# PinaVM and Tweto: Compilation and Optimization Techniques for SystemC

Matthieu Moy

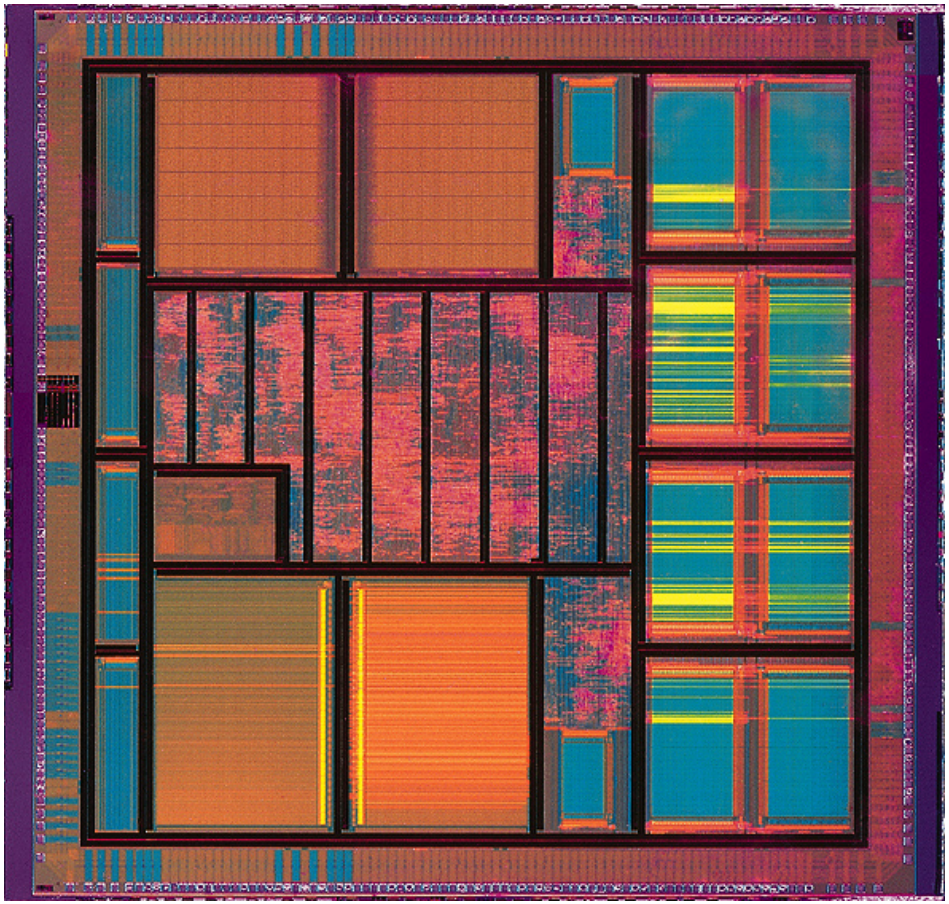
Verimag (Grenoble INP)
Grenoble
France

Journées compil', 04/2011

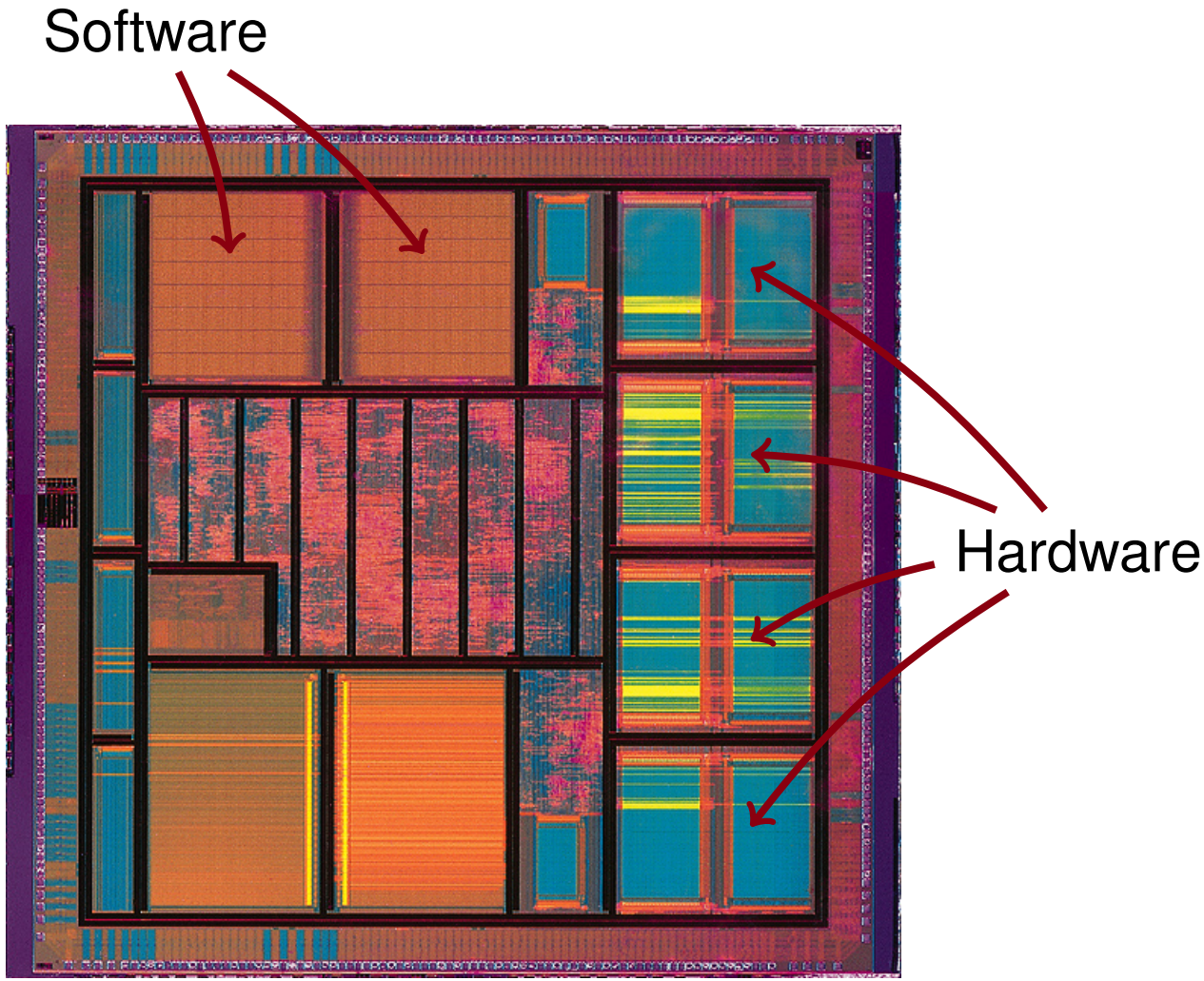
# Summary

# This section

# Modern Systems-on-a-Chip



# Modern Systems-on-a-Chip

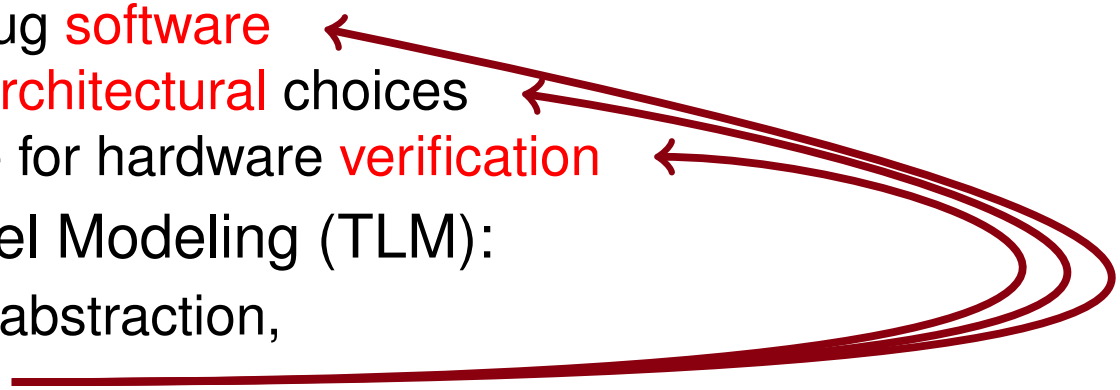


# SystemC and Transaction-Level Modeling

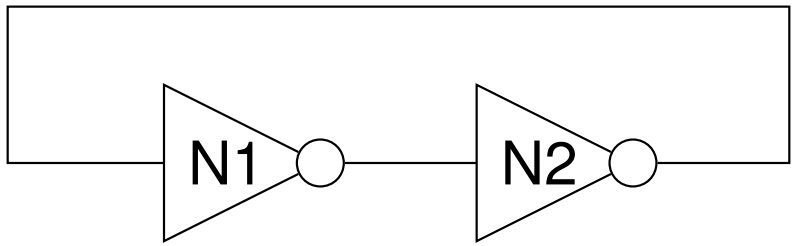
- (Fast) simulation essential in the design-flow
  - To write/debug software
  - To validate architectural choices
  - As reference for hardware verification

# SystemC and Transaction-Level Modeling

- (Fast) simulation essential in the design-flow
  - To write/debug software
  - To validate architectural choices
  - As reference for hardware verification
- Transaction-Level Modeling (TLM):
  - high level of abstraction,
  - suitable for
- SystemC:
  - Industry-standard for high-level modeling (TLM, ...) of Systems-on-a-Chip,
  - Library for C++ (compile with `g++ -lsystemc...`)



