Formally-Verified, Tight Timing Constraints for Machine Code

Charles Averill[1]

[1]**charles@utdallas.edu**

ABSTRACT

To provide a solid foundation for the problem of real-time patching, we present a formal framework for describing the timing properties of machine code to arbitrary levels of precision. This is accomplished via an extension of Picinæ, an infrastructure developed for machine-proving properties of native code in Coq. As a proof of concept, we provide a formally-proven, tight execution time bound for the context switch preparation routine of FreeRTOS. Finally, we outline directions for automatic specification generation, proof tactics, and program trace properties to enhance the utility and accessibility of the framework.

1 INTRODUCTION

Real-time systems, which must respond to events within strict time constraints, are ubiquitous in mission-critical domains such as aerospace, automotive, and medical devices. Ensuring the correctness and performance of these systems is vital, not only in terms of functional behavior but also in terms of non-functional properties such as execution time. One key challenge that arises during the lifecycle of such systems is the need for *patching*, or modifying the binary code to address bugs, vulnerabilities, or functional updates.

Patching real-time systems is inherently risky because any modification to the code may introduce unintended timing side effects, such as increasing the execution time of critical paths or introducing new timing dependencies. In many cases, traditional testing approaches that measure execution time through profiling or simulation are used to provide assurance that the patched code maintains its pre-patch timing properties. However, these approaches suffer from several limitations.

First, profiling and simulation are empirical methods that do not guarantee exhaustive coverage of all possible execution paths. While they may detect timing issues in commonly executed paths, they cannot ensure that malicious or edge-case situations do not result in timing violations. Furthermore, testing is often performed under specific hardware conditions, making it difficult to generalize the results to all operational environments. These limitations render the informal approach unsafe for systems where failure to meet timing deadlines could have catastrophic consequences.

Given these challenges, a formal approach to verifying timing properties is necessary to ensure the safe application of patches in real-time, mission-critical systems. By leveraging formal verification methods, it is possible to model the exact execution time of binary machine code in cycles. Existing formal approaches [1] at timing tend to focus on compiler-generated code, and largely utilize model checking over formal proofs. Implementing these methods in a dependently-typed, machine-checked proof software such as Coq provides the highest level of assurance of timing. Here, we present a framework for the formal verification of the execution time of binary code. This approach functions as the formal groundwork for a future project that specifically targets the timing behaviors of code before and after patching.

2 OVERVIEW

We utilize an existing framework, Picinæ [3], that formalizes the behavior of all major ISAs in Coq via symbolic interpretation. We chose Picinæ due to its versatility in expressing specifications for machine code, specifically in expressing specifications that involve *traces*, records of execution history, of code.

We specify timing properties as the sum of the cycle counts of each instruction in the program's trace. This allows us to write arbitrarily-precise timing boundaries, rather than just upper bounds. Furthermore, we can parameterize the timing bounds over the input state of the code, resulting in specifications perfect for prediction of execution time for specific scenarios.

The structure and interpretation of Picinæ timing proofs tends to be vastly simpler than those of Picinæ correctness proofs. Similarly to correctness proofs, we utilize loop invariants to outline the high-level structure of our proof and constrain the symbolic state when branching and looping are encountered.

While correctness invariants typically require complex, novel insight into the partial correctness of code up to a certain point, timing invariants just need to state the number of clock cycles that have passed up to the point of the invariant. Similarly, the proof obligations of correctness proofs require complex reasoning about binary modular arithmetic, loop optimizations, and memory properties, while timing proof obligations tend not to interact with any of these complications. Timing proofs tend to follow the pattern of:

- Step the symbolic interpreter forward until an invariant is reached
- Simplify the timing obligation down to a simple equality
- Substitute retained information about branching
- Dispatch a standard Coq arithmetic equality decision tactic to solve the goal

We provide three proofs of concept for our approach:

- `addloop` - A simple loop that adds two numbers by decrementing the left operand and incrementing the right operand until the left is zero (complete)
- `vTaskSwitchContext` - A function pulled from the FreeRTOS scheduler that prepares the CPU for switching contexts - control flow has no standard loops, but has a series of nested branches and a stack overflow checker that goes into an infinite loop when overflows are discovered (complete)
- `ChaCha20` - A custom implementation of the ChaCha20 cipher, modeled directly after the original RFC (incomplete)

Each of these examples has been lifted to the corresponding Coq format and has been given a timing specification. `addloop` and `vTaskSwitchContext` have completed proofs of timing.

