Exceptional Kernel

Using C++ exceptions in the Linux kernel

Halldór Ísak Gylfason, Gísli Hjálmtýsson
Department of Computer Science
Reykjavík University
Reykjavík, Iceland
{halldorisak, gisli}@ru.is

Abstract - **Driven by the desire to facilitate more maintainable and robust systems, modern programming languages offer explicit constructs to facilitate the handling of exceptional events. The use of exceptions is common in user space programming, and is an integral part of common programming styles and best practices. In spite of this exceptions are rarely used in kernel-space. In fact, some operating systems, such as Linux, refrain altogether from using modern language constructs.**

We have implemented C++ kernel level run-time support for Linux, supporting the full range of C++ language abstractions, including run time type checking and exception handling. Through detailed instrumentation we show that introducing these mechanisms incurs negligible cost to normal program flow. Moreover, by enhancing the user level GNU g++ implementation we have reduced the cost of throwing and catching exceptions sufficiently, to make their use viable in a variety of in several important scenarios.

1 Introduction

Exceptions are in common use in user space programming. Most modern programming languages offer some form of syntactic constructs to handle exceptional events, as exemplified by the Java programming language. The belief is widespread that the use of exceptions leads to more maintainable and robust systems; error handling code is separated from the normal flow and it can be enforced that all exceptional cases will be handled, as for example through the use of checked exceptions in Java [1]. Multiple modern programming styles and best practices encourage the use of exceptions as the vehicle to handle exceptional cases; when a function detects an error condition, which it does not know how to handle, an exception should be raised. Indirect or direct callers of the function set up handlers for particular types of exceptions that are caught from that function.

In spite of the common usage of exceptions in user space, their use in kernel-space has been limited. In fact, some operating systems do not exploit higher level language abstractions at all. In particular the popular Linux operating system is written in pure C. Whereas performance issues may negate the use of some modern languages such as Java, one of the driving factors behind the creation of C++ was for use in writing operating systems [2]. Some constructs were specifically introduced and designed based on observed patterns – and to address problems – in operating systems implementations. Although C++ does not offer strong type safety, nor enforce safety properties for example assured by Java, C++ offers an array of high level language abstractions valuable for the construction of operating systems, and provides type safety and compiler support far beyond that of C. In particular, the safety provided by language level polymorphism provides significant value as polymorphic behavior is widespread throughout any operating system.

In our work on the Pronto software router, we have used many of the advanced C++ constructs extensively including classes and virtual functions to achieve clarity, flexibility and extensibility. We have shown that these benefits come at no performance penalty compared to the Linux implementation [3,4]. Our desire to employ the full range of C++ abstractions in the kernel and in particular to use C++ exceptions in our work on Pronto is the driver behind the work presented herein.

Of course, handling exceptional conditions is relatively expensive, regardless of the mechanisms employed for implementation. Substantial fraction of the code in any operating system is there to resolve exceptional conditions. Handling such conditions requires i) detecting when an exceptional condition occurs, ii) determining where (i.e. by whom) such conditions should be handled, and finally iii) doing the work needed to recover resources and otherwise handle the condition to return the operating system to a state from where it is safe to resume normal execution. The cost of the first and the last are independent of the mechanisms employed to implement the second. The use of language level exception handling translates into machinery providing

complete and systematic approach to the second – i.e. to identify where a thrown exception should be handled.

The performance cost of using the language level exception machinery must be weighed against both its non-performance benefits and the total cost of handling a given exceptional condition. Important non-performance quality metrics include reliability, robustness, flexibility, maintainability and speed of development. In contrast, ad-hoc exception handling patterns common in particular in the Linux kernel consist of convoluted traces where exceptional function abort is communicated to the caller via an exceptional return value. As detailed below the performance overhead of throwing exceptions in C++ is appreciable when compared for example to a simple return with an integer error code. However, in many cases executing the exception handling code dominates the total cost of recovering from an error condition. In those cases the substantial benefits of exceptions warrant the relatively small cost.

However, since throwing a C++ exception is more expensive than returning from a function, there are clearly cases where using exceptions should be avoided. The latter would typically apply when the exception handling is trivial and simple return value is appropriate, or for exceptions that occur relatively frequently (and thus perhaps constitute a branch rather than a true exception) or operate on such fine time-scale that even a small overhead is a burden. However, rather than voiding the viability of using language level exceptions, the added cost instead determines the granularity appropriate for the use of exceptions, and/or the rarity at which the exceptional case must occur to justify the cost. Therefore, an important contribution of this paper is to quantify the cost of using the throw/catch mechanism with our run-time support, allowing the programmer to determine these tradeoffs.

Exceptions in C++ support polymorphic exception handling, and therefore provide for unified exception handling even as the kernel functionality is extended. Our Pronto kernel supports dynamic introduction of new data types into a running kernel, including late binding of type specific system calls [4]. However, introducing new types, e.g. a new tunneling facility, may introduce new types of exception conditions. Introducing simultaneously the corresponding new exception type (implementing the handler), allows us to handle the new exception condition(s) without changing the already running code. We give example of this in Section 5.2. In contrast, traditional ad-hoc methods cannot handle this scenario gracefully.

