Foundational Verification of Running-Time Bounds for Interactive Programs

ANONYMOUS AUTHOR(S)

Some important domains of software demand concrete bounds on how long functions may run, for instance for real-time cyberphysical systems where missed deadlines may damage industrial machinery. Such programs may interact with external devices throughout execution, where time deadlines ought to depend on, for instance, sensor readings (e.g. we only scramble to close a valve immediately when a sensor reports that a tank is about to overflow). We present the first software-development toolchain that delivers first-principles proofs of meaningful time bounds for interactive machine code, while allowing all per-application programming and verification to happen at the source-code level. We allow C-like programs to be proved against separation-logic specifications that constrain their running time, and such proofs are composed with verification of a compiler to RISC-V machine code. All components are implemented and proved inside the Coq proof assistant, producing final theorems whose statements depend only on machine-language formal semantics and some elementary specification constructions for describing running time. As a capstone case study, we extended a past verification (of a real microcontroller-based cyberphysical system) to bound time between arrival of network packets and actuation of an attached device.

1 Introduction

University students in computer science are quickly asked to reason about the running times of the programs they write, yet, despite increasing adoption of formal methods, we rarely find programs proved to meet running-time bounds. The problem is hard enough with the asymptotic bounds that populate a traditional introductory algorithms class. However, it is no comfort if a controller within a nuclear power plant triggers a meltdown thanks to a flawed linear-time algorithm that in practice runs for several seconds longer than is safe. Developers of real-time and embedded systems have been dealing in concrete timing deadlines since small, general-purpose computers were feasible to ship.

Where a cadre of today's formal-methods researchers depart from embedded-systems practitioners is in even greater paranoia along a certain dimension: why should we trust the programming tools used to establish time bounds? There may be a bug in a static analysis that establishes bounds automatically. Even worse, Turing taught us that any interesting property of program behavior, timing included, may be understandable only via program-specific reasoning that is hard to automate, motivating the use of tools based on Hoare logic and semi-automated proof… which are even more complex bits of software liable to be full of bugs themselves. And when we are done convincing ourselves of a source-level timing property, will the compiler make a mess of it by generating machine code in an unexpected way? These concerns are the natural timing-centric extensions of motivations for *foundational* verification, where not just applications but also verification and compilation tools are proved together in a common formal framework with machine checking of proofs.

Finally, textbook notions of running-time specification are insufficient, even when we swap out asymptotics for concrete bounds. The reason is that many systems of interest are *interactive*, engaging in input and output with their environments, where proper time bounds depend on inputs received. For instance, a cyberphysical system may be forgiven for taking a while to complete one step of interaction when it is just that an actuator is taking a long time to signal completion via an input routine. As another example, within reason, a network server should be permitted to take longer to return responses when it receives longer requests, though a more refined specification might enforce a policy about not accepting inputs that are so long as to figure in denial-of-service attacks.

In this paper, we report on our project bringing the two dimensions of paranoia (time bounds and foundational guarantees) and handling of input-dependent time requirements together for the first time. (It also seems that, even if we do not restrict attention to foundational proofs, the previous literature on time bounds handles interactivity only in bounding time between two I/O events, not supporting a broader notion of specification.) We present a toolchain for proving time bounds of interactive source programs and then passing them through a compiler guaranteed to preserve time bounds – and all of the reasoning is carried out in the Coq proof assistant, meeting the highest standards of trust-minimization. We introduce a novel specification style that combines I/O event traces with ghost state tracking timing-relevant metrics as counters. When an application has passed through our pipeline, the result is a Coq theorem whose statement only depends on RISC-V machine-code semantics and relatively basic specification constructions needed to explain the relationship between I/O histories and running time.

We follow the long tradition of foundational verifications that start with certain simplifications beyond leading-edge production systems. Most importantly, our measurement of time is *coarse-grained*, and we expose a source-level cost model that could be said to *leak* details of compilation. For the former, our semantics (of source, intermediate, and target languages) track such high-level metrics as numbers of assembly instructions, numbers of memory accesses, and numbers of jumps. For any real processor, those counters allow us to conclude conservative upper bounds on wall-clock time, but those bounds may be very pessimistic. While we are concerned here with microcontrollers where these simple bounds actually do let us derive relatively precise wall-clock bounds (e.g. processors with very minimal use of memory caches), we look forward to future work tracking metrics to support greater precision.

We also ask that source programs be proved against a cost semantics that somewhat telegraphs how the compiler expects to work, revealing costs motivated by the expected mixes of assembly instructions. Indeed, it is an inherent challenge in this domain to allow precise bounds to be proved at the source level without dependence on compiler details, since often compiler optimizations are crucial to meeting deadlines (or, often enough, engineers or certification agencies distrust higher compiler optimization levels and just force system design with laxer deadlines). A common flow is to use source-level analysis to annotate loops with information on how many times they run, which can then be pushed down to assembly code, where detailed worst-case execution time (WCET) analysis is carried out. A downside of that approach is the difficulty of debugging timing issues at the assembly level, whereas we allow timing problems to be found while stepping through Hoare-logic proofs of source files. In any case, it does clearly simplify the programmer's workflow to do per-application reasoning just at the level of source code, and we may find ourselves moving more toward established WCET-analysis styles in future work.

The next section reviews Bedrock2, the preexisting Coq framework that we chose to extend in this work. Then section 3 sketches the whole story of how we extend Bedrock2, from proofs about source programs to guarantees derived about their compilations. The following sections go back to present the pieces in more detail: extended operational semantics (section 4), compiler verification (section 5), and proof of source programs (section 6). We evaluate the framework on a number of case studies and compare it with related work (section 7).

As a preview of our method before we circle back to introducing it, we extended each language's semantics with ghost state tracking time-related metrics. We then found that the different aspects of Bedrock2 adjusted very naturally to the change. We need to tune the source-language semantics based on a conservative understanding of time that may be taken by compiled programs, where the most interesting design challenge came from planning for the possibility of local variables spilled to the stack. Perhaps the central design challenge we want to highlight is of specification patterns for I/O-dependent timing, which we illustrate through examples.

Our Coq implementation is attached as an anonymized supplement.

2 Bedrock2: The Framework We Extend

Bedrock2 is a Coq-based framework for writing C-like programs (with loops, functions, and access to stack and heap memory), verifying their correctness with separation logic, and translating them to RISC-V machine code with a verified compiler. It has been evaluated in two main case studies: a simple embedded system controlling a lightbulb based on network messages [Erbsen et al. 2021], proved down to the level of a Verilog processor description and its initial memory contents of machine code; and a moderately more complex network server speaking a cryptographic protocol [Erbsen et al. 2024], proved just down to the level of machine-code semantics. These case studies belong roughly to the equivalence class of the most thorough *end-to-end, foundational system verifications* carried out to-date. Though the work we report in this paper does not link with verified hardware, by choosing Bedrock2 as our starting point, we can build reasonable confidence that our methods are likely to extend to comprehensive end-to-end proof. Our prototype implementation is still *relatively* end-to-end, as changes we make to source/intermediate languages, compiler, and verification tools remain outside the trusted base when it comes to certifying individual machine-code programs, hence mistakes made therein must be caught by verification.

Our summary of Bedrock2 so far is reminiscent of better-known competitors like the Verified Software Toolchain [Appel 2014], based on the CompCert verified compiler [Leroy 2009]. Bedrock2 is interesting for a few central simplifications compared to CompCert. First, like CompCert, it presents a unified memory model through all languages and compilation stages – but instead of CompCert's structured model based on the C standard, it is instead a *machine language-style flat model*, revealing pointers as machine words. As a result, implementing and verifying memory-management libraries in Bedrock2 is relatively straightforward [Gruetter et al. 2024]. While this difference from CompCert is largely orthogonal to the questions we study in this paper, another difference is front-and-center in the way our proofs work, related to how Bedrock2 approaches *nondeterminism*.

