Solidity Compiler Tools Tutorial

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

Large solidity libraries and test sets take a long time to compile. Having to wait up to seven seconds to test a program can be irritating and wastes programmers' time. We have focused on building tools to help debug compiler performance, and parallelize compilation. Here, we present the basic functionality of our tool, and list best practices and gotchas.

Before this tutorial, the reader should be familiar with solidity and some examples, which can be accessed here: https://solidity.readthedocs.io/en/v0.4.23/ [1]

All code is based from our github link https://github.com/raphael-s-norwitz/solidity [2]. See the User Manual for details about where everything is located (e.g. git branches)

1. Ftime-report Profiling API

The TimeNode interface is a performance debugging profiler for the solidity compiler. Currently we have built it into the compiler via a command line flag **--ftime-report** [3] so that if one runs **$solc --ftime-report ./myContract.sol** with our solidity build, it will present a decent debugging report. Here is a basic hello world example.

```
{
    TimeNodeStack myStack;
    {
       TimeNodeWrapper hello(myStack,"hello");
       TimeNodeWrapper world(myStack,"world");
       TimeNodeWrapper    ex(myStack, "!!");
    }
    std::cout << myStack.printString(true) << "\n";
};
```

When run in isolation (As a BOOST test case, for instance) it will output this:

```
namespace/function name                                  unix begin time(μs)     time elapsed(μs)
-------------------------------------------------------------------------------------------------------
hello                                                    0                       1
 \_world                                                 0                       1
        \_!!                                             1                       0
```

The core class of our API is the TimeNodeStack, as demonstrated above. TimeNodeStack objects can be used to capture a set of debugging statements while they are in scope. In practice, you can define them yourself, or use the global TimeNodeStack object **t_stack** for compiler-wide profiling.

To get the profile report as a string, call the printString() function on a TimeNodeStack. As you see above the printString() function takes a bool, and in this case it is true. This bool is used to toggle the format of the output. In the output above, we see the string “!!” nested under “world”, which is in turn nested under “hello”. Think of this output as a tree, where all the nodes indented under a given node are descendants of that node. Thus here the root of the tree is ‘hello’ and it has one child, ‘world’, which in turn has another child ‘!!’. This implies that the time elapsed for “world” includes the time elapsed for “!!” and the time elapsed for “hello” includes the time elapsed for “world” and “!!”. When printString() is instead called with ‘false’ it won’t indent nested output, and thus for the same code the output would be:

```
namespace/function name                                  unix begin time(μs)     time elapsed(μs)
-------------------------------------------------------------------------------------------------------
hello                                                    0                       1
 \_world                                                 0                       1
 \_!!                                                    0                       0
```

Next you’ll notice the class TimeNodeWrapper. This is the class that you would use to perform profiling. By defining a TimeNodeWrapper object, passing in a TimeNodeStack object and a string to print/profile code around, you start benchmarking from the constructor, within which a ‘begin time’ reading is taken from chrono::high_resolution_clock. By default in the destructor, an ‘end time’ is taken the same way, and the difference between the start and the end is the number you will see on the report. Thus if two TimeNodeWrapper objects are created within the same block, you will see that the second will be listed as a child first, as it will start profiling later than the first and end at the same time [2].

You may have wondered why we created the scope in the prior example. It is precisely because we need the destructors to fire in order to capture the benchmarks. Had we had code like this:

```
{
    TimeNodeStack myStack;
    TimeNodeWrapper hello(myStack,"hello");
```

```
    TimeNodeWrapper world(myStack,"world");
    TimeNodeWrapper    ex(myStack, "!!");
    std::cout << myStack.printString(true) << "\n";
};
```

We would have gotten this:

namespace/function name unix begin time(μs) time elapsed(μs)

As if no TimeNodeWrapper objects had been created. This is because no destructor is ever called to get the end time for either profile before the printString() function is called.

