Performance, Correctness, Exceptions: Pick Three

Andrea Gussoni, Alessandro Di Federico,
Pietro Fezzardi, Giovanni Agosta

Politecnico di Milano

24 February 2019

Table of Contents

Motivations

Static binary translation has a variety of possible uses:

- Support for legacy code.
- Performance improvement for legacy architectures.
- Instrumentation of code.

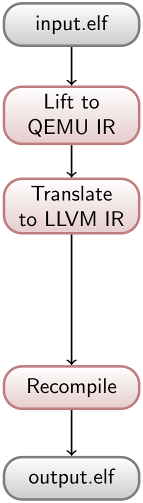
Goals

- Improve the performance of the translated binaries.
- Do not reinvent the wheel, use as much as possible *off-the-shelf* components.
- Be architecture independent, as the the whole `rev.ng` framework.

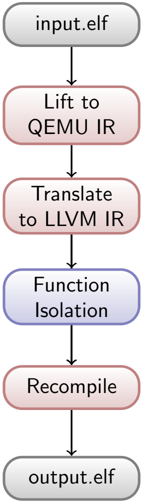
Table of Contents

rev.ng

rev.ng


input.elf
Lift to QEMU IR
Translate to LLVM IR
Recompile
output.elf

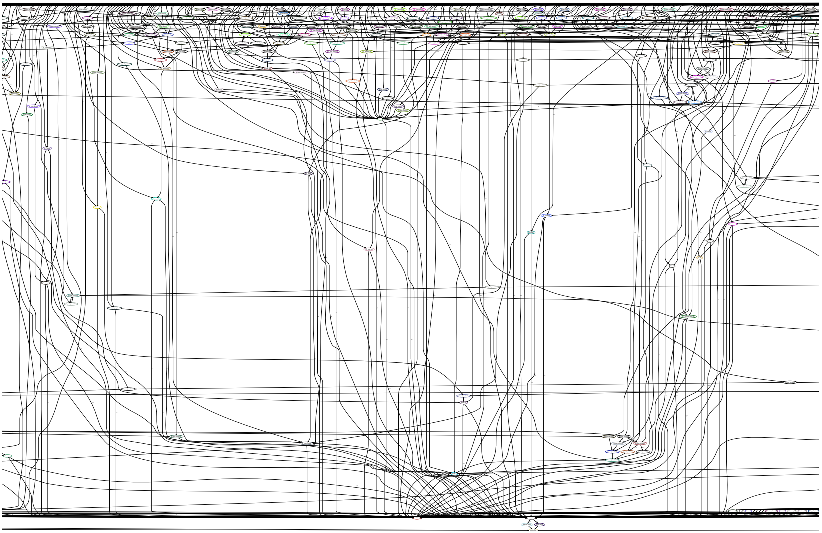
rev.ng


input.elf
Lift to QEMU IR
Translate to LLVM IR
Function Isolation
Recompile
output.elf

The root Function

- At the present time, the lifting phase places all the code recovered from the binary in a *single* (and often large) LLVM function, that we call `root`.

The root Function



The Dispatcher

- What about indirect branches or indirect function calls (e.g. `jmp rax`)?
- We need the *dispatcher*.

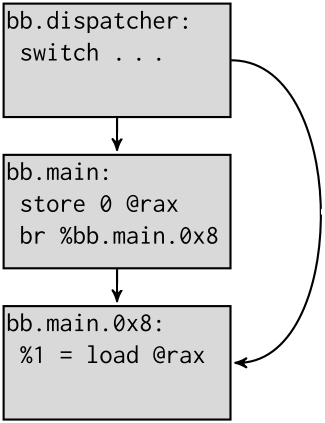
The Dispatcher

```
switch i64 @pc, label %dispatcher.default [
  i64 4194536, label %bb._init
  i64 4194542, label %bb._init.0x6
  i64 4194547, label %bb._init.0xb
  i64 4194560, label %bb._start
  i64 4194582, label %bb._start_c
  i64 4194614, label %bb._start.0x36
  i64 4194624, label %bb.deregister_tm_clones
  i64 4194645, label %bb.deregister_tm_clones.0x15
  i64 4194655, label %bb.deregister_tm_clones.0x1f
  i64 4194672, label %bb.deregister_tm_clones.0x30
  i64 4194688, label %bb.register_tm_clones
  i64 4194723, label %bb.register_tm_clones.0x23
  i64 4194733, label %bb.register_tm_clones.0x2d
  i64 4194744, label %bb.register_tm_clones.0x38
]
```

