30 Cycles or It's Free

Formal Timing and Correctness of Binary Code

Charles Averill

UTD Computer Security Group
The University of Texas at Dallas
Dartmouth College

September 2024

Background

- I work with Prof. Kevin Hamlen and Prof. Christophe Hauser

Background

- I work with Prof. Kevin Hamlen and Prof. Christophe Hauser
- We study *Formal Verification*, the process of writing mathematical proofs about code rather than relying on unit testing

Background

- I work with Prof. Kevin Hamlen and Prof. Christophe Hauser
- We study *Formal Verification*, the process of writing mathematical proofs about code rather than relying on unit testing
- We also specialize in binary code - this has a lot of cool implications (What happens if memory buffers overlap? How to handle modular arithmetic? What is a function call?)

Background

- I work with Prof. Kevin Hamlen and Prof. Christophe Hauser
- We study *Formal Verification*, the process of writing mathematical proofs about code rather than relying on unit testing
- We also specialize in binary code - this has a lot of cool implications (What happens if memory buffers overlap? How to handle modular arithmetic? What is a function call?)
- We often write proofs showing that a function is *correct* - its outputs meet some specification given the inputs - but our verification system is flexible enough to prove other properties

The Problem

- Nobody knows how long it takes code to run

The Problem

- Nobody knows how long it takes code to run (well, kind of...)

The Problem

- Nobody knows how long it takes code to run (well, kind of...)
- Refinement: nobody can write a formal proof that some code takes n clock cycles to execute

The Problem

- Nobody knows how long it takes code to run (well, kind of...)
- Refinement: nobody can write a formal proof that some code takes n clock cycles to execute
- That means that nuclear reactor controllers, autopilot software, elevators, and other things controlled by real-time code rely on hearsay and conjecture to know that they meet their essential timing constraints

The Problem

- Nobody knows how long it takes code to run (well, kind of...)
- Refinement: nobody can write a formal proof that some code takes n clock cycles to execute
- That means that nuclear reactor controllers, autopilot software, elevators, and other things controlled by real-time code rely on hearsay and conjecture to know that they meet their essential timing constraints
- These proofs would be useful in a few key critical areas:
  - Kernel-level real-time code such as the FreeRTOS kernel
  - User-level real-time code such as ArduPilot
  - Constant-time cryptography

The Solution

- Build formal models of software and CPUs
- Formally state your timing constraints w.r.t. code inputs
- Write a proof that your software meets your timing constraint
- Employ a machine to check that your proof is correct

The Challenges

- Build formal models of software and CPUs
  - Which models do you formalize? Languages? Interpreters/Compilers? CPU semantics? Hardware?
  - Some languages/compilers/CPUs have no formal specification, many more have no machine-readable formal specification
  - The proposition that the model matches the implementation is usually an assumption and unproven
- Formally state your timing constraints w.r.t. code inputs
- Write a proof that your software meets your timing constraint
- Employ a machine to check that your proof is correct

The Challenges

- Build formal models of software and CPUs
- Formally state your timing constraints w.r.t. code inputs
  - Timing constraints can be difficult to divine if code contains loops with complex termination conditions
  - Constraints should reference inputs/memory at beginning of function (rather than the memory state at the end) and it's difficult to show a path exists between the input and output state
- Write a proof that your software meets your timing constraint
- Employ a machine to check that your proof is correct

The Challenges

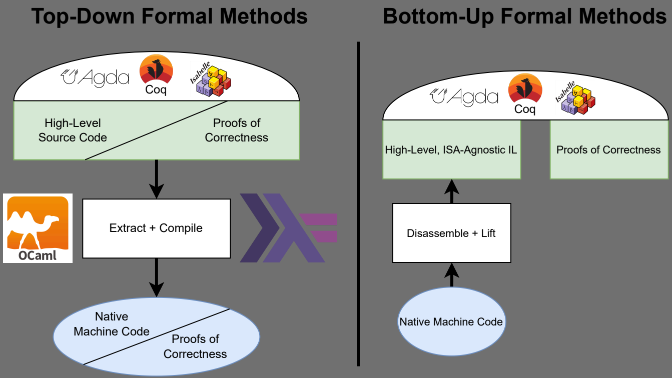
- Build formal models of software and CPUs
- Formally state your timing constraints w.r.t. code inputs
- Write a proof that your software meets your timing constraint
  - Writing formal proofs requires learning a proof language
  - Proofs about code must handle deep concepts such as decidability and termination
  - Reasoning about loops requires non-trivial insight learned by experience
- Employ a machine to check that your proof is correct

The Challenges

- Build formal models of software and CPUs
- Formally state your timing constraints w.r.t. code inputs
- Write a proof that your software meets your timing constraint
- Employ a machine to check that your proof is correct
    - Can we trust the machine to properly check?