# SystemC: Simple Example




```
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;

    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

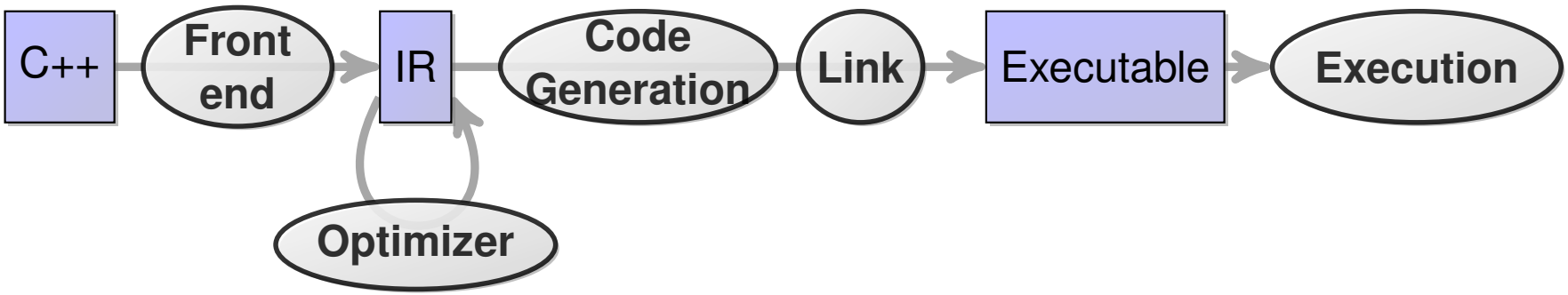
```
int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    // Instantiate modules ...
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // ... and bind them together
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS);
    return 0;
}
```

# Summary

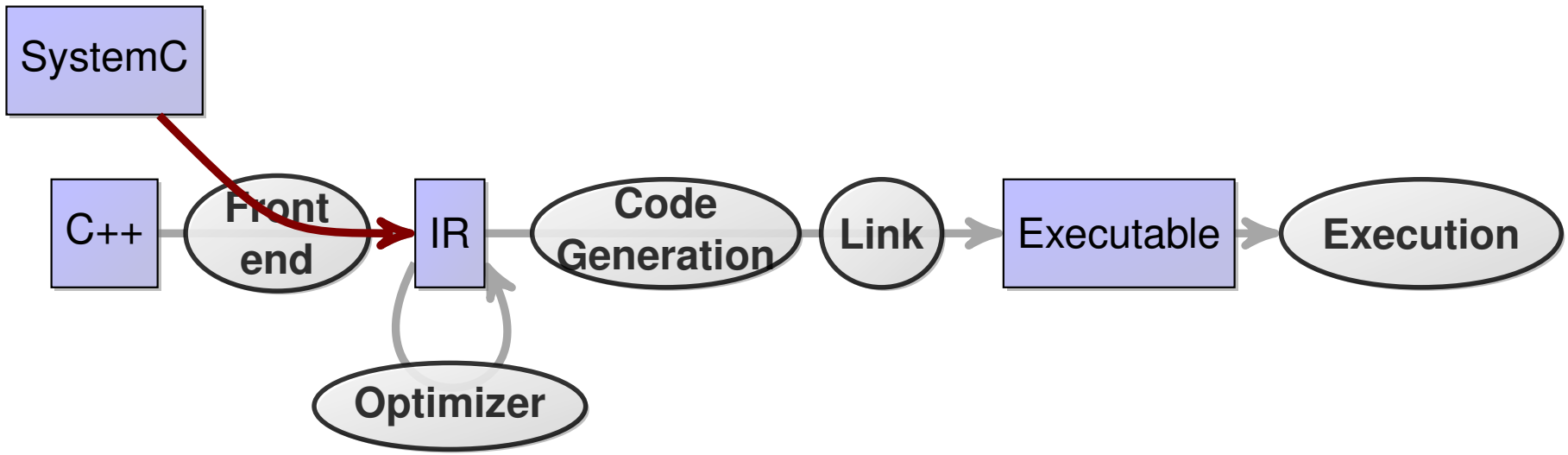
# This section

2 Overview of PinaVM and Tweto

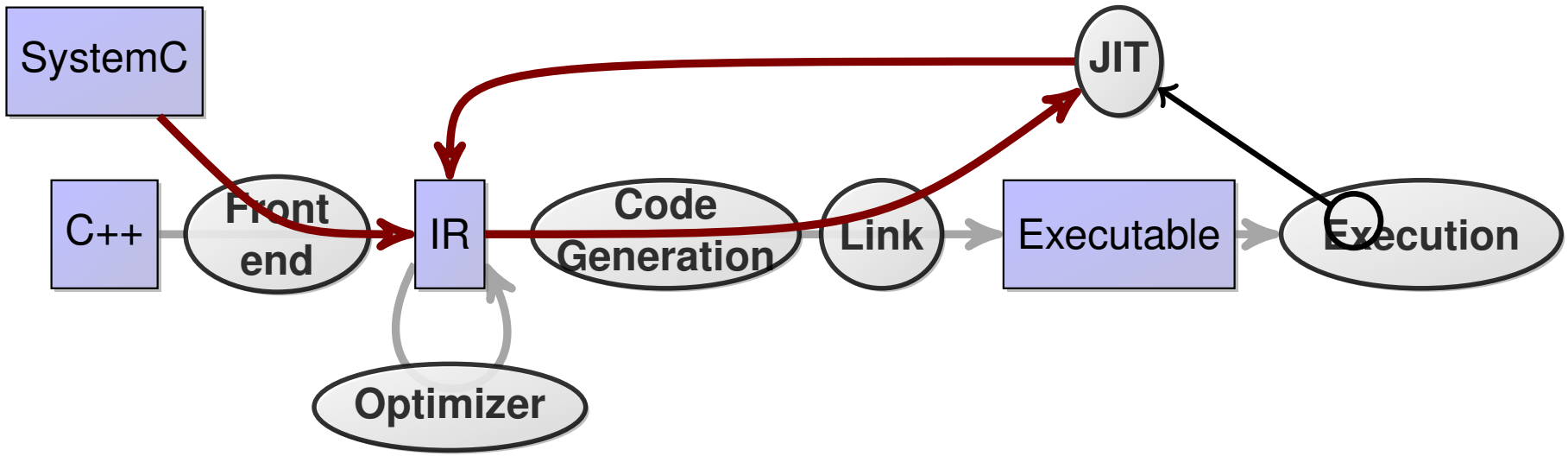
# This talk: PinaVM and Tweto



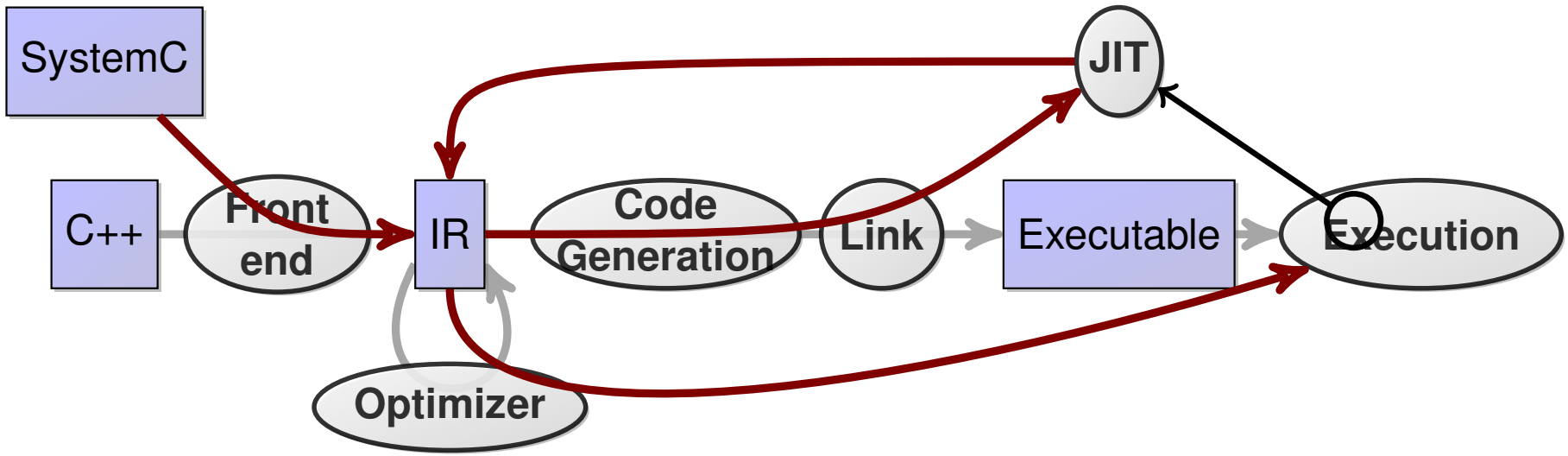
# This talk: PinaVM and Tweto



# This talk: PinaVM and Tweto

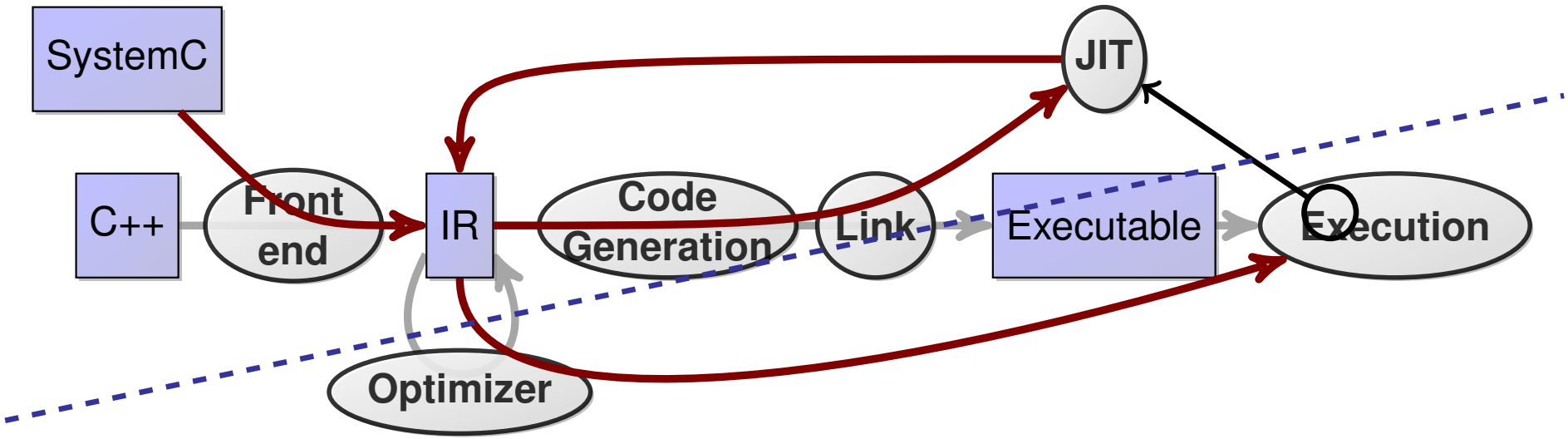


# This talk: PinaVM and Tweto



# This talk: PinaVM and Tweto

PinaVM: just like a compiler front-end,
but for the SystemC library

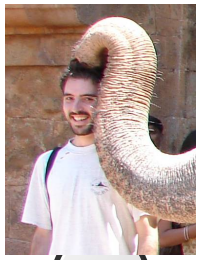



Tweto: use the information provided by PinaVM to optimize the program better than C++ compilers

# This talk: PinaVM and Tweto

PinaVM: just like a compiler front-end, but for the SystemC library

Matthieu Moy

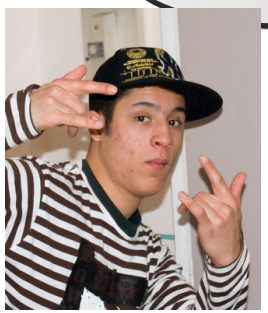


Claire Maïza

Kevin Marquet




Si-Mohamed Lamraoui



Claude Helmstetter

Tweto: use the information provided by PinaVM to optimize the program better than C++ compilers

# Summary

# This section

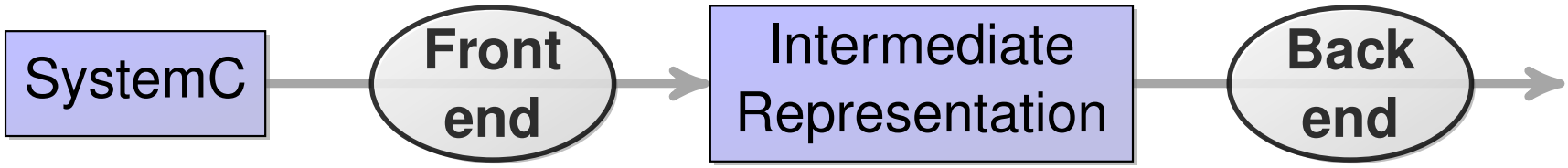
3 PinaVM: a SystemC Front-End

- SystemC Front-Ends
- The Beginning ...
- **P**inaVM **I**s **N**ot **A** **V**irtual **M**achine
- PinaVM: Summary

# This section

# SystemC Front-End

- In this talk: Front-end = “Compiler front-end” (AKA “Parser”)




Intermediate Representation = Architecture + Behavior

# When you *don't* need a front-end

- Main application of SystemC: simulation
  - Just need a C++ compiler + the library
- Testing, run-time verification, monitoring...
  - (Small) modifications of the SystemC library
- IDE integration, Debugging ...
  - Plain C++ front-ends can do most of the job.

# When you *don't* need a front-end

- Main application of SystemC: simulation
  - Just need a C++ compiler + the library
- Testing, run-time verification, monitoring...
  - (Small) modifications of the SystemC library
- IDE integration, Debugging ...
  - Plain C++ front-ends can do most of the job.

No reference front-end available on
`http://www.systemc.org/`

# When you *really* need a front-end

- Symbolic formal verification, high-level synthesis
- Visualization
- Introspection

- Advanced debugging features (architecture $\rightarrow$ source code, . . . )
- SystemC-specific compiler optimizations

# When you *really* need a front-end

- Symbolic formal verification, high-level synthesis
  - Need to extract almost everything about the platform
- Visualization
- Introspection
- Advanced debugging features (architecture $\rightarrow$ source code, . . . )
- SystemC-specific compiler optimizations

# When you *really* need a front-end

- Symbolic formal verification, high-level synthesis
  - Need to extract almost everything about the platform
- Visualization
  - Need to extract the architecture. Behavior is less important
- Introspection

- Advanced debugging features (architecture $\rightarrow$ source code, . . . )

- SystemC-specific compiler optimizations

# When you *really* need a front-end

- Symbolic formal verification, high-level synthesis
  - Need to extract almost everything about the platform
- Visualization
  - Need to extract the architecture. Behavior is less important
- Introspection
  - Information about the module (architecture) + possibly local variables (behavior)
- Advanced debugging features (architecture $\rightarrow$ source code, . . . )

- SystemC-specific compiler optimizations

# When you *really* need a front-end

- Symbolic formal verification, high-level synthesis
  - Need to extract almost everything about the platform
- Visualization
  - Need to extract the architecture. Behavior is less important
- Introspection
  - Information about the module (architecture) + possibly local variables (behavior)
- Advanced debugging features (architecture $\rightarrow$ source code, . . . )
  - Need architecture and behavior
- SystemC-specific compiler optimizations

# When you *really* need a front-end

- Symbolic formal verification, high-level synthesis
  - Need to extract almost everything about the platform
- Visualization
  - Need to extract the architecture. Behavior is less important
- Introspection
  - Information about the module (architecture) + possibly local variables (behavior)
- Advanced debugging features (architecture $\rightarrow$ source code, . . . )
  - Need architecture and behavior
- SystemC-specific compiler optimizations
  - Can use architecture information to optimize behavior

# Challenges and Solutions with SystemC Front-Ends

1. C++ is complex (e.g. `clang` ≈ 200,000 LOC)
2. Architecture built at runtime, with C++ code

```
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;
    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

```
int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // Binding
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS); return 0;
}
```

# Challenges and Solutions with SystemC Front-Ends

1. C++ is complex (e.g. `clang` ≈ 200,000 LOC)
   ⇝ Write a C++ front-end or reuse one (g++, clang, edg, . . . )
2. Architecture built at runtime, with C++ code
   ⇝ Analyze elaboration phase or execute it

```
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;
    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

```
int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // Binding
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS); return 0;
}
```

# Challenges and Solutions with SystemC Front-Ends

1. C++ is complex (e.g. `clang` ≈ 200,000 LOC)
   ⇝ Write a C++ front-end or reuse one (g++, clang, edg, . . . )
2. Architecture built at runtime, with C++ code
   ⇝ Analyze elaboration phase or execute it

```
SC_MODULE(no _gate) {
    sc_in<bo l> in;
    sc_out<b ol> out;
    void co ute (void) {
        //  havior
        in.read();
        );
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

Static Approaches

```
int sc_main(in  argc, char **argv) {
    // Elaborat  phase (Architecture)
    not_gate n1("
    not_g
    sc_signal<bool> s1, s2;
    // Binding
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS); return 0;
}
```

Dynamic Approaches

# Dealing with the architecture

## When it becomes tricky...

```
int sc_main(int argc, char **argv) {
    int n = atoi(argv[1]);
    int m = atoi(argv[2]);
    Node array[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            array[i][j]
                = new Node(...);
            ...
        }
    }

    sc_start(100, SC_NS);
    return 0;
}
```

# Dealing with the architecture

## When it becomes tricky...

- Static approach: cannot deal with such code
- Dynamic approach: can extract the architecture for individual instances of the system

```
int sc_main(int argc, char **argv) {
    int n = atoi(argv[1]);
    int m = atoi(argv[2]);
    Node array[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            array[i][j]
                = new Node(...);
            ...
        }
    }

    sc_start(100, SC_NS);
    return 0;
}
```

# Dealing with the architecture

## When it becomes *very* tricky...

```
void compute(void) {
    for (int i = 0; i < n; i++) {
        ports[i].write(true);
    }
    ...
}
```

# Dealing with the architecture

## When it becomes *very* tricky...

- One can unroll the loop to let `i` become constant,
- Undecidable in the general case.

```
void compute(void) {
    for (int i = 0; i < n; i++) {
        ports[i].write(true);
    }
    ...
}
```

# Existing SystemC front-ends

## An attempt at a classification

| | Static | Dynamic |
|---|---|---|
| Home-made parser | KaSCPar, sc2v, ParSyC, Scoot, SystemPerl. . . | |
| Existing parser | SystemCXML | DATE09, Pinapa, PinaVM |

