SJLJ EH:
C++ exception handling in PNaCl using setjmp()+longjmp()

Mark Seaborn
September 2013

Overview

PNaCl currently does not support C++ exception handling ("**EH**") at all for ABI-stable pexes. The PNaCl toolchain supports EH only when doing offline translation to a nexe, in which case it uses **zero-cost EH**.

In the long term, we want PNaCl to support zero-cost EH for ABI-stable pexes. However, as a stop gap, I propose to implement EH using setjmp()+longjmp ("**SJLJ**"). SJLJ EH has an overhead at runtime, but it will be quick to implement, and since it won't require ABI changes (because PNaCl already provides setjmp()+longjmp()), it will allow EH-using pexes to run in Chrome M31.

Some of the work involved in implementing SJLJ EH will be reused for implementing ABI-stable zero-cost EH, so implementing this stop-gap solution won't be wasted work.

> *Background:* "Zero-cost EH" means that enabling EH isn't supposed to incur any run

time costs on a program until it throws a C++ exception. In particular, there isn't supposed to be any run time overhead for entering and leaving a "try" block, or for calling a function that might throw an exception, because the compiler doesn't generate any extra code that runs on the non-exceptional code path. The only overhead is an increase in executable size, because the compiler will add extra data tables and code for handling exceptional cases.

Example of SJLJ code transformation

Consider the following C++ fragment:

```
int catcher_func() {
        try {
                int result = external_func();
                return result + 100;
        } catch (MyException &exc) {
                return exc.value + 200;
        }
}
```

(See Appendix C for the concrete LLVM that Clang compiles this to.)

I will add a PNaClLowerInvoke IR pass which will convert this to the following SJLJ-using pseudo-code. Note that unchanged code is in **bold**:

```
struct ExceptionFrame {
        jmp_buf jmpbuf;  // Context for returning to catch block
        struct ExceptionFrame *next; // Next frame in linked list
        void *action_list;  // Pointer to action list

        // Data returned to the catch block:
        void *exception;
};

__thread struct ExceptionFrame *__pnacl_eh_stack;  // Thread-local exception state

// RTTI for C++ exception.  Generated by Clang.
const std::type_info typeinfo_for_MyException = { … };

// Description of what the landingpad (catch block) wants to handle.
// Generated by PNaClLowerInvoke.
const char action_list_for_landingpad1[] = { … &typeinfo_for_MyException … };
```

```
int catcher_func() {
        struct ExceptionFrame frame;
        int result;
        if (!setjmp(&frame.jmpbuf)) {  // Save context
                frame.next = __pnacl_eh_stack;
                frame.action_list = &action_list_for_landingpad1;
                __pnacl_eh_stack = &frame;  // Add frame to stack
                result = external_func();
                __pnacl_eh_stack = frame.next;  // Remove frame from stack
        } else {
                // Handle exception.
                // This is a simplification.  Real code would call __cxa_begin_catch()
                // to extract the thrown object.
                MyException &exc = *(MyException *) frame.exception;
                return exc.value + 200;
        }
        return result + 100;
}
```

Overview of changes

Implementing SJLJ EH support will involve the following changes:

- Add a “PNaClLowerInvoke” IR pass. This pass will do two things:
    - **Lower control flow:** Expand each “invoke” instruction to a code sequence that pushes an exception frame into a thread-local stack and saves the current execution state using setjmp().
        - See example above.
    - **Lower C++ exception info:** Encode the action list in the “landingpad” instruction as data in the pexe’s data segment. This data will be interpreted by user code in the pexe, so its format will not be part of PNaCl’s stable ABI.
        - See below.
        - This part will be reused when we implement ABI-stable zero-cost EH support.

- Change the C++ library (or libraries) to use SJLJ.
    - This will involve changing GNU libsupc++ (bundled with GNU libstdc++), or LLVM’s equivalent, libcxxabi (used with LLVM’s libc++).
    - I’ll add a user-code implementation of _Unwind_RaiseException() which searches __pnacl_eh_stack for a matching handler. This function will contain functionality which is currently implemented in the C++ personality function.

