Dynamic Analyses for Understanding and Optimization of Enterprise Java Applications

Ph.D. Dissertation

Aleksandar Pantaleev
The Ohio State University
Outline

- Background and Problem Overview
  - Enterprise Java
  - Motivation and challenges
- Entity Bean ID Identification
  - Match EJB Tier inputs to Entity Bean IDs
- EJBQL Relationship Identification
  - Track query parameters and extract their function
- DTO Identification
  - Track accesses to object state in different EJB tiers
Java EE Tiers
A Complex Cluster

Cluster 1: Web Tier
- Cluster Node 1
  - Physical Machine 2
- Cluster Node 2
  - Physical Machine 3

Cluster 2: EJB Tier
- Cluster Node 1
  - Physical Machine 4
- Cluster Node 2
  - Physical Machine 5

Load Balancer
- Web Browser
- Physical Machine 1
- Database System
- Physical machine 6

- HTTP
- Forwarded HTTP
- Session Replication
- RMI
- JDBC
Challenges

- Horizontal Scalability
  - implemented through clustering services

- Memory Scalability
  - state propagated everywhere
  - memory scalability is non-existent

- Network Scalability
  - maintenance RMI calls typically exceed RMI calls doing "real work"
  - network scalability is bad
Object Lookup Service
Outline

- Background and Problem Overview
  - Enterprise Java
  - Motivation and challenges
- Entity Bean ID Identification
  - Match EJB Tier inputs to Entity Bean IDs
- EJBQL Relationship Identification
  - Track query parameters and extract their function
- DTO Identification
  - Track accesses to object state in different EJB tiers
Typical J2EE Roles

- **J2EE Application Server developer**
  - Concerned with *how*, not *what*
  - Application servers provide services that applications need

- **J2EE application developer**
  - Concerned with *what*, not *how* (ideally, not even that)
  - Enterprise culture states that app developers should be able to focus on their business needs

- **J2EE administrator**
  - Concerned with deploying and integrating apps into servers
  - Configures the general services to work with specific apps
Entity Bean ID Identification

- Leads to Intelligent Proxy Service
- Intelligent proxy works at the object level
- Entity Beans directly represent database data
- Entity Beans have unique IDs
- The proxy needs to see an Entity Bean ID in order to dispatch the request accordingly
Possible Use Cases

- Client directly passes a primitive-type id
- Client directly passes a composite primary key
- Client passes the data that comprises a composite primary key within the same remote call
- Client passes the data that comprises a composite primary key within multiple remote calls
public void addEndEntityProfile(Admin, int, String, EndEntityProfile) {
    LocalRaAdminSessionBean r0;
    int i0;
    boolean $z1;
    i0 := @parameter1;
    $z1 = r0.isFreeEndEntityProfileId(i0);
}

private boolean isFreeEndEntityProfileId(int) {
    LocalRaAdminSessionBean r0;
    int i0;
    EndEntityProfileDataLocalHome $r2;
    Integer $r3;
    r0 := @this;
    i0 := @parameter0;
    $r2 = r0.profiledatahome;
    $r3 = new Integer;
    $r3.<init>(i0);
    $r2.findByPrimaryKey($r3);
}
Implementation

- Code instrumentation of the Jimple representation of the enterprise application
- All direct assignments are instrumented
- Local variables are kept track of based on the stack depth and the variable name
- Static and instance fields are kept track of based on their classname, identity hashcode (if exists), and field name
- Root nodes correspond to parameters input to the EJB Tier
- All nodes except roots have references to their origin
public void addEndEntityProfile(Admin, int, int, String, EndEntityProfile) {
    LocalRaAdminSessionBean r0;
    int i0, i1;
    boolean $z0, $z1;
    i0 := @parameter1;
    i1 := @parameter2;
    $z0 = r0.isFreeEndEntityProfileId(i0);
    $z1 = r0.isFreeEndEntityProfileId(i1);
}
Experimental Study

- EJB Certificate Authority (EJBCA) application
  - Open-source, commercially deployed
  - Comes with its own test suite
  - Multiple tiers: EJB, Web, console client, GUI desktop client
  - A total of 635 classes

- Duke's Bank, Pet Store
  - Smaller applications
Experimental Results

- **Analysis Precision**
  - matched 141 out of 152 `find` invocations (~93%) in EJBCA
    - 3 of the 11 unmatched ones had values originating within the EJB tier
  - 100% in Duke's Bank and Pet Store