Now see the following pieces of code:

```
{
    TimeNodeStack myStack;
    {
       TimeNodeWrapper hello(myStack,"hello");

       sleep(1);

       hello.pop();

       TimeNodeWrapper world(myStack,"world");

       sleep(1);

    }
    std::cout << myStack.printString(true) << "\n";
};
```

```
{
    TimeNodeStack myStack;

    TimeNodeWrapper hello(myStack,"hello");
    sleep(1);
    hello.pop();

    TimeNodeWrapper world(myStack,"world");
```

```
    sleep(1);
    hello.pop();

    std::cout << myStack.printString(true) << "\n";
};
```

Both will give you the following output:

```
namespace/function name                                   unix begin time(μs)     time
elapsed(μs)
-------------------------------------------------------------------------------------------------------------------------------
hello                                                              0               1000103
world                                                        1000108               1000090
```

Note that in this tree, hello is now a sibling of world. Here the only difference was a call hello.pop(), which is going to remove hello from the TimeNodeStack. The pop() call takes a TimeNodeWrapper object and makes it a child of the previous TimeNodeWrapper created (a root, as in this case where there are no TimeNodeWrappers created associated with a given TimeNodeStack). Concretely, the pop() function captures the end point of a benchmark, and then sets a flag so that the destructor will not attempt to find the endpoint or append it to the prior TimeNodeWrapper's children list. It also checks that the debug string you specified to the object matches the string that is popped off the stack, such that if you write code like this:

```
{
    TimeNodeStack myStack;
    {
       TimeNodeWrapper hello(myStack,"hello");

       sleep(1);

       TimeNodeWrapper world(myStack,"world");

       hello.pop();

       sleep(1);

    }
    std::cout << myStack.printString(true) << "\n";
};
```

where you call pop() on hello, which is not the last element created, it will throw an std::runtime_error [2].

The TimeNodeWrapper and TimeNodeStack represent a complete interface to profile, and when using our tool, we recommend you always use them. That said, we do have a lower level interface which could be preferred in rare circumstances. See the following code:

```
{
    TimeNodeStack new_stack;

    new_stack.push("hello");
    new_stack.push("world");
    new_stack.push("!!");
    new_stack.pop();
    new_stack.pop();
    new_stack.pop();


    std::cout << new_stack.printString(true) << "\n";

}
```

Is functionally identical to the first code example and will print:

```
namespace/function name                                    unix begin time(μs)     time elapsed(μs)
-------------------------------------------------------------------------------------------------------------
hello                                                                0               1
 \_world                                                             0               1
       \_!!                                                          1               0
```

Note that here you call push(), given a debug string, and pop() directly on the TimeNodeStack. While this may appear simpler in this case, you must remember to add a pop() call on the stack before every possible point where your code could exit a block. This includes putting a pop() before every return statement. With the higher level API, there is no need to do this, as RAII will take care of it. Unlike with TimeNodeWrapper, here there are no checks on what is being popped, which may lead to confusion or erroneous benchmarks when a programmer has forgotten to call pop().

One possible scenario where such semantics could be favorable is if you have many TimeNodeWrapper objects going out of scope at once, and you want to make sure they are destructed in proper order (so that the TimeNodeWrapper objects which are created first are destroyed last), so you have to call pop() anyways. We still advise that you call pop on TimeNodeWrapper objects where possible, but the lower level semantics are there for tricky cases [2].

