Project 3

Compiler Implemented Fault Tolerance for Faults Originating in Hardware

ECE 466/566 Fall 2024
Due: April 18, 2024
You are encouraged to comment directly on this document!

Objectives

- Implement a code transformation that improves system reliability through code replication.
- Combine static and dynamic analysis to prove properties of code at runtime.
- Gain experience generating phi-nodes and ensuring the consistency of SSA-form.

Description

Soft errors are ***transient*** errors (temporary hardware errors) that arise in hardware due to cosmic ray strikes, device variation, thermal fluctuation, or other phenomena that may result in a random bit flip. Such random bit flips can lead to incorrect calculations, outputs, or other observable program failures. While soft errors have been rare in the past, as transistors continue to scale down, soft error rates are expected to increase, compromising the reliability of computation. Soft errors are already happening in the cloud at rates observable to the hyperscalers.

One way to mitigate faults is by replicating code. In the same way that two engines make an airplane more reliable and less likely to crash should one of the engines fail, we can use replication of code to provide greater reliability. The compiler is an ideal layer at which to deploy code replication since the compiler can do it in an automated manner. This reflects a general trend in compiler research and compiler use in the industry for using compilers to perform automated code transformations for a variety of purposes beyond just performance.

In this project, you will implement an algorithm that provides fault tolerance for soft errors by replicating code and protecting control flow. To prevent soft errors from corrupting data, we can replicate instructions in the program and compare intermediate results to make sure they are correct. If a discrepancy is detected, we presume a soft error is to blame and the program can be gracefully terminated or it can recover to a safe state. Since recovery requires more sophisticated transformations, we will only focus on detecting the errors and terminating the program.

There are two kinds of errors we want to catch:

- We want to detect if the computation of any integer or pointer value is incorrect.
- We want to guarantee that control flow occurs properly (for 566 only).

You may find it interesting to read the paper this assignment is based upon:

G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, “SWIFT: Software Implemented Fault Tolerance,” presented at the CGO '05: Proceedings of the International Symposium on Code Generation and Optimization, 2005.

Replicating Computation

We want to detect as many possible errors as we can through instruction replication. Here is an example of replicating an instruction and comparing that the outputs match:

Before	After

`%3 = trunc i64 %indvars.iv.next17 to i32`	`%3 = trunc i64 %indvars.iv.next17 to i32`
	`%4 = trunc i64 %2 to i32`
	`%ftcmp18 = icmp eq i32 %3, %4`
	`%ftcmp_zext19 = zext i1 %ftcmp18 to i32`
	`call void @assert_ft(i32 %ftcmp_zext19, i32 1)`

Note, the `trunc` instruction is duplicated. Then, we compare the results using `icmp` and forward the value to `assert_ft`, which will abort the program if `icmp eq` evaluates to false.

It’s important to note also that we must fully duplicate dependence chains as well. The trunc instructions get different arguments, and that’s because the instructions that feed the trunc are also duplicated. Here’s the full basic block so that you can see the full replication of dependence chains.

Before replication:

```
for.body:
  %indvars.iv16 = phi i64 [ 0, %entry ], [  %indvars.iv.next17, %for.body ]
  call void @usqrt(i64 %indvars.iv16, %struct.int_sqrt* %q)
  %indvars.iv.next17 = add i64 %indvars.iv16, 2
  %3 = trunc i64 %indvars.iv.next17 to i32
  %cmp = icmp slt i32 %3, 100
  br i1 %cmp, label %for.body, label %for.end
```

