Implementation of exception handling

David Chase

Originally appeared in JCLT, June, 1994

1 Intro

Exception-handling is a feature found in one form or another in various languages. Recent examples include Ada, Modula-2+, C++, Modula-3 and Eiffel. These languages all share the "termination model" of exception handling, where control transfers into the scope of the handler and activation records separating the exceptional event from its handler are discarded[1]. One important goal for exception-handling is that the unexceptional case should execute as quickly as possible, and it is generally assumed that execution of the exceptional case may be made more expensive to further this goal (the Modula-3 report provides guidelines, both to implementors and programmers—it suggests that 10000 instructions may be spent in the exceptional case to save one instruction in the normal case). Another important goal is that a debugger should be able to manipulate exceptional events—given an exception, it should be able to determine where the exception will be caught (if anywhere) and if an exception will not be caught, then the debugger should be notified of this event in the context in which the exception was raised. It is desirable that exception-handling not impede optimization of programs, and there are situations in which support for asynchronous exceptions would be useful.

This article discusses implementation techniques for exception handling, and how these goals are met (or not). Specific examples (at the machine-code level) will be presented in terms of Sparc assembly language and the Sparc System V Release 4 Processor-specific Applications Binary Interface (abbreviated as "the ABI"). The ABI (and machines which implement it, or the very-similar SunOS 4.x calling conventions) are widely available, and the author is extremely familiar with the details of this architecture and ABI. (The important things to know for most examples in this article are that `call` writes the current PC into register `%o7`, executes the next (delay slot) instruction, and transfers control to the target of the call. The standard callee-side linkage includes a `save` instruction, which renames register `%o7` to be `%i7`.)

I don't know of any good references describing most of the techniques discussed in this paper. Most of what I learned, I either learned from or in the

[1]Cedar Mesa and PL/1 both permit "resumption", where the failing operation may be retried in the original context. Eiffel allows "resumption", but only of the failing block within the same routine as the handler which catches the exception.

process of working with (in chronological order) Mick Jordan, Wayne Gramlich, Kinman Chung, Howard Gayle, Peter Kiehtreiber, and Mike Ball, while working on Modula-2+, Modula-3, a Sparc optimizing code generator, a C++ runtime system, and a proposed Sparc version 9 ABI.

2 Mapping of high-level exception handling

At the source language level, exception handling typically allows *finalization*, *re-raising*, and *return*. Finalization corresponds, more or less, to user-defined unwinding; if a block B is exited, then a piece of code C must also be executed. If C is entered because an exception exiting B was thrown, then unwinding continues in enclosing blocks and activation records. In Modula-3 the syntax for this is `TRY` B `FINALLY` C `END`; in C++ the code C is the set of destructors for objects stack-allocated in B. There is no special re-raising feature in Modula-3, but in C++ an unadorned `throw;` rethrows the currently active exception and in Ada `RAISE;` does the same. An exception is said to return when control exits the source level handler via normal (unexceptional) flow of control.

I could add Eiffel as well.

All languages discussed here pass some amount of information with a thrown exception, ranging from a simple exception tag (Ada) to an object of arbitrary type and size (C++). It simplifies the presentation of synchronous exception handling to assume that a language-specific run-time system supplies subroutines to push, examine, and pop this information on an exception data stack (this article is primarily concerned with the smooth and efficient interaction of the exceptional transfer of control with the compiler and debugger). The translation of a `throw` or `RAISE` includes code to push the relevant data on the data stack, and the translation of handler which returns to normal code must pop the top of the exception data stack. A stack (and not just a single variable) is required because none of these languages prevents the use of exception handling nested within handlers, destructors or finalization code.

At a lower level within the compiler, the interface to the exception handling system could include the following entry points for raising, reraising, and pushing and popping data:

`__ex_raise(ed,et)` Push `ed` onto the the exception data stack, and enter the exception dispatcher. This is used to implement a source language `throw(x)` or `RAISE e`. The second parameter `et` supplies type information.

`__ex_reraise()` Assuming that there is already data on the exception stack, enter the exception dispatcher using that data. This is used to re-enter the dispatcher after finalization code is executed, or to implement a `throw;` or `RAISE;`.

