# Exception handling in LLVM, from Itanium to MSVC

Reid Kleckner
David Majnemer

# Agenda

- Exception handling: what it is, where it came from
- Introduction to the landingpad model used in LLVM and GCC
  - Elegant simplicity of the landingpad model
- Introduction to the MSVC model
  - Problematic requirements of the MSVC model
- Introduction to the new LLVM IR model
  - Compromise between block scoping and free-form control flow

# What is exception handling?

- Provides non-local control flow transfers to suspended frames
- Returns alternative data not described by function return types
- Non-local exits considered important as library layering accumulated
- Bjarne et al design C++ exceptions from 1984-1989
- “Exception handling for C++” is published by Bjarne and Andrew Koenig in 1989

# How is exception handling implemented?

- Bjarne and Koenig outlined two implementation strategies in 1989
- Portable exception handling:
  - Built on linked lists and setjmp/longjmp
  - Ideal for C transliteration (CFront)
  - Interoperates across EH-unaware code produced by other vendors
- Efficient exception handling:
  - Built on PC lookup tables that determine which EH actions to take
  - Requires reliable stack unwinding mechanism
  - Need call frame information (CFI) to restore non-volatile registers and locate return addresses
- Different vendors made different choices

# Borland implements C++ and SEH in 1993

- Implementation approach similar to “portable” EH described in ‘89
- Windows toolchain ecosystem was diverse, needed interoperability
- SEH allowed recovering from CPU traps (integer divide by zero, etc)
- SEH **also** allowed resuming in the trapping context
  - Usable for virtual memory tricks or making divide by zero produce a value
- Microsoft adopted SEH for Windows, fs:00 becomes TLS slot for EH

# HP landingpad model for Itanium

- HP had years of experience getting C++ EH right in multiple compilers
  - Major user of CFront, eventually transitioned to aC++
- HP popularized the landingpad model through the Itanium C++ ABI
- Uses “successive unwinding”: restores the register context of each frame on the stack with cleanups until the right catch is reached
  - Major departure from ‘89 models, which both pinned objects with destructors in memory
- Language-specific data area (LSDA) contains two tables:
  - Call site table: map from PC range to landingpad label plus action table index
  - Action table: array of type information references and next action chains
  - At most one landingpad label per call
- GCC adopted the Itanium C++ ABI, LLVM followed later

# LLVM IR for landingpads

- Invokes are calls with an unwind edge
- %ehvals represent an alternate return value in EAX:EDX on x86
- Landingpad must be first non-phi instruction in basic block
- Catch handler dispatch uses compare and branch on selector

```
define void @f()
      personality i32 (...)* @__gxx_personality_v0 {
  ...
  invoke void @maythrow()
      to label %normal unwind label %lpad
normal:
  ...
lpad:
  %ehvals = landingpad { i8*, i32 }
      catch i8* null
  ...
}
```