- Minor tweak to pnacl-ld.py to expose EH-related symbols.

    - The list BCLD_ALLOW_UNRESOLVED currently contains symbols such as “memcpy” which ABI simplification passes operate on. We’ll need to add “__pnacl_eh_stack” to this list so that PNaClLowerInvoke can use the definition provided by libsupc++.

- Add the option “--exception-handling={none|zerocost|sjlj}” to pnacl-clang.
    - Deprecate or remove “--pnacl-enable-exception-handling”.
    - We want to keep the option of “--exception-handling=zerocost” so that developers who’re currently using zero-cost EH with offline translation (e.g. QuickOffice) can continue to do so without getting a performance regression from switching to SJLJ EH.
    - We’d like to keep the option of “--exception-handling=none” (which is the current behaviour) so that developers can strip all C++ EH from their pexes to reduce the pexe size if they know they’re not relying on EH. This is different from using “-fno-exceptions”:
        - It works at link time, whereas “-fno-exceptions” is a compile-time option.
        - It can strip EH code paths from toolchain libraries too (e.g. from libstdc++) without recompiling them.
        - It doesn’t give an error when using “try” or “throw” in C++ source, whereas “-fno-exceptions” does.

- Improve test coverage of EH on the PNaCl bots. See below.

EH test coverage

I’m currently testing the SJLJ EH implementation against two sets of tests:

- Targeted EH tests I’ve written myself. I plan to add these to the NaCl Scons build.
- GCC’s torture tests (from gcc/testsuite/g++.dg/eh).
    - PNaCl currently runs a subset of these on the PNaCl toolchain bots. Unfortunately, PNaCl runs only 38 out of 106 tests: the rest specify special test runner options via DejaGNU comments (of the form “// { dg… }”) and don’t work under PNaCl’s simple torture test runner.

I also intend to try running the EH-related tests in:

- libc++
- libcxxabi

Eli pointed me to this proprietary C++ conformance test suite: http://www.peren.com/pages/cppvs_set.htm. However, it’s probably not worth the hassle to use a proprietary test suite.

Lowering C++ exception info

Introduction to G++ action lists

In LLVM IR, a function call which may catch and handle a C++ exception is represented using an “invoke”+“landingpad” instruction pair. For example:

```
define void foo() {
        invoke void @external_func() to label %cont unwind label %lpad
cont:
        ; Non-exceptional code path...
lpad:
        %lp_result = landingpad { i8*, i32 } personality … <action list>
        ; Exception-handling code path…
}
```

<action list> is a list of clauses specifying which C++ exception types the landingpad handles. The exception being thrown is tested against each action clause in turn. Each clause is one of the following:

- **catch i8* @ExcType**
  - This means that the landingpad should be entered if the C++ exception being thrown has the type @ExcType (or a subtype of @ExcType). @ExcType is a pointer to the std::type_info (RTTI) for the C++ exception type.
  - This is generated from a “catch” block in the C++ source.

- **filter [i8* @ExcType1, …, i8* @ExcTypeN]**
  - This means that the landingpad should be entered if the C++ exception *doesn’t* match any of the types in the list.
  - Note that this inverts the meaning of “catch”, which is why it takes a list of C++ types rather than a single type.
  - This is used to implement **C++ exception specifications**, such as:
    ```
    void foo() throw(ExcType1, …, ExcTypeN) { bar(); }
    ```
    When Clang compiles foo(), the call to bar() will be generates as an “invoke” with the filter above, unless Clang knows that bar() doesn’t throw.

- **cleanup**
  - This means that the landingpad should always be entered.
  - This is used for calling objects’ destructors.
  - This can only appear at the start of the action list.

Without optimisation, the action lists that Clang generates are fairly simple: There’s at most one “filter” clause, and, if present, it’s at the end of the list, because C++ exception specifications

only apply to whole functions.

With optimisation, functions can be inlined. This causes action lists to be concatenated together, so “catch” and “filter” clauses can be interleaved.

