ERLANG JIT COMPILER

LUKAS LARSSON

lukas@erlang-solutions.com

@garazdawi

Erlang factory San Francisco Bay Area 2012

OUTLINE

- JIT compilers
- The VM today
- The Idea
- The Prototype

INTRODUCTION TO JIT COMPILERS

Compiled	AHOT	On demand JIT
Interpreted	Method based eventual JIT	Trace based eventual JIT

INTRODUCTION TO JIT COMPILERS

Compiled	AHOT    	On demand JIT
Interpreted	Method based eventual JIT	Trace based eventual JIT

INTRODUCTION TO JIT COMPILERS

Compiled	AHOT    	On demand JIT   v8
Interpreted	Method based eventual JIT	Trace based eventual JIT

INTRODUCTION TO JIT COMPILERS

Compiled	AHOT    	On demand JIT   v8
Interpreted	Method based eventual JIT  **JägerMonkey**	Trace based eventual JIT

INTRODUCTION TO JIT COMPILERS

	AHOT	On demand JIT
Compiled	LLVM COMPILER INFRASTRUCTURE, GCC, Visual C++, Haskell	Microsoft .NET, v8
Interpreted	Method based eventual JIT: Java, JägerMonkey	Trace based eventual JIT: Java, TraceMonkey, JägerMonkey, LuaJIT, pypy

METHOD VS TRACING JIT COMPILER

```
add(A,B,C) ->
    Y = A + B,
    X = Y + C.
```

METHOD VS TRACING JIT COMPILER

```
add(A,B,C) ->
  Y = A + B,
  X = Y + C.
```

```
if is_small(A) and is_small(B)
    Y = untag(A) + untag(B)
    if (fits_in_small(Y))
        Y = tag(Y)
        goto *nextInstr
goto mixed_add
```

METHOD VS TRACING JIT COMPILER

```
add(A,B,C) ->
  Y = A + B,
  X = Y + C.
```

```
if is_small(A) and is_small(B)
    Y = untag(A) + untag(B)
    if (fits_in_small(Y))
        Y = tag(Y)
        goto *nextInstr
goto mixed_add
```

```
mixed_add:
    if is_small(A)
        if is_big(B)
            A = add_big(small_to_big(A),B)
        else if is_float(B)
            A = add_float(small_to_float(A),B)
        else
            return badarith
    else if is_small(B)
        if is_big(A)
            A = add_big(small_to_big(B),A)
        else if is_float(A)
            A = add_float(small_to_float(B),A)
        else
            return badarith
    else if is_big(A)
        if is_big(B)
            A = add_big(A,B)
        else if is_float(B)
            A = add_float(big_to_float(A),B)
        else
            return badarith
    else if is_float(A)
        if is_float(B)
            A = add_float(A,B)
        else if is_big(B)
            A = add_float(big_to_float(B),A)
        else
            return badarith
    else
        return badarith
goto *nextInstr
```

INTRODUCTION TO JIT COMPILERS

Method based JIT

add(B, C, Y)

```
add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

INTRODUCTION TO JIT COMPILERS

Method based JIT

```
add(B, C, Y)
    if is_small(B) and is_small(C)
        A = untag(B) + untag(C)
        if fits_in_small(A)
            A = tag(A)
            goto add_x
    goto mixed_add
```

```
add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

INTRODUCTION TO JIT COMPILERS

Method based JIT

```
add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

```
add(B, C, Y)
    if is_small(B) and is_small(C)
        A = untag(B) + untag(C)
        if fits_in_small(A)
            A = tag(A)
            goto add_x
    goto mixed_add
add_x:
    if is_small(A) and is_small(Y)
        X = untag(A) + untag(Y)
        if fits_in_small(Y)
            X = tag(X)
            goto return_x
    goto mixed_add
```

INTRODUCTION TO JIT COMPILERS

Method based JIT

```
add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

```
add(B, C, Y)
    if is_small(B) and is_small(C)
        A = untag(B) + untag(C)
        if fits_in_small(A)
            A = tag(A)
            goto add_x
    goto mixed_add
add_x:
    if is_small(A) and is_small(Y)
        X = untag(A) + untag(Y)
        if fits_in_small(Y)
            X = tag(X)
            goto return_x
    goto mixed_add
return_x:
    return X
```

INTRODUCTION TO JIT COMPILERS

Method based JIT

mixed_add not expanded!

```
add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

```
, C, Y)
s_small(B) and is_small(C)
 A = untag(B) + untag(C)
 if fits_in_small(A)
    A = tag(A)
    goto add_x
 goto mixed_add
add_x:
 if is_small(A) and is_small(Y)
    X = untag(A) + untag(Y)
    if fits_in_small(Y)
       X = tag(X)
       goto return_x
    goto mixed_add
return_x:
 return X
```

INTRODUCTION TO JIT COMPILERS

Method based JIT

```
add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

```
add(B, C, Y)
    if is_small(B) and is_small(C)
        A = untag(B) + untag(C)
        if fits_in_small(A)
            A = tag(A)
            goto add_x
    goto mixed_add
add_x:
    if is_small(A) and is_small(Y)
        X = untag(A) + untag(Y)
        if fits_in_small(Y)
            X = tag(X)
            goto return_x
    goto mixed_add
return_x:
    return X
```

Trace based JIT

```
add(B, C, Y)
```

INTRODUCTION TO JIT COMPILERS

```
add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

Method based JIT

```
add(B, C, Y)
    if is_small(B) and is_small(C)
        A = untag(B) + untag(C)
        if fits_in_small(A)
            A = tag(A)
            goto add_x
    goto mixed_add
add_x:
    if is_small(A) and is_small(Y)
        X = untag(A) + untag(Y)
        if fits_in_small(Y)
            X = tag(X)
            goto return_x
    goto mixed_add
return_x:
    return X
```

Trace based JIT

```
add(B, C, Y)
    guard is_small(B)
```

If a guard fails we go back to interpreted mode

INTRODUCTION TO JIT COMPILERS

Method based JIT

```
add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

```
add(B, C, Y)
    if is_small(B) and is_small(C)
        A = untag(B) + untag(C)
        if fits_in_small(A)
            A = tag(A)
            goto add_x
    goto mixed_add
add_x:
    if is_small(A) and is_small(Y)
        X = untag(A) + untag(Y)
        if fits_in_small(Y)
            X = tag(X)
            goto return_x
    goto mixed_add
return_x:
    return X
```

Trace based JIT

```
add(B, C, Y)
    guard is_small(B)
    guard is_small(C)
```

INTRODUCTION TO JIT COMPILERS

Method based JIT

```
add(B, C, Y)
    if is_small(B) and is_small(C)
        A = untag(B) + untag(C)
        if fits_in_small(A)
            A = tag(A)
            goto add_x
    goto mixed_add
add_x:
    if is_small(A) and is_small(Y)
        X = untag(A) + untag(Y)
        if fits_in_small(Y)
            X = tag(X)
            goto return_x
    goto mixed_add
return_x:
    return X
```

Trace based JIT

```
add(B, C, Y)
    guard is_small(B)
    guard is_small(C)
    A = untag(B) + untag(C)
```

```
add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

INTRODUCTION TO JIT COMPILERS

```
add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

Method based JIT

```
add(B, C, Y)
    if is_small(B) and is_small(C)
        A = untag(B) + untag(C)
        if fits_in_small(A)
            A = tag(A)
            goto add_x
    goto mixed_add
add_x:
    if is_small(A) and is_small(Y)
        X = untag(A) + untag(Y)
        if fits_in_small(Y)
            X = tag(X)
            goto return_x
    goto mixed_add
return_x:
    return X
```

Trace based JIT

```
add(B, C, Y)
    guard is_small(B)
    guard is_small(C)
    A = untag(B) + untag(C)
    guard fits_in_small(A)
```

INTRODUCTION TO JIT COMPILERS

```
add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

Method based JIT

```
add(B, C, Y)
    if is_small(B) and is_small(C)
        A = untag(B) + untag(C)
        if fits_in_small(A)
            A = tag(A)
            goto add_x
    goto mixed_add
add_x:
    if is_small(A) and is_small(Y)
        X = untag(A) + untag(Y)
        if fits_in_small(Y)
            X = tag(X)
            goto return_x
    goto mixed_add
return_x:
    return X
```

Trace based JIT

```
add(B, C, Y)
    guard is_small(B)
    guard is_small(C)
    A = untag(B) + untag(C)
    guard fits_in_small(A)
    A = tag(A)
```

INTRODUCTION TO JIT COMPILERS

```
add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

Method based JIT

```
add(B, C, Y)
    if is_small(B) and is_small(C)
        A = untag(B) + untag(C)
        if fits_in_small(A)
            A = tag(A)
            goto add_x
    goto mixed_add
add_x:
    if is_small(A) and is_small(Y)
        X = untag(A) + untag(Y)
        if fits_in_small(Y)
            X = tag(X)
            goto return_x
    goto mixed_add
return_x:
    return X
```

Trace based JIT

```
add(B, C, Y)
    guard is_small(B)
    guard is_small(C)
    A = untag(B) + untag(C)
    guard fits_in_small(A)
    A = tag(A)
    guard is_small(A)
    guard is_small(Y)
    X = untag(A) + untag(Y)
    guard fits_in_small(X)
    X = tag(D)
```

INTRODUCTION TO JIT COMPILERS

```
add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

Method based JIT

```
add(B, C, Y)
    if is_small(B) and is_small(C)
        A = untag(B) + untag(C)
        if fits_in_small(A)
            A = tag(A)
            goto add_x
    goto mixed_add
add_x:
    if is_small(A) and is_small(Y)
        X = untag(A) + untag(Y)
        if fits_in_small(Y)
            X = tag(X)
            goto return_x
    goto mixed_add
return_x:
    return X
```

Trace based JIT

```
add(B, C, Y)
    guard is_small(B)
    guard is_small(C)
    A = untag(B) + untag(C)
    guard fits_in_small(A)
    A = tag(A)
    guard is_small(A)
    guard is_small(Y)
    X = untag(A) + untag(Y)
    guard fits_in_small(X)
    X = tag(D)
    return X
```

ADVANCED METHOD JIT EXAMPLE

