Sharc: Simulator for Hardware Architecture and Real-time Control

Paul K. Wintz*
University of California, Santa Cruz
Santa Cruz, California, USA
pwintz@ucsc.edu

Yasin Sonmez*
University of California, Berkeley
Berkeley, California, USA
yasin_sonmez@berkeley.edu

Paul Griffioen
University of California, Berkeley
Berkeley, California, USA
griffioen@berkeley.edu

Mingsheng Xu
University of California, Santa Cruz
Santa Cruz, California, USA
mxu61@ucsc.edu

Surim Oh
University of California, Santa Cruz
Santa Cruz, California, USA
soh31@ucsc.edu

Heiner Litz
University of California, Santa Cruz
Santa Cruz, California, USA
hlitz@ucsc.edu

Ricardo G. Sanfelice
University of California, Santa Cruz
Santa Cruz, California, USA
ricardo@ucsc.edu

Murat Arcak
University of California, Berkeley
Berkeley, California, USA
arcak@berkeley.edu

ABSTRACT

Tight coupling between computation, communication, and control pervades the design and application of cyber-physical systems (CPSs). Due to the complexity of these systems, advanced design procedures that account for these tight interconnections are paramount to ensure the safe and reliable operation of control algorithms under computational constraints. This paper presents the *Simulator for Hardware Architecture and Real-time Control* (Sharc) to assist in the co-design of control algorithms and the computational hardware on which they are run. Sharc simulates the execution of a user-specified control algorithm on a given processor microarchitecture configuration, evaluating how computational constraints affect the dynamical properties of the closed-loop system. We illustrate the power of Sharc by examples of MPC applied to adaptive cruise control and the stabilization of an inverted pendulum. Sharc can be found at github.com/pwintz/sharc.

CCS CONCEPTS

• **Computer systems organization → Embedded and cyber-physical systems**; **Processors and memory architectures**; *Real-time system architecture*; • **Software and its engineering →** *Software verification and validation.*

KEYWORDS

Microarchitecture simulator, Feedback Control, Real-time systems, Safety

ACM Reference Format:
Paul K. Wintz, Yasin Sonmez, Paul Griffioen, Mingsheng Xu, Surim Oh, Heiner Litz, Ricardo G. Sanfelice, and Murat Arcak. 2025. Sharc: Simulator for Hardware Architecture and Real-time Control. In *28th ACM International Conference on Hybrid Systems: Computation and Control (HSCC '25), May 6–9, 2025, Irvine, CA, USA*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3716863.3718046

1 INTRODUCTION

Cyber-physical systems (CPSs) are engineered systems that incorporate digital sensors, computers, and actuators interacting with physical processes. CPSs are ubiquitous in modern critical infrastructure such as transportation systems, energy delivery, and health care. Typically, a CPS has strong coupling between its computational hardware, physics, and control algorithms. CPU designers, however, usually optimize for generic instruction sets—not for a specific algorithm—whereas control designers typically do not design their algorithms for a particular hardware architecture. Many properties of the physical system, including stability, safety, and liveness, are affected by the latency of computing the next control input. The computational delay depends on the underlying computational hardware on which the control algorithm is run. Thus, to develop CPSs that satisfy demanding design specifications, engineers must jointly consider the computing power and control methods. To address this challenge, this paper introduces a tool called the *Simulator for Hardware Architecture and Real-time Control* (Sharc) that simulates the physical evolution of a CPS and the execution of control algorithms on different computing platforms to incorporate realistic controller delays in the closed-loop simulation. The ability to incorporate accurate computation delays into simulations allows system designers to gain a better understanding of how computational limitations affect the behavior of the system, thereby informing better designs.

To allow for quick and easy installation of Sharc, a Dockerized version of Sharc is provided. To aid in the development of Sharc and implementation of controllers and dynamics using Sharc, the project is configured to support running Docker images in a Devcontainer. A suite of unit tests is included in the Sharc to verify Sharc's internal logic.

*Both authors contributed equally to this research.

HSCC '25, May 6–9, 2025, Irvine, CA, USA

1.1 Problem Setting

Creating tools to analyze CPSs is challenging due to the interactions between computational hardware, physics, and control software. There has been a trend toward consolidating control software components onto shared multicore processors to reduce size, weight, and power while improving performance. The adoption of multicore hardware in practical CPSs, including automotive [19], avionics [17], and medical systems [24], brings many benefits but also introduces a new host of challenges for modeling and analysis. In a multicore processor, some resources, such as the caches, memory bus, and random access memory (RAM), are shared between cores, which affects the timing of the control software in complex ways, making the timing difficult to model and predict.

