Implementation of exception handling–II
Calling conventions, asynchrony, optimizers, and debuggers

David Chase

Originally appeared in JCLT, October, 1994

1 Non-standard calling conventions

One constraint imposed by the PC-based exception-handling techniques is that the dispatcher must be able to unwind the stack from the point where the exception is raised to the point where it will be handled. This would seem to inhibit the use of non-standard calling conventions. However, the PC-based techniques can themselves be used to solve this problem, provided that exception handling is properly modeled in the optimizer/code generator, and given some minimal cooperation from the exception dispatcher and debugger. To see this, note that exception handling as it appears in C++, Modula-3, Ada, and Eiffel is just a way of talking about stack unwinding. To customize unwinding for non-standard calling conventions, the compiler generates its own unwinders, and inserts entries for them into the same range tables used for language-level unwinders. Of course, there are several details to get right. I'll attempt to illustrate these by way of an example.

1.1 An example solution

The standard Sparc calling convention requires that a called subroutine not alter its caller's registers `%i0-%i7` and `%l0-%l7`. The usual way of accomplishing this is with paired `save` and `restore` instructions. However, these registers could also be saved and restored with explicit store and load instructions, especially if only a few are used, so that entry to a subroutine (call it G) which uses only register `%l0` might look like (assume the body contains a call somewhere):

```
sub     %sp,64,%sp        !  Don't ask.
st      %o7,[%sp+96]      !  Preserve return address.
st      %l0,[%sp+100]     !  We'll use this one.
```

and exit might look like

```
ld      [%sp+100],%l0    !  Restore caller's %l0.
ld      [%sp+96],%o7     !  Restore return address.
add     %sp,64,%sp       !  Leave %sp like we found it
jmp     %o7+8            !  Return.
nop                      !  Unoptimized for clarity
```

In normal (not debuggable) execution, the dispatcher unwinds the stack frame-by-frame, looking for a handler and discarding the frame if none is found. At a call-site within G (above), if an exception is raised, there will be one of four outcomes.

1. The return PC is in a range, and the exception is handled, and the handler returns to normal code. The dispatcher does not need to unwind the frame, so it is not a problem.

2. The return PC is not in any range. The dispatcher needs to discard this frame and try the next one.

3. The return PC is in a range, but not one that handles the exception that was raised. The dispatcher needs to discard this frame (same as the second case).

4. The return PC is in a range, and the exception is handled, but then reraised. This ultimately reduces to the first or second case, because the exception must be reraised (if the compiler is correct) outside the scope (PC range) in which it was last caught.

If this were a conventional frame, the frame can be discarded by executing the following piece of code in the context of the activation record that failed to match an exception.

```
call     __ex_reraise      !  Transfer control back to dispatcher
restore                    !  Discard non-matching frame (delay slot)
```

Habitual readers of Sparc assembly language will recognize this as a tail-call to the exception reraiser from a conventional activation record. The `restore` instruction in the delay slot of the call discards G's frame and establishes the frame of G's caller. In particular, the return address register (`%o7`) is set as if `__ex_reraise` had been called from the location at which G was called (this is, after all, what it means to be a tail call). For the unconventional code, all that is necessary is customized code that has this same effect:

```
ld      [%sp+100],%l0    !  Restore caller's %l0.
ld      [%sp+96],%o7     !  Restore return address.
add     %sp,64,%sp       !  Leave %sp like we found it
mov     %o7,%g1          !  Begin windowless tail call idiom
call    __ex_reraise
mov     %g1,%o7          !  Copy back old %o7 (delay slot)
```

If this code appears as the handler for an exception, then the exception dispatcher will execute it and the frame will disappear all by itself, and the handler will be re-entered from the ancestor frame. For all calls that don't already have a handler, this handler is created.

Now consider calls that do have handlers. Those handlers that return into the current frame are not a problem, since no unwinding is necessary. Those that do not, will eventually call `__ex_reraise`. However, this call is just a call like any other; if the handler is not also nested within some outer `try` block, then it is just another instance of a call without a handler, and its customized unwinding handler will be created as above.