```
add(B,C) ->
    D = add(B, 2, 3),
    add(C, D, 4).
```

```
add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

ADVANCED METHOD JIT EXAMPLE

```
add(B,C) ->
    D = add(B, 2, 3),
    add(C, D, 4).

add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

```
add(B, C)
    C1 = C
    C = 2, Y = 3
    if is_small(B) and is_small(C)
        A = untag(B) + untag(C)
        if fits_in_small(A)
            A = tag(A)
            goto x_add1
    goto big_arith
add_x1:
    if is_small(A) and is_small(Y)
        X = untag(A) + untag(Y)
        if fits_in_small(Y)
            X = tag(X)
            goto return_x1
    goto big_arith
```

ADVANCED METHOD JIT EXAMPLE

```
add(B,C) ->
    D = add(B, 2, 3),
    add(C, D, 4).

add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

```
add(B, C)
    C1 = C
    C = 2, Y = 3
    if is_small(B) and is_small(C)
        A = untag(B) + untag(C)
        if fits_in_small(A)
            A = tag(A)
            goto x_add1
    goto big_arith
add_x1:
    if is_small(A) and is_small(Y)
        X = untag(A) + untag(Y)
        if fits_in_small(Y)
            X = tag(X)
            goto return_x1
    goto big_arith
```

```
return_x1:
    B = C1
    C = X, Y = 4
    if is_small(B) and is_small(C)
        A = untag(B) + untag(C)
        if fits_in_small(A)
            A = tag(A)
            goto x_add2
    goto big_arith
add_x2:
    if is_small(A) and is_small(Y)
        X = untag(A) + untag(Y)
        if fits_in_small(Y)
            X = tag(X)
            goto return_x2
    goto big_arith
return_x2:
    return X
```

ADVANCED TRACING JIT EXAMPLE

```
add(B,C) ->
    D = add(B, 2, 3),
    add(C, D, 4).

add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

```
C1 = C
C = 2, Y = 3
guard (is_small(B))
guard (is_small(C))
A = untag(B) + untag(C)
guard(fits_in_small(A))
A = tag(A)
guard (is_small(A))
guard (is_small(Y))
X = untag(A) + untag(Y)
guard (fits_in_small(X))
X = tag(X)
B = C1
C = X, Y = 4
guard (is_small(B))
guard (is_small(C))
A = untag(B) + untag(C)
guard(fits_in_small(A))
A = tag(A)
guard (is_small(A))
guard (is_small(Y))
X = untag(A) + untag(Y)
guard (fits_in_small(X))
X = tag(X)
return X
```

ADVANCED TRACING JIT EXAMPLE

```
add(B,C) ->
    D = add(B, 2, 3),
    add(C, D, 4).

add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

```
C1 = C
C = 2, Y = 3
guard (is_small(B))
guard (is_small(C))
A = untag(B) + untag(C)
guard(fits_in_small(A))
A = tag(A)
guard (is_small(A))
guard (is_small(Y))
X = untag(A) + untag(Y)
guard (fits_in_small(X))
X = tag(X)
B = C1
C = X, Y = 4
guard (is_small(B))
guard (is_small(C))
A = untag(B) + untag(C)
guard(fits_in_small(A))
A = tag(A)
guard (is_small(A))
guard (is_small(Y))
X = untag(A) + untag(Y)
guard (fits_in_small(X))
X = tag(X)
return X
```

* Constant folding
* Escape analysis
* type unboxing
* CSE

optimize

```
guard (is_small(B))
guard (is_small(C))
D = untag(B) + 9
X = untag(C) + D
if fits_in_small(X)
    X = tag(X)
else
    X = int_to_big(X)
return X
```

ADVANCED TRACING JIT EXAMPLE

```
add(B,C) ->
    D = add(B, 2, 3),
    add(C, D, 4).

add(B,C,Y) ->
    A = B + C,
    X = A + Y.
```

```
C1 = C
C = 2, Y = 3
guard (is_small(B))
guard (is_small(C))
A = untag(B) + untag(C)
guard(fits_in_small(A))
A = tag(A)
guard (is_small(A))
guard (is_small(Y))
X = untag(A) + untag(Y)
guard (fits_in_small(X))
X = tag(X)
B = C1
C = X, Y = 4
guard (is_small(B))
guard (is_small(C))
A = untag(B) + untag(C)
guard(fits_in_small(A))
A = tag(A)
guard (is_small(A))
guard (is_small(Y))
X = untag(A) + untag(Y)
guard (fits_in_small(X))
X = tag(X)
return X
```

* Constant folding
* Escape analysis
* type unboxing
* CSE

optimize

```
guard (is_small(B))
guard (is_small(C))
D = untag(B) + 9
X = untag(C) + D
if fits_in_small(X)
    X = tag(X)
else
    X = int_to_big(X)
return X
```

Some cleanup logic as we can use `wordsize` bits when unboxed, but only `wordsize-4` when boxed

METHOD JIT COMPILERS

- Method/function based
  - Pros
    - Small memory footprint
    - All branches compiles to native
  - Cons
    - Compiles code which might not be in heavy use
    - Handles dependencies bad
    - Hard to do unboxing optmizations
- Trace based
  - Pros
    - Only compiles hot path
    - Automatic inlining
    - Room for much more aggressive optimizations
  - Cons
    - Performance loss while tracing a path
    - “Branchy” code performs badly

HOW THE VM WORKS TODAY

- JIT compilers
- The VM today
- The Idea
- The Prototype

HOW THE VM WORKS TODAY

Erlang Source File

```
fact(0) ->
    1;
fact(N) when N > 0 ->
    N * fact(N-1).
```

HOW THE VM WORKS TODAY


```
fact(0) ->                    {function, fact, 1, 2}.
    1;                          {label,2}.
fact(N) when N > 0 ->           {test,is_eq_exact,{f,3},[{x,0},{integer,0}]}.
    N * fact(N-1).              {move,{integer,1},{x,0}}.
                                return.
                              {label,3}.
                                {test,is_lt,{f,1},[{integer,0},{x,0}]}.
                                {allocate_zero,1,1}.
                                {gc_bif,'-',{f,0},1,[{x,0},{integer,1}],{x,1}}.
                                {move,{x,0},{y,0}}.
                                {move,{x,1},{x,0}}.
                                {call,1,{f,2}}.
                                {gc_bif,'*',{f,0},1,[{y,0},{x,0}],{x,0}}.
                                {deallocate,1}.
                                return.
```

HOW THE VM WORKS TODAY


```
fact(0) ->                      i_func_info_IaaI 0 fact fact 1
    1;                          i_is_eq_exact_immed_frc f(01DFBCC4) x(0) 0
fact(N) when N > 0 ->           move_return_cr 1 x(0)
    N * fact(N-1).              i_fetch_cr 0 x(0)
                                i_is_lt_f f(01DFBC9C)
                                allocate_zero_tt 1 1
                                i_fetch_rc x(0) 1
                                i_minus_jId j(00000000) 1 x(1)
                                move_ry x(0) y(0)
                                move_call_xrf x(1) x(0) fact:fact/1
                                i_fetch_yr y(0) x(0)
                                i_times_jId j(00000000) 1 x(0)
                                deallocate_return_Q 1
```

HOW THE VM WORKS TODAY


```
fact(0) ->
    1;
fact(N) when N > 0 ->
    N * fact(N-1).
```

```
{move,{x,1},{x,0}}.
{call,1,{f,2}}.
```

```
i_func_info_IaaI 0 fact fact 1
i_is_eq_exact_immed_frc f(01DFBCC4) x(0) 0
move_return_cr 1 x(0)
i_fetch_cr 0 x(0)
i_is_lt_f f(01DFBC9C)
allocate_zero_tt 1 1
i_fetch_rc x(0) 1
i_minus_jId j(00000000) 1 x(1)
move_ry x(0) y(0)
move_call_xrf x(1) x(0) fact:fact/1
i_fetch_yr y(0) x(0)
i_times_jId j(00000000) 1 x(0)
deallocate_return_Q 1
```

HOW THE VM WORKS TODAY

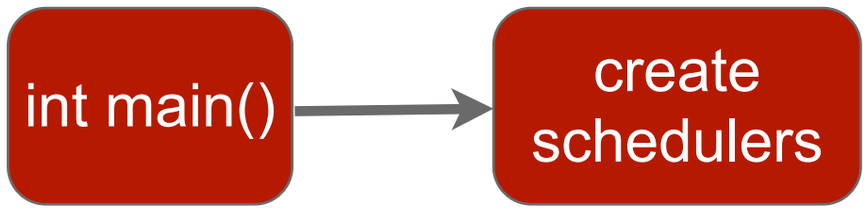

```
fact(0) ->                      i_func_info_IaaI 0 fact fact 1
    1;                          i_is_eq_exact_immed_frc f(01DFBCC4) x(0) 0
fact(N) when N > 0 ->           move_return_cr 1 x(0)
    N * fact(N-1).              i_fetch_cr 0 x(0)
                                i_is_lt_f f(01DFBC9C)
                                allocate_zero_tt 1 1
                                i_fetch_rc x(0) 1
                                i_minus_jId j(00000000) 1 x(1)
{move,{x,0},{y,0}}              move_ry x(0) y(0)
                                move_call_xrf x(1) x(0) fact:fact/1
                                i_fetch_yr y(0) x(0)
                                i_times_jId j(00000000) 1 x(0)
                                deallocate_return_Q 1
```