Beyond traditional CPUs, specialized computing hardware is becoming prevalent in CPSs, including highly-parallelized processors (GPUs), configurable hardware (e.g., field programmable gate arrays, or FPGAs), and domain-specific accelerators (such as Tesla's Full Self-Driving Application-Specific Integrated Circuit [2]). These components are critical for the unique demands of real-time machine learning, computer vision, and other applications. On the other hand, control software is also becoming increasingly complex, with diverse and rapidly changing resource requirements and performance goals [10]. Computation-aware control algorithms require detailed information about the performance of the computational hardware via strong performance monitoring capabilities in order to understand how to safely and effectively optimize the hardware and control algorithms.

Creating tools that can assess and simulate complicated interactions between hardware and software is critical for automotive, avionics, and other domains in which computationally-intensive processes play a significant role at runtime. In particular, it is important to ensure that advanced control algorithms can run on the available computing hardware with high confidence. For example, automotive original equipment manufacturers (OEMs) need to evaluate whether updated software for advanced controllers or perception algorithms can be safely deployed on vehicle models of the previous year. To ensure that advanced control algorithms can run on available computing hardware, regression analysis of candidate controllers should be concurrently tested *in situ* for deployed systems. Some algorithms for achieving robust autonomy, such as model predictive control (MPC), have limited deployment due to insufficient computing power. By using SHARC to simulate, analyze, and optimize controllers and hardware platforms, engineers can design solutions that improve the use of onboard energy and computational sources, allowing cheaper and more efficient implementations of advanced control schemes.

1.2 Literature Review

Prior research has investigated quantifying the computational demands of various control algorithms and mitigating the effects of computational delays. The authors of [3, 4, 6, 8, 22] investigate the effects of the worst-case execution time on the performance of several control algorithms, including linear quadratic control (LQR), model predictive control (MPC), and state-dependent Riccati equation (SDRE) nonlinear control. In particular, linear and nonlinear MPC schemes that preserve stability and performance under computational delays are presented in [12, 14, 31] with a brief introduction given in [14, Section 7.6]. However, none of these schemes provides a tool or methodology that explicitly accounts for the effects of many kinds of computational hardware, including hardware that has not yet been fabricated. In contrast to these works, SHARC is able to analyze the effects of diverse microarchitectures—including hypothetical configurations—on the performance of the closed-loop system. This allows the user to conduct both microarchitecture design exploration and control design optimization.

CPU manufacturers and computer architecture researchers rely on microarchitectural simulation to explore the hardware design space, prototype new hardware ideas, and evaluate application performance on hypothetical hardware. Microarchitecture simulators model the internal hardware architecture of CPUs, including branch predictors, instruction fetch and decode units, functional units, instruction schedulers, and memory subsystems with multiple levels of caches. Some noteworthy microarchitecture simulators are *gem5* [5], *ChampSim* [13], *ZSim* [25], *MARS* [30], and *Sniper* [9]. While these simulators enable accurate application simulation and performance modeling, they are not designed to simulate controllers interacting in a closed-loop with a physical system. In particular, they cannot be directly integrated into a model where the controller interacts with a physics simulation since microarchitecture simulators do not incorporate methods for synchronizing the passage of time in dynamical simulations with the execution of the controller code inside the microarchitecture simulator.

Prior works have proposed tools for testing control algorithms interacting with physical systems via co-simulations and via hardware-in-the-loop (HIL). Co-simulation tools [15, 29] provide simulations of the CPU capable of modeling hardware events such as interrupts, but they do not provide a cycle-accurate timing simulation, which is necessary to ensure the safe and reliable operation of control algorithms under computational constraints. Hardware-in-the-loop simulation integrates real-time hardware into a simulated environment, enabling realistic validation of control algorithms. The system's physics are simulated on a real-time platform, interacting in a closed loop with actual hardware, such as an embedded controller. HIL is widely used in automotive and aerospace applications to evaluate controllers for autonomous vehicles, flight control systems, and industrial automation, ensuring robust performance before deployment [11, 20]. However, traditional HIL setups rely on fixed hardware, limiting flexibility in exploring different architectures or optimizing control algorithms under varying constraints. SHARC overcomes these limitations by using a reconfigurable cycle-accurate microarchitectural simulator in a closed-loop with a simulation of the system's physics.

The remainder of the paper is structured as follows. Section 2 introduces the modeling framework for the physics and the computational hardware, as used by SHARC. Section 3 describes the implementation and basic usage of the simulator, with a serial execution mode described in Section 3.1 and a parallel mode described in Section 3.2. Two examples are presented in Section 4. In particular, section Section 4.1 contains an example of adaptive cruise control for longitudinal vehicle control via linear MPC and Section 4.2 demonstrates simulation of an inverted pendulum system stabilized by a nonlinear MPC controller. To conclude, Section 5 describes future research directions.

2 MODELING

In this section, we introduce our modeling framework for the physics, controller, and computational hardware of a CPS, and their interconnection.

