Document number: N4049

Date: *2014-05-27*

Project: Programming Language C++, Evolution Working Group

Reply-to: Daniel Gutson <daniel.gutson@tallertechnologies.com>,
Angel Bustamante <angel.bustamante@tallertechnologies.com>*”,*
Pablo Oliva <pablo.oliva@tallertechnologies.com>,
Marcos Diaz <marcos.diaz@tallertechnologies.com>*”*

0-overhead-principle violations in exception handling

I. Motivation

Sometimes the 0-overhead principle is not honored; one has to pay for things that are not used. An instance where this can be observed is the Exception Handling mechanisms implemented by the diverse toolchains.
In an average computing environment, this has little or no importance; the overhead is minimal and its impact is negligible.
But on embedded systems, where memory is scarce, the overhead imposed by unused language constructs can exceed the capabilities of the available hardware, as we will show is the case for exceptions.

In this document, we analyze the reasons why the 0-overhead principle doesn’t hold regarding memory usage. Our intent is to generate a careful discussion about the possible ways to deal with this issue and the need for a standardized solution, since QoI alone cannot solve it. This document’s scope is limited to the discussion alone; the analysis of proposals to handle the problems that will be presented is left for future work.
The decision of whether or not exceptions should be used is a valid choice that programmers should be allowed to make freely. And if a choice is made to avoid exceptions, their inherent memory costs should have no impact on the program.

II. Introduction

Any exception implementation mechanism incurs in penalties in performance, RAM and ROM usage.
The performance penalty that comes with exceptions can be avoided by not making use of them. Thus, in this document we will focus on the memory penalties that come with exception use, since both for RAM and for ROM, the 0-overhead principle does not hold; programs that avoid exception usage still have memory costs associated with them, as we will analyze and show in a case study. In that case study, the exceptions' memory costs could not be avoided just by setting compilation parameters; instead, toolchain modifications were also necessary to address them thoroughly.
We will focus on the RAM memory costs, and leave code memory considerations for a separate document; however, the conclusion and root cause discussed in this document also hold for code memory.

As an example, we will present a case study where the RAM memory impact of exception handling mechanisms is so significant that it hindered the development of C++ applications for a given microcontroller.

For additional information on the impact of Exceptions Handling on performance and memory, the reader can look at EH implementation guidelines in [PerformanceTR].

III. Why the 0-overhead-principle cannot be honored: don't blame QoI.

a) Distinguishing between user code and standard library code is tricky.
The STL uses exceptions. Since the CPP produces a single compilation unit which may include both standard header files and user files, and standard header files contain classes with methods that throw exceptions, a simple analysis of the compilation unit looking for exceptions-related code tells nothing of the programmer's will to use exceptions.

b) Although unused template functions are guaranteed not to generate code, avoiding all exception-throwing functions of the STL is costly.
If some subset of the STL is not intended to be used due to the EH generated code, extra work is necessary in order to re-implement functions, like *sqrt*(), which may throw std::domain_error.

c) The boundary between user code and library code is hard to establish.
Even without considering the STL, the implementation may still use other libraries that have exceptions. And if the toolchain could distinguish between user and library code, including headers or linking to libraries that do in fact use exceptions turns the analysis of exception uses into an extremely difficult, if not unfeasible, task.

d) Toolchain optimizations are not enough to suppress exception handling overhead.
STL implementations usually come equipped with buffers, functions and global objects for supporting exceptions. Even if exceptions are not present in the user code, the dead code elimination [DCE] and link time optimization [LTO] that some toolchains provide cannot always eliminate exception handling code, since that code might be needed by a library that uses exceptions. Whole program optimization [WPO] cannot always determine whether code that uses exceptions is actually called, since late binding breaks the static call-trace analysis of the inter-procedural optimizations.

IV. How is this issue being currently addressed?

Toolchains such as GNU provide command line flags to disable exceptions (i.e. *-fno-exceptions*). As of the writing of this paper, this flag only affects the front-end [GCCManual] but it has no impact in the libraries (neither the STL nor low level libraries, such as libgcc): the library's binary, that is linked to the program, may have exceptions enabled and require exception handling resources. Therefore, this workaround does not solve the problem.
Some extra work is being done by the authors of this paper to make this flag affect the back-end of the toolchain and the libraries' selection mechanism, so pre-built libraries (with and without EH support) can be conditionally linked to, depending on the flag that tells if exceptions are used. This specific flag (to be passed as an argument to the toolchain) is required because inferring exception avoidance from code analysis alone is not feasible, as seen in section III.

V. Case study: GNU Toolchain, RTEMS on ARM with 32K RAM

The case study scenario is the development of programs for a modified [RTEMS] operating system running on the MBED LPC1768 microcontroller, a device with an ARM CORTEX-M3 core, 512KB ROM and 32 KB RAM [MBED].

The GNU toolchain was used, with the following component versions:

- GCC: versions 4.8.1 and 4.8.2 (no changes observed for our analysis).
- LIBC: newlib 2.0.0 - 2.1.0 [1] (no changes observed for our analysis).
- STL: libstdc++-v3. (at first the one distributed with gcc 4.8.1, and then the one distributed with gcc 4.8.2).
- Assembler and Linker: binutils 2.23.2 - 2.24 (no changes observed for our analysis).