BEAM EMU LOOP

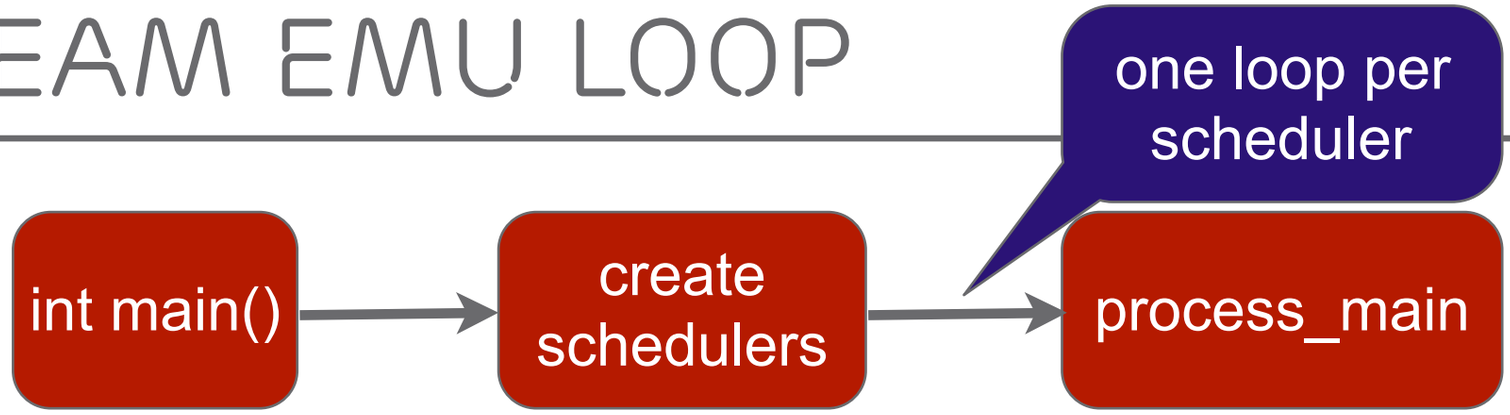
BEAM EMU LOOP

int main()

