Exception Handling in the Choices Operating System

Francis M. David, Jeffrey C. Carlyle, Ellick M. Chan, David K. Raila, and Roy H. Campbell

University of Illinois at Urbana-Champaign, Urbana IL 61820, USA,
{fdavid,jcarlyle,emchan,raila,rhc}@uiuc.edu,
http://choices.cs.uiuc.edu/

Abstract. Exception handling is a powerful abstraction that can be used to help manage errors and support the construction of reliable operating systems. Using exceptions to notify system components about exceptional conditions also reduces coupling of error handling code and increases the modularity of the system. We explore the benefits of incorporating exception handling into the Choices operating system in order to improve reliability. We extend the set of exceptional error conditions in the kernel to include critical kernel errors such as invalid memory access and undefined instructions by wrapping them with language-based software exceptions. This allows developers to handle both hardware and software exceptions in a simple and unified manner through the use of an exception hierarchy. We also describe a catch-rethrow approach for exception propagation across protection domains. When an exception is caught by the system, generic recovery techniques like policy-driven micro-reboots and restartable processes are applied, thus increasing the reliability of the system.

1 Introduction

Improving operating system reliability has always been a hot research topic [1,2,3,4]. In today's world, this goal is more important than ever before because the need for better networking, security, performance and features has resulted in increasingly complex systems. Resilience to faults, in both hardware and software, is an important factor affecting reliability [5]. There are many serious errors that can occur inside an operating system [6]. Deadlocks, race conditions, and buffer overflows are some software errors that can cause operating system failure. Hardware errors, such as memory access errors, can also occur in a running system. Most operating systems will terminate user processes that encounter hardware errors; however, the operating systems themselves usually crash if they encounter such errors when executing critical kernel code [7].

On detecting critical errors in kernel code, Linux, Mac OS and several other UNIX-like operating systems call a panic function which brings the system to a halt. Microsoft Windows displays a kernel stop error message in a blue background [8]. Some of these signaled failures [9] might be avoidable through techniques like micro-rebooting [10], automatic data structure repair [11] or device

C. Dony et al. (Eds.):Exception Handling, LNCS 4119, pp. 42–61, 2006.

driver recovery [12]. These techniques can be deployed more effectively when a framework for detecting and communicating errors within the operating system kernel is available.

Exceptions are currently widely used in user-space code to handle software error conditions. We were motivated to explore the benefits of incorporating exception handling support in an operating system in order to build a more robust and reliable kernel that can detect and recover from errors in both hardware and software. Creating software exceptions upon encountering hardware errors and allowing them to be handled by operating system code in the same manner as explicitly thrown software exceptions results in a flexible and unified approach for conveying both hardware and software errors.

In this paper, we explore the feasibility and limitations of using exception handling in an operating system. In particular, we describe our experiences using C++ exception handling support in the object oriented Choices operating system [13]. Exceptions can be thrown for any detected error that can be attributed to a process. This paper specifically addresses errors signaled by the processor. Processor exceptions generated by errant code or faulty hardware are converted to software exceptions and are thrown using C++ language semantics. This is accomplished with no modifications to the compiler or the run time library.

Choices has support for switching memory protection domains within the kernel in order to isolate parts of the kernel from each other. This support is similar to the work done in the Nooks project [4]. To allow exceptions to work across protection boundaries, they are caught at the entry point of the callee domain and are re-thrown in the caller domain.

Exceptions present a strong framework for building recovery solutions in the operating system. Exception handlers are used to enhance Choices with support for automatically restartable processes and a simple form of micro-rebooting. Policies are used along with these recovery mechanisms and provide more flexibility in controlling and managing them. In addition to these generic recovery mechanisms provided by Choices, exceptions also allow kernel developers to write custom localized error handlers.

Once the exception handling framework was implemented, less than 100 lines of code in the Choices kernel were required to implement automatically restartable processes and micro-reboots. Without the use of exceptions, implementing these recovery mechanisms would involve the design of complex asynchronous signaling schemes to send and receive error notifications. The semantics of such a scheme might be unfamiliar to new developers. Exception handling abstracts away complexity and presents a clean and widely accepted means to manage errors.

Our exception framework exploits the polymorphism provided by C++ and is easily portable and maintainable. Machine and processor independent code in Choices only work with abstract exception objects. This results in more maintainable code when newer exception objects are added to a particular architecture. The code is also portable because machine and processor independent code need not change when exception support is added to newer architectures.

An important advantage of using C++ exceptions to clean up after a serious operating system kernel error is the automatic language-supported garbage collection during stack unwinding. The C++ exception handling framework automatically calls destructors for objects associated with a stack frame when unwinding the stack. This provides some benefits by reducing memory leaks when a kernel process is terminated by an exception.

There is a trade-off between space and performance when switching between different C++ compiler implementations of exceptions. The performance overhead when using exception handling is negligible if the compiler implementation of exceptions uses tables [14] instead of time consuming context saves and restores [15]. We evaluate this trade-off in more detail in section 5.

Researchers have worked on including exception handling support in the Linux kernel [16]; however, unlike our work, they have not incorporated support for generating C++ exceptions from hardware errors. We compare and contrast our work with other related research in more detail in section 6.

Our contributions in this paper include:

1. A unified framework for handling both hardware and software generated errors in kernel space code using exceptions.
2. A design for handling exceptions across virtual memory protection domains within the Choices kernel.
3. A description of initial experiences with operating system recovery mechanisms, namely automatically restartable processes and micro-reboots, that can be deployed using exceptions.
4. An evaluation of the space and performance overhead associated with the use of exception handling in an operating system kernel.

The remainder of this paper is organized as follows. Section 2 presents a brief introduction to exceptions and the Choices operating system. It also describes the terminology used in this paper. In section 3, we discuss our design and implementation of the exception handling framework within Choices. Section 4 illustrates some applications and usage of the exception framework. We then present results of some experimental evaluations in section 5, explore related work in section 6 and conclude in section 7.