After replication:

```
for.body:
  %indvars.iv16 = phi i64 [ 0, %entry ], [ %indvars.iv.next17, %for.body ]
  %1 = phi i64 [ 0, %entry ], [ %2, %for.body ]
  %ftcmp46 = icmp eq i64 %indvars.iv16, %1
  %ftcmp_zext47 = zext i1 %ftcmp46 to i32
  call void @assert_ft(i32 %ftcmp_zext47, i32 15)
  call void @usqrt(i64 %indvars.iv16, %struct.int_sqrt* %q)
  %indvars.iv.next17 = add i64 %indvars.iv16, 2
```

```
%2 = add i64 %1, 2
%ftcmp = icmp eq i64 %indvars.iv.next17, %2
%ftcmp_zext = zext i1 %ftcmp to i32
call void @assert_ft(i32 %ftcmp_zext, i32 0)
%3 = trunc i64 %indvars.iv.next17 to i32
%4 = trunc i64 %2 to i32
%ftcmp18 = icmp eq i32 %3, %4
%ftcmp_zext19 = zext i1 %ftcmp18 to i32
call void @assert_ft(i32 %ftcmp_zext19, i32 1)
%cmp = icmp slt i32 %3, 100
%5 = icmp slt i32 %4, 100
%ftcmp22 = icmp eq i1 %cmp, %5
%ftcmp_zext23 = zext i1 %ftcmp22 to i32
call void @assert_ft(i32 %ftcmp_zext23, i32 3)
br i1 %cmp, label %for.body, label %for.end
```

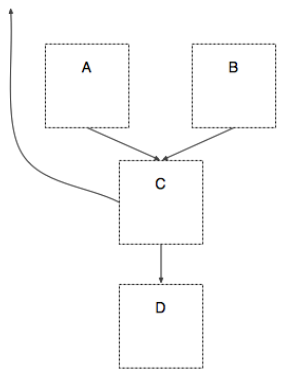
Note, every original statement has a replicated instruction, a comparison that tests if the replicated result is the same as the original, and a call to assert_ft to raise an exception if the first argument is false. The other argument to assert_ft is for debugging purposes only.

However, it is not feasible to replicate all computations. For example, we cannot replicate branches since they update the program counter. Hence, the next section will discuss how to handle control flow. Also, there is no point in replicating stores since there is no way to be sure they stored the same value to memory without replicating the entire memory space, which is too expensive to ever be practical in consumer electronics. By this argument, we also will not replicate allocas. Similarly, we do not replicate call instructions since calls can lead to stores.

For this project, you must implement a transformation that duplicates all eligible instructions, and their dependences, and checks if they compute the correct result.

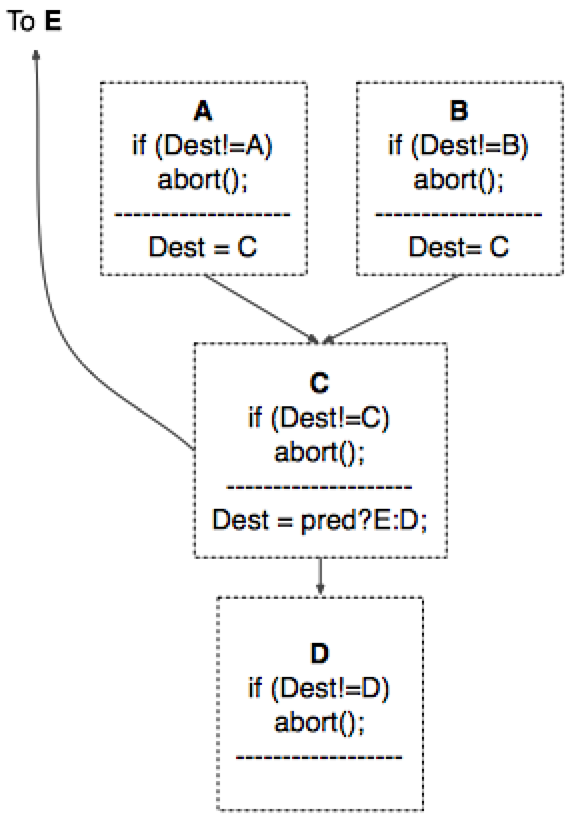
Verifying Control Flow (566 only)