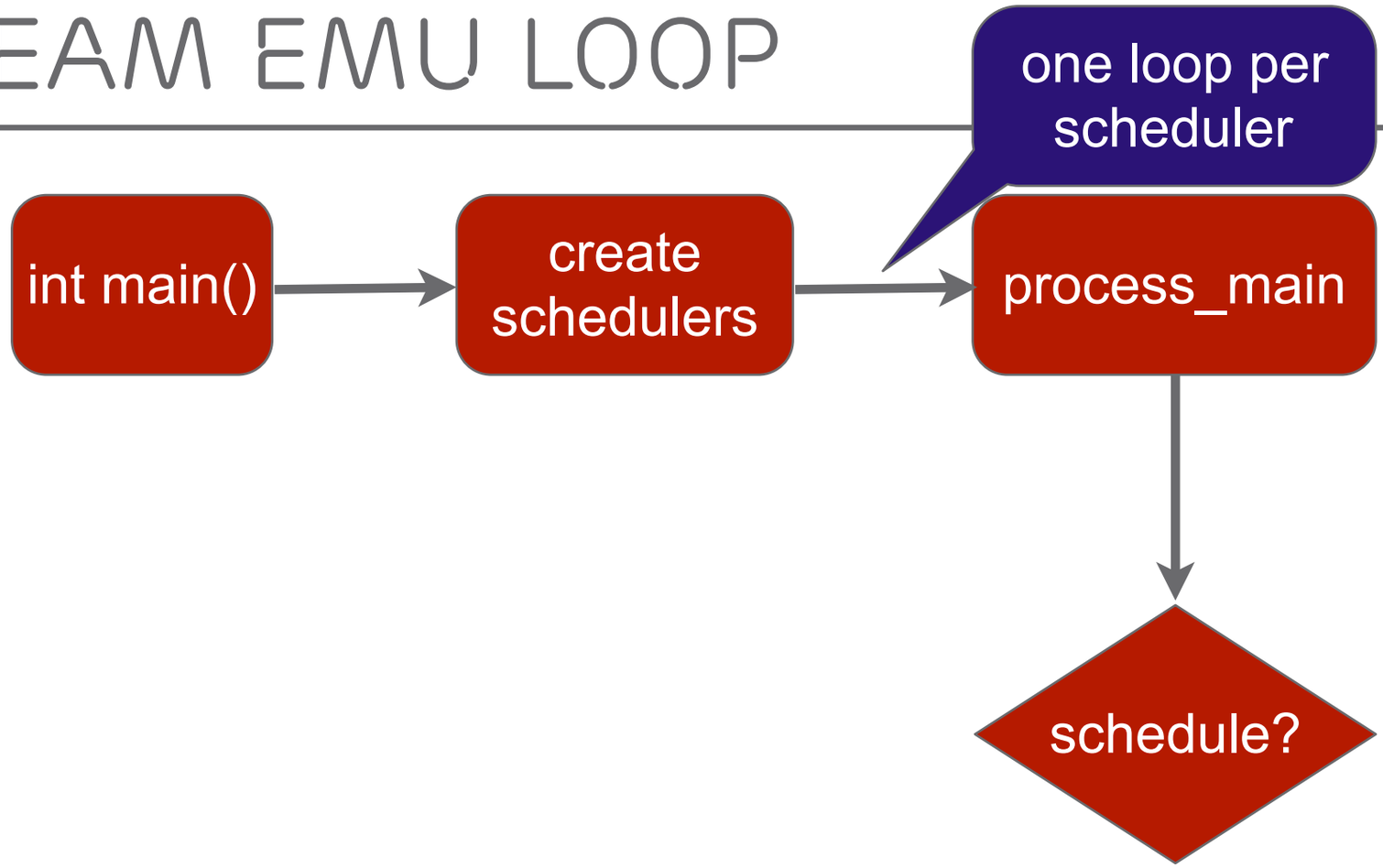
BEAM EMU LOOP



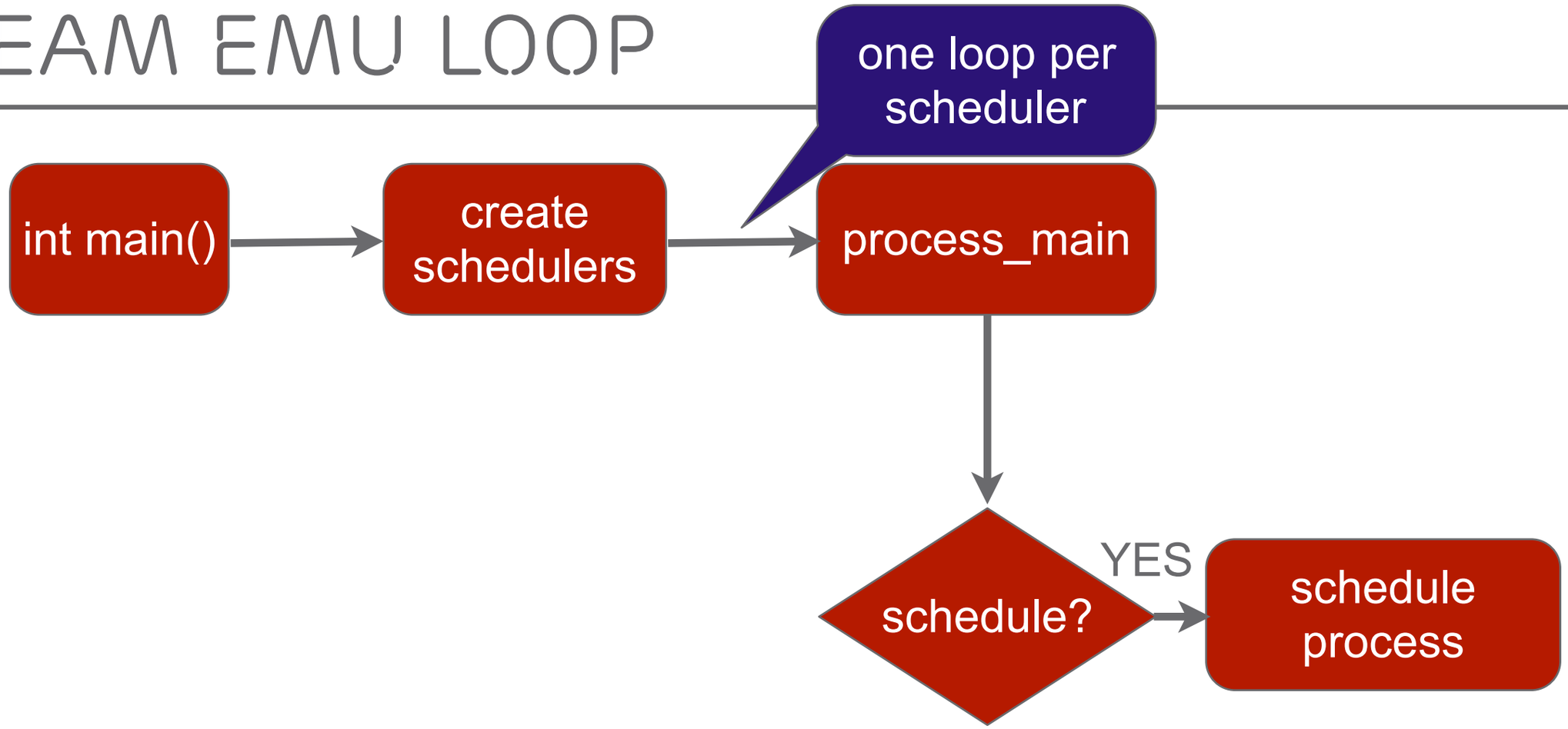
BEAM EMU LOOP



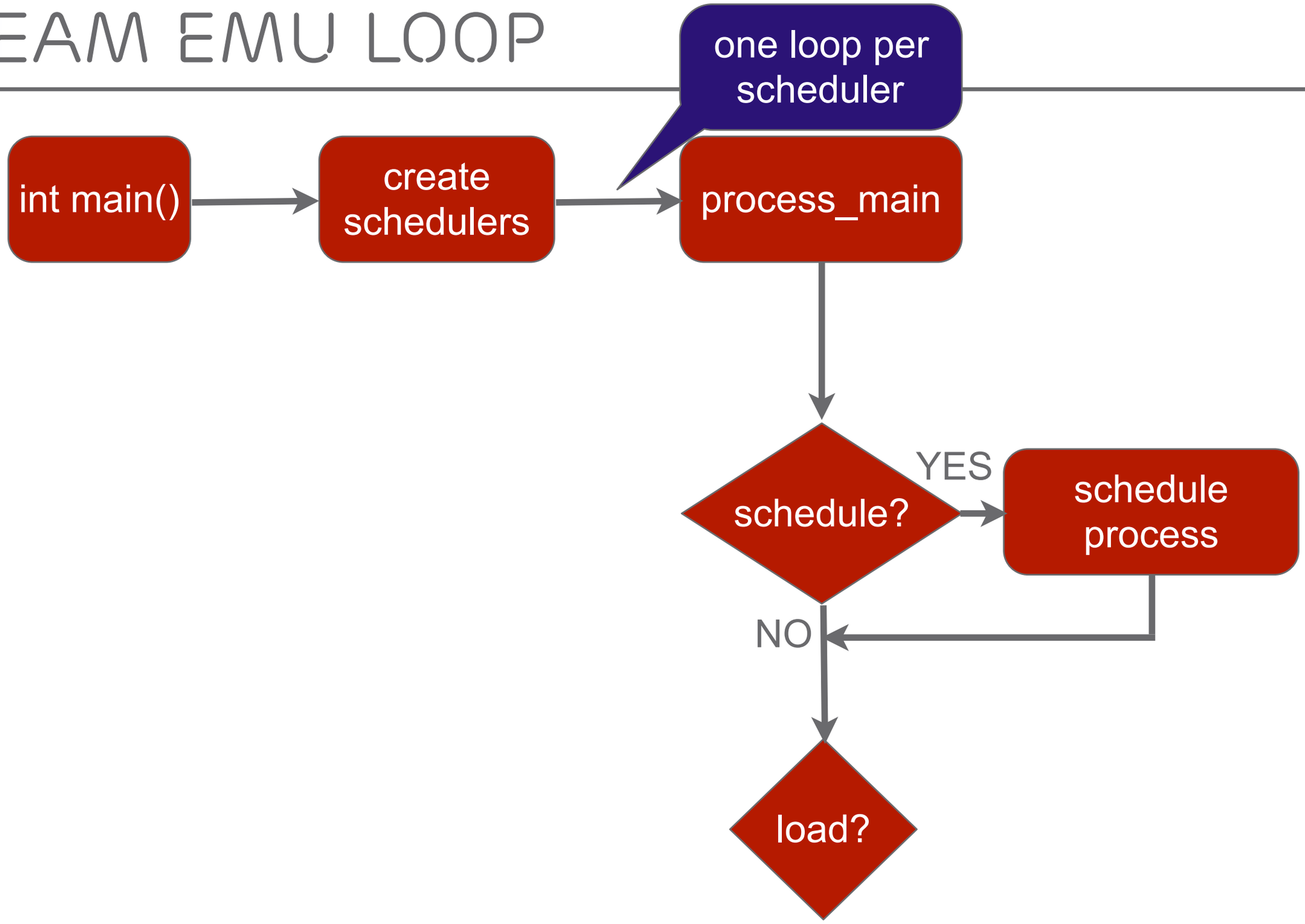
BEAM EMU LOOP



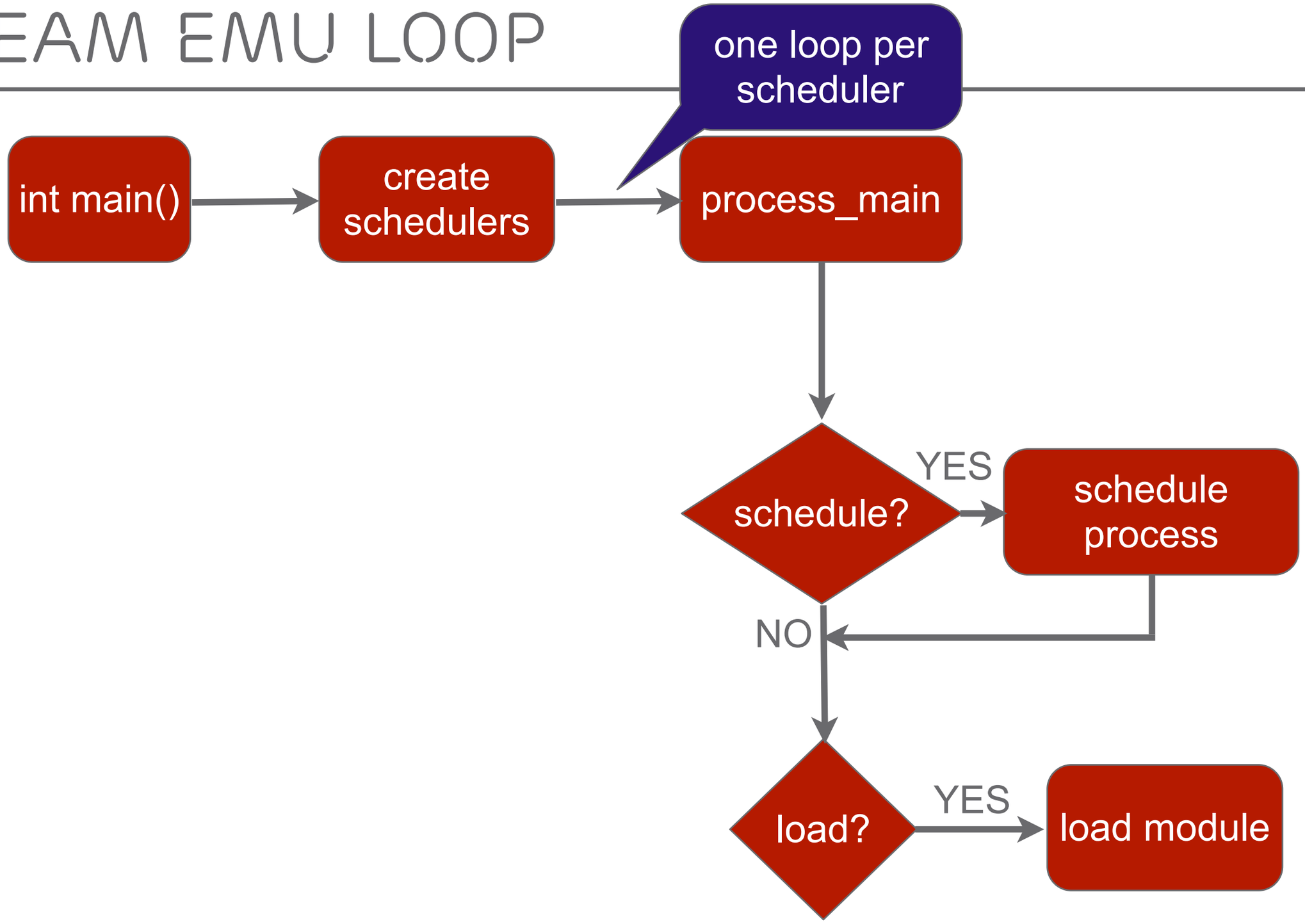
BEAM EMU LOOP



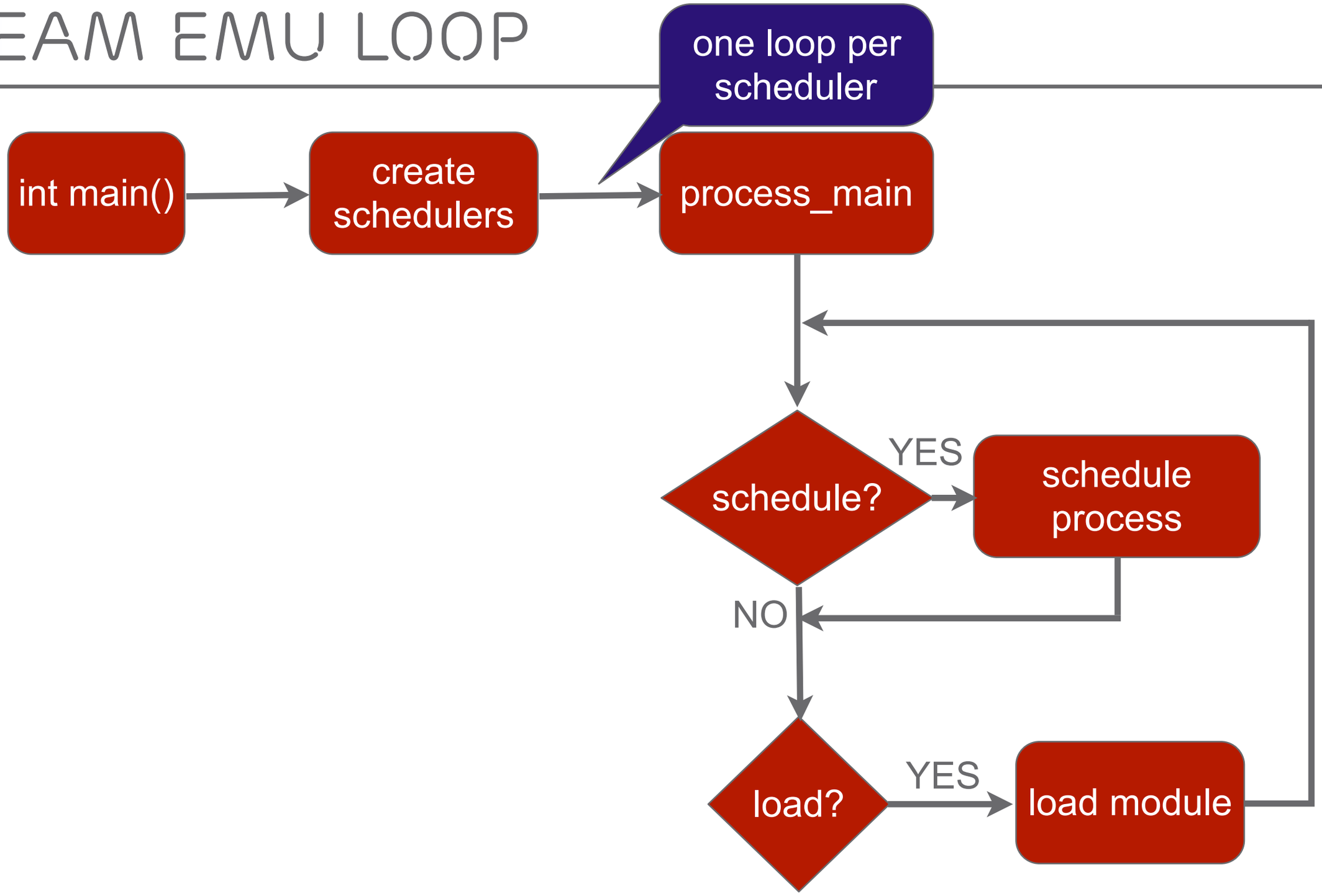
BEAM EMU LOOP



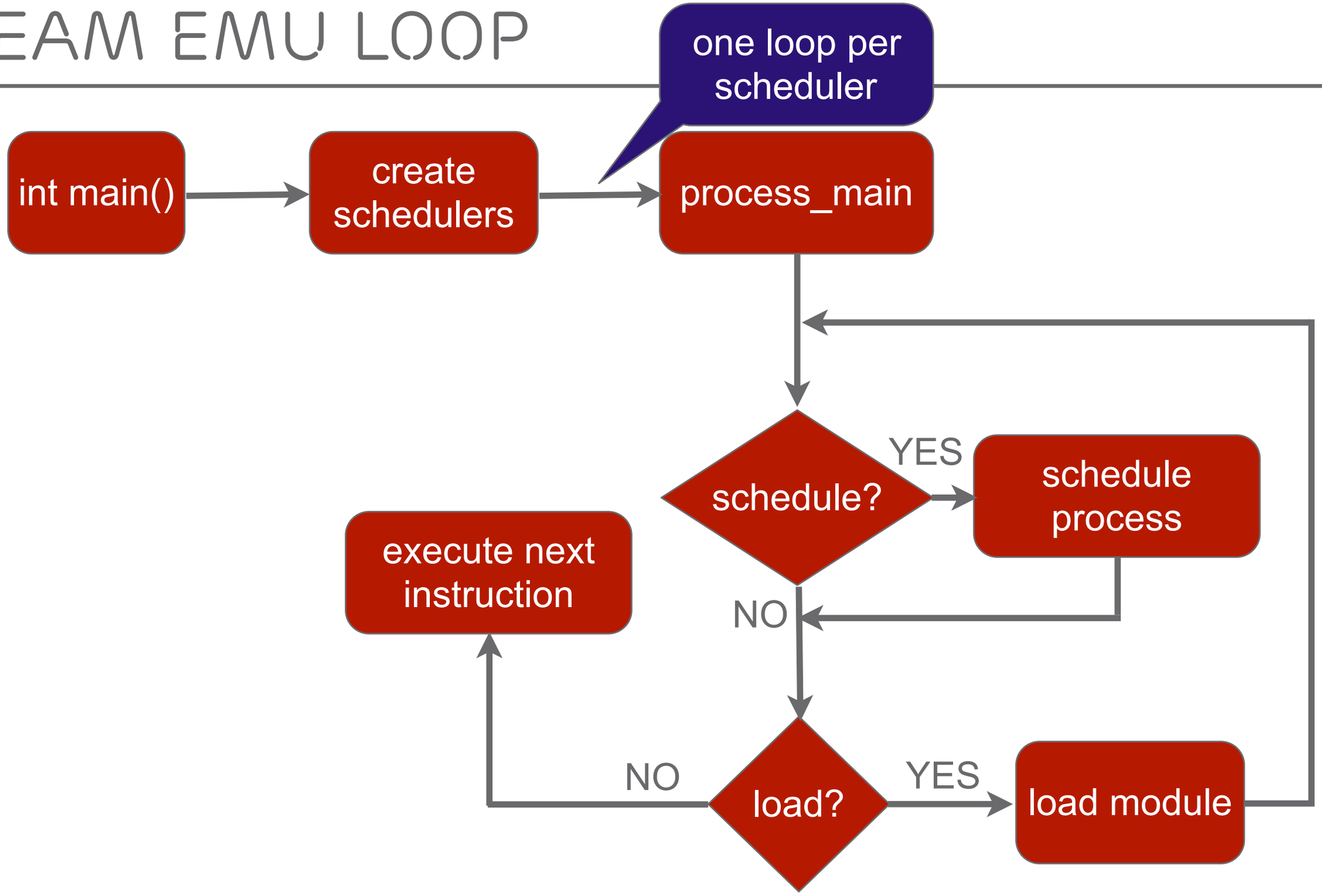
BEAM EMU LOOP



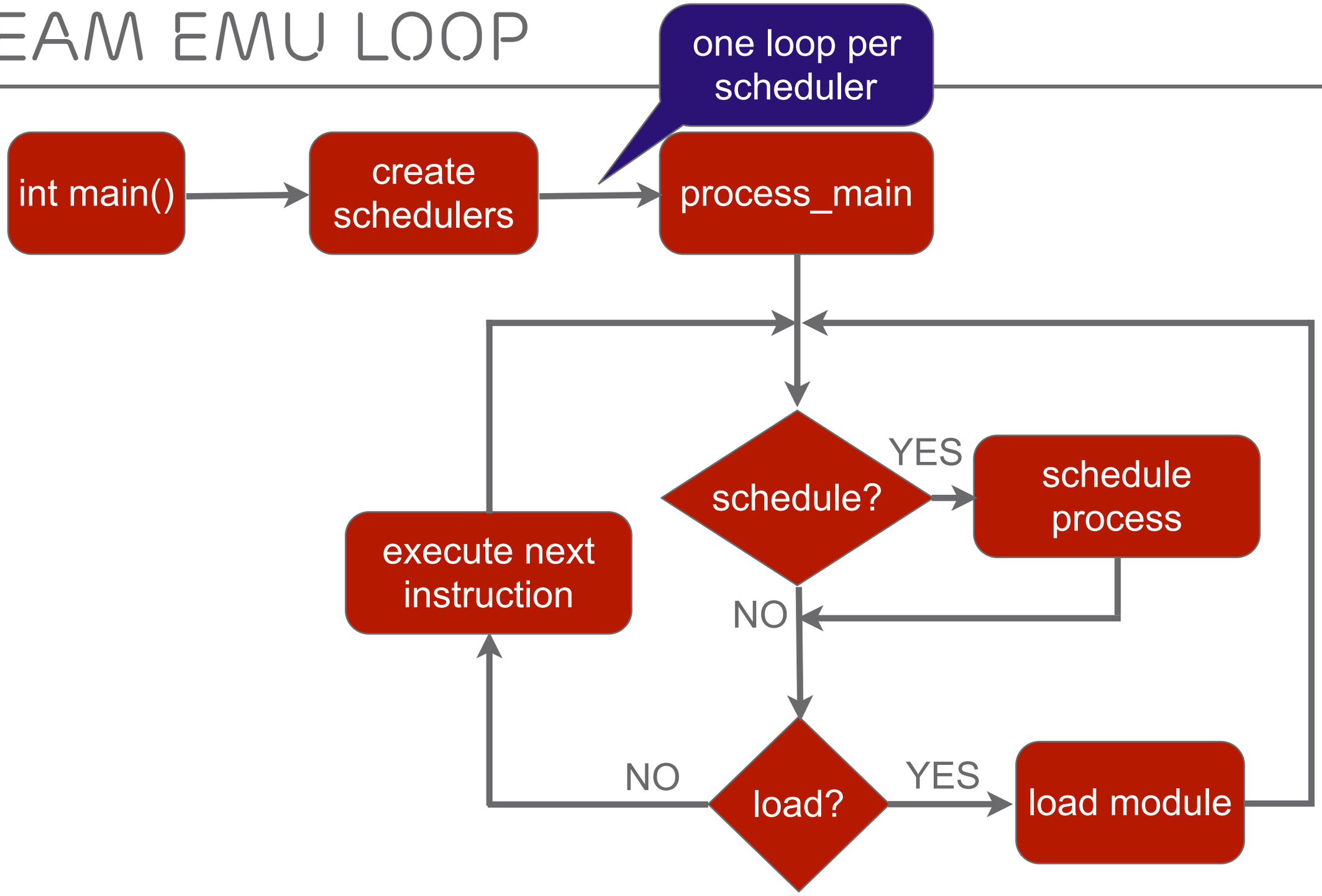
BEAM EMU LOOP



BEAM EMU LOOP



BEAM EMU LOOP



THE IDEA

- JIT compilers
- The VM today
- The Idea
- The Prototype

THE REQUIREMENTS

- Work across modules
- Fast switch between interpreter and JIT:ed code
- Preemptive
- Fast prototype to prove concept
- Easy to maintain as BEAM evolves

THE IDEA

- Tracing JIT for Erlang code
- Recursive loops called most are “hot” paths
- Collect a trace of basic blocks
- JIT:ed code work directly on Erlang stack
  - Immediate swap from JITed to interpreted code
- Abort JIT run at rescheduling
- Different optimization strategies depending on trace lifetime
- LLVM compiler toolchain do heavy lifting

THE PROTOTYPE

- JIT compilers
- The VM today
- The Idea
- The Prototype

THE PROTOTYPE

- Frej Drejhammar from SICS
- Tracing JIT compiler using LLVM in runtime and clang in compile time

