Exception Handling

Interfaces, Implementations, and Evaluation

David Bindel          E. Jason Riedy

U.C. Berkeley
What do we want?

We want to produce programs which can

• detect exceptional conditions and
• *react to them*.

We also want these programs to be

• supported by our friendly neighborhood programming environments and
• amenable to optimization on current and future platforms.
• The standard specifies *interface requirements*
• Interfaces have multiple good implementations
• Good design is hard, but interface criteria include
  • Minimality
  • Orthogonality
  • Clarity
• We prefer explicit and local control and data flow
Motivating and rejected examples
  • Deferring debugging...

Survey of software interfaces

Hardware support

Hardware / software mapping

A scarecrow proposal
Motivating / Archetypal Examples

- Algorithms that exceptions make risky
  - abort, wasting minimal work (Eureka exit)
  - then possibly do something else (complex multiply, scaling)

- Slightly change the arithmetic
  - substitute a limit for an exceptional result (continued fractions, replacement)

- Soften the arithmetic’s boundaries
  - extend the dynamic range (long products, counting mode)

- Communicate the quality of a result
Rejected Examples

Any of the following could overly constrain our choices:

- Supporting heavy modifications to the arithmetic.
  - UN, OV, etc.
- Allowing extremely non-local, implicit control and data flow.
- Considering any particular hardware implementation.
- Requiring specific debugging tools...
Deferring Debugging

• General purpose tools handle uninitialized data.
  • Purify, valgrind, etc.

• Different applications need different retrospective diagnostic facilities.

• We’re not sure how to support future debugging tools. (path-based, etc.)

We need to keep debugging in mind, but it is a “quality of implementation” issue.
Survey of Software Interfaces

- Motivating and rejected examples
- Survey of software interfaces
  - Try-catch
  - Flag testing
  - Explicit trapping
  - Substitution
  - Flag-carrying types
  - Conditional branching FP ops
- Hardware support
- Hardware / software mapping
- A scarecrow proposal
try {
    fast and sloppy code
}
catch (exceptional cases) {
    slow and careful
}

Floating-point mechanism exists in:
- fpmenu
- Ada
- Numerical Turing
- BASIC
- Common Lisp (optional)
- Borneo (specification)
Try-Catch

Language aspects:

- Scope is always specified as linguistic blocks.
- Extent:
  - Can called functions also raise exceptions?
  - Are “thrown” exceptions specified statically?
- How do callers / callees communicate which exceptions are interesting?
- Is the try block interrupted precisely?
- Can execution be resumed or statements restarted?
Try-Catch

Benefits:
- Matches existing, non-FP practice.
- Limits optimization impact to blocks.

Drawbacks:
- Existing practice is often mis-managed.
Observations:

- Case without resumption / restart can be implemented through either traps or flags.
- Catching invalid is often followed by testing in-scope variables to determine *which* invalid op occurred.
double f (double x) {
    save environment
    do work;
    if (flags raised) do alternate work;
    restore environment
    merge proper flags
    return result
}

Exists in:
- C99
- Many platform-dependent libraries
- Borneo (specification)
double f (double x) {
    fenv_t fenv;
    feholdexcept(&fenv);
    do work;
    if (flags raised) do alternate work;
    fesetenv(&fenv);
    return out1+out2;
}

Exists in:

• C99
• Many platform-dependent libraries
• Borneo (specification)
Flag-Testing

Language aspects:
- **Scope**: Are flags set by block, or though a global datum?
- **Extent**: How do flags pass through subroutines?

Benefits:
- Predictable control flow.

Drawbacks:
- All operations share state.
- Subexp movement and compile-time evaluation often incorrect.
- Flag tests clutter code.
double f (double x) {
    fenv_t fenv; feholdexcept(&fenv);
    do work;
    if (flags raised) do alternate work;
    fesetenv(&fenv);
    return out1+out2;
}

Observations:

- Almost all uses follow the above pattern, including a few operations to set output flags implicitly.
- Compilers must virtualize and track flags for optimization.
Explicit Trapping

Implementations
- Sun’s libm9x
- SIGFPE handling (wmexcp, fpmenu)

Aspects
- Scope: dynamic
- Extent: dynamic

Benefits
- Unknown

Drawbacks
- No portable interfaces
- Nigh-impossible to use
- Serious non-local, implicit effects
FPE_PRESUB(FE_INVALID,+INFINITY)
   for (i = 0; i < n_items; ++i)
       newprice[i] = price[i] + bidincr[i];
FPE_END_PRESUB

Exists in:
   • IEEE defaults
   • fpmenu: presub and counting
FPE_COUNT(&cnt)
    for (i = 0; i < N; ++i)
        out *= A[i] + B[i];
FPE_END_COUNT

Exists in:
    - IEEE defaults
    - fpmenu: presub and counting
Language aspects:
- Static scope, but static or dynamic extent
- How do you determine the replaced type?
- Do you consider operands? Get the sign?
- Location of count or other implicit operands?

Benefits:
- Well-defined, can have very limited scope
- Many implementation / optimization options

Drawbacks:
- Only two functionalities out of how many?
In the continued fraction code:

```c
double f, f1, ...;
flagdouble r;
int j;
...
    r = d1/d;
    f1 = -r * d;
    if (!flagtest_and_clear(r, INVALID))
        continue;
// fixup
...
```

- Explicit syntax for the desired result.
- Useful when only a few items are flagged.
Flag-Carrying Types

Language aspects:

- Scope and extent match value types’.
- Static typing = static flags
- Relies on expression evaluation typing
Flag-Carrying Types

Benefits:

- Everything is explicit.
- Optimizations use existing frameworks.
- User control over which expressions require flags.
- Programmers understand data-flow.