To verify control flow, we insert additional code to validate that control is transferring correctly. For example, consider the following control flow graph:



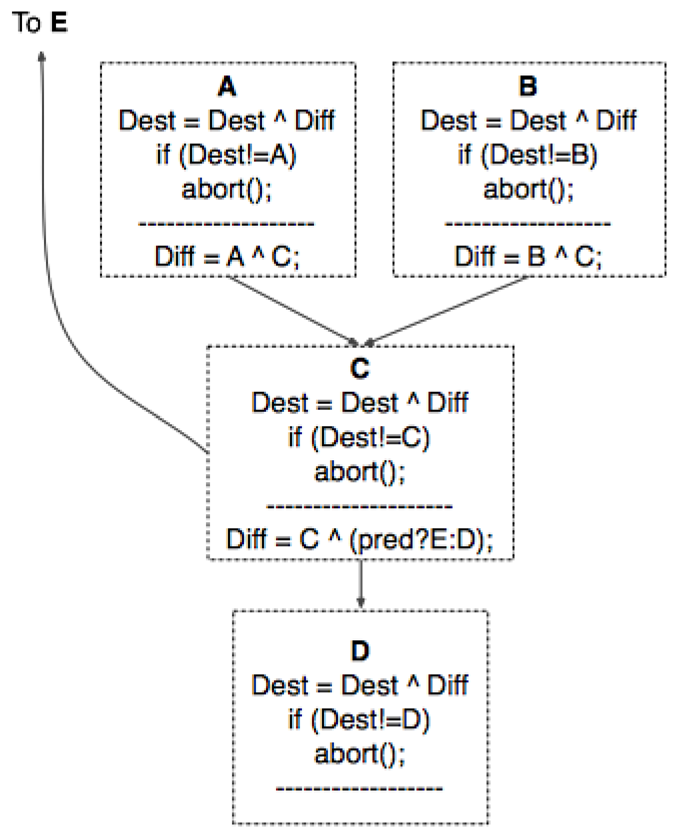
If a soft error occurred on the predicate for a branch, then the branch could go the wrong way. On the other hand, if it occurred in the PC offset, the branch could jump to the wrong instruction. Hence, it's possible that the branch in A could randomly transition to block D or that block C could fall through to D when it was supposed to follow the back edge.

To detect such illegal transitions, we will insert extra code that computes where control is supposed to flow and then verify that the flow was correct after the branch executes. For example, here is an example of the idea, but not the implementation we will use:



This code works if the branch lands on the wrong block, as is the case if the predicate is somehow corrupted. The Dest variable will not match the ID of the block, so it is known that an error occurred while executing the branch.

However, what happens if the branch's offset is corrupted and it jumps to the middle of a block? In that case, we will never know that an error occurred because we do the check at the beginning of the block. We could add another check just before assigning Dest a value, but that just adds overhead. So, instead, we develop a more sophisticated mechanism.



We use an extra masking register, Diff, to store the bitwise difference of the current block's ID and the ID for the block control should transfer to. After jumping to the new block, the mask is used to update the Dest value. If it doesn't match, just as before, we know there is a problem. However, if we jump somewhere in the middle of a block, the Dest value will not be updated until control transfers again, and in this case, the error will be detected.

For this project, you will implement this algorithm.

