Low-cost Deterministic C++ Exceptions for Embedded Systems

Citation for published version:

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 28th International Conference on Compiler Construction (CC2019)

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Low-Cost Deterministic C++ Exceptions for Embedded Systems

James Renwick
University of Edinburgh
Edinburgh, UK
s1227500@sms.ed.ac.uk

Tom Spink
University of Edinburgh
Edinburgh, UK
tspink@inf.ed.ac.uk

Björn Franke
University of Edinburgh
Edinburgh, UK
bfranke@inf.ed.ac.uk

ABSTRACT
The C++ programming language offers a strong exception mechanism for error handling at the language level, improving code readability, safety, and maintainability. However, current C++ implementations are targeted at general-purpose systems, often sacrificing code size, memory usage, and resource determinism for the sake of performance. This makes C++ exceptions a particularly undesirable choice for embedded applications where code size and resource determinism are often paramount. Consequently, embedded coding guidelines either forbid the use of C++ exceptions, or embedded C++ tool chains omit exception handling altogether. In this paper, we develop a novel implementation of C++ exceptions that eliminates these issues, and enables their use for embedded systems. We combine existing stack unwinding techniques with a new approach to memory management and run-time type information (RTTI). In doing so, we create a compliant C++ exception handling implementation, providing bounded runtime and memory usage, while reducing code size requirements by up to 82%, and incurring only a minimal runtime overhead for the common case of no exceptions.

CCS CONCEPTS
• Software and its engineering → Error handling and recovery; Software performance; Language features;

KEYWORDS
C++, exceptions, error handling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CC ’19, February 16–17, 2019, Washington, DC, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6277-1/19/02...
https://doi.org/10.1145/3302516.3307346

1 INTRODUCTION
One of the major benefits of C++ is its promise of zero-cost abstractions and its rule of “you don’t pay for what you don’t use”. This allows the use of various safer and more productive modern programming idioms, without the performance drawbacks with
Figure 1: Example code showing a possible scenario in which exceptions are used. Function $A$ has a try/catch block, that encompasses the call to $B$. $B$ allocates a local object, and calls $C$, which also allocates a local object, and throws an exception.

void $C()$
{
    Bar bar;
    throw 0;
}
void $B()$
{
    Foo foo;
    $C()$;
}
void $A()$
{
    try {
        $B()$;
    } catch (int& p) {
        // Catch exception
    }
}

The most common implementation of C++ exception handling, in use by both e.g. GCC and Clang, makes use of table-based stack unwinding. It seeks to eliminate any runtime overhead in the case where no exceptions are thrown. This is achieved by storing abstract instruction sequences required for identifying the catch handler, restoring the call stack and machine state, and invoking object destructors in Unwind Tables, which are generated at compile-time and embedded in the final program binary. When an exception is thrown, the exception object is allocated on the heap, and a 2-stage process for stack unwinding is started: The first stage, shown in Figure 2, requires an initial phase to scan over the call stack, looking for a catch handler. This phase does not unwind the stack, it identifies how far back the stack should be unwound.

1.2 Contributions

In this paper we contribute to the development of a novel C++ exception implementation particularly suitable for use in embedded systems. We address following issues:
(1) Reducing the binary size increase caused by exceptions.
(2) Permitting the bounding of memory usage and execution time when using exceptions, i.e. supporting determinism.
(3) Maintaining or improving the performance of exceptions.

Although we focus on embedded systems, we will ensure that these goals also apply to general-purpose applications.

2 DETERMINISTIC C++ EXCEPTIONS

Stack unwinding forms a core part of the mechanisms underpinning exceptions. It is also responsible for its poor execution times and increased binary sizes, and is part of the reason for exceptions being non-deterministic.

Sutter [19] has only very recently proposed re-using the existing function return mechanism in place of the traditional stack unwinding approaches, requiring no additional data or instructions to be stored, and little to no overhead in unwinding the stack. Furthermore, by removing stack unwinding’s runtime reliance on tables encoded in the program binary itself, the issue of time and spatial determinism is solved. As it is possible to determine the worst-case execution times for programs not using exceptions, it follows that exception implementations making use of the same return mechanism must also be deterministic in stack unwinding.

