

Future-proofing the GCC compiler for an automotive grade microcontroller

Author Manfred Kreutzer – ABIX

Co-author Marcel Beemster - Solid Sands

ABIX & the project

ABIX is a service provider for the development of software development tools in the embedded and IoT field. In a current project ABIX is tasked with upgrading and improving the GCC compiler for the Belgian company Melexis, a global supplier of microelectronic semiconductor solutions.

The customer & its products

Melexis develops a family of efficient 16-bit RISC-type microcontrollers for a broad variety of applications in the automotive and mechatronic sectors ranging from tire pressure monitoring sensors to BLCD motor controllers. The microcontrollers' instruction sets, register files and co-processors are customized to meet the requirements of these application specific purposes.

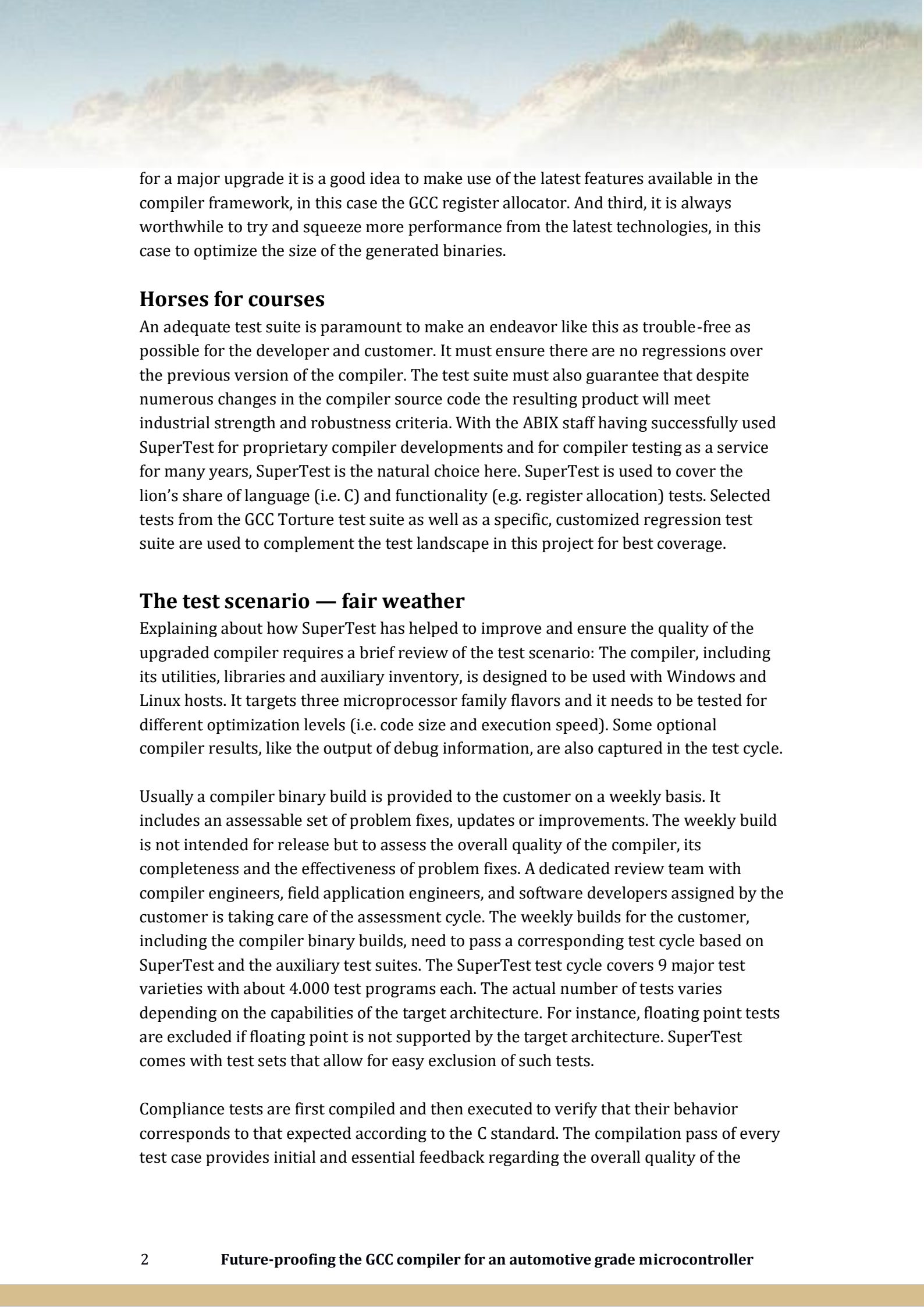
The compiler & the expertise

The GCC compiler was selected as the compiler of choice many years ago. It was adapted to meet the characteristics of this 16-bit architecture family. Special attention was paid to the efficient utilization of the register file, where smaller registers can be combined to broader registers, and to size optimization of the target code to enable small memory footprints and reduce power consumption.

