C++ ABI for IA-64: Exception Handling

Revised 9 September 1999

Revisions

[990909] Original version.

General

In what follows, we define the C++ exception handling ABI, at three levels:

- the base ABI, interfaces common to all languages and implementations;
- the C++ ABI, interfaces necessary for interoperability of C++ implementations; and
- the specification of a particular runtime implementation.

Definitions

The descriptions below make use of the following definitions:

Base ABI

The minimal level of specification is effectively that of the definition in the IA-64 Software Conventions document. It describes a framework which can be used by an arbitrary implementation, with a complete definition of the stack unwind mechanism, but no significant constraints on the language-specific processing. In particular, it is not sufficient to guarantee that two object files compiled by different C++ compilers could interoperate, e.g. throwing an exception in one of them and catching it in the other.

It is intended that nothing in this section be specific to C++, though some parts are clearly intended to support C++ features.

Data Structures

Exception Handle

An exception handle is an address pointing to an exception object consisting of an exception control header followed by user exception information. The handle points to the user information, so the control information is accessed at a negative offset from it. The control information consists of:

```
language-specific information
Unexpected_Handler      unexpectedHandler;
Terminate_Handler       terminateHandler;
Exception_Destructor    destructor;
```

The runtime may at any time call unexpectedHandler to report an unexpected exception; if NULL, it should call terminateHandler. The runtime may at any time call terminateHandler to terminate execution; if NULL, it should call exit. The runtime or the eventual user code handler may call destructor to destroy and if necessary deallocate the exception

object; it may be NULL if no destruction is necessary.

Unwind State Handle

The stack unwind process maintains its current state via an opaque structure containing unwound machine state, accessible only via a procedural API, TBD.

Exception Handler Framework

The standard ABI exception handling / unwind process begins with a call to __eh_throw, described below. This call specifies an exception object, and an exception class (a __uint_64).

The runtime framework then starts a two-phase process:

- In the search phase, the framework repeatedly calls the personality routine, with the EH_SEARCH_PHASE API described below, first for the current PC and register state, and then unwinding a frame to a new PC at each step, until the personality routine reports either success (a handler found) or failure (no handler). It does not actually restore the unwound state, and the personality routine must access the state through the API. If failure is reported, it ... (calls terminateHandler?).
- If the search phase reports success, the framework restarts in the cleanup phase. Again, it repeatedly calls the personality routine, with the EH_CLEANUP_PHASE API described below, first for the current PC and register state, and then unwinding a frame to a new PC at each step, until the personality routine reports success. At that point, it restores the register state, and branches to the PC, which has been set by the personality routine to the landing pad address.

Throwing an Exception

Exceptions are initially thrown, after creating an exception object, by calling:

```
typedef enum {
    EH_THROW_INITIAL,
    EH_THROW_RETHROW,
    EH_THROW_RESUME
} EH_THROW_KIND;

void __eh_throw (
    EH_THROW_KIND kind,
    __uint_64   exception_class,
    void *      exception_object,
    ...
);
```

<To Be Specified>

Personality Routine

The personality routine has the following prototype:

```
typedef enum {
    EH_SEARCH_PHASE,    // Search phase call
    EH_CLEANUP_PHASE    // Cleanup phase call
} EH_PHASE;
typedef enum {
    EH_HANDLER,         // Handler landing pad found
    EH_PROCEED,         // Proceed to next frame
    EH_UNEXPECTED,      // Unexpected exception
    EH_TERMINATE,       // Terminate thread/program
    ...
} EH_RESULT;

EH_RESULT personality (
```

```
        int         version,
        EH_PHASE    phase,
        __uint_64   exception_class,
        void *      exception_object,
        void *      language_specific_data_area,
        Unwind_State *unwind_state
    );
```

If called with phase==EH_SEARCH_PHASE, the personality routine may:

- Determine that control should be passed to user code, for any purpose including a real catch or cleanup and resume, in which case it returns EH_HANDLER. In this case, the framework starts over with the unwind process, now passing EH_CLEANUP_PHASE.
- Determine that the current exception is unexpected and processing cannot proceed, in which case it returns EH_UNEXPECTED. In this case, the framework ...
- Determine that the program must be terminated, in which case it returns EH_TERMINATED. In this case, the framework calls terminateHandler if non-NULL, or exit() otherwise, neither of which will return.
- Determine that nothing interesting happens in this frame, in which case it returns EH_PROCEED. In this case, the framework unwinds a frame and calls it again.

If called with phase==EH_CLEANUP_PHASE, the personality routine may:

- Determine that control should be passed to user code, for any purpose including a real catch or cleanup and resume, in which case it uses the API to set the PC and any registers which need to contain data, and returns EH_HANDLER. In this case, the framework restores the register state and transfers control to the PC in Unwind_State. This user code may either simply proceed with program execution, or may perform cleanup actions and rethrow/resume.
- Determine that the current exception is unexpected and processing cannot proceed, in which case it returns EH_UNEXPECTED. In this case, the framework ...
- Determine that the program must be terminated, in which case it returns EH_TERMINATED. In this case, the framework calls terminateHandler if non-NULL, or exit() otherwise, neither of which will return.
- Determine that nothing interesting happens in this frame, in which case it returns EH_PROCEED. In this case, the framework unwinds a frame and calls it again.

Unexpected Exception Handler

Define the prototype and behavior of the Unexpected_Handler. Delete the exception object? Rethrow?

Terminate Handler

Define the prototype and behavior of the Terminate_Handler. Delete the exception object or pass it to the handler? Perhaps pass all of the personality parameters to the handler so debug traceback is easy?

Unexpected Exception Handler

Define the prototype and behavior of the Destructor. Do we need a size in the standard exception object header?

Debugging

An idea: If we were to define a special value of exception_class to identify a debugging unwind, and put a debug entry in the exception object, the personality routines could support language-specific debug traceback.

C++ ABI

The second level of specification is the minimum required to allow interoperability in the sense described above. This level requires agreement on:

- Standard runtime initialization, e.g. pre-allocation of space for out-of-memory exceptions.
- The layout of the exception object created by a throw and processed by a catch clause.
- When and how the exception object is allocated and destroyed.
- The API of the personality routine, i.e. the parameters passed to it, the logical actions it performs, and any results it returns (either function results to indicate success, failure, or continue, or changes in global or exception object state), for both the phase 1 handler search and the phase 2 cleanup/unwind.
- How control is ultimately transferred back to the user program at a catch clause or other resumption point. That is, will the last personality routine transfer control directly to the user code resumption point, or will it return information to the runtime allowing the latter to do so?
- Standard runtime initialization, e.g. pre-allocation of space for out-of-memory exceptions.
- Multithreading behavior.

Common C++ Runtime Implementation

The third level is a specification sufficient to allow all compliant C++ systems to share the relevant runtime implementation. It includes, in addition to the above:

- Format of the C++ language-specific unwind tables.
- APIs of the functions named __allocate_exception, __throw, and __free_exception (and likely others) by HP, or their equivalents.
- API of landing pad code, and of any other entries back into the user code.
- Definition of what HP calls the exception class value.

The vocal attendees at the meeting wish to achieve the third level, and we will attempt to do so. Whether or not that is achieved, however, a second-level specification must be part of the ABI. *<To be specified.>*

Please send corrections to Jim Dehnert.