Document number: N4234

Date: 2014-10-10

Project: Programming Language C++, Evolution Working Group

Reply-to: *Daniel Gutson “daniel dot gutson at tallertechnologies dot com”*

0-overhead-principle violations in exception handling - part 2

I. Table of Contents

II. Introduction

This paper continues the research presented in N4049, with a new small research and its results trying to answer the following question:

If the programmer writes a program that uses the STL without using EH, can implementations honor the 0-overhead principle (i.e. not including any EH-related machinery in the generated binary) by using current optimizations, or there are points in the standard that prevents such honoring?

In other words, the research tries to determine whether implementations are capable to get rid of the exception handling machinery via some of the current optimizations (i.e. Whole Program Analysis, Link Time Optimization).

III. Motivation and Scope

Exception handling machinery can result in an undesirable overhead in some environments, such as embedded systems with low resources (specially RAM). Some overhead analysis was presented in N4049.

The programmer should be able to have the choice to not use exception handling, expecting the overhead to be null according to the 0-overhead principle.

Both the overhead analysis and whether not using exception handling is a good idea or not, is out of the scope of this paper.

This paper focuses in understanding whether the implementations have the ability to honor the 0-overhead principle or not, considering current optimizations. Rather than proposing changes to the Standard, the goal is to start a discussion based on the results and conclusions presented here.

Note: the same toolchain should be able to be used both in a program that uses EH and another that doesn't, and the 0-overhead principle should prevail. Using a specially-fitted STL implementation without EH support is not considered (e.g. an STL implementation built with the -fno-exceptions gcc flag) since EH is part of the language, and defining a subset of the language is not a desirable goal as it was for the Embedded C++ approach in the past.

IV. The experiment

The experiment consisted in writing four small programs (see figure 1) that use the STL, without any exception handling-related action, building it (using gcc and clang) with whole-program-analysis and link-time optimizations, and looking for exception handling-related symbols in the generated binary.

That is, the approach was to start from the application source code, then look for EH symbols in the binary, then correlate them in the STL implementation source code, and finally find the Standard relevant sections:

application source code → binary → STL impl. source code → Standard

The program #1 (figure 1) contained:

- an allocator definition that called ::operator new(nothrow)
- the main() function
- a call to vector::push_back which is not a noexcept method
- a (*strongly defined*) function to override the toolchain's default (which throws an exception thus adding a dependency to the EH mechanism). This function is called when a pure-virtual method is called, and is a documented implementation technique to cut the added EH dependency.

The program #2 was exactly as program #1 but using std::list instead of std::vector in the main() function.

The program #3 used also a list, but the Allocator called malloc instead of ::operator new(nothrow) in the Allocator::allocate method.

Finally, program #4 used a map with the Allocator calling malloc too.

The table 1 provides a summary of the experiments:

	Container	**Allocation Method**
program #1	vector	::operator new(nothrow)
program #2	list	::operator new(nothrow)
program #3	list	malloc
program #4	map	malloc

Table 1: summary of the experiments

Since the results of the analysis were equivalent both toolchains, details of the GNU toolchain only are presented here.

Materials & Methods

The program #1 was:

```
#include <vector>
#include <new>
#include <limits>
#include <stdio.h>
using namespace std;

template<typename T>
class Allocator {
public :
    typedef T value_type;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef std::size_t size_type;
    typedef std::ptrdiff_t difference_type;

    template<typename U>
    struct rebind {
        typedef Allocator<U> other;
    };

    Allocator() = default;
    Allocator(const Allocator&) = default;
    template<typename U>
    inline Allocator(const Allocator<U>& other)
    {
    }

    inline pointer address(reference r) { return &r; }
    inline const_pointer address(const_reference r) { return &r; }

    inline pointer allocate(size_type cnt,
       typename std::allocator<void>::const_pointer = 0)
    {
      return reinterpret_cast<pointer>(::operator new(cnt * sizeof (T),
nothrow));
    }

    inline void deallocate(pointer p, size_type)
    {
        ::operator delete(p);
    }

    inline size_type max_size() const
    {
        return std::numeric_limits<size_type>::max() / sizeof(T);
    }

    inline void construct(pointer p, const T& t) { new(p) T(t); }
    inline void destroy(pointer p) { p->~T(); }

    inline bool operator==(Allocator const&) const { return true; }
    inline bool operator!=(Allocator const& a) const { return
!operator==(a); }
};    //    end of class Allocator


extern "C" {
void __cxa_pure_virtual()
{
    fprintf(stderr, "pure method called!\n");
}
}

int main()
{
    vector<int*, Allocator<int>> v;
    v.push_back(new (nothrow) int(1));
}
```
Figure 1: the program used