Why do G++ action lists exist?

The reason for encoding C++ exception types in an action list appears to be to reduce the size of the compiler-generated code.

For example, consider a function with an exception spec and a catch block:

```
void foo() throw (Exc1, Exc2, Exc3) {
        try {
                external_func();
        } catch (MyException &exc) {
                my_handler();
        }
}
```

Suppose we didn’t have action lists, and the landingpad code were always entered when handling an exception. The landingpad for the call to external_func() would have to do a number of calls to a __cxa_*() (libsupc++-provided) function to check the exception type:

```
lpad:
        std::type_info *exc_type, void *exception = landingpad;
        if (__cxa_is_subtype(exc_type, &__typeid_for_MyException)) {
                my_handler();
        } else if (!__cxa_is_subtype(exc_type, &__typeid_for_Exc1) &&
                   !__cxa_is_subtype(exc_type, &__typeid_for_Exc2) &&
                   !__cxa_is_subtype(exc_type, &__typeid_for_Exc3) {
                std::unexpected();  // Report C++ exception spec violation
        } else {
                _Unwind_Resume(exception);  // Allow exception to propagate up
        }
```

Instead, using action lists, GCC and Clang generate landingpad code that looks like this:

```
lpad:
        int exc_id, void *exception = landingpad;
        if (exc_id == 1) {  // Check for MyException
                my_handler();
        } else if (exc_id < 0) {  // Check for “filter” mismatch
                std::unexpected();  // Report C++ exception spec violation
```

```
} else {
        _Unwind_Resume(exception);  // Allow exception to propagate up
}
```

The C++ personality function uses the action table to map the exception type to an integer ID, and the landingpad just checks the integer ID. The machine code checking an integer ID is smaller than the machine code for passing the address of a global variable to a function and checking the result.

That said, the compiler-generated exception-handling code is still voluminous. There are opportunities for making this code smaller than GCC and Clang don't seem to take.

Lowering G++ action lists

Normally, LLVM's backend lowers landingpads' action lists by converting them to **.gcc_except_table** data (also known as the **LSDA** -- Language Specific Data Area). This data is then interpreted at runtime by the C++ personality function.

The .gcc_except_table format is described by this blog post by Ian Lance Taylor from January 2011:
http://www.airs.com/blog/archives/464
The format doesn't seem to be formally documented anywhere. I suspect it was invented by GCC (rather than being part of the earlier Itanium ABI).

PNaClLowerInvoke pass will do a similar lowering, converting the LLVM IR's action list into data in the pexe's data segment. I will use a similar format to .gcc_except_table. Since the converted data will be interpreted by user code, the format will not be part of PNaCl's stable ABI.