Given the structure of Picinæ timing proofs, we are confident that the task can be completely automated for the majority of code, and partially automated for all code. In Section 3 we cover specific implementation details of the approach. In Section 4 we cover our plans for automation and general system improvements.

3 TECHNICAL DETAILS

3.1 Picinæ Proof Structure

Proofs in Picinæ are accomplished via abstract interpretation over an intermediate representation of machine code. The structure of these proofs is governed by a user-provided set of loop invariants that constrain the control flow of the target program. Given a set of invariants, an input point, and a set of exit points for the target program, the user is provided an obligation to prove that from each entry point or invariant, the control flow reaches a provable invariant or postcondition.

Consider the following RISC-V program `addloop`, which adds the contents of registers `t0` and `t1` by incrementing and decrementing until `t0` is zero.

```
func:
    li t2, 1
    andi t3, t3, 0
addloop:
    beq t0, t3, end
    addi t1, t1, 1
    sub t0, t0, t2
    beq t3, t3, addloop
end:
```

At termination, we would like register `t1` to contain the sum of the original operands, which we will call x and y. The key to proving this property is the knowledge that at the beginning of each iteration in the loop,

$$\texttt{t0} + \texttt{t1} = x + y.$$

As the loop exits and branches to the label `end`, we will have the knowledge that

$$\texttt{t0} = 0,$$

so we will have that

$$\texttt{t1} = x + y.$$

Picinæ allows us to elegantly represent this argument:

```
(* Assembled RISC-V code *)
Definition add_loop_riscv (a : addr) : N :=
    match a with
    | 0x8  => 0x00106393 (* ori     t2,zero,1 *)
    | 0xc  => 0x000e7e13 (* andi    t3,t3,0 *)
    | 0x10 => 0x01c28863 (* beq     t0,t3,20 <end> *)
    | 0x14 => 0x00130313 (* addi    t1,t1,1 *)
    | 0x18 => 0x407282b3 (* sub     t0,t0,t2 *)
    | 0x1c => 0xffce0ae3 (* beq     t3,t3,10 <add> *)
    | _ => 0
    end.
Definition addloop_start := 0x8.
Definition addloop_exit (t:trace) :=
  match t with (Addr a,_)::_ => match a with
  | 0x20 => true
  | _ => false
  end | _ => false end.

(* Computes the IL for the assembled code *)
Definition lifted_addloop := lift_riscv add_loop_riscv.

(* Correctness specification *)
Definition postcondition (x y : N) (s : store) :=
    s R_T1 = Dword (x (+) y).

(* Invariant set *)
Definition addloop_correctness_invs (_ : store) (p : addr) (x y : N) (t : trace) :=
    match t with (Addr a, s) :: _ => match a with
        | 0x8 => Some (s R_T0 = Dword x /\ s R_T1 = Dword y)
        | 0x10 => Some (exists t0 t1,
            s R_T0 = Dword t0 /\ s R_T1 = Dword t1 /\ s R_T2 = Dword 1 /\ s R_T3 = Dword 0 /\
                t0 (+) t1 = x (+) y)
        | 0x20 => Some (postcondition x y s)
        | _ => None
        end
    | _ => None
    end.

(* Main proof *)
Theorem addloop_partial_correctness:
  forall s p t s' x' x y
         (ENTRY: startof t (x',s') = (Addr 0x8,s)) (* Define the entry point of the function *)
         (MDL: models rvtypctx s)
         (T0: s R_T0 = Dword x)                    (* Tie the contents of T0 to x *)
         (T1: s R_T1 = Dword y),                   (* Tie the contents of T1 to y *)
  satisfies_all
    lifted_addloop                                 (* Provide lifted code *)
```

```
    (addloop_correctness_invs s p x y)              (* Provide invariant set *)
    addloop_exit                                    (* Provide exit point *)
  ((x',s')::t).
Proof.
    intros.
    apply prove_invs.

    (* Base case *)
    simpl. rewrite ENTRY. step. auto.

    (* Inductive step setup *)
    intros.
    eapply startof_prefix in ENTRY; try eassumption.
    eapply preservation_exec_prog in MDL; try (eassumption || apply lift_riscv_welltyped).
    clear - PRE MDL. rename t1 into t. rename s1 into s'.

    (* Most of proof starts here *)
    destruct_inv 32 PRE.

    (* Starting from our entrypoint at address 0x8, we'll do some setup and then
       step to the next invariant *)
    destruct PRE as [T0 T1]. step. step.
    (* This goal is the invariant at 0x10 and has taken us to the case in which
       the loop has just started *)
    exists a, b. auto.

    (* We now have to deal with the cases where the loop terminates and where it
       loops around. Here we get our invariant as an assumption. *)
    destruct PRE as [t0 [t1 [T0 [T1 [T2 [T3 Eq]]]]]].
    (* Termination case - time to prove the postcondition *)
    step.
        rewrite N.eqb_eq in BC. subst. psimpl in Eq.
        unfold postcondition. psimpl. rewrite T1.
        rewrite Eq. reflexivity.
    (* Loop case - prove the invariant again *)
    step. step. step.
        rewrite N.eqb_neq in BC. exists (t0 (-) 1), (1 (+) t1). repeat split.
        psimpl. assumption.
Qed.
```

