Solidity Compiler Tools User Manual

Killian Rutherford (KRR2125)
Raphael Norwitz (RSN2117)
Jiayang Li (JL4305)

1. **Introduction**
2. **FTime-Report Tool**
    a. **Elements**
        i. **TimeNode**
        ii. **TimeNodeStack**
            1. **void push(std::string name)**
            2. **std::string pop()**
            3. **std::string printString(bool tree)**
            4. **void print()**
            5. **void print_recursive(const TimeNode& x, const std::string& arrow, std::stringstream& ss, bool tree)**
        iii. **TimeNodeWrapper**
            1. **Constructor**
            2. **Destructor**
            3. **void TimeNodeWrapper::pop()**
    b. **Usage**
        i. **Flag details**
        ii. **Within Compiler Source Code**
    c. **Test Cases and Results**
3. **Attempts at Optimising Using Tool**
    a. **Optimise()**
    b. **Analyze()**
4. **Parallelisation**
    a. **Usage**
    b. **Elements**
        i. **Join_contracts**
        ii. **Time_test**
    c. **Other**
        i. **Diff.py**
        ii. **Copyshell script**
    d. **Results**
5. **Parallelisation Tool**
    a. **Elements**
        i. **int compile()**
        ii. **void distribute_tasks()**
        iii. **void parallel_compile()**
    b. **Usage**
        i. **Command line details**
        ii. **[path_to_compiler]**
        iii. **[flags]**
        iv. **[directory]**
        v. **[extension]**
    c. **Test Cases**
6. **References**

1. Introduction

This User Manual goes through the f-time profiling and the parallelism tool we created, diving into specific usage, test cases and function implementations. The manual also documents our attempts at using these tools to optimise the runtime of the solidity compiler. For details on the solidity language, see the official documentation at: https://solidity.readthedocs.io/en/v0.4.23/.[1] After discussing with the solidity compiler community, who liked the idea of a built-in profiling tool, we designed our tool based off of Ftime Report in GCC [2]. Having struggled to find concrete optimizations with the profiling tool, we moved on to build parallelized compilation tool, which we rigorously document here.

See the Design Document for a more linear narrative of the problems we encountered and decisions we made.

For a lighter and more practical introduction to our tool, see the Tutorial.

All code is found on our github page https://github.com/raphael-s-norwitz/solidity [3] under various branches (dev_tool, dev_tool_wrapper, dev_optimise, dev_optimise_profile, dev_analyze_profile, dev_junk, dev_parallel_tool, dev_parse_profile).

2. FTime-Report Tool

a. Elements

The below elements can be found on either the dev_tool or dev_tool_wrapper branch. Dev_tool consists of our original design, dev_tool_wrapper includes a wrapper which implements RAII more thoughtfully and handles pop() in the destructor as opposed to users having to call it. [3]

i. TimeNode

The TimeNode class is defined in libsolidity/interface/FTime.h and FTime.cpp files. It’s private members consist of chrono::high_resolution_clock times to define begin and end; along with set functions which set these values correspondingly (setBegin(), setEnd()) and get functions which obtain these values (getBegin(), getEnd()).

The constructor simply instantiates the vector of TimeNodes which acts as a list of that TimeNode’s children.

No specific destructor is specified - default destructor [3].

ii. TimeNodeStack

TimeNodeStack class is defined in libsolidity/interface/FTime.h and FTime.cpp files. Private members include a vector of TimeNodes serving as the stack, as well as a print_stack vector of TimeNodes and a start time to calibrate all timing differences printed out.
A boolean print_flag is also defined (if this is set to true, pop() will print new nodes to stdout) - which is useful if the --ftime-report flag has been defined. There is also an plain output flag --ftime-report-no-tree.

The Constructor simply instantiates start time.

The Destructor checks if the stack is non empty and gives an error if it is not; otherwise falls to the default destructor [3].

1. void push(std::string name)

Creates a TimeNode object with the given name argument. It then sets the begin time of that node and pushes it onto the stack.

2. std::string pop()

If the stack has multiple elements, the last node from the stack is popped, the end time of it is set, and it is added to the children list of the new top node of the stack.

If the stack has one element, it means the end of the profiling because the pops and pushes for the highest level call has occurred. Thus the time report should be printed with a printString() call, or it can be added to the print_stack, which can be dumped to standard out with an explicit print() call. A pop() call on an empty stack throws an std::runtime_error.

The return value is the string name member of the TimeNode popped [3].

3. std::string printString(bool tree)

Prints all nodes to a string using the print_recursive function. Note that if the print_flag is on, anytime the stack size reaches 1, the time report will be printed to stdout, and printString() will return an empty output. Hence if a user wants to call printString on a local TimeNodeStack, which only looks at elements in the print_stack, the print_flag should remain off, which it is by default. TimeNodes which have yet to be popped will not show up in the string returned.

4. void print()

Simply prints the return value of printString to stdout.

5. void print_recursive(const TimeNode& x, const std::string& arrow, std::stringstream& ss, bool tree)

Recurse through all the node’s children and print out in format. If bool tree is true, format the output as a tree structure by adding indents to the arrow string.

iii. TimeNodeWrapper

TimeNodeWrapper class is defined in libsolidity/interface/FTime.h and FTime.cpp files; on the dev_tool_wrapper branch.

We introduced this class so that we could utilize RAII to call pop() in the class destructor, minimizing the number of statements needed to profile a block of code.

Private members of string name, popped flag (in case pop() was called before destructor); and TimeNodeStack reference member [3].