Given these clear advantages, we have based our implementation on this design, with a core difference being the replacement of their use of registers with function parameters, allowing for much easier interoperability with C code, which can simply provide the parameter as necessary.

A limitation with [19] is that they require all exceptions be of the same type, leaving much of the standard exception-handling functionality up to the user. Our novel approach includes a method of throwing and catching exceptions of arbitrary types (as with existing exception handling), without imposing any meaningful execution-time penalties when exceptions do not occur. We also reduce the size of run-time type information (RTTI), and maintain determinism over existing implementations.

Our scheme makes use of a stack-allocated object that records the necessary run-time information for throwing an exception, such as the type and size of the exception object. This state is allocated in a single place and is passed between functions via an implicit function parameter injected into functions which support exceptions. The state is initialised by throw expressions, and is re-used to enable re-throwing. catch statements use the state in order to determine whether they can handle the exception. After a call to a function which may throw exceptions, a run-time check is inserted to test whether the state contains an active exception.

```
1 void foo(__exception_t& __exception) throws {
2     // Exception state '__exception' assigned to by throw
3     throw SomeError();
4 }
5
6 int main() {
7     // State automatically allocated in 'main'
8     __exception_t __exception_state;
9     foo(__exception_state);
10     // check for exception
11     if (__exception_state.active) goto catch;
12 }
```

Figure 4: Pseudocode showing injected exception state object (line 7) and parameter (line 1)
Figure 4 gives an example of the variables the compiler will automatically inject during code generation. Line 1 shows the normally hidden implicit exception state parameter `__exception`. Line 3 assigns values corresponding to the `SomeError` type to the exception state parameter. Line 7 shows the automatically allocated `__exception_state` variable, emitted by all `noexcept` functions. Line 8 shows the exception state object being automatically passed to the `throws` function `foo`. Line 10 shows the check inserted after each call to a throwing function to test for an active exception.

The design of this implementation performs almost all of the exception handling directly within the functions being executed, allowing the optimizer to work most effectively, and by implementing exceptions on top of normal execution flow; code used in returning from functions is re-used when unwinding the stack following exceptions. This helps to reduce code size and unpredictability, and overcomes the largest roadblock in achieving deterministic execution time. This approach combines the exception-handling mechanism of C++ with the inter-mixed error-checking and non-error-handling code integral to the design of [9].

### 2.1 Throws Exception Specifier

A fundamental issue is to decide which functions should take the implicit exception parameter that our design requires (i.e. which functions could throw exceptions). The natural choice would be to apply it to all functions with C++ linkage, except for those marked `noexcept`, in keeping with the current exception implementation. However, the issue with this approach would be that C++ and C functions would then have different calling conventions. Whilst C++ functions can in general be called without special handling from C, this is only due to the coincidence that the two calling conventions are identical.

With our proposed scheme, the decision to have C++ functions support exceptions by default is impractical, as C++ has certain language holes when it comes to specifying linkage. Specifically, neither classes, structs nor their members, such as function pointers, can have a specified linkage. Therefore, we propose to have functions be declared `noexcept` by default. This change in default exception specification preserves compatibility with C at the cost of a language change in C++.

Like `noexcept`, we require functions that `might` throw encode this into their signature (similar to [19]), as this impacts on how they are called. Thus, to indicate that functions can throw exceptions, we introduce an exception specifier (that is part of the function prototype), as shown in Figure 5.

![Figure 5: Example function marked with our proposed throws exception specifier (line 1) allowing exception propagation](image)

**Table 1: Exception State Fields**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>The &quot;type id&quot; variable address identifying the exception object’s type</td>
</tr>
<tr>
<td>base_types</td>
<td>Pointer to an array of &quot;type id&quot;s identifying the exception object’s base class types</td>
</tr>
<tr>
<td>ptr</td>
<td>Whether the exception object is a pointer to a non-pointer type</td>
</tr>
<tr>
<td>size</td>
<td>The size, in bytes, of the exception object</td>
</tr>
<tr>
<td>align</td>
<td>The alignment, in bytes, of the exception object</td>
</tr>
<tr>
<td>ctor</td>
<td>The address of the object’s move or copy constructor</td>
</tr>
<tr>
<td>dtor</td>
<td>The address of the object’s destructor</td>
</tr>
<tr>
<td>buffer</td>
<td>The address of the exception object</td>
</tr>
<tr>
<td>active</td>
<td>Whether the exception is currently active</td>
</tr>
</tbody>
</table>

### 2.2 Throwing Exceptions and Exception State

The `exception state object` exists to hold run-time information on the exception currently in progress. Every function marked `noexcept`, including the program main function, and functions used as thread entry-points, allocates and initialises its own exception state object, effectively making `noexcept` functions boundaries beyond which exceptions cannot propagate.