The primary contribution of this paper is a complete kernel level run-time support for C++ in the Linux kernel. In particular our run-time support enables the full use of C++ exceptions in the Linux kernel, but notably also includes support for global constructors and destructors, and dynamic type checking. Our kernel level support is based on open source commodity components, specifically the GNU gcc/g++ compiler and its exception implementation, the C++ABI version independent standard interface. As such we believe that our approach is applicable for other operating systems, but we have not verified this. In particular our optimizations for exception handling are platform and OS independent.

A significant additional contribution is in Section 6 where we quantify the cost of the throw/catch mechanism based on detailed measurements. We show and quantify a number of typical kernel scenarios where the use of exceptions is viable and valuable, including a system call invocation and use of exceptions in the Linux data forwarding path. Our measurements indicate that the throwing of exceptions, with the GNU g++ compiler, is relatively expensive. However, analyzing the implementation, we have identified and implemented several optimizations, appropriate for kernel level, reducing the cost of throwing exceptions by an order of magnitude. We demonstrate that the overhead of introducing exceptions does not impede normal flow of the program. We give measurements showing both absolute cost of exception handling, and the relative cost by comparing it to common exception handling functions.

Although our C++ kernel level run-time support is complete in that it supports all C++ facilities, the primary complexity, and perhaps controversy, centers on our support of exceptions. The rest of the paper therefore mostly discusses that aspect of our library.

The rest of the paper is organized as follows. In Section 2 we discuss related work. In Section 3 we discuss our kernel level run-time support, diving into the implementation og exceptions in Section 4. In Section 5 we show examples of practical use of exceptions, in particular for use in system calls, and for use in the data path of the Pronto Linux based router. In Section 6 we report on our measurements, followed by discussion in Section 7. In Section 8 we conclude.

2 Related work

Current approaches to kernel level exception handling strongly resemble the state of affairs in the seventy's when research on exceptions and exception handling

started to emerge. J. B. Goodenough [5,6] provides an excellent overview of the common techniques used for error handling, before introduction of exception handling mechanisms in programming languages. These methods – employing error codes, separate return values, non-local goto etc. – are still common at kernel level. The same applies for all the issues associated with such ad-hoc exception handling – issues that ultimately drove specific exception handling constructs into modern programming languages, and specifically C++.

Among early programming languages to provide support for exceptions were CLU [7,8], PL/1 [9] and Mesa [10]. The semantic of exceptions in these languages differs in number of areas, including whether handler association is static or dynamic, whether resumption is possible at the site where the exception was thrown, whether handler location is static or dynamic and whether exception can be propagated several levels automatically.

Implementation of exceptions has caused significant amount of attention. In Appendix 1 of [11] two implementation methods of C++ exceptions are described on a high level. Much of that discussion applies to languages implementing similar exception model, including Java. The first method employs dynamic registration based on the setjmp/longjmp methods, also covered in depth in [12]. This approach incurs some (albeit small) cost when entering try blocks, but has the advantage that it is portable in the sense that it is possible to generate ANSI C from a C++ program. The second method is based on a table of program counter values that map try blocks into program counter values for handlers. This method incurs no run-time overhead when entering a try block (zero instructions), at the expense however that throwing exceptions incurs increased overhead in comparison to the setjmp/longjmp method. This table driven approach is further detailed in [13,14]. Hewlett-Packard's implementation of exception handling for the IA-64 based on the table-driven approach is described in [15]. The implementation is "in a way that leaves the door open for optimizations, even in the presence of exceptions" [15] employing optimized stack layouts through landing pads similar to ideas presented in [17]. The GNU g++ compiler implementation, on which we build our run-time library, implements exceptions in a similar manner as described in [15].

The performance impact of exception handling, and in particular the throw/catch mechanism as now used in most languages including C++, has been studied extensively. The use of exceptions may introduce additional control flows into the program, even for procedures that do not use exceptions potentially invalidating some conventional optimizations. However, as suggested in [16], accounting for the additional semantics may reduce or remove this penalty. The additional control flows must be incorporated into the basic block model, which puts more constraints on register allocation in terms of variable lifetime. However, other error-handling techniques, such as checking return codes with if-statements, do negatively impact control flow. In general the use of exceptions does not inhibit compiler optimizations, but does however put more requirements on the back-end of the compiler. [17] demonstrates how optimized frame layouts through non-standard calling conventions can be used even with the PC-based stack unwinding approach of implementing exceptions.

Qualitative measures have shown that while the effect on program size is noticeable (up to 20%), the overall impact on run-time performance is small (see for example [18,19]). Our measurements show that compiling our run-time support into the kernel results in less than 3% performance degradation in the Linux data path. Whereas the two methods of implementing exceptions described above either optimize the exception handling (setjmp/longjmp) or the normal flow, [20] attempts to optimize the throwing of exceptions without sacrificing the performance in the normal path, based on the observation that some Java programs frequently throw exceptions. The method of [20] detects hot exception paths at runtime, and for those paths it *inlines* all the methods from the thrower to the catcher, into the catcher. Finally throws are eliminated by replacing the throw with the explicit control flow to the catch. This produces significant performance savings.