- Hard to classify: Quiny (purely dynamic approach)
- Commercial tools (closed, not detailed here): Synopsys, Semantic Design, NC-SystemC (Cadence)

## Self-advertisement

*A Theoretical and Experimental Review of SystemC Front-ends*, Marquet et al. FDL 2010

# This section

# Before it started: Pinapa [Moy et al, EMSOFT 05]

AKA “my Ph.D’s front-end”

- Pinapa’s principle:
  - Use GCC’s C++ front-end
  - Compile, dynamically load and execute the elaboration (`sc_main`)
- Pinapa’s drawbacks:
  - Uses GCC’s internals (hard to port to newer versions)
  - Hard to install
  - No separate compilation
  - Based on complex Abstract Syntax Tree (AST)
    (e.g. one construct for `for`, one for `while` and one for `do ... while`)
  - Ad-hoc match of SystemC constructs in AST

# Static Single Assignment

- Non-SSA program

```
x = 42;
x = x + 1;
y = x;
```

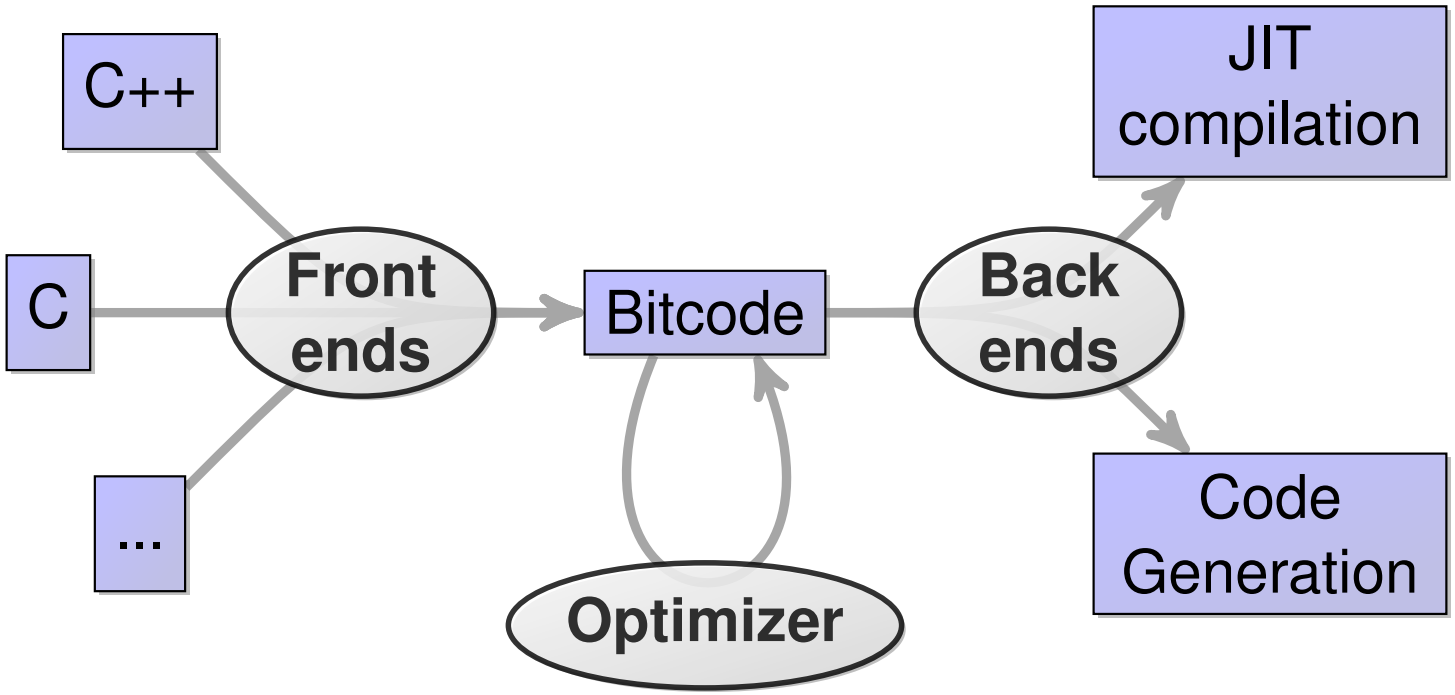
- SSA program

```
x1 = 42;
x2 = x1 + 1;
y = x2;
```

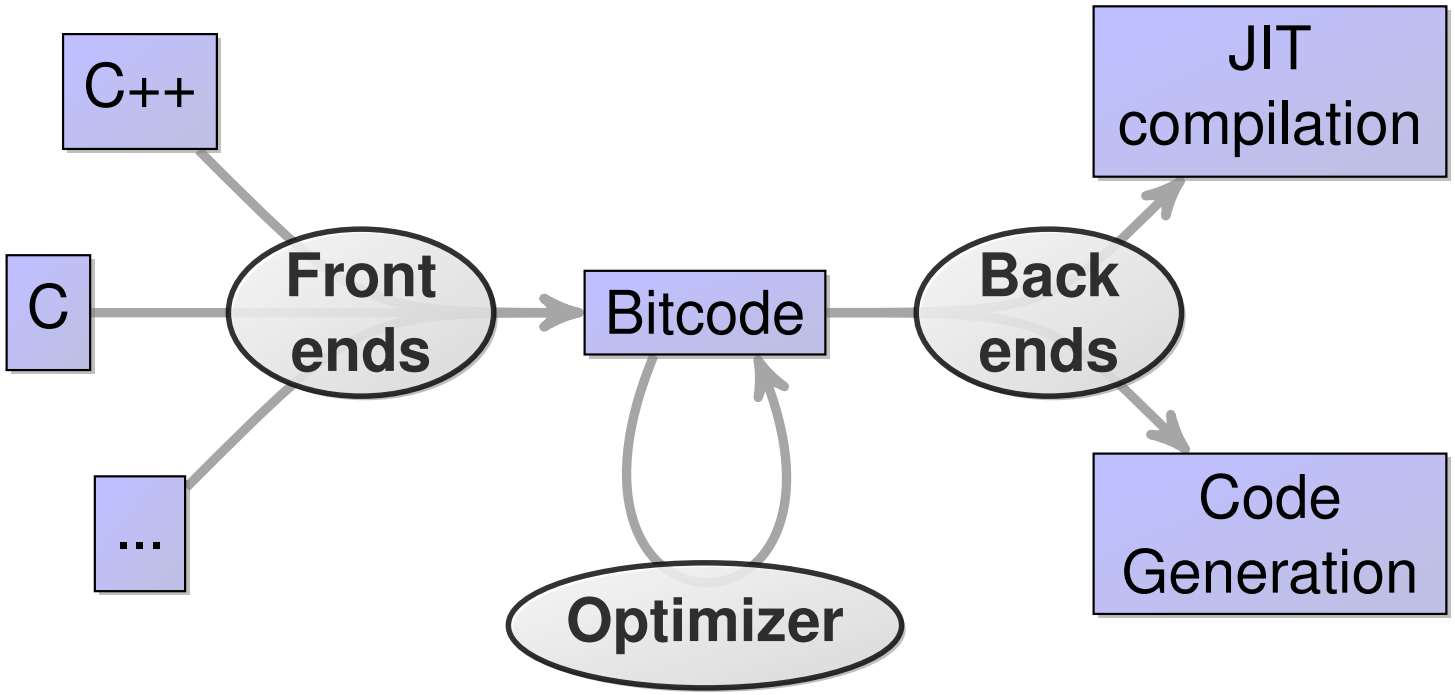
- SSA form widely used by modern compilers...
- ...and by some formal verification tools[1]
- Candidates C++ front-end providing SSA form:
  - GCC $\geq$ 4.0
  - LLVM

[1] *Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form*. Besnard et al. AVOCS 2009