GCC has now been successfully used for several years for this microcontroller family but with the release of GCC version 10 in 2020 the decision to upgrade was made. ABIX has been entrusted to execute this project due to the many years of experience the ABIX team has gathered in designing and developing compiler frameworks and compiler extensions and due to its expertise with the creation of GCC and LLVM compiler extensions for internationally renowned customers from the RISC and DSP microprocessor IP sectors.

Challenges

For this upgrade project there are three major challenges to deal with: First, it needs to be ensured that the upgraded compiler is as robust, trustworthy, and performant as the previous version that the customers relied upon for many years. Second, when deciding



for a major upgrade it is a good idea to make use of the latest features available in the compiler framework, in this case the GCC register allocator. And third, it is always worthwhile to try and squeeze more performance from the latest technologies, in this case to optimize the size of the generated binaries.

Horses for courses

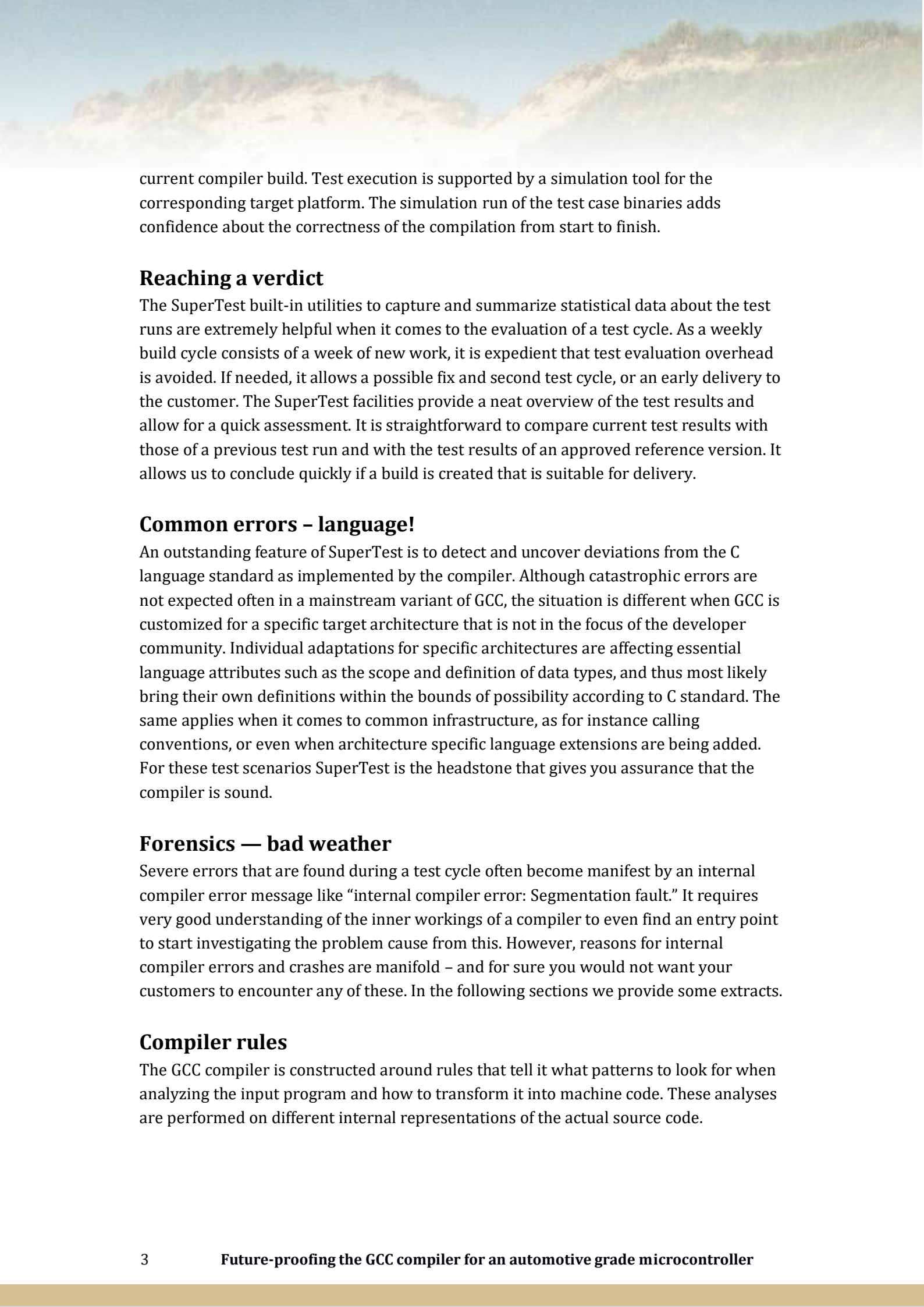
An adequate test suite is paramount to make an endeavor like this as trouble-free as possible for the developer and customer. It must ensure there are no regressions over the previous version of the compiler. The test suite must also guarantee that despite numerous changes in the compiler source code the resulting product will meet industrial strength and robustness criteria. With the ABIX staff having successfully used SuperTest for proprietary compiler developments and for compiler testing as a service for many years, SuperTest is the natural choice here. SuperTest is used to cover the lion's share of language (i.e. C) and functionality (e.g. register allocation) tests. Selected tests from the GCC Torture test suite as well as a specific, customized regression test suite are used to complement the test landscape in this project for best coverage.