Here is an example of control flow checking in code with added instructions in bold (this is not the same code as shown before):

```
entry:
  %x = alloca [3 x double], align 16
  %solutions = alloca i32, align 4
  %q = alloca %struct.int_sqrt, align 4
  %arraydecay = getelementptr inbounds [3 x double]* %x, i32 0, i32 0
  call void @SolveCubic(double 1.000000e+00, double -1.050000e+01, double
3.200000e+01, double -3.000000e+01, i32* %solutions, double* %arraydecay)
  %Diff = xor i32 1792, 67335
  br label %for.body

for.body:                                         ; preds = %for.body, %entry
  %indvars.iv16 = phi i64 [ 0, %entry ], [ %indvars.iv.next17, %for.body ]
  %Diff18 = phi i32 [ %Diff, %entry ], [ %Diff20, %for.body ]
  %Dest = phi i32 [ 1792, %entry ], [ %Dest19, %for.body ]
  %Dest19 = xor i32 %Diff18, %Dest
  %cfg_cmp = icmp eq i32 %Dest19, 67335
  %ft = zext i1 %cfg_cmp to i32
  call void @assert_cfg_ft(i32 %ft, i32 %Dest19, i32 0)
  call void @usqrt(i64 %indvars.iv16, %struct.int_sqrt* %q)
  %indvars.iv.next17 = add i64 %indvars.iv16, 2
  %0 = trunc i64 %indvars.iv.next17 to i32
  %cmp = icmp slt i32 %0, 100
  %1 = select i1 %cmp, i32 67335, i32 133633
  %Diff20 = xor i32 67335, %1
  br i1 %cmp, label %for.body, label %for.end

for.end:                                          ; preds = %for.body
  %Diff21 = phi i32 [ %Diff20, %for.body ]
  %Dest22 = phi i32 [ %Dest19, %for.body ]
  %Dest23 = xor i32 %Diff21, %Dest22
  %cfg_cmp24 = icmp eq i32 %Dest23, 133633
  %ft25 = zext i1 %cfg_cmp24 to i32
  call void @assert_cfg_ft(i32 %ft25, i32 %Dest23, i32 1)
```

You might notice that the IDs used in this code are fairly random. That was intentional. You should design a unique ID for each block that is unlikely to occur in the program. For example, it might be something like this:

(Num(BB) << 20) | (sizeof(BB) << 8) | ((Num(BB)*size(BB)) % 37)

where Num(BB) is a unique number that identifies the block, and size(BB) is the number of instructions in the block.

PHI Nodes

When implementing the control flow checking, you may not use loads and stores. Instead, you will need to insert PHINodes to merge values on all incoming edges. You need to develop a strategy to make this easy. Here is a recommendation:

Traverse over all basic blocks:

- Create PHINodes for Dest and Diff, but do not try to supply their operands yet.
- Insert the code to compute the new Dest, its comparison, and call assert_ft_cfg to detect an error.
- Insert the code at the end of the block to compute the new Diff. Note, you need to detect if the branch is conditional, and if it is, select the correct destination block ID to use in your diff calculation. Hint: use the `select` instruction.
- Use two maps to remember the dest and diff values for each basic block, for example destMap[bb] = dest and diffMap[bb] = diff.

Traverse over the basic blocks:

- Update operands for the phi-nodes. Use the destMap and diffMap to fill in the operands for all predecessors.

The assert_ft and assert_cfg_ft functions

The runtime functions shown in the code snippets above, assert_ft and assert_ft_cfg, will be added to the module during compile time. You will be given a function that adds them to the module before your pass runs. It's important that you do not instrument these functions or it could lead to an infinite recursion.

`assert_ft` takes two arguments:

- argument 0: an i32 that is 1 if the instructions matched and 0 if they did not.
- argument 1: a unique i32 identifier for each call to assert_ft that is used for debugging. You may choose whatever value you want.

`assert_cfg_ft` takes two arguments:

- argument 0: an i32 that is 1 if the dest matched the label and 0 if they did not.
- argument 1: is the destination that was calculated, for debugging purposes.
- argument 2: a unique i32 identifier for each call to assert_cfg_ft that is used for debugging. You may choose whatever value you want.

Interaction with Scalar Optimizations

Due to the high cost of code replication, I've added code to the p3 tool to perform O2-level optimization first. This will ensure the code you replicate is as lean as possible. You should not perform optimizations after your transformation or they may remove the replicated code, defeating the purpose of the transformation.