We have called out the importance of timing specifications that depend on a program's I/O interactions. Bedrock2 treats I/O calls as nondeterministic, extending a ghost-state log with their parameters and return values, much in the style of process algebra, and much like how CompCert and others model I/O. However, the CompCert proof approach exerts back pressure on their whole framework toward forced *determinism* of most languages. For instance, the CompCert memory model enforces that memory allocation is deterministic in producing symbolic pointers, where attempting to compare symbolic pointers triggers undefined behavior, to stop programs from noticing real nondeterminism. Somewhat similarly, the results of I/O operations are considered fixed at the start of execution, where deviation from expectations triggers undefined behavior, recovering determinism. In contrast, *Bedrock2 exposes nondeterminism in every language semantics, for both internal choices (like memory allocation) and external choices (like I/O).* This difference actually interacts importantly with the other one we highlighted (concrete pointers) to support verification of memory-management routines within the source language.

CompCert carefully avoids nondeterminism to simplify proofs of semantics-preservation for compiler phases. How does Bedrock2 avoid the pain that CompCert dodges? The key semantics technique is *omnisemantics* [Charguéraud et al. 2023]. Every language above machine code receives a *big-step* semantics, allowing simple compiler-phase proofs that proceed by induction on execution judgments, without the complications of CompCert's family of simulation diagrams, accounting for how small-step executions may go locally out-of-sync but then resynchronize. However, nondeterminism in traditional big-step semantics would be a nonstarter for such proofs. Hence the omnisemantics principle of *big-step semantics that return sets of possible final states,*

rather than just single states. Compiler phases are proved to preserve sets of possible outcomes, which are treated as overapproximations of possible behaviors, supporting compiler phases that resolve nondeterminism (e.g. by substituting in concrete memory-management routines).

Here are a few sample rules from the baseline Bedrock2 semantics. The judgment $\Downarrow$ takes in states of the form $c/m/\ell/t$ for program commands c, memories m (partial functions from machine words to bytes), local-variable environments ℓ, and I/O traces t. An output of the judgment is a set Q of such configurations, overapproximating possible outcomes.

$$\textsc{eval-store}\quad \frac{(x,a) \in \ell \qquad (a+n) \in \mathtt{dom}\ m \qquad (y,v) \in \ell \qquad Q(m[(a+n) := v]/\ell/\tau)}{x[n] = y/m/\ell/\tau \Downarrow Q}$$

$$\textsc{eval-input}\quad \frac{\forall n.\ Q(m/\ell[x := n]/\tau :: \mathtt{IN}\ n)}{x = \mathtt{input}()/m/\ell/\tau \Downarrow Q}$$

$$\textsc{eval-seq}\quad \frac{c_1/m/\ell/\tau \Downarrow Q_1 \qquad (\forall m'\ \ell'\ \tau'.\ Q_1(m'/\ell'/\tau') \Longrightarrow c_2/m'/\ell'/\tau' \Downarrow Q)}{c_1; c_2/m/\ell/\tau \Downarrow Q}$$

Rule EVAL-STORE demonstrates the handling of atomic deterministic steps: simply assert that the modified state satisfies the postcondition Q. Rule EVAL-SEQ demonstrates threading through of postconditions in control flow, requiring an intermediate postcondition Q_1 for the first part of a sequence $c_1; c_2$, such that c_2 must be proved to achieve the overall postcondition Q, starting from any configuration in the intermediate postcondition Q_1. Most interestingly, rule EVAL-INPUT showcases the treatment of atomic *nondeterministic* steps: use universal quantification to ensure that *the postcondition holds for any way the nondeterminism may be resolved*. In this case, we consider all possible return values of `input()`, in each case writing the result into local variable x *and* adding it to the I/O trace τ. Bedrock2 is parameterized over a set of available external calls, which actually receive uniform treatment via a rule that generalizes EVAL-INPUT.

It may be apparent that omnisemantics are quite similar to Hoare logic. The key difference is that loop rules look much more like those of classic big-step semantics, with no need to produce a loop invariant. To prove compiler phases, rule induction over big-step omnisemantics judgments is more straightforward than over Hoare-logic judgments. For instance, there is no need to "compile" loop invariants to construct target-language judgments. There is also no need to consider proof cases for structural rules like a rule of consequence (which need not be included in omnisemantics but can be proved as derived rules). The specifics of omnisemantics are also tuned to be accepted as inductive predicate definitions by proof assistants. While omnisemantics is basically an inductive formulation of weakest preconditions, the more-standard definitions of weakest preconditions fail strict-positivity checks in proof assistants. It is also important that omnisemantics maintain the convention from e.g. separation logic [Reynolds 2002] that a program that may hit undefined behavior *under even a single possible resolution of nondeterminism* has no semantics.

Bedrock2 is also presented as including a program logic, which just means that the baseline omnisemantics of source and intermediate languages are already so close to Hoare triples that a simple wrapper definition suffices to present them that way. A library of notations and tactics is developed to allow more-or-less conventional separation-logic proofs to be carried out on top. While alternative approaches like interaction trees [Xia et al. 2019] reformulate the shape of specifications, Bedrock2 retains the simple structure of Hoare logic, just enriching state with the ghost I/O log.

We hope this recap of Bedrock2's semantics style is convincing, preparing for our later discussion of how we adapt compiler proofs. In contrast to what expectations may be set by CompCert,

we avoid having to deal with different kinds of simulation diagrams or determinizing oracles, instead always proceeding by relatively straightforward rule induction on omnisemantics big-step judgments.

3 Overview of Metrics Logging

Now we are ready to summarize how we extend the full Bedrock2 stack with support for upper-bounding running time. The central idea is addition of ghost state to track different monotonic counters, to complement the prior ghost state logging I/O operations. While this idea is familiar from past work not treating I/O (e.g. Haslbeck and Lammich [2022]), we found it notable how omnisemantics already provides all the plumbing needed to generalize to I/O. There are also new twists coming from the combination of theorems down to machine code and sound foundational verification of bounds solely at the source level (which we believe has not been achieved before).

The basic structure used to track metrics throughout the program logic and compiler is the `MetricLog` object, which is a 4-tuple of integers (*instructions*, *stores*, *loads*, *jumps*). These components refer to the number of RISC-V instructions, memory stores, memory loads (including instruction fetches), and jump instructions respectively. Most of our development can be considered parametric in the details of metrics to track, setting the stage for follow-on work that tracks more detailed metrics as needed for more precise modeling of modern processors. For instance, we might want to store some information on memory-access locality, to predict cache behavior.

We track metrics at this level of granularity because it is coarse enough to be reasonable to work with in practice, while simultaneously being precise enough to derive nontrivial wall-clock time upper bounds on realistic hardware. For instance, for load and store instructions, we can account for memory caches by using worst-case latencies for accessing main memory. For microcontrollers with modest use of caches, these bounds may be precise enough to be useful. To model nontrivial processor pipelines where timing depends on which other instructions are running, we can also use worst-case timing. Again, a natural future-work direction is tracking more granular metrics to improve on such modeling.

It is important to note that we only concern ourselves with *upper bounds* on program metrics. This choice greatly simplifies the handling of optimization stages in the compiler, as the high-level operational semantics need not concern themselves with how much optimization might occur. However, we recognize that in some scenarios, such as in the design of cryptographically safe protocols, matching lower bounds might be desirable. This capability is currently out of the scope of this work.

Recall from section 2 that Bedrock's baseline semantics works with configurations of the form $c/m/\ell/\tau$.

- c is a command, i.e. a piece of Bedrock2 source code.
- m is a memory state, i.e. a partial map from memory locations to values.
- ℓ is a set of local variables, i.e. a partial map from identifiers to values.
- τ is an I/O trace, i.e. the list of input and output events so far.

To implement metrics logging, we augment the configuration type to a 5-tuple $c/m/\ell/\tau/\mu$. The new entry μ is a `MetricLog` object, as detailed above. Like the I/O trace, it can be thought of as the metrics "used so far." We elaborate on this perspective further in sections 4 and 5.

With metric logs in the configuration type, we may now simply use the same proof machinery as correctness proofs, while also allowing postconditions to talk about the difference between starting and ending metrics after an evaluation. Crucially, the postcondition remains an arbitrary Coq expression – which means timing statements have complete access to the memory state, local variables, and I/O trace of the program. Metrics bounds may therefore be as coarse- or fine-grained

as desired, from static constant bounds to highly specific predicates that incorporate the entire I/O history of the program execution.