1. Constructor

Takes as input TimeNodeStack (whether a globally or locally defined one), and string name; Calls the TimeNodeStack.push(name), i.e. creates a new node and pushes it on the stack.

2. Destructor

Checks if the popped flag is on; does nothing if it is set, otherwise pop the stack and set the flag to true and default destruct. Note that we are operating under the assumption TimeNodeWrapper objects will be destructed in reverse order they are constructed (put on the Memory Stack Frame) within a function block. If this is violated, the names will not match up and an error will be thrown (see below).

3. void TimeNodeWrapper::pop()

Simply checks by name if the node has already been popped (throws error if it has). Otherwise it pops the TimeNodeStack reference and does an extra check to make sure the popped name is equal to the name saved in the constructor (i.e. the right Node is being popped). It throws an std::runtime_exception if these do not match.

b. Usage

Please see https://solidity.readthedocs.io/en/v0.4.23/ [1] for solidity documentation and how to build the compiler. Once the compiler solc has been built running the ./scripts/build.sh script on a branch which supports (e.g. dev_tool, dev_tool_wrapper), the flag can be fed into the solc command line arguments.

i. Flag details

Once the solidity compiler has been compiled, and the solc executable produced, a solidity file or multiple solidity files can be compiled through solc A.sol B.sol ....
Please see the solidity documentation for explanation regarding already existing flags, such as --optimize and --bin to produce binary source files. The FTimeReport flag is added with
`--ftime-report` **or** `--ftime-report-no-tree`
The no-tree flag prints what has been profiled without the tree indentation, which may be desirable in cases where there are many nested profiling statements.

The flag is handled in solc/CommandLineInterface.cpp in the handleFtimeReport function. This only deals with printing the global t_stack variable which is defined in FTime.h; by printing everything in the print_stack and setting the print_flag on for future additions [3].

ii. Within Compiler Source Code

To profile code within the compiler, the global t_stack variable, which has been defined in FTime.h, should be used. The print() functions and print_flags are called in the CommandLineInterface.cpp file, so no additional work is needed to add debug statements to the existing output.

Local TimeNodeStacks can also be defined and used, but in this case the print_flag of the local TimeNodeStack needs to be turned on when the stack is defined or print() must be called to get the debug string. Note that the TimeNodeStack needs to be accessible everywhere you wish to put profiling statements.

The FTime-report tool source is located in libsolidity/interface, so to import it into other libraries within solidity, it is necessary to add the libsolidity to the target_link_libraries within the specific library's CMakeLists.txt [4].

Besides this, one needs to add #include <libsolidity/interface/FTime.h> at the beginning of the source file in which the code wanting to be profiled is located.

Then, one can either explicitly stack.push(string) and stack.pop() around the code block wanting to be profile, or use the RAII TimeNodeWrapper. See tutorial for further details.

c. Test Cases and Results

See tutorial for simple test cases; as well as the design document and sections below for further uses. Boost test cases are located on the dev_tool or dev_tool_wrapper branch; in the test/libsolidity/FTimeTest.cpp file.

Some of the tests are simple, and check that standard functions like push() and pop() don't throw errors when and only when expected. We use std::regex to evaluate the printed output, making sure that the trees contained the correct strings and were indented as expected. The most complicated tests involved string parsing with regex; as these were quite challenging to get right. Please see the above file again for specific cases.

3. Attempts at Optimising Using Tool

Initial runs of the FTimeReport Tool show optimise, parse and analyse functions within the compiler taking a lot of time.

a. Optimise()

