

Fast Native Collections

Abstract

Software engineers seek a tool that can be run on programs of any size and complexity. A major obstacle for interpreters that can be used to implement a set of dynamic program analyses is the need to model native/system method invocations. The objectives of this proposal are to augment the set of JPF's NativePeers(NP) to handle java programs frequently used in software engineering research, to create NP for common java library classes and to automate NP generation for native methods.

Contact

student: Elena Sherman <esherman "at" cse "dot" unl "dot" edu> (ES)

mentor: Neha Rungta <neharungta "at" gmail "dot" com> (NR)

co-mentor: Peter Mehlitz <pcmehlitz "at" gmail.com> (PM)

Program & Timeline

This project is funded by a NASA Ames Internship. It follows the GSoC timeline (start 05/24, finish 08/16)

5/24 - ES visited Ames, set up the repository on ES's machine, arranged weekly Skype meetings (Thursdays).

Repository

The sources for this project are available from a Mercurial repository on <http://babelfish.arc.nasa.gov/hg/jpf/jpf-fast-native>

Description

Initial description for approaching problem:

Step 1 – learn more by augment the set of JPF's NativePeers(NP) to handle java programs from the Software-artifact Infrastructure Repository (SIR) <http://sir.unl.edu/php/previewfiles.php>

Step 2 - generating fast native collections. Start with trivial ones and moving to cases when a callback is required into JPF code due to collection method being overwritten by the artifact under test. Peter provided me with some insight on the latter case.

Step 3 - automatically detect and generate missing native calls ...

Progress Report

September 24, 2010 - Trying to convert it to a legitimate JPF extension. Started with writing test cases for the fast classes, i.e. ones with the native peers written for them. The test cases only created for the methods that have been rewritten. The classes that tests have been written for are ArrayDeque, ArrayList, ArrayLinkedList. Left: HashMap, PriorityQueue and Vector.

July 22, 2010 - Results so far: Looked through the classes implementing Set and Map interfaces. All classes implementing the Set interface either extend one of the classes that implement Map interface or are not appropriate to have fast-native code. Thus the attention was focused on classes extending Map interface. A small number of methods for the HashMap class have been implemented using fast-native. The limitation is due to the underlying data structure of the HashMap class. For the second part of the project: added nonaXML and sienna applications. Also I started on putting the report for the fast-native collection project. Next: continue on the report and all insights that might be helpful for someone working on similar project. Continue to work on the second part of the project – provide test cases for nanoXML and sienna. Also work on the jtops application to add it to the project together with its test cases.

June 25, 2010 - Results so far: still continue to implement collections classes that implement the List interface. java.util.ArrayList is fully implemented. java.util.LinkedList is almost done. Left: java.util.Vector and java.util.Stack. Next step: continue to work on the List collections and start working on the collection that implement Set interface. Also to begin gradually advance to the third step of the project. Look at SIR programs and determine what kind of system/native calls are not modeled. The idea is to separate calls that do not affect the validation of the system, i.e. does not change the program's state, and calls that do affect the state change. For the former ones a low-cost environment technique can be developed to seamlessly integrate the generation of method stubs needed for system/native calls to run on JPF.

June 17, 2010 - Results so far: the detection if an equals() method is overridden by the class or not is completed. For the ArrayDeque.java containing elements that does not override equals() the collection's methods contains(), removeFirstOccurence() and removeLastOccurence() run at least hundreds time faster with the Native peers than without them. The next step is to implement fast collections for other classes. Right now for 4 concrete classes that extend List interface: java.util.ArrayList, java.util.LinkedList, java.util.Vector, java.util.Stack.

June 10, 2010 - Results so far: all methods that are possible and make sense to implement are implemented for ArrayDeque.java. The table below summarizes the results for a string collection with 30K elements.

The model class had to be reinstalled back for ArrayDeque.java. This concrete collection class extends AbstractCollection class that implements removeAll and retainAll methods and ArrayDeque class does not overrides them. But these two methods are good candidates for having native peers because they iterate over the elements of the collection. However AbstractCollection uses iterators to go over the elements of the receiver object. More likely creating native peers for AbstractCollection that call iterator.hasNext() and iterator.next() would be very costly. Thus we created a model for ArrayDeque.java by copying its original implementation and adding @Override for removeAll(Collection c) and retainAll(Collection c).

From the table below it can be seen that if no call-back is required, as in the clean() method, then the speed-up is an order of magnitude. Even though the native peers with call backs are faster than their regular JPF execution their speedup are not that impressive. The majority of call backs are performed on o.equals calls. If we can detect that methods like equals()/hashCode()/compareTo() are not overwritten then we can use the MJEnv information to obtain the returned value of those methods directly instead of pushing it back to JPF. Thus the next step in the project is to find way to detect if a method has been overridden or not.

method name	time regular	time native peer	speedup = (time_reg - time_native)/time_reg	comments
doubleCapacity()	93,155	93,125	0%	there are no loops in this method but it calls twice System.arraycopy. Our intuition was that a call to one native peer of doubleCapacity should be less expensive than two calls to System.arraycopy. The calls to this method are done infrequently - only 20 times for 30K elements. More likely the gain is lost in the noise.
contains(Object o1)	245	188	23%	worst case, makes a call to o.equals(o1) on each iteration
removeFirstOccurence(Object o1)	234	177	24%	worst case, makes a call to o.equals(o1) on each iteration
removeLastOccurence(Object o1)	245	182	26%	worst case, makes a call to o.equals(o1) on each iteration
removeAll(Collection c)	1266	683	46%	c has 2 elements
retainAll(Collection c)	3450	1302	60%	c has 5 elements
clear()	120	11	91%	no call backs!

June 03, 2010 - Results so far: there will be no model classes for collection. Even though model classes guarantee no changes to the field names of a classes creating them for all collection classes is not practical since the whole class hierarchy for collections should be considered. Preliminary results show that implementing fast native collection reduces the simulation time. For example implementing clear() method for java.util.ArrayDeque reduced execution time from 47 ms to 16 ms - by 2/3 for ArrayDeque size of 10,000 elements. Next steps: continue to collect collection methods that would benefit from Native Peers and start looking into call backs examples as in gov.nasa.jpf.test.basic.JPF_gov_nasa_jpf_test_basic_MJI.

Project Blog

(most recent on top)

2010-05-21 (ES) - updated progress report

2010-06-13 (ES) - updated progress report

2010-06-03 (ES) - updated progress report and the project link

2010-05-27 (ES) - added timeline

2010-05-08 (ES) - abstract added

2010-05-07 (NR) - project page created