The third case involves those sites where handlers exist, but none happens to match. The choice here depends upon how handler matching is implemented. One way to implement handler matching is to open code it; that is, if there are any handlers at all associated with a site, the dispatcher will transfer to the equivalent of a handler switch statement (running in the same context as the handlers) which selects one handler, or, if none matches, calls `__ex_reraise`. In this case, the transformation is as above. Another way to implement handler matching is to do more work in the dispatcher. In this case, the exception information describes (in some encoding or another) what exceptions may be caught, and if none match, then the dispatcher never transfers to the activation record; instead, it just discards it. To ensure that a non-standard frame is correctly discarded, however, this cannot be done by the exception dispatcher. Instead, the compiler must fill in default (`ELSE`, for Modula-3, or `catch (...)` for C++) handlers for all sets of handlers that lack them, and the default handlers perform the custom unwinding.

From the point of view of the compiler writer, this is an excellent interface, because there is no need to restrict calling sequence optimizations because exception handling might not work. If new optimizations are discovered, they can be introduced without modifying the exception handling system; that is, there are no compatibility problems. Unfortunately, other stack-walking "clients" (such as debuggers) need additional help, because the stack-walking described here is destructive—once a frame has been "walked", it is gone for good. In addition there is no way to tell which "handlers" dismantle stack frames and which handlers perform language-level exception handling—when a debugger changes its focus from one activation record to its ancestor, it should not run destructors and finalization routines.

For other architectures, similar problems are posed by "shrink-wrapping" the saving of callee-saves registers, and by aggressive scheduling of the instructions which save and restore callee-saves registers. All these create the possibility of registers being in some non-standard state at various points within a program.

1.2 Non-destructive stack walking

The two crucial requirements for non-destructive stack walking are the need to distinguish the code which restores activation records from the code which implements language-level exception handling, and the need to run that code

on a copy of the machine state (that is, to run it without destroying activation records).

The choices include:

1. Merely identify the code which performs the activation record unwinding, and require that the debugger interpret the machine instructions on a copy of the machine state until it reaches a branch to a well-defined location (such as `__ex_reraise`).

2. Generate, in the debugging information, an encoding of the steps necessary to restore the ancestor activation record for a given PC range.

3. Generate a second unwinder for the activation record only, which operates on a representation of the machine state. On Unix machines, this might be either a `struct sigcontext` (Berkeley Unix OS variants) or an `mcontext_t` (System V Unix OS variants).

There are various reasons to prefer one technique to another. The first choice is most flexible, and comes "for free" with certain implementations of asynchronous exception handling. On the other hand, it requires that a debugger be able to interpret machine instructions (as opposed to merely loading and storing values from a debugged process).

The second choice would work with a less capable debugger, but it carries with it the risk of limiting (in the expressiveness of its interface) the "creativity" of compiler writers (an unpardonable sin).

Both the first and second techniques are not useful if the execution of the program itself requires stack walking. For instance, one possible implementation of Ada9x's "abort-deferred" blocks walks the stack to see whether an abort should be queued or immediately processed. This is not practical with the first two techniques, and barely practical with the third. For this example, if non-standard linkages are to be used, then it is probably best to use a counter to monitor whether or not aborts are currently deferred.

2 Asynchronous exception handling

Asynchronous exceptions are those that may occur at any point in the program, for no predictable reason, and with arbitrary frequency (this is a worst case). The PC-range exception-handling implementation can be used to provide reliable unwinding (that is, stack will not be corrupted) in the face of asynchronous exceptions provided that two restrictions are met:

1. If the rate of exception delivery exceeds some threshold, some of the exceptions may be discarded. The last exception delivered is never discarded.

2. The *average* rate of exception delivery must be below some threshold in order to guarantee that unwinding makes progress. There is no limit on the peak rate. The threshold is determined by the speed of the underlying machine and the time needed to execute code in small exception handlers.