2.1 Physics and Controller

A physical system controlled by a controller is typically called a *plant*. The physics of a plant are often modeled as a differential equation, which we write as

$$\dot{x} = \tilde{f}(t, x, u, w), \tag{1a}$$

$$y = h(x, u, w), \tag{1b}$$

where the plant has state $x \in \mathbb{R}^{n_x}$, control input $u \in \mathbb{R}^{n_u}$, output $y \in \mathbb{R}^{n_y}$, and a disturbance $w \in \mathbb{R}^{n_w}$. The disturbance (or *exogenous input*) is given as a function $t \mapsto w(t) \in \mathbb{R}^{n_w}$ for all $t \geq 0$. Although physical systems are nicely represented mathematically by differential equations, most methods for numerically simulating continuous-time systems use discretization. In Sharc, we use a discrete model on an evenly-spaced time grid with period $T > 0$, defined by $t_k := kT$ for each $k \in \mathbb{N}$. We write the discrete dynamics of the plant as

$$x_{k+1} = f(t_k, x_k, u_k, w(t_k)), \tag{2}$$

where f is a discretization of the physics with sample time T, and $x_k := x(t_k)$, $u_k := u(t_k)$, and $y_k := y(t_k)$ for each k. Figure 1 illustrates the discretization of the continuous-time physics in (1).

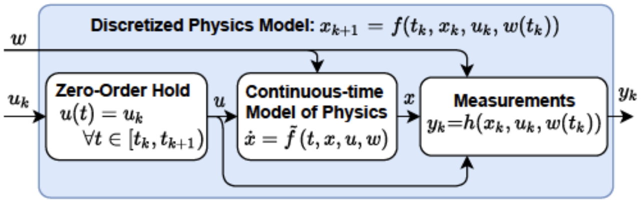
Discretized Physics Model: $x_{k+1} = f(t_k, x_k, u_k, w(t_k))$
w
u_k
Zero-Order Hold
$u(t) = u_k$
$\forall t \in [t_k, t_{k+1})$
u
Continuous-time
Model of Physics
$\dot{x} = \tilde{f}(t, x, u, w)$
x
Measurements
$y_k = h(x_k, u_k, w(t_k))$
y_k

Figure 1: A continuous-time physics model as in (1) can be discretized by interpolating a discrete input u_k using zero-order hold and sampling the y_k output at sample times t_k.

The input u is generated by a control algorithm that evaluates a control function $(t, y) \mapsto g(t, y) \in \mathbb{R}^{n_u}$. We assume that the control values are only updated at sample times. When discretizing (1), as shown in Fig. 1, Sharc interpolates the input between consecutive time steps t_k and t_{k+1} using zero-order hold. In particular, $u(t) = u_k$ for all $t \in [t_k, t_{k+1})$, where u_k is the value of the control input received from the controller at t_k. The output y_k represents periodic sensor measurements.

Users of Sharc can implement the physics of their system in two ways. If they are starting with a continuous-time system, they can provide $\tilde{f}$ to have Sharc automatically generate and evaluate the discretization f via numerical integration. Alternatively, users can provide f directly if they wish to implement other types of models, such as hybrid systems or stochastic differential equations.

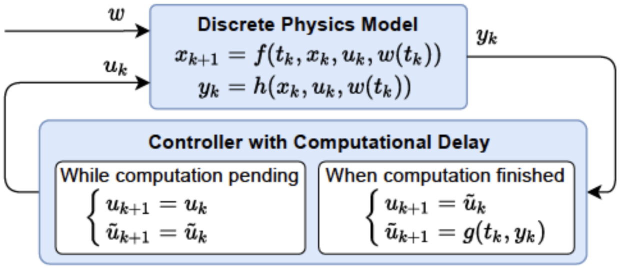
w
u_k
Discrete Physics Model
$x_{k+1} = f(t_k, x_k, u_k, w(t_k))$
$y_k = h(x_k, u_k, w(t_k))$
y_k
Controller with Computational Delay
While computation pending
$\begin{cases} u_{k+1} = u_k \\ \tilde{u}_{k+1} = \tilde{u}_k \end{cases}$
When computation finished
$\begin{cases} u_{k+1} = \tilde{u}_k \\ \tilde{u}_{k+1} = g(t_k, y_k) \end{cases}$

Figure 2: A feedback diagram of the closed-loop model of the physics with a controller that has computational delays. After a control computation is started but before its computation time has elapsed in the simulation, it is stored as $\tilde{u}_k$. The values of u and $\tilde{u}$ are held constant until the computation is finished.

2.2 Interaction between Physics and Controller with Computation Delays

In an idealized system, the controller g provides the next control value $g(t_k, y_k)$ immediately at time t_k. In realistic systems, however, computing the control update takes time, so the new control value is not available until after some computation delay $\tau_k > 0$. To capture the delay, we execute the controller code to calculate $g(t_k, y_k)$ and store the "pending" control value in a memory variable $\tilde{u}_{k+1} := g(t_k, y_k)$ until the next sample time after $\tilde{t}_{k+1} := t_k + \tau_k$. To determine the computation delay τ_k, Sharc simulates the execution of the controller code on a given processor using a cycle-accurate microarchitectural simulator, as described in Section 2.3. The system continues to use $u = u_k$ until the next sample time after $\tilde{t}_k$, at which point the input is set to the control value that was being computed: $u = \tilde{u}_k$.