Now how does this look in practice? In /libsolidity/codegen/Compiler.cpp, in the function compileContract() we have added the following example [4]:

```
void Compiler::compileContract(
    ContractDefinition const& _contract,
    std::map<const ContractDefinition*, eth::Assembly const*> const&
_contracts,
    bytes const& _metadata
)
{
    TimeNodeWrapper profileCompileContract(t_stack,
"Compiler::compileContract");
      ContractCompiler runtimeCompiler(nullptr, m_runtimeContext,
m_optimize);
    runtimeCompiler.compileContract(_contract, _contracts);
    m_runtimeContext.appendAuxiliaryData(_metadata);

    // This might modify m_runtimeContext because it can access runtime
functions at
    // creation time.
    ContractCompiler creationCompiler(&runtimeCompiler, m_context,
m_optimize);
    m_runtimeSub = creationCompiler.compileConstructor(_contract,
_contracts);
    TimeNodeWrapper profileOptimize(t_stack, "CompilerContext::optimse");
    m_context.optimise(m_optimize, m_optimizeRuns);
}
```

Here we first create the profileCompileContract TimeNodeWrapper. This object will be the ‘parent’ profiling statement for all the ContractCompiler constructor for runtime compiler, ContractCompiler compileContract() function, appendAuxiliaryData() function, ContractCompiler constructor for creation compiler, ContractCompiler compileConstructor() fuction and the optimize() call. Each of these may have many TimeNodeWrapper objects within them, profiling different parts of the compilations process. Lastly we see a constructor for another TimeNodeWrapper, profileOptimize, which will profile only the optimize() call. We can see that it is a direct child of the profileCompileContract object, and will thus be right below it in the tree. Both of these profiling instances will have their end time set by their destructors (since there is no pop call) [4].

Critically note that we do not define a TimeNodeStack here. Throughout the compiler, we advise that you use the globally defined t_stack TimeNodeStack instance, so that the profiles that you create will be seamlessly integrated with the existing --ftime-report and --ftime-report-notree flags, and visible from the solc executable.

Thus say we run **$solc --ftime-report <path>/<to>/<example>.sol** we get something like the following:

```
-------------------------------------------------------------------------------------------------------------------------------------------------------------
CLI::readInputFilesAndConfigureRemappings                                373             0
CompilerStack::setRemappings                                            9773            16
CompilerStack::setEVMVersion                                            9910            1
CompilerStack::compile                                                  9917            5927
 \_CompilerStack::parse                                                 9920            404
 \_CompilerStack::analyze                                               10326           1792
 \_CompilerStack::compileContract: Oracle                               12123           2860
          \_CompilerStack::createMetadata                               12139           152
          \_Compiler::compileContract                                   12308           2449
                   \_ContractCompiler::compileContract                  12312           547
                            \_ContractCompiler::initializeContext       12313           15
                            \_ContractCompiler::appendFunctionSelector  12330           165
                            \_ContractCompiler::appendMissingFunctions  12496           361
                   \_ContractCompiler::initializeContext                12866           6
                   \_ContractCompiler::appendMissingFunctions           12907           0
                   \_ContractCompiler::appendMissingFunctions           12908           0
                   \_CompilerContext::optimse                           12913           1838
          \_ContractCompiler::initializeContext                         14846           18
          \_ContractCompiler::appendMissingFunctions                    14933           0
 \_CompilerStack::compileContract: OracleUser                           14988           853
          \_CompilerStack::createMetadata                               14996           118
          \_Compiler::compileContract                                   15128           601
                   \_ContractCompiler::compileContract                  15131           183
                            \_ContractCompiler::initializeContext       15131           4
                            \_ContractCompiler::appendFunctionSelector  15136           74
                            \_ContractCompiler::appendMissingFunctions  15212           100
                   \_ContractCompiler::initializeContext                15319           3
                   \_ContractCompiler::appendMissingFunctions           15341           0
                   \_ContractCompiler::appendMissingFunctions           15342           0
                   \_CompilerContext::optimse                           15345           382
          \_ContractCompiler::initializeContext                         15762           4
          \_ContractCompiler::appendMissingFunctions                    15795           0
CommandLineInterface::actOnInput                                        16094           113
```

Here there are two contracts, Oracle and OracleUser, and we can see our profiling statements work exactly as described, with optimize being the direct child of Compiler::compileContract in both cases.