- **Analysis Cost**
  - online and offline version
  - EJBCA online: ~279% overhead, offline: ~150%
  - EJBCA offline trace: 183MB, 15m33s to process it
  - Smaller apps online: ~183% Duke's Bank, ~195% Pet Store
Optimizations and Enhancements

- Track PKs passed as members of a complex data structure
  - Java Collection
  - DTO
- Unwrap such a complex data structure and track its parts separately
Outline

- Background and Problem Overview
  - Enterprise Java
  - Motivation and challenges
- Entity Bean ID Identification
  - Match EJB Tier inputs to Entity Bean IDs
- EJBQL Relationship Identification
  - Track query parameters and extract their function
- DTO Identification
  - Track accesses to object state in different EJB tiers
EJBQL Relationship Identification

- EJBQL is J2EE's query language, similar to SQL
- Object lookup service must be aware of query parameters flowing in client requests
- Ideally, it will also be aware of their relationships (to optimize query execution)
- Analysis: upgrade to previous one
  - Additional output
  - Tracking of query return values (Entity beans)
Preprocessing and VFG changes

- Parse application code to identify possible types of query parameters
  - Parsing source is easier than parsing bytecode/Jimple
- Parse queries to identify parameter relationships
  - Most common relationships extracted

- VFG nodes may have more than one origin
- A special type of node holds Entity Bean information (return values from queries)
Implementation and Experiments

- Implemented by instrumenting Jimple
- Experimental setup is the same as before
- Precision:
  - 43 out of 45 (96%) in EJBCA
  - 100% for the smaller apps (7/7 for Duke's Bank, 4/4 for Pet Store)
- Cost: ~293% overhead
- Relationship coverage: 87%
Enhancements

- Track query parameters entering the EJB tier in a larger structure (DTO, Collection)
- Unwrap an Entity bean returned from a query and track its parts separately
- Unwrap Collections of Entity beans returned from queries and track them
- Increase the relationship coverage
Outline

- Background and Problem Overview
  - Enterprise Java
  - Motivation and challenges
- Entity Bean ID Identification
  - Match EJB Tier inputs to Entity Bean IDs
- EJBQL Relationship Identification
  - Track query parameters and extract their function
- DTO Identification
  - Track accesses to object state in different EJB tiers
Data Transfer Objects

- DTO design pattern
  - Reduces the number of remote accesses
  - Often used in Enterprise Java software
- Goal: identify DTOs in an EJB application
  - Find classes whose instances implement the pattern
  - Approach based on dynamic analysis
- Motivation: pre-processing step for the previous analyses
- Published at the Fifth International Workshop on Dynamic Analysis (WODA'07)
DTO Lifecycle
Dynamic Analysis

- Focuses on the state of objects
  - Tracks field reads & writes as opposed to method entries & exits
  - Fields of interesting (Serializable) objects tagged when the objects are created

- Matches state transitions against the DTO lifecycle
  - Requires precise identification of the EJB tier of the initiator of a state transition:
    - Handled by traversing the call stack

- Interested only in application classes
  - Robust with respect to reflection

- JVMTI used with Java 6 JVM (agent written in C)
Experimental Study

- EJB Certificate Authority (EJBCA) application
- Experimental Results
  - 132 interesting (Serializable) clases in EJBCA code
  - 13 deemed DTOs after manual inspection
  - 11 of those 13 were utilized by the application’s test suite (loaded in the JVM)
  - Analysis precision: big picture
    - One false positive
    - One false negative
False Positive and False Negative

- **False positive: not really false**
  - Carries state
  - Problem is, never allows direct access to its state, so technically it is not a DTO

- **False negative: wrapped by another DTO**
  - True DTO
  - All reads & writes of its fields carried out through the wrapper
  - Wrapper classified as DTO, but wrapped object appears to have never moved between tiers
Run-time Performance

- **Start-up overhead**
  - 1m32s without agent
  - 91% overhead with full agent deployed
  - 29% overhead with dummy agent (JVMTI capabilities enabled, but all agent event handlers return immediately)

- **Application overhead**
  - 4m53s without agent
  - 263% overhead with full agent deployed
  - 31% with dummy agent
Conclusions and Future Work

- Three dynamic analyses presented
- All of them contribute information for the use of an object-level lookup service
- In turn, such a service would alleviate the scalability issues inherent in Java EE applications
- Enhancements to the analyses as discussed
- Build the services in the proposed architecture