large-scale and mission-critical ones

Theory

- Type change: $c_T = \langle T, T, f^T \rangle$

  Pre-type Post-type Transform function

- Upgrade: $U = \langle n, \{c_i\} \rangle$

  Serial # Set of type changes

Theory (cont.)

- Transform functions:
  - Act on one object at a time
  - Preserve object identities, i.e. object references survive type change
  - Should not modify any data
- Upgrades have to be *complete*, i.e. should not affect other types

Implementation Design (1)

- Based on Thor [ECOOP99]
  - Distributed client/server OODBMS
  - Optimistic concurrency control
  - Servers store objects, validate transactions
  - Clients cache objects, operate on cached objects on behalf of apps
  - Object identity partly location dependent
  - Objects in client cache indexed by a resident object table (ROT)
  - Each ROT entry stores a dispatch vector pointer and a field pointer for object
  - Pointers in *empty* entry are null
  - Pointers in *full* entry are up-to-date

Implementation Design (2)

- Single Server
  - Upgrades stored at the server and pushed to clients
  - Objects transformed by clients before used by apps
  - Objects transformations are regarded as modifications
  - Invariant: *All full ROT entries represent up-to date objects.*
  - At receipt of new upgrade, client
    - aborts running transaction if it used affected objects to guarantee atomicity
    - scans ROT and empties affected entries to preserve invariant

Implementation Design (3)

- Single server continued
  - Special client cooperate with garbage collector to transform rarely used objects
  - Upgrades complete in order: when all objects affected by the oldest upgrade has been transformed, it is discarded
  - Problem: objects change sizes across transformations
  - Size decreases---overwrite original obje
  - Size increases---find space on the same serve page:
    - If succeed, update page offset table
    - Otherwise, write to another page and replace original object with surrogate
  - Reference through surrogate is shortcut when the referring obje is modified

Implementation Design (4)

- Multiple servers
  - A master server stores the master copy of all upgrades
  - Upgrades pushed from master server to other servers
  - Each server sends upgrades to its clients and each client processes upgrades as in the single server implementation
  - Problem: need to maintain consistency when a client talks to multiple servers---an upgrade may arrive at one server before another
  - Client acts as relay to restore consistency: ea commit request is tagge with serial number of newest upgrade at clien

Conclusion

- Lazy type changes preserve database availability and can be efficiently implemented — essential for large-scale mission-critical databases

Future work

- Project focused on persistent objects: how avoid recompiling applications? Universal application framework?
- How to extend to non-database environment