This about finishes up the basic functionality of our TimeNode tool. It is a simple, but effective way to profile the performance of the solidity compiler and it's stand alone implementation should make it portable to other C++ projects [2].

For more detailed tests and uses, see the User Manual Section 2c and the Design Document.

2. The async test

The async test in the /parallelization_test/ directory of our dev_optimize branch.

To build, navigate to the directory and run make. Then run ./test <a number of files to test> ./test_files.

See User Manual section 4 for more detailed tests and results.

3. The parallelization tool

The parallelization tool is in the /parallel/ directory of our dev_parallel_tool branch.

3.1 Goal:

To address the issue that it takes a relatively long time to compile large solidity libraries, we think about leveraging modern hardware's parallel features. Therefore, we build a high level parallelization tool to run the solidity compiler asynchronously. Studying and using the tool and its code will give users a picture of how parallelization can help speed up compiling as well as some of its limitations [2].

3.2 How to use:

First, install gcc 7.0 and then compile the tool with Makefile or with the 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> <extension>**.

<path_to_compiler> is the path to the compiler executable. **<flags>** is the command line flags a user would like to feed into the compiler. **<directory>** is where sources live and **<extension>** is the extension for the sources.

Assume that we put some ".sol" files in a directory called **"/rc/test"** and would like to compile them with the flag **"--bin --ignore-missing"**, then the command to compile the sources parallelly would be

$parallel "solc" "--bin --ignore-missing" "/rc/test" ".sol"

If you don't want the tool to output stderr message, then append **2> /dev/null** to the end of the above command. This will mute the stderr. After the process finishes, you can find the binary files in the "bin" folders in your top-level directory and each subdirectory [4].

3.3 Assumptions:

We assume that the top-level **<directory>** will contain all solidity files that are safe to compile parallelly. By contrast, any batch of solidity files that depend on each other to compile (for example, files that import other files), which means they need to be compiled together, will be

placed in the same subdirectory under the top-level directory, so that all of them will be targeted by a specific thread in our design.

```
/*
If isRoot is true, the helper function will scan for source files and
distribute them to different threads later.

If the function finds a subdirectory, it will recursively call itself by
setting isRoot to false, so that the next level distribute_tasks will put
all source files in the subdirectory together in the same task.
*/

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

3.4 Improvement on what is already available

According to part 2 of this tutorial (see User Manual and Design Document), the parallel compiling algorithm can achieve more than a 60% speed up than the original solidity compiler when compiling over 50 independent solidity files on multi-core machines. We expect the tool to have even greater performance as the magnitude of the solidity libraries scale up and more CPU resources become available.

3.5 Issues for the tool

The parallelization tool may report an error that it is not able to import another solidity file y if it compiles a file x that used “import y” instead of “import ./y”[1]. We cannot assume that the user executes the compile command in the directory of x and y. And therefore, if the user uses the tool in other directories, the referenced directory for y will be incorrect. A quick fix to this is to replace all “include y” in the solidity sources with “include ./y”. We have also considered changing the current_path of the process when entering the compiling process to fix this issue, but due to the fact that all threads share the same current_path and the parallelized nature of our tool, we cannot guarantee the current_path is determined when the compiling process happens.

4. References:

[1] "Layout of a Solidity Source File — Solidity 0.4.21 ... - Read the Docs." http://solidity.readthedocs.io/en/v0.4.21/layout-of-source-files.html. Accessed 24 Apr. 2018.
[2] "raphael-s-norwitz (Raphael Norwitz) · GitHub." https://github.com/raphael-s-norwitz. Accessed 26 Apr. 2018.
[3] "Using the GNU Compiler Collection (GCC): Developer Options." https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html. Accessed 26 Apr. 2018.
[4] "ethereum/solidity - GitHub." https://github.com/ethereum/solidity. Accessed 26 Apr. 2018.

All code taken from our github repository: github.com/raphael-s-norwitz/solidity