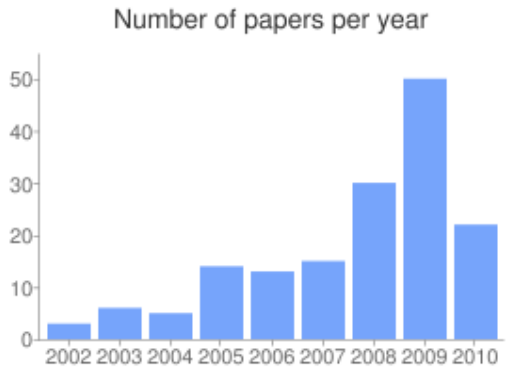
# LLVM: Low Level Virtual Machine



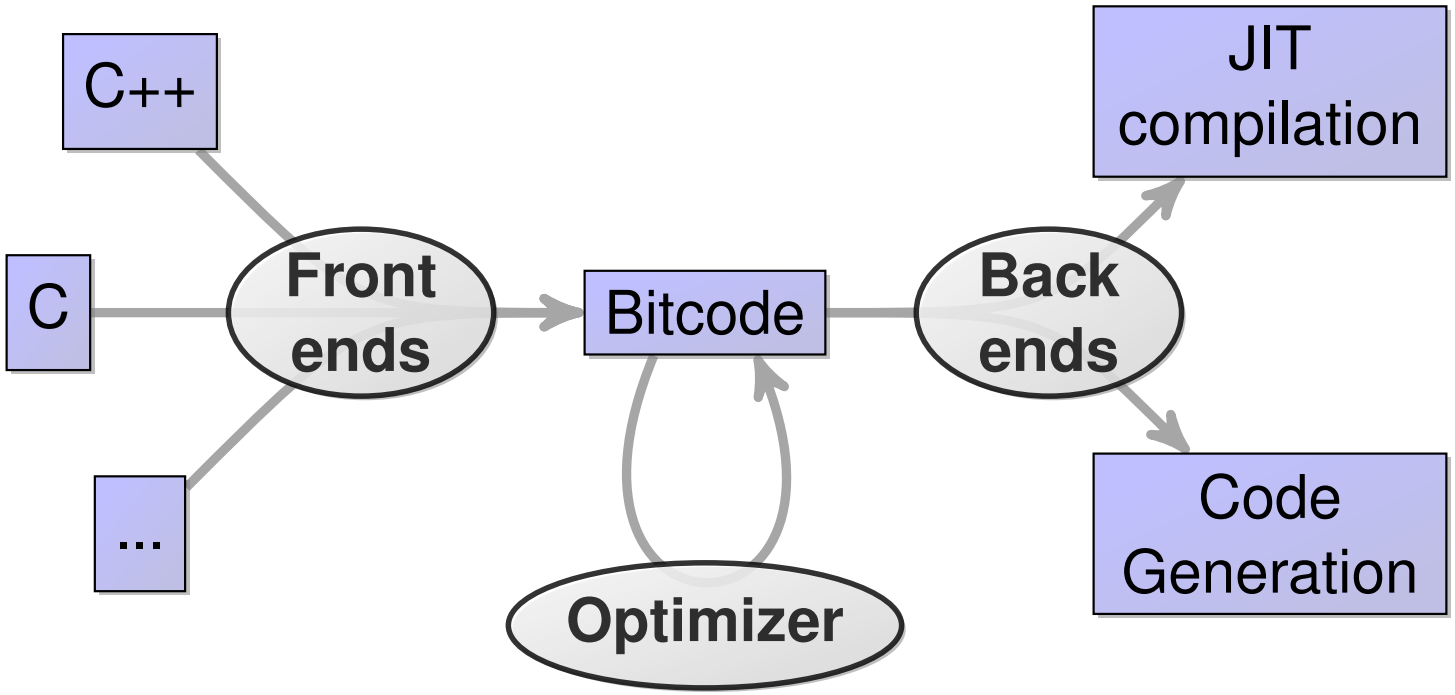
# LLVM: Low Level Virtual Machine



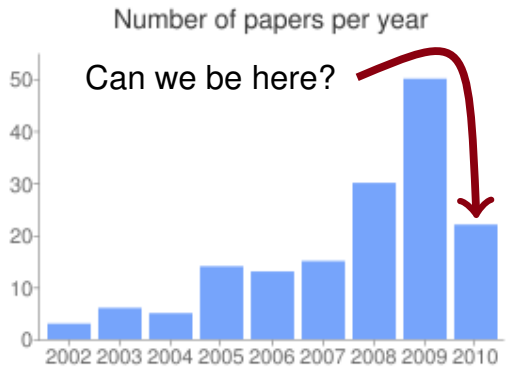

- Clean API
- Clean SSA intermediate representation
- Many tools available



# LLVM: Low Level Virtual Machine

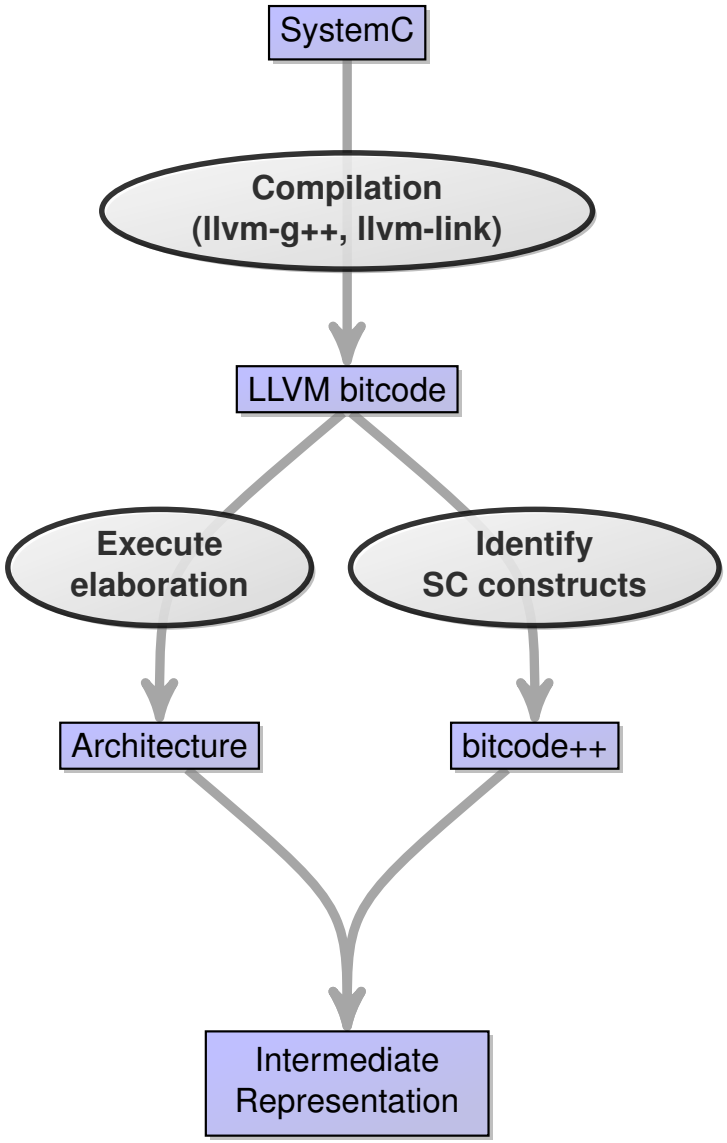



- Clean API
- Clean SSA intermediate representation
- Many tools available

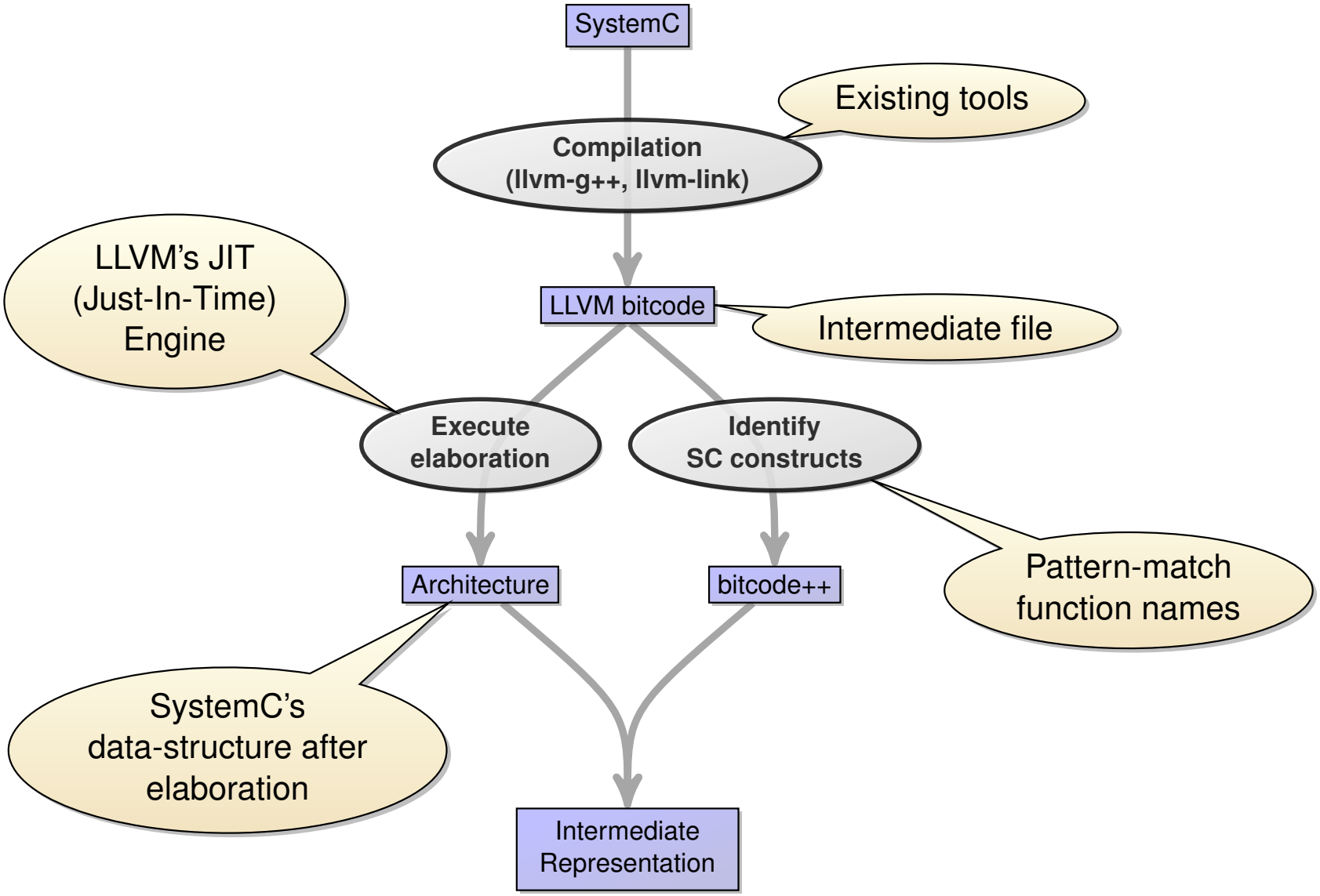


# This section

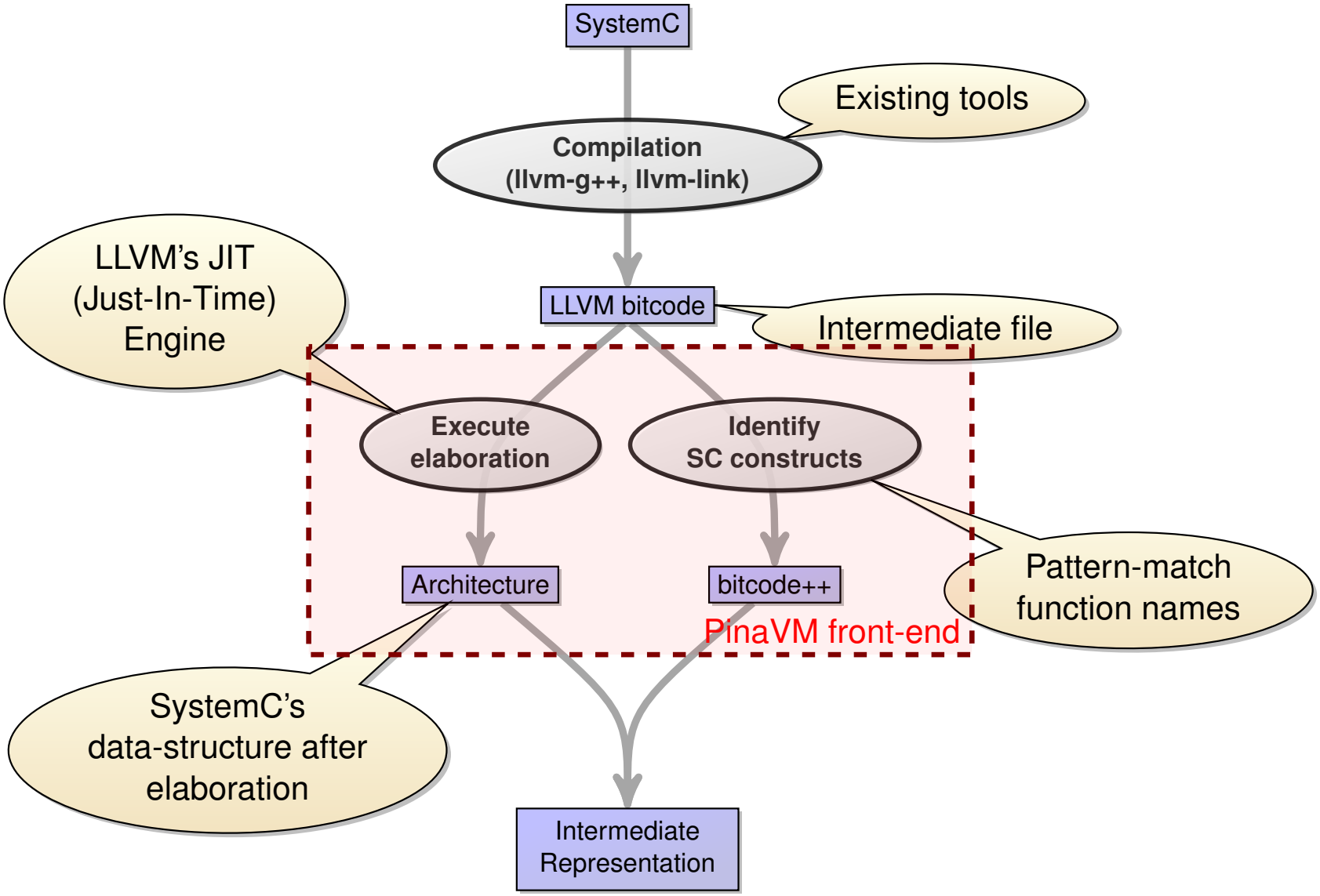
# PinaVM: Architecture



# PinaVM: Architecture



# PinaVM: Architecture



# SystemC constructs in LLVM's bitcode

- SystemC (C++) source code:

```
void compute() {
    out.write(true);
}
```

- Bitcode after compilation with `llvm-g++`:

```
define linkonce_odr void @_ZN6Source7computeEv(%struct.Source* %this) {
entry:
  %0 = alloca i8
  %"alloca point" = bitcast i32 0 to i32
  store i8 1, i8* %0, align 1
  %1 = getelementptr inbounds %struct.Source* %this, i32 0, i32 1
  %2 = getelementptr inbounds %"struct.sc_core::sc_out<bool>"* %1, i32 0, i32 0
  call void @_ZN7sc_core8sc_inoutIbE5writeERKb
                                  (%"struct.sc_core::sc_inout<bool>"* %2, i8* %0)
  br label %return

return:
  ret void
}
```

# SystemC constructs in LLVM's bitcode

- SystemC (C++) source code:

```
void compute() {
    out.write(true);
}
```

- Bitcode after compilation with `llvm-g++`:

```
define linkonce_odr void @_ZN6Source7computeEv(%struct.Source* %this) {
entry:
  %0 = alloca i8
  %"alloca point" = bitcast i32 0 to i32
  store i8 1, i8* %0, align 1
  %1 = getelementptr inbounds %struct.Source* %this, i32 0, i32 1
  %2 = getelementptr inbounds %"struct.sc_core::sc_out<bool>"* %1, i32 0, i32 0
  call void @_ZN7sc_core8sc_inoutIbE5writeERKb
                                  (%"struct.sc_core::sc_inout<bool>"* %2, i8* %0)
  br label %return

return:
  ret void
}
```

# SystemC constructs in LLVM's bitcode

- Simplified Bitcode:

```
define void Source::compute(%this) {
  ; "out.write(true)" compiled into:
    ; piece of code computing %data = true = 1

    ; piece of code computing %port as
    ; a function of %this

    call sc_core::sc_inout::write(%port, %data)
    ret void
}
```

# SystemC constructs in LLVM's bitcode

- Simplified Bitcode:

```
define void Source::compute(%this) {
  ; "out.write(true)" compiled into:
    ; piece of code computing %data = true = 1

    ; piece of code computing %port as
    ; a function of %this

    call sc_core::sc_inout::write(%port, %data)
    ret void
}
```

- Computation of %port and %data (argument of SystemC primitive)
  - Unknown statically (depend on `this`)
  - Computable for each module once we know `this`!
- What PinaVM does:
  - Extract (slice) pieces of code computing %data and %port
  - JIT-compile and execute them after fixing `%this`

# Back to the example

- Computation of “port” in `port.write(true)`:

```
define linkonce_odr void
          @_ZN6Source7computeEv(%struct.Source* %this) {
entry:
  %0 = alloca i8
  %"alloca point" = bitcast i32 0 to i32
  store i8 1, i8* %0, align 1
  %1 = getelementptr inbounds
          %struct.Source* %this, i32 0, i32 1
  %2 = getelementptr inbounds
          %"struct.sc_core::sc_out<bool>"* %1, i32 0, i32 0

  call void @_ZN7sc_core8sc_inoutIbE5writeERKb
         (%"struct.sc_core::sc_inout<bool>"* %2, i8* %0)
  br label %return
return:
  ret void
}
```

# Back to the example

- Computation of “port” in `port.write(true)`:

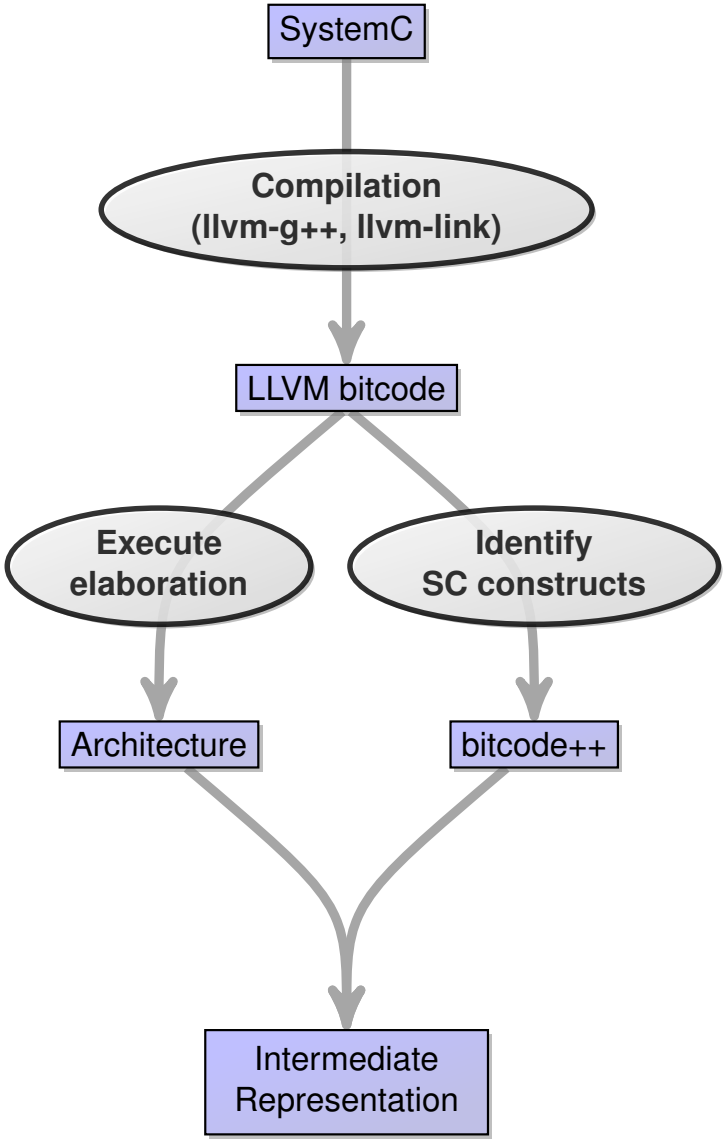
```
define linkonce_odr void
          @_ZN6Source7computeEv(%struct.Source* %this) {
entry:
  %0 = alloca i8
  %"alloca point" = bitcast i32 0 to i32
  store i8 1, i8* %0, align 1
  %1 = getelementptr inbounds
          %struct.Source* %this, i32 0, i32 1
  %2 = getelementptr inbounds
          %"struct.sc_core::sc_out<bool>"* %1, i32 0, i32 0

  call void @_ZN7sc_core8sc_inoutIbE5writeERKb
         (%"struct.sc_core::sc_inout<bool>"* %2, i8* %0)
  br label %return
return:
  ret void
}
```