The target platform was Linux x86_64 (Ubuntu 14.04).
The GNU toolchain was used with the following command line:

```
g++ -std=c++11 -fwhole-program -flto -static -Wl,--gc-sections test.cpp
```

Note: the use of -fno-exceptions made no difference (as explained below), and in any case in an ideal situation, the toolchain should not need to receive "hints" (nothing outside the program).

The litmus test was as follow:

```
nm a.out | c++filt | grep excep
```

The disassembly was performed with the following command line:

```
objdump -d a.out | c++filt
```

V. Results

The generated binary from program #1 contained EH-related symbols, despite the WPA and LTO optimizations:

```
0000000000402db0 T __cxa_allocate_exception
0000000000403980 T __cxa_current_exception_type
0000000000402ea0 T __cxa_free_exception
00000000006c7088 V DW.ref._ZTISt9exception
0000000000401e70 t check_exception_spec(lsda_header_info*, std::type_info
const*, void*, long)
00000000004026a0 t __gxx_exception_cleanup(_Unwind_Reason_Code,
_Unwind_Exception*)
00000000004028c0 T std::bad_exception::what() const
00000000004028b0 T std::exception::what() const
00000000004028f0 T std::bad_exception::~bad_exception()
0000000000402890 T std::bad_exception::~bad_exception()
0000000000402890 T std::bad_exception::~bad_exception()
00000000004028d0 T std::exception::~exception()
0000000000402880 T std::exception::~exception()
0000000000402880 T std::exception::~exception()
000000000040184d t std::move_iterator<int**>
std::__make_move_if_noexcept_iterator<int**, std::move_iterator<int**>
>(int**)
0000000000401742 t int** std::__uninitialized_move_if_noexcept_a<int**,
int**, Allocator<int*> >(int**, int**, int**, Allocator<int*>&)
00000000006c6a50 V typeinfo for __cxxabiv1::__foreign_exception
00000000006c6a20 V typeinfo for std::bad_exception
00000000006c6a10 V typeinfo for std::exception
0000000000498200 V typeinfo name for __cxxabiv1::__foreign_exception
00000000004981c0 V typeinfo name for std::bad_exception
00000000004981a6 V typeinfo name for std::exception
00000000006c6aa0 V vtable for std::bad_exception
00000000006c6a60 V vtable for std::exception
```

Figure 2: output of the 'nm' command showing EH-related symbols from program #1

A call graph to the function __cxa_allocate_exception (defined in the libc++ ABI) was performed from the generated binary:

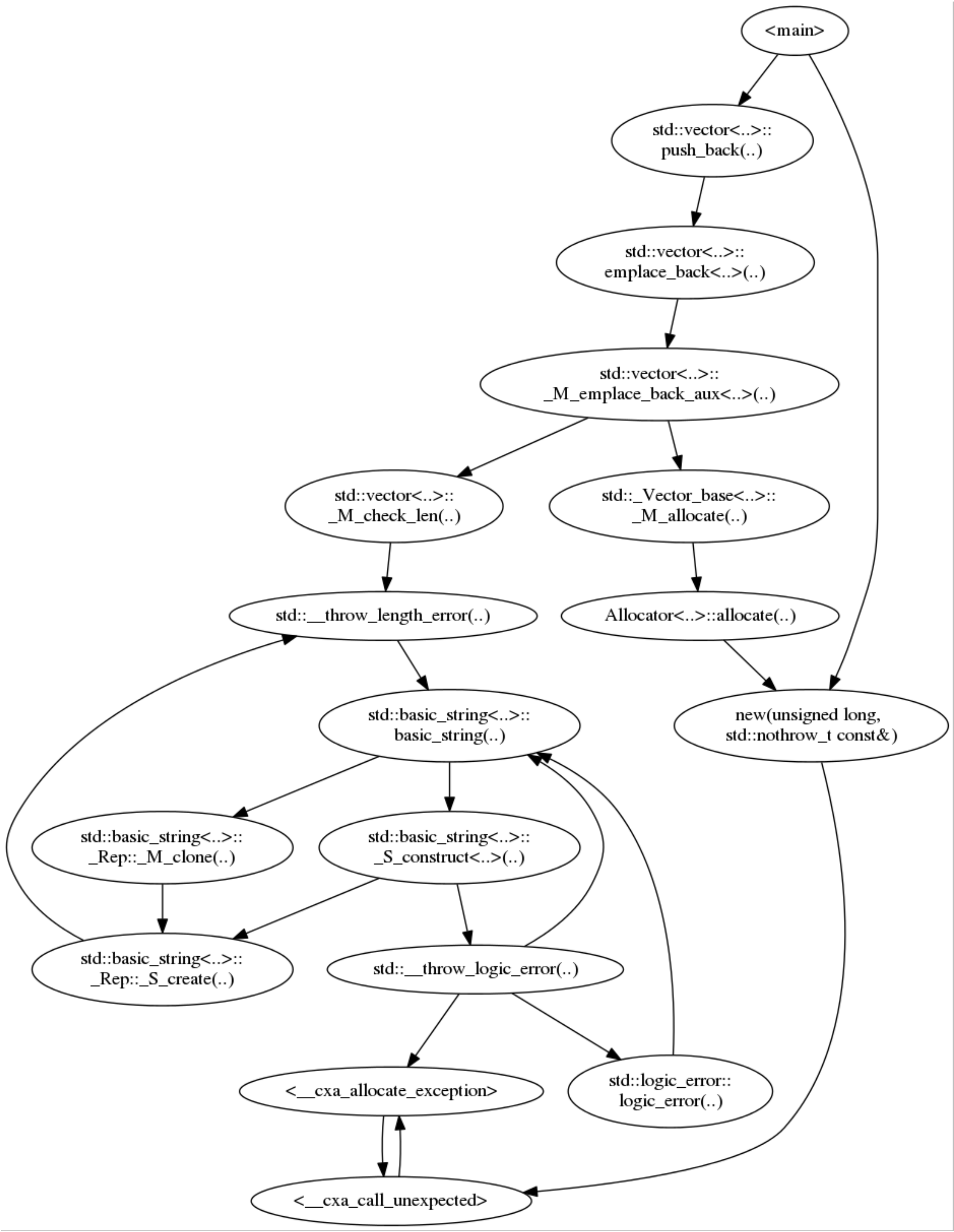
<main>
std::vector<..>:: push_back(..)
std::vector<..>:: emplace_back<..>(..)
std::vector<..>:: _M_emplace_back_aux<..>(..)
std::vector<..>:: _M_check_len(..)
std::_Vector_base<..>:: _M_allocate(..)
std::__throw_length_error(..)
Allocator<..>::allocate(..)
std::basic_string<..>:: basic_string(..)
new(unsigned long, std::nothrow_t const&)
std::basic_string<..>:: _Rep::_M_clone(..)
std::basic_string<..>:: _S_construct<..>(..)
std::basic_string<..>:: _Rep::_S_create(..)
std::__throw_logic_error(..)
<__cxa_allocate_exception>
std::logic_error:: logic_error(..)
<__cxa_call_unexpected>

Figure 3: call graph of __cxa_allocate_exception (program #1, using std::vector)