2 Background

2.1 Exceptions

Exceptions are events that disrupt the normal execution flow of a program. Languages like Java and C++ provide constructs for programmers to write code to both generate exceptions and handle them. In C++, an exception is generated using the `throw` keyword. The `catch` keyword is used to define a code block that handles exceptions.

Exceptions have several advantages [17] over traditional mechanisms for conveying errors. The use of exception handling allows software developers to avoid

return value overloading and clearly separate error handling code from regular code. Using error codes to signal error conditions either requires ugly global variables or requires propagating them down the call stack, sometimes through methods that do not care about them. Exceptions, on the other hand, are directly dispatched to methods that have handler code. Yet another benefit of using exceptions is apparent in the object-oriented world. When exceptions are expressed using objects, class hierarchies can be used to classify and group error conditions.

2.2 The Choices Operating System

Choices is a full featured object-oriented operating system developed at the University of Illinois at Urbana-Champaign. The Choices kernel is implemented as a dynamic collection of interacting objects. System resources, policies and mechanisms are represented by objects organized in class hierarchies. The system architecture consists of a number of subsystem design frameworks [18] that implement generalized designs, constraints, and a skeletal structure for customizations. Key classes within the frameworks can be subclassed to achieve portability, customizations and optimizations without sacrificing performance [19]. The design frameworks are inherited and customized by each hardware specific implementation of the system providing a high degree of reuse and consistency between implementations.

Choices has been ported to and runs on the SPARC [20], Intel x86 [21] and ARM platforms. Similar to User Mode Linux [22], a virtual machine port called Virtual Choices [23] which runs on Solaris and Linux has also been developed. Choices is currently being used as a learning tool in operating systems courses at the University of Illinois at Urbana-Champaign. It is also being used as an experimental platform for research in operating systems for mobile devices and multi-core processors.

2.3 Terminology

Since there is varied usage of the terms used in this paper, we clearly define the terms we use. The terms fault, error and failure are used as defined in [9].

As shown in figure 1, processor interrupts can arise from both hardware and software sources. Peripheral devices can use hardware interrupts to communicate asynchronous events to the processor. For instance, a serial port may interrupt the processor to indicate it has received data.

In addition to hardware sources of interrupts, the software running on a processor may also directly cause an interrupt. Software sources of interrupts can be classified into two categories. Processors generally provide a means for invoking an interrupt via the instruction set. The x86 architecture provides the INT instruction, and the ARM architecture provides the SWI instruction. This mechanism is used to implement system calls. The other category is processor exceptions. A processor exception indicates that some sort of "exceptional" event has occurred during execution. There are many causes of processor exceptions,

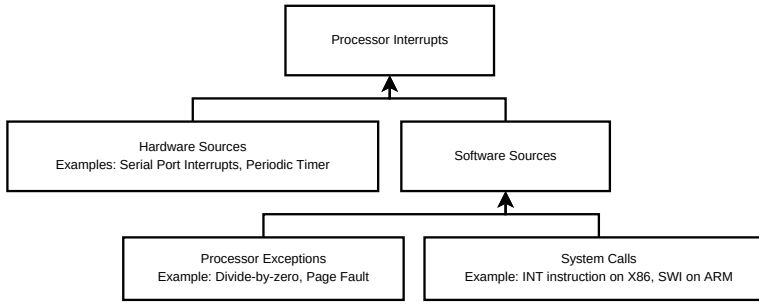

Fig. 1. Terminology

and possible processor exceptions vary between architectures. Example processor exceptions include division by zero, execution of an undefined instruction, and page faults. It is important to note that not all processor exceptions indicate errors. For instance, page faults are usually handled by the operating system and only result in an error if the faulting process has no valid mapping for the requested page.

In the following sections, words that are in italics, like *Process*, are Choices classes.

3 Exception Handling in Choices

Our kernel-level exception handling support has been implemented in the ARM port of Choices. The Choices ARM port currently runs on hardware development platforms based on the Texas Instruments OMAP [24] architecture. Choices also runs under the QEMU [25] ARM system emulator which emulates the hardware found in the Integrator family [26] of development boards. Since most of our exceptions code is processor and machine independent, porting it to another platform is not a difficult task. Also, a large portion of work described in this paper is OS independent and not tightly coupled with Choices. Several ideas and techniques presented in this paper are directly applicable to other operating systems as well. This is discussed in more detail in section 7.

3.1 Creating C++ Exceptions from Processor Exceptions

The first implementation of Choices predates the introduction of exceptions into the C++ language. Therefore, C++ exceptions were not used in the code. We added support for C++ language exceptions to Choices by porting Choices to compile under the GNU g++ compiler which automatically includes all of its exception management libraries.

When using exception handling in an operating system, the use of language exceptions to represent error conditions resulting from processor exceptions is a

natural next step. This results in a seamless and uniform framework for working with operating system error conditions. C++ "catch" statements would then be able to handle serious processor exceptions like invalid instructions which might be caused by memory corruption or other programming faults. There is, however, a caveat when handling errors caused by memory corruption. If the memory corruption affects the exception handling code or support data itself, recovery may be difficult. Other than this scenario, exceptions are a useful error management mechanism. The following paragraphs describe our implementation of exception handling functionality in Choices.

Choices has a per processor *InterruptManager*. It manages all of the initialization, handler registration and dispatch of hardware interrupts, software interrupts and processor exceptions in a unified manner. All processor interrupts are first delivered to the *InterruptManager*. The *InterruptManager* then dispatches the interrupt to a pre-registered handler. The *InterruptManager* is a machine independent abstract C++ class. It only includes code for handler registration and dispatch, and it delegates hardware initialization and interrupt number lookup to machine specific subclasses.

