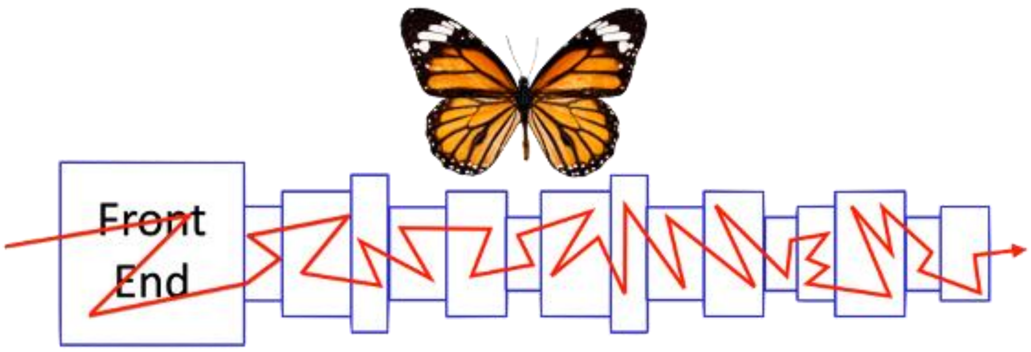
Code Generator Development and Validation
Why SuperTest should be your first choice for compiler code generator development and validation



SuperTest for code generator development and validation

The code generator is a critical part of the compiler that requires more specific testing than the rest of the compiler. SuperTest™ is the best test suite to do this for C and C++ compilers. Other test suites are only designed to verify language compliance. Unlike other tools, SuperTest can be used from the very start of new compiler development projects.

SuperTest contains many specific strategies to put the code generator through its paces: First of all, by the very nature of compilers, every test passes the code generator – the final elements of the picture above. Every test in the suite challenges the code generator. Then we have our arithmetic suite that verifies the details of arithmetic at the bit-exact level. Arithmetic depends on the data model of the compiler and it relies on the exact choice of instruction and bit masks by the code generator. SuperTest's Calling Convention Tester gives the function call interface a workout. Together with efficient register allocation, getting the calling conventions right is one of the major challenges in the construction of the code generator.

If you start with code generator development, SuperTest has another unique advantage: the *Code Generator Trainer*. It is discussed in the next section.

The SuperTest test suites are organized according to the language standard specification. This makes it easy to omit library testing from the initial test runs. SuperTest does not depend on a working library implementation to run the language (compiler) specific tests and it needs only a minimal interface to report the self-checking test results. SuperTest can be used from the start of the code generator development process.

Code generator testing is an important part of compiler testing: often, newly developed C and C++ compilers rely on either the GCC or CLANG+LLVM platform. For those new compilers, few changes, if any, are needed in the front-end. Their front-ends are shared in many compiler implementations and are well tested (also by users of SuperTest).

Code generators, on the other hand, are target specific and they are aimed at specialized or application specific processors. They have a small audience and rely on a smaller ecosystem to find and correct errors. Using high quality validation tools for the code generator helps achieve and maintain compiler quality. SuperTest is such a tool.

Code Generator Trainer for Code Generator Development

Code Generator Trainer is a test suite in SuperTest that is specifically aimed at code generator development. It can be used from the very start of code generator creation. It requires no run-time support other than a way to return the test result to the driver. It consists of twenty-five levels that introduce new code generator functionality one step at a time.

The first level, *level00*, establishes the communication with the test driver by leaving the result of *main()* in an agreed location. If that is too complex to start with, you can choose a temporary solution that is to be fixed later when the register allocator configuration is tackled. From there on all the next steps are focused on code generator specifics. Level00 also sets up load and store instructions of primitive types.

The next level introduces signed and unsigned integer operations and comparisons. Every type, comparison and operation has its own test so every time there is exactly one small issue to tackle. Subsequent levels introduce control flow (branches), more types, structures, arrays, etc. The later levels are also heavier on the register allocator, including the tests that introduce longer argument lists to function calls.

