Compilers description of implementation

-Constant folding

For just constant folding I simply looked for the pattern of load constant load constant binary operation or load constant unary operation, where load constant could be a LDC, a LDC2_W, or ConstantPushInstruction. When I find this sequence, I then call a method passing it those instruction handles and letting it get the values from the load constant instruction and getting the instruction from the arithmetic operation and performing the operation on those values, based on the java bytecode documentation, and then inserting the resulting instruction back into the instruction list and deleting the previous instructions (obviously when doing this we have to repoint the branch instructions which pointed to the removed instructions to the newly inserted instruction and the same for exception handlers for the removed instructions). This was performed repeatedly until no more optimisations were made (I set a boolean to false at the start of every pass, and every time an optimisation was made, I set a boolean to true, at the end it would return to the start of the while loop and test if the boolean was true or false), then the code was assumed to be optimised.

There were only a few cases that weren't covered by this approach, namely branch instructions. The integer comparison operator is a branch instruction (IFEQ ,IFGE, IFGT, IFLE,IFLT,IFNE,IF_ICMPEQ,IF_ICMPGE,IF_ICMPGT,IF_ICMPLE,IF_ICMPLT,IF_ICMPNE). When you do an integer comparison of the form boolean a = b < c (where b and c are ints) the resulting byte code is apparently something like this (I only tested a few cases):

load value
load value
integer comparison (opposite of what is in the java code) label1
iconst_1
goto label2
label1: iconst_0
label2: something

(For IFEQ and the other zero comparison instructions there's only one load value, but the rest is the same.)

In this case the code would optimise the entire thing into either iconst_1 or iconst_0 by running the integer comparison and seeing if the result was true or not. If true it would return iconst_0 else iconst_1, insert that into the instruction list and then delete all the instructions.

For actual branch statements (if statements) this pattern is not followed, from what I found through testing, the pattern looks something like this:

load
load
integer comparison label1
[code of branch body]
label1: [code following the if statement]

To optimise this, I looked at the result of the integer comparison as before, then if it was true then I added a goto statement back into the code going to the target of the branch statement. Then I delete everything from the start of the if statement to the end, including the goto instruction I inserted. So basically if the if statement in the original java code was going to fail, I would just delete the entire if statement. If the integer comparison in the byte code was false, then I would get rid of the load load integer comparison and just keep the code of the branch body since I know it would get run. The branch body could then be optimised - in my code, when I meet a branch statement (integer comparison) that I couldn't evaluate, I would break the loop and restart,

optimising everything before that again so I could get a load constant load constant compare pattern. This is important because before, I allowed it to continue optimising the code without knowing the result of the branch statement. This caused problems because variables can be reassigned within the branch statement, and if you don't know the result of the comparison you don't know if the branch statement would be run or not so you don't know if the variable reassignment inside the branch body would be run or not, and if you just assume it was going to run then you will end up with the wrong result a lot of the time. So you have to know the result of the integer comparison before optimising the code after the branch instruction. There's another way to do it without repetitively optimising everything before the branch instruction before continuing which is to keep a track of which variables are reassigned within the branch and not optimising those variables in subsequent code, but I think the result is probably the same.

- Random branching!

Let's say we have the following code:

```
ldc 50
ldc 10
ldc 10
iadd
goto label1
ldc 5
ldc 5
label1:
iadd
```

My code would just run through the code optimising stuff as it goes along, so it would optimise ldc 5 ldc 5 iadd to sipush 10, which is obviously incorrect. I have been informed that this particular pattern might not be possible to generate from java source code (in particular, the goto iadd). However it just demonstrates how fragile and brittle my algorithm is when branch instructions are added – when I first wrote my program I did not have this kind of branching in mind and my current implementation still doesn't handle it.

To handle this kind of branching full program simulation would clearly do it. Alternatively we could do something like check if the iadd is a target, and don't optimise it if it is. There are so many possibilities though, full program simulation would be way easier than considering every possible case.