# Landingpad selector dispatch example

```
int main () {
  try {
    maythrow();
  } catch (A) {
    puts("A");
  } catch (B) {
    puts("B");
  }
}
```

```
define i32 @main() … {
entry:
  invoke void @maythrow()
          to label %try.cont unwind label %lpad
try.cont:
  ret i32 0
lpad:
  %0 = landingpad { i8*, i32 }
          catch { i8*, i8* }* @_ZTI1A
          catch { i8*, i8* }* @_ZTI1B
  %1 = extractvalue { i8*, i32 } %0, 0
  %2 = extractvalue { i8*, i32 } %0, 1
  %3 = tail call i32
      @llvm.eh.typeid.for(...@_ZTI1A...)
  %isA = icmp eq i32 %2, %3
  br i1 %isA, label %catch.A,
        label %catch.fallthrough
```

```
catch.fallthrough:
  %5 = tail call i32
     @llvm.eh.typeid.for(...@_ZTI1B...)
  %isB = icmp eq i32 %2, %5
  br i1 %isB, label %catch.B,
        label %eh.resume

catch.A:
  ...
catch.B:
  …

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

# Advantages of LLVM’s landingpad model

- Basic blocks are single-entry single-exit, simplifying dataflow and SSA formation
- Keeps control flow graph for EH dispatch in code (conditional branches)
    - SimplifyCFG can and does tail merge similar catch handlers
    - No unsplittable blocks, easier to find insertion points
- Invokes inlined by chaining “ret” to normal label and “resume” to unwind label
- Only one special control transfer: unwind edge from invoke
- Unfortunately, Windows EH does not use landingpads

# Windows exception handling model

- Tables map from program state number to “funclet” pointers
- State number tracked through PC tables and explicitly in memory
- Each funclet shares the parent frame via EBP/RBP
    - Runtime provides the “establishing frame pointer” via regparm
    - Funclet assumes SP has dynamically changed, similar to dynamic alloca
- Funclets implement three major actions:
    - SEH filter: Should this exception be caught, retried, or propagated outwards
    - Cleanup: Cleanup code, like C++ destructor calls or finally blocks
    - Catch: User code from the catch block body

# Windows exception handling phases

1. Exception is raised to OS
2. Walk stack, call each personality until the exception is claimed
    - The SEH and CLR personalities call active filter funclets during this phase
3. Call each personality again to run cleanups
    - Personality controls what happens if cleanups raise an exception
4. Personality of catching frame handles the exception
    - C++ personality calls catch funclet, uses SEH to detect C++ rethrow
5. Personality resets register context to the parent frame

# Windows exception handling implications

- Contrast to successive unwinding: Only one register context reset
- All EH occurs with the exceptional frame on the stack!
  - The C++ exception object lives in the frame of the throw
  - Stack pointer is reset at the closing curly of the catch block
- Successively unwinding to landingpads cannot be compatible with MSVC EH
  - Mingw will never have MSVC-compatible exception handling
- Chose to use MSVC personality rather than invent new split-frame personality

# Possible strategy: frontend outlines funclets

- Frontend outlining would satisfy the personality routine
- Good separation of concerns, keep C++ knowledge in Clang
- Creates **massive** optimization barrier
  - Local optimization problems become much harder interprocedural problems
  - No ability to reason about escaped local variables used in funclets
- Personality provides frame pointer, would need to teach backend how to reason about the layout of another function's frame
  - Lambdas and blocks are easy because we control the call site
  - Parent function cannot be inlined, doing so would perturb the frame
- Ultimately decided to outline SEH filters in the frontend
  - Difficult to optimize, impossible to reason about control flow
- Let's try backend outlining with landingpads...

# Pattern match away landingpads

- Attempted to use landingpads and a pile of intrinsics, outline catches and cleanups into new functions during WinEHPrepare
- Funclet bounds were inferred from intrinsic calls (@llvm.eh.begincatch, etc)
- SSA values live across funclet bounds were demoted (similar to SJLJ EH)
  - Shared demoted stack allocations with @llvm.localescape / @llvm.localrecover
- Pattern matched selector comparisons to recover dispatch logic data

# Landingpads, MSVC-style

```
throw:
  invoke void @foo() … unwind label %lp

lp:
  %sel = landingpad i32 catch %rtti* @A.type, catch %rtti* @B.type
  %forA = call i32 @llvm.eh.typeid.for(%rtti* @A.type)
  %isA = icmp eq i32 %sel, %forA
  br i1 %isA, label %catch.A, label %catch.fallthrough

catch.fallthrough:
  %forB = call i32 @llvm.eh.typeid.for(%rtti* @B.type)
  %isB = icmp eq i32 %sel, %forB
  br i1 %isA, label %catch.B, label %eh.resume
```

# Landingpads, MSVC-style

```
throw:
  invoke void @foo() … unwind label %lp

lp:
  %sel = landingpad i32 catch %rtti* @A.type, catch %rtti* @B.type
  %forA = call i32 @llvm.eh.typeid.for(%rtti* @A.type)
  %isA = icmp eq i32 %sel, %forA
  br i1 %isA, label %catch.A, label %catch.fallthrough

catch.fallthrough:
  %forB = call i32 @llvm.eh.typeid.for(%rtti* @B.type)
  %isB = icmp eq i32 %sel, %forB
  br i1 %isA, label %catch.B, label %eh.resume
```

# Landingpads, MSVC-style: hard mode