Drawbacks:

- Verbose (sub- and dynamic typing help)

Observations:

- Flagged compile-time constants keep flags.
- Subexpressions can be lifted.
complex operator* (complex x, complex y) {
    let
        double operator*(double, double) =
            trapping_mult(double, OVERFLOW: ov_label,
                           UNDERFLOW: un_label,
                           INVALID: not_complex_label);
        double operator+(double, double) =
            trapping_add(double, INVALID: infs_label);
        double operator-(double, double) =
            trapping_sub(double, INVALID: infs_label);
    in {
        return complex (real(x)*real(y) - imag(x)*imag(y),
                        real(x)*imag(y) - imag(x)*real(y));
    }
    ov_label:
        ...
}
Hardware Support

- Motivating and rejected examples
- Survey of software interfaces
- **Hardware support**
  - Existing hardware: flags
  - Existing hardware: traps
  - Flags versus traps
- Hardware / software mapping
- A scarecrow proposal
Existing HW: Flags

- Basic operations:
  - Save registers
  - Restore registers
  - Test flags

- One or more registers visible in ISA
  - May include “last instruction” flags

- May be additional internal storage
  - e.g. with reorder buffer entry
Existing HW: Traps

- Basic operations:
  - Enable trapping
  - Disable trapping
  - Set handler
- Currently require OS support
  - Need privileged mode to set handler
  - Handler runs in privileged mode
- Trap enable/disable on IA32 costs more than flag save/restore
Flags versus Traps

- Traps are an optimization for flag test and branch
  - But flag tests are reasonably inexpensive!
  - Flag tests need only occur at synchronization points (identified by programmer or compiler)

- There are other possible optimizations:
  - Execution predicated on flag settings
  - Conditional branch FP ops
  - And others...

- Compiler could optimize away explicit tests
HW/SW Mapping

- Motivating and rejected examples
- Survey of software interfaces
- Hardware support
- Hardware / software mapping
  - Extended range: a case study
  - fpmenu implementation notes
  - Interfaces and implementations
  - Performance
- A scarecrow proposal
scaled_double prod;
for (i = 0; i < n; ++i)
    prod *= a[i];

Extend range by implementing a scaled precision:
  • No exceptions: scale on every operation
  • Flags: test after each operation
  • Traps: use “counting mode”

Can optimize first two cases by blocking.
scaled_double prod;
for (i = 0; i < n; i += BLOCK) {
    prod_tmp = fast product over block
    if (no range exception)
        prod *= prod_tmp;
    else
        prod *= scaled subproduct
}

- Compiler ideally generates this from previous code
- Otherwise, little worse than blocking matrix codes
  - Could probably use similar automatic tuning
fpmenu:

- Uses SIGFPE handler + ugly C macros to implement try/catch and replacement
- On exception
  - try/catch: restore state, jump to user
  - substitution: decode, compute, writeback
  - other: re-execute instruction
fpmenu:

- handler choice really needs compiler input
- must manually add fwait instructions
- makes most optimizations dangerous
- context save penalty on try/catch entry
- instruction re-execution and toggling traps are both expensive

Compiler support would help, but some problems are intrinsic to trap-based handling.
Interfaces and Implementations

Several implementations for software interfaces
- Compile flag test and branch to trapping code
- Implement try-catch handling with flag tests

Software resources need not map directly to HW
- Map HW invalid flag to multiple software flags
- Support flag-carrying types with virtualized flags
- Merge local HW flags into virtual global register

We standardize interface requirements, not implementations. Simple basic interfaces are easier to reason about and permit adequate room to optimize.
Performance: Traps v. Flags

- Platforms tested: PPro@233MHz, P3@800MHz, and P4@1.4GHz
- Tested continued fractions, long products.
- Results: Blocked flag tests are fine.
  - Blocked flag tests usually faster than trapping.
  - Immediate flag tests are considerably slower.
    - Detrimental in tight loops like long product.
    - Same as trapping in continued fractions.
  - Saving machine state for try-catch is slow.
Scarecrow Proposal

- Motivating and rejected examples
- Survey of software interfaces
- Hardware support
- Hardware / software mapping
- A scarecrow proposal
A scarecrow is a spooky outline of a straw-man. **SHALL** be able to tie flags to value types.

- Compilers already need this for optimizations.
- Can use this to deliver information on exceptions affecting results.
- Generalizes to vectors in many ways.
Helpers:

**SHALL** provide a presubstitution mechanism
- Presubstitute the result of an *expression*.
- Any unhandled conditions remain raised.
- Types are known.

**SHALL** provide scaled-exponent type

- Many uses, can be heavily optimized.
- Must explicitly determine the substitute’s sign.
- Note that doubled-precisions don’t round correctly.
**Scarecrow Proposal**

**SHALL** require programmers to declare interest in flags
- Scope and extent?
- All flags or particular flags?

**SHALL** provide an invalid hierarchy

**MAY** allow users to add conditions
- Borneo’s admit-yields is a good example.
- Static scope and extent flag declarations make user-defined, software flags reasonable.
edit Move exception handling optimizations to an informative annex.
  
  • Describe fast trapping or conditional operations as optimizations

edit Eliminate signaling NaNs.

• Only current use of signaling NaNs: debugging.
You’re still here? Go home.