The Challenges

- Build formal models of software and CPUs
- Formally state your timing constraints w.r.t. code inputs
- Write a proof that your software meets your timing constraint
- Employ a machine to check that your proof is correct



Formal Models of CPUs

- Picinæ is a verification framework designed here at UTD, used for writing proofs about binary code

Formal Models of CPUs

- Picinæ is a verification framework designed here at UTD, used for writing proofs about binary code
- Supports most major ISAs – x86, Amd64, Arm, RISC-V – and can be extended to more

Formal Models of CPUs

- Picinæ is a verification framework designed here at UTD, used for writing proofs about binary code
- Supports most major ISAs – x86, Amd64, Arm, RISC-V – and can be extended to more
- Built around the concept of *symbolic interpretation*, or stepping through the code without knowing the inputs, and keeping track of how the state changes

Formal Models of CPUs

- Picinæ is a verification framework designed here at UTD, used for writing proofs about binary code
- Supports most major ISAs – x86, Amd64, Arm, RISC-V – and can be extended to more
- Built around the concept of *symbolic interpretation*, or stepping through the code without knowing the inputs, and keeping track of how the state changes
- Is a *software* verification system - implements ISA but not hardware features like caching

Formal Timing Constraints

- What does it mean to constrain the execution time of a function?

Formal Timing Constraints

- What does it mean to constrain the execution time of a function?
  - We want some **arithmetic expression** that is **parametrized by the inputs** to the function

Formal Timing Constraints

- What does it mean to constrain the execution time of a function?
  - We want some **arithmetic expression** that is **parametrized by the inputs** to the function
- What kind of units do we want to use?

Formal Timing Constraints

- What does it mean to constrain the execution time of a function?
  - We want some **arithmetic expression** that is **parametrized by the inputs** to the function
- What kind of units do we want to use?
  - **Clock cycles**. Seconds would be nice, but cycles are fundamental unit of time in the CPU, correlation between cycles and seconds varies over time due to things like heat and charge (we don't want to verify physics stuff, too hard)
  - "CPU Instruction" to "Clock Cycles" relation is often known and documented in the CPU manual

Formal Timing Constraints

- What does it mean to constrain the execution time of a function?
  - We want some **arithmetic expression** that is **parametrized by the inputs** to the function
- What kind of units do we want to use?
  - **Clock cycles**. Seconds would be nice, but cycles are fundamental unit of time in the CPU, correlation between cycles and seconds varies over time due to things like heat and charge (we don't want to verify physics stuff, too hard)
  - "CPU Instruction" to "Clock Cycles" relation is often known and documented in the CPU manual
- Given our time constraint expression, how do we relate it to a program in a formal manner?

Formal Timing Constraints

- What does it mean to constrain the execution time of a function?
  - We want some **arithmetic expression** that is **parametrized by the inputs** to the function
- What kind of units do we want to use?
  - **Clock cycles**. Seconds would be nice, but cycles are fundamental unit of time in the CPU, correlation between cycles and seconds varies over time due to things like heat and charge (we don't want to verify physics stuff, too hard)
  - "CPU Instruction" to "Clock Cycles" relation is often known and documented in the CPU manual
- Given our time constraint expression, how do we relate it to a program in a formal manner?
  - **Program traces** - a history of the CPU, stored as a list of states, annotated by the address of the instruction that created that state