Evaluating the Effectiveness of a Fault Tolerance Transformation

To determine the effectiveness of your technique, two techniques will be used:

- estimation of code coverage (provided in the template code).
- fault injection.

Code Coverage. To estimate code coverage, a statistic will be generated that shows you how many branches an how many instructions your transformation was able to protect. For example, here's an example output for `susan` before your implementation:

```
  21  - number of functions
5335  - number of instructions
 844  - number of loads
  70  - number of stores
   0  - number of protected CFG edges
 818  - number of CFG edges
   0  - number of protected instructions
```

After you've implemented some code, the bolded lines will show some number of instructions. Note, you will never reach full coverage because the code you insert adds overhead and is not redundant. Also, some types of instructions cannot be handled by your pass, so they will not be covered either.

Fault Injection. Errors will be injected into the code to test if your pass is able to catch them and abort. Here's how we'll do it:

- First, the code is optimized and transformed using your pass.
- Next, a special pass corrupts the inputs of one instruction. It could be an original or a replicated instruction.
- Then, we run the corrupted program and see if an error is detected.

We will repeat this process for each benchmark many times (10 times) and estimate a probability of error detection. If your code is able to match the detection rate of my tool (+/- a fudge factor to be determined), then you will earn full marks.

Extra Credit: Enhancing the Technique

You are not required to implement an optimization, but you may do so for extra credit. Once you are done implementing both code replication and control flow checking, think about how to improve this technique. Here are some ideas:

1. **Easy**. Reduce the number of comparisons and checks. Rather than checking the result of every computation, only check some of the computations. But, you must do so without significantly changing error coverage. Furthermore, you must check any instruction that can raise an exception or lead to bad behavior before it executes, namely loads, stores, and branch conditions (e.g. icmp that feeds a branch).
2. **Medium.** Replicating by two does not enable recovery. But, replicating by three allows voting on the correct answer. Rather than duplicating code, triplicate the code and vote on the best answer. If two of the instructions agree, then the answer with 2 votes wins. assert_ft should be called if no majority opinion is reached. This should be enabled by a flag (-bonus) in your submission and otherwise turned off.

2. Hard. The arguments passed to functions are not protected, nor are we protecting control flow when jumping into a function. Extend your pass in the following ways:

- Verify that control passes to functions properly by passing a Dest and Diff argument to functions and verifying that the correct function was reached in the entry block.
- Verify that control returns to functions properly by verifying the return path.
- Duplicate all function arguments of integer or pointer type so that each argument can be checked.

Implementation

Code Development and Testing Setup

You may implement your project in either C or C++. Use one of these files to implement and test your code:

	Stub implementation to edit	Binary to use for testing
C++	projects/p3/C++/p3.cpp	build/p3
C	projects/p3/C/swft.c	build/p3

This is only a starting point! A stub version of the entry point to the optimization (SoftwareFaultTolerance) function has already been implemented for you in either C or C++ starter file.

You may not change the name of the `SoftwareFaultTolerance` function. You may add or modify other content of the files except for how the statistics are dumped. These will be used by the grader script and must be retained as is. However, you may add more statistics if it's helpful.

Manual Setup for VCL or in the container

Make a build directory in the source folder, configure it using cmake, and then build and test your code. Note, if you are working inside the container, you will need to go to the host shared volume under /ece566. Please adjust the PREFIX path appropriately for your use case.

C	`cd PREFIX/ncstate_ece566_spring2024/projects/p3/C`
	`mkdir build`
	`cd build`
	`cmake ..`
	`make`

C++	`cd PREFIX/ncstate_ece566_spring2024/projects/p3/C++`
	`mkdir build`
	`cd build`
	`cmake ..`
	`make`

Testing

Test your code using wolfbench as you did in Project 0.