The model used by Sharc for the closed-loop dynamics of a CPS's controlled by a computationally delayed controller is

$$x_{k+1} = f\big(t_k, x_k, u_k, w(t_k)\big) \tag{3a}$$

$$y_k = h\big(x_k, u_k, w(t_k)\big) \tag{3b}$$

$$\underset{\substack{\text{(Computation} \\ \text{in progress)}}}{\text{If } t_{k+1} < \tilde{t}_k,} \begin{cases} u_{k+1} = u_k \\ \tilde{u}_{k+1} = \tilde{u}_k \\ \tilde{t}_{k+1} = \tilde{t}_k \end{cases} \tag{3c}$$

$$\underset{\substack{\text{(Computation} \\ \text{finished)}}}{\text{If } t_{k+1} \geq \tilde{t}_k,} \begin{cases} u_{k+1} = \tilde{u}_k \\ \tilde{u}_{k+1} = g(t_k, y_k) \\ \tilde{t}_{k+1} = t_k + \tau_k \text{ for computing } g(t_k, y_k), \end{cases} \tag{3d}$$

where the initial state $x_0 \in \mathbb{R}^{n_x}$ and initial control value $u_0 \in \mathbb{R}^{n_u}$ are given, the initial pending control value is $\tilde{u}_0 = g(t_0, y_0)$, and $\tilde{t}_0 := \tau_0$ is the time required to execute $g(t_0, y_0)$. In (3c), the memory variable $\tilde{u}$ and the end time $\tilde{t}$ of the computation are held constant while the computation is in progress. When the computation finishes, u is set to $\tilde{u}$. Then $\tilde{u}$ and $\tilde{t}$ are updated to record a new execution of the calculation of $g(t_k, y_k)$. Figure 2 shows the feedback diagram for the closed-loop system in (3).

2.3 Computational Hardware Simulation

In this section we introduce how we model and simulate the computational hardware. To calculate the computational delay of a given controller, we use a microarchitectural simulator. Existing microarchitectural simulators can execute accurate simulations of arbitrary computer programs on a given hardware platform. We will discuss in Section 3 how Sharc jointly simulates the hardware execution of a control algorithm and the physics of a system. Sharc uses the *Scarab Microarchitectural Simulator* [1, 21] to perform hardware simulation. To simulate the execution time of a control algorithm, Scarab processes either compiled x86 binaries or traces of x86 assembly instructions. The microarchitectural simulator is agnostic to the programming language because it consumes compiled assembly code.

Although it is possible to directly measure application execution times on a physical processor, simulating a controller executable with Scarab provides the following advantages:

(1) Scarab allows for arbitrary modification of hardware parameters, such as the cache size, clock frequency, and the depth and width of the CPU pipeline, to allow for analysis of hypothetical computing platforms without necessitating fabrication. Thus, by using Scarab with Sharc, we can prototype new hardware components and measure the resulting changes in a system's dynamical performance.

(2) Scarab produces detailed statistics, making the internal hardware state observable and thereby allowing for better performance analysis. This is in contrast to physical CPUs, which only provide limited visibility through existing performance monitoring unit (PMU) counters.

Both of these aspects are crucial for performing optimal hardware-software co-design of control systems.

The simulation of control algorithms in Scarab provides high precision and fidelity because Scarab models both the architectural and microarchitectural states of the CPU at the level of individual clock cycles. The architectural state includes all registers, program counters (PC), and the memory of the processor as specified by the instruction set architecture (ISA). The microarchitectural state comprises the tables and internal meta-data utilized by the branch predictor [26], prefetchers, and cache replacement mechanisms. In a microarchitectural simulator, each simulated assembly instruction moves through a pipeline of various stages during its lifetime, including the *fetch*, *decode*, *execution*, and *retirement* stages. At each stage, the instruction triggers events along its path. Modern CPUs implement instruction pipelines that are deep and wide, meaning that there can be hundreds of instructions in the pipeline at the same time, each one triggering events in every cycle. The full pipeline of instruction processing is accurately modeled by Scarab.

Furthermore, the modern CPU architecture, as emulated by the Scarab simulator, follows an out-of-order CPU design that can be divided into two parts, as shown in Fig. 3. The *front-end* identifies the next instructions to be fetched from main memory, stores them into the fetch target queue (FTC) and instruction cache, and decodes them. The front end is also responsible for handling control-flow instructions, such as jumps and branches, utilizing a TAgged GEometric (TAGE) history length branch predictor [26], branch target buffer (BTB), and return address stack (RAS) predictor.