Current Limitations

- One mayor problem of the dispatcher is that every time we need to pass through it, we pay an *high cost* in terms of performance.
- The CFG of the `root` function contains a lot of unnecessary edges, and this leads to a *mazy* topology.
- This topology prevents a lot of `opt` optimizations.

Current Limitations


$$\begin{tikzcd}
\boxed{\texttt{bb.dispatcher: switch ...}} \arrow[d] \arrow[dd, bend left] \\
\boxed{\texttt{bb.main: store 0 @rax; br \%bb.main.0x8}} \arrow[d] \\
\boxed{\texttt{bb.main.0x8: \%1 = load @rax}}
\end{tikzcd}$$

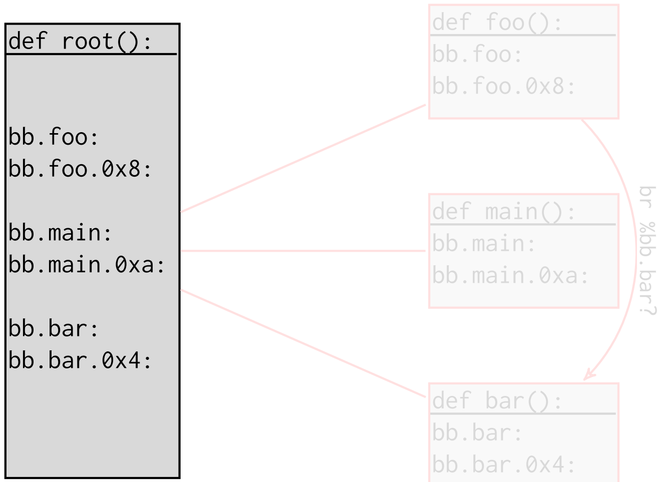
```
bb.dispatcher:
 switch . . .

bb.main:
 store 0 @rax
 br %bb.main.0x8

bb.main.0x8:
 %1 = load @rax
```

Table of Contents

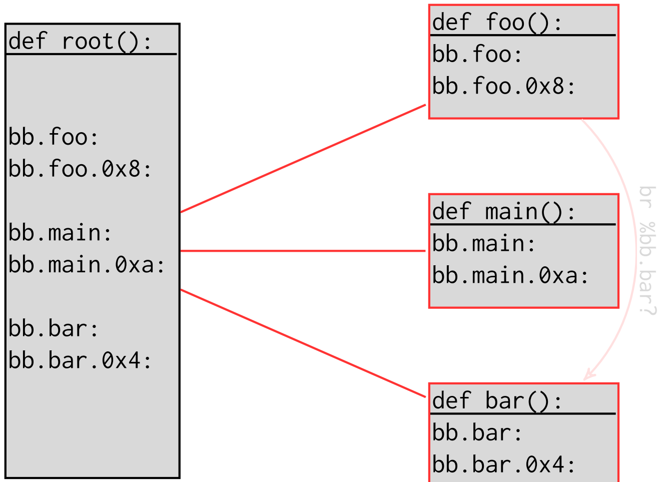
A naive approach

- The natural thing to do is try to *reconstruct* (with some approximations) the original *function layout*.
- Will things break? (Spoiler: yes, they will).