These restrictions were chosen because they seem to correspond to the semantics of Ada, Modula-3, and C++, extended to the asynchronous case where necessary, and because they place minimal demands on other resources (for instance, interrupt deferral or memory allocation). Other choices may be more appropriate for other applications.

This implementation technique also depends upon certain properties of the underlying machine. It is easiest to explain and implement if each machine instruction is atomic—either it executes to completion with the defined change to machine state, or else it is not yet executed, and the state is not changed at all. This means that the exception dispatcher need only keep track of which instruction was about to execute. A good low-overhead representation of asynchronous exception information is PC range tables, with the addition of nesting to both simplify table generation and to provide an interface for the debugger's unwinding needs.

2.1 One implementation

One way to implement clean unwinding in the face of asynchronous exception handling is to break the subroutine save and restore sequences into smaller restartable chunks, and generate a handler for each of these chunks. By "restartable chunk", I mean that if any proper prefix of the chunk has executed, then the chunk may be restarted from the beginning without changing its observable effect[1]. Only the last instruction in the chunk may not be repeated. The effect of a single-instruction chunk is either execute to completion (once) or not at all.

Given a (simplified) subroutine in which the code takes the form

$$S_1, S_2, \ldots, S_N, \mathit{body}, R_N, \ldots, R_2, R_1$$

where each S_i is a save-sequence chunk and each R_i is a restore-sequence chunk, N handlers are created. The first handler H_1 corresponds to the PC ranges from just before the first instruction of S_2 to just before the last instruction of S_2 (that is, after completion of S_1 but before completion of S_2) and from just before the first instruction of R_1 to just before the last instruction of R_1. The code in H_1 is a copy of the code in R_1.

For i less than N and greater than one, H_i is associated with the range from just after completion of S_i to just before completion of S_{i+1} and all of R_i except the last instruction. The body of H_i is just a copy of R_i, followed by a branch to H_{i-1}. Handler H_N is associated with code from the beginning of *body* to just before the end of R_N, and contains a copy of R_N followed by a branch to H_{N-1}.

Of course, one problem with asynchronous exceptions is that they can also occur during the execution of the handlers themselves. In order to deal with this problem, each handler H_i "handles itself". That is, if an exception occurs

[1]The "observable effect" weasel words are inserted to allow repeated execution of instructions that are known to be "dead" if an exception has been raised. A common example of this is initialization of local variables.

before handler H_i is complete, then H_i is restarted[2]. Because of the way that handlers are constructed, this is guaranteed to be harmless; only execution of the last instruction in a handler cannot safely be repeated.

To work correctly in the presence of asynchronous exceptions, the previous example would use the following handlers. (Remember that the handler associated with an instruction is executed if an exception is delivered *before* execution of the instruction is complete.) This example includes the two additional handlers necessary for tail calls out of leaf routines.

```
S1      sub     %sp,64,%sp          HL1
S2      st      %o7,[%sp+96]        H1
S3      st      %l0,[%sp+100]       H2
        ...                         H3
R3      ld      [%sp+100],%l0       H3
R2      ld      [%sp+96],%o7        H2
R1      add     %sp,64,%sp          H1
        jmp     %o7+8               HL1
        nop                         HL1
!  Begin handlers
H3      ld      [%sp+100],%l0       H3
H2      ld      [%sp+96],%o7        H2
H1      add     %sp,64,%sp          H1
HL1     mov     %o7,%g1             HL1
        call    __ex_reraise        HL1
HL2     mov     %g1,%o7             HL2
        ba,a    HL1                 HL1
```