Employing language level exception mechanism at kernel level has not received substantial attention. Some operating systems are written at least partly in C++ but generally they do not employ C++ exceptions. SPIN [21], written in Modula-3 to provide a higher level of safety for kernel extension, uses exceptions, which is an integral part of the Modula-3 language. Windows NT provides a facility called Structured Exception Handling (SEH) [22,23], which can be used in kernel device drivers. Although the SEH is similar to C++ exception handling but has different semantics. Interestingly SEH is at the core an operating system facility and thus independent of any compiler. SEH uses dynamic registration of try-blocks and thus its usage does affect normal flow. Each thread is associated with a stack of exception registrations, through the thread information block. Conceptually, an entry into a try block pushes an exception registration on the stack, although language dependent compilers may optimize this – the Microsoft

C++ compiler uses one per function. Exception registrations contain a function pointer to a handler function that returns a value indicating if it wishes to handle the exception. In the SEH model, exceptions are thrown explicitly with the RaiseException Win32 routine and, in contrast to the C++ exception model, exceptions in the SEH model are singular integers. However, the SEH model also covers processor-level errors, such as divide-by-zero, access violations and stack overflows, which requires support from the operating system. When an exception is raised, either implicitly or explicitly, the operating system calls the handler functions on the exception registration stack in sequence. Each handler function decides whether to handle the exception, to pass it through or to resume execution at the point where the exception was raised. Thus the SEH model allows resumption. One drawback of SEH is that it does *not* call destructors during stack unwinding, however the *__finally* block can be used to clean up resources. The Microsoft C++ compiler implements C++ exceptions on top of the SEH model and as a consequence the catch(…) block under that compiler catches all C++ exceptions as well as processor-level errors, such as a segmentation fault. Segmentation faults on Linux are not caught when compiled with the GNU g++ compiler.

Exceptions in the context of real-time systems are analyzed in [24] in where the authors evaluate an array of exception handling implementations against a set of requirements deemed essential for real-time systems, including predictability. Neither the table-driven approach nor the dynamic registration are predictable in this sense since they are proportional to the length of the calling chain/handler chain, which cannot be predicted statically. Since the primary objective is predictability the authors are willing to accept cost in the normal flow.

3 Using the C++ kernel level run-time support for Linux

The new C++ kernel level run-time support for Linux provides complete run-time support for C++, including support for virtual functions, memory allocation operators, global constructors/destructors, dynamic type checking and exceptions. The code is installed by applying a patch to the Linux kernel and enables the full use of C++ using the GNU g++ compiler. Programmers that have used C++ in Linux kernel modules have primarily been using classes and virtual functions, but not global constructors. dynamic type checking and exceptions. Using even this small part of C++ requires each programmer to write some supporting routines. Using the rest of C++ includes porting the C++ ABI that accompanies GNU g++ to the Linux kernel, and to enable global constructors and destructors.

Using our new C++ kernel level run-time support, programming in C++ at kernel level becomes similar to programming in user space. The compiler compiles files ending with .cc as C++ file. However, the Linux kernel distribution is written in vanilla C, so typically C++ source files do need to include C files. This introduces a problem not commonly encountered in user space, as some of the C++ keywords have been used as identifiers in some of the Linux header files. To combat this, we have provided two inclusion files – begin_include.h and end_include.h, respectively – with our distribution that should be used to enclose the Linux C header files, as shown in Figure 1. These two files use `#define's` and `#undef's`, respectively to redefine these identifiers to names accepted by the C++ compiler.

```
#include <begin_include.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <end_include.h>
```

Figure 1 – Inclusion from Linux C header files

The begin_include/end_include files take care of renaming C++ keywords temporarily, such as new and virtual, that Linux programmers use as variable names. It is however possible that some of the Linux source contains struct initializations that are incompatible with C++, which causes problems if such structs compiled inside the kernel proper (with the C compiler) need to be referenced from the C++ code. These initializations must currently be changed by hand.

4 Exceptions in the Linux kernel

The GNU g++ compiler implements exceptions according to the table driven approach using the application binary interface (ABI). The (user level) implementation of the ABI accompanies the compiler as part of the standard run-time library. When using C++ exceptions, GNU g++ generates calls to the ABI. For example the throw operator is transformed into a call to the ABI function *__cxa_allocate_exception* followed by *__cxa_throw*. GNU G++ versions 3.x implement the C++ ABI specification for IA-64 [25,26]. The aim of the C++ ABI specification is to standardize the object layout and the interface of the object code to the runtime system. Thus code compiled with old versions of the compiler should be compatible with newer releases and, more

```
struct task {
   volatile long state;
   struct thread_info *thread_info;
   …
   siginfo_t *last_siginfo; /* for ptrace use*/
#ifdef CONFIG_CXX_RUNTIME
   struct {
      void *caughtExceptions;
      unsigned int uncaughtExceptions;
   } exa_eh_globals;
#endif
};
```

Figure 2 - The Linux task struct. Our changes to support exceptions appear inside the #ifdef shown in the Figure.

ambitiously, object code from different compilers should be compatible.

Conceptually GNU g++ stores a table that maps each instruction pointer value to register state, using Dwarf2 [27] to encode the frame info. The Dwarf2 frame info table is stored in an interpretive form – instructions must be interpreted to compute the register state for a certain program counter value – to limit the size of the information in the object code.

The first step in our work to support kernel level exceptions is to create and include an implementation of the C++ ABI in the kernel. We start by carving out of the user level library code the ABI implementation as the basis for our implementation. Simply porting the user level code to the kernel level requires some changes. The *malloc* function used to reserve space on the heap for exceptions is replaced with the kernel level *kmalloc* function. The GNU library is thread-safe and uses locking to guard certain data structures, with the aid of the *pthread* library. However, the *pthread* library is not available in kernel space we have modified the locking mechanism use spinlocks.