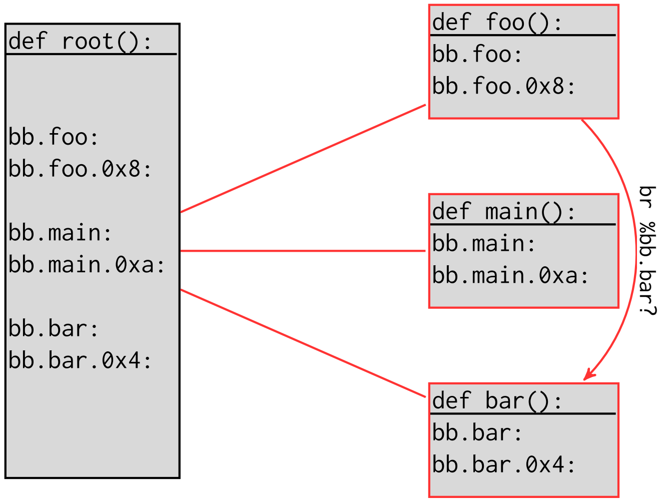
Bird View


def root():
bb.foo:
bb.foo.0x8:
bb.main:
bb.main.0xa:
bb.bar:
bb.bar.0x4:
def foo():
bb.foo:
bb.foo.0x8:
def main():
bb.main:
bb.main.0xa:
def bar():
bb.bar:
bb.bar.0x4:
br %bb.bar?

Bird View


def root():
bb.foo:
bb.foo.0x8:
bb.main:
bb.main.0xa:
bb.bar:
bb.bar.0x4:
def foo():
bb.foo:
bb.foo.0x8:
def main():
bb.main:
bb.main.0xa:
def bar():
bb.bar:
bb.bar.0x4:
br %bb.bar?

Bird View


def root():
bb.foo:
bb.foo.0x8:
bb.main:
bb.main.0xa:
bb.bar:
bb.bar.0x4:
def foo():
bb.foo:
bb.foo.0x8:
def main():
bb.main:
bb.main.0xa:
def bar():
bb.bar:
bb.bar.0x4:
br %bb.bar?

- What if we make the isolated functions and the `root` function *coexist*?

Isolated and Non-Isolated Realms

We define these two *realms*:

Isolated Realm In this realm, we have a new LLVM function for each function discovered by the FBDA.

Non-Isolated Realm In this realm the original `root` function has been preserved, basically unaltered.

To the Isolated Realm and Back

- Transitioning to the isolated realm is easy, every time we find a basic block that is an *entry point* of a function, we call the corresponding isolated function in the isolated realm.
- The transition in the opposite direction is more complicated, our idea is to exploit the *exception handling mechanism* provided by LLVM.

To the Isolated Realm and Back

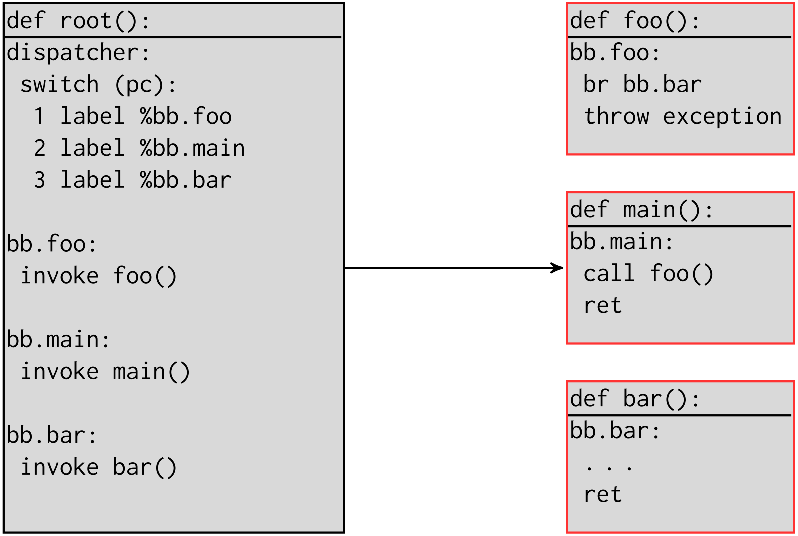
```
def root():
dispatcher:
 switch (pc):
  1 label %bb.foo
  2 label %bb.main
  3 label %bb.bar

bb.foo:
 invoke foo()

bb.main:
 invoke main()

bb.bar:
 invoke bar()
```

```
def foo():
bb.foo:
 br bb.bar
 throw exception
```

```
def main():
bb.main:
 call foo()
 ret
```

```
def bar():
bb.bar:
 . . .
 ret
```