- constant variable folding and dynamic variable folding

I handled constant and dynamic variable folding with one function since constant (non-reassigned) variables is just a special case of dynamic variable folding and can be optimised in the same way. As mentioned above I implemented the optimise-above-code-until-we-get-ldc-ifle mainly to fix the problem of variables getting reassigned within branches.

To keep track of variable values I had a map varmap which mapped variables to their values. It worked like this: At the beginning of the function, varmap is empty, and when it starts optimising the function it will look for the pattern of load constant store, which it will optimise into nothing, and put the value of that load constant into the varmap corresponding to the variable specified in the store instruction. Then when we see a load variable instruction, we look in our varmap to see if it contains the variable specified in the load instruction. If it does we just replace the load variable instruction with an equivalent PUSH constant instruction (my replacement function works like this: insert the instruction we want into the instruction list using instructionlist.insert, then we do instructionlist.delete but in a try catch block so that when it throws a target lost exception we use the Apache-provided exception handling code to set the target of the instruction(s) that pointed to

the deleted instruction handle to the constant push instruction handle that we're replacing it with). Then when we encounter a store instruction that we can't yet optimise because it's not in the pattern ldc store, we delete the key-value pair for that variable in the varmap so that we don't replace future load variable instructions with that variable value in the varmap since the variable for that value has changed and we don't yet know the new value. The algorithm is essentially a evaluate-replace cycle where you evaluate the value of a variable then replace all future occurrences of that variable up to whether it gets reassigned. It works because the variables only stop getting replaced when we see a reassignment, and by that point we will have replaced all occurrences of that variable so the old value is no longer needed in the varmap.

There's a different approach that I thought of to handling variable assignments which is to do a full program simulation rather than recursive folding. With full program simulation you could use a class like ExecutionVisitor to run every instruction and simulate the running stack as well as the know all the values of all the variables at all times, then when you see a store instruction you could just pop off the top value in the stack and you know what value you can use to replace future occurrences of that variable up till its next assignment. The problem with full program simulation is infinite loops, which I also haven't handled in my code because JUNIT will get stuck in an infinite loop testing the code, so it can't test your code if you have an infinite loop basically.

Assuming no infinite loops, I think full program simulation could work. When you get an exception you would just go to the catch block in the original program and continue execution there. Via this method I think you could just run the method until it returns and just pop off the value on the stack and just optimise the method that way. This method seems really hacky but it gives the same result as recursive folding and basically is doing the same thing except in one go. If you want to preserve IO you could just watch out for invokevirtual instructions and just output the top value on the stack (e.g if we invoke java/io/Printstream.println then just print whatever value is on the stack). We'll just keep the IO in order (by using a queue structure, naturally) by just putting the output instructions onto the structure as they come up and lastly the return instruction.

I think the biggest advantage of using full program simulation is for dealing with programs with backward jumps. Full program simulation would always have the correct stack at all times so when we have a backward jump to a previous instruction that examines the stack, the stack may be different to before so now the previously run instructions could do something different. With full program simulation all this would be taken care of, and the same with variable assignments.

In my current implementation I have not handled dup and dup2 but I would handle dup by looking for a previous load constant instruction and then replace dup with that instruction, effectively duplicating that instruction. With dup2 I would look for the two previous load constant instructions and duplicating them.

Another thing is you have to put each delete statement in its own try-catch block, to deal with multiple delete statements you could do a for loop trying to delete each instruction individually. The point is that each try block must contain only one instruction otherwise once one of the instructions throws an exception execution jumps to the catch block and the rest of the statements in the try block aren't executed so you will end up with some errors. I had this problem and spent ages trying to figure out where the bug was.

Note that this algorithm originally assumes that all variables can eventually be folded into a constant, in that case it would work perfectly, folding the first variable into nothing, then replacing all occurrences of that variable with a constant, and repeating this process for all other variables in sequence.