4 Metrics-Aware Operational Semantics

Now we can present in more detail how we modify the operational semantics of the several languages in the Bedrock2 stack. One challenge is formulating source-level costs that are as tight as possible while remaining accurate, in the face of all compiler optimizations that we plan for.

4.1 Semantics of Expressions

Bedrock2 features a language of pure expressions, which is simple enough that semantics can be deterministic and generally standard. They make a good opportunity to introduce the main ideas of how we model metrics.

The baseline semantics includes a function evalexpr that computes values of expressions. For instance:

$$\mathsf{evalexpr}(m, \ell, \mathrm{literal}\, v) = v$$
$$\mathsf{evalexpr}(m, \ell[x \mapsto v], \mathrm{var}\, x) = v$$

We extend evalexpr with metrics μ as an extra input and output.

$$\mathsf{evalexpr}(m, \ell, \mathrm{literal}\, v, \mu) = (v, C_{\mathrm{LIT}}(\mu))$$
$$\mathsf{evalexpr}(m, \ell[x \mapsto v], \mathrm{var}\, x, \mu) = (v, C_{\mathrm{SET}}(\mu))$$

Functions C_{LIT} and C_{SET} compute worst-case timing effects of all of the ways we anticipate their respective operations might be compiled. At the lower levels of the compiler pipeline, the language semantics can determine the appropriate value of the $C\_$ cost functions directly, by simply writing down the appropriate numbers based on the instructions emitted. In order to enable metrics proofs about Bedrock2 source programs that transfer fully down to the RISC-V level, we then propagate these costs up the compiler pipeline with as tight of bounds as is possible in each compiler stage. This flow leads to various peculiarities in the top-level definitions for the Bedrock2 semantics.

For instance, the precise number of memory accesses for a basic arithmetic operation at the Bedrock2 level depends on whether the operands are already stored in registers. We could simply assume no variables will be stored in registers and give pessimal bounds on each use of a variable, but the resulting bounds would be much looser than desired. Instead, we do slightly better by establishing a naming convention on Bedrock2-level variables. Specifically, any Bedrock2 variable whose name is prefixed with `reg_` is required to be placed in a register by the register allocator, or else the register-allocation compiler phase fails. This way, a Bedrock2-level metrics proof can safely use the tighter bounds for these variables, and the proof will correctly propagate down the pipeline.

In order to implement this strategy in practice, the semantics of all languages in the Bedrock2 stack are parameterized over a value `isReg`, which is a Boolean function on variable names that determines whether they should be considered to be "register variables." At the Bedrock2 level, we use `isRegStr`, which simply checks whether the (string) name starts with `reg_`. Bedrock2 also included an intermediate language FlatImp, where straightline code is flattened into three-address code. At that level, variable names are integers, and we use `isRegZ`, which checks whether the integer is less than 32 (these are the ones that compile down to registers at the RISC-V level).

Since we prove in each compiler phase that `isReg` will never go from true to false for a given variable (note that the converse is safe and manifests in-practice), the $C\_$ cost functions can safely access this information and use it to compute the best known bound for each variable access. This convention is also why the cost functions take variable names as parameters.

Hence, we finally amend the evalexpr rule for variable assignment:

$$\mathsf{evalexpr}(m, \ell[x \mapsto v], \mathrm{var}\, x, \mu) = (v, C_{\mathrm{SET}}(\mathtt{isReg}, x, \mu))$$

4.2 Semantics of Commands

The approach above adapts quite directly to the omnisemantics of commands, where in a sense it picks up compatibility with I/O-trace reasoning "for free." For instance, here is one of the two original rules for loops, and how we adapt it for metrics.

$$\textsc{eval-while-done}\ \frac{(x, 0) \in \ell \qquad Q(m, \ell, \tau)}{(\mathrm{while}\, x\, \mathrm{do}\, c)/m/\ell/\tau \Downarrow Q} \qquad \textsc{eval-while-done-metrics}\ \frac{(x, 0) \in \ell \qquad Q(m, \ell, \tau, C_{\text{WHILE-DONE}}(\mathtt{isReg}, x, \mu))}{(\mathrm{while}\, x\, \mathrm{do}\, c)/m/\ell/\tau/\mu \Downarrow Q}$$

In addition to the `isReg` parameter, each language semantics is also parameterized over a value `phase` (not shown explicitly above), representing the compiler phase to compute metrics bounds with respect to. This mechanism is necessary because in some cases, compiler phases can *increase* metrics costs. In particular, the spilling phase adds a preamble and postamble to every function body and function call, which represents an unavoidable increase in bounds we must account for somewhere.

Therefore, we currently pass a `phase` that is simply either `PreSpill` or `PostSpill`. If the phase is `PreSpill`, then the relevant cost functions in the semantics are artificially more expensive, and the specification of a function call also has added padding. In other words, to prove a metrics bound on a high-level source program, the user must write a proof with respect to a semantics where these operations take longer than they "should." Then, we arrange the compiler pipeline so that `phase` changes exactly once, from `PreSpill` to `PostSpill` exactly at the spilling phase.

At every compiler phase where this parameter does not change, it is effectively invisible, so the same proof goes through. However, at the spilling step, we get a proof obligation that is weaker than the hypothesis; i.e. we only need to prove the metrics bound with respect to a semantics where function calls take the correct amount of time. So we offset the fixed preamble and postamble cost this way, without ever producing a visible effect in the postcondition itself.

4.3 Design Alternatives

One possibly more natural style one might imagine is to attach metrics to the inference rules of the semantics itself, since the semantics corresponds so closely to the execution of a program. As an arbitrary example, it may be more intuitive to write the branch rule reproduced here:

$$\textsc{eval-if-true}\ \frac{\mathsf{evalexpr}(m, \ell, e, \mu) = (v, \mu') \qquad v \neq 0 \qquad c_1/m/\ell/\tau/C_{\mathrm{IF}}(\mu') \Downarrow Q}{(\mathrm{if}\, e\, \mathrm{then}\, c_1\, \mathrm{else}\, c_2)/m/\ell/\tau/\mu \Downarrow Q}$$

as instead an ordinary omni-big-step rule with a single metrics object threaded through, where for each operation and inductive constructor, instead of taking a starting metrics and giving an ending metrics, we simply return an upper bound on how long that step took:

$$\textsc{eval-if-true}'\ \frac{\mathsf{evalexpr}(m, \ell, e) = (v, \mu_1) \qquad v \neq 0 \qquad c_1/m/\ell/\tau \Downarrow_{\mu_2} Q}{(\mathrm{if}\, e\, \mathrm{then}\, c_1\, \mathrm{else}\, c_2)/m/\ell/\tau \Downarrow_{C_{\mathrm{IF}}(\mu_1+\mu_2)} Q}$$

There are two reasons this approach is markedly less convenient. First, we can directly observe that in the EVAL-IF-TRUE rule, the metrics modifications are made in series: first μ to μ', then μ' to

$C_{\text{IF}}(\mu')$, then $C_{\text{IF}}(\mu')$ to the argument of the postcondition. Conversely, in the EVAL-IF-TRUE' rule, the metrics modifications are made in parallel: μ_1 from the expression evaluation and μ_2 from the recursive semantic judgment are added together. In practice, when proving concrete programs, this style would result in highly branching expressions that do not lend themselves to proof automation nearly as well as the highly linear expressions resulting from our actual formulation.

The second advantage to our style is that it mirrors the structure of the inductive proofs used internally in e.g. the compiler-correctness proof. When proving a fact by structural induction on the evaluation judgment, our style automatically relates the metrics of the goal judgment with precisely the desired starting point in the inductive hypotheses. The precise meaning of this difference is more evident in subsection 5.1.

4.4 Machine-Language Semantics

In contrast, the semantics that we inherit and extend for RISC-V machine language follows a small-step style. However, it still uses omnisemantics to characterize nondeterminism. The small-step relation steps to a *set* of possible next states, dependent on how nondeterminism might be resolved. One example of a context where nondeterminism arises is reading from a memory address that is mapped to a device, where the device gets to choose what value is read. Compiler verification must bridge the gap from a higher-level notion of external calls to lower-level interactions, where one external call may actually become multiple accesses to device-mapped addresses. Otherwise, there are no other surprises worth noting to understand how we extended the stack.