In user space information on active exceptions is kept in a thread-local storage. To achieve this we have added the corresponding structure to the process information block, the *task_struct*. This is depicted in Figure 2. This structure is strictly speaking only required to be able to rethrow exceptions, i.e. to use the throw operator without arguments. Thus the above modification to the Linux *task_struct* could be omitted by sacrificing this use of the *throw* operator.

When creating an ELF executable, GNU g++ secretly links two object files at the front and the back, crtbegin.o and crtend.o. This is necessary to ensure that global constructors and destructors are run, and that the Dwarf2 frame info is registered to the C++ ABI, thus enabling exceptions. GNU g++ adds initialization code into the *.init* ELF section and cleanup into the *.fini* section. To allow exceptions and global constructors to be used in the kernel the Makefile rule for the kernel image and kernel modules is modified to link with those two files. Furthermore we must manually ensure that the initialization routines are called, since the kernel module loader in Linux pays no attention to the ELF .init section. We accomplished this by clever use of preprocessor macros, that change the definition of the module initialization functions, *module_init* and *module_exit*.

GNU g++ registers the exception table and the stack frame layout on module load, but the user level C++ ABI implementation does not process it in any way. Thus the first time an exception is thrown in or through a module the information must be processed, which includes sorting the stack frame info in program counter order using heap-sort. This means that throwing the first exception is quite expensive. For our kernel level C++ ABI implementation we have opted for processing this information at module load.

These additions to the kernel are available as a kernel patch and the size of it is negligible compared to the rest of the kernel. Total size of the Linux kernel (excluding blanks and comments) is around 4 millions LOC, while the size excluding all drivers and including only the i386 architecture is around 1,2 million LOC. The size of the C++ ABI is around 7000 LOC. Compiling the C++ ABI into the kernel image, with exceptions and runtime type information (RTTI) enabled, increases the kernel image by 2%;. Note however that we still compiling most of the C files with gcc, which is not emitting exception frame info. When compiling the whole kernel with exceptions enabled, which makes it is possible to throw exceptions

```
class OSException
{
public:
   char* getMessage();
   OSException(char* msg,int sev);
   int getSeverity();
   virtual void report();
   enum tSeverity {MINOR=1,MAJOR,FATAL};
private:
   char* message;
   int severity;
};

class NetworkException : public OSException
{
   …
};

class ProntoException : public NetworkException
{
   …
};
```

Figure 3 - Simple exception hierarchy

through the whole kernel, the increase in size of the kernel image is around 10%. This code increase will however generally not affect the normal flow. Code that uses exceptions extensively increases slightly more in size. In extreme cases, when the code does little other than exception handling we have observed an increase in size for a single object file of up to 40%.

Evaluation of performance and overheads is given in Section 6.

5 Using kernel level exceptions

In this section we give few examples of the use of exceptions within the Linux kernel. These examples are from our own actual use in our Pronto router project.

In this example we use as an example class hierarchy, depicted in Figure 3, where each exception corresponds to a sub-system. The top-level exception class – *OSException* – consists of a message, severity and a virtual method, report, which by default performs a *printk* of the message if the severity is *MAJOR* or *FATAL*. The other two exceptions defined, are *NetworkException* derived from *OSException*, and *ProntoException* derived from *NetworkException*.

5.1 System Calls

The most straightforward use of exceptions in kernel space is in system calls. The Pronto architecture introduces three new systems calls to the Linux kernel. New types of packet processors [4] (an abstract data type for processing in the data-path of the Pronto router) can be plugged into the operating system at run-time and their behavior manipulated through type specific system calls that are dynamically linked through virtual functions. To promote safety it is beneficial to catch all exceptions thrown by packet processors.

Figure 4 shows the use of exceptions to guard a systems call. In this example the *sys_pproc_type_call* is the entry point from the system call. It's only function is to dispatch a method invocation to thePProcKType, which is an object in a dynamically loaded module. To guard the dispatch, the virtual call is performed inside a try block.

The system call catches all *ProntoExceptions* and calls the virtual function *report*. The packet processors can throw subclasses of *ProntoException*, and customize the report function. Finally all other exceptions are caught with the second clause. This could include processor-level errors, if the operating system provides support for

```
asmlinkage int
sys_pproc_type_call(int pptype, int call, void* args)
{
   int retval = -ENOSYS;
   try {
      if (thePProcKType) {
         retval = thePProcKType->syscall(pptype, call, args);
      } else {
         printk(KERN_ERR "pproc not loaded");
      }
   } catch(ProntoException & exception) {
      exception.report();
   } catch(...) {
      printk(KERN_ERR "Unknown Exception occurred");
   }
   return retval;
}
```

Figure 4 - Using exceptions to guard system call dispatch

mapping processor level errors into catch-able exceptions – a topic that we are currently researching for the Linux platform.

5.2 Using try-blocks in the data path

The Pronto data path consists of a classifier that maps packets to flows. Each flow is associated with a forwarding path consisting of chains of packet processors. Each forwarding path may have multiple branches. Examples of packet processors include basic IP forwarding, tunnel entry/exit, NAT functionality, and more. Packet processors are dynamically added to the router at run-time.

Figure 5 shows how we employ exceptions in this critical part of the data path. A *try* block guards the processing of a packet as it is sent through the chain of packet processors associated with the flow they belong to (identified by the call to the classifier above the try). As in the previous example, there are two *catch* statements, one catching all ProntoExceptions, the other catching all.