```
throw:
  invoke void @foo() … unwind label %lp

lp:
  %sel = landingpad i32 catch %rtti* @A.type, catch %rtti* @B.type
  %forA = call i32 @llvm.eh.typeid.for(%rtti* @A.type)
  %forB = call i32 @llvm.eh.typeid.for(%rtti* @B.type)
  %isA = icmp eq i32 %sel, %forA
  %isB = icmp eq i32 %sel, %forB
  %isAorB = or i1 %isA, %isB
  br i1 %isAorB, label %catch.AorB, label %eh.resume
```

# Landingpads, MSVC-style: hard mode

```
throw:
  invoke void @foo() … unwind label %lp

lp:
  %sel = landingpad i32 catch %rtti* @A.type, catch %rtti* @B.type
  %forA = call i32 @llvm.eh.typeid.for(%rtti* @A.type)
  %forB = call i32 @llvm.eh.typeid.for(%rtti* @B.type)
  %isA = icmp eq i32 %sel, %forA
  %isB = icmp eq i32 %sel, %forB
  %isAorB = or i1 %isA, %isB
  br i1 %isAorB, label %catch.AorB, label %eh.resume
```

# Lesson

**Turning apple sauce back into apples does not work!**

# Other lessons learned

- Discovered lexical scoping requirements in tables
  - Previously believed we could produce denormalized tables: try ranges around every invoke
- LLVM IR does not have scope information! It is a graph
  - Lack of nesting information ensured our demise

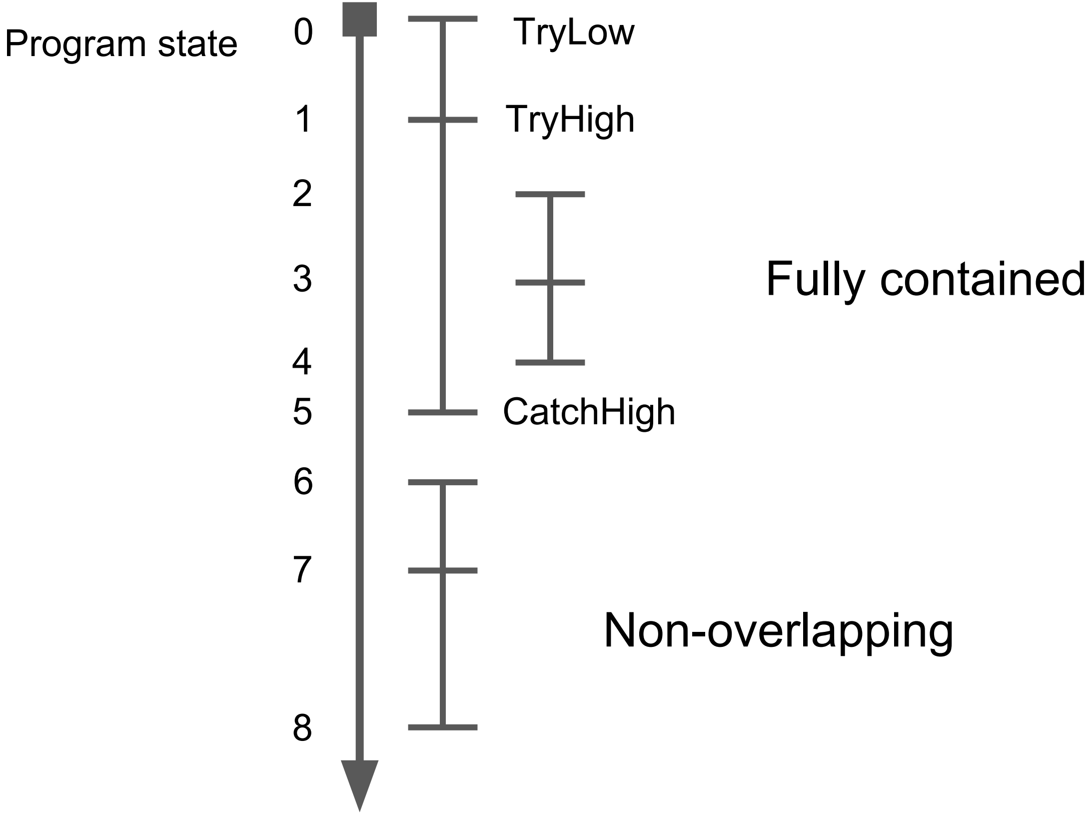
# C++ personality scoping impositions

- The compiler is required to emit code+tables which are lexically nested
  - Tables + runtime must agree on current state of the program
- TryBlockMap is an array of: tuple of states (TryLow, TryHigh, CatchHigh) + array of catch handlers
  - Intervals must be non-overlapping or contained within another interval
  - Catch handlers must have distinct addresses, no reuse permitted
- Forces the compiler's output to resemble valid C++ source code
  - Doesn't necessarily need to have the **same** scopes as the source program

# TryBlockMap state numbering constraints