Program Trace Example

state

```
add t5, t0, t1
```

state' = state [R_T5 := R_T0 + R_T1]

```
ori s0, a0, a1
```

state'' = state' [R_S0 := R_A0 + R_A1]

```
beq t5, s0, Label
```

state''' = state'' [R_T5 = R_S0]
or
state''' = state'' [R_T0 + R_T1 = R_A0 + RA_1]

state''' = state'' [R_T5 != R_S0]
or
state''' = state'' [R_T0 + R_T1 != R_A0 + RA_1]

total cycles = 0

total cycles = 3

total cycles = 3 + 4

total cycles = 3 + 4 + (5 + (ML - 1))

total cycles = 3 + 4 + 3

CPU Timing Documentation

3.8. Instruction Timing

The instruction timing listed in the table below shows the required clock cycles for executing a certain instruction. These instruction cycles assume a bus access without additional wait states and a filled pipeline.

Average CPI (cycles per instructions) values for "real applications" like for executing the CoreMark benchmark for different CPU configurations are presented in CPU Performance.

Table 36. Clock cycles per instruction

Class	ISA	Instruction(s)	Execution cycles
ALU	`I/E`	`addi` `slti` `sltiu` `xori` `ori` `andi` `add` `sub` `slt` `sltu` `xor` `or` `and` `lui` `auipc`	2
ALU	`C`	`c.addi4spn` `c.nop` `c.addi` `c.li` `c.addi16sp` `c.lui` `c.andi` `c.sub` `c.xor` `c.or` `c.and` `c.add` `c.mv`	2
ALU	`I/E`	`slli` `srli` `srai` `sll` `srl` `sra`	3 + SA[6]/4 + SA%4; FAST_SHIFT[7]: 4; TINY_SHIFT[8]: 2..32
ALU	`C`	`c.srli` `c.srai` `c.slli`	3 + SA[9]; FAST_SHIFT[10]:
Branches	`I/E`	`beq` `bne` `blt` `bge` `bltu` `bgeu`	Taken: 5 + (ML-1)[11]; Not taken: 3
Branches	`C`	`c.beqz` `c.bnez`	Taken: 5 + (ML-1); Not taken: 3
Jumps / Calls	`I/E`	`jal` `jalr`	5 + (ML-1)
Jumps / Calls	`C`	`c.jal` `c.j` `c.jr` `c.jalr`	5 + (ML-1)

CPU Timing Documentation

```
Definition neorv32_cycles_upper_bound (ML : N) (s : store) (instr : N) :=
    let reg_or_max (reg : N) : N := ... in
    (* addi slti sltiu xori ori andi add sub slt sltu xor ... : 2 *)
    let op := riscv_opcode instr in
    if op =? 51 then (* 0110011 : R-type *)
        let '(funct7, rs2, rs1, funct3, rd, opcode) := decompose_Rtype instr in
        if contains [0;2;3;4;6;7] funct3 then
            (* add sub xor or and slt sltu *)
            Some 2%N
        else
            (* sll srl sra : [ 3 + shamt/4 + shamt%4 ]*)
            (* shamt := rs2 *)
            (* Constant shift times with FAST_SHIFT_EN or TINY_SHIFT_EN *)
            Some (3 + (reg_or_max rs2 / 4) + ((reg_or_max rs2) mod 4))%N
    else if op =? 3 then (* 0000011 : I-type *)
        let '(imm, rs1, funct3, rd, opcode) := decompose_Itype instr in
        (* lb lh lw lbu lhu : [ 5 + (ML - 2) ] *)
        Some (5 + (ML - 2))%N
    ...
```

Formalization of Program Execution Time

We can write a function that computes the total execution time for a program trace:

```
Definition trace : Type := list (addr * store).

Definition cycle_count_of_trace (t : trace) : N :=
    List.fold_left N.add (List.map (fun '(a, s) => time_of_addr s a) t) 0.
```

Formalization of Program Execution Time

We can write a function that computes the total execution time for a program trace:

```
Definition trace : Type := list (addr * store).

Definition cycle_count_of_trace (t : trace) : N :=
    List.fold_left N.add (List.map (fun '(a, s) => time_of_addr s a) t) 0.
```

But we have a problem! You can't actually run code in Picinæ, so how do we actually use this function?