It is worth noting in this example, that as new types of packet processors, say for example IPSec tunnel entry,

```
int pronto_ip_rcv(struct sk_buff *skb, …)
{
   …
   flow = ((classifier_module*)classifier)->lookup(skb);
   if( flow ){
      try {
         flow->arrive( skb );
      } catch(ProntoException & exception) {
         exception.report();
         kfree_skb(skb);
      } catch(...) {
         printk("Unknown exception occured");
         kfree_skb(skb);
      }
   }
   …
}
```

Figure 5 - Using exceptions in the Pronto router forwarding path

are introduced they may in turn introduce new subclasses of the *ProntoException*, defining a new handler (the report method). This way the Pronto data path is capable of performing type specific exception handling for new dynamically installed types!

6 Evaluation

In this section we evaluate our run-time support via detailed measurements. We first discuss the cost of dynamic type checking, followed by an in depth measurements of the overhead of exceptions.

Our measurements were performed on an Intel Pentium 3,996.859 MHz running the Linux 2.6.6 kernel that has been patched to include Pronto and the C++ runtime library.

6.1 Dynamic type checking

The cost of dynamic type checking in C++ is highly dependent on the method used to encode the runtime type information in the objects. GNU g++ associates with each class a type information object that encodes the type of the class as a mangled string and puts a pointer to this object in the virtual table for the class. GNU g++ uses weak symbols to reduce the dynamic type checking to a pointer comparison, thus avoiding the more expensive string comparison. Each time a class, containing virtual functions, is used in a source file, GNU g++ generates the virtual table, type information object and type name string as weak symbols and the user space linker ensures that there is only one copy of this object, which renders the simple pointer comparison sufficient. However, the kernel module loader, which in the 2.6 versions of the kernel is exclusively in kernel space, does not handle these weak symbols correctly and always relocates references to weak symbols to the weak definition within each object file that is being loaded. Therefore multiple type information objects may exist for the same class and pointer comparison becomes insufficient when doing dynamic type check across kernel modules. To avoid this overhead we have modified the kernel module loader to handle these weak symbols; the first time a weak symbol is encountered it is added to the symbol map, and on subsequent encounters the relocation is done to the first symbol. This modification is included in the C++ kernel-level library.

For the purpose of measuring the cost of the dynamic cast operator we implemented a class hierarchy, A←B←C←D←E where A is the root of the class hierarchy, and a method that receives a pointer to A and performs a dynamic cast to type B*. This table lists the measurements for the dynamic cast operator when passing pointers to instances of B, C, D and E respectively. The results are given in Table 1.

Table 1 - Cost of dynamic cast

Class	Cost (µs)
B	0.11
C	0.16
D	0.21
E	0.26

The results indicate that the cost of each additional level in a class hierarchy is 0.05 µs. In comparison, when using string comparison the cost rises more quickly, and is furthermore influenced by the length of the class name and common prefixes; by using a common prefix of 2 characters and increasing the class name to 9 characters we measured the cost of dynamic cast using the lowest level class (E) to be 0.67 µs.

6.2 Absolute cost of kernel level exceptions

For the purpose of measuring the absolute cost of throwing an exception, we implemented a kernel module that throws an integer out of a function, which is caught in the direct caller.

To put the absolute numbers in context we measured the performance of the Linux *printk* function, which is commonly used in exceptional circumstances to communicate error messages to users. The time to print a string of length 6 – *printk("Error\n")* – was measured to be 18.14 µs.

Although based on the user level implementation provided with the GNU g++ distribution our kernel level implementation contains a number of important optimization. To appreciate the impact of these optimizations, we first discuss the overhead before our optimizations, and give the breakdown of the overhead, showing where our performance improvements are coming from, building up to the conclusion of the subsection showing the absolute cost using our optimized run-time support.

Without the optimization the minimum time duration from the point of throw to the point of catch measured was 12.70 µs. Since our usage of exceptions involves throwing objects we also measured the performance of throwing the *ProntoException* mentioned above. The time from the throw to the catch increased to 13.08 µs in this case, showing relatively little difference in

performance when throwing objects compared to an integer; the principle difference involves copying of larger content to the heap area.

To target our optimizations we analyzed the implementation of exceptions in GNU g++. The implementation of exceptions in the C++ ABI in GNU C++ can be characterized by three things: Ease of debugging, independence of processor architecture and independence of programming languages. All of these aspects are potentially harmful for performance and, with the possible exception of the independence of processor architecture. We measured in detail the breakdown of the cost of throwing exceptions. This breakdown is tabulated in Table 2.

Table 2 - Breakdown of cost of throwing an exception using the GNU g++ ABI implementation.

Portion	%
Allocation and initialization of resources (*__cxa_allocate_exception*, *__cxa_throw*)	3.3 %
Locating the handler – first phase of stack unwinding (*_Unwind_RaiseException*)	56.5 %
Actual unwinding – second phase of stack unwinding (*_Unwind_RaiseException_Phase2*)	36.5 %
Installation of the runtime context at handler (*uw_install_context*)	2.6 %
Manipulation of the runtime stack of "active" exceptions (*__cxa_begin_catch, __cxa_end_catch*)	1.1%

From Table 2 we see that the unwinding of the stack accounts for 93% of the time. We also observe that the stack is unwound two times, both time incurring substantial cost. The first phase, the search phase, looks for a handler for the exception without restoring the unwound state. If a handler is found the second phase, the cleanup phase, commences to restore the state to the stack frame that contains the handler. The reason for this two-phased approach is that in the case that no handler is found the stack frames have not been destroyed and the debugger can inspect the state of the frame that threw the exception. However, for our use at kernel level, we don't see this cost justified. In fact, we feel that even in user space the programmer should have the option of having the compiler optimize this debugging help out of the code. The first optimization in our run-time support is therefore to unwind the stack in one phase.