# Back to the example

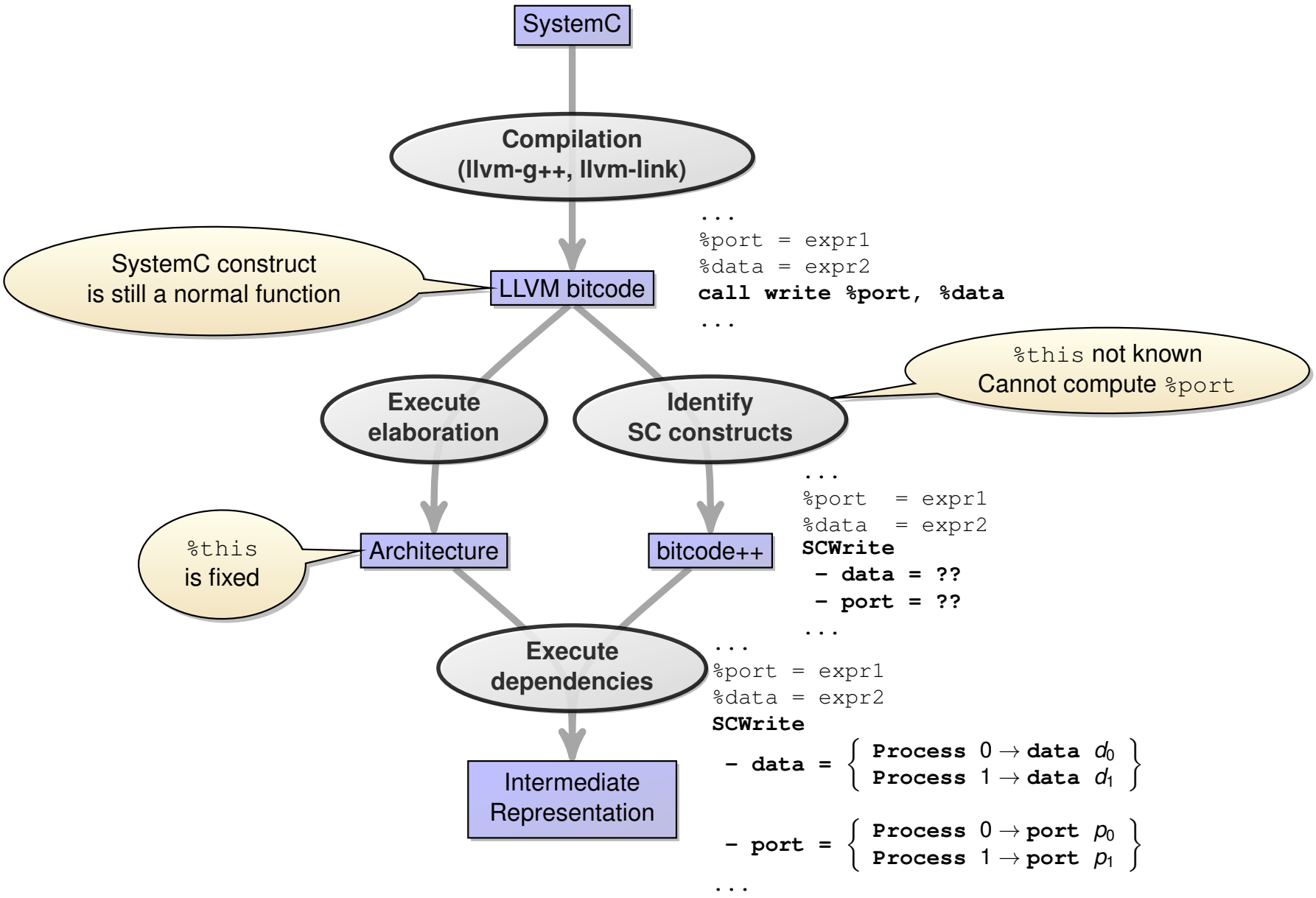
- Computation of “port” in `port.write(true)`:

```
define private %"struct.sc_core::sc_inout<bool>"*
        function_to_jit(%struct.Source* %this) {



  %1 = getelementptr inbounds
          %struct.Source* %this, i32 0, i32 1
  %2 = getelementptr inbounds
          %"struct.sc_core::sc_out<bool>"* %1, i32 0, i32 0

  ret %"struct.sc_core::sc_inout<bool>"* %2
}
```

# PinaVM: Enriching the bitcode?



# PinaVM: Enriching the bitcode?



# This section

# Summary

- PinaVM relies on executability (JIT Compiler) for:
  - Execution of elaboration phase ($\approx$ like Pinapa)
  - Execution of sliced pieces of code
- Using a virtual machine to write a SystemC front-end is a *really* good idea!
- Could have benefited from some higher-level constructs in bytecode (builtin object and method calls?)

# PinaVM

- Open Source:
  `http://gitorious.org/pinavm/pages/Home`
- Still a prototype, but very few fundamental limitations
- $\approx$ 3000 lines of C++ code on top of LLVM
- Experimental back-ends for
  - Model-checking (using SPIN)
  - Execution (Tweto)
- *PinaVM: a SystemC front-end based on an executable intermediate representation*. Marquet et al. EMSOFT 2010.

# Summary

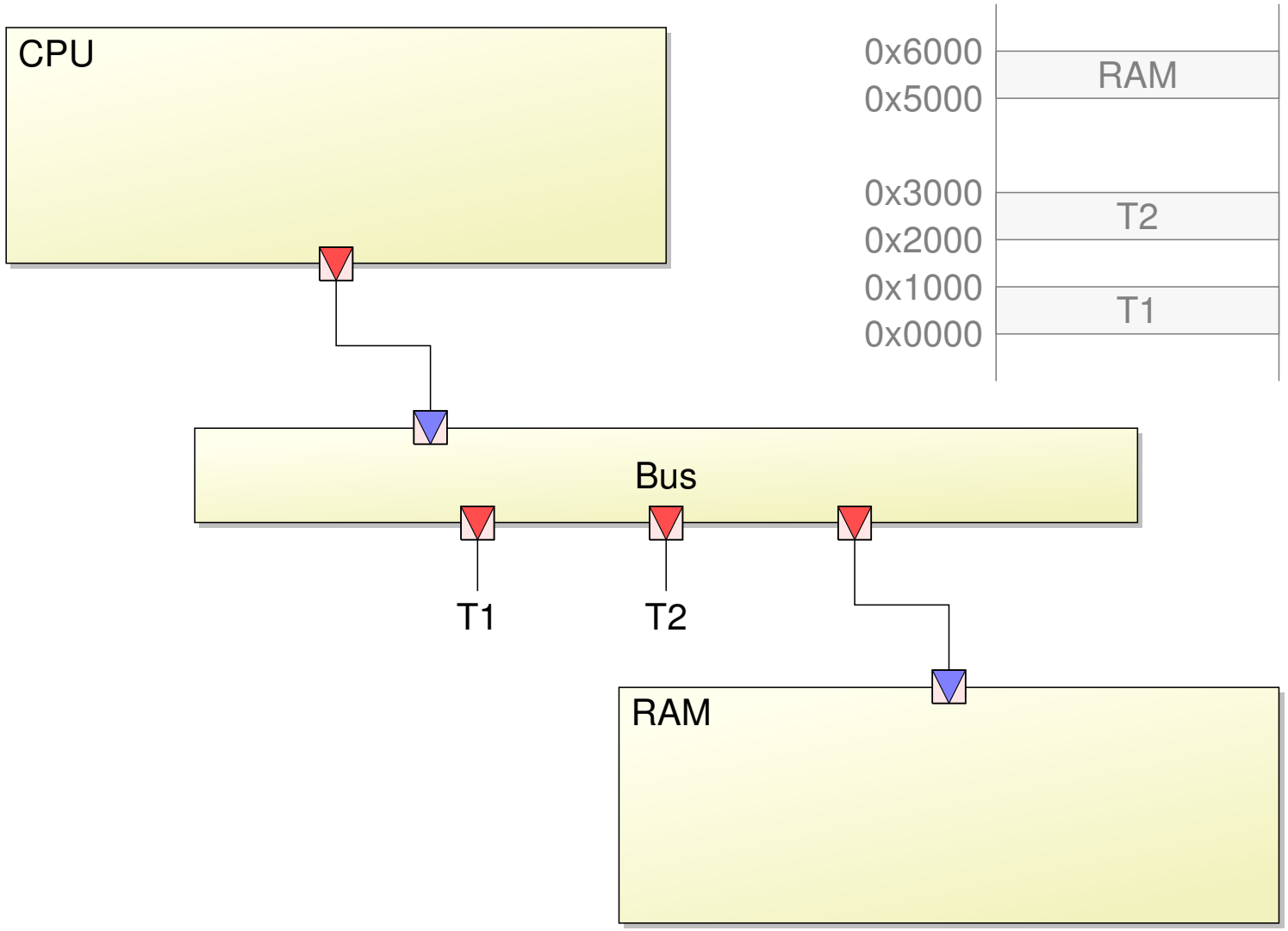
# This section

4 Tweto: TLM With Elaboration-Time Optimizations
- Limitations of Plain C++ Compilers and How to do Better
- Exploiting Constant Data
- Dealing with Addresses: Protocol-Aware Optimizations

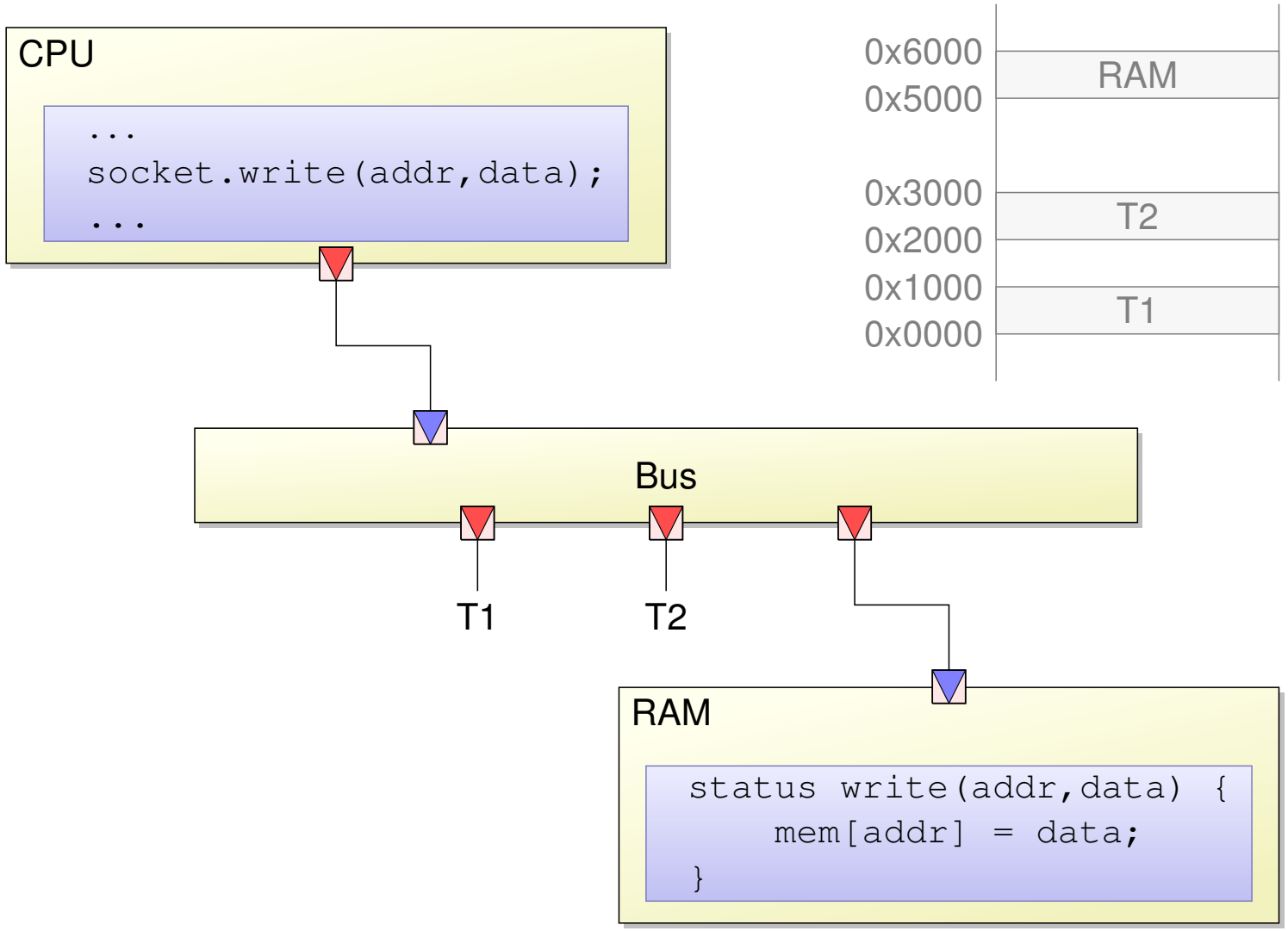
# This section

4 Tweto: TLM With Elaboration-Time Optimizations
- Limitations of Plain C++ Compilers and How to do Better
- Exploiting Constant Data
- Dealing with Addresses: Protocol-Aware Optimizations