1. Make a directory for testing. In the `p3/C` or `p3/C++` is fine, but you can put it anywhere you want:
   a. `mkdir p3-test`
   b. `cd p3-test`

2. Assuming that `p3-test` is in your C or C++ directory:

No faults	`path/to/wolfbench/configure --enable-customtool=/path/to/p3`
With faults	`path/to/wolfbench/configure --enable-customtool=/path/to/p3 --enable-faultinjecttool=/path/to/fi` I've created an extra program for you called fi (for fault injection) in your source code directory that you can use to inject faults. If you update your repo, you will see it getting built along side your project code. You will find the binary in the same folder as p3 binary.

4. Compile the benchmarks to test your code with and without optimization:
   No support for fault tolerance:
   ```
   make EXTRA_SUFIX=.None CUSTOMFLAGS=”-verbose -no-swft” all test
   ```
   With your pass:
   ```
   make EXTRA_SUFIX=.SWFT CUSTOMFLAGS=”-verbose ” all test
   ```
5. To verify that the test cases continue to function properly after optimization, use an extra make flag that will compare the outputs to the correct outputs:
   ```
   make EXTRA_SUFIX=.SWFT compare
   ```

6. To check with errors injected, make a testing directory that supports fault injection, maybe call it p3-faults. Then, test the code with and without your transformation. You should see many failures in the form of seg-faults and other errors when your pass is disabled. When your pass is enabled, you should see aborts that report a message that a soft error was detected.
   ```
   path/to/wolfbench/configure --enable-customtool=/path/to/p3 --enable-
   faultinjecttool=/path/to/fi
   make EXTRA_SUFIX=.None CUSTOMFLAGS=”-verbose -no-swft” all test
   make EXTRA_SUFIX=.SWFT CUSTOMFLAGS=”-verbose ” all test
   ```

You can pass flags to the fi tools using FIFLAGS, which is important for changing number of errors and kind of errors.

```
make EXTRA_SUFIX=.SWFT CUSTOMFLAGS=”-verbose ”   \
     FIFLAGS="--inject-count=10 --no-flow-errors " all test
```

7. If you encounter a bug, you can run your tool in a debugger this way:
    1. `make EXTRA_SUFFIX=.SWFT clean`
    2. `make EXTRA_SUFFIX=.SWFT DEBUG=1`

   This will launch lldb using your tool on the first un-compiled input file, which is likely the one failing. You can set breakpoints directly in your source files. This option will only work on the VCL image for debugging with lldb installed.
   For more info on how to use lldb.

Cloning Instructions and Dependences

To replicate code, you should make use of the Instruction::clone() function or for C, LLVMInstructionClone(LLVMValueRef). This will make a copy of one instruction.

After making a copy, you need to insert it into the code into the basic block. To insert the cloned instruction beside the original:

C	C++

`LLVMValueRef I = ...`
`LLVMValueRef Clone = LLVMCloneInstruction(I);`
`LLVMBuilderRef Builder;`
`LLVMPositionBuilderBefore(Builder,I);`
`LLVMInsertIntoBuilder(Builder,Clone)`
	`Instruction *original = ...`
	`Instruction *clone = original->clone();`
	`clone->insertBefore(original);`

Next, you need to update the operands to receive the copied version of the operand, not the original.

C	C++

`LLVMValueRef I = ...`
`LLVMValueRef Clone = ...`
`LLVMValueRef NewOperand = ...`
`LLVMSetOperand(Clone, 0, NewOperand);`
	`Instruction *original = ...`
	`Instruction *clone = ...`
	`Instruction *newOperand = ...`
	`clone->setOperand(0, newOperand);`

Tracking Cloned Instructions and Handling Dependences

Here is one strategy for duplicating all code and updating all appropriate dependences:

```
cloneMap = {} // empty map, use O(1) lookup
for all instructions, i:
       if okay to clone i:
          c = clone(i)
          insert c in to basic block beside i
          cloneMap[i] = c // map each original to each clone

for all cloned instructions, c:
     for i in operands range:
          // change each operand to match the cloned version
          if getOperand(c,i) is an instruction with a clone:
             setOperand(c, i, cloneMap[getOperand(c,i)])
```

The basic idea is that you map the original to the clones. Once you have made all of the clones, you can just update the clones to use operands from the other clones. Because of SSA form, each instruction's operand can be coming from a single instruction which *may* have a clone. If it does, update it.