In the case where some variables cannot be folded into constant, my code still works in that as I mentioned elsewhere in this text, if I notice that a store instruction is reached without having a restart flag set to true, that means no optimisation was done prior to reaching the store instruction which means the variable cannot be folded into a constant. Then I just continue without the value

for that variable in the varmap, so that future occurrences of that variable won't be replaced with constants. But if we can fold it again then we will, so if we can fold a future store instruction away then we can continue to replace occurrences of that variable, but if we run another pass through the program it will again get removed upon hitting the first store instruction that we couldn't optimise away. So I guess there could be a problem in that we might restart the for loop without replacing all occurrences of the variable where we "know" the value of that variable and then losing the value of the variable when we reach the initial un-optimisable store instruction. For example, let's say we have this code:

```
int a = 1;
int b = 0;
int c = a / b;
if (a > b){
    c = 5;
    System.out.println(c);
}
if (a > b){
    System.out.println(c);
}
```

The optimised output of my original program would optimise the first System.out.println(c); into print 5, but the second one would not be optimised into print 5 (iload_3 would remain iload_3), even though we "know" that c at the point of the second if statement is 5.

The most obvious way to solve this problem would be to not restart the loop when meeting an if statement but this would meet the problem I mentioned earlier where we wouldn't know the value of a variable since we wouldn't be able to fold out the loop without re-optimising the code before the loop. So we HAVE to restart the optimisation when we meet a loop we haven't folded out.

The second solution I thought of is to keep track of the value of every variable at every point in the program, based on instruction position. The algorithm would work like this (I didn't implement this yet): when you fold a store instruction, mark ALL instructions from that point onwards with the value of that store instruction, until the next store instruction for that variable. This would proceed without restarting so it would go through the entire program marking every instruction handle (I would probably do this using the setAttribute method) with the value of the variable up to the reassignment for that variable. The problem of course is once again we don't know where to stop, if we meet a branch instruction we won't know if the reassignment within the branch block would be executed or not, so we don't know if we should mark the instructions there or not.

The problem I described above arose as a result of deleting the store instructions during optimisation. If we could leave deletion of the store instructions to the very end of optimisation then it would solve the problems I just described. So the new algorithm would work the same as the old except it would not delete the store instructions during the for loop, but once we exit the for loop we would go through the entire instruction list looking for ldc store patterns and delete all of them. This would solve the problem I previously described. Of course, since we're no longer changing the instruction list, we remove the restart=true instruction from that code block. However, because we're still treating a lone store instruction as an un-optimisable store instruction we will shoot ourselves in the foot by putting the value of the variable into varmap, then on the next instruction deleting said variable from varmap, and getting stuck in an infinite loop when we encounter a branch instruction (this issue is not resolved because at this point we are still working under the assumption that all branch instructions can be folded). The solution of course is that when we see a store instruction we look at the previous instruction and see if it's a store constant instruction, if it is then we'll simply assume that the value has been store into varmap and do nothing. Actually that would result in an error when you optimise a load variable instruction into a push instruction and the next instruction is a store instruction, if you assume the value has already been stored into varmap because the previous instruction was a push instruction you'd be wrong because it hasn't.

So you need to store it again, even though this might be redundant in some cases, it is necessary for dealing with the case mentioned above. This is the algorithm used in the current version of my program, and I think it handles variable reassignments correctly in the following cases:

1. We see ldc istore. In this case we just do what we did previously.
2. We optimise something into ldc, then go to the next store instruction. This case is handled by the new program.
3. We see store and the previous instruction is not a load constant. What does this mean? Well, in this case we would break and restart the loop if restart = true, else do nothing. If we break and restart the loop something might change. Otherwise we proceed. If we proceed it means the store could not be optimised so we don't have to do anything.

I couldn't think of any other cases so I'm pretty sure that covers everything.

-Exception handling

I tried several methods to get exception handling to work but did not solve the general case. Both methods involved first getting the exception table by using method.getCode().getExceptionTable(). Then from this exception table I create an exception map mapping the old positions of startPC to handlerPCs. Essentially I create a map linking statements inside the try block to the start of the catch block. In order for this map to work I added attributes to each InstructionHandle, this attribute held the original position of each instruction (handle.addAttribute("origpos", handle.getPosition());). By using this method I was able to later on use this exception map for whatever I wanted. For the first method, when optimising a binary sequence returned an error, I simply deleted the binary sequence and all the instructions up to the start of the catch block for that binary pattern. To do this, in the catch block for the optimising binary patterns function I looked up the "origpos" attribute of the instruction in the binary pattern and look up the exception map for that value. The result from the exception map is the original position of the instruction that I want to go to, so then I look through the instruction handle list for the instruction with that value in its origpos attribute. Once I have found this value I simply use my delete_instructions function to delete all instructions from where the exception was thrown up to that instruction (which is the start of the catch block).