As figure 3 shows, an internal method of the STL implementation (_M_check_len) calls a function that finally throws a length_error exception (see §23.3.7.3 [vector.capacity]). Additionally, the operator new(nothrow) uses EH in order to catch the exceptions thrown by the new_handler (as shown in figure 4).

```
_GLIBCXX_WEAK_DEFINITION void *
operator new (std::size_t sz, const std::nothrow_t&) _GLIBCXX_USE_NOEXCEPT
{
  void *p;

  /* malloc (0) is unpredictable; avoid it.  */
  if (sz == 0)
    sz = 1;

  while (__builtin_expect ((p = malloc (sz)) == 0, false))
    {
      new_handler handler = std::get_new_handler ();
      if (! handler)
        return 0;
      __try
      {
        handler ();
      }
      __catch(const bad_alloc&)
      {
        return 0;
      }
    }

  return p;
}
```

Figure 4: GNU implementation of new(nothrow) showing the EH dependency

There are also two STL global objects (system_category_instance and generic_category_instance) that also use EH but are not shown in the graphs.

The binary generated from program #2 (the one that uses std::list) only contained the EH symbols which come from the ::operator new(nothrow) mentioned above. The call graph of the generated binary is shown in figure 5:

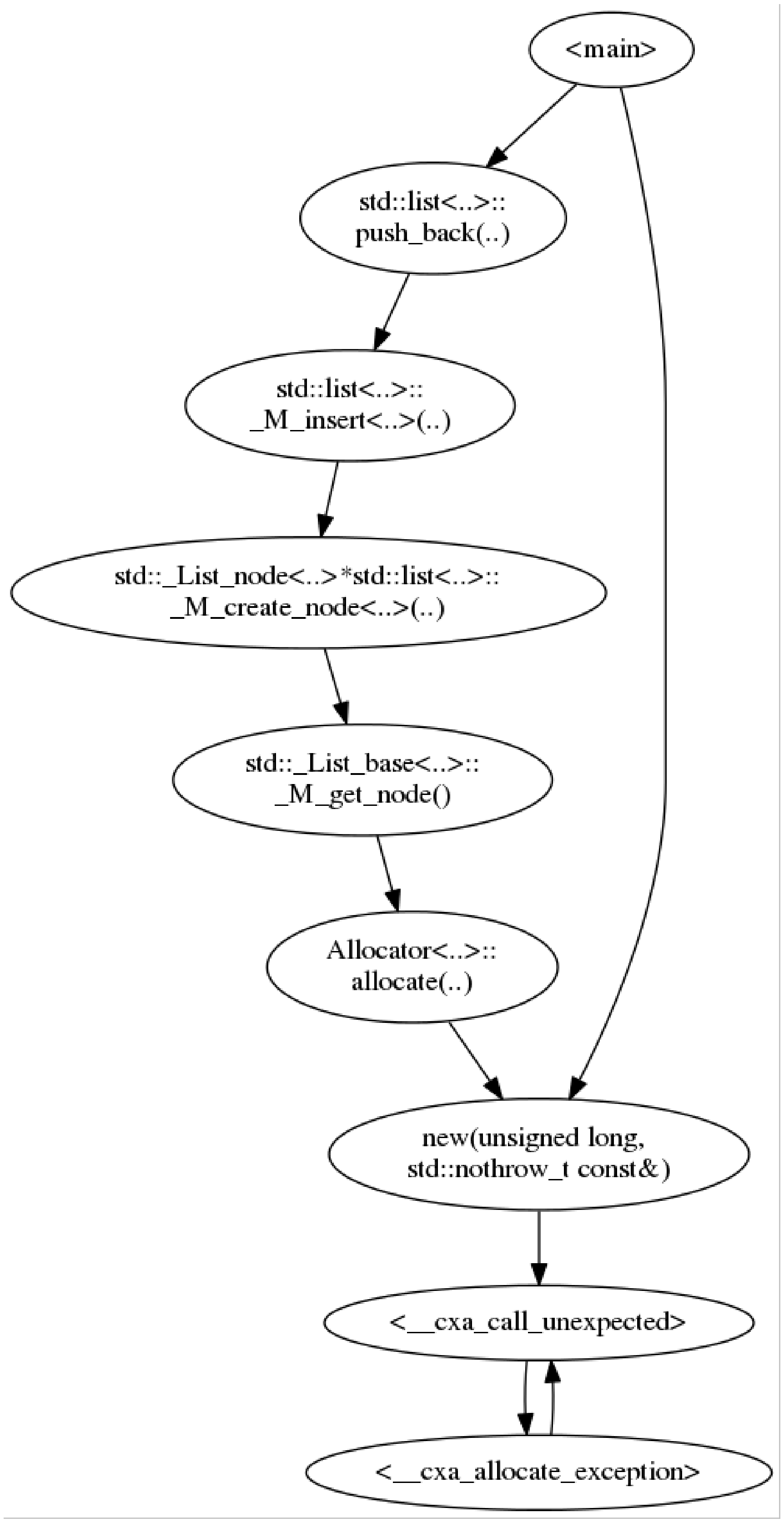

Figure 4: call graph of __cxa_allocate_exception (program #2, using std::list)

Finally, the binaries generated from program #3 and #4 (those using malloc) did not have any EH symbols at all.
Table 2 summarizes the results:

	Container	Allocation Method	EH symbols

program #1	vector	::operator new(nothrow)	from §23.3.7.3
			from ::operator new (nothrow)

program #2	list	::operator new(nothrow)	from ::operator new (nothrow)
program #3	list	malloc	none
program #4	map	malloc	none

Table 2: results summary

VI. Conclusions

- Given the analyzed results, the application developer can achieve 0-overhead principle in terms of EH by providing a custom allocator with malloc/free as the allocation primitives, except for std::vector