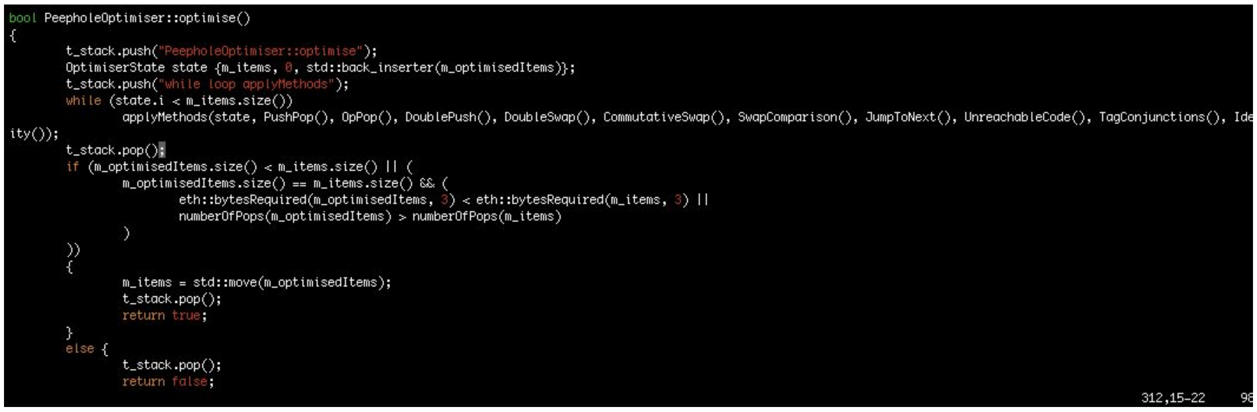
Using the tool on a simple contract Coin.sol, we see CompilerContext::optimise takes up quite a lot of the time. Below results are conducted on the dev_optimise_profile branch.

```
killianrutherford1@mbp ~/Desktop/Courses/cppDesign/proj/solidity $ solc --ftime-report Coin.sol
Warning: This is a pre-release compiler version, please do not use it in production.
Coin.sol:15:5: Warning: Defining constructors as functions with the same name as the contract is deprecated. Use
"constructor(...) { ... }" instead.
   function Coin() public {
   ^ (Relevant source part starts here and spans across multiple lines).
namespace/function name                              unix begin time(μs)   time elapsed(μs)
-----------------------------------------------------------------------------------------------------------------
CLI::readInputFilesAndConfigureRemappings                285                 221
CompilerStack::setRemappings                             516                 0
CompilerStack::setEVMVersion                             554                 0
CompilerStack::compile                                   557                 3629
 \_CompilerStack::parse                                  557                 125
 \_CompilerStack::analyze                                683                 1247
 \_CompilerStack::compileContract: Coin                  1934                2239
    \_CompilerStack::createMetadata                      1942                223
    \_Compiler::compileContract                          2272                1582
       \_ContractCompiler::compileContract               2277                400
          \_ContractCompiler::initializeContext          2283                20
          \_ContractCompiler::appendFunctionSelector     2304                234
          \_ContractCompiler::appendMissingFunctions     2542                134
       \_ContractCompiler::initializeContext             2684                7
       \_ContractCompiler::appendMissingFunctions        2722                0
```

```
  \_ContractCompiler::appendMissingFunctions    2724      0
  \_CompilerContext::optimse                    2726      1115
    \_Assembly::optimse                         2728      1103
      \_OptimiserSettings::optimise             2728      1093
        \_OptimiseInternal                      2729      1082
          \_OptimiseInternal                    2732      939
            \_test1                             2734      10
            \_runPeephole                       2744      615
              \_PeepholeOptimiser::optimise     2746      297
                \_while loop ApplyMethods       2747      288
              \_PeepholeOptimiser::optimise     3045      307
                \_while loop ApplyMethods       3045      279
            \_runDeduplicate                    3361      0
            \_test3                             3362      0
            \_test1                             3362      8
            \_runPeephole                       3371      298
              \_PeepholeOptimiser::optimise     3371      291
                \_while loop ApplyMethods       3371      265
            \_runDeduplicate                    3670      0
            \_test3                             3670      0
          \_test1                               3675      1
          \_runPeephole                         3677      58
            \_PeepholeOptimiser::optimise       3677      26
              \_while loop ApplyMethods         3677      24
            \_PeepholeOptimiser::optimise       3704      28
              \_while loop ApplyMethods         3704      25
          \_runDeduplicate                      3737      0
          \_test3                               3737      0
          \_test1                               3737      0
          \_runPeephole                         3738      32
            \_PeepholeOptimiser::optimise       3738      30
              \_while loop ApplyMethods         3739      27
          \_runDeduplicate                      3771      0
          \_test3                               3771      0
          \_test1                               3772      0
          \_runPeephole                         3772      33
            \_PeepholeOptimiser::optimise       3773      31
              \_while loop ApplyMethods         3773      28
          \_runDeduplicate                      3806      0
          \_test3                               3806      0
\_ContractCompiler::initializeContext           3933      6
\_ContractCompiler::appendMissingFunctions      3982      0
\_Assembly::optimse                             3983      156
  \_OptimiserSettings::optimise                 3983      147
    \_OptimiseInternal                          3983      138
      \_OptimiseInternal                        3984      8
        \_test1                                 3984      0
        \_runPeephole                           3984      3
          \_PeepholeOptimiser::optimise         3984      1
            \_while loop ApplyMethods           3985      0
```

_runDeduplicate	3988	0
_test3	3989	0
_OptimiseInternal	3993	23
_test1	3993	0
_runPeephole	3994	20
_PeepholeOptimiser::optimise	3994	18
_while loop ApplyMethods	3994	15
_runDeduplicate	4015	0
_test3	4015	0
_test1	4017	1
_runPeephole	4019	63
_PeepholeOptimiser::optimise	4019	28
_while loop ApplyMethods	4020	24
_PeepholeOptimiser::optimise	4048	31
_while loop ApplyMethods	4049	28
_runDeduplicate	4084	0
_test3	4084	0
_test1	4084	0
_runPeephole	4085	31
_PeepholeOptimiser::optimise	4085	29
_while loop ApplyMethods	4086	26
_runDeduplicate	4117	0
_test3	4118	0
CommandLineInterface::actOnInput	4243	1018

Using the tool in further detail, we break up into further levels of detail above, and see that almost all of the time in CompilerContext::optimise is split up in the while loop applyMethods in libevmasm/PeepholeOptimiser.cpp (see snippet below). [4]



```
bool PeepholeOptimiser::optimise()
{
	t_stack.push("PeepholeOptimiser::optimise");
	OptimiserState state {m_items, 0, std::back_inserter(m_optimisedItems)};
	t_stack.push("while loop applyMethods");
	while (state.i < m_items.size())
		applyMethods(state, PushPop(), OpPop(), DoublePush(), DoubleSwap(), CommutativeSwap(), SwapComparison(), JumpToNext(), UnreachableCode(), TagConjunctions(), Ide
ity());
	t_stack.pop();
	if (m_optimisedItems.size() < m_items.size() || (
		m_optimisedItems.size() == m_items.size() && (
			eth::bytesRequired(m_optimisedItems, 3) < eth::bytesRequired(m_items, 3) ||
			numberOfPops(m_optimisedItems) > numberOfPops(m_items)
		)
	))
	{
		m_items = std::move(m_optimisedItems);
		t_stack.pop();
		return true;
	}
	else {
		t_stack.pop();
		return false;
                                                                                312,15-22     9
```