To the Isolated Realm and Back



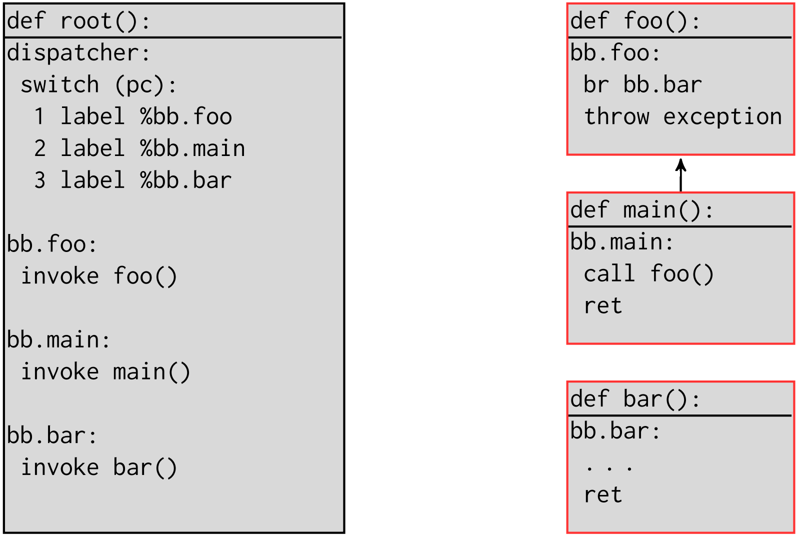
```
def root():
dispatcher:
 switch (pc):
  1 label %bb.foo
  2 label %bb.main
  3 label %bb.bar

bb.foo:
 invoke foo()

bb.main:
 invoke main()

bb.bar:
 invoke bar()
```

```
def foo():
bb.foo:
 br bb.bar
 throw exception
```

```
def main():
bb.main:
 call foo()
 ret
```

```
def bar():
bb.bar:
 . . .
 ret
```

To the Isolated Realm and Back



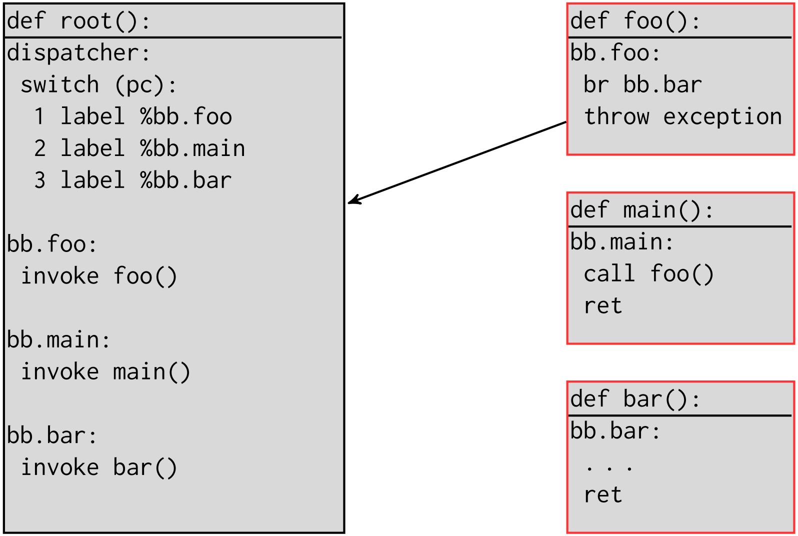
```
def root():
dispatcher:
 switch (pc):
  1 label %bb.foo
  2 label %bb.main
  3 label %bb.bar

bb.foo:
 invoke foo()

bb.main:
 invoke main()

bb.bar:
 invoke bar()
```

```
def foo():
bb.foo:
 br bb.bar
 throw exception
```

```
def main():
bb.main:
 call foo()
 ret
```

```
def bar():
bb.bar:
 . . .
 ret
```