Formalization of Program Execution Time

We can write a function that computes the total execution time for a program trace:

```
Definition trace : Type := list (addr * store).

Definition cycle_count_of_trace (t : trace) : N :=
    List.fold_left N.add (List.map (fun '(a, s) => time_of_addr s a) t) 0.
```

But we have a problem! You can't actually run code in Picinæ, so how do we actually use this function?

Answer: we write a proof saying that for all possible inputs to function, the output is some XYZ number of cycles!

Example: `addloop`

RISC-V Assembly

```
        ori     t2,zero,1
        andi    t3,t3,0
add:
        beq     t0,t3,20 <end>
        addi    t1,t1,1
        sub     t0,t0,t2
        beq     t3,t3,10 <add>
end:
```

RISC-V Binary

```
0x00106393000e7e13
  01c2886300130313
  407282b3ffce0ae3
```

Picinæ IL

```
Move R_T2 (BinOp OP_OR (Word 0 32) (Word 1 32))
Move R_T3 (BinOp OP_AND (Var R_T3) (Word 0 32))
If (BinOp OP_EQ (Var R_T0) (Var R_T3)) (Jmp (Word 32 32)) Nop
Move R_T1 (BinOp OP_PLUS (Var R_T1) (Word 1 32))
Move R_T0 (BinOp OP_MINUS (Var R_T0) (Var R_T2))
If (BinOp OP_EQ (Var R_T3) (Var R_T3)) (Jmp (Word 16 32)) Nop
```

Example: addloop

```
(*                                               -> function inputs <-     *)
Definition addloop_timing_invs (_ : store) (p : addr) (x y : N) (t:trace) :=
match t with (Addr a, s) :: t' => match a with
  | 0xc  => Some (s R_T0 = x /\ s R_T2 = 1 /\
     cycle_count_of_trace t = 2 + 2)
  | 0x10 => Some (exists t0, s R_T0 = t0 /\ s R_T2 = 1 /\ s R_T3 = 0
         /\ t0 <= x /\
     cycle_count_of_trace t' = 4 + (x - t0) * (12 + (ML - 1)))
                          (* 2 + 2 + (x - t0) * (3 + 2 + 2 + (5 + (ML - 1)) *)
  | 0x20 => Some (
     (* This one is our timing constraint! *)
     (* ML is memory latency - the time in cycles to do a mem read *)
     (* time of addloop(x, y) = 9 + time_mem + x * (12 + time_mem) *)
     (*                       = [setup time] + x * [loop time]     *)
     cycle_count_of_trace t' = 9 + (ML - 1) + x * (12 + (ML - 1)))
               (* 2 + 2 + (5 + (ML - 1)) + x * (3 + 2 + 2 + (5 + (ML - 1))) *)
  | _ => None end
| _ => None
end.
```

A Real Example

- A toy example will not get me published somewhere nice

A Real Example

- A toy example will not get me published somewhere nice
- Looking back at motivation: can we find some real-world, real-time code to formally time?

A Real Example

- A toy example will not get me published somewhere nice
- Looking back at motivation: can we find some real-world, real-time code to formally time?
- Initially looked at ArduPilot, open-source autopilot software

A Real Example

- A toy example will not get me published somewhere nice
- Looking back at motivation: can we find some real-world, real-time code to formally time?
- Initially looked at ArduPilot, open-source autopilot software
- Why bother messing around in userspace? Let's time something from FreeRTOS so that everybody gets some benefit

A Real Example

- A toy example will not get me published somewhere nice
- Looking back at motivation: can we find some real-world, real-time code to formally time?
- Initially looked at ArduPilot, open-source autopilot software
- Why bother messing around in userspace? Let's time something from FreeRTOS so that everybody gets some benefit

FreeRTOS is a market-leading embedded system RTOS supporting 40+ processor architectures with a small memory footprint, fast execution times, and cutting-edge RTOS features and libraries including Symmetric Multiprocessing (SMP), a thread-safe TCP stack with IPv6 support, and seamless integration with cloud services. Its open-source and actively supported and maintained.

vTaskSwitchContext

- Prepares to switch CPU context between available ready tasks

vTaskSwitchContext

- Prepares to switch CPU context between available ready tasks
- Chooses next task based on known priority values

vTaskSwitchContext

- Prepares to switch CPU context between available ready tasks
- Chooses next task based on known priority values
- Very critical - if timing leak here, planes crash, reactors melt down, etc.