The second optimization we performed concerns the aspect of the GNU implementation that separates the exception library in two disjoint sets – the language independent unwind library, and the language specific library. The GNU compiler suite implements a set of languages, including C++, Ada and Java, all of which have exception facilities. The actual unwinding is generic for all languages, while the location of a handler within a specific function is language specific, customizable through a "personality routine" encoded in the Dwarf2 frame info. The throw operator in C++ is transformed by g++ into a call to *__cxa_allocate_exception*, followed by *__cxa_throw*, and when that function has finished initializing the C++ specific parts of the exception object the generic *_Unwind_RaiseException* function is called. This means that when the actual unwinding starts there is an additional function on the stack that the implementation will have to unwind. By manually inlining the _Unwind_RaiseException function into the *__cxa_throw* function this is avoided in our library.

The two above mentioned optimizations brought the performance of throwing of an integer through one function from 12.70 μs to around 6 μs.

To enhance the performance further the third optimization improves the mechanisms used to search for the handler. When an exception is thrown, the C++ ABI needs to locate the Dwarf2 frame descriptor entry for each function that the exception goes through, including the function where the throw is located as well as the function where the exception is caught. The C++ ABI accomplishes this by a linear search through a sorted linked list of objects representing the main program as well as each dynamically linked library (in the Linux kernel this corresponds to the kernel image and each kernel module) followed by a binary search through an array containing the frame descriptor entries sorted by program counter. Once the frame descriptor entry has been located the Dwarf2 instructions are *interpreted* to compute the frame state for the function at the current program counter value. The frame state consists of a state for each register that specifies if and where the register has been saved, a rule to compute the canonical frame address, and a pointer to a "language specific data." In the case of C++ the language specific data is the exception table for the function, if one exists. Our optimization caches this frame state data in a hash table, indexed by program counter. When an exception is thrown the first time through a function, or more specifically the first time through a certain place in the

function, the frame state is computed and subsequently inserted into the hash table. Subsequent throws through this place result in a successful lookup in the hash table which saves the time to locate the frame descriptor entry and the interpretation of the Dwarf2 instructions. Thus, the optimization detects the exception paths at runtime and caches data to speed up the process. The importance of this optimization increases the more exceptions are used in the kernel since the time needed to locate the frame descriptor entry for a function is proportional to the number of modules that use exceptions and the number of functions within those modules.

Further optimizations are possible, for example, stripping the implementation of all language independent code and or avoiding the allocation of the memory in the kernel-heap, possibly by allocating special pages for the exception structures.

However with these three optimizations we have manged to reduce the the absolute cost of a one level throw from 12.70 µs to 2.14 µs, or about a tenth of the cost of a trivial printk. As we discuss below this seems quite acceptable for exceptional events in a number of important scenarios.

As expected the cost of exceptions is dependent on the number of stack frames that the exception is thrown through. Table 3 tabulates how the number of stack frames affects the cost of throwing an exception.

Table 3 - Absolute cost of throwing exceptions through a number of stack frames.

# Stack Frames	Cost (µs)
1	2.14
2	2.52
3	2.85
4	3.21
5	3.59

We observe that cost increases with each stack frame about 0.35 µs. However, it should be taken into account that when using other types of error handling techniques the cost also increases with number of stack frames traversed.

For comparison, Table 4, tabulates the same cost before our optimizations. We observe that the increase in cost for each stack frame in the GNU g++ implementation without our optimizations is around 2.5 µs. Hence, the effect of our optimization is even more impressive when throwing the exception through multiple functions.

Table 4 - Absolute cost of throwing exceptions through a number of stack frames without our kernel level optimizations.

# Stack Frames	Cost (µs)
1	12.70
2	15.43
3	18.12
4	20.46
5	23.09

6.3 The cost of using exceptions in system calls

For the purpose of measuring the difference of implementing a system call with error codes versus an implementation that utilizes exceptions we implemented a system call in Linux that invokes a virtual function that immediately returns an error code, which the calling function checks. For comparison we implemented a version of the system call where the function throws the same error code, which is subsequently caught in the callee. The actual measurement is performed in a user space program with the clock function, since context switches may occur during the measurement. We measure the average cost per call when invoking the call repeatedly (ten million times) to alleviate the lack of precision inherent in the clock function.

Without exceptions, we measure the average time of the execution of the system call to be 0.22 µs. In comparison we measure the average time using exceptions was 2.46 µs. The difference is consistent with our observed cost of throwing an exception.

Although this is an order of magnitude difference two observations are important. First, most system calls perform more expensive useful functions. For those system calls the 2 µs overhead is small. Second, if the exception is truly an exceptional event, say occurring once every 1000 calls the overhead is around 2.3 µs compared to 220 µs which is negligible. These measurements do however indicate that exceptions should not be used gratuitously in these settings and be reserved for exceptional events.