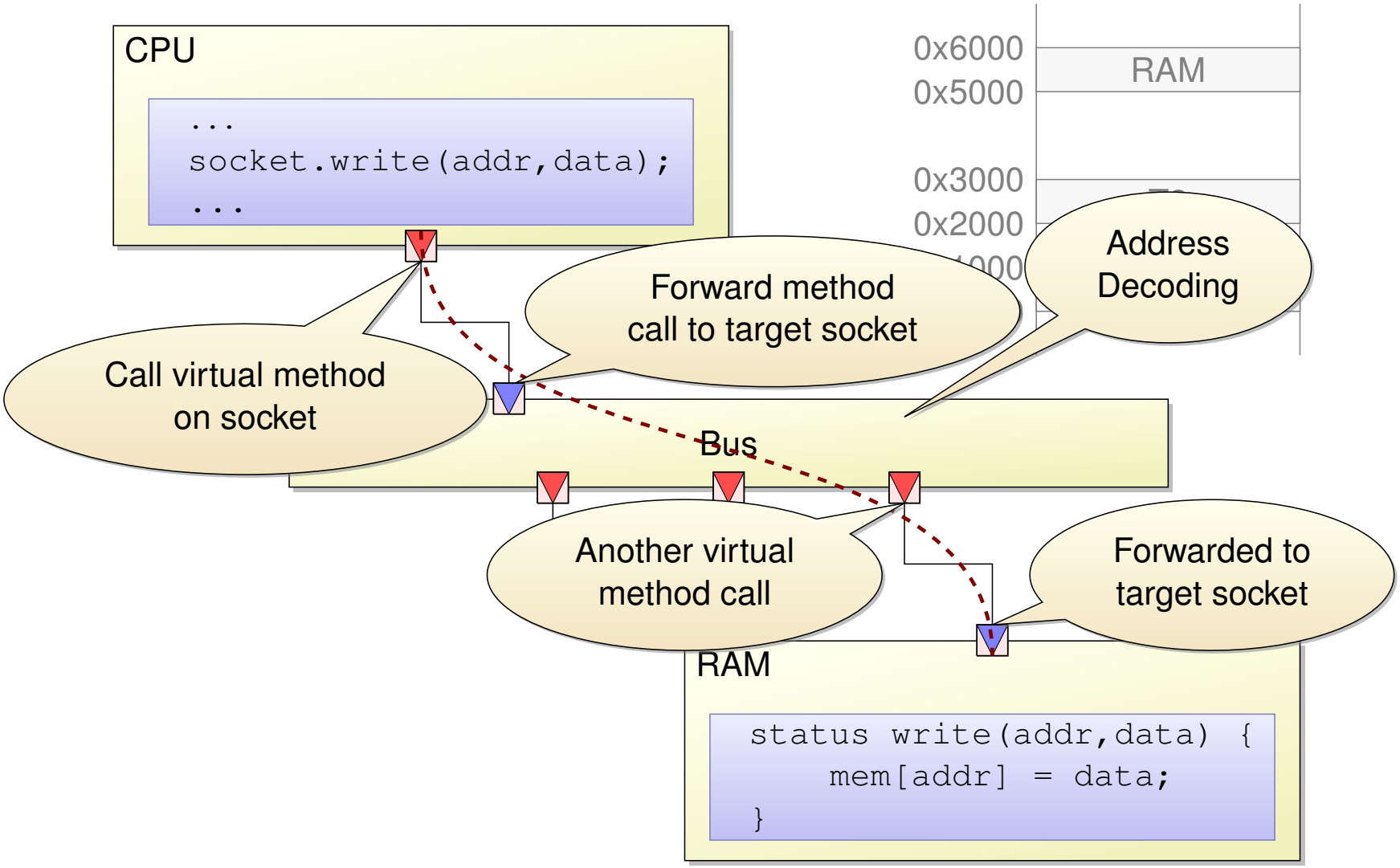
# Typical Transaction Journey



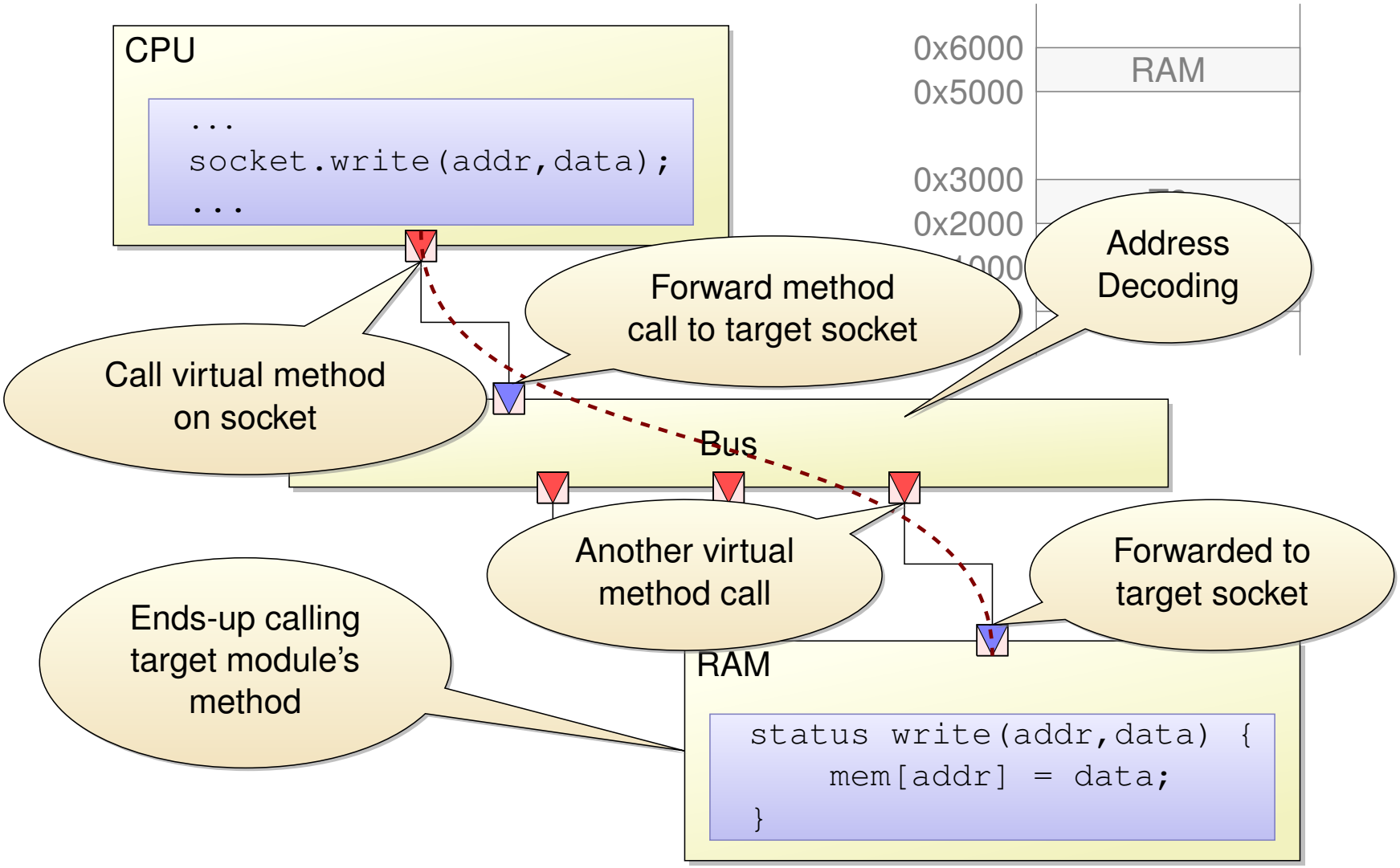
# Typical Transaction Journey



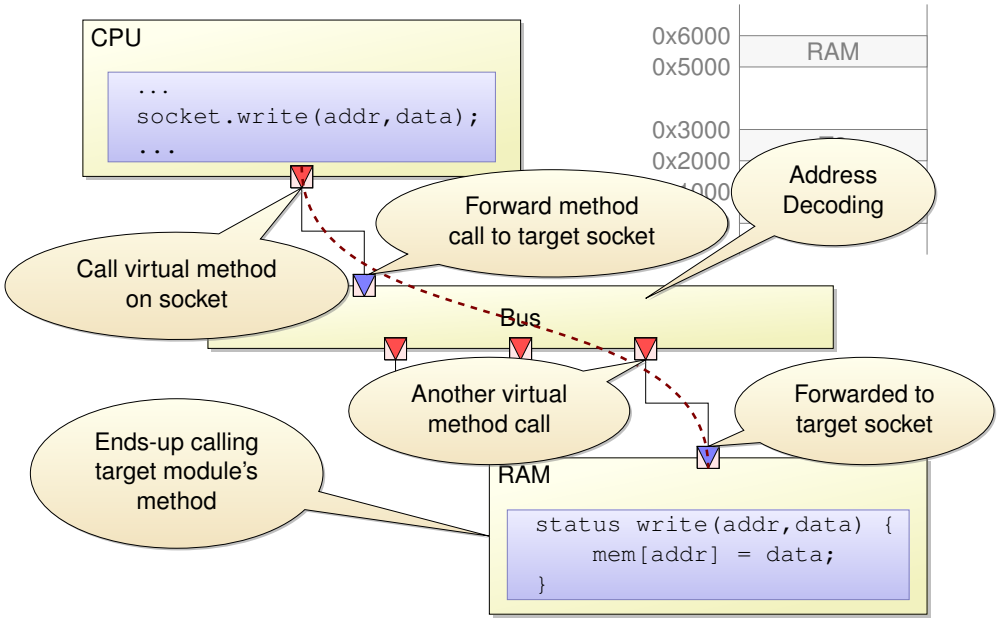
# Typical Transaction Journey



# Typical Transaction Journey

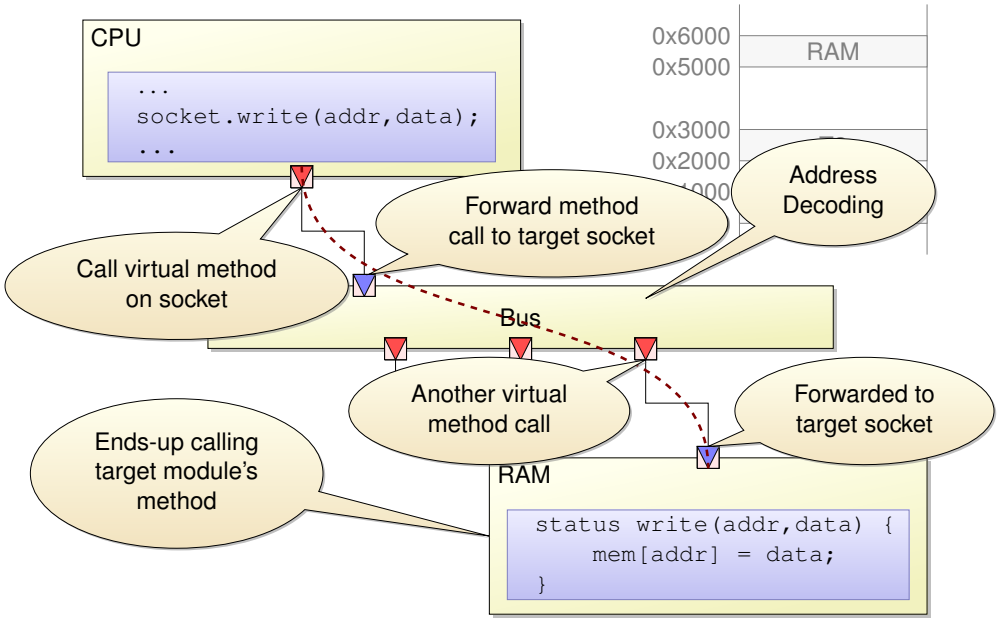


# Typical Transaction Journey




- Many costly operations for a simple functionality
- Work-around: backdoor access (DMI = Direct Memory Interface)
  - CPU get a pointer to RAM's internal data
  - Manual, dangerous optimization

# Typical Transaction Journey




- Many costly operations for a simple functionality
- Work-around: backdoor access (DMI = Direct Memory Interface)
  - CPU get a pointer to RAM's internal data
  - Manual, dangerous optimization

Can a compiler do as good as DMI, automatically?