The address of this object is passed to any of the `called` functions that are marked `throws`, allowing the exception to propagate down the call chain as far as the first `noexcept` function, where, if not handled, the program is terminated.

Following a `throw` expression, the fields in the exception state object are populated at the `throw`-site with values known at compile-time corresponding to the exception object given in the expression. The active flag is set to true, the exception object is moved into a buffer, and its address assigned to the `buffer` field of the exception state.

If the `throw` expression is directly within a `try` statement, the code jumps to the first catch block for handling. Otherwise, provided the current function is marked as `throws`, the exception is propagated; execution is transferred to the function epilogue as if returning normally, but the return value for the function, if any, is not initialised.

During the return sequence, local objects have their destructors executed as if a standard function return had occurred, efficiently unwinding the stack. The exception being marked active is not made visible to destructors, as destructors are necessarily `noexcept` in this context ("If a destructor called during stack unwinding exists with an exception, std::terminate is called (15.5.1)"[17]). In this way, the exception state is untouched until entering the initial catch block. If the current function is not marked as `throws`, propagation is replaced by an immediate call to `std::terminate` to terminate the program.

After each function call to a function marked as `throws`, a check is automatically inserted by the compiler to test whether the active flag has been set, and thus whether the exception has been propagated. This is the largest overhead as a result of our changes - requiring a test for each function called regardless of whether an
exception was thrown or not. There is perhaps a good case to be made for its necessity, however. By indicating that a function can throw, via throws, the developer is encoding into the function signature the fact that an error may occur during its execution. Thus regardless of performance requirements, some test must be made for that error in a correct program.

### 2.3 Handling Exceptions

Once an exception has been thrown, and if it does not result in a call to \texttt{std::terminate}, control is transferred to the first available catch block.

Before each catch block executes, it first compares for value equality the “type id” of the exception type it handles, and the type marked in the type field of the exception state. If they are equal, or if the type the catch block handles is a base type of the type in the type field, then the catch block is entered. Otherwise, execution jumps to the next catch block, and repeats this process until the catch block is a catch-all, or all blocks have been exhausted. If no blocks remain, the exception continues propagation as described previously.

Once a catch block is executed, the active flag of the exception state is cleared, memory is allocated on the stack for the exception object, and the move or copy constructor is invoked to transfer ownership of the exception object into the current block.

If an exception was to occur during the move or copy operation, \texttt{std::terminate} is called to satisfy the C++ specification, which states “If the exception handling mechanism, after completing evaluation of the expression to be thrown but before the exception is caught, calls a function that exits via an exception, \texttt{std::terminate} is called (15.5.1).”

Catch filters can take two forms: (1) by value, where the catch variable represents an object allocated within the catch block, or (2) by reference, where the catch variable is a reference to the exception object allocated elsewhere.

In the latter case, an additional unnamed variable is introduced in which to store the exception object, and its allocation and initialisation is performed as described above using the equivalent non-reference type. The catch variable reference is then bound to this unnamed variable. In this way, even when the exception object is caught by reference, it is still owned by the catch block.

Following a successful move/copy of the exception object, the destructor is called on the original exception object instance within the buffer. Naturally, as destructors are necessarily noexcept, were the destructor to exit with an exception, again \texttt{std::terminate} would be called.

With the catch variable successfully initialised, the catch block then commences execution. Once complete, execution moves to a point past the final catch block in the current scope and the catch variable is destroyed as usual.