5 Verifying the Compiler

The Bedrock2 compiler consists of several *compiler phases*, in each of which a "higher-level" input program p_H is compiled to a "lower-level" output program p_L. Without consideration of metrics, the statement of the correctness of a compiler stage is relatively simple:

$$\left(p_H/m/\ell/\tau \Downarrow Q\right) \rightarrow p_L/m/\ell/\tau \Downarrow Q$$

The version extended to consider metrics is:

$$\left(p_H/m/\ell/\tau/\mu_H \Downarrow Q\right) \rightarrow p_L/m/\ell/\tau/\mu_L \Downarrow \left\{(m',\ell',\tau',\mu_L') \,|\, (m',\ell',\tau',\mu_H') \in Q,\ \mu_L' - \mu_L \leq \mu_H' - \mu_H\right\} \quad (1)$$

We can prove that correctness of compiler phases is still preserved under composition: if we perform a compilation phase from p_H to p_M followed by p_M to p_L, the predicate Q becomes

$$\begin{aligned}
&\exists \mu_M'.\ \left(\exists \mu_H'.\ Q(m',\ell',\tau',\mu_H') \wedge \mu_M' - \mu_M \leq \mu_H' - \mu_H\right) \wedge \mu_L' - \mu_L \leq \mu_M' - \mu_M \\
\Leftrightarrow\ &\exists \mu_H'.\exists \mu_M'.\ Q(m',\ell',\tau',\mu_H') \wedge \mu_L' - \mu_L \leq \mu_M' - \mu_M \leq \mu_H' - \mu_H \\
\Leftrightarrow\ &\exists \mu_H'.\ Q(m',\ell',\tau',\mu_H') \wedge \mu_L' - \mu_L \leq \mu_H' - \mu_H,
\end{aligned}$$

where the equivalence holds in the forward direction because we can arbitrarily choose e.g. $\mu_M' = \mu_L' - \mu_L + \mu_M$ (and the reverse direction is clear).

To greatly simplify, the correctness theorem for each compiler phase is effectively an instance of Equation 1. The full compiler-correctness theorem is then effectively the composition of each of these theorems for compiler phases, as above.

Importantly, the evaluation judgment $\Downarrow$ is a parameter of the compiler stage and may have a different meaning on either side of the implication. In particular, some evaluation judgments for the same language (Bedrock2, FlatImp, or RISC-V) are defined with differing cost semantics. This variation allows us to account for compiler phases that may increase timing bounds, such as register spilling, without making it necessary to modify the postcondition, as described in subsubsection 5.2.4. Accounting for compiler phases that decrease timing bounds, like inlining and dead-code elimination, is automatic, since we only deal with upper bounds.

5.1 Design Alternatives

Notably, the given version of the metrics statement is more general and nicer to work with than this possibly more intuitive and ostensibly simpler formulation.

$$(p_H/m/\ell/\tau/\mu \Downarrow Q) \to p_L/m/\ell/\tau/\mu \Downarrow \{(m', \ell', \tau', \mu_L') \mid (m', \ell', \tau', \mu_H') \in Q \wedge \mu_L' \leq \mu_H'\}$$

The critical difference is that our formulation allows the "starting metrics" of the high-level program and low-level program to differ. In particular, this choice has the technical advantage that we can prove such a statement directly by structural induction on the omni-big-step judgment in the hypothesis. Note that the shape of an induction proof on the "simpler" form gives inductive hypotheses with the same *starting* metrics as the original program, whereas what we need to prove the compiler phase is for the *ending* metrics in the inductive hypotheses to match the starting metrics of the original program.

5.2 Compiler Phases

We modified the proofs of all of the phases to add metrics bounds, following the form described in section 5. Some changes to the compiler itself were required in register allocation, but all other phases of the compiler remain unchanged from the original Bedrock2 code base. We discuss our modifications to each phase in turn.

Recall that the Bedrock2 compiler uses an intermediate language called FlatImp. FlatImp ASTs are parameterized by a type for tracking binders: prior to register allocation, these temporaries are strings corresponding to Bedrock2 variables; in later phases, these are integer indices, where an index less than 32 corresponds to one of the 32 architectural registers of RISC-V, while an index greater than 32 identifies a stack slot in the present function's stack frame.

5.2.1 FlattenExpr phase. This phase flattens nested expressions to use assignments to new temporary variables. Its was the easiest proof to adapt, as something of a sanity-check for our correctness condition. Essentially, we chose the cost functions for nested expressions to match up with the costs introduced by new assignments for their subexpressions.

5.2.2 UseImmediate and DeadCodeElim phases. These phases are exclusively compiler optimizations, so no interesting innovations were needed. The only necessary changes were to thread metrics through the existing correctness proofs, verifying along the way that the metrics cannot inadvertently increase.

5.2.3 RegAlloc phase. The register-allocation phase of the compiler is responsible for assigning the source-level variables of Bedrock2 to registers. While Bedrock2 (and indeed the start language of this phase, `FlatImp.stmt string`) has an unbounded number of temporaries, the RISC-V processor has only 32 architectural registers, so some variables may need to be stored on the stack instead. Register allocation is responsible for determining an assignment of `string` temporaries to registers and stack slots that provides good performance in the generated code.

Register allocation in Bedrock2 transforms the binders from strings to integers, with integers below 32 corresponding to registers and integers above 32 used as abstract handles to stack slots. Importantly, the register-allocation phase does not modify the FlatImp AST, apart from changing the string temporaries to explicit registers. Instead, the emitted AST uses the same operations on registers and stack slots. The spilling phase (next in the pipeline) is responsible for emitting explicit load and store operations for temporaries stored on the stack.

Following prior work on verified compilers (e.g. CompCert [Leroy 2009]), the Bedrock2 register allocator uses *translation validation* [Pnueli et al. 1998], in which the algorithm that maps source temporaries to RISC-V registers and stack slots is not itself verified but instead has its output

checked by a verified checker. The verified checker confirms that the mapping of source temporaries to registers and stack slots respects the liveness analysis of the program, and we prove in Coq that if the checker returns true, the source and register-allocated ASTs have the same semantics.

Modifications for metrics. Our chief modification to the register-allocation phase is to update the register-allocation procedure to attempt to respect source-level directives about which temporaries should go in registers. As discussed in section 4, we modified the Bedrock2 source language to allow the programmer to annotate a source variable to say that it should be placed in a register; the source programmer then enjoys lower metric costs for operations on that variable. Currently, our annotation scheme simply requires the source programmer to prefix a variable's name with `reg_` to indicate that it should go in a register. Correspondingly, the definition of `isRegStr` (subsection 4.1) simply checks for the presence of this prefix.

We modified the register-allocation algorithm to prioritize placing these register variables in RISC-V registers (i.e., assigning them to temporaries with values less than 32). The register allocator does not guarantee that it will find such an assignment; indeed, no such guarantee is possible since nothing prevents the source programmer from defining a program with more than 32 simultaneously live register variables. Conversely, the register allocator may choose to place non-register variables in registers, should they be available. Such placements do not affect metrics bounds since the source semantics provide an upper bound on the execution costs of programs, so the compiler is free to choose a faster implementation.

We further modified the verified checker to check that all source register variables are indeed assigned to architectural registers, i.e., are assigned to abstract registers with indices less than 32. If the checker finds a mapping of a source register variable to a target stack slot, it will fail and print an error, since further compilation could result in an executable whose costs are greater than the bound proved on the source program. With this modification, we were able to establish preservation of metrics bounds in register allocation with only straightforward, administrative modifications to the Coq theorem and proof.

In none of our experiments so far did compilation fail due to inability to place requested variables in registers. The task is significantly easier for RISC-V, with 32 general-purpose registers, than it would be for e.g. classic x86.

5.2.4 Spilling phase. Once register allocation is complete, the spilling phase is responsible for inserting explicit load and store instructions for stack variables, as required by RISC-V's addressing modes. Spilling also inserts loads and stores to save and restore caller-saved registers for function calls. The prior Bedrock2 proof of correctness for spilling established an invariant that after spilling, all temporaries that appear in the AST are registers (i.e., have values less than 32).