After completion of all twenty-five levels with about a thousand tests in total you can be confident that the code generator is more than 99% C coverage complete. It is likely that you can do a run with full SuperTest without too much embarrassment.

However, it is also likely that you still need to do a lot of work on the code generator in order to generate more optimized code with more clever register allocation and scheduling. Code Generator Trainer provides you with an easy to run collection of unit tests that provides an anchor for additional improvements without the fear of breaking the code generator.

Mid-Level optimizations and the code generator

Three important tasks of the code generator are instruction selection, register allocation, and scheduling. In addition, there may be additional components in the compiler that perform target specific tasks. For example, 'lowering' (emulating) unsupported IR-level constructs and converting control flow into predicated execution are partly target specific.

The code generator is also called the 'back-end' because it is the final part of the compiler, after the front-end and the mid-level (mostly target independent) optimizations. For code generator specific tests to reach the code generator in their intended form, it is best to avoid the mid-level optimizations by turning optimizations off. Otherwise, the mid-level optimizations may unintentionally transform the to-be-tested programming construct into another form and fail to test the intended target instructions. Note that even without optimization options, some compilers do still optimize the code. If possible, it is best to turn this off as well.

In later stages of code generator testing it is necessary to also test with optimizations turned on. The mid-level optimizations act like a pinball machine that may hand out strange curve balls to the code generator. This is good testing material, but it provides less control over exactly what is tested in the code generator.

SuperTest's hand written tests

SuperTest contains a huge collection of hand written tests. It has been developed in close association with multiple generations of compiler development systems, including the CoSy® compiler development system. CoSy is a highly retargetable compiler development system that has been used for more than hundred different target architectures. Code generator development, and code generator validation, have always been of primary importance for CoSy.

For that reason, SuperTest contains a complete set of code generator specific tests. These were developed first by enumerating all combinations of constructs, operators and types relevant to the code generator, and then by using code coverage analysis for the code generator to find any gaps, both with and without optimizations turned on. In addition, errors found during code generator development (and other parts of the compiler) were also turned into SuperTest tests.

The Depth Suite for target specific arithmetic

For many good reasons the C language does not specify the number of bits used for the arithmetic types. It only specifies that, for example for the 'int' type, at least 16 bits must be used. But that does not imply that C's arithmetic is loosely defined. To the contrary, the C specification does require that in an actual implementation, the sizes and significant bits used for arithmetic types *are* precisely defined. They are part of the 'implementation defined' data model.

It means that an implementation of C is free to choose the number of bits for arithmetic, but once that choice is made, it needs to be stated and adhered to. Then, based on this choice, the C specification defines unambiguously how arithmetic must behave. For example, if the addition of two unsigned integers overflows, the result is wrapped around using the modulo of the maximum unsigned integer plus one.

In this case, the *maximum unsigned integer* is an implementation defined parameter. While the C definition of arithmetic is therefore quite precise, its dependence on implementation defined properties makes it hard for a general purpose test suite to verify it.

Yet SuperTest can do this. SuperTest includes more than thirty *depth suites* for different implementation specific data models. SuperTest's depth suites are generated and contain millions of data model specific arithmetic tests for both integer and floating point types. These tests cover all combinations of arithmetic types, operators and values, in particular for the corner cases that a general purpose test cannot begin to match.
And what if your particular data model is not one of the thirty included depth suites?

Then Solid Sands creates a new one for that data model.

The tests in the depth suites should also be running with and without optimizations. Without optimizations they directly map to the arithmetic implementation in the code generator, which, as mentioned, is a good thing. With optimizations, they are also targeting the mid-level, but target specific, constant folding transformations. Compile-time constant folding also has to adhere to the rules of the target arithmetic.

The Calling Convention Tester for function calls and register allocation

The Calling Convention Tester (*call-tester*) is designed to give the target specific calling conventions implementation a good workout. The calling conventions of a compiler are implementation specific. They define how the argument values are passed from the calling function to the called function, and also how the result value travels in the opposite direction.