### 2.4 Re-Throwing

One of the issues with maintaining and passing references to a single instance of exception state via the exception state parameter is that more than one exception may be active at a time, and crucially the initial exception may require to be re-thrown following the intermediary exception.
void store_exception(
    std::exception::allocator< std::exception::const char >* obj)
throws {
    // Move the exception object into a global
    result.exception = std::move(obj);
}
// ---------------------------------
exception will cause the stack to be unwound, as local object de-
structors will run as the stack is unwound, potentially overwriting
the exception object. Allocating the exception object on the heap
is a solution, but the allocation may result in system calls, which
(a) potentially have unbounded execution time, and (b) may fail
if memory is exhausted. As our target is to support deterministic
exceptions, standard heap allocation is not an option.

We propose a statically sized buffer using a simple constant-
time stack allocator for our exception object, with a heap-backed
fall-back upon buffer exhaustion for those applications who
cannot compute or do not require run-time determinism. The size
of this buffer will be given a default value by the ABI in use, but
will be optionally application-overridden at link-time via a linker
command-line parameter, to optimise or prevent any and all heap
allocation.

In the worst-case scenario, our solution matches the performance
of current exception implementations, as at least that of g++ uses a
similar buffer. However, as we free our exception objects as early
as possible, we expect usage of this buffer to be less than other
implementations, and with our novel ability to customise its size
(and in combination with earlier improvements), we uniquely offer
deterministic allocation.

3 STANDARDS COMPLIANCE

In this section we consider the compliance of our exception im-
plementation against the ABI specification, and the C++ standards
document.

3.1 ABI Compliance

Existing ABI specifications mandate the use of tables and manual
stack unwinding. However, since using the existing function return
mechanism is a superior method of achieving stack unwinding, our
implementation deviates entirely from their design. Therefore, we
will not evaluate our solution against any existing ABI specifi-
cations. A side effect of this is that our own ABI is significantly easier
to implement, as the stack unwinding and exception handling code
is generated in a platform-independent way by the compiler.

3.2 C++ Standard Compliance

In general, our implementation conforms to the C++ standard’s re-
quirements for exceptions. However, we have observed four clauses
where there are deviations:

\begin{verbatim}
bool type_is_base(void* type, const char** bases) noexcept {
    for (; bases[0] != nullptr; bases++) {
        if (static_cast<const void*>(bases[0]) == type)
            return true;
    }
    return false;
}
\end{verbatim}

Figure 8: Example iteration over base entries, the candidate
base is passed to the type parameter and the array of base
“type id”s to the bases parameter.

\begin{verbatim}
Clause 18.1.4. describes the interaction between the exception
object and std::exception_ptr, which is designed for exception
objects allocated on the heap. This is therefore not suited to our
stack-based implementation, however our replacement provides
similar functionality.

Clause 18.2.2. requires local temporaries, such as return values,
to have their destructors called when unwinding. However, clang’s
existing exception implementation does not conform to this part
of the specification, and as a result our implementation also does
not conform. This is a bug in the clang compiler (on which our
implementation is based), and if this is repaired, our implementation
will also be repaired.

Clauses 18.2.3 and 18.2.4. require the destruction of base class
instances and sub-objects already constructed when an exception
occurs during a constructor. This feature is not yet implemented,
and will be addressed in future work, however it does not require
any conceptual changes, as following from its simplicity and simi-
arity to local object destruction, we foresee no issues that might
occur in its implementation.

4 EVALUATION

We have implemented our novel scheme in the clang compiler, and
present results showing the performance of our implementation,
compared to the existing implementation.

4.1 Experimental Set-up

For our experiments, we used two different machines with two
different platforms:

- **x86-64 Machine** Desktop computer running Ubuntu 18.04, Intel
  8700k @ 3.7-4.7GHz, 32GB 3200MHz DRAM, Samsung 980 Evo NVMe
  storage.
- **Arm Machine** Raspberry Pi 3 Model B running Raspian Stretch
  Lite, ARM1176JZFS ARMv6 @ 700MHz, 512 MB DRAM, 16GB
  SD storage.

All binaries were built with identical flags, other than those deter-
mining the type of exceptions to use. Run-time type information
is disabled (–fno-rtti), the symbol table is removed (–s), and
–O3 optimisation with link-time optimisation (–flto) is employed.
Clang-7.0 was used as the compiler, along with libstdc++ and
libgcc on Arm, and libcpp and libunwind on x86.