Modifications for metrics. We did not modify the implementation of the spilling phase, since no changes were required to account for metrics bounds.

The correctness proof for spilling required somewhat significant modifications to account for metrics. Since our cost semantics are parameterized by `isReg`, we do not need special machinery to reason about the extra costs of operations on stack-allocated variables—in fact, the additional cost at the source level is determined by the costs of the loads and stores added during spilling. It is thus relatively straightforward to add metrics proofs that account for the costs of spilled vs. register operations.

On the other hand, the costs of saving and restoring caller-saved registers do not arise automatically from our register-aware cost semantics. Instead, we must additionally parameterize the cost semantics by whether spilling has occurred or not—we add a flag `phase` which can be either

`PreSpill` or `PostSpill`. The `PreSpill` costs add a conservative upper bound to all function calls on the number of loads and stores required to save and restore all caller-saved registers.

Such augmentation is both necessary and convenient for our framework, in that it allows metrics bounds at the highest level of abstraction to be propagated unchanged down to the RISC-V level. Since the spilling phase adds boilerplate code around every function, the only way to account for the increase in upper bounds is to hardcode this difference into the cost semantics – but this behavior is actually an advantage, as it means the programmer can think entirely about high-level semantics, and the minutiae of what happens in the compiler internals are handled automatically.

5.2.5 FlatToRiscv phase. The final phase of the compiler, FlatToRiscv, compiles FlatImp programs with all variables in registers (`FlatImp.stmt Z`) to RISC-V instructions. The post-spilling FlatImp AST, with all temporaries in registers, is low-level enough that the translation to RISC-V is a simple recursive traversal of the AST. The main complexities in this pass come from handling the various addressing modes and operand formats of RISC-V, and transforming FlatImp's structured control flow into jumps and conditional branches. The proof for this pass in Bedrock2 is quite long and involved, mostly due to conceptually uninteresting details relating to control flow and data layouts.

Modifications for metrics. We did not modify the implementation of the FlatToRiscv phase. However, the correctness proof did require modifications to add metrics bounds. In particular, this phase ultimately determines the costs for each FlatImp AST node, since those costs count RISC-V operations. Once we added these costs, the proofs required some administrative modifications to solve the metrics goals. However, since metrics goals in this phase typically compare one FlatImp AST node to a small number of RISC-V instructions, these goals could typically be dispatched straightforwardly with tactic automation.

6 Source-Level Timing Proofs

6.1 Compatibility and Interfacing with Non-Metrics Proofs

One advantage of the way our metrics framework is structured is that source-program proofs have complete freedom as to what extent to employ it. Indeed, the metrics-instrumented proof framework can be used for correctness proofs that do not say anything about metrics at all. If the program specification does not mention anything about metrics, then no metrics proof obligations will be generated, and the proof will look identical to a proof written without the framework.

As a result, the process of turning an existing correctness proof into a proof-with-metrics is very streamlined. If the specification is left untouched, the conversion process is typically limited entirely to renaming auto-generated variable names (or perhaps not, if the original proof was robust enough) and occasionally moving some subgoal proofs around due to differing lemma structures.

Nevertheless, it is sometimes desirable for non-metrics proofs to coexist in the same codebase, so that metrics developments can live side-by-side with unrelated endeavors that do not need them. We therefore separate the program logic, semantics, weakest-precondition predicate, and loop lemmas into separate modules, compatible with the rest of the codebase. While this structure results in some code duplication between the metrics and non-metrics versions of these modules, it allows timing proofs to be merged into the main branch without affecting existing proofs.

Another side benefit of structuring the framework this way is that correctness and timing proofs can often work off of each other in tandem. As a simple example, proving that a loop terminates (Bedrock2's Hoare logic already enforced total correctness) and proving a time bound on a loop can reuse the same machinery: the same decreasing measure used for the termination proof can appear in the timing portion of the loop invariant, since they are highly conceptually similar (in both cases, we want an upper bound on how much more looping can occur).

```
Definition ipow := func! (x, e) ~> ret {
  ret = $1;
  while (e) {
    if (e & $1) { ret = ret * x };
    e = e >> $1;
    x = x * x
  }
}.
```

Fig. 1. Exponentiation by squaring

6.2 Proving Metric Bounds

Adding metrics to a fully straightline program proof is nearly fully automated. The straightline tactic used for correctness proofs will automatically carry metrics through purely straightline code, thanks to the structure of the definition of the semantics as described in subsection 4.3. The only additional task for the programmer is to discharge the final metrics goal, which is trivial via the automation described in subsection 6.3.

Adding metrics to slightly more complicated programs is quite easy. For example, the sample program in Figure 1, which raises an integer `x` to the e-th power by repeated squaring, can be shown to have a time bound of some constant plus another constant times the most significant bit of the exponent. The difference between the correctness proof and the correctness-plus-metrics proof is only switching the framework from the non-metrics version to the metrics version, adding the metrics bound in the specification, and 13 real lines of proof code (plus one arithmetic lemma about most significant bit).

By way of example, we show two modifications of specifications to existing programs that add time bounds. First we again look at integer power, whose specifications without and with time bounds are given in Figure 2. This program is simple enough that we prove the tightest possible bounds that can be proved in the framework. The time bound is given in terms of the most significant bit of the input, which is one off from its base-2 logarithm. The specification itself simply adds a clause to the postcondition, describing an upper bound on how much the metrics log could have changed.

The second example comes from the much larger-scale proof of concept described in detail in subsection 6.4. The specific example in Figure 3 comes from a function in the lowest-level driver. It performs a basic write operation by first busylooping until an I/O device is ready. Each query to the devices leaves an entry in the I/O trace; therefore, the specification can be written in terms of the length of the trace to account for the amount of time spent looping. In particular, our metrics specification here is a constant bound for the non-loop portion of the function plus a multiplicative bound with respect to the length of the (high-level abstracted) trace.

6.3 Proof Automation

We have developed a moderately large array of tactics to streamline the process of discharging metrics goals as much as possible. We briefly summarize them here.

At the lowest level, the tactics that operate directly on metric logs – namely, `unfold_MetricLog` and `simpl_MetricLog` – culminate in `solve_MetricLog`, where after the metric log objects are sufficiently unpacked, we simply apply the linear-arithmetic solver `blia` to solve the resulting systems of linear inequalities automatically. Not only is the programmer freed from spelling out arguments that are not illuminating, but an amusing method of confirming tightness of bounds is

```
(* spec without metrics *)

#[export] Instance spec_of_ipow : spec_of "ipow" :=
  fnspec! "ipow" x e ~> v,
  { requires t m mc := True;
    ensures t' m' mc' := unsigned v = unsigned x ^ unsigned e mod 2^64
  }.


(* definitions for metrics *)

Definition initCost := {| instructions := 12; stores := 2; loads := 13; jumps := 0 |}.
Definition iterCost := {| instructions := 76; stores := 16; loads := 98; jumps := 2 |}.
Definition endCost :=  {| instructions := 6; stores := 1; loads := 9; jumps := 1 |}.

Definition msb z := match z with
                    | Zpos _ ⇒ Z.log2 z + 1
                    | _ ⇒ 0
                    end.

(* spec with metrics *)

#[export] Instance spec_of_ipow : spec_of "ipow" :=
  fnspec! "ipow" x e ~> v,
  { requires t m mc := True;
    ensures t' m' mc' := unsigned v = unsigned x ^ unsigned e mod 2^64 ∧
      (mc' - mc <= initCost + (msb (word.unsigned e)) * iterCost + endCost)%metricsH
  }.
```

Fig. 2. Comparison of the specification of exponentiation by squaring with and without metrics

enabled: keep tweaking the specification with tighter and tighter bounds until automated proof no longer succeeds. (The failure does manifest in a way that is helpful for understanding *why* it occurs, focusing in on one formula of linear arithmetic that could not be proved.)