This format is documented in ExceptionInfoWriter.cpp:

```
// The ExceptionInfoWriter class converts the clauses of a
// "landingpad" instruction into data tables stored in global
// variables.  These tables are interpreted by PNaCl's C++ runtime
// library (either libsupc++ or libcxxabi), which is linked into a
// pexe.
//
// This is similar to the lowering that the LLVM backend does to
// convert landingpad clauses into ".gcc_except_table" sections.  The
// difference is that ExceptionInfoWriter is an IR-to-IR
// transformation that runs on the PNaCl user toolchain side.  The
// format it produces is not part of PNaCl's stable ABI; the PNaCl
// translator and LLVM backend do not know about this format.
//
```

```
// Encoding:
//
// A landingpad instruction contains a list of clauses.
// ExceptionInfoWriter encodes each clause as a 32-bit "clause ID".  A
// clause is one of the following forms:
//
//  1) "catch i8* @ExcType"
//     * This clause means that the landingpad should be entered if
//       the C++ exception being thrown has type @ExcType (or a
//       subtype of @ExcType).  @ExcType is a pointer to the
//       std::type_info object (an RTTI object) for the C++ exception
//       type.
//     * Clang generates this for a "catch" block in the C++ source.
//     * @ExcType is NULL for "catch (...)" (catch-all) blocks.
//     * This is encoded as the integer "type ID" @ExcType, X,
//       such that: __pnacl_eh_type_table[X] == @ExcType, and X >= 0.
//
//  2) "filter [i8* @ExcType1, ..., i8* @ExcTypeN]"
//     * This clause means that the landingpad should be entered if
//       the C++ exception being thrown *doesn't* match any of the
//       types in the list (which are again specified as
//       std::type_info pointers).
//     * Clang uses this to implement C++ exception specifications, e.g.
//         void foo() throw(ExcType1, ..., ExcTypeN) { ... }
//     * This is encoded as the filter ID, X, where X < 0, and
//       &__pnacl_eh_filter_table[-X-1] points to a -1-terminated
//       array of integer "type IDs".
//
//  3) "cleanup"
//     * This means that the landingpad should always be entered.
//     * Clang uses this for calling objects' destructors.
//     * ExceptionInfoWriter encodes this the same as "catch i8* null"
//       (which is a catch-all).
//
// ExceptionInfoWriter generates the following data structures:
//
//   struct action_table_entry {
//     int32_t clause_id;
//     uint32_t next_clause_list_id;
//   };
//
//   // Represents singly linked lists of clauses.
//   extern const struct action_table_entry __pnacl_eh_action_table[];
```

```
//
//   // Allows std::type_infos to be represented using small integer IDs.
//   extern std::type_info *const __pnacl_eh_type_table[];
//
//   // Used to represent type arrays for "filter" clauses.
//   extern const int32_t __pnacl_eh_filter_table[];
//
// A "clause list ID" is either:
//  * 0, representing the empty list; or
//  * an index into __pnacl_eh_action_table[] with 1 added, which
//    specifies a node in the clause list.
//
// Example:
//
//   std::type_info *const __pnacl_eh_type_table[] = {
//     // defines type ID 0 == ExcA and clause ID 0 == "catch ExcA"
//     &typeinfo(ExcA),
//     // defines type ID 1 == ExcB and clause ID 1 == "catch ExcB"
//     &typeinfo(ExcB),
//     // defines type ID 2 == ExcC and clause ID 2 == "catch ExcC"
//     &typeinfo(ExcC),
//   };
//
//   const int32_t __pnacl_eh_filter_table[] = {
//     0,   // refers to ExcA;  defines clause ID -1 as "filter [ExcA, ExcB]"
//     1,   // refers to ExcB;  defines clause ID -2 as "filter [ExcB]"
//     -1,  // list terminator; defines clause ID -3 as "filter []"
//     2,   // refers to ExcC;  defines clause ID -4 as "filter [ExcC]"
//     -1,  // list terminator; defines clause ID -5 as "filter []"
//   };
//
//   const struct action_table_entry __pnacl_eh_action_table[] = {
//     // defines clause list ID 1:
//     {
//       -4,  // "filter [ExcC]"
//       0,  // end of list (no more actions)
//     },
//     // defines clause list ID 2:
//     {
//       -1,  // "filter [ExcA, ExcB]"
//       1,  // else go to clause list ID 1
//     },
//     // defines clause list ID 3:
```

```
//     {
//       1,  // "catch ExcB"
//       2,  // else go to clause list ID 2
//     },
//     // defines clause list ID 4:
//     {
//       0,  // "catch ExcA"
//       3,  // else go to clause list ID 3
//     },
//   };
//
// So if a landingpad contains the clause list:
//   [catch ExcA,
//    catch ExcB,
//    filter [ExcA, ExcB],
//    filter [ExcC]]
// then this can be represented as clause list ID 4 using the tables above.
//
// The C++ runtime library checks the clauses in order to decide
// whether to enter the landingpad.  If a clause matches, the
// landingpad BasicBlock is passed the clause ID.  The landingpad code
// can use the clause ID to decide which C++ catch() block (if any) to
// execute.
//
// The purpose of these exception tables is to keep code sizes
// relatively small.  The landingpad code only needs to check a small
// integer clause ID, rather than having to call a function to check
// whether the C++ exception matches a type.
//
// ExceptionInfoWriter's encoding corresponds loosely to the format of
// GCC's .gcc_except_table sections.  One difference is that
// ExceptionInfoWriter writes fixed-width 32-bit integers, whereas
// .gcc_except_table uses variable-length LEB128 encodings.  We could
// switch to LEB128 to save space in the future.
```