- From the three analyzed containers (vector, list, and map) only vector forced a violation to the 0-overhead principle when calling vector::push_back (even when the constructor/move/assignment of the value_type don't throw exceptions).
- operator new(nothrow) pulls-up the EH machinery since it has to catch the exceptions thrown by the new_handler callback.

VII. Possible solution approaches

Regarding the check-length of vector

- STL-specific approaches:
  - add nothrow/noexcept versions of the non-noexcept methods (such as vector::push_back, emplace_back) with an error return code, e.g.:
    ```
    void push_back (const value_type& val, nothrow_t) noexcept;
    void push_back (value_type&& val, nothrow_t) noexcept;
    ```
  - add a new noexcept equivalent version of the push_back methods but named "push/emplace_back_no_capacity_check" or alike
  - turn the check optional in the standard
- C++-generic approaches:
  - (sketchy) Make the 'noexcept' qualifier applicable to pointers (a la cv-qualifiers) so calling methods to a noexcept-declared pointer will invoke the noexcept-version of the methods, allowing overloading distinguished by the noexcept qualifier.

Regarding the ::operator new(nothrow) issue

A non throwing new_nothrow_handler callback (with its setter and getter functions) could be provided in the standard library:

```
typedef void (*new_nothrow_handler)() noexcept;
new_nothrow_handler set_new_nothrow_handler (new_nothrow_handler new_p)
noexcept;
new_nothrow_handler get_new_nothrow_handler() noexcept;
```

Additionally, provide an opt-in standard std::nothrow_allocator that use the nothrow version of new.

VIII. Future Work

More research should be done for the rest of the STL containers.
Write the proposals to address the issues presented in this report.

IX. Acknowledgements

I thank Federico García Cruz for the disassembly and call graph generation. I also thank the members of the "C++ for Embedded and Real Time systems" mailing list in general, and to Jens Maurer and Martin Sebor in particular for their useful feedback.

X. Acronyms

EH	-	Exception Handling
LTO	-	Link-Time Optimizations
STL	-	Standard Template Library
WPA	-	Whole Program Analysis