Choices registers special handlers with the *InterruptManager* for all processor exceptions like invalid instructions or memory access errors that require the generation of language exceptions. When the processor is handling an interrupt, the context of the interrupted process is available in a *Context* object associated with the *Process*. If the interrupt is due to a processor exception that signals an error, the registered special handler is invoked. The handler creates an appropriate *Exception* object and associates it with the *Process* object representing the currently running process. The *Exception* object includes a saved copy of the current context and a stack backtrace. This information is useful for debugging. The handler then updates the program counter (PC) in the *Context* of the interrupted process to point to the `throwException` function (see listing in figure 2). The special processor exception handler now returns, and the context of the interrupted process (with the modified PC) is restored to the processor. This causes the process to immediately jump to `throwException`.

The code in `throwException` extracts the saved exception object from the *Process* object and throws it. Thus, it appears as if the process called a function throwing a C++ exception at the exact instruction address where it encountered a processor exception. This implementation allows for precise identification of the instruction that encountered the error and allows for a smooth translation of

```
void throwException() {
  // Throw saved exception object
  throw thisProcess()->getException();
}
```

Fig. 2. Code for the function that throws the exception

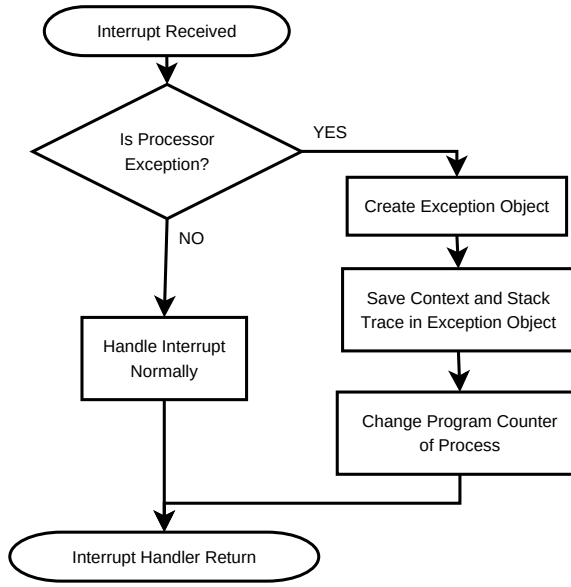

Fig. 3. Processor exception control flow

processor exceptions to C++ exceptions. Figure 3 illustrates the flow of control in the interrupt handler for a processor exception.

Another implementation option that was considered was directly modifying the processor exception causing instruction and replacing it with a call to `throwException`. The idea was to execute the calling instruction on returning from the interrupt handler. On processors with separate instruction and data caches, this technique would require a cache flush in order to ensure that the modified instruction is correctly fetched when execution is resumed. But this design is not safe when there are multiple processes that share the same code. This implementation was therefore discarded in favor of the PC modifying technique.

Since processor exceptions may occur anywhere in the code, the stack unwinding libraries must be prepared to handle exceptions within any context. This functionality is enabled by a special GNU g++ compiler flag, "-fnon-call-exceptions". Without the use of this flag, only explicitly thrown exceptions are handled. This option is typically used from user-space code that attempts to throw exceptions from signal handlers. In the kernel, this allows an exception to be correctly dispatched even in the absence of an explicit throw call.

Unlike the x86 architecture, which detects a large set of processor exceptions such as divide-by-zero, protection faults, device unavailable, invalid opcode, alignment checks and so on, the ARM processor only classifies exceptions into three types. The C++ exception class hierarchy in Choices for the ARM processor is modeled in figure 4. *ARMDataAccessException* is thrown when the processor encounters an error while trying to fetch data. This is the result of a

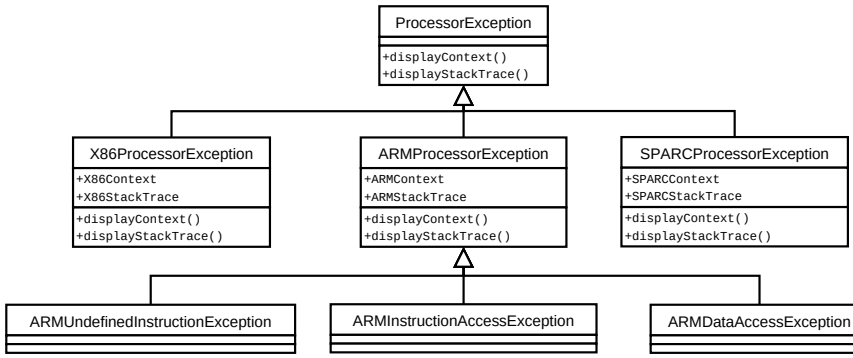

Fig. 4. Processor exception classes

page fault when virtual memory is enabled. *ARMInstructionAccessException* is thrown when the processor encounters an error while trying to fetch an instruction. This is also called a prefetch abort. *ARMUndefinedInstructionException* is thrown when the instruction is invalid. This is equivalent to the Invalid OpCode exception for x86. All processor exceptions are derived from an abstract superclass called *ProcessorException*. *ProcessorException* exports abstract methods implemented by subclasses to access the context and stack trace of processor-specific exceptions. This classification allows processor-specific code to respond differently to different types of exceptions. The use of the abstract superclass allows machine and processor independent code to catch all processor exceptions independent of the architecture and handle them using generic mechanisms.

3.2 Cross Domain Exceptions

Choices supports a simple form of virtual memory based protection domains within kernel code in order to detect errors because of invalid accesses to memory. This was designed to isolate code within the kernel similar to the Nooks project [4] and run device drivers in untrusted isolated domains. Protection is possible for I/O regions, kernel text sections and other defined sensitive code or data regions. For example, the serial port driver is the only object in the system that is allowed to access the serial port I/O addresses. Wrapper objects are used to switch the domain associated with a process when calling a method on an object associated with a different domain. When the method call is completed, the wrapper switches the process back to its original domain. The stack and the heap of the process are also switched when switching to an untrusted domain. They are switched back when returning to the caller. But, the normal method call return is not the only possible return path. Switching back to the calling domain is also necessary when an exception is thrown. In order to correctly switch back to the calling domain in the presence of exceptions, the wrapper catches all exceptions thrown in the called domain and rethrows them in the calling domain

after switching back domains. Objects allocated in the callee domain are also visible in the calling domain. There is therefore no need for copying of exception objects between the domains.