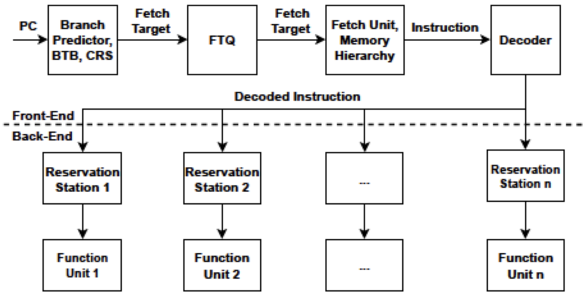

Figure 3: Scarab's Architecture. Modern CPUs are comprised of the *Frontend*, responsible for predicting future executed instructions (Branch Predictor), buffering their instruction address (FTQ), fetching them from the instruction cache (Fetch), and decoding their arithmetic operation (Decoder). Decoded instructions are then forwarded to the *Backend*, which contains the instruction schedulers (Reservation Stations) selecting ready instructions to be processed by the functional units (Units 1 through N).

The *back-end* consumes the stream of instructions provided by the front-end and executes them through different functional units based on the instruction type (e.g., loads, stores, Arithmetic Logic Unit (ALU), vector instruction queues) acting as reservation stations [28]. The instruction scheduler picks instructions as soon as they are ready (all source operands are available) and forwards them to the appropriate functional units. The execution stage also detects mispredicted branch instructions to trigger pipeline flushes ensuring correct execution. To emulate, serve, load, and store instructions, the simulator models three cache levels and implements a detailed Dynamic Random Access Memory (DRAM) model utilizing Ramulator [16].

The Scarab simulator provides observability of over a thousand low-level events, including the number of executed CPU cycles, mispredicted control-flow instructions, data and instruction cache misses, and a tally of the number of cycles each functional unit is busy. Analyzing these statistics reveals which CPU components limit the performance of a particular program and thereby provides insights into how to improve the hardware architecture or software implementation. Scarab features two simulator frontends: execution-driven and trace-based. We utilize trace-based simulation to supply instructions to the CPU pipeline. The traces, captured using DynamoRIO [7], preserve a precise continuous sequence of dynamically executed instructions including memory addresses for load and store operations. DynamoRIO is a runtime code manipulation system that enables dynamic analysis, profiling, and optimization by allowing arbitrary modifications to application instructions on various architectures and operating systems.

3 SHARC SIMULATOR

In Sharc, the microarchitectural simulator is executed in parallel with a simulation of the physics. Figure 4 illustrates how Sharc simulates the physics and the control algorithm in parallel. The simulation of the physics is executed through a user-provided implementation of a Python interface, which may call external physics simulators. The particular dynamics of a system are defined by writing a subclass of a Python class named `Dynamics`, provided by Sharc. Pseudocode for a `MyDynamics` subclass of `Dynamics` is shown here:

```
class MyDynamics(Dynamics):
  def evolve_state(self, t0, x0, u, tf):
    # Evolve the state from t0 to tf given
    # initial state x0 and control input u.
    return xf # Final state of the system at tf.

  def get_output(self, x, u, w):
    return y # Generate output

  def get_exogenous_input(self, t):
    return w # Generate exogenous input
```

Similarly, a controller is defined in C++ by writing a subclass of a C++ class provided with Sharc named `Controller`. Pseudocode for a `MyController` subclass of `Controller` is as follows:

```
class MyController : public Controller {
    void calculateControl(int k, double t, const Vec &y){
        // Evaluate u = g(t, y) and set the
        // object's 'control' property to the result.
        control = u;
    }
};
```

Whenever a new control $g(t, y)$ is computed in a Sharc simulation, the code in the `calculateControl` function is simulated by Scarab to determine the computational delay of computing $g(t, y)$. A noteworthy feature of Sharc's design is that the same controller code can be used by Sharc as would be deployed on an actual cyber-physical system.

3.1 Serial Mode

The Sharc simulator supports two modes. While the *serial* mode is optimized for maximum accuracy, the *parallelized* mode minimizes simulation time through parallel processing. We will describe the serial mode in the following and refer to Section 3.2 for a detailed description of the parallel mode. When running in serial mode, Sharc executes the controller in a single subprocess that runs for the entire duration of the simulation.[1] The controller subprocess simulates the controller with Scarab using an "execution-driven" mode, which allows for statistics, such as CPU cycle counts, to be accessed during the execution of the simulation, as opposed to having to wait until the simulation completes. At each time step, Sharc sends the current time step k, the current time t_k and output y_k to the controller. The exogenous input w is generated at each time $t \in [t_k, t_{k+1}]$ using the `get_exogenous_input` function. Once the values are received by the controller, Scarab begins recording statistics while the control value is computed, at which point the statistics are saved, and the control value u is sent back to the Python process running the physics simulator. The inter-process communication between the dynamics simulator and the controller is accomplished using named pipe files. After u is received by the physics simulator, Sharc reads the computation statistics from Scarab, which it uses to determine the computation time τ for computing u.