The call-tester is a generator that creates pairs of functions, with each function of the pair in its own file. One of the functions is the *caller*, it calls the other function called the *callee*. The call-tester generates random (but configurable) sequences of arguments that must be transferred from caller to callee. The functions are self-checking, so the callee verifies that it has received the right values, and the caller verifies the callee's return value.

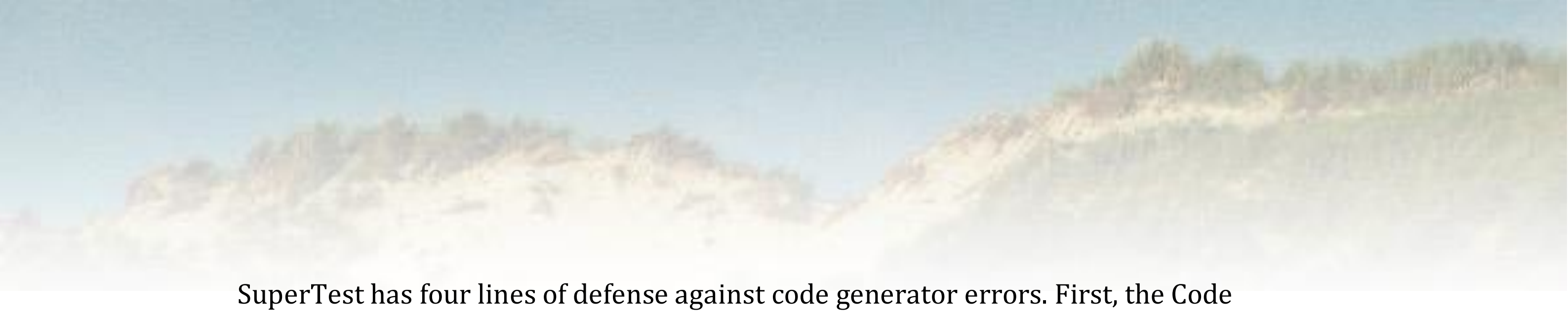
The implementation of the calling conventions is closely linked to the register allocator and its optimizations. Register allocation is complicated because an optimal register assignment is important for generated code performance. So, although that is not its main purpose, the call-tester is also good at testing the register allocator.

Turning optimizations on does not affect the calling convention. But it does affect the *register pressure* around function calls. With optimizations, many more registers are typically in use and the register allocator must make hard decisions about which values to keep in which registers. So here too, it makes sense to start with no optimizations in the initial development of the compiler calling conventions and register allocator, and later raise the bar by turning optimizations on.

The utility of the call-tester is not limited to testing the internal implementation of the calling conventions. The pair of functions can also be compiled by two different compilers to verify that they both agree about the calling convention implementation. Similarly, the pair can be compiled by two different versions of the same compiler in order to check that the calling conventions implementation has not changed between updates.

SuperTest: Your guard against code generation errors

All executable tests in SuperTest end up in the code generator, for the simple reason that the code generator is a necessary part of the compiler's functionality. This property already provides substantial code generator test coverage.



SuperTest has four lines of defense against code generator errors. First, the Code Generator Trainer in SuperTest is a collection of tests that is organized to make the development of a code generator as smooth as possible. Its backbone of a thousand tests also helps to keep all subsequent code generator modifications straight.

Second, SuperTest's hand-written tests contain a full collection of tests for all the operations that can be expected in the code generator. These tests were created and perfected over the course of developing many retargetable compiler systems.

Third, the depth suite is capable of testing the code generator's implementation of all arithmetic operations, types and their combinations. The depth suite verifies the target dependent corner cases of arithmetic because it is generated for the specific implementation defined data model of the compiler.

Fourth, the Calling Convention Tester generates tests to verify the implementation of the compiler's calling convention. It serves as a stress test for the implementation of the target dependent register allocator.

That is why SuperTest should be your first choice for compiler code generator development and validation.