Over the course of a program proof, a metrics object will often accumulate a long chain of addition operations, since each straightline command adds on to the previous metrics. Crucially, these are highly linear and non-branching chains, as explained in subsection 4.3. We can therefore employ a tactic `flatten_MetricLog`, which turns these long chains of additions into a single one, combining all constants. This measure is one of several to boost practical performance, making the linear-arithmetic-solver strategy viable for real and complicated programs.

At a higher level, we also have several tactics that deal with the cost functions denoted $C\_$ in this paper. Notably, these functions are often deeply nested as a result of the aforementioned long chains of additions, and internal details related to the Coq kernel cause severe performance issues at `Qed`-time when they are unfolded naively. We therefore provide a carefully crafted `cost_unfold` tactic, which performs these unfolds in a manner that placates the kernel.

Finally, we provide some convenience tactics `cost_solve` and `cost_hammer`, which perform necessary unfolding and destructing of these cost functions and then pass the goals to several

```
(* function definition *)

Definition spi_write := func! (b) ~> busy {
    busy = $-1;
    i = $patience; while i { i = i - $1;
      io! busy = MMIOREAD($0x10024048);
      if !(busy >> $31) { i = i^i }
    };
    if !(busy >> $31) {
      output! MMIOWRITE($0x10024048, b);
      busy = (busy ^ busy)
    }
  }.


(* spec without metrics *)

Global Instance spec_of_spi_write : spec_of "spi_write" := fun functions ⇒ forall t m b mc,
  word.unsigned b < 2 ^ 8 →
  MetricWeakestPrecondition.call functions "spi_write" t m [b] mc (fun T M RETS MC ⇒
    M = m ∧ exists iol, T = t ;++ iol
    ∧ exists ioh, mmio_trace_abstraction_relation ioh iol
    ∧ exists err, RETS = [err] ∧ Logic.or
      (((word.unsigned err <> 0) ∧ lightbulb_spec.spi_write_full _ ^* ioh
        ∧ Z.of_nat (length ioh) = patience))
      (word.unsigned err = 0
        ∧ lightbulb_spec.spi_write word (byte.of_Z (word.unsigned b)) ioh)).


(* definitions for metrics *)

Definition mc_spi_write_const := mkMetricLog 348 227 381 204.
Definition mc_spi_mul := mkMetricLog 157 109 169 102.

(* spec with metrics *)

Global Instance spec_of_spi_write : spec_of "spi_write" := fun functions ⇒ forall t m b mc,
  word.unsigned b < 2 ^ 8 →
  MetricWeakestPrecondition.call functions "spi_write" t m [b] mc (fun T M RETS MC ⇒
    M = m ∧ exists iol, T = t ;++ iol
    ∧ exists ioh, mmio_trace_abstraction_relation ioh iol
    ∧ exists err, RETS = [err] ∧ Logic.or
      (((word.unsigned err <> 0) ∧ lightbulb_spec.spi_write_full _ ^* ioh
        ∧ Z.of_nat (length ioh) = patience))
      (word.unsigned err = 0
        ∧ lightbulb_spec.spi_write word (byte.of_Z (word.unsigned b)) ioh)
        ∧ (MC - mc <= mc_spi_write_const + Z.of_nat (length ioh) * mc_spi_mul)%metricsH).
```

Fig. 3. SPI write example

of the lower-level metric-log-solving tactics. Since we provide a wide array of tactics of varying strength, we are able to choose the appropriate performance/convenience tradeoff for any given metrics subgoal.

6.4 Full-Stack Verification of Responsiveness to Network Requests

We equipped the Bedrock2 lightbulb-controller [Erbsen et al. 2021] software stack with concrete bounds on the computation required to handle a command: about 10 million instructions, plus retrying accesses to peripheral devices when they are busy. This bound is rather loose due to adding up the maximum cost of each language construct without regard for context, but even so it is tight enough to satisfy the exercise of verifying an application-relevant latency bound (specifically, the 40ms latency implied by reasonable assumptions about the peripherals, bus hierarchy, and instructions-per-cycle of the Rocket-derived [Asanovic et al. 2016] FE310 microcontroller is still faster than the mechanical response times of the electromechanical relay and the lightbulb it controls). We will now describe the components of the software stack, the specifications that budget their contribution to the overall execution time, the system-level specification, and the integration proof.

The Bedrock2 lightbulb system is intended to serve as a stand-in for control systems of industrial appliances designed for simplicity and reliability. A single processor core performs input and output using peripheral-control registers mapped at fixed locations in the address space. Interrupts are disabled; instead a top-level event loop poll peripherals when waiting for input. As Bedrock2 C semantics only support programs that always terminate, this top-level loop is written in machine code (using Coq's notation system for assembly-like syntax but with ad-hoc label-offset calculation). The body of the loop and the initialization code that runs before it are written in Bedrock2 C, and the machine code generated by the Bedrock2 compiler relies on invariants set up and maintained by the assembly wrapper. The source code falls into three board categories: chip-peripheral drivers, the ethernet driver, and lightbulb-specific logic for handling network packets. All three are relevant to the worst-case running-time bound, all three perform I/O, the first two contain loops, and the last two manipulate variable-size data in memory.

6.4.1 System Specification. For handling each request, the specification proven in past work covers the reading of the packet from the network controller, computing on its content to figure out the appropriate response, and enacting it. This sequence of events is naturally captured as a regular-expression-like predicate P on MMIO traces. Our addition is to bound the worst-case metric cost associated with producing these events (modulo polling). However, without metrics, the system-level theorem without metrics has been stated as an *invariant* that each state individually must satisfy: the I/O trace is always (a prefix of) some number of repetitions of handling a request, $P' +\!\!+ P^*$. An invariant relating metrics to I/O can be meaningful as a worst-case cost specification, but it would capture *amortized* costs: e.g., any n requests can be processed in $k \cdot n$ instructions. As the use case our system is a stand-in for may not be able to tolerate every millionth request taking a million times longer than usual, we present the system-level specification in a different format.

Intuitively, we want to say that, if the system is just about to receive a request and has already accumulated I/O trace t and metrics μ, it shall proceed until it has a trace $\Delta t +\!\!+ t$ (our actual implementation extends traces at the beginning rather than the end, to line up better with standard list operations) such that Δt satisfies the trace predicate for handling at that request, and metrics μ' at that point must satisfy $\mu' - \mu \leq C_{\text{req}}$. A quantifier-free invariant could reference either μ or μ' but not both. The standard trick of existentially quantifying the future state (creating an invariant that says that the system will reach a good state soon enough) is also not straightforwardly applicable because P specifies an entire request-action interaction and would not accept a proper suffix of it.

Considering the suffix-completion of P instead would unfortunately weaken the worst-case-cost aspect of the specification to guarantee only the ability to complete *some* suffix of P in time.

Lacking a straightforwardly applicable general framework, we instead formalize the bounded-cost-reactive behavior of the system as an ad-hoc transition system in the style of linear temporal logic. Handling one entire request forms (by executing many instructions) *one transition* of the specification, and we assert that it is an invariant always throughout these steps that the next step will complete with acceptable cost. To accommodate start-up code for which we have not proven worst-case bounds, we wrap that entire specification in the eventually operator: eventually the system will boot up and then handle each successive request as fast as specified. The last notion is captured using an omni-small-step combinator defined in terms of `always` and `eventually`:

```
Definition successively (R : State → State → Prop) : State → Prop :=
  always (fun s post ⇒ eventually step (fun s' ⇒ R s s' ∧ post s') s) (fun _ ⇒ True).
```

For the per-step specification R, we use the metric-adapted version of the specification of the body of the event-handler loop from the original case study, minus information about local variables. It is called `handle_request_spec` and shown in Figure 4. It relies on the `loop_cost` definition that specifies how long execution of one loop iteration may take at most, expressed as a function of the packet length *and* the I/O interaction trace length. The latter argument is needed because in the protocol used to talk to the network-interface card, reading each byte of the network packet has to start by polling the SPI peripheral to know whether it has received the next byte from the network card, and we do not know (or make any assumptions about) how long the response takes. Instead, we simply allow our implementation to spend an execution cost of `mc_spi_mul` per I/O interaction that was necessary to read the packet. The number of instructions presented at the beginning of subsection 6.4 was computed by evaluating the `loop_cost` definition for the largest supported packet length (and a 0 `trace_length`).