4.1.1 Benchmark Application. Lacking an obvious existing realistic
benchmark with which to test exceptions, we instead developed our
own microbenchmark. xmlbench is a simple XML parser, targeting
ease of use and functionality as a real XML parser. It organically
includes a good mixture of functions that can and can not throw
exceptions, and also those which do throw exceptions and those
which are “exception neutral”, i.e. allowing exception propagation.
By supplying different XML files as input to the parser, we can
precisely control the rate of exceptions. Errors in input XML syntax,
and external conditions (such as the input file missing, or an out-
of-memory condition) can be influenced in a realistic way. This
approach to benchmarking is similar to [15], who also use an XML
parser as a representative application.
### 4.2 Results

The following sections detail the results of our observations on various facets of the exception handling infrastructure.

#### 4.2.1 Unwind Library Overhead

Statically-linked binaries participate in whole-program optimisation, and as such have the exception handling and unwind functions removed when using our implementation. This allows for a direct comparison against binaries that use the standard implementation.

Table 2 shows the results of the binary built with both the standard and the deterministic exception implementations. The results for both x86-64 and Arm show substantial decreases in binary size, when using our deterministic implementation. This measurement indicates how large the unwinding code is. Arm’s unwinding code is smaller, but by removing it we see a decrease in binary size of 36.3 Kilobytes, a 62.7% reduction for our program. x86-64 has a much larger unwind library, with our implementation saving 84.7 Kilobytes, an 82.3% reduction in size for identical functionality.

On both platforms, the code section (.text) grows due to the inclusion of additional instructions for checking the exception state. On x86-64, the removal of the unwind tables compensates for this growth, with a reduction of 0.3%. However, on Arm missed optimization opportunities by the compiler leads to an net increase of roughly 8%.

#### 4.2.2 Application Benchmark Performance

We generated two sets of XML files one with syntax errors, and one without, and parsed them with xmlbench. The syntax error is contained within the deepest element, resulting in an exception thrown with the stack at its largest number of function calls. For each run, we measured the total run-time of the xmlbench program, including the start-up time.

Figure 9 shows the average execution time in microseconds per XML element parsed by our benchmark, for both the Arm (Figure 9a) and x86-64 (Figure 9b) platforms. On the Arm platform, the results show there is little difference between execution times, except for the standard exception implementation generally performing the worst of the four, when faced with an exception. As was expected, the standard implementation performs better when no exception is raised with only 156 elements, as our implementation must check for active exceptions. This difference disappears as more elements are processed.

On the x86-64 platform, our deterministic implementation takes slightly longer per element than the standard implementation for larger numbers of nodes. This is primarily due to the overhead of checking whether an exception has occurred. As is expected, the difference in time for our implementation depends on whether an exception was thrown or not is very small, particularly in comparison to the same for the standard implementation, which at its peak has a 2.3% overhead.

The benchmark clearly shows how throwing a single exception increases execution time for the standard exception implementation, given the lengthy unwind procedure. However, with our implementation, the improvements to exception handling speed are not quite enough to compensate for the overhead in the absence of exceptions. With more than a single exception, and particularly given the results for 488,281 elements, where the trend suggests that our implementation scales better than the existing one, it should be enough for our solution to out-perform the current one.

As shown in Table 3, our implementation creates a small additional execution time overhead in comparison to the standard implementation, however, the difference is negligible and within the standard deviation.

#### 4.2.3 Exception Propagation

To test exception propagation, we used a different benchmark, which simply called the same function recursively as many times as indicated by Stack Frames, with a local object having a simple custom destructor within each frame. After Stack Frames calls, an exception is thrown, and caught by reference from the first invocation. The benchmark was compiled with the same flags as xmlbench.

The CPU time was measured between the first function call, and arrival in the catch handler. The experiment was run multiple times for each configuration, and the mean and standard deviation of the results are tabulated in Table 4.

The results show a significant difference in execution time between the two exception implementations. Our implementation is on average 98× faster at performing exception propagation on x86-64. The factor of improvement increases from 68× to 138× as the number of stack frames increases, indicating that the performance penalty seen in the current implementation increases as the number of frames increases. Our implementation performs best on x86-64 with deep call stacks.