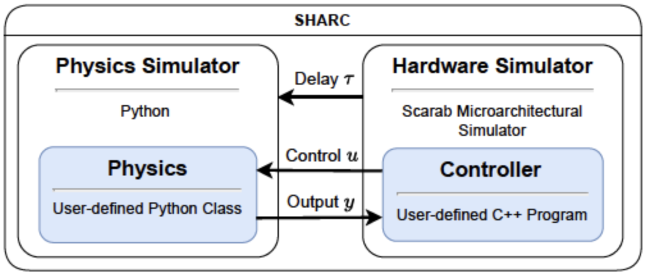

Figure 4: Diagram of the Sharc simulator structure. The physics simulator and hardware simulators run in separate processes, with inter-process communication done via named pipe files.

3.2 Parallel Mode

For computationally intensive control algorithms, running simulations in serial mode can take a long time. Due to simulating the microarchitectural components of a CPU, the time to simulate a controller algorithm with Scarab can be over 10,000 times longer than computing the same algorithm on physical hardware. To address this challenge, we developed a method to improve simulation times through parallel execution.

Recall that the discretized physics are assumed to use a periodic time grid $t_0 \coloneqq 0$, $t_1 \coloneqq T$, $t_2 \coloneqq 2T, \ldots,$ with both the sensor measurements and control inputs discretized at a constant sampling rate of T. This allows Sharc to parallelize simulations across time steps. Figure 5 provides an overview of the parallelized approach, where full-state feedback is used for simplicity ($y_k = x_k$). Each column of Steps 2 and 3 are executed in parallel.

The simulation takes the initial state value $x_0 \in \mathbb{R}^{n_x}$ and initial control value $u_0 \in \mathbb{R}^{n_u}$ (which is to be applied until the first control update is finished) and runs over a time horizon $K \in \mathbb{N}$. It is divided into batches each containing c-many time steps, where c is typically the number of processors available. During Step 1, the simulator assumes that each control update u_{k+1} can be computed within one sample time ($\tau_k < T$) and thus always updates the control at the next time step. The resulting sequence $x_{k_0}, x_{k_0+1}, \ldots, x_{k_0+c}$ is an initial guess of the trajectory that the system would take if every control update is computed within one sample time ($\tau_k < T$). Since Step 1 assumes no delays in computing the control inputs, Scarab is not used to record computation times, so this step is executed very quickly.

In Steps 2 and 3, Sharc backtracks to check whether the computational delay of updating the control at each time step is actually less than T. In particular, Sharc creates c-many processes—one for each time step in the batch. Each process is assigned a unique $t_k \in \{t_{k_0}, t_{k_0+1}, \ldots, t_{k_0+c-1}\}$. Using the precomputed value x_k and u_k, the simulator recomputes the control update $u_{k+1} = g(t_k, y_k)$, but this time Sharc runs the controller executable with DynamoRIO to generate a trace of its execution (Step 2) which is then simulated using Scarab to generate the computation time τ_k (Step 3).

[1] Here, "serial" vs. "parallel" mode refers only to whether one time step or many are computed concurrently. In the serial mode, parallelization is used to run controller and physics concurrently.

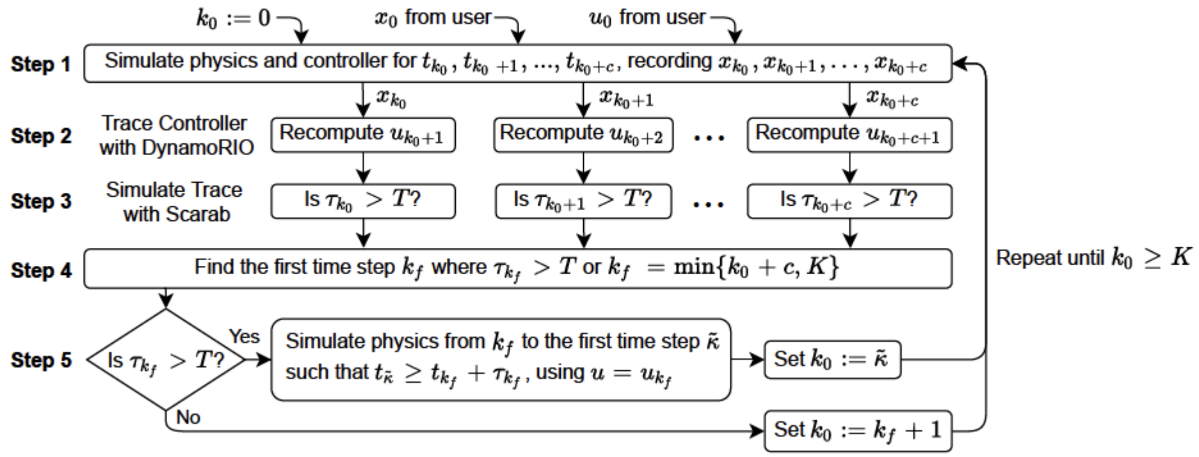
$k_0 := 0$
x_0 from user
u_0 from user
Step 1
Simulate physics and controller for $t_{k_0}, t_{k_0+1}, \ldots, t_{k_0+c}$, recording $x_{k_0}, x_{k_0+1}, \ldots, x_{k_0+c}$
x_{k_0}
x_{k_0+1}
x_{k_0+c}
Step 2
Trace Controller with DynamoRIO
Recompute u_{k_0+1}
Recompute u_{k_0+2}
Recompute u_{k_0+c+1}
Step 3
Simulate Trace with Scarab
Is $\tau_{k_0} > T$?
Is $\tau_{k_0+1} > T$?
Is $\tau_{k_0+c} > T$?
Step 4
Find the first time step k_f where $\tau_{k_f} > T$ or $k_f = \min\{k_0 + c, K\}$
Repeat until $k_0 \geq K$
Step 5
Is $\tau_{k_f} > T$?
Yes
Simulate physics from k_f to the first time step $\tilde\kappa$ such that $t_{\tilde\kappa} \geq t_{k_f} + \tau_{k_f}$, using $u = u_{k_f}$
Set $k_0 := \tilde\kappa$
No
Set $k_0 := k_f + 1$