The conclusion of the final end-to-end theorem we proved is formalized as a straightforward combination of `successively` and `handle_request_spec` and preceded by `eventually` to account for the fact that we did not prove bounds for the bring-up code. It is shown at the bottom of Figure 4. This theorem also assumes bring-up-related and administrative properties (that do not mention semantics of any language) about `initial_state`, similarly to its metrics-free counterpart in Erbsen et al. [2021].

6.4.2 Component Specifications. The compiler specification is adapted as in section 5, and the specification of the machine-code loop is adapted similarly to specifications of looping constructs generated by the compiler. Specifications of Bedrock2 C functions are adapted as discussed in section 6. After some experimentation with specifying the SPI-peripheral driver (see Figure 3), the same specification pattern was applied uniformly throughout the case study.

We found the costs of some subroutines to be best characterized in terms of the runtime input they receive. For example, cost of reading a packet from the network card takes time proportional to the length of that packet. However, as we sought to establish true worst-case bounds, we also included the maximum range of that variation in the specification of the function: Thus, the postcondition of the ethernet receive function which returns the number of `bytes_written` is as follows:

```
exists recv buf,
m =* bytes p_addr recv * bytes (word.add p_addr bytes_written) buf * R ∧
lan9250_recv _ recv ioh ∧
word.unsigned bytes_written + length buf = 1520 ∧
length recv = word.unsigned bytes_written ∧
```

```
Definition loop_cost(packet_length trace_length: Z): RiscvMetrics :=
  (60 + 7 * packet_length) * mc_spi_xchg_const +
  lightbulb_handle_cost + trace_length * mc_spi_mul +
  loop_compilation_overhead.

Definition handle_request_spec(t t': trace)(mc mc': RiscvMetrics) :=
  exists dt, t' = dt ++ t ∧
  exists ioh, metric_SPI.mmio_trace_abstraction_relation ioh dt ∧ (
    (* Case 1: Received packet with valid command: *)
    (exists packet cmd,
        (lan9250_recv packet +++ gpio_set 23 cmd) ioh ∧
        lightbulb_packet_rep cmd packet ∧
        (mc' - mc <= loop_cost (length packet) (length ioh))) ∨
    (* Case 2: Received invalid packet: *)
    (exists packet,
        (lan9250_recv packet) ioh ∧
        not (exists cmd, lightbulb_packet_rep cmd packet) ∧
        (mc' - mc <= loop_cost (length packet) (length ioh))) ∨
    (* Case 3: Polled, but no new packet was available: *)
    (lan9250_recv_no_packet ioh ∧
        (mc' - mc <= loop_cost 0 (length ioh))) ∨
    (* Case 4: Received too long packet *)
    (lan9250_recv_packet_too_long ioh) ∨
    (* Case 5: SPI protocol timeout *)
    ((TracePredicate.any +++ spi_timeout) ioh)).

Theorem metric_lightbulb_correct: forall (initial : MetricRiscvMachine) R,
    valid_machine initial →
    getLog initial = [] →
    regs_initialized.regs_initialized (getRegs initial) →
    getNextPc initial = word.add (getPc initial) (word.of_Z 4) →
    getPc initial = code_start ml →
    (program RV32IM (code_start ml) (fst (fst out)) * R *
       LowerPipeline.mem_available (heap_start ml) (heap_pastend ml) *
       LowerPipeline.mem_available (stack_start ml) (stack_pastend ml))%sep
      (getMem initial) →
    subset (footpr (program RV32IM (code_start ml) (fst (fst out))))
      (of_list (getXAddrs initial)) →
    eventually riscv.run1
      (successively riscv.run1
         (fun s s' : MetricRiscvMachine ⇒
            handle_request_spec (getLog s) (getLog s')
              (getMetrics s) (getMetrics s'))) initial.
```

Fig. 4. End-to-end theorem and its two most important supporting definitions

Table 1. Overview of changes to compiler phases to add metrics-logging support

compiler phase	code?	semantics?	source diff
FlattenExpr	no	yes	27
UseImmediate	no	no	125
DeadCodeElim	no	no	71
RegAlloc	yes	yes	90+115
Spilling	no	yes	403
FlatToRiscv	no	yes	133

```
((mc' - mc <= (55+7*bytes_written)*spi_xchg_const + (length ioh)*spi_poll_cost))
```

6.4.3 Proof Techniques and Proof Reuse. We use the following proof rule (coinduction principle) to establish successive completion of requests required by the top-level specifications:

```
forall R s invariant,
invariant s →
(forall s, invariant s → eventually step (fun s' ⇒ R s s' ∧ invariant s') s) →
successively R s.
```

To instantiate this proof rule, we reuse the memory- and local-state invariant from the non-metrics-equipped lightbulb case study (removing the trace-related part that is now handled separately). Even though we completely changed the meaning of the top-level specification of the event loop, we were able to reuse the low-level invariant and the proof scripts that drive the symbolic execution of that machine code almost entirely, only making nontrivial changes in the final proof of the (new) postcondition. In that final machine-code-level proof, we relied on tactic automation for checking our metric accounting, very similarly to what is described for C-like code in subsection 6.3.

Either way, the key step is instantiating `eventually step` with the low-level execution judgement in the postcondition of the compiler-correctness theorem. Then we use the omnisemantics weakening rule to switch to proving that the postcondition of the compiler-generated code (which is in turn stated in terms of the postcondition of the source code in the style we have already discussed) implies `R:=handle_request` and the memory-layout invariant.

Conclusion. These system-theorem changes were devised, formalized, debugged, and proved within two person-days of work, changing about 1000 lines of code. Even though the introduction of metric bounds did require a shift of perspective in the system-level specification, the specification style we chose for conveniently including metric bounds is well within the design space that would be on the table without considering metrics, and our new system-spec-level proofs are about half the size of the previous ones. Anecdotally, the main challenges in adapting the proofs were similar to those of legacy-software engineering in nature and can be attributed to adapting somebody else's code from years ago, rather than the change itself. Thus, it seems that adding worst-case bounds on cost metrics to a whole-system theorem is not particularly costly.

6.5 Summary of Effort Changing the Compiler and Its Proof

To round out our discussion of engineering effort needed to adapt the original Bedrock2-related artifacts, we show in Table 1 what we needed to change in the compiler and its proof. The only compiler phase that needed implementation changes was RegAlloc, as noted in subsubsection 5.2.3. The semantics of intermediate languages had changes for most phases. We give the total number

of lines of changes to Coq files, as reported by Git. They sum up to just under 1000 lines, which we consider to be a pretty reasonable level for a compiler of this moderate complexity level. The diff for phase RegAlloc is split between changes to the unverified register allocator and its verified translation validator.

7 Related Work

To our knowledge, no past project has supported running-time specifications that may depend on details of I/O interaction with the outside world, not just on sizes of data structures passed as inputs to functions. However, some of the other relatively unusual benefits of our approach are also found in past work, as we now survey.

There has been plenty of work on static analysis to bound running time of programs, for instance on type systems for functional programs with effective inference of linear space [Hofmann and Jost 2003] and then polynomial time [Hoffmann and Hofmann 2010] bounds. The results have been extended and applied to languages as complex as OCaml [Hoffmann et al. 2017]. Such ideas have been combined with advanced type-system features like refinement types to infer time bounds for classic data-structure operations [Wang et al. 2017]. Verified time-preserving compilation to lower-level languages complements these projects, and the past work we have mentioned also rarely tackles functional-correctness verification as well, though some important program properties intertwine functional and timing requirements, as we support in our program-logic-style proofs.

A notable freestanding tool for pushing time bounds through compilation is from Bonenfant et al. [2007], translating the functional source language Hume to assembly. Using an empirically validated model of costs of low-level instructions, they automatically produce bounds that make sense at the source level but are accurate about compiled code.