The test scenario — fair weather

Explaining about how SuperTest has helped to improve and ensure the quality of the upgraded compiler requires a brief review of the test scenario: The compiler, including its utilities, libraries and auxiliary inventory, is designed to be used with Windows and Linux hosts. It targets three microprocessor family flavors and it needs to be tested for different optimization levels (i.e. code size and execution speed). Some optional compiler results, like the output of debug information, are also captured in the test cycle.

Usually a compiler binary build is provided to the customer on a weekly basis. It includes an assessable set of problem fixes, updates or improvements. The weekly build is not intended for release but to assess the overall quality of the compiler, its completeness and the effectiveness of problem fixes. A dedicated review team with compiler engineers, field application engineers, and software developers assigned by the customer is taking care of the assessment cycle. The weekly builds for the customer, including the compiler binary builds, need to pass a corresponding test cycle based on SuperTest and the auxiliary test suites. The SuperTest test cycle covers 9 major test varieties with about 4.000 test programs each. The actual number of tests varies depending on the capabilities of the target architecture. For instance, floating point tests are excluded if floating point is not supported by the target architecture. SuperTest comes with test sets that allow for easy exclusion of such tests.

Compliance tests are first compiled and then executed to verify that their behavior corresponds to that expected according to the C standard. The compilation pass of every test case provides initial and essential feedback regarding the overall quality of the



current compiler build. Test execution is supported by a simulation tool for the corresponding target platform. The simulation run of the test case binaries adds confidence about the correctness of the compilation from start to finish.

Reaching a verdict

The SuperTest built-in utilities to capture and summarize statistical data about the test runs are extremely helpful when it comes to the evaluation of a test cycle. As a weekly build cycle consists of a week of new work, it is expedient that test evaluation overhead is avoided. If needed, it allows a possible fix and second test cycle, or an early delivery to the customer. The SuperTest facilities provide a neat overview of the test results and allow for a quick assessment. It is straightforward to compare current test results with those of a previous test run and with the test results of an approved reference version. It allows us to conclude quickly if a build is created that is suitable for delivery.

Common errors – language!