The second method I tried for exception handling was to not try to delete any instructions but to simply just retain the exception table from the source code. This happened to be problematic because when I got the exception table, the catch_type attribute of the CodeException objects that populated the table were integers and not ObjectTypes which is what I need for building a new exception table from the methodGen. So for simple testing purposes I just used a preset ObjectType which was just Exception to build the new exception table. Now the new exception table was built no problem and it was correct and all the previous instructions had the correct handlers. However the problem was that after the catch block the statements after it weren't getting optimised. That problem was caused by the store instruction detection block breaking the for loop, which I fixed by detecting if a modification was made to the instruction list at the time of detection. Note that this detection only happened if the store instruction could not be optimised away (in the pattern ldc istore -> nothing). So an example would be iload_1 istore_2, in this case since iload_1 was not optimised into a load instruction, we can't optimise away the istore_2. At this point the code would previously have set the restart flag (the for loop is placed in a while loop which runs if the restart flag is set to true) to true and broke the for loop so that it would restart and try to fold everything before the store statement. The problem with this is that is if you had code like this:

```
int x = 12345;
int y = 54321;
//System.out.println(x < y);
y = 0;
try{
    int c = x / y;
}
```

```
        catch(ArithmeticException e){
            System.out.println(x > y);
        }
        return x > y;
```

In particular the int c = x / y would have been optimised into:

```
sipush        12345
iconst_0
idiv
istore_3
```

In this case since the sipush iconst idiv could not been optimised into a load constant instruction, we would get stuck trying to optimise away istore forever.

The optimisation would stop at the store instruction for c because we could not evaluate x / y since it would have resulted in an exception (division by zero). The stopgap solution was to get rid of the restart = true statement in the if block that handled store instructions without a load constant preceding. However this caused the rest of the code to not be optimised - optimisation would just halt at the store instruction for c. The solution is to leave the store instruction alone if it was un-optimisable. To be sure that the store instruction was un-optimisable, simply check if restart was true or not. If it was true then some modification to the code happened so we wanted to restart the loop by breaking the loop at that point. On the other hand if restart was false then that means no optimisation was done before we got to the store instruction which means that the code up to the store instruction could not be further optimised and if we restart the loop, again no optimisation would happen, which means that the store instruction was un-optimisable. This allowed us to leave the code in the try block un-optimised whilst optimising all the code that follows, including the catch block and everything else.

The code as it is has a few quirks. One is that the newly inserted instructions do not have handlers. For example the aforementioned load load idiv would get optimised into sipush iconst idiv, but the newly inserted instructions sipush and iconst would not have the "origpos" attribute. I could add it in but I figured that since they only push a constant onto the stack they should not raise any exceptions. That might be wrong and it would be quite easy to add the origpos attribute back onto the new instructions. When doing the load replacement, simply get the origpos attribute from the load instruction and add it back to the inserted constant push instruction that replaces it. Another quirk is that when you use the addExceptionHandler method on the methodGen object it will apparently auto-increment your endPC. For example if you do
methodGen.addExceptionHandler(handle, handle, hand, exceptiontypesmap.get(value));
Even though the two parameters provided as startPC and endPC are the same (handle) the exception table generated will show the startPC as handle and endPC will be handle.getNext(). I don't know if this causes any semantic problems since you might get the catch instruction being its own exception handler- this might lead to an infinite loop. Haven't yet figured out the way to prevent this from happening, though an easy way I can think of is to change the code that creates the exceptionmap. Instead of looking through the list of handles for each instruction handle that has the index in the range start to end, and adding them individually to the exceptionmap, it would probably be better to find the instruction handle whose position is startPC and the instruction handle whose position is endPC, and we add the origpos attribute to these instruction handles. Then when we finish the for loop and it's time to generate the new exception table, we can look through the exception table we got earlier from the method, and for each exception find the instruction handle whose attribute origpos is the the same as the startPC for that exception and the instruction handle whose attribute origpos is the same as the endPC for that exception. Then we feed those two instruction handles as arguments to the addExceptionHandler function.