As seen below, there's no one applyMethods that seems to be taking that much longer than any other (i.e. the methods). It seems rather a combination of all of them which takes a long time, so if their calls can be minimized somehow, that may improve performance. See Design document for some attempts that were made.

```
\_CompilerContext::optimse                          2739      50342
  \_Assembly::optimse                               2741      46939
    \_OptimiserSettings::optimise                   2741      43510
      \_OptimiseInternal                            2741      40108
        \_OptimiseInternal                          2743      33196
          \_test1                                   2745      9
          \_runPeephole                             2755      18447
            \_PeepholeOptimiser::optimise           2756      7466
              \_while loop ApplyMethods             2757      6473
                \_applyMethods                      2757      17
                  \_applyMethods                    2757      14
                    \_applyMethods                  2757      12
                      \_applyMethods                2757      10
                        \_applyMethods              2758      8
                          \_applyMethods            2758      6
                            \_applyMethods          2759      4
                              \_applyMethods        2760      2
                                \_applyMethods      2761      0
                                  \_applyMethods    2761      0
                \_applyMethods                      2775      16
                  \_applyMethods                    2776      13
                    \_applyMethods                  2776      9
                      \_applyMethods                2776      7
                        \_applyMethods              2776      5
                          \_applyMethods            2776      3
                            \_applyMethods          2777      2
                              \_applyMethods        2777      1
                                \_applyMethods      2777      0
                                  \_applyMethods    2777      0
                \_applyMethods                      2793      13
                  \_applyMethods                    2793      11
                    \_applyMethods                  2793      9
                      \_applyMethods                2793      7
                        \_applyMethods              2793      5
                          \_applyMethods            2793      3
                            \_applyMethods          2794      2
                              \_applyMethods        2794      1
                                \_applyMethods      2794      0
                                  \_applyMethods    2795      0
                \_applyMethods                      2809      13
                  \_applyMethods                    2809      11
                    \_applyMethods                  2809      8
                      \_applyMethods                2809      6
                        \_applyMethods              2809      5
                          \_applyMethods            2809      3
                            \_applyMethods          2810      1
                              \_applyMethods        2810      1
                                \_applyMethods      2810      0
                                  \_applyMethods    2810      0
                \_applyMethods                      2823      13
```

```
\_applyMethods                    2824          11
  \_applyMethods                  2824          8
    \_applyMethods                2824          6
      \_applyMethods              2824          5
        \_applyMethods            2824          3
          \_applyMethods          2824          2
            \_applyMethods        2825          1
              \_applyMethods      2825          0
                \_applyMethods    2825          0
```

Even with larger contracts (compilationTests/corion/provider.sol), we see optimise and applymethods taking large amounts of time, so this seems like a good place to be looking to shave off compile time. [4]

```
killianrutherford1@mbp ~/Desktop/Courses/cppDesign/proj/solidity $ solc --ftime-report
test/compilationTests/corion/provider.sol | grep ApplyMethods

\_while loop ApplyMethods             242722        11
      \_while loop ApplyMethods        248394        3557
      \_while loop ApplyMethods        252041        3350
      \_while loop ApplyMethods        255873        3300
      \_while loop ApplyMethods        259627        3523
    \_while loop ApplyMethods          263523        467
    \_while loop ApplyMethods          264004        455
    \_while loop ApplyMethods          264471        452
    \_while loop ApplyMethods          264989        452
  \_while loop ApplyMethods            266536        0
  \_while loop ApplyMethods            266547        14
\_while loop ApplyMethods              266587        471
\_while loop ApplyMethods              267071        458
\_while loop ApplyMethods              267540        457
\_while loop ApplyMethods              268063        459
      \_while loop ApplyMethods        271548        290
      \_while loop ApplyMethods        271847        289
      \_while loop ApplyMethods        272176        288
    \_while loop ApplyMethods          272496        11
    \_while loop ApplyMethods          272509        11
    \_while loop ApplyMethods          272527        11
    \_while loop ApplyMethods          272543        11
  \_while loop ApplyMethods            272675        0
  \_while loop ApplyMethods            272699        13
\_while loop ApplyMethods              272729        10
\_while loop ApplyMethods              272742        11
\_while loop ApplyMethods              272758        10
      \_while loop ApplyMethods        283165        7557
      \_while loop ApplyMethods        290915        6169
      \_while loop ApplyMethods        297218        5768
```

```
        \_while loop ApplyMethods      303377         5787
        \_while loop ApplyMethods      309959         7950
        \_while loop ApplyMethods      318073         6675
        \_while loop ApplyMethods      325557         6284
      \_while loop ApplyMethods        332443         142
```

b. Analyse()