`__ex_return()` This is used at the normal exit from a source language exception handler. It pops the topmost item off the exception data stack, leaving the stack in the same state it was in before the exception caught here was originally raised.

For example, this piece of C++

```
double foo(int i, int j) throw (int) {
  double rval
  try {
    if (j == 0) throw(i);
    rval = (double) i / (double) j;
  } catch (int) {
    if (i != 0) throw;
    rval = 0.0;
  }
  return rval;
}
```

might be compiled into (ignoring both name-mangling and the gorier details of run-time type information):

```
foo:
      save     %sp,-112,%sp
a:
      tst      %i1               !  Branch to b if j != 0
      bne      b
      nop
      mov      %i0,%o0           !  Parameter i to throw
      set      TC_int,%o1        !  Type code for ``int''
      call     __ex_raise        !  Will ``return'' to c
      nop
b:
      ...                        !  Int-to-float conversion omitted
      fdivd    %f4,%f2,%f0
      ba       e
      nop
c:                               !  Code for catch clause
      tst      %i0               !  Branch to d if i == 0
      be       d
      nop
      call     __ex_reraise      !  ``throw;'' -- does not return
      nop
d:
      call     __ex_return       !  Tell dispatcher to discard its data
      sethi    %hi(DC_zero),%g1  !  Load a constant zero
      ld       [%g1+%lo(DC_zero)],%f0
e:
      jmp      %i7+8             !  Also written as ``ret''
      restore
```

Again, note that the code implementing the full C++ exception-handling semantics has been omitted; this code is intended only to illustrate the control flow. This code also lacks the annotations needed to make the exceptional transfers of control actually happen.

From the compiler's point of view, there is nothing special about `__ex_raise` and `__ex_reraise`. These are functions like any other, with similar calling conventions, and may be treated as raising exceptions like any other. In particular, in the case of blocks nested within blocks, it may be very convenient to treat the call to `__ex_reraise` occurring within an inner block as raising an exception within an outer block. It is often possible to optimize such nested reraising into a simple `goto`, but this optimization is not required.

In the case of asynchronous exception handling, it becomes difficult (probably impossible) to make any sort of sensible guarantee about correct maintenance of the stack, or whether all exceptions can even be properly accounted for. This assumes a model where asynchronous exceptions can occur at any point in the program (including within the handler itself) and at arbitrary rates. Partial solutions exist, but they are complicated and depend upon details of the target architecture and binary interfaces. I'll attempt to present one solution in a subsequent article.

3 Basic implementation techniques

All termination-style exception-handling techniques share some basic features. When an exception E is raised, the *dispatcher* must determine what activation record contains a handler for E, and transfer control to that handler in that activation record. This search always involves traversal of a stack, either a special exception stack that is explicitly maintained by the normal-case code as it executes, or by walking up the stack of activation records. An exception-handling implementation should have reasonable answers (such as "yes", "yes", "well", "none", "low") to the following questions.

- Can a compiler and linker generate the information required? (If any is needed.)
- Can the information be made position-inpendent?
- How does this interact with existing languages, standards, and conventions?
- What is the normal case direct overhead? That is, what additional code is executed on behalf of the exception-handling system?
- What is the normal case hidden overhead? For example, what optimizations are inhibited? What extra state (that must be maintained) is associated with a thread or a process?

As a bonus, some of the implementation techniques described here generalize nicely, either allowing asynchronously raised exceptions, or permitting the description (for the sake of debuggers and other unwinders) of non-standard calling conventions.

3.1 Threaded exception stacks

One portable and easy-to-understand technique is based on the use of setjmp, longjmp, and a separate exception stack. This is used in portable versions of the Modula-3 and Eiffel compilers, and macros exist to obtain exception-handling in C itself.

There's a TR from DEC SRC by Eric Roberts that describes this, but I don't have the reference handy.