On Arm, the improvement in execution time remains almost constant, at 32× to 33× for 100 and 1000 frames, and 35× speedup for 10,000 frames. This suggests that our implementation will consistently offer improved performance independent of the number of frames.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Implementation</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arm</td>
<td>Standard</td>
<td>59,188 bytes</td>
</tr>
<tr>
<td>Arm</td>
<td>Deterministic</td>
<td>22,068 bytes</td>
</tr>
<tr>
<td>x86-64</td>
<td>Standard</td>
<td>105,312 bytes</td>
</tr>
<tr>
<td>x86-64</td>
<td>Deterministic</td>
<td>18,600 bytes</td>
</tr>
</tbody>
</table>

Table 2: Final binary sizes for xmlbench benchmark, with the standard and deterministic exception implementations.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>156</td>
<td>✗</td>
<td>42.7ms (0.6ms)</td>
<td>41.7ms (0.6ms)</td>
</tr>
<tr>
<td>156</td>
<td>✓</td>
<td>43.7ms (0.6ms)</td>
<td>43.0ms (0.0ms)</td>
</tr>
<tr>
<td>3,906</td>
<td>✗</td>
<td>158.7ms (2.1ms)</td>
<td>162.7ms (1.5ms)</td>
</tr>
<tr>
<td>3,906</td>
<td>✓</td>
<td>160.7ms (1.5ms)</td>
<td>162.0ms (2.0ms)</td>
</tr>
<tr>
<td>3,905</td>
<td>✗</td>
<td>3.3ms (0.6ms)</td>
<td>3.3ms (0.6ms)</td>
</tr>
<tr>
<td>3,905</td>
<td>✓</td>
<td>3.3ms (0.6ms)</td>
<td>3.3ms (0.6ms)</td>
</tr>
<tr>
<td>97,655</td>
<td>✓</td>
<td>68.0ms (1.0ms)</td>
<td>70.7ms (0.6ms)</td>
</tr>
<tr>
<td>97,655</td>
<td>✓</td>
<td>69.0ms (1.0ms)</td>
<td>70.3ms (0.6ms)</td>
</tr>
</tbody>
</table>

Table 3: Overall execution time summary for xmlbench on the Arm and x86-64 platforms.
Table 4: Execution times for the exception propagation benchmark.

<table>
<thead>
<tr>
<th>Stack Frames</th>
<th>Implementation</th>
<th>Time</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arm</td>
<td>Standard</td>
<td>1351.3μs</td>
<td>74.3μs</td>
</tr>
<tr>
<td>100</td>
<td>Deterministic</td>
<td>41.7μs</td>
<td>0.6μs</td>
</tr>
<tr>
<td>1,000</td>
<td>Standard</td>
<td>9.6μs</td>
<td>0.2μs</td>
</tr>
<tr>
<td>1,000</td>
<td>Deterministic</td>
<td>0.3μs</td>
<td>0.0μs</td>
</tr>
<tr>
<td>10,000</td>
<td>Standard</td>
<td>94.8μs</td>
<td>1.0μs</td>
</tr>
<tr>
<td>10,000</td>
<td>Deterministic</td>
<td>2.7μs</td>
<td>0.1μs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack Frames</th>
<th>Implementation</th>
<th>Time</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86-64</td>
<td>Standard</td>
<td>114.7μs</td>
<td>21.6μs</td>
</tr>
<tr>
<td>100</td>
<td>Deterministic</td>
<td>1.7μs</td>
<td>0.8μs</td>
</tr>
<tr>
<td>1,000</td>
<td>Standard</td>
<td>2572.0μs</td>
<td>231.2μs</td>
</tr>
<tr>
<td>1,000</td>
<td>Deterministic</td>
<td>29.0μs</td>
<td>0.0μs</td>
</tr>
<tr>
<td>10,000</td>
<td>Standard</td>
<td>8823.8μs</td>
<td>960.0μs</td>
</tr>
<tr>
<td>10,000</td>
<td>Deterministic</td>
<td>64.0μs</td>
<td>6.4μs</td>
</tr>
</tbody>
</table>

Table 5: Execution times for the exception re-throwing benchmark.