According to the ftime report, the analyze pass CompilerStack::analyze generally takes around 50% of the total compile time when compiling the Coin contract. Below results are conducted on the dev_analyze_profile branch.

```
namespace/function name                                unix begin time(μs)   time elapsed(μs)
-----------------------------------------------------------------------------------------------------
CommandLineInterface::processInput()                        358              3384
 \_CLI::readInputFilesAndConfigureRemappings                363              257
 \_CompilerStack::setRemappings                             631              2
 \_CompilerStack::setEVMVersion                             680              0
 \_CompilerStack::compile                                   684              3016
    \_CompilerStack::parse                                  684              175
    \_CompilerStack::analyze                                860              1573
    \_CompilerStack::compileContract: Coin                  2438             1254
       \_CompilerStack::createMetadata                      2450             266
       \_Compiler::compileContract                          2856             661
          \_ContractCompiler::compileContract               2861             303
             \_ContractCompiler::initializeContext          2869             26
             \_ContractCompiler::appendFunctionSelector     2896             94
             \_ContractCompiler::appendMissingFunctions     2994             168
          \_ContractCompiler::initializeContext             3175             7
          \_ContractCompiler::appendMissingFunctions        3200             0
          \_ContractCompiler::appendMissingFunctions        3201             0
          \_CompilerContext::optimse                        3203             311
       \_ContractCompiler::initializeContext                3603             6
       \_ContractCompiler::appendMissingFunctions           3641             0
CommandLineInterface::actOnInput                            3757             172
```

Splitting the analyze pass further by applying our tools to .sol files that is relevant to the analyze pass, we have the following result:

```
namespace/function name                                unix begin time(μs)   time elapsed(μs)
-----------------------------------------------------------------------------------------------------
CLI::readInputFilesAndConfigureRemappings                   376              263
CompilerStack::setRemappings                                650              2
CompilerStack::setEVMVersion                                698              0
CompilerStack::compile                                      701              4253
 \_CompilerStack::parse                                     701              167
 \_CompilerStack::analyze                                   872              1588
    \_SyntaxChecker::checkSyntax                            874              31
       \_SemVerMatchExpressionParser::parse                 882              4
       \_SemVerMatchExpression::matches                     888              0
    \_DocStringAnalyser::analyseDocStrings                  907              16
    \_GlobalContext::declarations                           1036             1
```

```
  \_NameAndTypeResolver::NameAndTypeResolver          1040          20
  \_GlobalContext::setCurrentContract                 1095          0
  \_GlobalContext::currentThis                        1096          1
  \_GlobalContext::currentSuper                       1101          1
  \_ReferencesResolver::resolve                       1110          7
  \_ReferencesResolver::resolve                       1117          5
  \_ReferencesResolver::resolve                       1125          1
  \_ReferencesResolver::resolve                       1127          0
  \_ReferencesResolver::resolve                       1129          4
  \_ReferencesResolver::resolve                       1134          1
  \_ReferencesResolver::resolve                       1136          0
  \_ReferencesResolver::resolve                       1137          1
  \_ReferencesResolver::resolve                       1139          0
  \_ReferencesResolver::resolve                       1140          5
  \_ReferencesResolver::resolve                       1151          4
  \_ReferencesResolver::resolve                       1156          6
  \_TypeChecker::checkTypeRequirements                1164          216
  \_PostTypeChecker::check                            1382          4
  \_StaticAnalyzer::analyze                           1388          15
  \_ViewPureChecker::check                            1404          24
  \_SMTChecker::SMTChecker                            1429          831
  \_SMTChecker::analyze                               2261          15
 \_CompilerStack::compileContract: Coin               2469          2478
  \_CompilerStack::createMetadata                     2482          323
  \_Compiler::compileContract                         2928          1739
    \_ContractCompiler::compileContract               2934          510
      \_ContractCompiler::initializeContext           2941          26
      \_ContractCompiler::appendFunctionSelector      2968          287
      \_ContractCompiler::appendMissingFunctions      3260          182
    \_ContractCompiler::initializeContext             3451          7
    \_ContractCompiler::appendMissingFunctions        3499          0
    \_ContractCompiler::appendMissingFunctions        3500          0
    \_CompilerContext::optimse                        3503          1162
  \_ContractCompiler::initializeContext               4752          8
  \_ContractCompiler::appendMissingFunctions          4813          0
CommandLineInterface::actOnInput                      5069          234
```

As you can see, SMTChecker::SMTChecker and TypeChecker::checkTypeRequirements took most of the time in the analyze pass. For SMTChecker, the compiler spent more than 800 microseconds on the constructor of the object [3]. The relevant part of the code follows:

```
if (noErrors)
{
        t_stack.push("SMTChecker::SMTChecker");
        SMTChecker smtChecker(m_errorReporter, m_smtQuery);
        t_stack.pop();
        for (Source const* source: m_sourceOrder)
                smtChecker.analyze(*source->ast);
}
```

libsolidity/interface/CompilerStack.cpp

Next, let’s look at the SMTChecker constructor in libsolidity/formal/SMTChecker.cpp

```
SMTChecker::SMTChecker(ErrorReporter& _errorReporter, ReadCallback::Callback const& _readFileCallback):
#ifdef HAVE_Z3
	m_interface(make_shared<smt::Z3Interface>()),
#elif HAVE_CVC4
	m_interface(make_shared<smt::CVC4Interface>()),
#else
	m_interface(make_shared<smt::SMTLib2Interface>(_readFileCallback)),
#endif
	m_errorReporter(_errorReporter)
{
	(void)_readFileCallback;
}
```

Another thing we tried was to apply our profiling tool to Z3Interface constructor, and it turned out that when the Z3Interface constructor was called, approximately 800 microseconds already passed since ViewPureChecker::check, which is the previous pass of SMTChecker constructor, finished. Therefore, we believe the std::function ReadCallback::Callback const& _readFileCallback is a major bottleneck of the compiler’s speed.