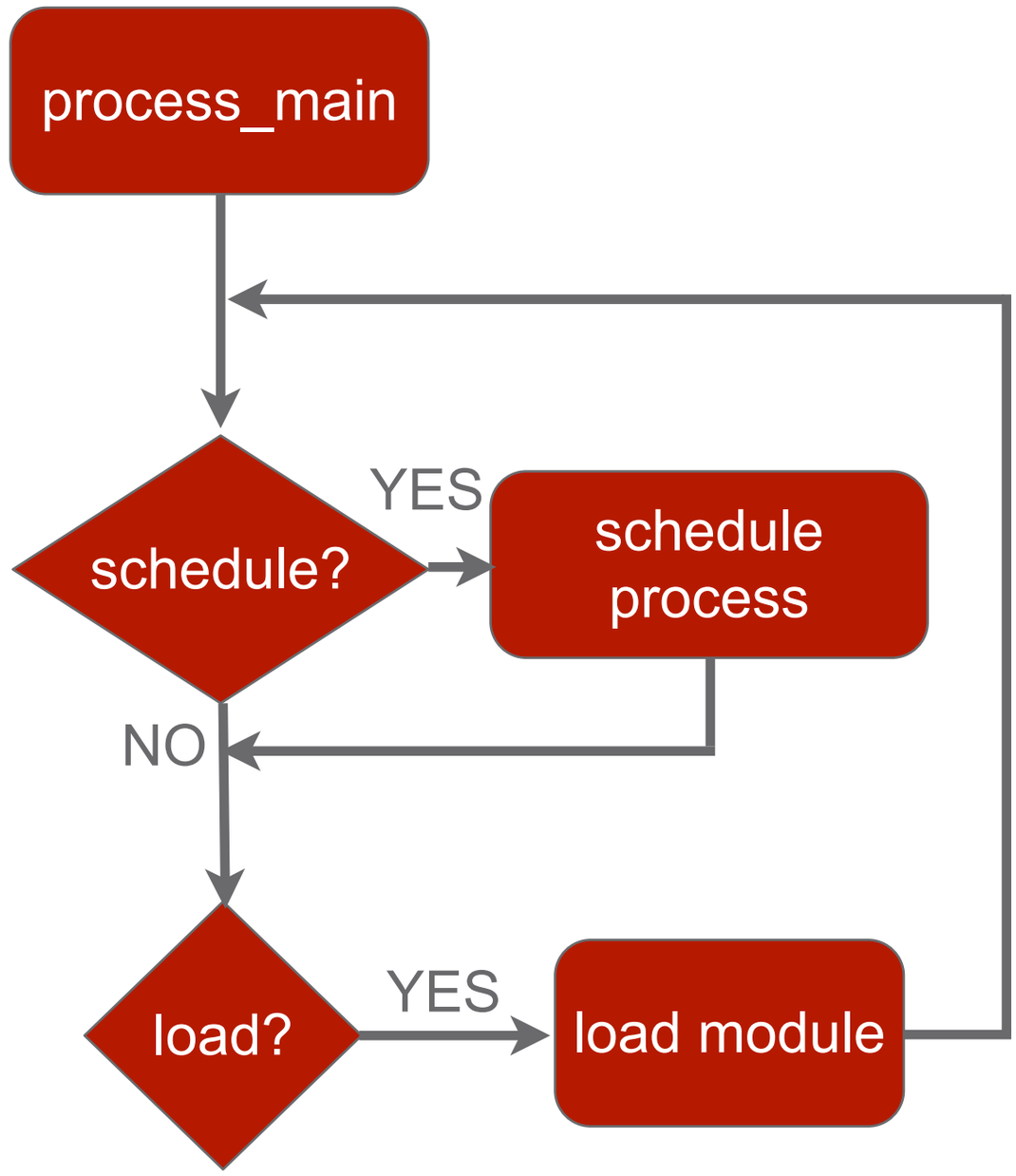
THE PROFILER

- New instruction inserted after function entry
  - Counts function calls
  - Flag Erlang process for tracing after N calls

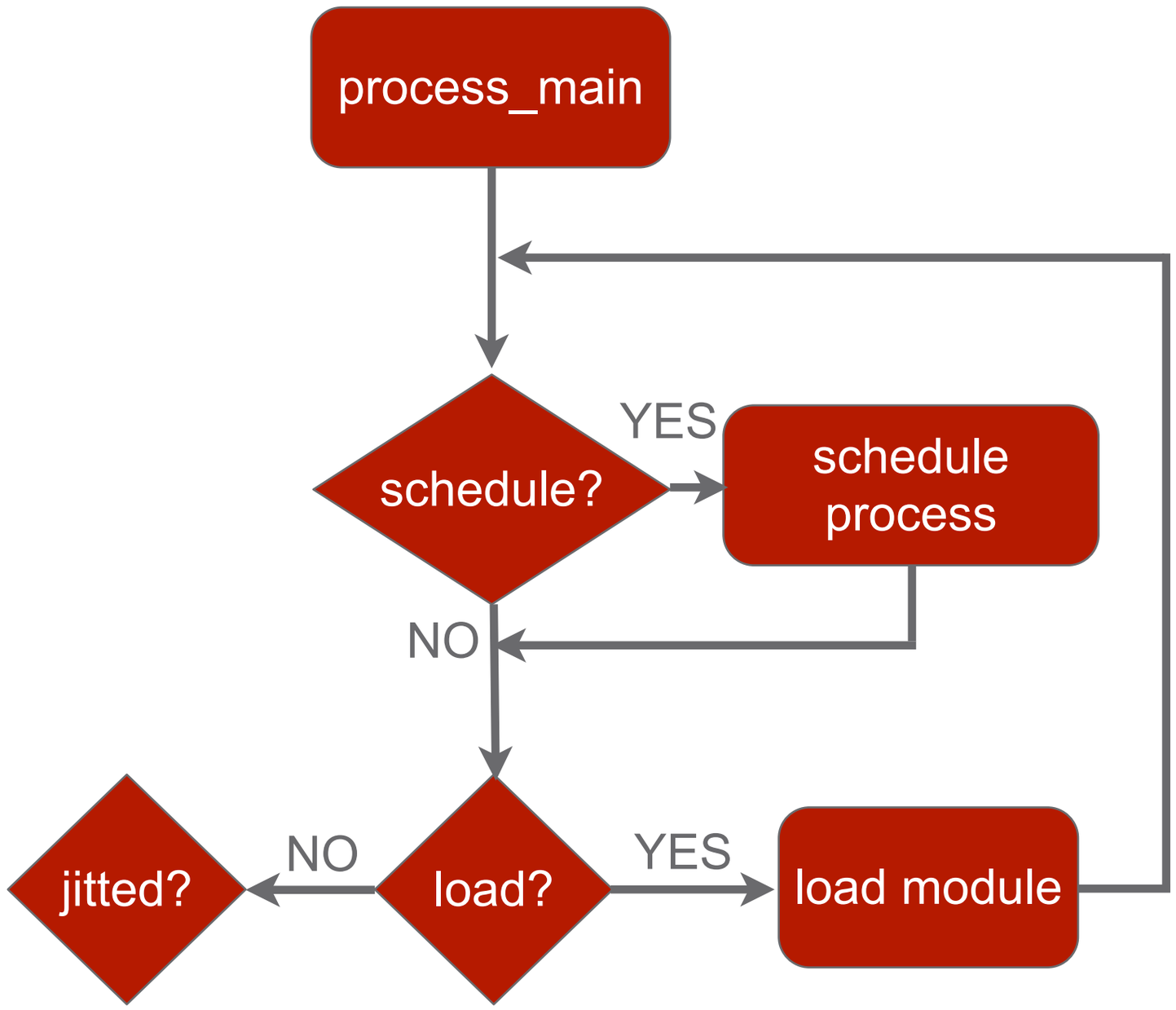
THE NEW VM

- libclang to analyze C code in beam_emu.c
  - C library interface to clang
- Extract all local variables to top of process_main
  - So JIT code can jump into plain VM loop
- beam_emu_plain.c and beam_emu_trace.c
  - Plain VM is normal interpreter with lifted locals and each basic block (BB) is given a unique label
  - Tracing VM adds trace recording instruction before BBs
- Dispatch table instead of code-threading
  - Switch opcodes when tracing vs normal execution