# Basic Ideas

- Do statically what can be done statically ...
- ... considering “statically” = “after elaboration”
- Examples:
  - Virtual function resolution
  - Inlining through SystemC ports
  - Static address resolution

# This section

4 Tweto: TLM With Elaboration-Time Optimizations
- Limitations of Plain C++ Compilers and How to do Better
- Exploiting Constant Data
- Dealing with Addresses: Protocol-Aware Optimizations

# Constant data

- Things that are not `const` for C++, but constant for SystemC:
  - `this` pointer for each module
    $\Rightarrow$ becomes constant if we specialize functions
  - Architecture (set during elaboration, does not change after)
    $\Rightarrow$ marked (by user or within SystemC) with
    `tweto_mark_const()`

# Exploiting constant data

- Specialize process: for each instance, duplicate body, setting `this` pointer to its value
- Execute constant loads using data collected by `tweto_mark_const()`
- Resolve indirect calls once the called function is constant
- Specialize calls when their argument is constant

# Specialize Process (simplified)

- Starting point:

```
define @systemc_process(%struct.Module* %this) {
entry:
    ...
    ... ; uses %this
    ...
}
```

- Intermediate:

```
define @systemc_process_specialized() {
entry:
    %this = %struct.Module* <addr-of-this-module>;
    call @systemc_process (%this)
      ; use llvm::inlineFunction on this call
}
```

- Result: function with `this` replaced by constant

# Execute constant loads

- Replace `inttoptr` instructions by constant integer if possible
- `inttoptr` syntax:
  `<result> = inttoptr <ty> <value> to <ty2>`
- Example:

```
; X = (int *)42;
%X = inttoptr i32 42 to i32*
; Y = *X
%Y = load i32* %X
```

⇒

```
%Y = i32* <value at address 42>
```

# Resolve indirect calls

- Replace indirect `call` instructions by direct `call`.
- Example:

```
%1 = load <pointer_to_fun>
%2 = call %1(<function args>)
```

⇒

```
%2 = call fun(<function args>)
```

# Specialize calls

- Replace calls with constant args by specialized functions.
- Example:

```
read(int a){a+1;}
```

$\Rightarrow$

```
read_42(){42+1;}
```

# This section

4 Tweto: TLM With Elaboration-Time Optimizations
- Limitations of Plain C++ Compilers and How to do Better
- Exploiting Constant Data
- Dealing with Addresses: Protocol-Aware Optimizations

# What can we hope? (1)

```
#define TIMER_ADDR 0x1000
#define START_REG_OFFSET 4

cpu::compute () {
    ...
    // start timer
    socket.write(TIMER_ADDR+START_REG_OFFSET, 0);
    ...
}
```

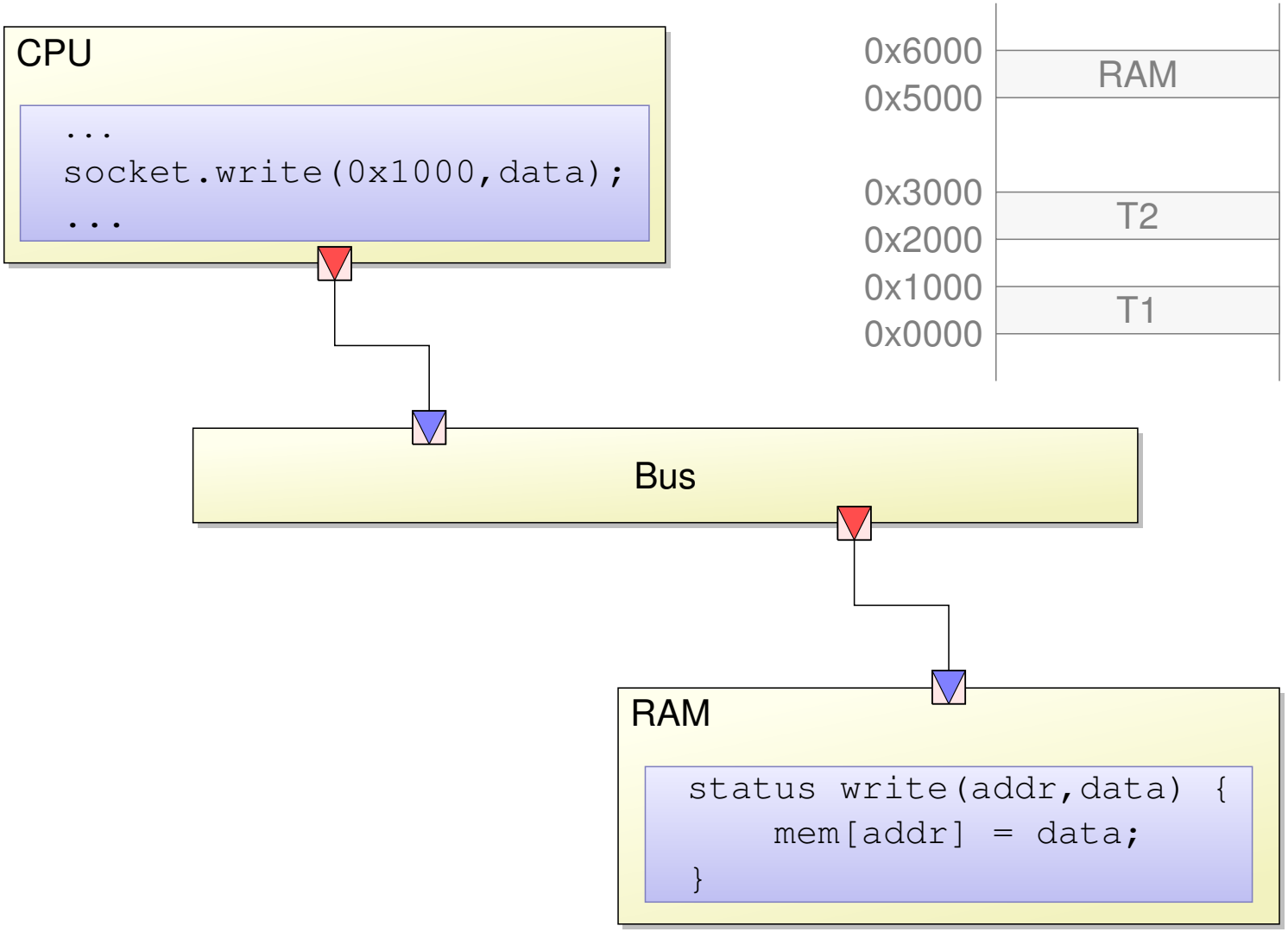
$\Rightarrow$ address known statically,
but address resolution still done dynamically

# What can we hope? (2)

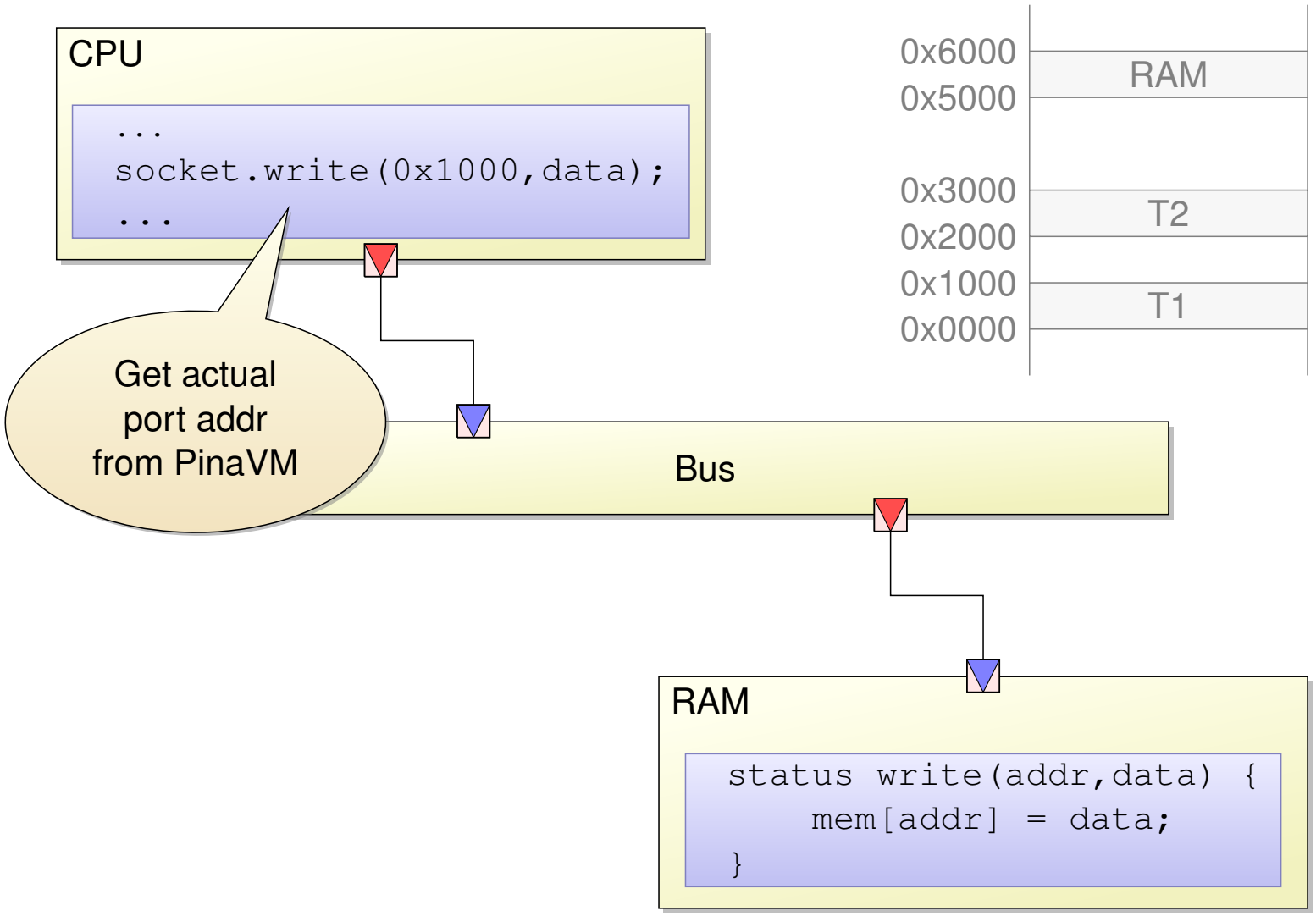
```
cpu::compute () {
    ...
    // Clear portion of RAM
    for (addr = 0x1000; addr < 0x2000; addr++) {
        socket.write(addr, 0);
    }
    ...
}
```

$\Rightarrow$ address not constant,
but simple analysis could allow static address resolution