The implementation uses user-mode privileges for untrusted domains. A protection domain in Choices only protects against errors in software code like invalid accesses to memory. We have not yet considered deliberately malicious code. We are in the process of refactoring more device drivers to isolate them into untrusted domains.

Unlike other systems that support cross domain exceptions, like Java, there is no need for serialization or deserialization of the exception objects. This is because the protection domains in Choices are virtual memory based and exception objects are visible in the calling domain. There exist no network or language representation issues that force the need for a serialized representation of objects.

4 Applications

4.1 Stack Garbage Collection

An important advantage when using exceptions to recover after a serious error is the automatic language supported garbage collection of stack objects during unwinding. The C++ exception handling framework automatically cleans up objects associated with a stack frame when unwinding the stack. This prevents some memory leaks and is useful in user processes. Using language exceptions in kernel processes brings these benefits to the operating system. For example, when a *HashTable* object is created on the stack by a kernel process, it requests heap memory to store internal data structures. This heap memory is reclaimed only when the destructor of the object is called. If exception handling is not used, and the fault is handled by just terminating the process, the stack is discarded and the heap allocated memory cannot be reclaimed. Exception handling does not eliminate all possible memory leaks during stack unwinding. Memory leaks can still occur when some objects allocated on the heap are not reclaimed because references to them were not stored into stack objects before the exception was thrown.

This issue does not arise in user-space because the entire process memory is reclaimed when the process is terminated. Microkernel operating systems are also less prone to this problem because system processes are treated like user processes and have access to their own private heap memory. In this case, terminating a kernel process correctly reclaims all associated memory. The Choices kernel, like most other monolithic operating system kernels, is susceptible to such memory leaks because it shares the same heap allocator for the trusted kernel domain across all kernel processes. Using exception handling reduces these memory leaks, which would otherwise persist until a reboot and result in poor memory utilization.

4.2 Restartable Kernel Processes

Transient memory errors due to cosmic radiation or buggy hardware can cause an operating system process to crash. Restarting processes afresh after they crash is usually an effective recovery technique. Microsoft Windows, for example, allows user-space services to be configured so that they are automatically restarted when they crash. The same principle can also be applied to kernel processes in Choices. With the exception handling framework in place, it is easy to write a wrapper for processes to be automatically restarted if an unhandled exception unwinds to the first stack frame.

A *RestartableProcess* class encapsulates the entry point of a restartable process and wraps a C++ try catch block around the initial start call to the process. The try catch block is enclosed in an infinite loop. This results in the ability to create infinitely restartable kernel processes within Choices. The process is terminated only when it explicitly calls the `die()` function. This implementation of restarts differs from a normal terminate and re-create because it does not destroy the *Process* object. The same object is reused and continues to represent the restarted process. Restart time is only governed by the time taken to run the exception handling routines. Thus, restarts through exceptions could be faster than restarts through terminating and re-creating processes if the stack does not contain many objects that need to be destroyed.

Exception handling also helps reduce memory leaks during restarts as described in 4.1. Without exceptions, repeated restarts that leak memory would eventually result in a full kernel heap, rendering the system unusable.

4.3 Micro-reboots

A simple form of micro-rebooting [10] is also easily implemented using exception support. We implement micro-reboots as function call level retries with an optional reboot (reconstruction) of the callee object. This is different from the process level restarts described in the previous section.

When calling into exclusively owned objects that might cause an exception, a *MicroReboot* wrapper is used. The wrapper is implemented as a preprocessor macro. It is designed to retry the request if it encounters an exception. Some errors however, will occur again on a simple retry if the state of the object is corrupted. It is possible that checking and repairing object state could fix the fault and prevent it from occurring again. Objects can export a method called `sanitize` that cleans up internal state. If an exception is raised within an object that exports this method, the wrapper runs the state sanitizer and retries the request. If state sanitization isn't supported, the object is micro-rebooted, that is, destroyed and reconstructed. The current implementation places the burden of writing a state sanitizer on the developer, but this does not preclude the possibility of including an automatic state cleanup mechanism like automatic data structure repair [11] or environment modifying mechanisms like RX [27].

Our implementation of micro-reboots only applies to objects that are restored to correct state when destroyed and reconstructed. This requirement, however, applies to all micro-rebootable systems [10].

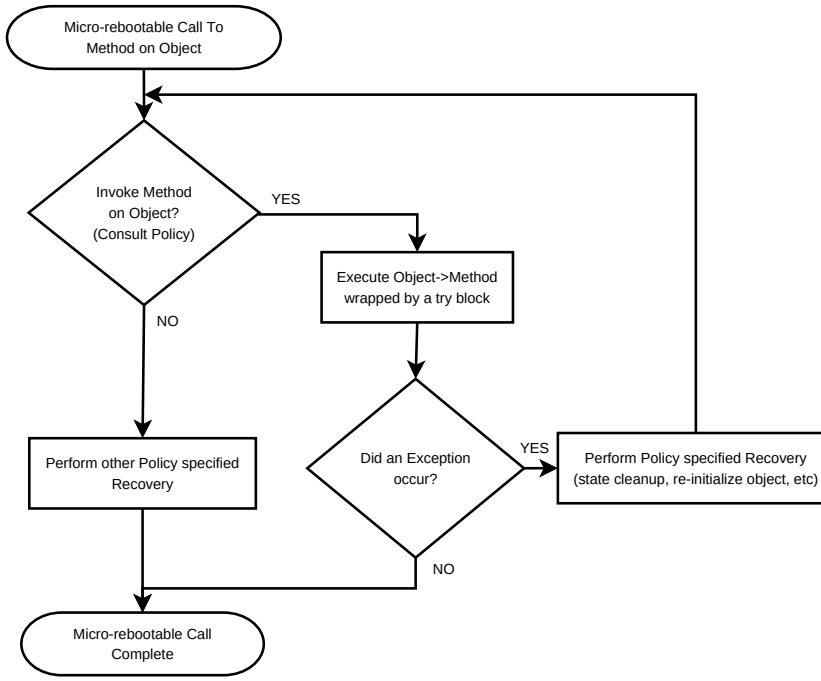

Fig. 5. Policy-driven micro-reboots

4.4 Policy-Driven Restarts

The implementation of infinitely restartable processes described in 4.2 does not provide much flexibility in control or management. In order to provide developers more control over restartable processes, this implementation has been extended with support for specification of simple policies that govern restarts. For example, a policy can specify a limit on the number of times a process is restarted. A similar policy specification is also available for controlling micro-reboots. The flow of control for policy-driven micro-reboots is shown in figure 5. Policy specification requires process-specific semantic information and is therefore entrusted to kernel developers. Our current implementation stores policy information in integers which are used as arguments to the *RestartableProcess* object or the micro-reboot macro.