6.4 The cost of using try blocks in the data path

To test the viability of using exceptions in fine timescale intensive workload, we measure the cost of using try blocks in the Pronto data path, as shown in the example of Section 5.2. We measure packet latency, using 64 byte UDP packages at a rate of 10000 packets per second through a router. The router was equipped with a single 993 MHz Intel Pentium 3 processor, having Intel PRO/1000 Gigabit Ethernet interfaces. The interfaces

issue DMA directly. We measure the latency by time-stamping each packet immediately after the packet is in main memory and again just before it is transferred to the output card with DMA.

For comparison we measured the packet latency in the unmodified Linux kernel, and the packet latency in that scenario was 4.29 µs. Using Pronto (Linux based) router compiled without exceptions and RTTI, the observed packet latency is 4.24 µs. Although the difference is negligible, it is interesting since Pronto uses classes and virtual functions. As we have verified in prior work the use of classes and virtual functions does not incur a measurable overhead in our scenario. When compiling in the full C++ runtime support, with exceptions and RTTI and using try-blocks in the Pronto data path, the packet latency grows to 4.36 µs – an overhead of 2.8%. This is consistent with the results of other writers – adding support for exceptions seems to impose a slight runtime overhead. Since try-blocks do not emit any code, this can only be ascribed to less aggressive (or successful) optimizations performed by g++. Using exceptions should not have considerable effect on the instruction cache in the normal flow, since under optimization level O2, g++ positions all catch handlers at the end of each method, ensuring that the normal flow is not cluttered with exception code. This effect is currently partially achieved in the Linux kernel by hand – the normal flow is positioned at the beginning of a function, labeled error handlers are positioned at the end, and goto statements are used in the normal flow to invoke the error handling code.

6.5 Throwing exceptions in the data path

To measure the cost of throwing exceptions under intensive workload on fine timescale, we measure the impact on thrown in the data forwarding path of the Pronto router. The setup is as follows. An exception packet processor is configured to throw an exception for every packet. The classifier catches the exception which it handles by sending the packet to the IP-forward packet processors that injects the packet to the device queue of the output device. For comparison the same experiment is performed where the first packet processor returns an error code rather than throwing the exception. The setup is depicted in Figure 6.

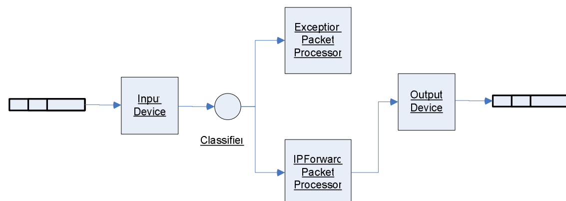
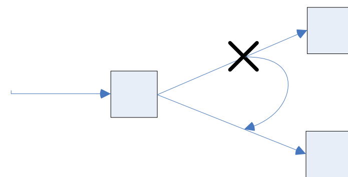


Figure 6 - Setup for measuring the cost of throwing exceptions in the Pronto data path.

When throwing exceptions per-packet in this setup, we measured the packet latency to be 8.8 µs. (Since multiple branches are typically used in multicast, the packet is copied with the *skb_copy* function before sending it down the second branch, which increases the packet latency). Returning only an error code we observe the latency to be 5.8 µs. The difference is slightly higher than the minimum cost of exceptions in the previous subsection, or 3 µs. However, more importantly this cost if of the same order as the total latency in normal forwarding mode (4.36). Of course this implies that it would be unwise to throw an exception for every packet. However, even for events that occur once every 100 packets the cost of using exceptions would have limited impact on router throughput (less than 1%), and is altogether negligible for even less frequent events.

As a related example consider using exceptions for fast recovery upon link failures.



Assuming link failures are rare, the overhead on total throughput would clearly be negligible. Even when the failure occurs, the overhead of throwing the exception would only imply a handful of additional packets lost, even for very fast links. Moreover, in comparison to the true cost of handling the exception, routing update, local flow state updates on all active flows, the absolute cost of the exceptions is negligible.

Another interesting example to consider is to use exceptions to handle exceptional protocol conditions where an ICMP message should be generated. In our analysis of traffic in our network we determined that the volume of ICMP traffic is 1 packet per every 3500 packets or 0.03%. If the average packet latency is 4 µs, the aggregate packet latency is 14 ms and the additional overhead of 2 µs is negligible in comparison.

7 Discussion

As mentioned before, further optimizations of the exception handling mechanisms beyond what we describe above are possible although in some cases a modification of the compiler would be required. Ideally

the GNU g++ compiler would include our optimizations and allow the programmer to turn them on. Although user-level programs tend not to be as time critical, doing so would lower the barrier for exceptions and make them more useful.

However, in spite of our optimizations, throwing an exception is a relatively expensive operation in relation to other atomic constructs, such as function calls and control structures. Consequently exceptions should not be used gratuitously and be reserved for exceptional events.

We are interested in investigating the possibility of mapping processor-level errors to C++ exceptions, to gain this benefit that the Windows SEH has over the C++ exception mechanism. The ultimate goal is to build a robust pluggable-kernel. Such mechanism would allow us to catch all exceptions originating within the pluggable kernel-modules and to remove such modules from the kernel to avoid allowing them to crash the system.