Figure 5: Diagram of Sharc's parallel mode using a simulation horizon of K time steps parallelized across c processors. Each column is executed in parallel. In this diagram, full-state feedback $y_k = x_k$ is used, for simplicity.

In Step 4, Sharc searches for the first time step k_f where the computation time τ_{k_f} exceeds T. If such a time step is found, then any subsequent time steps ($k > k_f$) are invalid because the states were generated using control values that were applied (in the simulation) before the controller could compute them. If all of the control updates took less than the sample time ($\tau_k < T$), then k_f is defined as the last sample in the batch ($k_f := k_0 + c$) or the simulation ($k_f := K$), whichever is first.[2]

In Step 5, Sharc checks whether there was a missed computation ($\tau_{k_f} > T$), revises the simulation trajectory accordingly, and then continues to the next batch. If $\tau_{k_f} \leq T$, then controller has computed each update within the sample time, so the simulation either moves to the next batch with $k_0 = k_f + 1$ (if $k_f < K$) or terminates (if $k_f = K$). On the other hand, if $\tau_{k_f} > T$, then the control computation that was started at t_{k_f} will not be available at t_{k_f+1}, violating the assumption in Step 1. Thus, Sharc must recompute the system's trajectory starting from x_{k_f} using $u = u_{k_f}$ until the control update finishes. As in Section 2.1, let $\tilde{t}_{k_f} := t_{k_f} + \tau_{k_f}$ (which typically is not a sample time) and let $\tilde\kappa$ be the smallest integer such that $\tilde{t}_{k_f} \leq \tilde\kappa T$. In other words, $\tilde\kappa T$ is the first sample time after the computation that started at t_{k_f} finishes. We recompute the portion of the trajectory computed in Step 1 from k_f to $\tilde\kappa$ according to (3a) with $u_k = u_{k_f}$ held constant:

$$x_{k+1} = f\big(t_k, x_k, u_{k_f}, w(t_k)\big) \quad \forall k \in \{k_f, k_f + 1, \ldots, \tilde\kappa - 1\}.$$

Then, the simulation moves to the next batch, using $k_0 := \tilde\kappa$ or terminates (if $\tilde\kappa \geq K$). The simulation from k_f to $\tilde\kappa$ does not require using Scarab since we already know the end time of the pending computation, so it can be computed quickly without parallelization. At $\tilde\kappa$, however, the controller will start computing a new control value, so Sharc starts a new batch of parallelization.

Due to the large slowdown incurred by using Scarab, parallelization is important for simulating computationally intensive control algorithms, but parallelization somewhat reduces the fidelity of the simulation. In particular, the parallelized mode is somewhat less accurate in determining computation times because running each time step in a separate process prevents the simulator from accounting for some sequential computational effects between time steps, such as memory caching. In contrast, running Sharc in serial mode allows transient memory effects to persist between time steps. The results, however, of Section 4.1, below, show that there is only a small difference between the delays calculated by the parallelized and serial modes. Given the large reduction in simulation durations, the trade-off between accuracy and speed often justifies the use of the parallelized mode.

Parallelization is useful for mitigating the 10,000× slowdown incurred by simulating the controller with Scarab. The speedup in the parallelized approach, compared to the serial mode comes from Steps 2 and 3 in Fig. 5. To quantify the possible improvements gained by parallelizing, Theorem 3.1 describes how much simulation time is reduced by using the parallel approach instead of the serial approach. In particular, it examines how much time it takes to compute N many jobs parallelized across c many CPU cores. In this case, each job corresponds to running Scarab once to determine the computation time of a control input at a particular time step.