Each of the handlers except the last two contains only a single instruction, so either the instruction executes, or not. The last two handlers deal with the details of a "leaf-routine tail call". The details are specific to Sparc, but the principles are not, and this also serves to illustrate the importance of deciding which registers must be preserved across exception dispatch. The first instruction of H_{L1} places a copy of `%o7`, which contains the return address, into register `%g1`. Register `%g1` is chosen because it is the only one generally writeable in this situation (the input and local registers are off limits, and the output registers are potentially occupied by parameters, though they are not in this case). The call instruction, if executed, will write the current `PC` value into `%o7`. If it is not executed, then `%o7` may safely be re-copied into `%g1`, so it is also handled by H_{L1}. If an exception arrives before execution of the delay-slot instruction, the return address can only be found in register `%g1`, and it is necessary to take special action to recover it, before trying again to call `__ex_reraise`. This is accomplished with an unconditional branch, itself guarded by its target. Note that the two handlers H_{L1} and H_{L2} are paired with the leaf-tail-call idiom, as

[2]Assume, for the moment, that the compiler-generated handlers are always correct and that the portion of the stack on which they run is at least readable. Otherwise, this could loop infinitely.

opposed to being derived in some algorithmic sense from the patterns of register use within the subroutine.

Note that this example requires that the dispatcher not alter the contents of various registers. Obviously, `%o7` and `%g1` are off limits if leaf tail-calls are to be permitted. Also forbidden are the input and local registers, since those are not allowed to be volatile across calls (according to the ABI) and because an exception may arrive in a called subroutine before they can be saved. Less obviously, the registers used to return values from subroutines are also off limits. The reason for this is that there are plausible situations in which a piece of information necessary for cleanup can only reside in a result register. For example, immediately after a subroutine which opens a file has returned, the only copy of the file descriptor must be in `%o0`. If the file is to be closed in the cleanup code, then that register cannot be altered. In the case of Sparc and its ABI, there are spare parameter registers `%o4` and `%o5` which are never used to return values from subroutines, and thus are suitable for this purpose. This choice assumes that if a piece of code performs some action which might need to be undone in a handler, then any crucial values in those registers will have been copied into stable registers before the action is begun.

2.2 Treatment of calling convention idioms

On potential difficulty with support for asynchronous exceptions is that it appears to impose onerous restrictions on code generators; that is, these intricate tables must be generated for all subroutines. In practice, this is not the case. The requirements on convention-conforming code generators are minimal and simple, given some support for a few special cases in the exception dispatcher. Only use of unconventional register saving techniques requires the more complex treatment described above.

In the case of Sparc, the conventional models for subroutine compilation are one of

1. Normal window. For a normal-window subroutine, the first instruction is a `save`, and the last is a `restore`. This is considered to be the default case by the dispatcher, which means that the compiler need not do anything out of the ordinary. The caller's address is expected to be in `%i7`. Note that before execution of the `save`, the caller's registers are active (as in a leaf routine), but an unadorned `save` can be treated as a special case by the dispatcher.

2. Leaf routine. Within a leaf routine, the caller's register set is still active. The caller's address is expected to be in `%o7`. For reliable asynchronous unwinding, the compiler must make a note of this. However, because leaf routines are common, it makes sense to note "leaf routine" in a single exception entry and deal with it in the dispatcher. This reduces the number of exception table entries generated, and makes life simpler for compilers that emit leaf routines.

Note that the idiom for a leaf tail call (see previous example, with handlers H_{L1} and H_{L2}) contains a single instruction at which `%o7` does not contain the proper return address. If a three-address instruction[3] in a leaf routine writes register `%o7`, it is assumed that the non-zero source operand contains the caller's `PC` instead.

3. Big window. A big window routine is just one where the new stack pointer cannot be generated with an immediate offset. Typically, the first two instructions build a 32-bit constant, and the third is a `save`. In this case, the first instructions can be marked as "leaf", and the rest can be treated as "normal".

Some of these choices require a moderate amount of detail work in the exception dispatcher, but it actually reduces to just a few bit-testing operations to detect the exceptional cases (and thus reduce the burden on existing compilers, as well as the usual-case size of exception tables). This is not a toy list, either—the intent here was to show that a complete treatment for a current popular architecture is not out of reach. On the other hand, enumerating the small number of exceptions-to-the-rule that the dispatcher must treat specially requires careful study of an architecture, its ABI(s), and compiler conventions.