<table>
<thead>
<tr>
<th>Stack Frames</th>
<th>Implementation</th>
<th>Time</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arm</td>
<td>Standard</td>
<td>4.1ms</td>
<td>0.1ms</td>
</tr>
<tr>
<td>100</td>
<td>Deterministic</td>
<td>0.2ms</td>
<td>0.0ms</td>
</tr>
<tr>
<td>1,000</td>
<td>Standard</td>
<td>33.5ms</td>
<td>0.4ms</td>
</tr>
<tr>
<td>1,000</td>
<td>Deterministic</td>
<td>1.8ms</td>
<td>0.0ms</td>
</tr>
<tr>
<td>10,000</td>
<td>Standard</td>
<td>318.4ms</td>
<td>6.3ms</td>
</tr>
<tr>
<td>10,000</td>
<td>Deterministic</td>
<td>18.2ms</td>
<td>0.1ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack Frames</th>
<th>Implementation</th>
<th>Time</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86-64</td>
<td>Standard</td>
<td>1071.7μs</td>
<td>363.3μs</td>
</tr>
<tr>
<td>100</td>
<td>Deterministic</td>
<td>22.7μs</td>
<td>13.8μs</td>
</tr>
<tr>
<td>1,000</td>
<td>Standard</td>
<td>3.8ms</td>
<td>0.1ms</td>
</tr>
<tr>
<td>1,000</td>
<td>Deterministic</td>
<td>0.2ms</td>
<td>0.0ms</td>
</tr>
<tr>
<td>10,000</td>
<td>Standard</td>
<td>22.0ms</td>
<td>0.3ms</td>
</tr>
<tr>
<td>10,000</td>
<td>Deterministic</td>
<td>2.4ms</td>
<td>0.1ms</td>
</tr>
</tbody>
</table>

The improvements in execution time are a result of not having to resolve the encoded unwind information, and walk the stack twice to locate the catch handler before unwinding.

4.2.4 Re-Throwing. To measure exception re-throwing, we used another benchmark, which called the same function recursively as many times as indicated by Stack Frames, with a local object having a custom destructor within each frame. After Stack Frames calls, an exception is thrown, but unlike before, caught and re-throw by each function. The benchmark was compiled with the same flags as xmlbench.

The CPU time was measured between the first function call, and arrival in the catch handler. The experiment was run multiple times for each configuration and the mean and standard deviation of the results are tabulated in Table 5.

These results show a similar pattern to those for straight propagation. On x86-64 for 10,000 stack frames, the current implementation takes almost 3× longer to catch and re-throw than simply propagating, while our implementation is 38× slower than when simply propagating. This factor of slowdown is to be expected given how fast our implementation is when not re-throwing, but also due to the multiple steps required to move the exception object into and then out of each stack frame. This is clearly a good target for optimisation, as such transfer is unnecessary. Despite this, our implementation yielded a 9× speedup over the standard implementation.

For 1000 stack frames on x86-64, the difference between execution times is larger, with our implementation executing 19× faster. With 100 frames, the results have a very high standard deviation, but there is a clear order of magnitude between the two times, indicating that the difference in performance remains largely independent of the number of frames.

On Arm, the additional time taken to initialise the exception state in our implementation is evident, as it takes roughly five times longer than in the propagation benchmark. The standard implementation also suffers a performance hit, but one which decreases as the number of frames increases, suggesting that a large part of it is constant. This matches the fact that in the standard
implementation the exception object is allocated on the heap, creat- ing a large up-front cost, but one which is amortised across many re-throwings.

In reality, however, re-throwing at every frame is highly un- usual behaviour, making both this amortisation and our worsening performance unlikely to be noticed.

4.3 Timing and Memory Determinism

We base our understanding of the language and algorithmic require- ments for determinism on Verber and Cohnaric [20]. In general, the requirements for deterministic execution fall into two categories: deterministic memory usage, and deterministic execution time.

For memory usage, we can further subdivide the requirement into stack usage, and heap usage. When no exception occurs, our stack usage is static. A fixed-size allocation for the exception state occurs per noexcept function invocation, throwing destructor invocation, and try block entry.

