Ditto

AJ Shankar and Ras Bodik

What is Ditto?
Ditto is an automatic optimizer for run-time data structure invariant checks written in Java. It speeds up most checks asymptotically in the size of the data structure they're checking.

Why the name "Ditto"?
"Ditto", or ", is a shorthard way of repeating something that's already been said. The Ditto optimizer uses the technique of *incrementalization* to speed up checks drastically by reusing the results of computations that have already occurred.

What's a short example of what Ditto can do?
Imagine you're using a list data structure in such a way that you'd like its elements to be ordered. Since various bits of your code add and remove from the list, maintaining this invariant might be tricky, and so you decide to write a runtime invariant check:

```
boolean isOrdered(List l) {
  if (l == null || l.next == null)
    return true;
  if (l.value >= l.next.value)
    return false;
  return isOrdered(l.next);
}
```

This check defintely does the job, but slows down your program immensely, as it must be run over the entire list every time a change is made. Ditto will speed the check up by a factor of 10 or more.

How does it work?
Please read the paper, or for a more succinct and graphical explanation, view the Powerpoint slides. Here is the abstract for the paper:

> We present Ditto, an automatic incrementalizer for dynamic, side-effect-free data structure invariant checks. Incrementalization speeds up the execution of a check by reusing its previous executions, checking the invariant anew only on the changed parts of the data structure. Ditto exploits properties specific to the domain of invariant checks to automate and simplify the process without restricting what mutations the program can perform. Our incrementalizer works for modern imperative languages such as Java and C#. It can incrementalize, for example, verification of red-black tree properties and the consistency of the hash code in a hash table bucket. Our source-to-source implementation for Java is automatic, portable, and efficient. Ditto provides speedups on data structures with as few as 100 elements; on larger data structures, its speedups are characteristic of non-automatic incrementalizers: roughly 5-fold at 5,000 elements, and growing linearly with data structure size.

How is it used?
Ditto is implemented as a bytocode transformation and an accompanying runtime library,

so it can be integrated into a standard Java build process. Since it operates on bytecodes, it is JVM-independent.

Is the code available?
Yes. The latest version is Ditto 1.0. Here's how to use it:

1. Make sure you have Java 1.5.
2. Download ditto.jar, or compile it from the source.
3. If you do not have them already, download the following libraries: Javassist, GNU Trove, and Apache Commons Collections.
4. Set your classpath to include all of the above libraries.
5. To make sure that Ditto is working, you may want to run it on some sample programs. Download the source to some here and unpack. Then compile them using `javac` (using your new classpath). The samples are, in order, an ordered linked list, a binary tree, an associative list, a red-black tree, and a hash table. Examine TestDriver.java to see how the tests are run. Invoke `java TestDriver.main 3200 3` to run the benchmark on a red-black tree of size 3200, etc. Note the total time and invcount (number of times the recursive invariant was invoked).
6. To use Ditto, follow this two-step process.
    1. First, run `java incrementalizer.Transform arg1 arg2`, where arg1 is a list of invariant functions to optimize, and arg2 is a list of classes to write barrier. For instance, to transform the examples, run `java incrementalizer.Transform TestDriver_invariants.txt TestDriver_classes.txt`. Ditto transforms the classes in place and overwrites their class files. To get a detailed view of the transformation, add the flag `--verbose` to the end of the argument list.
    2. Then, run the program as you would normally, with your new classpath. It should run faster. For instance, at 3200 nodes, the incremenalized version of the RB tree should be about 8x faster.
7. Feel free to modify the source. Bug fixes are definitely appreciated

What are the current limitations of the code?

1. The write barrier list length is fixed at 1000. This is only a problem if you write to the elements of a data structure you're checking more than 1000 times between running its invariants. (This might not be great debugging practice.) If this is a problem, increase the size of the `written` array in runtime.Dispatch and recompile.
2. In the list of invariant functions to incrementalize, if there is an invariant that encompasses multiple functions, you must list all the functions, starting with the entrypoint function first. (This is so Ditto knows what functions to try to optimize, and doesn't bother caching every function call, such as calls to size(), etc.)
3. Currently, Ditto uses a header field to determine what invariants an object is used by. Each invariant has one bit in this field, which is 32 bits long. Thus, only 32 invariants can be run during a program execution. (Note that if there are multiple instantiations of a data structure, that counts as multiple invariants, since each data structure's invariant is being maintained separately.) A temporary fix, if this becomes a problem, is to change the type of the `used` field in runtime.IncObject to `long`, giving you another 32 bits. A long-term solution would probably involve doing something smarter.

I have another question.
If you are trying to find a bug in Ditto, the best place to start is to turn on the DEBUG flag in runtime.Dispatch. This will dump out a wealth of information.

Please email me any further questions. aj at cs dot berkeley dot edu. Thanks!