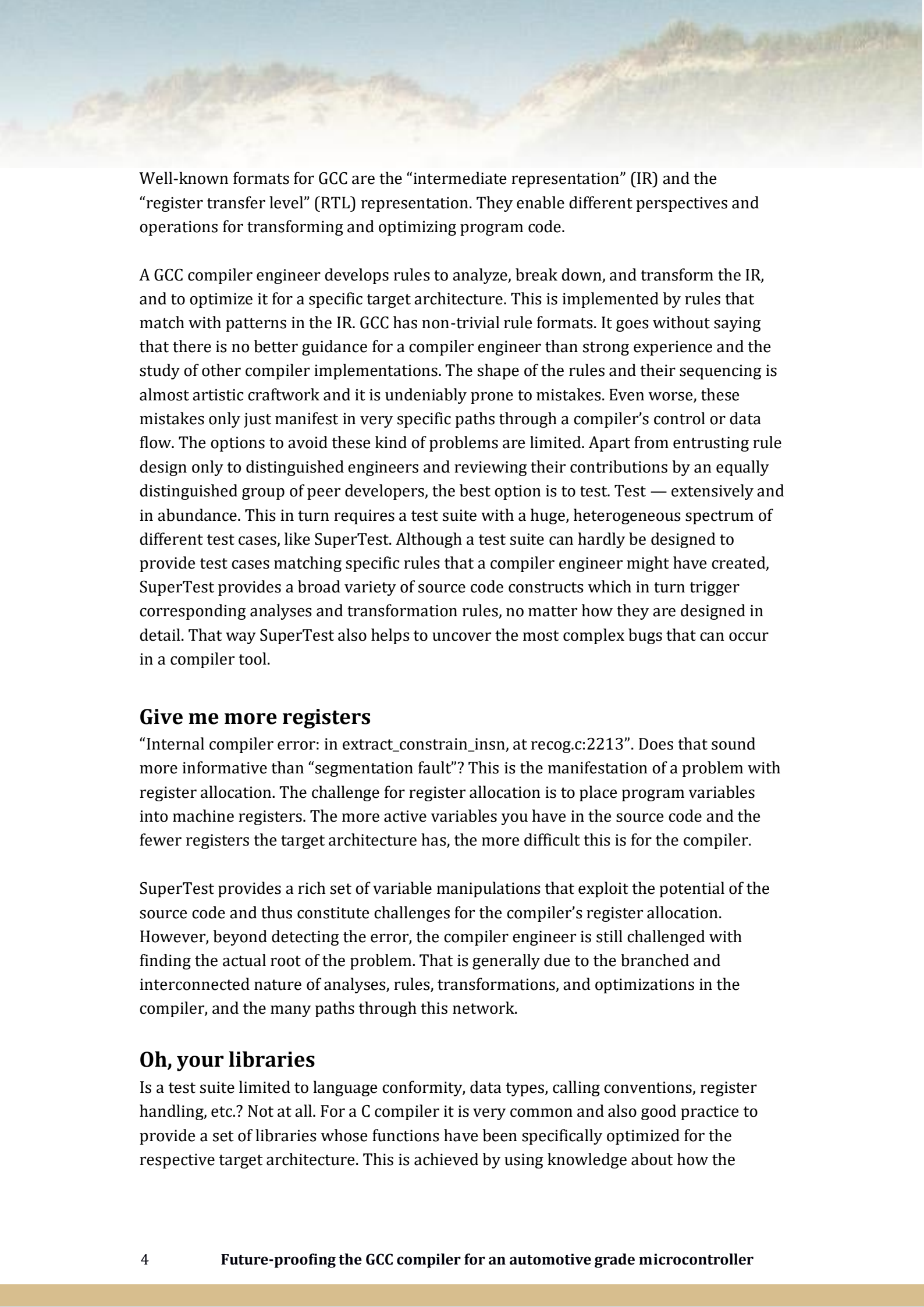
An outstanding feature of SuperTest is to detect and uncover deviations from the C language standard as implemented by the compiler. Although catastrophic errors are not expected often in a mainstream variant of GCC, the situation is different when GCC is customized for a specific target architecture that is not in the focus of the developer community. Individual adaptations for specific architectures are affecting essential language attributes such as the scope and definition of data types, and thus most likely bring their own definitions within the bounds of possibility according to C standard. The same applies when it comes to common infrastructure, as for instance calling conventions, or even when architecture specific language extensions are being added. For these test scenarios SuperTest is the headstone that gives you assurance that the compiler is sound.

Forensics — bad weather

Severe errors that are found during a test cycle often become manifest by an internal compiler error message like “internal compiler error: Segmentation fault.” It requires very good understanding of the inner workings of a compiler to even find an entry point to start investigating the problem cause from this. However, reasons for internal compiler errors and crashes are manifold – and for sure you would not want your customers to encounter any of these. In the following sections we provide some extracts.

Compiler rules

The GCC compiler is constructed around rules that tell it what patterns to look for when analyzing the input program and how to transform it into machine code. These analyses are performed on different internal representations of the actual source code.



Well-known formats for GCC are the “intermediate representation” (IR) and the “register transfer level” (RTL) representation. They enable different perspectives and operations for transforming and optimizing program code.