When an exception is thrown, the object is either allocated with a fixed size determined by the catch block, or with a variable size that can be bounded by the maximum allocation size of all exception objects thrown. The functions called by our implementation have a well-defined order and number known at compile-time and are not recursive. While pointers are used to store this address, their addresses are fixed within the range of the exception object buffer, or the current stack frame.

Our implementation does not require heap allocation. However, it is possible for the program to allocate an arbitrary amount of memory, if local object destructors throw exceptions. For example, during stack unwinding, destructors may throw an exception, causing their own local objects to be destroyed, which in turn may throw additional exceptions.

However, since this potential execution flow is known at compile- time, and is akin to normal function calls made directly via their function definition, rather than via function pointer, static analysis tools are able to give a bound for the number and type of exceptions thrown, and thus the maximum size of allocation. Furthermore, due to the nature of stack unwinding, recursion is not possible, except where introduced explicitly by the developer’s code, and thus the number of stack frames is known.

Deterministic execution time generally refers to the ability to calculate the worst-case execution time (WCET), as demanded by most real-time systems. For execution time, unlike with existing implementations, it is trivial to calculate the WCET of all of our operations. Aside from the allocation and initialisation of the local exception state, and the checking of the active flag (all of which have deterministic timing) we have three operations.

The first allocates the exception object in the exception buffer. In this case, the allocator is a stack allocator, which merely advances a pointer, except where the developer does not specify a correct maximum bound on exception allocation. The second frees the exception object, which again is trivially a pointer decrement. The final function resolves the base class "type id’s when matching polymorphic exception types against catch handlers. Our imple- mentation’s novel approach guarantees a fixed number of iterations through the list of base classes, again known at compile time and derived from the filter of the catch block.

Therefore, should a WCET analysis tool give correct predictions for programs not using exceptions, it would also give such predic- tions for our implementation, meeting the criteria for determinism.

5 RELATED WORK

Modern exception handling, including the semantics of raising (throwing) and handling (catching) exceptions, exception hierar- chies, and control flow requirements were first introduced in [6].

In [11] inherent overheads, both in terms of execution time and memory usage, in the current C++ implementations of exceptions are identified. A further evaluation on how C++ features impact embedded systems software is presented in [16]. This report single out exceptions and its prerequisite run-time type information (RTTI) as "the single most expensive feature an embedded design may consider".

Lang and Stuart [12] describe stack unwinding as it relates to exceptions in real-time systems and show how the worst-case execution time of the stack unwinding is unbounded. This has later been confirmed in [9]. For our own definition of determinism, we take insight from [20], which details many of the requirements necessary for languages to be susceptible to time analysis.

A new exception handling approach which is better suited to static analysis is presented in [13], but it requires substantial code rewriting. SetJmp/LongJmp and table-based exception handling are discussed in [3]. The results demonstrate large increases in binary size and an execution time overhead of 10-15%. Gyllfason and Hjalm- tysson [8] develop a variant of table-based exception handling for use within the Ilumixen port of the Linux kernel.

Most relevant to the work present in this paper is [19]. It focuses on propagating exceptions down the stack by duplexing the return value of functions so as to fit the exception object itself within the register or stack memory used for the return value, and, importantly, to re-use the existing function return mechanism to perform stack unwinding. This approach hinges on two complex changes to C++’s function calling convention, though. This kind of change would be significant, breaking compatibility with C code and all existing C++ libraries. Instead, our novel exception implementation maintains the existing exception model by finding a new way to store and match exceptions at run-time, with little to no additional performance impact when exceptions are not thrown.

6 SUMMARY & CONCLUSIONS

In this paper we have developed a novel implementation of C++ exceptions, which better suits the requirements of embedded systems, while still upholding the C++ core design tenets. Our implementa- tion shows binary size decreases of up to 82.3% over existing implementations, performance improvements of up to 32% when handling exceptions, minimal execution time overhead on x86-64, and none on the Arm platform. We have shown that our implementa- tion achieves memory and execution time determinism, making it significantly better-suited to calculating worst-case execution times, a major hurdle in the adoption of exceptions.

Our future work will investigate further code size improvements resulting from code optimizations and shrinking of the exception state object, while reducing execution time by allocating the active flag in the exception state to a register.
REFERENCES