2.3 Disambiguation of synchronous and asynchronous exceptions

Another problem related to proper choice of exception parameter registers is the need to make distinctions between exceptions synchronously propagated out of procedure calls and exceptions asynchronously raised at the return location following a call. Again using Sparc as an example, consider the following piece of code:

```
(1)   call     fopen
(2)   nop
(3)   st       %o0,[%fp-4]
```

Asynchronous exceptions can either be raised at (1) or at (2), before the `fopen` has been entered, or at (3) after `fopen` has returned. Unfortunately, if `fopen` were to synchronously propagate an exception to its caller, that would also appear to be raised at (3). In the synchronous case, it is likely (for the sake of the example, if nothing else) that the file has not been opened, and thus does not need to be closed. On the other hand, if the exception was raised asynchronously, then the file has been opened, and does need to be closed. Clearly, the two situations must be distinguished if exception handling will be reliable. For RISC architectures, one clever hack is to raise exceptions asynchronously at `exception_pc+2`; since all instructions are word-aligned, this cannot be confused with any legal instruction, and because it is larger than the synchronous

[3] In practice, all current Sparc compilers use `add %g0,%reg,%o7` or `or %g0,%reg,%o7` to implement a register-to-register move, and turning this convention into a restriction is not likely to cause any harm.

exception PC, it will appear "after" it in any sorted list of tables, which corresponds to its later occurrence in the execution stream (note that the synchronous propagation location call_pc+8 was chosen so that it would appear "after" any exceptions raised asynchronously at call_pc or call_pc+4).

2.4 Implementation of nested exception tables

There's certainly more than one way to implement nested exception tables. The algorithm given here is relatively simple, but expects relatively shallow nesting depths. This appears to be a reasonable assumption either for compiler-generated unwinders (the maximum nesting depth should be the number of registers saved by the caller) or for most languages, where the nesting depth (of ordinary blocks, not blocks guarded by exception handlers) appears to rarely exceed 15.

An entry in a PC-range table as a start pc, end pc, handler, and handler information (all encoded in a position-independent way, but that is not important here). To add support for nesting of ranges, each entry also needs a parent "pointer" to its enclosing range, if any. The ordering of the entries is modified so that the primary key is the start pc (sorted least first) and the secondary key is the end pc (sorted least last). This means that an entry's parent must occur earlier in the table.

The binary search of the table must be modified slightly to account for the possibility of nested ranges:

```
// Search the table for a range containing ``pc'' from
// indices (including) ``lo'' up-to-but-not-including ``hi''
while (lo < hi) {
  mid = (lo + hi)/2;
  if(begin_pc(mid) > pc) hi = mid;
  else if (end_pc(mid) < pc) {
    if (lo == hi - 1 && lo > first) {
      // See if predecessor block or any range enclosing it
      // contains PC. Because lo = mid = hi-1, incrementing
      // lo will stop future iteration.
      temp = lo - 1;
      while (parent(temp) != 0 && end_pc(temp) < pc) temp = parent(temp);
      if (end_pc(temp) >= pc) return temp;
    }
    lo = mid + 1;
  } else {
    // Don't just assume a hit - a descendant
    // range containing pc takes priority.
    if (lo == hi - 1) return lo;
    else lo = mid + 1; // Overshoot now, step back later (above).
  }
return -1;
```

The compiler can use nested tables to separate the programmer-specified unwinding code from the unwinding code that is necessary to move from acti-

It would be nice to discuss exactly how many instructions must be executed to discard a frame. Some people might like some more assurances that this terse little algorithm is correct, too.

vation record to activation record in a standard-conforming way. Each entry is tagged (either within the entry itself, or perhaps in the handler information) to indicate whether it is programmer-specified or compiler-gnerated. To walk from one activation record to the next, the debugger must scan the range containing the current PC to find any that were compiler-generated, and simulate the effects of those handlers to obtain the next stack frame. In the case that the compiler-generated ranges are themselves nested within other ranges, it is necessary that the compiler-generated handlers end by calling `__ex_reraise` so that the debugger can detect where each handler ends[4]. When ranges are nested like this, it is also necessary that the range table entries have some indicatation of which ones correspond to an unwind into the ancestor activation record, and which ones are internal to an activation record. Without this, the debugger would not be able to tell the difference between unwinding within a single activation record and unwinding across a self-recursive call.