In terms of TypeChecker::checkTypeRequirements, we decided to divide the checkTypeRequirements function into 3 parts and recorded the time spent on each part. We found that the third part is what usually makes the type checking slow. [4]

```
bool TypeChecker::visit(ContractDefinition const& _contract)
{
	t_stack.push("TypeChecker::visit 1");
	m_scope = &_contract;
	/* omitted */
	t_stack.pop();
	t_stack.push("TypeChecker::visit 2");
	/ * omitted */
	t_stack.pop();
	t_stack.push("TypeChecker::visit 3");
	for (auto const& n: _contract.subNodes())
		if (!visited.count(n.get()))
			n->accept(*this);
	t_stack.pop();
	/* omitted */
	return false;
}
```

libsolidity/analysis/TypeChekcer.cpp

_TypeChecker::checkTypeRequirements	1233	228
_TypeChecker::visit 1	1234	49
_TypeChecker::checkContractDuplicateFunctions	1237	11
_TypeChecker::checkContractDuplicateEvents	1250	3
_TypeChecker::checkContractIllegalOverrides	1254	4
_TypeChecker::checkContractAbstractFunctions	1260	10
_TypeChecker::checkContractBaseConstructorArguments	1276	4
_TypeChecker::visit 2	1284	1
_TypeChecker::visit 3	1287	100
_TypeChecker::checkContractExternalTypeClashes	1388	38

Ftime-report targeting TypeChecker

The for loop in TypeChecker::visit 3 takes around 50% of the time of the TypeChecker::checkTypeRequirements pass. In this piece of the code, the compiler is visiting each ASTNode in the contract and checking if there is any error. Therefore, we believe it is reasonable for this part to use longer time and it would be difficult, if not impossible, to speed this part up.

4. Parallelisation

For details into the thought processes behind parallelisation and methods attempted, please see the Design Document section 3.

The final parallelisation code which was used to run results is located in the dev_optimise branch, in the directory parallelisation_test/ [3].

a. Usage

This code is to see the effect of using C++'s async and seeing possible time gains to running several number of source files within a library split over different numbers of processors.

E.g. given 10 solidity files, A.sol - J.sol, and 4 processors, we would compare the times running:

solc A.sol B.sol C.sol D.sol E.sol F.sol G.sol H.sol I.sol J.sol
(on 1 processor)
Vs.

CPU1: solc A.sol B.sol C.sol
CPU2: solc D.sol E.sol F.sol
CPU3: solc G.sol H.sol

CPU4: solc I.sol J.sol

As is mentioned, there are several limitations to this which the user should be aware of by reading the design document, mainly that these files must be independent from one another, and cannot have interlinked dependencies or redefinitions of classes, etc…

The Makefile specifies C++ 17 to be used, as experimental filesystem is needed to traverse filesystems and directories. (In the future this may need to be changed to just filesystem)

Compiling the executable can be done by simply typing make within the parallelisation_test/ directory. A test executable is produced, and is run with two command line arguments: N (the number of files wanting to run the comparison over), and the path to the directory containing these files. The source code described below is mainly located in parallelisation_test/select_contracts_2.cpp [3].

b. Elements

i. Join_contracts

In order to find out if N number of files from a directory path given can be compiled with one another; and potentially parallelised over multiple CPU's, join_contracts takes in a compiler string, a select_flag for this compiler, a path to the directory, a test string consisting of the files to compile and N.

It iterates through the solidity files in the directory using experimental::filesystem. The function adds the new file to the test_string, and runs the compiler with the test_string, checking to see if the binary produced is valid.

If it is, the test_string is updated to include this new file. The above process is repeated and continues until N is reached or until all files in the directory are exhausted.

Join_contracts returns the number of files it was able to compile, if this is less than the original N, this means not enough files in the directory were able to satisfy the user's initial request.

ii. Time_test

After join_contracts is executed, and the test_string variable contains N files which do not have interdependencies or issued, time_test is called; taking in arguments of vector of all the files, test_string, compiler command and flags, and directories for single thread and multi thread binary output.

Using the thread library, the number of CPU's on the machine is calculated, the files are split optimally into different strings for each CPU. For multiprocessing, a results vector of future<int> is created. Using the async() call, each task on a CPU is executed and pushed onto the vector. get() is later on called on each element of the vector to obtain the timing result. This is done using chrono::high_resolution clock; and single processor compiling is also executed, printing out both results.

All binary files during compilation are outputted to respective single and multi processing directories.

See design document for some notes on the binary files produced in some cases with large numbers of varied files.

c. Other

i. Diff.py

A python script diff.py is also provided in the parallelisation_test/ directory to ensure all binary files within both single thread and multi thread directories match.

ii. Copyshell script

A bash shell script copy_shell.sh is provided to automatically run three consecutive times: make the test executable, run with a specific N and path string, and run the diff python file on the the binary output directories.