3.2 Picinæ for Timing

Picinæ's proof structure turns out to be so generic that we can directly embed the concept of timing into the system via a property of program traces. For example, assume we have a function describing the number of cycles a given instruction takes to execute. By mapping this instruction timing function over the program's trace and summing up the list, we obtain the exact number of cycles the program has taken to execute:

```
Axiom ML : N.

Definition instr_at_addr (a : addr) : N :=
    ...

Definition cycles_of_instr (s : store) (instr : N) : option N :=
    ...

Definition cycle_count_of_trace (t : trace) : N :=
```

```
    List.fold_left N.add (List.map
        (fun '(e, s) => match e with
            | Addr a =>
                cycles_of_instr s (instr_at_addr a)
            | Raise n => max32
            end) t) 0.
```

We can now begin to define invariants and postconditions in terms of `cycle_count_of_trace`, quantifying over the infinite set of traces and program states. Consider the following proof of timing for the previously-specified `addloop`:

```
Variable ML : N.

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 = Dword x /\ s R_T2 = Dword 1 /\ cycle_count_of_trace t = 2 + 2)
    | 0x10 => Some (exists t0, s R_T0 = Dword t0 /\ s R_T2 = Dword 1 /\ s R_T3 = Dword 0
        /\ t0 <= x /\
        (* setup time + (number of iterations) * (duration of one iteration) *)
        cycle_count_of_trace t' = 4 + (x - t0) * (12 + (ML - 1)))
    | 0x20 => Some (cycle_count_of_trace t' = 9 + (ML - 1) + x * (12 + (ML - 1)))
    | _ => None end
| _ => None
end.

Theorem addloop_timing:
  forall s p t s' x' a b
         (ENTRY: startof t (x',s') = (Addr 0x8,s)) (* Define the entry point of the function *)
         (MDL: models rvtypctx s)
         (T0: s R_T0 = VaN a 32)                   (* Tie the contents of T0 to a *)
         (T1: s R_T1 = VaN b 32),                  (* Tie the contents of T1 to b *)
  satisfies_all
    lifted_addloop                                 (* Provide lifted code *)
    (addloop_timing_invs s p a b)                  (* Provide invariant set *)
    addloop_exit                                   (* Provide exit point *)
  ((x',s')::t).
Proof using.
    intros.
    apply prove_invs.

    (* Base case *)
    simpl. rewrite ENTRY.
    (* Whammer is a general-purpose timing proof tactic *)
    repeat whammer.

    (* Inductive step setup *)
    intros.
    eapply startof_prefix in ENTRY; try eassumption.
    eapply preservation_exec_prog in MDL; try (eassumption || apply addloop_welltyped).
    clear - PRE MDL. rename t1 into t. rename s1 into s'.

    (* Meat of proof starts here *)
    destruct_inv 32 PRE.

    (* Addr 0xc *)
    destruct PRE as [T0 [T2 Cycles]].
```

```
    repeat whammer.

    (* Addr 0x10 *)
    destruct PRE as [t0 [T0 [T2 [T3 [T0_A Cycles_t]]]]].
    step.
    - (* t0 = 0 -> exit *)
        apply N.eqb_eq in BC; subst.
        repeat whammer.
    - (* t0 <> 0 -> loop again *)
        assert (1 <= t0) by (apply N.eqb_neq in BC; lia).
        repeat whammer.
            rewrite msub_nowrap; psimpl; lia.

            rewrite msub_nowrap, N_sub_distr.
        all: whammer.
Qed.
```