Modifying libsupc++

If we want to support SJLJ EH alongside zero-cost EH, there are two ways we could deal with this in libsupc++:

1. Build libsupc++ twice. Select the PNaCl-SJLJ code path using an #ifdef.

2. Have a single build of libsupc++ which includes both the PNaCl-SJLJ EH code and

zero-cost EH code, but allow selecting the former at link time using some symbol redefinitions.
    - For example: libsupc++ defines _pnacl_sjlj_Unwind_RaiseException, and we pass “--defsym _Unwind_RaiseException=_pnacl_sjlj_Unwind_RaiseException”.

I am inclined to do (2). (1) seems unnecessarily heavyweight when there are only a couple of functions that need to be different in the SJLJ case.

Question: Enable EH by default?

Should the link-time default be “--exception-handling=none” or “--exception-handling=sjlj”?

Enabling EH carries a pexe size overhead and a runtime overhead. The runtime overhead will go away when we stabilise zero-cost EH, but the pexe size overhead will remain, so it might be worth disabling EH by default in the long term.

Maybe we could provide an “--exception-handling=auto” option which enables EH only if exceptions are used?

- Could this enable EH only if “throw” is used? We’d decide this based on whether __cxa_throw() is called.
    - This wouldn’t change program behaviour. A program that never throws definitely doesn’t need any landingpads.
    - This probably wouldn’t be very useful, because C++’s “new” operator throws, and most C++ programs use “new”.
- Could this enable EH only if “catch” is used? We’d decide this based on whether any landingpads contain a “catch” clause.
    - We would exclude “cleanup” clauses. Clang generates these for running destructors, which would be the main source of bloat when enabling EH.
    - Whether this is useful depends on whether libstdc++ contains any “catch” blocks. A pexe might contain “catch” blocks but not, in practice, rely on recovering from any exceptions at run time.
    - We might have to exclude “catch (...)” (catch-all) blocks from the consideration, if these tend to be used for cleanup only.
    - This would change program behaviour, because destructors wouldn’t get run if an uncaught exception occurs.

Conclusion: “--eh=none” will be the default initially, since that is the status quo. We might revisit this. If “--eh=auto” turns out to be workable, we might make that the default, but this requires further investigation.

Possible optimisations

Reducing the overhead of setjmp()

PNaCl's setjmp() is relatively expensive, because:

- it is implemented as a function call;
- the setjmp() function saves all callee-saved registers.

We could improve this by having the LLVM backend inline setjmp() and save only the registers that are live.

PNaCl's current ABI also burns 1024 bytes for each jmp_buf, which is somewhat excessive. In principle, we could reduce the jmp_buf size to as little as 4 bytes. We could have each llvm.nacl.setjmp() call do an implicit alloca, stash the data there, and save the address of the alloca into the jmp_buf.

Such optimisations are probably not worthwhile if SJLJ EH is only a stop gap and we want to implement zero-cost EH. They might be worthwhile if we really wanted to minimise ABI surface area and only support SJLJ EH.

Pexe size: Reducing the size of landingpad code

Suppose we have a landingpad block that handles one exception type and runs no destructors:

```
void foo() {
        try {
                external_func();
        } catch (MyException &) {
                caught = 123;
        }
}
```

GCC and Clang+LLVM both generate the following landingpad code from that (even with optimisation enabled):

```
        movq    %rax, %rdi
        cmpq    $1, %rdx   // Check exception type ID matches MyException
        je      .L4
        call    _Unwind_Resume
.L4:
        // Code above here could be omitted
        call    __cxa_begin_catch
        movl    $123, caught(%rip)
        call    __cxa_end_catch
```

The check for the exception type ID should be superfluous, because the personality function

should never invoke this landingpad if the exception doesn't match.