Policies can be made more expressive when encapsulated in an object. Exceptions can interact with these policies and provide helpful feedback for recovery. For example, this allows a stack overflow exception to be handled by increasing the size of the stack before the restart. This is somewhat similar to the effect achieved in the RX work [27] which improves recoverability by changing environment parameters and restarting. Policy objects are not yet implemented in Choices.

4.5 Improved Error Reporting

The use of exception handling is attractive to many kernel developers because it provides a simple mechanism for providing detailed information about an error condition and allows the developer to avoid return value overloading. For instance, consider the `kmalloc` function in the Linux kernel. `kmalloc` is used to allocate kernel addressable memory. The return value of this function is overloaded. If the return value is non-null, the value is a pointer to the newly allocated memory. A null return value indicates that an error was encountered; however, there is no indication as to why the call failed. Using exceptions, it is easy to create and throw an *Exception* object with rich information about the cause of the error and there is no reason to overload the meaning of the return value.

This method of reporting errors does have its drawbacks. It conflicts with some object oriented design paradigms [28] because implementation information may be exposed to objects earlier in the call chain that were interacting with an abstract interface. Using a class hierarchy for exceptions is useful in this case because components at high abstraction levels can interact with abstract exceptions. System developers should also carefully design their code so that exceptions are handled at appropriate abstraction levels.

5 Evaluation

5.1 Compiler Implementations

The GNU g++ compiler supports two different implementations for C++ exceptions. The first implementation is extremely portable [15] and is based on the `setjmp/longjmp` (SJLJ) pair of functions. This implements exceptions using a `setjmp` call to save context at C++ try blocks and in all functions with local objects that need to be destroyed when an exception unwinds stack frames. The `longjmp` call is used to restore saved contexts during stack unwinding. Since the context saves are performed irrespective of whether an exception is eventually raised or not, this implementation suffers a performance penalty. For example, this penalty is observed when the `new` operator is used to allocate memory from the heap. The standard library version of `new` is designed to throw a "bad_alloc" exception if the allocation fails. This design results in a context save at the beginning of every memory allocation request. Using SJLJ exceptions in an operating system also requires special precautions because the implementation uses state variables that need to be updated when switching contexts.

The DWARF2 [29] debugging information format specification allows for a table driven alternative implementation of exceptions that only executes extra code when an exception is actually thrown. There is no performance overhead during normal execution. This approach uses a static compiler-generated table which defines actions that need to be performed when an exception is thrown. Table 1 compares these two implementations. The trade-off in this approach is size for performance. Modern compilers implement exceptions using the table driven approach for better performance.

Table 1. Comparing SJLJ and Table-Driven implementations of exceptions

Characteristic	**SJLJ**	**Table-Driven**
Portability	Portable	Not portable
Normal execution performance	Affected because of frequent context saves	Not affected because tables are computed at compile time
Exception handling performance	Fast because context restore is cheap	Slower because of table based unwinding
Space overhead	Code to save contexts	Table entries for unwinding

We have studied the size and performance impacts of our exception handling framework under both SJLJ and table-driven implementations. To test SJLJ exceptions, we build an ARM compiler from the published GNU g++ source code and enable SJLJ exceptions as part of the build process. For table driven exceptions, we use a version of the GNU g++ compiler published by Codesourcery [30] that implements table driven exceptions and also conforms to the ARM Exception Handling ABI [31]. We use the same version (4.1.0) of both these compilers.

5.2 Space Overhead

Adding exception handling support results in a larger operating system kernel. Researchers have reported kernel size increases of about 10% when adding exception handling to the Linux kernel [16]. Similar numbers for Choices are unavailable because the GNU g++ compiler does not allow disabling of exception handling support for C++ code. In this section we compare Choices kernels using SJLJ exceptions with kernels using table-driven exceptions. For each of these implementations, we build two versions of the kernel. One version only supports normal explicitly thrown exceptions. The other version includes support for mapping processor exceptions to language exceptions by using the "-fnon-call-exceptions" compiler flag.

All kernels were compiled to the ELF format with g++ optimization level 2. We measure the size of the .text section which holds program instructions, the .data section which holds program data, the .bss section which holds uninitialized

Table 2. Section sizes (in bytes) for different exception handling implementations

Choices	**SJLJ**		**Table-Driven**	
ELF section	Normal	Processor Exceptions	Normal	Processor Exceptions
.text	1,252,980	1,296,176	1,063,600	1,066,484
.data	29,056	29,056	28,500	28,500
.bss	297,984	297,984	297,740	297,740
exception data	9,476	10,980	117,364	131,284
everything else	275,868	288,152	272,044	274,512
Total	1,865,364	1,922,348	1,779,248	1,798,520