8 Summary

In this paper we have discussed and evaluated our new C++ kernel level run-time support for Linux, that allows programmers to use the full power of C++ in kernel space programming, including global constructors and destructors, dynamic type checking and exceptions. Our new kernel-patch works without any modification to the compiler, and is compiler version-independent, so long as g++ adheres to the C++ ABI specification for IA-64; indeed we used the 3.4 version of the ABI with the 3.3 version of the compiler.

Our run-time support builds on the GNU g++ distribution, optimizing the GNU implementation reducing the performance overhead by an order of magnitude. The cost of exceptions is low in comparison to some other error handling operations in the Linux kernel, such as the *printk* function, and viable in many situations of interest.

Finally we have quantified the cost of exceptions which serves as an important guide for programmers to determine when the use of exceptions can be justified.

9 Acknowledgements

We would like to thank Pétur Runólfsson for porting the C++ runtime support from version 2.96 of g++ to versions 3.x. Also, many thanks to Heimir Þór Sverrisson, for help on various aspects, including measurements.

10 References

[1] G. Bracha, J. Gosling, B. Joy, G.Steele. “The Java Language Specification”. Addison-Wesley, 2000.
[2] B. Stroustrup, The Design and Evolution of C++. Addison Wesley, Reading, MA, 1994
[3] G. Hjálmtýsson. “The Pronto Platform – A Flexible Toolkit for Programming Networks using a Commodity Operating System” in the Proceedings of OpenArch 2000, Tel Aviv, Israel, March 2000.
[4] G. Hjálmtýsson, H. Sverrison, B. Brynjúlfsson, Ó. Helgason. “Dynamic Packet Processors – A new abstraction for router extensibility”.in the Proceedings of OpenArch 2003, San Fransisco,
[5] John B. Goodenough. “Exception Handling: Issues and a Proposed Notation”. Comm. ACM 18, 12 (Dec 1975), 683-696.
[6] John B. Goodenough. “Structured Exception Handling. Conf. Rec., Second ACM Symp. On Principles of Programming Languages”, Palo Alto, Clif., Jan. 1975, pp. 204-224)
[7] B. Liskov, A. Synder. “Exception Handling in CLU”. IEEE Trans. On Software Engineering SE-5,6 (Nov 1979), 547-557.
[8] B. Liskov. "A History of CLU", Laboratory for Computer Science, MIT, Technical Report, April 1992.
[9] D. McLaren. “Exception handling in PL/I”. In Proceedings ACM Conference on Language Design for Reliable Software, Mar. 1977, 101-14.
[10] James G. Mitchell, W. Maybury, R. Sweet. Mesa Language Manual, Technical Report CSL-78-1, Xerox Research Center, Palo Alto, CA, February 1978.
[11] A. Koenig, B. Stroustrup. “Exception Handling for C++”, Journal of Object Oriented Programming, Vol. 3, No. 2, July/Aug. 1990
[12] D. Cameron, P. Faust, D. Lenkov, M. Mehta. “A Portable Implementation of C++ Exception Handling”, Proc. USENIX C++ Conference, August 1992.
[13] S. Drew, K. J. Gough, J. Ledermann. “Implementing Zero Overhead Exception Handling”. Tech. Rep. Technical Report 95-12, Faculty of Information Technology, Queensland University of Technology, 1995.
[14] J. Lajoie. “Exception Handling – Supporting the runtime mechanism”, C++ Report, Vol. 6, No. 3, March-April 1994
[15] Cristophe de Dinechin. “C++ Exception Handling”. IEEE Concurrency, October-December 2000, pp 72-79.
[16] J. Hennessy. “Program Optimization and Exception Handling” , Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, p.200-206, January 26-28, 1981, Williamsburg, Virginia.
[17] D. Chase. “Implementation of exception handling-II. Calling conventions, asynchrony, optimizers, and debuggers”, Journal of C Language Translation, Vol 6, No. 1 , October 1994.
[18] M. J. O’Riordan. “Technical Report on C++ Performance”. JTC1/SC22/WG21 - Papers 2002.
[19] J. L. Schilling. “Optimizing Away C++ Exception Handling”. ACM SIGPLAN Notices, Vol. 33, Iss. 8 (August 1998) pp. 40-47
[20] H. Komatsu, T. Nakatani, T. Ogasawara. “A Study of Exception Handling and Its Dynamic Optimization in Java”. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications,

pp. 83-95, Oct. 2001.
[21] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C., "Extensibility, Safety, and Performance in the SPIN Operating System," Proc. 15th SOSP, Copper Mountain, CO, Dec. 1995, 267-284.
[22] M. Pietrek. "A Crash Course on the Depths of Win32TM Structured Exception Handling". Microsoft System Journal, January 1997.
[23] S. Niezgoda, L. Holt, D. Wojciech. "Some assembly required: NT's structured exception handling". BYTE 18, 12, pp. 317-322, 1993.
[24] J. Lang, D. Stewart. "A Study of the Applicability of Existing Exception-Handling Techniques to Component-Based Real-Time Software Technology". ACM Trans. on Programming Languages and Systems, Vol. 20, No. 2, March 1998,pp. 274-301.
[25] "Itanium C++ ABI". http://www.codesourcery.com/cxx-abi
[26] "C++ ABI for Itanium: Exception Handling". http://www.codesourcery.com/cxx-abi/abi-eh.html
[27] Tool Interface Standards (TIS) "DWARF Debugging Information Format Specification Version 2.0". TIS Committee May 1995