vTaskSwitchContext

- Prepares to switch CPU context between available ready tasks
- Chooses next task based on known priority values
- Very critical - if timing leak here, planes crash, reactors melt down, etc.
- Constrained, yet interesting control flow and memory problems to solve

vTaskSwitchContext

- Prepares to switch CPU context between available ready tasks
- Chooses next task based on known priority values
- Very critical - if timing leak here, planes crash, reactors melt down, etc.
- Constrained, yet interesting control flow and memory problems to solve
- TLDR; a real-world example with real security implications that provides a compelling, yet doable case study for the practicality/utility of Picinæ timing proofs

vTaskSwitchContext

```
Definition time_of_vTaskSwitchContext (t : trace) (mem : addr -> N) : Prop :=
  (* is the scheduler suspended? *)
  if (uxSchedulerSuspended gp mem) =? 0 then
    cycle_count_of_trace t = (* total number of cycles equals... *)
    25 + 3 * time_branch + 17 * time_mem +
        (* This branch condition isn't well-documented,
           need to dig into source *)
    (if
     (mem[4 + mem[gp - 920 + (31 - clz (uxTopReadyPriority gp mem) 32) * 20]])
          =? ((gp - 916) + (31 - clz (uxTopReadyPriority gp mem) 32) * 20)
       then
         22 + (clz (uxTopReadyPriority gp mem) 32) + 5 * time_mem
       else
         19 + time_branch + (clz (uxTopReadyPriority gp mem) 32) + 3 * time_mem
       (* time_branch = 5 + (memory latency - 1)
          # of cycles for a successful/taken branch *)
       (* time_mem = 5 + (memory latency - 2),
          # of cycles for a memory retreival *))
  else
      cycle_count_of_trace t = 5 + time_branch + 2 * time_mem.
```

Another Example: Timing Attacks

- Some cryptographic ciphers can be reversed if an attacker has knowledge of how long the cipher takes to run

Another Example: Timing Attacks

- Some cryptographic ciphers can be reversed if an attacker has knowledge of how long the cipher takes to run
- This problem has spawned the field of *Constant-Time Cryptography*, where we write ciphers that do not have timing differences depending on the value of the input

Another Example: Timing Attacks

- Some cryptographic ciphers can be reversed if an attacker has knowledge of how long the cipher takes to run
- This problem has spawned the field of *Constant-Time Cryptography*, where we write ciphers that do not have timing differences depending on the value of the input
- Some future/ongoing work for us: verify that an encryption cipher is algorithmically invulnerable to timing attacks

Another Example: Timing Attacks

- Some cryptographic ciphers can be reversed if an attacker has knowledge of how long the cipher takes to run
- This problem has spawned the field of *Constant-Time Cryptography*, where we write ciphers that do not have timing differences depending on the value of the input
- Some future/ongoing work for us: verify that an encryption cipher is algorithmically invulnerable to timing attacks
- How to show invulnerability?

Another Example: Timing Attacks

- Some cryptographic ciphers can be reversed if an attacker has knowledge of how long the cipher takes to run
- This problem has spawned the field of *Constant-Time Cryptography*, where we write ciphers that do not have timing differences depending on the value of the input
- Some future/ongoing work for us: verify that an encryption cipher is algorithmically invulnerable to timing attacks
- How to show invulnerability? Code is invulnerable to timing attacks if no sensitive data appears in your timing expression!

TLDR

- We developed a way to write proofs about the time machine code takes to run

TLDR

- We developed a way to write proofs about the time machine code takes to run
- Real-time code needs these guarantees to ensure the physical safety of real-time critical systems

TLDR

- We developed a way to write proofs about the time machine code takes to run
- Real-time code needs these guarantees to ensure the physical safety of real-time critical systems
- These guarantees can show the timing safety of cryptographic ciphers

TLDR

- We developed a way to write proofs about the time machine code takes to run
- Real-time code needs these guarantees to ensure the physical safety of real-time critical systems
- These guarantees can show the timing safety of cryptographic ciphers
- This approach shows that trace properties are an elegant way to add arbitrary capabilities to our proof system