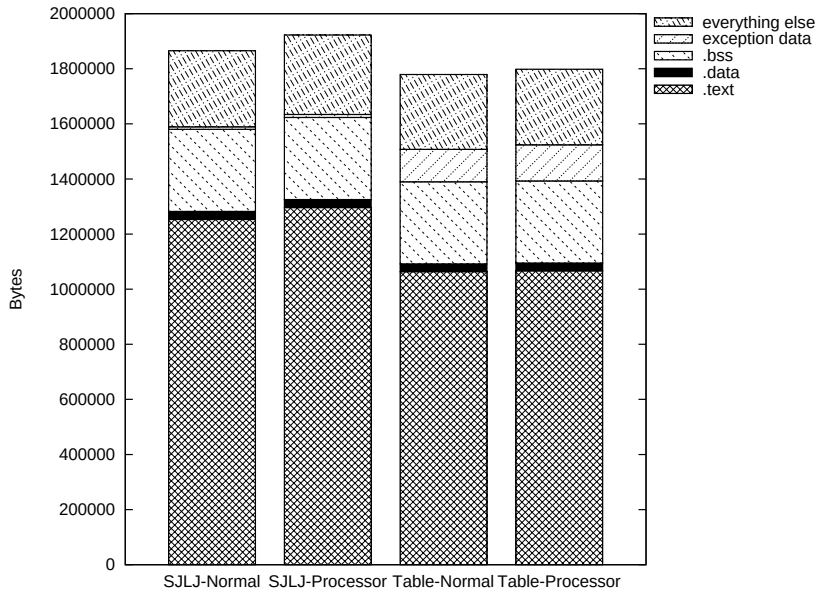

Fig. 6. Size comparison of Choices using different exception handling mechanisms

data and the size of the sections holding exception handling data, such as tables and indices. Table 2 shows the results of our measurements. The sizes are also displayed in graphical form in figure 6.

The figure shows that the use of SJLJ exceptions increases the size of the program text area compared to table-driven exceptions. This is because of the extra instructions for saving and restoring context that are inserted into all functions that define local objects. A small portion of the kernel (0.5%) is reserved for data that is used during exception handling. Adding support for mapping processor exceptions results in just a 3% increase in the size of the kernel. This increase is due to some extra exception handling code and data.

The kernel compiled with table-driven exceptions is about 4% smaller in size. There is a significant reduction in the size of the program text compared to SJLJ exceptions due to the elimination of extra instructions for saving context. But this reduction in size is offset by the large number of exception table entries. It is possible to reduce this overhead using table compression [14]. When processor exceptions support is added, the size of the kernel only increases by 1%.

5.3 Performance

In this section, we present the results of several performance tests of our kernel extensions. All experiments were conducted on the Texas Instruments OMAP1610

Table 3. Time (in microseconds) for creating and restarting processes

Operation	**SJLJ Exceptions**	**Table-Driven Exceptions**
Create *Process*	6529	6330
Create *RestartableProcess*	6543	6347
Restart *Process*	6345	6200
Restart *RestartableProcess*	4320	4515

H2 hardware development kit which is based on an ARM926EJ-S processor core. The processor clock frequency is set to 96MHz.

The overhead of using a single try block in a function was measured to be 14 nanoseconds for table-driven exceptions. This is less than the time taken to run two hardware instructions on the processor. Thus, table-driven exceptions have no noticeable impact on performance. For SJLJ exceptions, this overhead was measured to be 833 nanoseconds. This reflects the significant number of extra instructions that are executed by this implementation to support exception handling. But this overhead is still small compared to the time taken for most high-level kernel operations in Choices. For example, creating processes takes thousands of microseconds.

Table 3 shows the performance impact of using SJLJ exceptions versus table-driven exceptions for restartable processes. The overhead involved in creating a restartable process compared to a normal process is negligible for both cases. The table also shows the performance of restarts implemented through exceptions and the performance of restarts implemented through a terminate and re-create approach. These measurements are for a process that encounters an error when the call stack is one level deep and there are no objects on the stack. For both SJLJ and table-driven exceptions, the *RestartableProcess* performs slightly better because of the extra overhead involved in re-creating a normal process when it dies. It is possible that the stack unwinding for a *RestartableProcess* might take longer than re-creating the *Process* object if the call stack is deep or if a large number of objects need to be destroyed. SJLJ exceptions performs better for a restart of a *RestartableProcess* because exception propagation using `longjmp` is faster than unwinding the stack using tables.

Micro-reboot is implemented as a pre-processor macro and uses a single try block. The overhead associated with a micro-rebootable function call is essentially the overhead associated with the single try block. The overhead for domain crossings is similar to that of micro-reboot calls. These overheads are negligible when using table-driven exceptions.

5.4 Recovery Effectiveness

The automatic restart and micro-reboot wrappers have been extensively tested with test processes and manually injected faults. These wrappers were very effective at recovering from errors in the test cases. Initial results when converting some Choices kernel processes like the dispatcher to automatically restartable

processes are encouraging. The dispatcher is what picks up ready processes from the scheduler and runs them. It is a critical Choices process that cannot afford to crash. In the original implementation, any error in the dispatcher resulted in the process being killed, rendering the system unusable. This behavior was changed by converting the dispatcher to an infinitely restartable process. In a QEMU based fault-injection experiment that was performed 1000 times, we randomly insert transient memory abort errors (both instruction and data) into the dispatcher. We found that the process was restarted and Choices was able to continue running other processes 78.9% of the time. We counted a restart as successful if Choices was able to load and run a program without errors after the restart. The failure to recover in some cases was due to locks that were held by the dispatcher when it crashed. Releasing acquired resources before restarting the process would increase the possibility of successful recovery.

We are in the process of gathering data to evaluate the effectiveness of the micro-reboot wrapper in recovering from errors within Choices. We expect our data to further support the findings in previous work [32,10] discussing the effectiveness of this technique in enhancing reliability.

6 Related Work

Motivated by the desire to use language frameworks for handling exceptions, researchers in Reykjavik University have added C++ style exception handling to the Linux kernel [16]. But their implementation is for the procedural C language based Linux environment. This unfortunately means that they cannot take advantage of rich exception objects and object oriented exception hierarchies. They have also not yet designed a mechanism to map processor errors to C++ exceptions. They have, however, optimized the exception handling code in order to increase performance. This and other techniques on enhancing exception handling performance [33,34,14,35] are complementary to our work and can be used to improve the performance of our system as well.