Calling assert_ft and assert_cfg_ft

Inserting a call in LLVM IR is much like creation of any other instruction. We need to know:

- The function we want to call.
- The arguments to the function.

The Builder object may be used for creating a call:

C++

```
Instruction *I = ... // instruction to check
BasicBlock::iterator it(I);
it++; // insert before next instruction
IRBuilder<> Builder(&*it); // insert before I
std::vector<Value*> args;
args.push_back(Builder.getInt32(1)); // example
args.push_back(Builder.getInt32(35)); // unique id
Function *F = M->getFunction("assert_ft");
Builder.CreateCall(F->getFunctionType(),F, args);
```

C

```
LLVMValueRef I = ... // instruction's result to check
LLVMBasicBlockRef BB = ... // I's basic block
LLVMBuilderRef Builder;
LLVMPositionBuilder(Builder, BB, LLVMGetNextInstruction(I));
// Get reference to assert_ft function
LLVMValueRef Fun = LLVMGetNamedFunction(Module, "assert_ft");
// Make args array
LLVMValueRef *Args = [LLVMConstInt(LLVMInt32Type(),0,true),
LLVMConstInt(LLVMInt32Type(),0,true)];
// Build the call
LLVMBuildCall2(Builder, LLVMTypeOf(Fun), Fun, Args, 2,"");
```

Calling `assert_cfg_ft` works the same way, you just need to pass an additional argument and change the name of the function when looking up the function object in the module object.

Getting Help

History shows that my specs are sometimes incomplete or incorrect. Therefore, please start early. If you run into problems, please post a question to Piazza.

Grading

ECE 466 students are allowed to work in groups of two. ECE 566 students must work individually.

Questions to answer in a report. When you submit your code, also submit a report that answers the following questions. Please include data to justify your claims.

1. How many instructions were added as a result of code replication and control flow checking, both separately and combined? Do not include the results from the Extra Credit optimization in this answer.
2. How much slower is the program as a result of code replication and control flow checking, both separately and combined? Use the same stipulations as Question 1.
3. Bonus: Describe any bonus work you implemented and how effective it was. Provide a quantitative analysis.

Uploading to GradeScope: Upload your p2.cpp or cse.c file. If you added other files or made changes to other files, please submit those, too. If you modified the CMakeLists.txt, upload that as well. Include the report as a pdf along with your code. It will be ignored by the autograder script but made available to the instructor.

We will test your code using the test cases provided in wolfbench and with some secret cases we did not provide.

The assignment is out of 100 points total. <u>If you make no attempt and submit the provided code without meaningful changes (i.e. white space and comments do not count), you earn a maximum of 10 points total</u>. Otherwise, assigned according to the rubric.

ECE 466
10 points: Compiles properly with no additional warnings or errors.
05 points: Code is well commented and written in a professional coding style.
60 points: Implements code replication correctly.
15 points: Detects all errors (that it should detect).
10 points: Report, points divided evenly per question.

ECE 566
10 points: Compiles properly with no additional warnings or errors.
05 points: Code is well commented and written in a professional coding style.
60 points: Implements code replication and control flow checking correctly.
15 points: Detects all errors (that it should detect).
10 points: Report, points divided evenly per question.

Bonus Points
10 points: ECE 466 students only, for implementing Control Flow Checking
05 points: Easy optimization (All students)
10 points: Medium optimization (All students)
15 points: Hard optimization (All students)

Note, you will only earn bonus points if you have a fully working project. You may only earn one category of Bonus Points. In other words, you can’t get 20 points by implementing an Easy with a Hard.