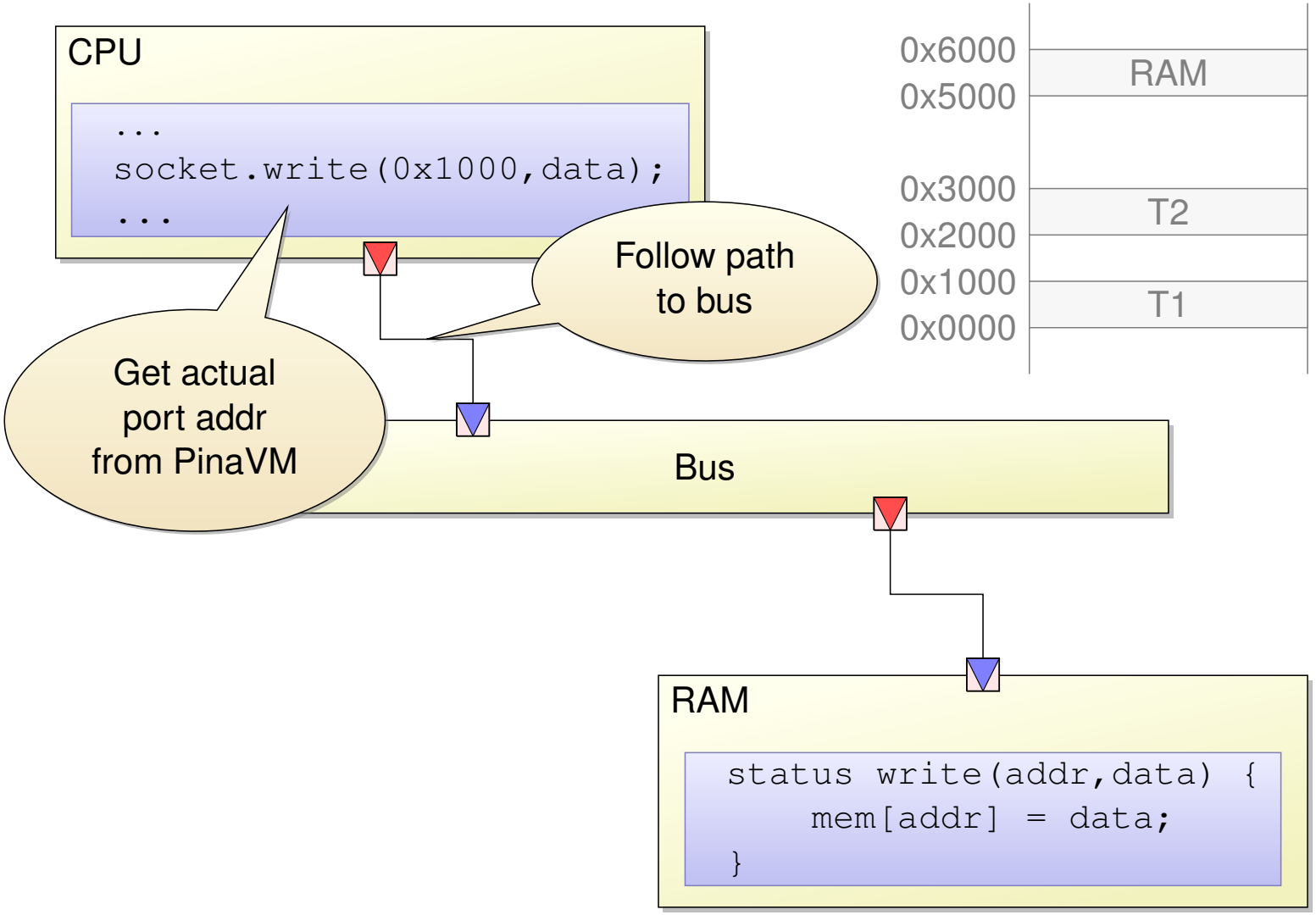
# Dealing with addresses *Statically*



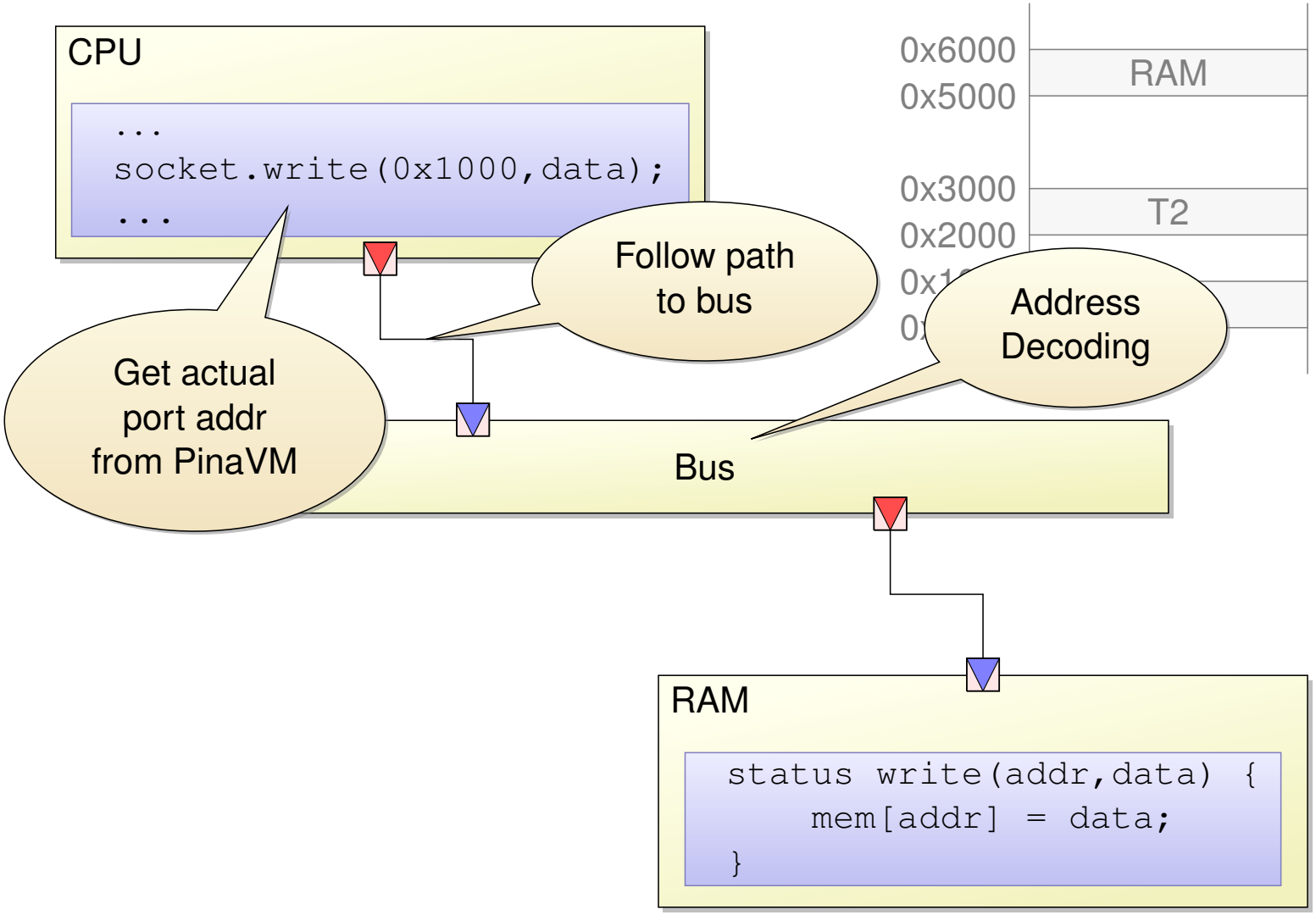
# Dealing with addresses *Statically*



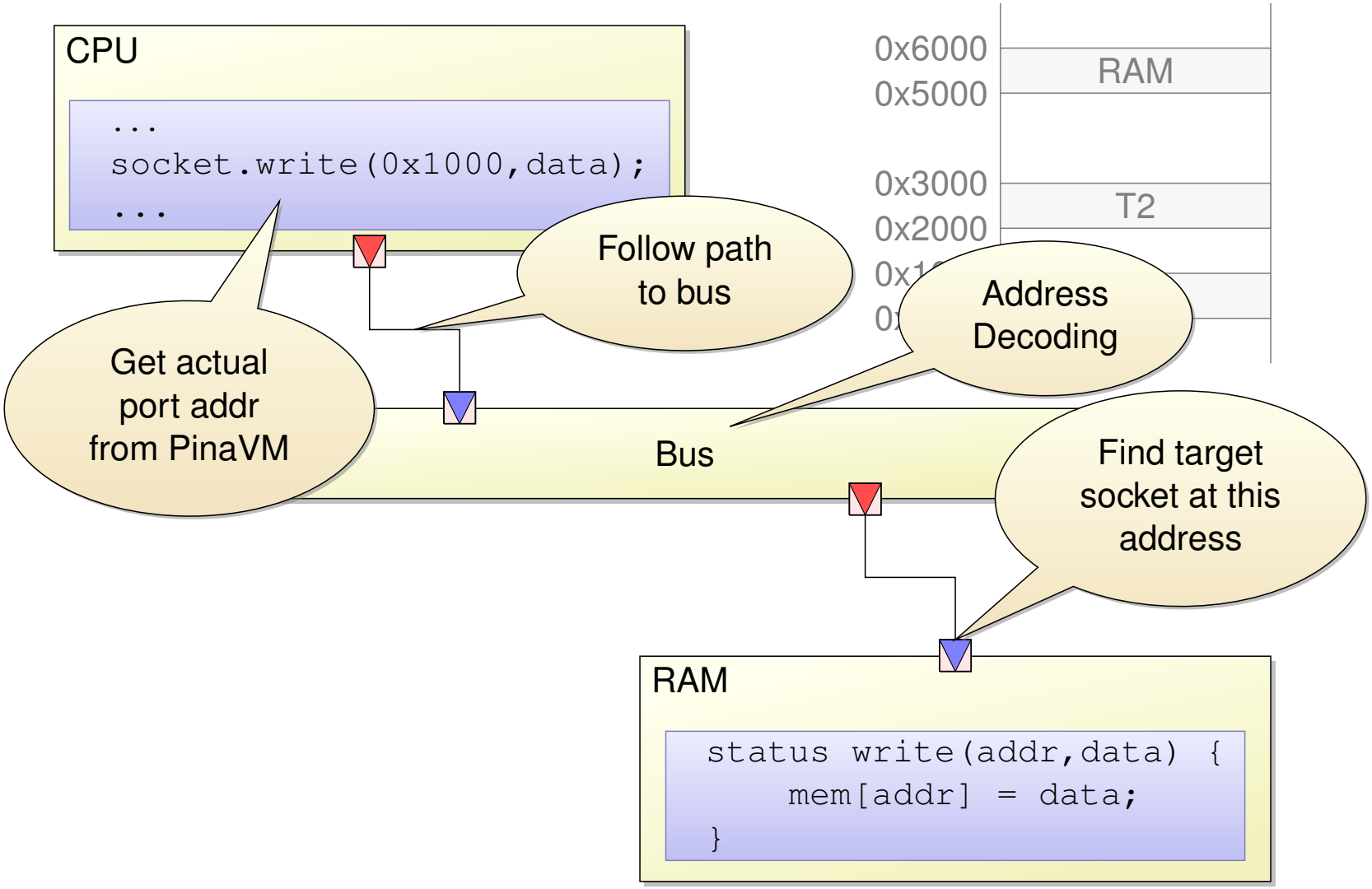
# Dealing with addresses *Statically*



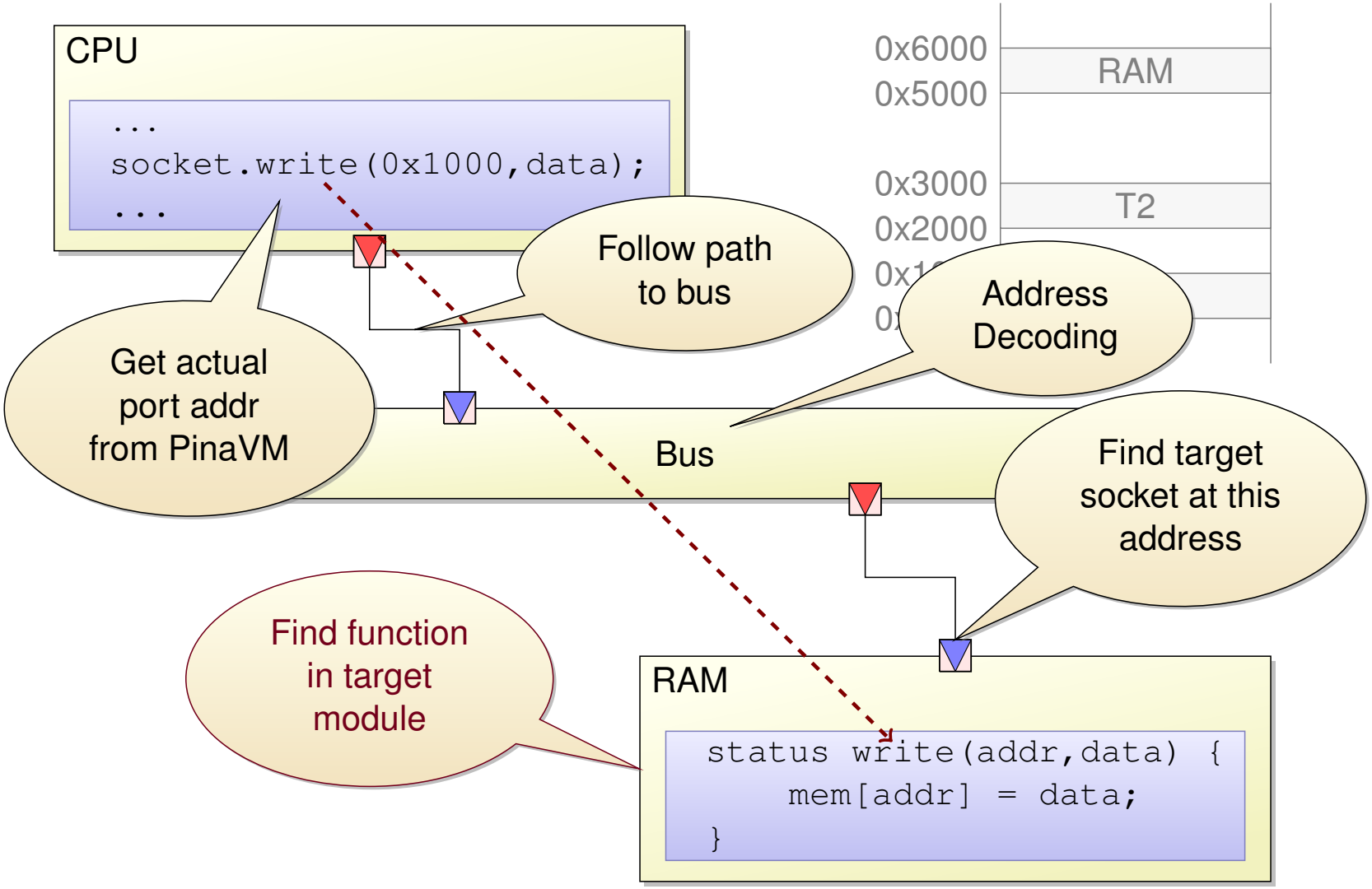
# Dealing with addresses *Statically*



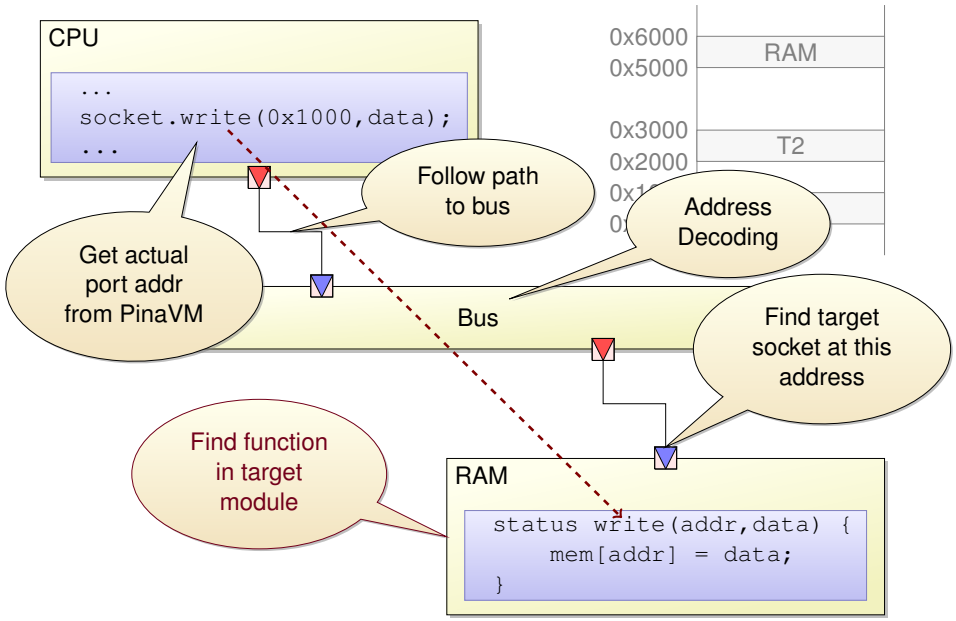
# Dealing with addresses *Statically*



# Dealing with addresses *Statically*



# Dealing with addresses *Statically*




- Possible optimizations:
  - Replace call to `socket.write()` with `RAM.write()`
  - Possibly inline it

# Optimizing Register Banks

```
// typical code executed when receiving transaction
status target::write(addr, data) {
    switch(addr) {
        case REG_1:
            f();
            break;
        case REG_2:
            g();
            break;
        default:
            return TLM_ADDRESS_ERROR_RESPONSE;
    }
    return TLM_OK_RESPONSE;
}
```

$\Rightarrow$ Accesses to constant addresses could execute `f` or `g` directly

# Summary

# This section

5 Conclusion

# PinaVM and Tweto

- PinaVM (front-end): prototype implemented
  (Kevin Marquet, Post-Doc Verimag; myself when I have time)
- Tweto (optimizer): work in progress
  (Claude Helmstetter, Post-doc LIAMA;
  Si-Mohamed Lamraoui, TER M1 Verimag)
- Other back-ends for PinaVM:
  - Promela: model-checking with SPIN
    (prototype; Kevin Marquet)
  - Simple: abstract interpretation with ConcurInterproc
    (draft; Kevin Marquet)
  - 42: Control Contracts
    (draft; Pierre-Yves Delahaye, TER Ensimag)

# Thank you

Questions?

http://gitorious.org/pinavm/pages/Home