A GCC compiler engineer develops rules to analyze, break down, and transform the IR, and to optimize it for a specific target architecture. This is implemented by rules that match with patterns in the IR. GCC has non-trivial rule formats. It goes without saying that there is no better guidance for a compiler engineer than strong experience and the study of other compiler implementations. The shape of the rules and their sequencing is almost artistic craftwork and it is undeniably prone to mistakes. Even worse, these mistakes only just manifest in very specific paths through a compiler’s control or data flow. The options to avoid these kind of problems are limited. Apart from entrusting rule design only to distinguished engineers and reviewing their contributions by an equally distinguished group of peer developers, the best option is to test. Test — extensively and in abundance. This in turn requires a test suite with a huge, heterogeneous spectrum of different test cases, like SuperTest. Although a test suite can hardly be designed to provide test cases matching specific rules that a compiler engineer might have created, SuperTest provides a broad variety of source code constructs which in turn trigger corresponding analyses and transformation rules, no matter how they are designed in detail. That way SuperTest also helps to uncover the most complex bugs that can occur in a compiler tool.

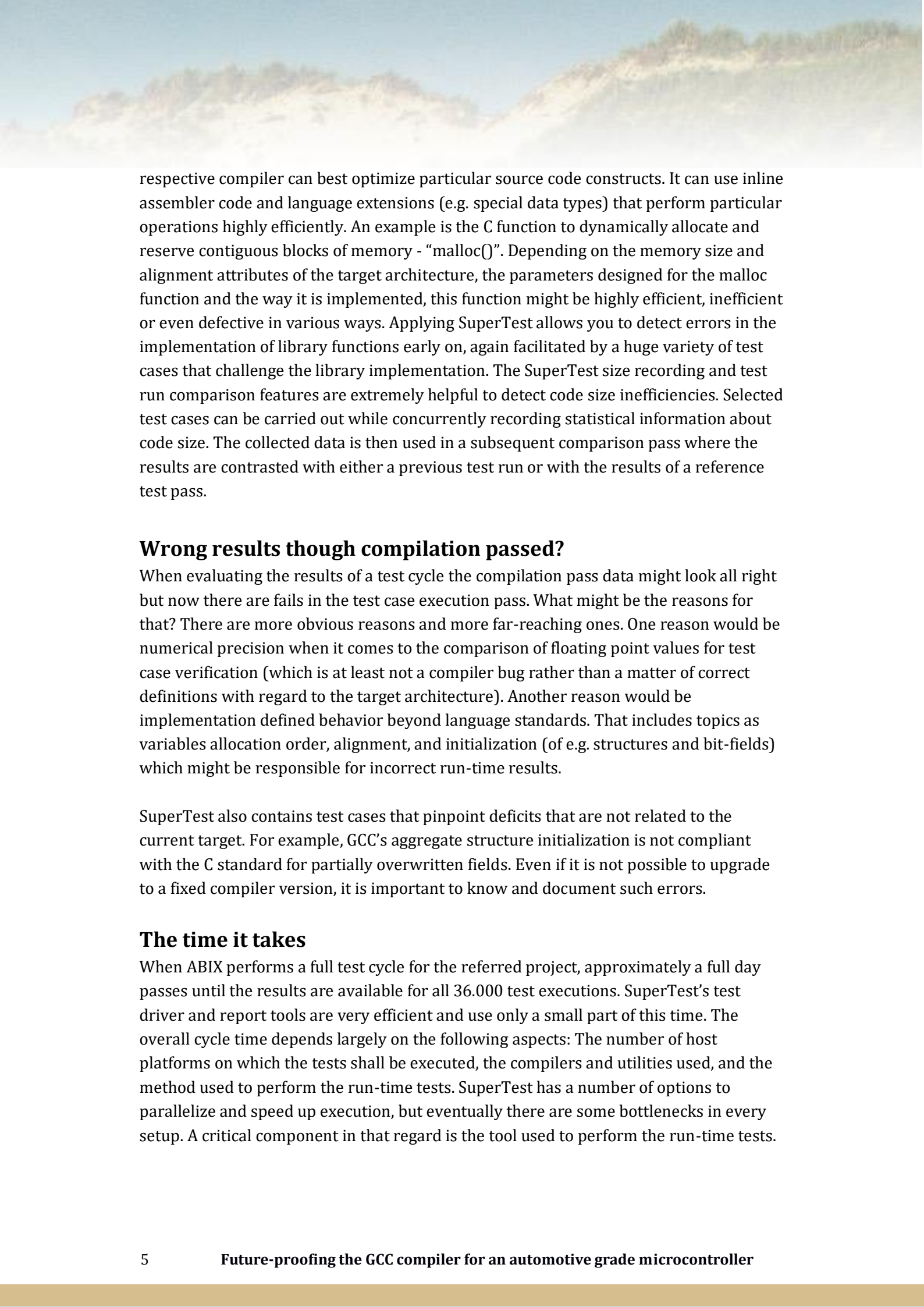
Give me more registers

“Internal compiler error: in extract_constrain_insn, at recog.c:2213”. Does that sound more informative than “segmentation fault”? This is the manifestation of a problem with register allocation. The challenge for register allocation is to place program variables into machine registers. The more active variables you have in the source code and the fewer registers the target architecture has, the more difficult this is for the compiler.

SuperTest provides a rich set of variable manipulations that exploit the potential of the source code and thus constitute challenges for the compiler’s register allocation. However, beyond detecting the error, the compiler engineer is still challenged with finding the actual root of the problem. That is generally due to the branched and interconnected nature of analyses, rules, transformations, and optimizations in the compiler, and the many paths through this network.

Oh, your libraries

Is a test suite limited to language conformity, data types, calling conventions, register handling, etc.? Not at all. For a C compiler it is very common and also good practice to provide a set of libraries whose functions have been specifically optimized for the respective target architecture. This is achieved by using knowledge about how the



respective compiler can best optimize particular source code constructs. It can use inline assembler code and language extensions (e.g. special data types) that perform particular operations highly efficiently. An example is the C function to dynamically allocate and reserve contiguous blocks of memory - “malloc()”. Depending on the memory size and alignment attributes of the target architecture, the parameters designed for the malloc function and the way it is implemented, this function might be highly efficient, inefficient or even defective in various ways. Applying SuperTest allows you to detect errors in the implementation of library functions early on, again facilitated by a huge variety of test cases that challenge the library implementation. The SuperTest size recording and test run comparison features are extremely helpful to detect code size inefficiencies. Selected test cases can be carried out while concurrently recording statistical information about code size. The collected data is then used in a subsequent comparison pass where the results are contrasted with either a previous test run or with the results of a reference test pass.