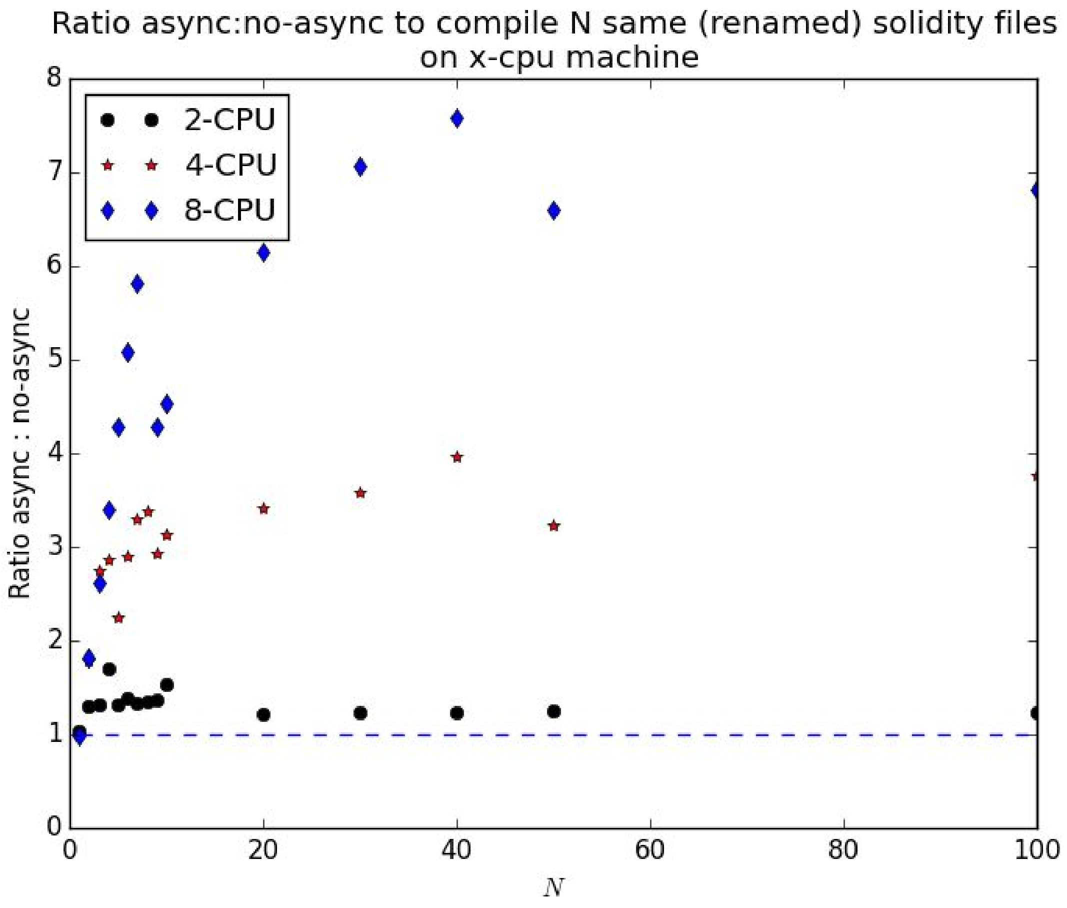
d. Results

(See Design Document for more in-depth discussion)

The first dataset we tested on was selecting a random contract, and copying it a certain number of times, whilst altering the name of the contract classes and functions, to prevent as much as possible the compiler from optimizing and predicting / cache hits, thereby trying to force it to recompile every single contract.

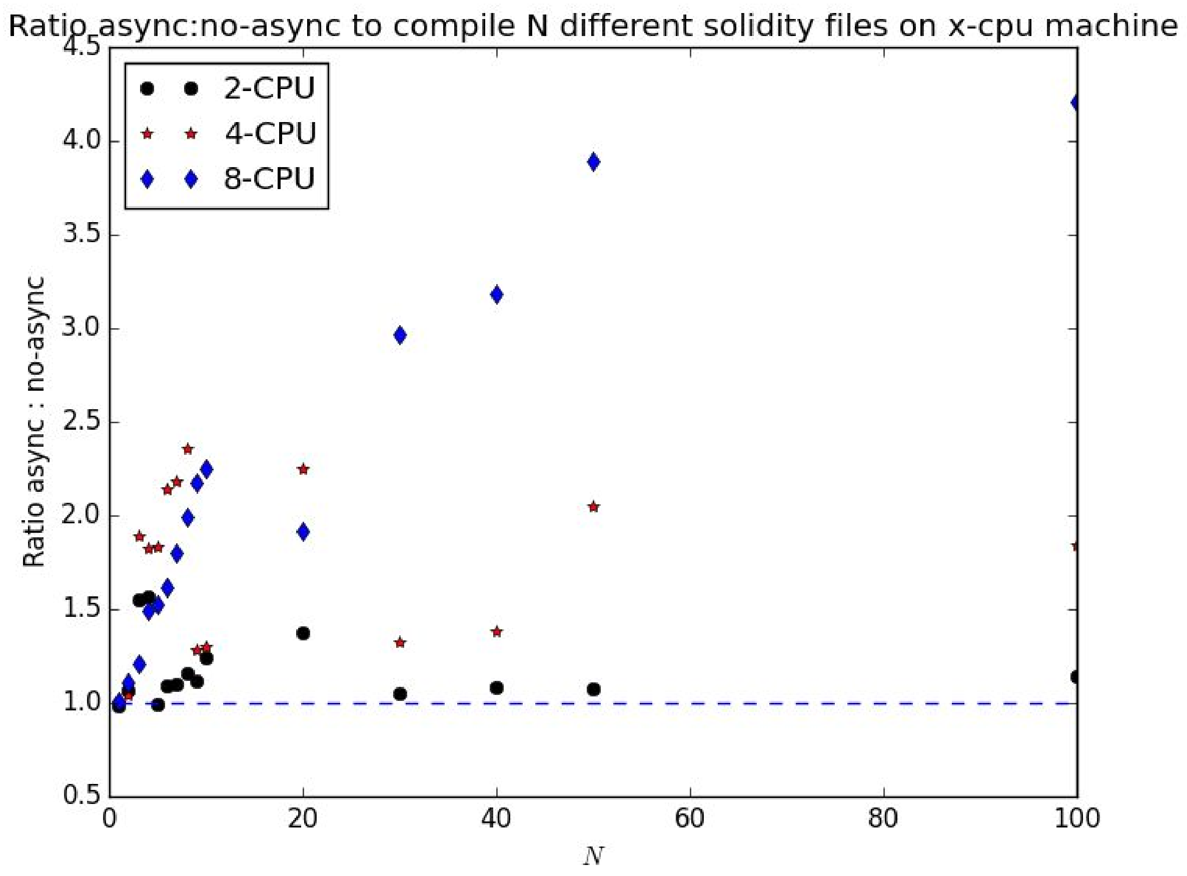
This was done with BlindAuction.sol file, which contains both a BlindAuction contract and a purchase contract. (see https://solidity.readthedocs.io/en/v0.4.23/solidity-by-example.html for the files) [1].