To the Isolated Realm and Back



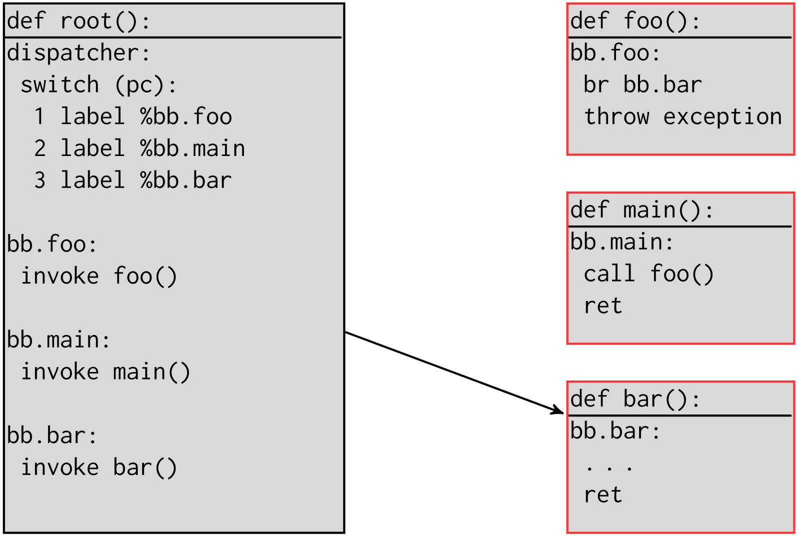
```
def root():
dispatcher:
 switch (pc):
  1 label %bb.foo
  2 label %bb.main
  3 label %bb.bar

bb.foo:
 invoke foo()

bb.main:
 invoke main()

bb.bar:
 invoke bar()
```

```
def foo():
bb.foo:
 br bb.bar
 throw exception
```

```
def main():
bb.main:
 call foo()
 ret
```

```
def bar():
bb.bar:
 . . .
 ret
```

To the Isolated Realm and Back



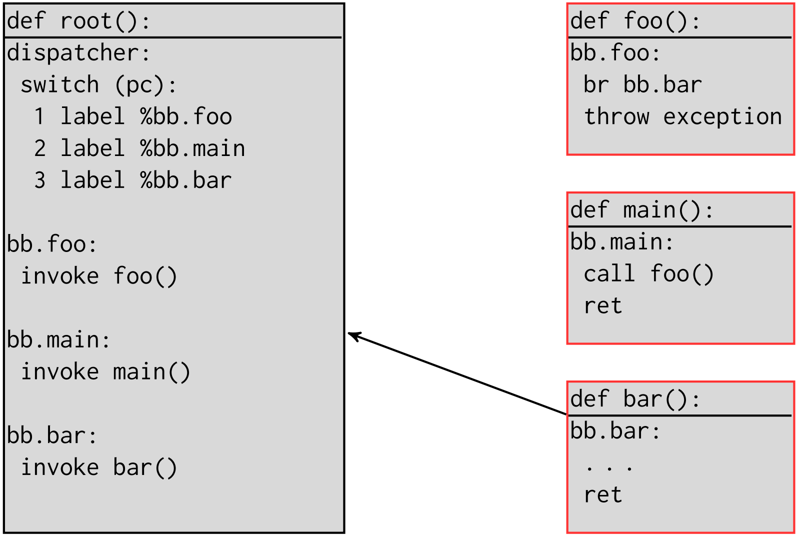
```
def root():
dispatcher:
 switch (pc):
  1 label %bb.foo
  2 label %bb.main
  3 label %bb.bar

bb.foo:
 invoke foo()

bb.main:
 invoke main()

bb.bar:
 invoke bar()
```

```
def foo():
bb.foo:
 br bb.bar
 throw exception
```

```
def main():
bb.main:
 call foo()
 ret
```

```
def bar():
bb.bar:
 . . .
 ret
```

To the Isolated Realm and Back



```
def root():
dispatcher:
 switch (pc):
  1 label %bb.foo
  2 label %bb.main
  3 label %bb.bar

bb.foo:
 invoke foo()

bb.main:
 invoke main()

bb.bar:
 invoke bar()
```

```
def foo():
bb.foo:
 br bb.bar
 throw exception
```

```
def main():
bb.main:
 call foo()
 ret
```

```
def bar():
bb.bar:
 . . .
 ret
```

Exception Handling Mechanism

Our fallback mechanism is implemented using:

- The exception support provided by the LLVM framework.
- The stack unwinding mechanism via `libgcc`.