```
try {
  f(0);
} catch (...) {
  try {
    f(1);
  } catch (...) {
    f(2);
  }
}

try {
  f(3);
} catch (...) {
  f(4);
}
```



# MSVC-style EH, take two

- New family of “pad” instructions representing funclet starts
  - **catchpad**, **cleanuppad**
- New family of terminator instructions representing funclet returns
  - **catchret**, **cleanupret**
- New family of instructions to inform LLVM of lexical nesting
  - **catchendpad**, **cleanupendpad**
- And last, but not least, a new type: token

# MSVC-style EH, take two

- SSA values with **token** type cannot be obscured
  - Cannot be PHI’d, cannot be stored/loaded to memory, cannot be in a select, etc.
  - Makes it possible to associate **catchpad** with **catchret**, **cleanuppad** with **cleanupret**
- Unwind edges inform us of lexical scopes
  - Instructions which unwind to **catchendpad** are “exiting” a catch handler
  - Instructions which unwind to **cleanupendpad** are “exiting” a cleanup

# New EH: Catches

```
int main () {
  try {
    maythrow();
  } catch (A) {
    handleA();
  } catch (B) {
    handleB();
  }
}
```

# Throwing the Exception

```
int main () {
  try {
    maythrow();
  } catch (A) {
    handleA();
  } catch (B) {
    handleB();
  }
}
```

```
…
invoke void @maythrow()
        to label %try.cont
        unwind label %dispatch.a
...
```

# Catching the Exception

```
int main () {
  try {
    maythrow();
  } catch (A) {
    handleA();
  } catch (B) {
    handleB();
  }
}
```

```
dispatch.a:
  %cpA = catchpad [%rtti.A* @A.type]
        to label %handle.a
        unwind label %dispatch.b

handle.a:
  invoke void @handleA()
        to label %catchret.A
        unwind label %catchend

catchret.a:
  catchret %cpA to label %exit
```

# Catching the Exception

```
int main () {
  try {
    maythrow();
  } catch (A) {
    handleA();
  } catch (B) {
    handleB();
  }
}
```

```
dispatch.b:
  %cpB = catchpad [%rtti.B* @B.type]
        to label %handle.b
        unwind label %catchend

handle.b:
  invoke void @handleB()
        to label %catchret.B
        unwind label %catchend

catchret.b:
  catchret %cpB to label %exit
```

# Catching the Exception: catchendpad

```
dispatch.a:
  %cpA = catchpad [...] to label %handle.a unwind label %handle.b
handle.a:
  invoke void @handleA() to ... unwind label %catchend
dispatch.b:
  %cpB = catchpad [%rtti.B* @B.type] to label %handle.b unwind label %catchend
handle.b:
  invoke void @handleB() to ... unwind label %catchend
catchend:
  catchendpad unwind to caller
```

# Result: it “just” works

- For the most part, the new IR survives LLVM’s optimizers
- New IR dramatically simplified WinEHPrepare
  - Removed ~2500 lines of broken code, currently only ~1200 lines of working code
- SimplifyCFG still merges blocks in two funclets ending in unreachable
  - WinEHPrepare has to undo this
- WinEHPrepare still demotes SSA values live across funclet boundaries
  - No pattern matching necessary
  - Register allocator would do better spill placement

# Future work

- Inlining into cleanups currently disabled
  - Need to associate call sites with parent funclet
  - Use operand bundles? Outline in WinEHPrepare?
- Funclet parent relationship is implicit
  - Relationship is discovered via unwind edges
  - Experiment with explicit parents?
- Push funclet spill insertion down into register allocator
- Make catchpad a switch? Make it splittable?

# Conclusion

- Clang now has MSVC compatible exception handling
- Clang has partial support for SEH, does not model non-call exceptions
  - Need a way to model edges from potentially trapping instructions
- New EH representation preserves core LLVM invariants (SSA!)
  - Relatively few changes required to most passes
- Work ongoing to simplify new representation