The same thing we do every night, Pinky - try to take over the world! Attempt 0

Oleg Šelajev

September 11, 2014

Contents

Abstract

Description should be no more than 350 words. Also, don't forget keywords and stuff.

Introduction

Software inevitably evolves during its lifetime. General dynamic system update problem considers ways to update or patch the executing program without interrupting its computation. One of the established ways to do that is to include specific updating code either into the kernel running the application or the application itself.

Updating application code at an arbitrary moment of time is inherently not safe. Applying on-the-fly changes to the running code at the arbitrary time is inherently not safe. Even if the code replacement is instantaneous across the whole program the new program can be not compatible with the existing state of the application. For example, imagine that the new version of the program adds an instance field to an existing class that is present, loaded and instantiated in the running program. The existing objects cannot have any meaningful value in the new field and if the application is not ready to handle default values like null, then the behavior of the updated program, including NullPointerExceptions, is different from running the new version of the program in a new process.

Other types of changes applied at an arbitrary moment without supervision of the system can lead to different erroneous conditions in the application. The common approach to mitigate the problem is to pause the application and apply the changes while nothing is executed in the program. However, it turns out that you cannot just pause the program during executing arbitrary code, so you must include locking mechanisms around certain actions that prevent threads from entering dangerous sections of the code when an update is scheduled.

This approach works, but it introduces a performance degradation due to increased number of locks in the application code. In this paper we are investigating the opposite trade-off. We don't introduce the locks that will herd threads into a safepoint, but we accept that updating application is not immediate and certain.

This paper is structured as follows, in chapter 2 we describe the general architecture of the framework for probabilistic dynamic application updates without safepointing; chapter 3 describes the exact consequences that certain types of changes to the code bring; in chapter 4 we discuss the implementation details of such framework and challenges that occurred during integrating the prototype with the existing "unsafe" updating functionality offered by JRebel. In the final chapter we discuss the results of this experiment, evaluate applicability of this approach in the industrial setting and offer the direction for the further research.

1 Theory and stuff

general architecture, oracle, it's responsibilities, queries, piggybacking on VM pauses, changes => effects => queries structure

2 More stuff

changes => effects => queries list, implementation details.

Dynamic update of an application is aq non-deterministic process, which success heavily depends on the state of the application at the moment of the update. Prior research[1] has identified that certain run-time phenomena might occur during the further run of the updated application. These phenomena could not be the intended behavior programmed into the application itself but rather an artifact of the dynamic update.

Consider the following manually created list of the run-time implications of the updating.

Tabel 3: Run-time Phenomena.

ID	Code change description	Possible run-time phenomenon
7	Class removed	Phantom objects
8	Class renamed	Phantom objects / Lost State
16	Modifier abstract added to class	Phantom objects
6	Class added	Absent state
22	Super class of class changed	Absent state
68/ 71	Instance/static field added to class	Absent state
21	Modifier static removed from inner class	Absent state
70/ 73	Instance/static field type changed in class	Lost state
65	Static initialization impl. changed in class	Oblivious update
30	Constructor impl. changed in class	Oblivious update
114	Static field value changed	Broken assumption
38/ 44	Instance/static method impl. changed (e.g., conditional statement, method split / merged)	Broken assumption / Transient inconsistency

Figure 1: Possible run-time implications of updating a Java program.

Note that these effects were observed during sequential update of the Breakout application through multiple versions that contained non-trivial changes to the application codebase.

[1] `http://www.researchgate.net/publication/220875483_Run-time_phenomena_in_dynamic_software_updating_causes_and_effects`

Tabel 2: Code change analysis of Breakout.

ID	Code change description	R1-R2	R2-R3	R3-R4	R4-R5	%
6	Class added	2	9	3	3	14
30	Constructor implementation changed in class		1	1		2
33	Instance method added to class	3	11	2	8	20
34	Instance method removed from class	2				2
35	Instance method renamed in class	4				3
37	Instance method return type changed in class	4				3
38	Instance method implementation changed in class	12	11	3	10	30
44	Static method implementation changed in class	4				3
68	Instance field added to class		2		2	3
84	Interface added			1		1
120	Resource added [a]	8	1	4	8	18
121	Resource removed [a]				1	1

a. ID 120 and 121 is a refinement of ID 117 'Environment state change' in 0

Figure 2: Change analysis of Breakout updates.

It is important to note that these effects are probable, but not inevitable. Consider the "modifier abstract added to the class" change and its "phantom objects" effect. If the class has not been instantiated or all instances of the class were garbage collected prior to updating, the effect of having phantom objects, which could not be instantiated in the new version of the code, will neither be present in the updated version. On of the challenges of the DSU is to predict and possibly minimize the probability of the effects appearing after the update.

Below is a list of changes to the application level classes that can occur in the JVM setting.

- Changes to method bodies
- Adding/removing Methods
- Adding/removing constructors
- Adding/removing fields
- Adding/removing classes
- Adding/removing annotations
- Changing static field value
- Adding/removing enum values
- Changing interfaces

- Replacing superclass
- Adding/removing implemented interfaces

Our approach to predicting the probability of certain effects being observed after the update is to enumerate the effects a given change can cause and create a list of queries to a central oracle to determine if given change will cause given effect under the current application state.

Below we list queries that the oracle can answer:

- Is class T loaded into the JVM?
- Are there any loaded subclasses of class T?
- Are there currently reachable objects of class T (or any subclass of T)?
- Does an object of class T with a field F instantiated to a non-default value exist?

The exact implementation of such oracle is described later in the paper. For now we provide a list of effects that possible changes are responsible for and a translation of the effects into predicates constructed from the queries listed above.

For example: Change: Instance field F added to the class T with default value V. Effect: Absent state (already instantiated objects will not have the field instantiated) Predicate to determine the observability of the effect:

- Is class T loaded into the JVM and
- Are there currently reachable objects of class T (or any subclass of T)?

If the oracle evaluates the predicate as truthful, the effect will be observed and updating application given current state is not safe. However, it is important to notice, that the current state of the application can change and it might be possible to safely update the application later. For example when all instances of class T are garbage collected and do not pose any threat.

There exist static analysis tools that given the old and new versions of the packaged Java application can list changes that are between these versions.

When our framework receives both old and new versions of the application, it determines the changes between these versions. Then it queries the oracle repeatedly to assess if the current application state does lead to any effects of the changes being observable. If there is no observable effects the update is applied and the new version of the application continues to run. If the update is not happening for a considerable amount of time, currently the updating framework logs the error and stops trying to proceed with the given update. Further development can improve this behavior to allow updates when some of the effects are observable or when given a more complex query system even further.

3 Conclusion and Future Work

How does class file look like? Check out JVMS [1].

References

[1] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java^{TM} Virtual Machine Specification*. Oracle America, Inc., java se 7 edition edition, February 2012.

[2] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

Appendices

Appendix A: A very important messy details

Text of the appendix A.