This approach is simple enough that it can be presented in terms of C source code. The entries on the dynamic exception stack contain a back-link, a `jmp_buf`, and some information used by the exception dispatcher to determine if a particular exception can be caught by this handler (this information is usually the address of a static data structure).

```
struct __ex_stack_entry {
  struct __ex_stack_entry * prev;
  jmp_buf context;
  mumble * handler_info;
  void * exception_scratch;
};
```

To enter a block guarded by an exception handler, it is necessary to capture the current context with `setjmp`, initialize the handler info, and link the stack entry onto a global exception stack. The entry is a stack-allocated local variable. Upon exit from the normal-case code, the topmost entry must be popped off of the exception stack.

```
{
  struct __ex_stack_entry this_entry;
  static mumble this_handler_info;
  static int this_which_ex;
  this_entry.handler_info = & this_handler_info;
  this_entry.prev = global_exception_stack;
  global_exception_stack = & this_entry;
  this_which_ex = setjmp(this_entry.context);
  if (this_which_ex == 0) {
    /* normal case code */
      ...
    global_exception_stack = & this_entry -> prev;
  else {
    /* handle an exception */
    switch(this_which_ex) {
      ...
    }
  }
}
```

If an exception is thrown, the exception dispatcher simply pops entries from the list until it finds one which is able to handle the thrown exception, and then calls `longjmp` to establish the context. Note that the compiler-generated handler information is customized for a given block. Each exception that can be caught is converted into a small integer which corresponds to a case in the switch statement. Any information associated with the exception itself (for instance, a pointer to the exception arguments or to the exception object) is stored in scratch space provided in the exception stack entry.

```
__ex_rt_dispatch (void * ex) {
  struct __ex_stack_entry e = global_exception_stack;
  int match;
  while (0 == (match = handles_exception(e,ex)))
    e = e -> prev;
  if (match == -1) abort();
  global_exception_stack = e -> prev;
  e -> exception_scratch = ex;
  longjmp(e -> context, match);
}
```

The performance of this approach, and the amount of code generated in-line, can be improved by noting that all the exception stack entry initialization can be performed by one customized piece of assembly language passed a pointer to the entry and a pointer to the handler information. However, each normal case entry and exit will still require initialization of the exception entry, as well as pushing and popping for the entry from the exception stack. This is not quite as bad as it appears, given a clever compiler, because most of the initialization of the entry is in fact loop-invariant (if it occurs within a loop)—the only operations which cannot be hoisted are the pushing and popping of the entry from the global exception stack. Of course, this is only possible if the guarded block is lexically within a loop.

As written, this approach is also not thread-safe, because the global exception stack is treated as a global variable. In the case of threads, either the exception stack is stored in thread-specific-data or thread-local-storage (if it is available), or in a global register (if the compilation system permits this), or else the run-time system for the language can manage the stacks itself at thread switch.

There's a POSIX draft standard that describes thread-specific-data, but I don't know the name of it.

Perhaps the major disadvantage to this technique is that it tends to impede optimization. If C is used as an intermediate language, calls to `setjmp` tend to turn off optimization. In addition, the C generator (either person or program) must include `volatile` annotations to protect against those optimizers that are not sufficiently conservative in the presence of calls to `setjmp`. (This is true, in one form or another, at the assembly language level as well.)

This technique does have several advantages. First, it is quite obvious how it works. It is clear what handlers are dynamically in scope, and the data structures can easily be manipulated by debuggers. Second, it does not rely on any special support from the compiler, linker, or run-time system. Third, it is robust in the face of non-standard subroutine linkage. If `setjmp` and `longjmp` work, then this works.

3.2 PC-based techniques

The overhead of initializing, pushing, and popping exception records can be avoided if a different representation is used. The PC-based techniques build data structures indicating which points in the program are interesting to the exception dispatcher (that is, which PCs are within blocks guarded by exception handlers), and the dispatcher refers to these data structures as it walks or unwinds the stack. Conceptually, this is very similar to a syntactic description of exception handling: "If an exception occurs while executing here, then go there".

There are (at least) three ways to encode such data structures. In the case of synchronous exception handling, exceptions occur only at call sites. Either a table identifying those call sites (one site per entry) can be built, or the interesting call sites can be identified within the code stream itself. A more general mechanism is to encode ranges of PCs at which exceptions could occur. This can be extended (with care) to also handle asynchronously raised exceptions.

3.2.1 In-code markers

The major (theoretical) complication to placing markers in the code stream is that existing code generation standards may not have made any provision for this. That is, a "tagged `NOP`" following a call cannot be guaranteed to mean anything, because existing standards may not ban it, and thus the existence of such a tag could just mean that an older compiler happened to emit it. In practice, compilers are not quite so capricious in their code generation, and so it suffices to choose something that they don't already do.