I don't know why GCC and LLVM don't optimise this. In principle, PNaCl could prune out the check and the _Unwind_Resume call. Whether this is worthwhile implementing depends on how much this reduces pexe size in typical EH-using programs, and whether EH gets used much.

This optimisation would apply to zero-cost EH as well as SJLJ EH.

Appendix A: PNaCl's current zero-cost EH implementation

The PNaCl linker currently accepts the option "--pnacl-allow-exceptions", which produces a non-ABI-stable pexe that uses zero-cost EH.

The resulting pexe interacts with PNaCl system code in the following ways:

- The pexe uses the "invoke"+"landingpad" and "resume" instructions.
- The pexe calls system-provided _Unwind_RaiseException() and other _Unwind_*() functions.
- The pexe's "landingpad" instruction contains two things:
    - A reference to a pexe-provided "**personality function**". This is typically __gxx_personality_v0(). This gets called by system code, specifically _Unwind_RaiseException().
    - An G++ **action list** (as described above): a list of "catch", "filter" and "cleanup" actions which may refer to C++ exception types. The LLVM backend converts this to a .gcc_except_table section, also known as the LSDA.

  When system code calls the personality function, it passes it a pointer to the system-generated LSDA. This creates an unfortunate interdependency between system code and user code, because user code is expected to read and interpret a system-generated data structure. (This problem is tracked as https://code.google.com/p/nativeclient/issues/detail?id=3118.)
- The pexe must define the external symbols malloc(), free(), and some others, which are called by system code. (This problem was originally tracked as https://code.google.com/p/nativeclient/issues/detail?id=3069.)

Since this is a complicated tangle of interactions between user code and system code, we don't want to expose it as a stable ABI.

Appendix B: Why not use LLVM's existing LowerInvoke pass?

LLVM's LowerInvoke pass has two modes:

- **"Remove EH" mode.** This converts "invokes" to "calls". "landingpad" blocks get

removed. PNaCl currently uses this for producing ABI-stable pexes.

- **SJLJ mode.** When passed "--enable-correct-eh-support" (a.k.a. “ExpensiveEHSupport”), implement “invoke” using SJLJ.

LowerInvoke’s SJLJ mode isn’t suitable for PNaCl because:

- It doesn’t expand out the “landingpad” instruction, only “invoke”. It appears to rely on the LLVM backend to generate .gcc_except_table sections.
- It makes dubious assumptions about the behaviour of setjmp() and longjmp() that probably aren’t portable. Rather than inserting one setjmp() per “invoke”, it inserts a single setjmp() at the start of a function. Since longjmp()’ing to that setjmp() is liable to lose the values of local variables, LowerInvoke modifies each “invoke” so that it first spills all live local variables to the stack.
    - Doing a single setjmp() per function might just move the costs around. It would be faster for an “invoke” in a loop, but it would be slower for a function like this:

```
void foo() {
        if (x)
                return;  // Common case
        try {
                …
        } catch (...) {
                …
        }
}
```

Also, it’s not clear to me whether anyone uses LowerInvoke’s SJLJ mode, or whether it actually works. Its coverage by llvm-lit tests is very limited.