The operating system (RTEMS) version was 4.11, with a new BSP implemented for this board.

The first approach was to create C programs on this platform; eventually, a decision was reached to start creating C++ programs. It was decided not to use exceptions due to the

[1] Despite the fact that this component does not belong to the GNU toolchain, from now on it will be assumed that the GNU toolchain includes newlib instead of eglibc for brevity purposes only.

memory overhead they impose, as will be shown later (see also [EASTL] for a discussion about exceptions in embedded environments).

On the first attempts to use RTEMS with C++ for small examples, the memory usage was so large that the programs would not run at all. A minimalistic C++ program was able to run, but it had a much larger RAM memory footprint (about 20 KB) than its C counterpart.
Doing some research on the causes of this overhead, two large buffers (*dependents_buffer* and *emergency_buffer*) in the BSS section of the program were found, with a combined size of almost 20 KB of RAM (which amounts to 63% of the available memory on this device). These two buffers are used by the exception handling implementation, in order to save space for exception control (the object being thrown, the local housekeeping state and the language-independent unwinding control block) [ARM EHABI]. The *emergency_buffer* contains reserved memory which can be used when attempting to throw an exception if the heap is exhausted. The *dependents_buffer* is the same but for ‘dependent exceptions’ [DEPENDENT_EXCEPTIONS].
Also, two global objects of types *__cxa_eh_globals* and *__eh_globals_init* were found (‘eh_globals’ and ‘init’); they are used to keep a registry of the caught and uncaught exceptions of a thread [ARM EHABI].
Although these two do not use much memory, they have global destructors that are registered via *atexit*; this causes a 400 Bytes RAM overhead per thread[2] for *atexit*’s internal structure (_atexit0).
No involvement of the Operating System code has been observed in the exception handling support, and there was neither any participation by newlib’s code in EH. Only toolchain (except newlib) code managed exceptions.

Workaround

A first attempt to reduce the memory footprint was to compile both the program and RTEMS with the ‘*-fno-exceptions*’ flag. This was not enough, and the RAM consumption showed no variation; the Standard Library had to be recompiled using the no exceptions flag to eliminate the two buffers. After the recompilation, the memory overhead was reduced, but the ‘eh_globals’ and ‘init’ objects were not removed.

Modifications had to be performed on both the C++ low-level library (libgcc) and the STL implementation (libstdc++), in order to manually remove the exception handling code and resources, so that the memory overheads could be avoided. We had to explicitly cut off the buffers, functions and global objects that belong to the exceptions’ implementation at linking and compiling time in order to delete code that wouldn’t be used (for instance, using the *fwhole-program* and *gc-sections* flags).

[2] This analysis of memory costs was performed using the OS provided threads, not the Standard C++11 threads.

VI. Conclusions

The toolchain cannot detect (without explicit directives) whether the user actually uses exceptions or not in order to honor the *0-overhead-principle*.
As a consequence, the target application is generated in a suboptimal manner; this may limit the range of devices on which the program can run, preventing the use of C++ for most low-cost embedded platforms.
With the increasing number of devices that the IoT trend will make available for developers [Gartner-IOT] [ABIR], providing language support for the resource constraints of those devices (see [ARMIOT]) is critical to make C++ one of the pillars of a market which is expected to grow at one of the fastest paces in the world of technology [GartnerTop10].

VII. Acronyms

BSP: Board Support Package, see [RTEMS-BSP]
BSS: Block Started by Symbol, see [Unix-BSS]
CPP: C Pre-processor.
EH: Exception Handling.
IoT: Internet of Things.
QoI: quality of implementation.
STL: Standard Template Library.

VIII. References

- [PerformanceTR] Technical Report on C++ Performance
- [EHControl] GCC - Exception Handling Control
- [GCCManual] Libstd++ GCC Manual
- [RTEMS] RTEMS
- [RTEMS-BSP] RTEMS BSP
- [MBED] Mbed LPC1768
- [Unix-BSS] Unix FAQs: BSS
- [DCE] GNU Optimizations - Dead Code Elimination
- [LTO] Link Time Optimization
- [ARM-EHABI] Exception handling ABI for the ARM architecture. Sections 8.4.1 and 12.2
- [DEPENDENT_EXCEPTIONS] N2179: Exception Propagation- GCC Patch, Dependent_exceptions definition, see line 180.
- [Gartner-IOT] "Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020". Gartner. 2013-12-12. Retrieved 2014-01-02.

- [ABIR] More Than 30 Billion Devices Will Wirelessly Connect to the Internet of Everything in 2020, ABI Research
- [ARMIOT] "Today's Internet of Things rely on inexpensive, small sensors. They need to be power efficient, and be able to run on batteries that last for 10 years. They also need to be design efficient for integration into things that cost less than $1 (possibly as little as 50 cents)."
- [GartnerTop10] Gartner Identifies the Top 10 Strategic Technology Trends for 2014
- [EASTL] Electronic Arts Standard Template Library

IX. Acknowledgments

We wish to thank Fabio Bustos for his assistance reviewing the document and Bjarne Stroustrup for his useful feedback.