A second complication to in-code markers is that introduction of gratuitous branches (to avoid the markers) or execution of extraneous `NOP`s can slow code down slightly. This is true, but the slowdown is typically less than that incurred by dynamically pushing and popping handlers from a separate stack.

A third complication is that there may be some limit on what data can easily be embedded within a code stream, either because of restrictions on the size of the tag fields, or because of restrictions on the linker relocations that may be in applied to program text, or because of conflicts with other restrictions (for example, position-independence).

One easy solution to two of these problems is to note that often languages with exception handling (for instance, C++ or Modula-3) use "mangled" versions of procedure names, and are not intended to be called from other languages without explicit directives to the compiler (and in fact, a programmer in a different language must work rather hard just to figure out the right name to use). Thus, it is possible to assume that normally only C++ calls C++, and that C++-to-C++ calling calling conventions may differ. For instance, the convention on a Sparc could be that the word following the delay slot following a call to a C++ function contains a handle for the exception information, and that all C++ functions return to `%i7+12`, rather than `%i7+8`. This avoids the cost of an extra branch.

Using this convention, a the call to `__ex_raise` in the earlier example would look like:

```
call      __ex_raise        !  Will ``return'' to c
nop
.word     exception_info_for_this_call
...                         !  Ordinary code after call
```

The exception information for this call would indicate that if the type of the thrown exception is an `int`, then transfer control to `c`. In turn, a normal return from a C++ procedure would be

```
jmp      %i7+12           !  Not ``jmp %i7+8''
restore
```

In this situation, the exception unwinder could walk the stack, deriving a sequence of call-site PC values, $\{\mathtt{pc_1}, \mathtt{pc_2}, \ldots, \mathtt{pc_n}\}$, checking the handlers described by the information at stored at $\mathtt{pc_i} + \mathtt{8}$ to see if it matches. If it does, then that activation record can be established and control transferred to the location indicated by that call-site's information.

Note that this assumes a standard representation for activation records, including spilled registers, so that the exception dispatcher can walk the stack, discover the various PCs, and re-establish the context in which the exception was raised. Using the Sparc ABI, this can be accomplished by spilling register windows to memory, walking the linked list of register save areas, and performing a restore with `%fp` equal to the stack pointer of the desired frame (call it F).

One difficulty here is that F's stack pointer may have been moved if the procedure it called (that is, the procedure from which the exception is propagating into F) uses an unconventional (but perhaps still ABI-conforming) technique for saving F's preserved registers. The conventional method is to use a `save` instruction to get a new register window, but it is also possible to move F's stack pointer and store the preserved registers in the newly allocated stack space. When F's registers are restored from the register save area obtained by walking the stack, they will still all have the correct value, except for the stack pointer. Stack relative temporaries will appear to have been corrupted[2], and stack storage will have leaked.

Thus, if this method is used, then care must be taken to ensure that conventional (and not just standard) methods are used to preserve registers. The other PC-based techniques share this restriction. Fortunately, all three PC-based techniques can also be (ab)used to solve this problem elegantly and efficiently.

A second difficulty for the Sparc ABI is that though the linker recognizes a 32-bit PC-relative relocation that allows the generation of position-independent annotations within the code stream, there is no way to get the assembler to generate it[3].

[2]The routine `alloca` also moves the stack pointer, but it is treated specially by the compiler, and these problems are avoided.

[3]For pre-1994 assemblers. This is also true for SunOS.

3.2.2 PC hash and range tables

The PC-hash and PC-range approaches don't require any change to the calling conventions, and don't require any changes to the code stream. In both cases, the exception dispatcher searches for each return PC in a table. If a matching entry is found (equal PC for hashes, or containing PC interval for a range table), then information in the entry indicates whether this entry also matches the particular exception that was thrown. If it does, then the entry contains (in addition) the location of the handler, and the dispatcher unwinds the stack to that point and transfers to the handler.