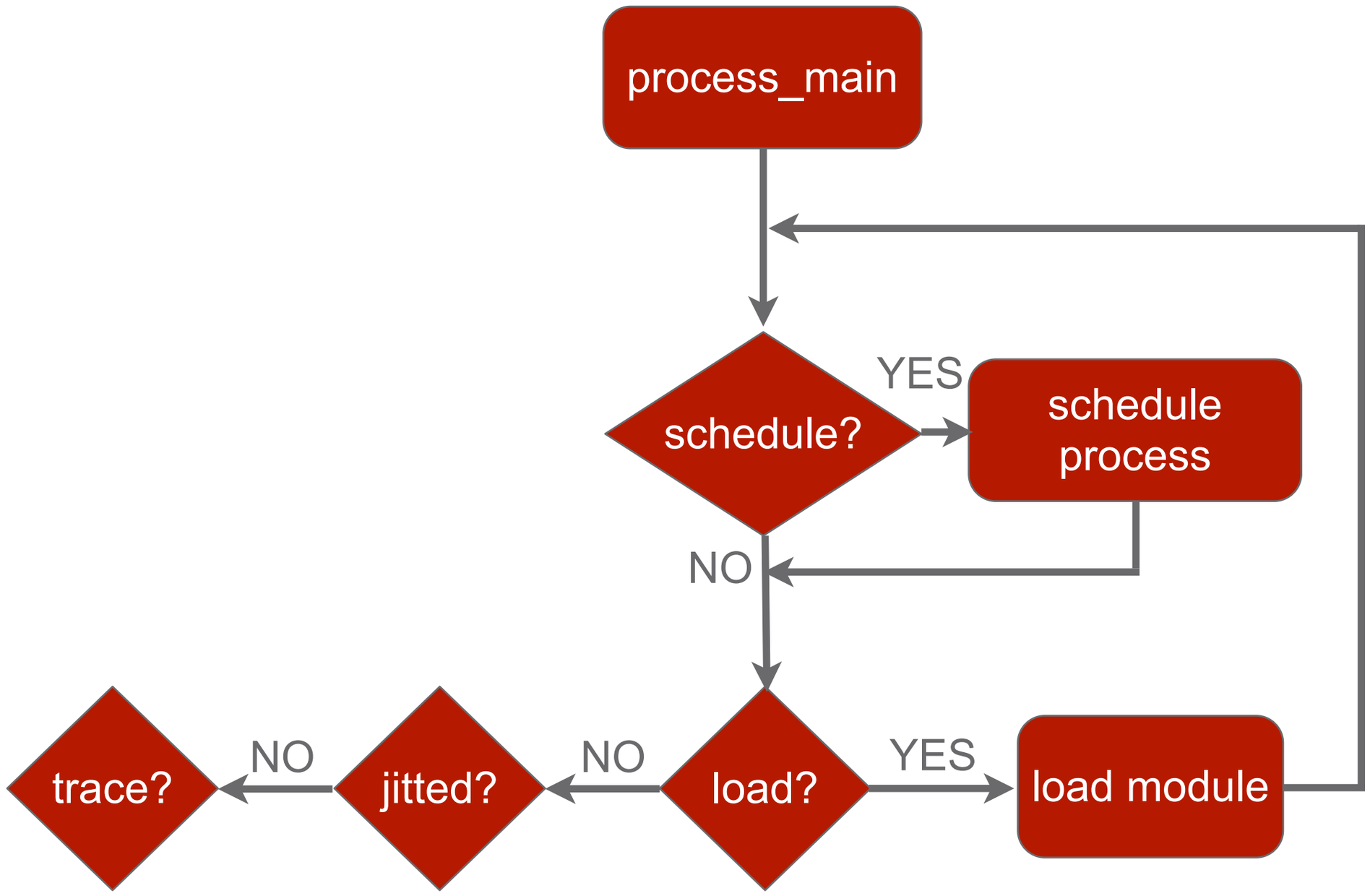
BEAM EMU LOOP



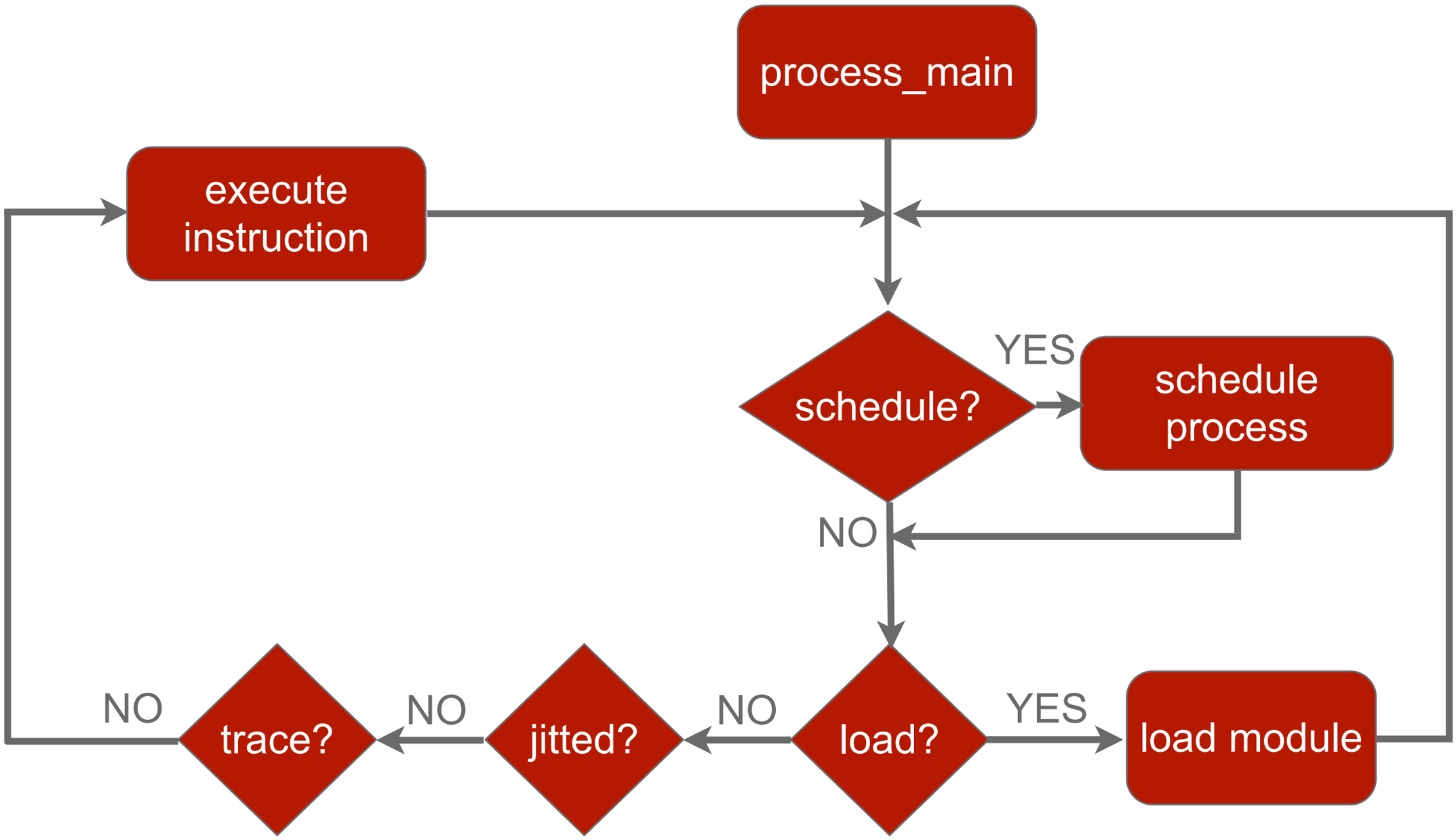
BEAM EMU LOOP



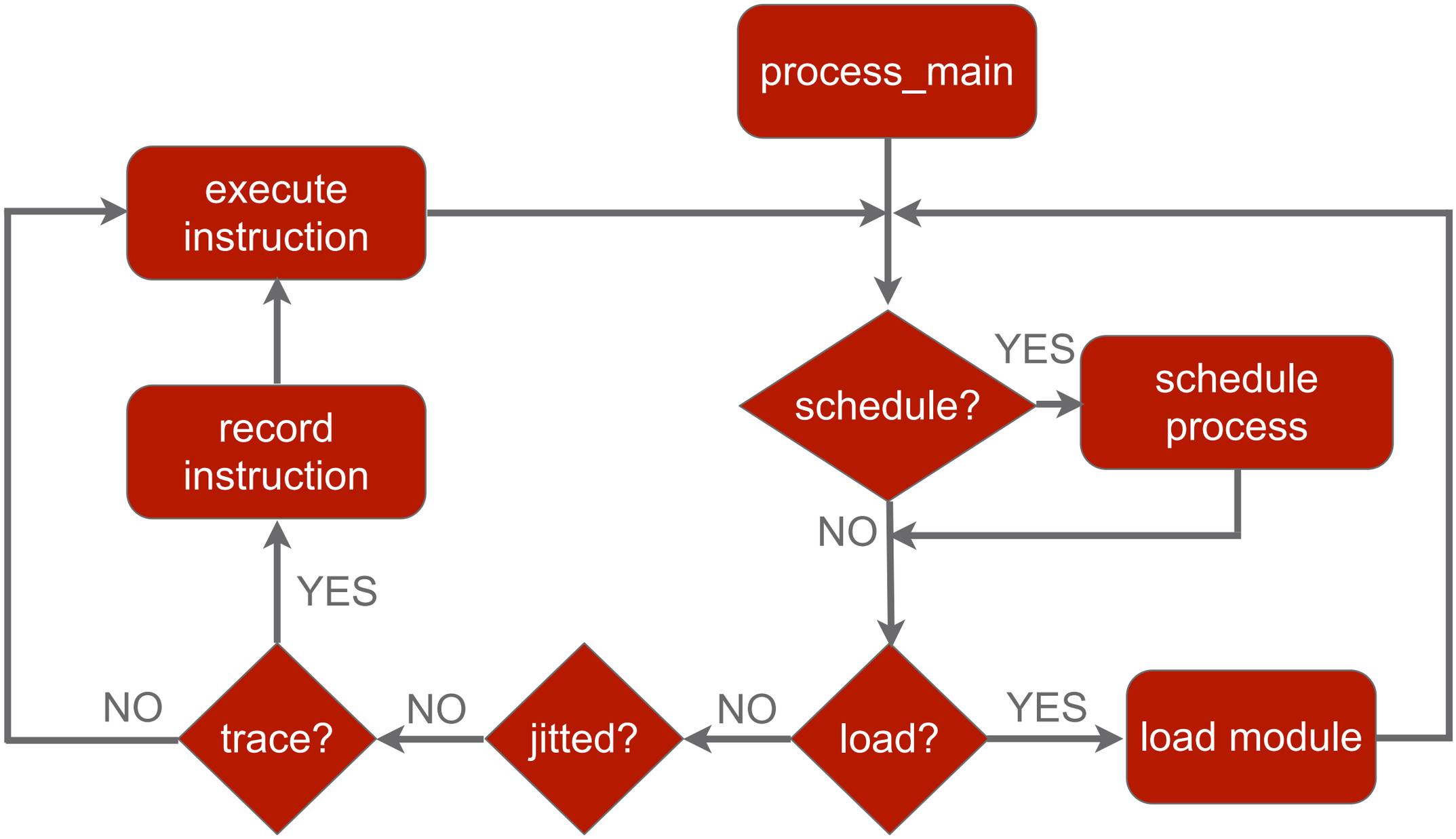
BEAM EMU LOOP



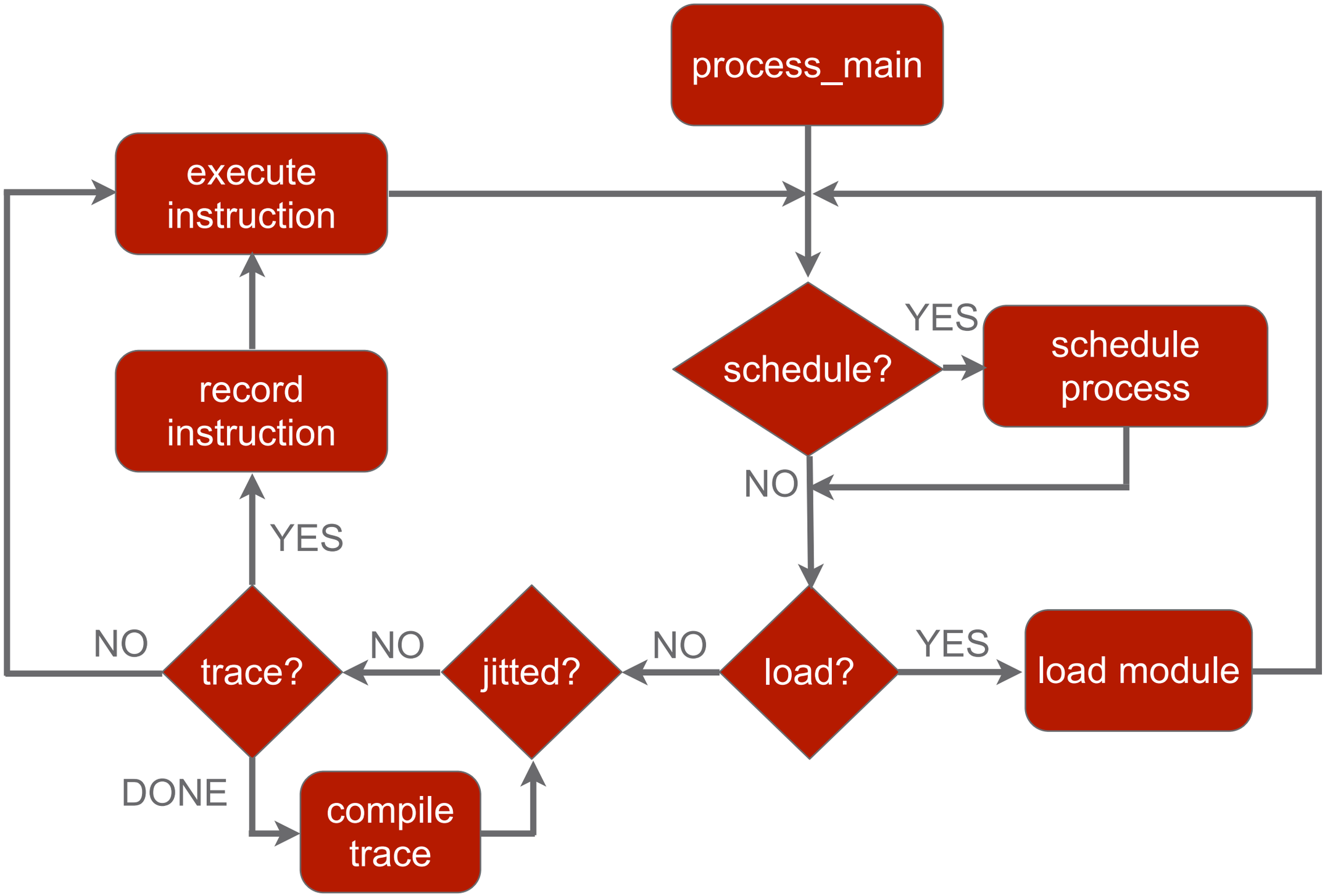
BEAM EMU LOOP



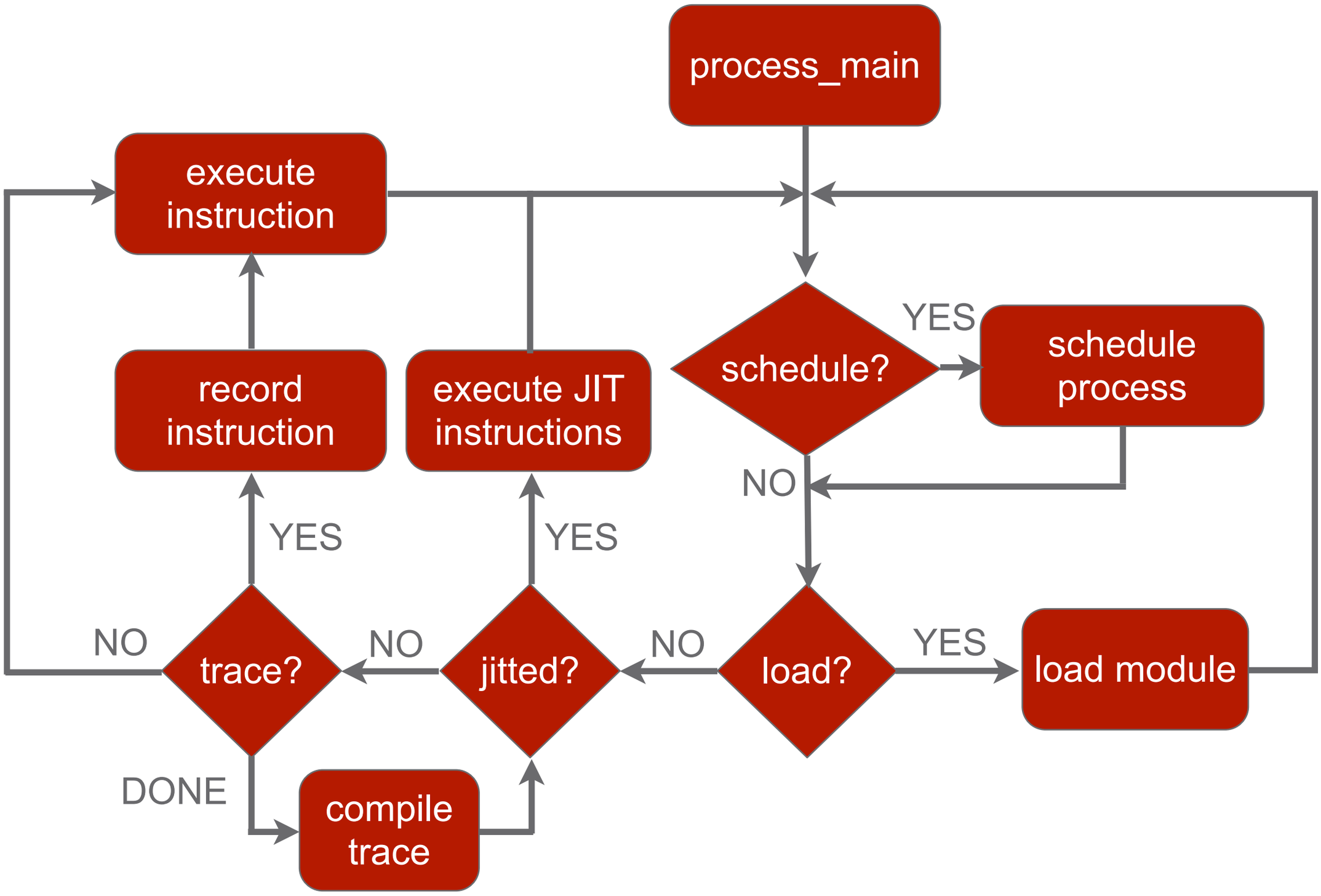
BEAM EMU LOOP



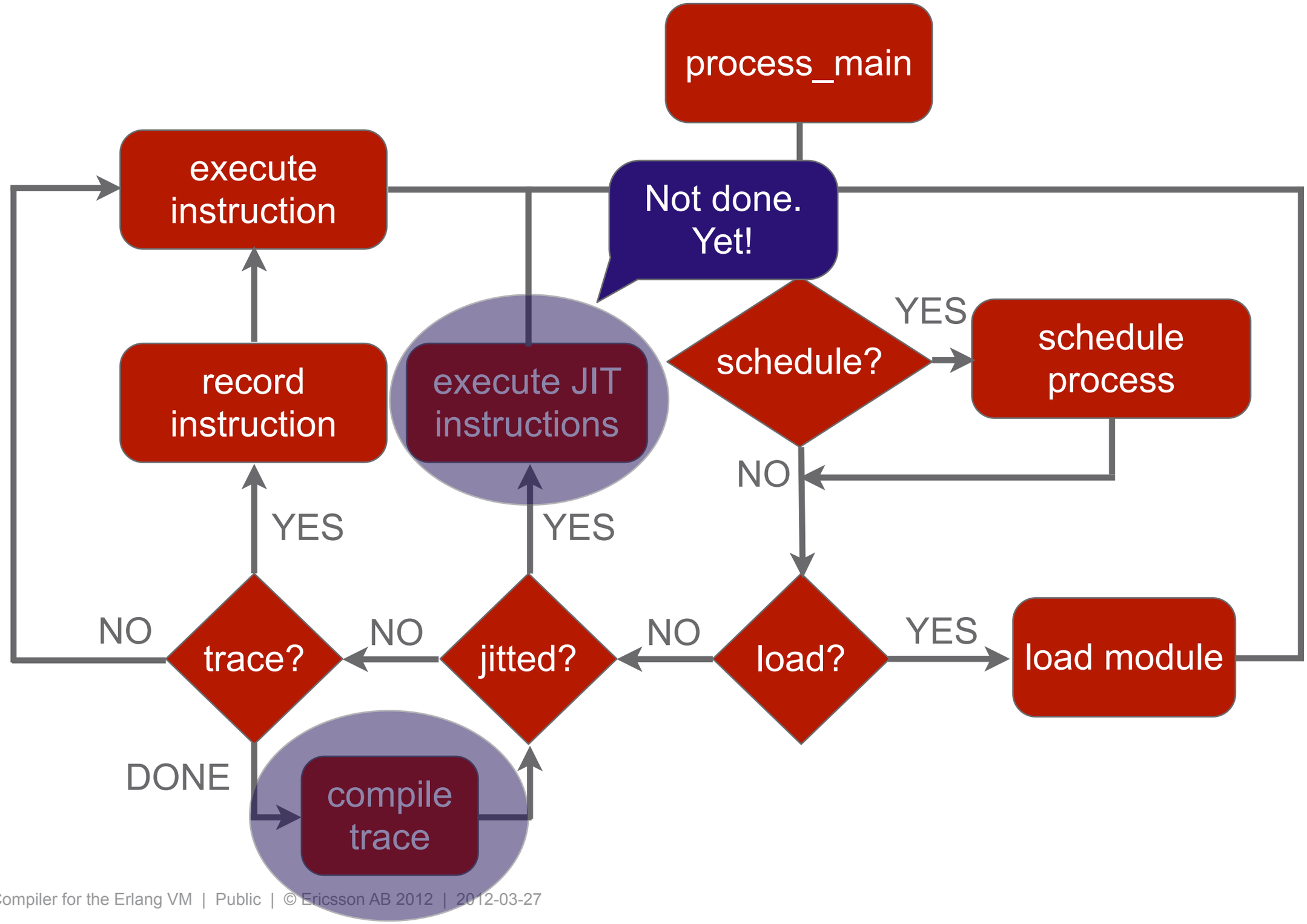
BEAM EMU LOOP



BEAM EMU LOOP



BEAM EMU LOOP



THE JITTED CODE

- Compile time
  - clang to compile BB into LLVM IR
- Run-time
  - LLVM combines each recorded BB into a string of BB

```
call_jit_for_anchor_1(r0, struct env *local_vars) {
    if !is_small(arg0)
        goto plain_vm_mixed_add
    if !is_small(arg1)
        goto plain_vm_mixed_add
    local_vars->Plus_i = untag(arg0) + untag(arg1)
    if !fits_in_small(local_vars->Plus_i)
        goto plain_vm_mixed_add
    r0 = tag(local_vars->Plus_tmp)

etc....
```

LIMITATIONS

- Only sequential interpreted code is faster
  - No automatic speedup in drivers or nifs
- Processes which communicate a lot will not see benefit

CONCLUSION

- Almost working JIT
- libclang and LLVM are great tools for productivity!
- Many things still to be done and to think about
  - Garbage collection of traces
  - receive loops
  - SMP support
  - Code loading
  - Optimizing traces
  - Tracing within NIFs/drivers
  - Re-write C code to erlang to be able to use JIT compiler?
  - Inter process JITing?

ERICSSON