3 Modeling exceptional transfers of control

It is important to pay attention to both explicit and implicit exception-handling overheads for the common case. The techniques described so far avoid any explicit normal-case overhead, but missed optimizations in the compiler can represent a substantial implicit overhead. For example, in the presence of unknown control flow, a compiler is forced to use a more conservative register allocation, or to avoid the use of registers altogether. This is an implicit overhead of exception-handling. In C, a compiler is allowed to demand that the programmer use "`volatile`" to identify variables that must not be allocated into registers, but this is not an option in C++, Modula-3, or Ada. Furthermore, poor information about control-flow inhibits the use of various optimizations.

The "obvious" answer for C++, where exceptions may only be raised synchronously, is to model the exceptional transfers of control within the compiler. This will add a number of edges to the control flow graph, but no more than are necessary. Furthermore, within the compiler, these transfers-of-control must be "first class"—it should be possible to modify their destinations to accommodate the needs of various optimizations (optimizations of the common case code, that is). A consequence of this is that the optimizer/code generator (basically, the last part of the compiler to transform the code in any significant way) must be responsible for generating the PC range tables. Another reason why this is necessary is that some optimizations may either rearrange or replicate code (including calls), which may in turn either rearrange or replicate entries within the PC range tables.

Imposing a synchronous-or-wrongly-compiled restriction is probably unrealistic, but the restriction is difficult to remove while still doing a good job of optimizing code. In some cases, threads can be used to avoid asynchronous control flow. In other cases, it may be sufficient to require that programmers use

[4]Recall that it is sometimes possible to optimize this into a branch.

`volatile` to indicate which variables are live across the throwing of an asynchronous exception. Even this is not necessarily enough; lacking any procedure calls within a try block, a C++ compiler might eliminate all the handlers completely (because there is no synchronous flow-of-control into the handlers, and thus they are "unreachable code"). It is also possible that the information in PC-range tables might be too precise, in the sense that it might only identify points (call sites) where exceptions might be raised, and not the ranges containing those call sites. Any exception raised in the range, but not at a call site, would appear to have no handler in the local activation record. It appears difficult to provide both good optimization and seamless support for asynchronous exception handling.

On the other hand, the problems are limited to the activation record in which the (asynchronous) exception is first raised. The compiler-generated unwinding information for removing stack frames is still correct, and will still be executed, and in the next activation record the exception will appear at a call site—that is, it will propagate synchronously.

4 Why bother?

One obvious question that must have occurred to most readers by now is: "Why bother? What's all this complexity buy me, especially since the support for asynchronous exceptions in C++ is not-quite-right?"

One reason to do this is to obtain (or at least not forbid) compatibility with other exception-handling systems. If people wish to mix Ada and C++, they should be able to, and it should be possible for each language to make some minimal sense out of the other language's exceptions.

Another reason is to enable the highest possible performance for standard-conforming code. The PC-range exception handlers have no explicit overhead for the normal case, and minimal implicit overhead if the optimizer is allowed to assume synchronous-exceptions-only.

These calling conventions also help in making an ABI more robust, and allow more room for innovation. The ability to guarantee that certain actions will be performed before a subroutine is exited permits the use of an abort/commit programming style, assuming the existence of a language which supports this. That is, no operation, whether it completes or fails, ever leaves a process in an undefined state. These guarantees also provide room for implementation innovation (insert example here).

5 Odds and ends

Using an altered calling sequence within C, one can also implement exception-handling by passing the address of an exception-state location. Routines which have no exception handlers of their own need only check it after each call, returning immediately if it is non-zero. Routines with handlers will save the

old value, pass on their own (the address of a local in their frame) and choose handlers based on the value of the variable.