Appendix C: Example LLVM IR for control flow expansion

Clang compiles catcher_func() from the earlier example to the following LLVM IR:

```
@_ZTS11MyException = linkonce_odr constant [14 x i8] c"11MyException\00"
@_ZTI11MyException = linkonce_odr unnamed_addr constant { i8*, i8* } { i8*
bitcast (i8** getelementptr inbounds (i8**
@_ZTVN10__cxxabiv117__class_type_infoE, i32 2) to i8*), i8* getelementptr
inbounds ([14 x i8]* @_ZTS11MyException, i32 0, i32 0) }

define i32 @_Z12catcher_funcv() {
entry:
  %call = invoke i32 @_Z13external_funcv()
          to label %invoke.cont unwind label %lpad

invoke.cont:
  %add = add nsw i32 %call, 100
```

```
  br label %return

lpad:
  %0 = landingpad { i8*, i32 }
          personality i8* bitcast (i32 (...)* @__gxx_personality_v0 to i8*)
          catch i8* bitcast ({ i8*, i8* }* @_ZTI11MyException to i8*)
  %1 = extractvalue { i8*, i32 } %0, 1
  %2 = tail call i32 @llvm.eh.typeid.for(
          i8* bitcast ({ i8*, i8* }* @_ZTI11MyException to i8*)) #2
  %matches = icmp eq i32 %1, %2
  br i1 %matches, label %catch, label %eh.resume

catch:
  %3 = extractvalue { i8*, i32 } %0, 0
  %4 = tail call i8* @__cxa_begin_catch(i8* %3) #2
  %value = bitcast i8* %4 to i32*
  %5 = load i32* %value, align 4, !tbaa !0
  %add1 = add nsw i32 %5, 200
  tail call void @__cxa_end_catch()
  br label %return

return:
  %retval.0 = phi i32 [ %add, %invoke.cont ], [ %add1, %catch ]
  ret i32 %retval.0

eh.resume:
  resume { i8*, i32 } %0
}
```

PNaClLowerInvoke expands this out to the following LLVM IR:

```
%ExceptionFrame = type { [1024 x i8], %ExceptionFrame*, i32 }

@__pnacl_eh_stack = external thread_local global i8*

define i32 @_Z12catcher_funcv() {
entry:
  %invoke_frame = alloca %ExceptionFrame, align 8
  %exc_info_ptr = getelementptr %ExceptionFrame* %invoke_frame, i32 0, i32 2
  %invoke_next = getelementptr %ExceptionFrame* %invoke_frame, i32 0, i32 1
  %invoke_jmp_buf = getelementptr %ExceptionFrame* %invoke_frame, i32 0, i32 0,
i32 0
  %pnacl_eh_stack = bitcast i8** @__pnacl_eh_stack to %ExceptionFrame**
  %invoke_sj = call i32 @llvm.nacl.setjmp(i8* %invoke_jmp_buf)
  %invoke_sj_is_zero = icmp eq i32 %invoke_sj, 0
  br i1 %invoke_sj_is_zero, label %invoke_do_call, label %lpad
```

```
invoke_do_call:
  %old_eh_stack = load %ExceptionFrame** %pnacl_eh_stack
  store %ExceptionFrame* %old_eh_stack, %ExceptionFrame** %invoke_next
  store i32 1, i32* %exc_info_ptr
  store %ExceptionFrame* %invoke_frame, %ExceptionFrame** %pnacl_eh_stack
  %call = call i32 @_Z13external_funcv()
  store %ExceptionFrame* %old_eh_stack, %ExceptionFrame** %pnacl_eh_stack
  br label %invoke.cont

invoke.cont:
  %add = add nsw i32 %call, 100
  br label %return

lpad:
  %landingpad_ptr = bitcast i8* %invoke_jmp_buf to { i8*, i32 }*
  %0 = load { i8*, i32 }* %landingpad_ptr
  %1 = extractvalue { i8*, i32 } %0, 1
  %matches = icmp eq i32 %1, 0
  br i1 %matches, label %catch, label %eh.resume

catch:
  %2 = extractvalue { i8*, i32 } %0, 0
  %3 = tail call i8* @__cxa_begin_catch(i8* %2) #2
  %value = bitcast i8* %3 to i32*
  %4 = load i32* %value, align 4, !tbaa !0
  %add1 = add nsw i32 %4, 200
  tail call void @__cxa_end_catch()
  br label %return

return:
  %retval.0 = phi i32 [ %add, %invoke.cont ], [ %add1, %catch ]
  ret i32 %retval.0

eh.resume:
  %resume_exc = extractvalue { i8*, i32 } %0, 0
  %resume_cast = bitcast i8* %resume_exc to i32*
  call void @__pnacl_eh_resume(i32* %resume_cast)
  unreachable
}
```