PC hash tables are probably more compact than range tables, and can be searched more quickly, but it isn't easy to convince a linker to generate them and they don't generalize to handle asynchronous exceptions. If a table cannot be generated at link-time, then it must be generated at run-time. Unless this hashing is deferred until an exception is actually raised, it can delay program initialization and increase the amount of real memory needed. Range tables generalize easily to handle asynchronous exceptions, but are somewhat slower to search. Range tables can be easier to generate, though this depends upon the object file format. For instance, table generation in SVR4's Extensible Linker Format (ELF) is straightforward, but it is not in "a.out" format.

Here is an example of how the entries in an exception range table might appear in ELF (this is very similar to the exception table entries used by Sun C++ 4.0 on Solaris, except that the assembler syntax has been simplified).

```
.word    range_begin-.
.word    range_end-range_end
.word    handler-range_begin
.word    handler_info+12-.
```

Each entry is entirely PC-relative. The start address of the range is obtained by adding the first word of the entry to the location of the entry. The ending address is obtained by adding the start address to the second word of the entry. The location of the handler is obtained by adding the start address to the third word of the entry. (Note that the beginning and end of the range and the handler are all expected to appear in the same (`.text`) section, and thus will not be further relocated once a library is generated.) The address of the language-specific information which indicates whether an exception should be matched by this entry is obtained by adding the address of the entry to the fourth word of the entry.

For the example from section 2, the single table entry might be:

```
.word    b-.
.word    0
.word    c-b
.word    exception_info_for_this_call+12-.
```

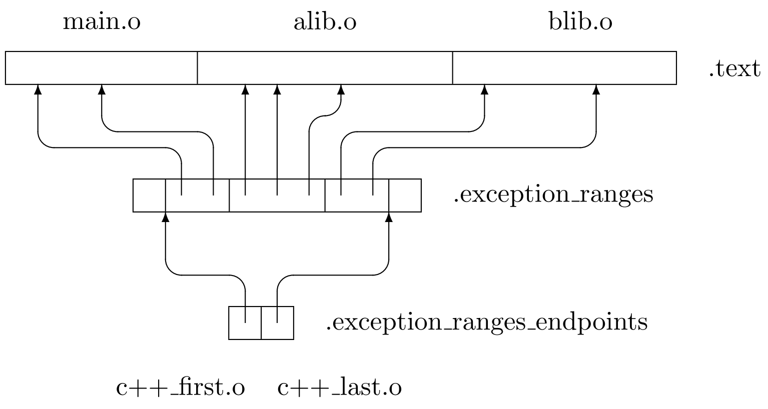
The range begins at `b`, the "normal" return location from the call. Because the exception handling is synchronous (meaning that exceptions are only raised at call sites) it is sufficient to only cover that instruction, so the (inclusive) length of the range is zero. The handler is at `c`, so the difference between `c` and `b` is placed in the third word. The fourth word contains the pointer to the handler information, which indicates that `c` handles `int` exceptions.

When C++ code is generated, the compiler sorts the entries by the beginning addresses of the ranges, and then emits all the entries into a separate section "`.exception_ranges`". When the object files are linked, the different sections (that is, "`.text`", "`.data`", "`.exception_ranges`" are grouped together by section name and concatenated in the same order as the object files which contained them. Thus, the linker automatically generates a single sorted, position-independent PC-range table for the library or program. Because the tables are position-indendent and in a separate section, they require no run-time initialization and may be shared between different activations of the same program or library. If no exception is raised, the tables may not ever appear within a program's working set.

Each program or library also requires some initialization code to register its range table with the dispatcher at run-time, since there is no obvious connection between different sections in a library. This code is contained in two language-specific object files bracketing the rest of the library. For example, a typical link line might look like:

```
ld <flags> c++_first.o main.o alib.o blib.o c++_last.o
```

Entries in yet another section, "`.exception_ranges_endpoints`" appear in the first and last files to identify the endpoints of each library's exception ranges, and code in the `.init` section of the first file registers those endpoints with the exception dispatcher. By registering a program or library's exception range table in the first file's initialization section, any exceptions raised in execution of static constructors (called from subsequent files' `.init` sections) will be properly handled.



4 Next

Next installment will contain a discussion of the interaction of exception handling with non-standard calling conventions, debuggers, asynchronous exception handling, and modeling exceptional transfers of control.