Function Isolation

- Function isolation is performed on the basis of the information provided by the Function Boundaries Detection Analysis pass.
- The accuracy of the FBDA is an important factor for performing an high quality function isolation.
- The quality of the function isolation determines how much the fallback-mechanism is actually employed.

Function Boundaries Analysis Limitations

- There are situations (e.g. exceptions in the original code), where the good (or even optimal) quality of the FBDA will not be sufficient.
- Our fallback mechanism guarantees that we can handle the execution in these situations.
- We handle exceptions with exceptions!

Table of Contents

Experimental Setup

- We used the SPECint 2006 benchmark suite.
- 4 configurations:
  - Native
  - QEMU
  - `rev.ng`
  - `rev.ng` with isolation

Experimental Results

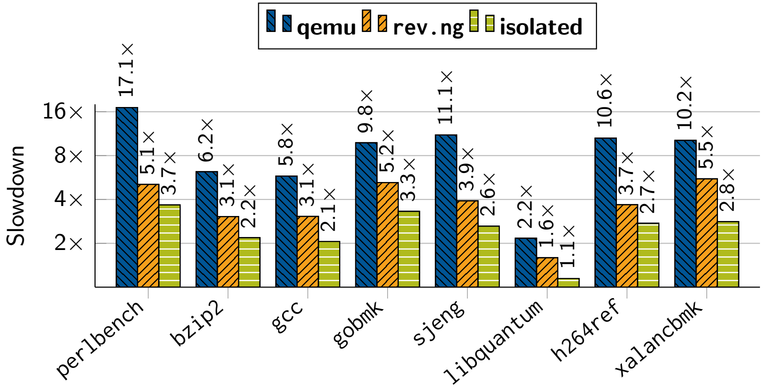


Figure: Slowdown of the different translation techniques compared to native code. Logarithmic scale. Lower is better.

Table of Contents

Future Work

- Recognize function parameters.
- Recognize return values.
- Promote global variables (registers) to local variables when possible.

Resources

- The function isolation feature has been implemented in `rev.ng` as a LLVM pass.
- The artifacts produced during the work, the code and the instructions to reproduce them are available at `https://rev.ng/gitlab/revng-bar-2019/artifacts`.
- If you are interested in more general instructions on how to get started with `rev.ng`, you can check the official website at `https://rev.ng/getting-started.html`.

Questions?

License

Backup Slides

Backup Slides

LLVM IR

```c
int counter;

int main(int argc) {
  if (argc > 5) {



    counter++;





  } else {
    myfunction();
  }


  return 1;

}
```

```llvm
@counter = common global i32 0

define i32 @main ( i32 %argc ) {
  %1 = icmp sgt i32 %argc , 5
  br i1 %1 , label %yes , label %no

yes :
  %2 = load i32 , i32 * @counter
  %3 = add i32 %2 , 1
  store i32 %3 , i32 * @counter
  br label %end

no :
  call void @otherfunction ()
  br label %end

end :
  ret i32 1

}
```

Exception Handling Mechanism

To do this, we mainly used the *exception handling* mechanism provided by LLVM. In our solution, this mechanism is in charge of recovering a potentially faulty situation, for example when static analysis cannot foresee the destination of a jump, taking care of redirecting the execution to a component that is in charge of understanding what to do next.

Exception Handling Mechanism

At the implementation level, for using exceptions we need to:

- Replace in the `root` function, each function entry basic block body with an `invoke` instruction (a peculiar `call` instruction) to the isolated function.
- In the isolated realm, each time we need to exit from the isolated function in an unexpected manner, throw an exception.
- Provide to LLVM a *personality function*, which is a function that is in charge of specifying the runtime behavior when an exception is thrown.

CSV

- `rev.ng` represents the current CPU state using the so called *CSV* (CPU state variable), which are LLVM *global variables*.
- In the general case, this is a great bottleneck for the performances (we need to go through memory).