The earliest work on proof-carrying code in the literature already considered certification of running-time bounds [Necula and Lee 1998], via instrumented semantics of incrementing counters, not too different from our own new operational semantics. However, no certifying compiler was presented that can produce the required assembly-level proof annotations from annotated source programs. Crary and Weirich [2000] added that ingredient, though targeting a typed assembly language rather than annotations to drive Hoare logic-style proofs. They support rich timing specifications tied to data-structure invariants but still no connection to I/O histories. Furthermore, this tradition of certifying compilers retains its usual tradeoffs compared to compiler verification as in our work: building a certifying compiler can be easier than verifying a compiler, but there is always the risk of lurking completeness bugs, where a compiler run produces an invalid certificate that later fails checking.

None of the results just summarized are *foundational*, in the sense of leading to proofs in a minimal, general-purpose logic (as underlie proof assistants). However, a notable example to the contrary is a verified worst-case-execution-time (WCET) analysis for CompCert [Maroneze et al. 2014]. Their analysis statically infers loop bounds on programs in an intermediate language and bounds time by combining that information with costs of basic blocks computed by further static analysis on assembly code. The whole analysis pipeline is automatic, a significant attraction. However, analysis bounds are only constant integers, not even able to depend on input data-structure sizes, let alone I/O interactions.

The Certified Complexity (CerCo) project [Amadio et al. 2014; Ayache et al. 2012] specifies the costs of instructions by instrumenting the source code with code that increases instruction counters, whereas in our approach, we instrument the semantics. The instrumented source code is then verified with Frama-C [Cuoq et al. 2012]. The compiler obtains the execution cost of each basic block by looking at the concrete machine code that it generated. Thus, proof of bounds for specific source programs relies on timing information ported back from compilation results. The compiler

pipeline, including generation of instrumentation code, is fully verified in proof assistants, but analysis of source programs is done with non-foundational tools, whereas our own pipeline produces end-to-end Coq proofs from relational specifications (which may combine functional and timing requirements) to operational semantics of machine code.

Probably most closely related to our work is that of Haslbeck and Lammich [2022] extending the Isabelle Refinement Framework. One relatively shallow difference is that their mechanized proofs (in Isabelle/HOL) bottom out in LLVM programs, not machine code, though we did face additional challenges in spanning that gap, for instance related to the time consequences of compiler-introduced spilling. This work is notable in being the only prior one we are aware of that involves reasoning about nondeterminism. However, the style of nondeterminism was more oriented toward program derivation by stepwise refinement, for instance in postponing decisions on which specific data structures to choose, among those satisfying high-level specifications. In other words, they handled *internal* nondeterminism that gives developers flexibility in planning out programs, while we also handle *external* nondeterminism from I/O interactions and demonstrate new patterns for effective specification-writing in that setting. A satisfying proof innovation in their work, which we might benefit from adopting, is the idea of *abstract currencies* of different metrics to collect during execution, where it is possible to "exchange" more abstract measurements for more concrete ones, for instance first to measure how many times each LLVM instruction is executed and then substitute upper bounds for time taken to run the compiled versions of those instructions.

8 Future Work

While this framework for time bounds on interactive programs is already usable in practice, it still has many exciting directions for further research.

The most immediate direction to take is to improve upon the bounds given by the framework itself. In particular, there are at present several places where our framework makes more conservative estimates than it could. For instance, the buffer added to function calls when performing spilling is currently a static constant amount, whereas the framework has access to several pieces of information (function arity, variables in registers) that could give tighter bounds. Furthermore, the bounds given when evaluating complex expressions assume that all intermediate results are stored in non-register variables, which is in fact unlikely in practice. Both of these cases are examples of places where the framework, although theoretically sound, could see practical improvement.

An orthogonal direction that could also be interesting is to allow the framework also to prove lower bounds on programs, as alluded to in section 3. While the framework is already fully general in the domain of program and compiler-stage specifications, the restriction to upper bounds is baked into the evaluation and program semantics. In principle, it would be possible to provide both upper and lower bounds in these places. However, more work would be necessary to make actual use of the resulting lower bounds, since the compiler has several optimization stages whose effects on lower bounds would be challenging to rigorously evaluate and prove. Nevertheless, a verified optimizing compiler with bounds on both sides would also be a novel development in this area.

References

Roberto M. Amadio, Nicolas Ayache, Francois Bobot, Jaap P. Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. 2014. Certified Complexity (CerCo). In *Foundational and Practical Aspects of Resource Analysis*, Ugo Dal Lago and Ricardo Peña (Eds.). Vol. 8552. Springer International Publishing, Cham, 1–18. https://doi.org/10.1007/978-3-319-12466-7_1

Andrew W. Appel. 2014. *Program Logics - for Certified Compilers.* Cambridge University Press. https://www.cs.princeton.edu/~appel/papers/plcc.pdf

Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The Rocket Chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016), 6–2.

Nicolas Ayache, Roberto M. Amadio, and Yann Régis-Gianas. 2012. Certifying and Reasoning on Cost Annotations in C Programs. In *Formal Methods for Industrial Critical Systems*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Mariëlle Stoelinga, and Ralf Pinger (Eds.). Vol. 7437. Springer Berlin Heidelberg, Berlin, Heidelberg, 32–46. https://doi.org/10.1007/978-3-642-32469-7_3

Armelle Bonenfant, Christian Ferdinand, Kevin Hammond, and Reinhold Heckmann. 2007. Worst-Case Execution Times for a Purely Functional Language. In *Implementation and Application of Functional Languages*, Zoltán Horváth, Viktória Zsók, and Andrew Butterfield (Eds.). Springer, Berlin, Heidelberg, 235–252. https://doi.org/10.1007/978-3-540-74130-5_14

Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2023. Omnisemantics: Smooth Handling of Nondeterminism. *ACM Transactions on Programming Languages and Systems* 45, 1 (March 2023), 5:1–5:43. https://doi.org/10.1145/3579834

Karl Crary and Stephnie Weirich. 2000. Resource Bound Certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '00*. ACM Press, Boston, MA, USA, 184–198. https://doi.org/10.1145/325694.325716

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C. In *Software Engineering and Formal Methods*, George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 233–247.

Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration Verification across Software and Hardware for a Simple Embedded System. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, 604–619. https://doi.org/10.1145/3453483.3454065

Andres Erbsen, Jade Philipoom, Dustin Jamner, Ashley Lin, Samuel Gruetter, Clément Pit-Claudel, and Adam Chlipala. 2024. Foundational Integration Verification of a Cryptographic Server. *Proc. ACM Program. Lang.* 8, PLDI, Article 216 (jun 2024), 26 pages. https://doi.org/10.1145/3656446

Samuel Gruetter, Viktor Fukala, and Adam Chlipala. 2024. Live Verification in an Interactive Proof Assistant. *Proc. ACM Program. Lang.* 8, PLDI, Article 209 (June 2024), 24 pages. https://doi.org/10.1145/3656439

Maximilian P. L. Haslbeck and Peter Lammich. 2022. For a Few Dollars More: Verified Fine-Grained Algorithm Analysis Down to LLVM. *ACM Transactions on Programming Languages and Systems* 44, 3 (July 2022), 14:1–14:36. https://doi.org/10.1145/3486169

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 359–373. https://doi.org/10.1145/3009837.3009842

Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polynomial Potential. In *Programming Languages and Systems*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 287–306.

Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 185–197. https://doi.org/10.1145/604131.604148

Xavier Leroy. 2009. A Formally Verified Compiler Back-End. *Journal of Automated Reasoning* 43, 4 (Dec. 2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

André Maroneze, Sandrine Blazy, David Pichardie, and Isabelle Puaut. 2014. A Formally Verified WCET Estimation Tool. In *14th International Workshop on Worst-Case Execution Time Analysis (2014)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 11–20. https://doi.org/10.4230/OASIcs.WCET.2014.11

George C. Necula and Peter Lee. 1998. Safe, Untrusted Agents Using Proof-Carrying Code. In *Mobile Agents and Security*, Giovanni Vigna (Ed.). Springer, Berlin, Heidelberg, 61–91. https://doi.org/10.1007/3-540-68671-1_5

A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernhard Steffen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–166.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. LICS*. IEEE Computer Society, 55–74.

Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: a functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 79 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133903

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51

(Dec. 2019), 32 pages. https://doi.org/10.1145/3371119