We can see that timing proofs are on the same order of size as correctness proofs, but have some interesting properties:

- Loop invariants are vastly simpler - instead of devising clever generic properties of data throughout an ongoing loop, our loops tend to take the form of

  $$\textit{time before loop} + (\textit{loop iteration count} \times \textit{time of one iteration}).$$

  Given this simplicity, there is a large potential for fully-automatic invariant generation for most loops

- Timing proofs use much of the same machinery of Picinæ, although the proof obligations tend to be much simpler. This allows for a wealth of automation tactics to be used, such as `whammer`, which is designed to reduce timing proof load

3.3 Trusted Computing Base

Additions to Picinæ's TCB consist only of the previosly-mentioned implementations of `cycles_of_instr` and `cycle_count_of_trace`. Our examples target the open-source NEORV32 RISC-V processor. Its datasheet [4] provides a table mapping each supported instruction with its maximum execution time in cycles, assuming a bus access without additional wait states and a filled pipeline. We directly implement this table in `cycles_of_instr`, keeping our TCB restrained to manufacturer-provided documentation.

Picinæ additionally trusts its lifter and the semantics for its abstract interpreter. Its interpreter semantics come directly from BAP [2], and its lifter largely depends on Ghidra for disassembly and semantic translation.

3.4 Example: `vTaskSwitchContext`

As a more practical example, we present the following timing specification for a fragment of the FreeRTOS scheduler. `vTaskSwitchContext` prepares the CPU for task switching, checking and setting various properties of the CPU to ensure soundness before this critical action. This function contains multiple branches, three of which enter infinite loops if a stack overflow is detected. These branches are *not* represented in the timing specification, as Picinæ proofs explicitly deal with terminating code. However, timing invariant sets are required to explicitly acknowledge infinite loops, therefore the case where infinitely-looping code is overlooked is impossible.

```
Definition time_of_vTaskSwitchContext (t : trace) (gp : N) (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
    + (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.
```

The proof of this postcondition includes approximately 100 lines of invariant specification and 250 lines of proof code, a linear increase in manual effort compared to instruction count when comparing against the `addloop` example. Throughout the development of this proof, we designed a number of automation tactics that drastically reduced the initially-larger amount of required manual proof work, and we're led to believe that this can be further improved upon in multiple areas.

4 CONCLUSION AND FUTURE WORK

We've demonstrated the ability to formally state and prove tight timing boundaries for arbitrary machine code. Furthermore, we show that Picinæ is capable of proving arbitrary properties of program traces, rather than only output specifications. We plan to demonstrate the expressivity of trace specifications through specifications of cache and pipelining behavior, giving tighter, or even exact, timing specifications for machine code.

Additionally, we intend to implement CFG-aware automation for invariant and proof generation, greatly reducing the required effort of human verifiers and allowing for the fully-formal timing verification process to enter software development pipelines.

Finally, the architecture of Picinæ opens the possibility for reasoning about the timing properties of *program transformations*, due to its inclusion of instruction decoders for ISAs such as RISC-V. An immediate target is our in-progress implementation of a verified CFI rewriter.

REFERENCES

[1] Roberto Amadio, Nicolas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. Certified complexity (cerco). volume 8552, pages 1–18, 08 2014.

[2] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 463–469, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[3] Kevin W. Hamlen, Dakota Fisher, and Gilmore R. Lundquist. Source-free Machine-checked Validation of Native Code in Coq. *FEAST 2019*, 11 2019.

[4] stnolting. The NEORV32 RISC-V Processor - Datasheet.