The above algorithm makes an assumption which is that all the original instruction positions or at least StartPC and EndPC will be preserved in some resulting instruction handle, which may not be

true. For example if the try block did not raise an exception, the code inside would get optimised, then the startPC or endPC may or may not be present in the resulting code depending on how the origpos was added back to the resulting code. If you just add the last instruction's origpos to the resulting instruction you might not end up having the startPC, and vice versa for endPC. The solution is to have the program scan for all instructions with a origpos in the range between startPC and endPC when generating the exception, and take the instruction with the lowest origpos in range to be startPC and the highest to be endPC, for each exception.

Note that the current code does not handle multiple catch blocks of the format try catch catch. If you do multiple catch blocks it will fail - actually what it seems to do is replace the exception table entries for the first catch block with the exception table entries for the second catch block (and probably onwards till the last catch block), instead of preserving them. In order to get it to preserve all the try catch blocks you will probably need to implement the code I mentioned above where you go through the original exception table and simply replace the startPC and endPC and handlerPC with the updated instruction handles. If we can guarantee that every startPC and endPC will be found in the origpos attribute of some instruction handle, then the code should be correct since I'm just simply updating the original exception table with newly updated instructions. Also I have no idea how to get from the int returned by getCatchType() to the ObjectType that is needed by the constructor method for exceptions. The BCEL documentation just says "it points to the exception class which is to be caught." which I have no idea what that is referring to. As for the stackmap frame error problem I used the cgen.setMajor(50) workaround, I didn't generate the proper stackmaptable (which would have been more difficult).

- loops

Loops are not handled in the code because I didn't have enough time to implement loop detection. To be more accurate the current implementation will revert to simple constant folding (i.e no variables are folded) if it encounters a backward jump in the code since that's indicative of a loop, and it doesn't know what to fold so as to not end up with infinite loops. I have thought of an algorithm that will handle loops though I did not have time to implement it. The algorithm is to first go through the code looking for this kind of pattern:

label1: ldc

ldc

int_compare

[loop body here]

iinc

goto label1

Once you find this kind of pattern you know that you have a for loop of type for(int i something; i compare something; i++){ [for loop body] }. Of course there are many different types of for loops and it might not be iinc, it could i = i+2 or i— for example, the comparison operator also might not consist of only two instructions and so on. So more accurately it might look like this:

label1: [do some kind of comparison]

[loop body]

do something to the counter
goto label1

Since loop detection doesn't seem straightforward I would have used a library like Soot for loop detection.

Once the loops are identified I could try to fold things within the loop body. I would add some kind of ignore attribute to the loop comparison and counter modification instructions so they don't get optimised away, whilst continuing to optimise the statements inside the loop body.

If I were to deal with dynamic variable assignments within loops, I would figure out which variables were dynamic and which were not. If a loop body contains a variable assignment, if the assignment expression contains a dynamic variable then I would not optimise it, otherwise I would optimise it. To begin with, the counter variable would be marked as dynamic, and for all variable assignments within loops, the variable itself is marked as dynamic as well. So if inside a loop we had a=a or a=i both times a would be marked as dynamic. But if we had something like a = b where we know that b is not a dynamic variable then it would be fine. If we just had a = 10 then we can optimise it away but the problem with a = b is in determining whether if b is a dynamic variable or not. But then if we had a = b; b = a; then it would get stuck in an infinite loop trying to figure out if b is dynamic and then trying to find out if a is dynamic without ever getting an answer. So we should add in some kind of loop detection whereby we use some kind of array structure to keep track of variables we've already "visited" so we don't get stuck in an infinite loop.