Wrong results though compilation passed?

When evaluating the results of a test cycle the compilation pass data might look all right but now there are fails in the test case execution pass. What might be the reasons for that? There are more obvious reasons and more far-reaching ones. One reason would be numerical precision when it comes to the comparison of floating point values for test case verification (which is at least not a compiler bug rather than a matter of correct definitions with regard to the target architecture). Another reason would be implementation defined behavior beyond language standards. That includes topics as variables allocation order, alignment, and initialization (of e.g. structures and bit-fields) which might be responsible for incorrect run-time results.

SuperTest also contains test cases that pinpoint deficits that are not related to the current target. For example, GCC's aggregate structure initialization is not compliant with the C standard for partially overwritten fields. Even if it is not possible to upgrade to a fixed compiler version, it is important to know and document such errors.

The time it takes

When ABIX performs a full test cycle for the referred project, approximately a full day passes until the results are available for all 36.000 test executions. SuperTest's test driver and report tools are very efficient and use only a small part of this time. The overall cycle time depends largely on the following aspects: The number of host platforms on which the tests shall be executed, the compilers and utilities used, and the method used to perform the run-time tests. SuperTest has a number of options to parallelize and speed up execution, but eventually there are some bottlenecks in every setup. A critical component in that regard is the tool used to perform the run-time tests.



For instance, many simulators cannot run multi-instance, and that forces the serialization of test execution. Although test cases in SuperTest are slim and streamlined, the performance of the simulation and/or run-time execution tool adds significantly to the overall processing time. One way to improve on bottlenecks like this is to distribute the test cycle to multiple real or virtual machines.

The big picture

All of the aforementioned portrayals might appear like individual considerations but are actually reflecting ABIX' experiences gathered during a challenging GCC compiler upgrade and improvement project. A project like this requires two primary ingredients, the experts knowing their soil – who ABIX can provide for your compiler project, and a comprehensive, industrial-strength test suite that includes all the knowledge about why, how, and where compilers might swerve from the racing line – for which we highly recommend Solid Sands' SuperTest. SuperTest leaves almost nothing to chance and has been an essential tool to help us create a robust and reliable compiler for our customers.

(c) Copyright 2021 by Solid Sands B.V., Amsterdam, the Netherlands
SuperTest™ is a trademark of Solid Sands B.V., Amsterdam, the Netherlands
All other trademarks herein are the property of their respective owners