Time ratios were averaged over 3 runs and results over different number of google instance n1-standard vCPU are shown. (n1-standard-2 (2 vCPUs, 7.5 GB memory) ; n1-standard-4 (4 vCPUs, 15 GB memory) ; n1-standard-8 (8 vCPUs, 30 GB memory) [5].



We also wanted to test out parallelisation on a more realistic data set (i.e. not just copies of a specific file) to see what could be a potential “real-world” speedup to benchmark and to avoid any thread taking advantage of cache memory. We were pointed by the solidity developer’s team towards https://github.com/allenday/github-solidity-all, [6] which contains all solidity files on github collected in one place.

We tested on 100 files from this repo (See Design Document):
8894 lines; 33,362 words; 294,515 characters



5. Parallelisation Tool

The tool should be accessed in the dev_parallel_tool branch in the parallel/ directory.

a. Elements

i. int compile(string compiler, string flags, string files, std::experimental::filesystem::path current)

The compile() function is defined in parallel/cmp.cpp. It is a function that generates the command to compile the solidity files and then calls the host environment's command processor to execute the command.

ii. void distribute_tasks(string& path, string& extension, vector<tuple<std::experimental::filesystem::path, vector<string>>>& pool, bool isRoot)

The distribute_tasks() function is defined in parallel/cmp.cpp. It populates the pool vector by filling it with batches of files that will be compiled together. It will leverage the C++ experimental filesystem library to search for files to compile. When isRoot is true, the function will consider all solidity files in the **path** to be independent from each other, and thus they can be distributed to different threads in any order. When isRoot is false, it will distribute all solidity files in **path** to a single task since we assume any solidity files in subdirectories need to be compiled together [3].

ii. void parallel_compile(string& compiler, string& flags, vector<string> &sources, vector<future<int>> &vec_res)

The parallel_compile() function is defined in parallel/cmp.cpp. It will evenly partition files in **sources** into different batches for compiling, and it will consider the hardware resources of the machine for the partition. Each partition will then be passed to a **compile()** task and then added to the **vec_res** waiting to be executed parallelly [3].

b. Usage

i. Command line details

To compile the parallelisation tool, use gcc 7 and run the following command:
```
G++-7 -o parallel -std=c++17 -O2 par_cmp.cpp -lstdc++fs -lpthread
```
The **parallel** executable takes 4 arguments, which are **[path_to_compiler]**, **[flags]**, **[directory]**, and **[extension]**.

An example command:
```
./parallel "solc" "--bin --ignore-missing" "parallel/test" ".sol"
```

ii. [path_to_compiler]

Path to the compiler executable. If the **solc** executable is in /usr/bin, then set it to be "/usr/bin/solc". If your environment has **solc** as a global executable, then you can just make the path_to_compiler "solc".

iii. [flags]

Flags the users would like to specify for the solidity compiler. If a user just wants the compiler to export binary files, then just use “--bin --ignore-missing”. Additionally, since our tool will automatically put the generated binary files under a “bin” folder in each directory, users should not include any output directory in the flags, which means they should not include -o in the flags.

iv. [directory]

The top level directory for the files to be compiled. Put all files that can be compiled independently under this directory and all other files into subdirectories under this directory.

v. [extension]

The extension of the solidity files. Usually it will just be “.sol”. Remember to put a dot before the extension. The parallelisation tool will only compile a file if it finds the extension in its filename.

c. Test Cases

Checkout the dev_parallel_tool branch of our github repo. In the folder **parallel**, there is a Makefile and a **test** directory that contains a collection of solidity files. Generate the parallelisation tool by “make” command and use **test**’s path as the **[directory]** argument for the parallelisation tool and compile the files parallely. After the compiling process finished, you will find the generated binaries in the bin folders under **test** or any of its subdirectories.

6. References

[1] "Solidity — Solidity 0.4.23 documentation - Read the Docs." https://solidity.readthedocs.io/. Accessed 26 Apr. 2018.
[2] "Using the GNU Compiler Collection (GCC): Developer Options." https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html. Accessed 26 Apr. 2018.
[3] "raphael-s-norwitz (Raphael Norwitz) · GitHub." https://github.com/raphael-s-norwitz. Accessed 26 Apr. 2018.
[4]"ethereum/solidity - GitHub." https://github.com/ethereum/solidity. Accessed 26 Apr. 2018.
[5] "Virtual Machine Instances | Compute ...." https://cloud.google.com/compute/docs/instances/. Accessed 26 Apr. 2018.
[6] "GitHub - allenday/github-solidity-all: All *.sol files from github in a ...." https://github.com/allenday/github-solidity-all. Accessed 26 Apr. 2018.

All code taken from our github repository: github.com/raphael-s-norwitz/solidity [3]
Manual, tutorial and design document located on submissions branch.