In UNIX-like systems, application programs are notified of issues like memory errors or divide by zero errors through signals instead of exceptions. Though it is possible to use compiler support to write signal handlers that throw exceptions, this behavior is deemed to be implementation defined in the ISO C++ standard [36]. Choices does not use signals, and exceptions are thrown from a function that is run by the errant *Process*. The SPIN operating system [37] written in Modula-3 also uses exception handling within the kernel, but processor errors are not mapped to language exceptions and instead cause the running process to be terminated.

Systems like CORBA [38] and Java also include support for cross-domain exceptions. These are implemented over heavyweight mechanisms like remote procedure call (RPC) and require serialization and deserialization of exception objects. Our implementation makes use of zero-copy transfer of exception objects between domains.

Most run-time systems choose to terminate a thread or the entire process if there is an unhandled exception. Microsoft's .NET framework [39] allows users to define global handlers for unhandled exception events. This avoids writing code to catch generic exception objects throughout the application. Java also allows for the specification of a handler which is called when a thread abruptly terminates due to an uncaught exception. Instead of terminating the thread, our design allows for the thread to automatically restart by resuming execution from its entry point. This design can also be easily implemented in other systems. Policy-driven exception handling has also been studied in other systems. Researchers have proposed a generic framework to design policy-driven fault management in distributed systems [40]. Recently, there has also been some work in designing policy-driven exception handling for composite web services [41].

There is a plethora of related work in fault tolerance. Researchers have been working on fault tolerance for decades. A NASA funded technical report [42] classifies existing techniques into single version techniques and multi-version techniques. Exception handling as described in this paper uses a single version of the software to try and recover from faults. Jim Gray described various techniques used to increase the mean time between failures for computer systems and concluded that transactions in addition to persistent process-pairs result in fault-tolerant execution [43]. Micro-rebooting [10] and automatic data structure recovery [11] are also single version techniques. Shadow device drivers [12] swap a faulty driver with a shadow driver for recovery. Ignoring errors for failure oblivious computing has also been explored [44]. Hypervisor based fault tolerance [45] runs OS replicas in sync with each other using a hypervisor. A reactive approach [46] builds self-healing software by detecting faults in an emulator and preventing that fault from occurring again. Error isolation and containment by using virtual memory protection has also been studied for device drivers [4]. Multi-version techniques include recovery blocks [47] and N-version software [48].

7 Conclusions and Future Work

Many of the ideas described in this paper are generic and are applicable to systems other than Choices. Creation of language exceptions from processor exceptions is possible in most operating systems that support only explicit exceptions. In particular, adding processor exception handling support to the exception enhanced Linux kernel can be accomplished by a technique similar to the PC-modifying method described in this paper. This would allow similar designs for restartable processes and micro-reboots to be used in Linux. Our designs for restartable processes and micro-reboots can also be used in user level application processes for some classes of exceptions. We are investigating the creation of a framework library that provides these mechanisms to application software.

Some exceptions are not recoverable through mechanisms described in this paper. Undefined instructions, for example could be encountered if there is corruption in memory. In this case, recovery may entail reloading the appropriate page of the executable from disk. Further research is needed in this area. We

have also not considered exceptions that might be triggered in destructors during stack unwinding or in the exception handlers themselves. This is a limitation of our work and needs further investigation. Exceptions, while providing a unified framework for communicating faults, do not ensure that the operating system is in a safe state. Fail-stop operation of the kernel might avoid corruption of data, but could also lead to loss of data. Research is needed to explore the safety of the operating system in the presence of exceptions and recovery mechanisms.

The implementation of micro-reboots described in this paper relies only on exception handling for recovery. It is possible that the object causing the crash might have corrupted external state. Recent research [49] has shown that there is a good likelihood that operating systems can be recovered to a correct state if checkpoints are used. If micro-reboots are combined with checkpoints, external state corruption can be reduced. This is an avenue for future research.

We have described a unified framework that allows the use of exceptions in an operating system to manage errors caused by both hardware and software. We have also explored exceptions based extensions to Choices that support recovery through policy-driven automatic process restarts and micro-reboots and shown that they exhibit acceptable performance. These results indicate that exception handling presents a strong foundation for building reliable operating systems.

Acknowledgments

We would like to thank Gabriel Maajid for helpful discussions during the writing of this paper. We are also grateful for the detailed comments provided by the anonymous reviewers which helped shape the final version of this paper. Part of this research was made possible by a grant from DoCoMo Labs USA and generous support from Texas Instruments and Virtio.

References

1. Randell, B.: Operating Systems: The Problems of Performance and Reliability. In: Proceedings of IFIP Congress 71 Volume 1. (1971) 281–290
2. Denning, P.J.: Fault Tolerant Operating Systems. ACM Comput. Surv. **8**(4) (1976) 359–389
3. Lee, I., Iyer, R.K.: Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System. In: FTCS. (1993) 20–29
4. Swift, M.M., Bershad, B.N., Levy, H.M.: Improving the Reliability of Commodity Operating Systems. In: Proceedings of the nineteenth ACM Symposium on Operating Systems Principles, New York, NY, USA, ACM Press (2003) 207–222
5. Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., Treuhaft, N.: Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical report, Berkeley, CA, USA (2002)
6. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.R.: An Empirical Study of Operating System Errors. In: Symposium on Operating Systems Principles. (2001) 73–88

7. Ganapathi, A.: Why Does Windows Crash? Technical Report UCB/CSD-05-1393, EECS Department, University of California, Berkeley (2005)
8. Blue Screen. (`http://support.microsoft.com/kb/q129845/`)
9. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.E.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing **1**(1) (2004) 11–33
10. Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., Fox, A.: Microreboot – A Technique for Cheap Recovery. In: Symposium on Operating Systems Design and Implementation, San Francisco, CA (2004)
11. Demsky, B., Rinard, M.: Automatic Data Structure Repair for Self-Healing Systems. In: Proceedings of the First Workshop on Algorithms and Architectures for Self-Managed Systems, San Diego, California (2003)
12. Swift, M.M., Annamalai, M., Bershad, B.N., Levy, H.M.: Recovering Device Drivers. In: Symposium on Operating Systems Design and Implementation. (2004) 1–16
13. Campbell, R.H., Johnston, G.M., Russo, V.: "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)". ACM Operating Systems Review **21**(3) (1987) 9–17
14. de Dinechin, C.: C++ Exception Handling. IEEE Concurrency **8**(4) (2000) 72–79
15. Cameron, D., Faust, P., Lenkov, D., Mehta, M.: A Portable Implementation of C++ Exception Handling. In: USENIX C++ Conference, USENIX (1992) 225–243
16. Glyfason, H.I., Hjalmtysson, G.: Exceptional Kernel: Using C++ Exceptions in the Linux Kernel. (2004)
17. Advantages of Exceptions. (`http://java.sun.com/docs/books/tutorial/essential/exceptions/advantages.html`)
18. Campbell, R.H., Islam, N., Johnson, R., Kougiouris, P., Madany, P.: *Choices*, Frameworks and Refinement. In Luis-Felipe Cabrera and Vincent Russo, and Marc Shapiro, ed.: Object-Orientation in Operating Systems, Palo Alto, CA, IEEE Computer Society Press (1991) 9–15
19. Russo, V.F., Madany, P.W., Campbell, R.H.: C++ and Operating Systems Performance: a Case Study. In: USENIX C++ Conference, San Francisco, CA (1990) 103–114
20. Raila, D.: The Choices Object-oriented Operating System on the Sparc Architecture. Technical report, The University of Illinois at Urbana-Champaign (1992)
21. Lee, L.: PC-Choices Object-oriented Operating System. Technical report, The University of Illinois at Urbana-Champaign (1992)
22. Dike, J.: A user-mode port of the Linux kernel. In: Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, Georgia (2000)
23. Tan, S., Raila, D., Liao, W., Campbell, R.: Virtual Hardware for Operating System Development. Technical report, University of Illinois at Urbana- Champaign (1995)
24. Texas Instruments OMAP Platform. (`http://focus.ti.com/omap/docs/omaphomepage.tsp`)
25. Bellard, F.: QEMU, a Fast and Portable Dynamic Translator. In: USENIX Annual Technical Conference, FREENIX Track. (2005)
26. ARM Integrator Family. (`http://www.arm.com/miscPDFs/8877.pdf`)
27. Qin, F., Tucek, J., Sundaresan, J., Zhou, Y.: Rx: Treating Bugs as Allergies - A Safe Method to Survive Software Failures. In: Symposium on Operating Systems Principles. (2005) 235–248
28. Miller, R., Tripathi, A.: Issues with Exception Handling in Object-Oriented Systems. In: European Conference in Object-Oriented Computing. (1997)

29. Tools Interface Standards: DWARF Debugging Information Format. (http://www.arm.com/pdfs/TIS-DWARF2.pdf)
30. Codesourcery. (http://www.codesourcery.com)
31. Exception Handling ABI for the ARM™architecture. (http://www.arm.com/miscPDFs/8034.pdf)
32. Candea, G., Fox, A.: Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. In: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, Washington, DC, USA, IEEE Computer Society (2001) 125
33. Drew, S., Gouph, K., Ledermann, J.: Implementing zero overhead exception handling. Technical report, Queensland University of Technology (1995)
34. Schilling, J.L.: Optimizing away C++ exception handling. SIGPLAN Not. **33**(8) (1998) 40–47
35. Thekkath, C.A., Levy, H.M.: Hardware and software support for efficient exception handling. In: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California (1994) 110–119
36. ISO 14882: C++ Programming Language. (http://www.iso.ch/)
37. Bershad, B.N., Savage, S., Pardyak, P., Sirer, E.G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C.: Extensibility, Safety and Performance in the SPIN Operating System. In: 15th Symposium on Operating Systems Principles, Copper Mountain, Colorado (1995) 267–284
38. Vinoski, S.: Distributed Object Computing with Corba. C++ Report Magazine (1993)
39. Microsoft .NET. (http://www.microsoft.com/net/)
40. Katchabaw, M.J., Lutfiyya, H.L., Marshall, A.D., Bauer, M.A.: Policy-driven fault management in distributed systems. In: Proceedings of the The Seventh International Symposium on Software Reliability Engineering, Washington, DC, USA, IEEE Computer Society (1996) 236
41. Zeng, L., Lei, H., Jeng, J.J., Chung, J.Y., Benatallah, B.: Policy-Driven Exception-Management for Composite Web Services. In: Proceedings of the 7th IEEE International Conference on E-Commerce Technology. (2005) 355–363
42. Torres-Pomales, W.: Software Fault Tolerance: A Tutorial. Technical Report NASA/TM-2000-210616, NASA Langley Research Center (2000)
43. Gray, J.: Why do computers stop and what can be done about it? In: Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems. (1986) 3–12
44. Rinard, M., Cadar, C., Dumitran, D., Roy, D.M., Leu, T., William S. Beebee, J.: Enhancing Server Availability and Security Through Failure-Oblivious Computing. In: Symposium on Operating Systems Design and Implementation. (2004) 303–316
45. Bressoud, T.C., Schneider, F.B.: Hypervisor-based fault tolerance. ACM Trans. Comput. Syst. **14**(1) (1996) 80–107
46. Sidiroglou, S., Locasto, M.E., Boyd, S.W., Keromytis, A.D.: Building a Reactive Immune System for Software Services. In: USENIX 2005 Annual Technical Conference. (2005)
47. Randell, B.: System structure for software fault tolerance. In: Proceedings of the International Conference on Reliable Software. (1975) 437–449
48. Avizienis, A.: The N-Version Approach to Fault - Tolerant Systems. In: IEEE Transactions on Software Engineering. (1985) 1491–1501
49. Chandra, S., Chen, P.M.: The Impact of Recovery Mechanisms on the Likelihood of Saving Corrupted State. In: ISSRE. (2002) 91–101