Theorem 3.1. *Consider a computational system managing N jobs, each requiring a fixed amount of time—defined as one unit of time—to execute on a single CPU core. The system employs c CPU cores for parallelization where $c \leq N$. Assume that the probability of failure for each job is i.i.d. with probability p and that the system restarts from the job index $k + 1$ after each failure at job index k. Then, the average time $\overline{T}$ to complete all the jobs is*

$$\overline{T}(c, p) = \frac{Np}{1 - (1 - p)^c}. \tag{4}$$

Proof. The computational process described herein constitutes a Bernoulli process as it consists of a sequence of independent binary random variables representing job success or failure. To analyze this, we calculate the average number of completed jobs, denoted as $\overline{K}$, for each parallel task. A closed-form expression for

[2] In practice, Sharc truncates any batches that would extend past K.

$\overline{K}$ is derived as follows:

$$\begin{aligned}\overline{K} &= \sum_{i=0}^{c-1}(i+1)\cdot\Pr\{K=i\} + c\cdot\Pr\{K=c\},\\ &= p\cdot\sum_{i=0}^{c-1}(i+1)\cdot(1-p)^i + c\cdot(1-p)^c = \frac{1-(1-p)^c}{p}.\end{aligned} \tag{5}$$

This result leads to the expression for the average time to complete all N jobs:

$$\overline{T}(c,p) = \frac{N}{\overline{K}} = \frac{Np}{1-(1-p)^c}. \tag{6}$$

□

Note that $\overline{T}(1,p) = N$, reflecting the case when all jobs are processed sequentially, and $\lim_{p\to 0}\overline{T}(N,p) = 1$, aligning with the expectation that in the absence of failures, the system completes all jobs in unit time. The parallelization gain, which is the speedup factor for running N jobs in parallel on c cores instead of running N jobs sequentially on one core, is

$$\delta(c,p) \triangleq \frac{\overline{T}(1,p)}{\overline{T}(c,p)} = \frac{N}{\left(\frac{Np}{1-(1-p)^c}\right)} = \frac{1-(1-p)^c}{p}. \tag{7}$$

Thus, when using c cores, the parallel approach is faster than the serial approach by a factor of $(1-(1-p)^c)/p$. In the ideal case with unlimited tasks and unlimited computational resources, $\lim_{c\to\infty}\delta(c,p) = 1/p$. Therefore, when using Sharc's parallel mode to simulate a system that has a uniform probability p at each time step of the control delay τ being larger than T, the expected parallelization speedup is never better than $1/p$, regardless of how many CPUs are used to parallelize the simulation.

4 NUMERICAL EXPERIMENTS

In this section, we present two examples of the Sharc simulator applied to systems using a model predictive controller (MPC). In Section 4.1, MPC is used for adaptive cruise control of a vehicle on a roadway. The resulting MPC problem is linear and thus can be solved efficiently, allowing for the serial mode of Sharc to run simulations in a reasonable time. We provide a comparison with the parallelized mode to demonstrate the similarity of the results. In Section 4.2, MPC is used to balance an inverted pendulum on a rolling cart. For the inverted pendulum, the MPC problem is nonlinear, which is significantly more computationally expensive to solve than linear MPC. Thus, the serial simulation mode is not practical, and only results from the parallelized mode are presented.

4.1 Adaptive Cruise Control

In this section, we present an example of applying Sharc to an adaptive cruise control (ACC) system used for longitudinal control of a vehicle on a highway. The dynamical model used in this example is adapted from [27]. In particular, we consider an ego vehicle velocity v and a desired velocity $v_{\mathrm{des}} := 15\,\mathrm{m/s}$. The ego vehicle is following a public *front* vehicle that has velocity $t \mapsto v_{\mathrm{F}}(t)$ that we do not control and is considered as an exogenous input to the system. The headway from the front of the ego vehicle to the rear of the front vehicle is denoted h.

The acceleration of the ego vehicle is

$$\dot{v} = \frac{1}{M}\Big(u^{\mathrm{a}} - u^{\mathrm{b}} - F\Big),$$

where M is the mass of the vehicle, u^{a} is acceleration force, u^{b} is braking force, and F is a resistive force on the ego vehicle due to drag and friction. The controlled quantities are u^{a} and u^{b}. Assuming travel on a level roadway, $v \mapsto F(v) := \beta + \gamma v^2$,where $\beta \geq 0$ and $\gamma \geq 0$ are determined empirically. Values for τ, β, γ, and M can be found in [27, Table 1]. The resulting dynamics are

$$\dot{h} = v_{\mathrm{F}}(t) - v \tag{8}$$

$$\dot{v} = \frac{1}{M}\Big(u^{\mathrm{a}} - u^{\mathrm{b}} - F(v)\Big). \tag{9}$$

The quadratic friction term $F(v)$ makes the system nonlinear, so we linearize F around $v_0 \geq 0$ as $v \mapsto (\beta - \gamma v_0^2) + 2\gamma v_0 v$. The state of the system is $x := (h, v) \in \mathbb{R}^2$ and the input is $u := (u^{\mathrm{a}}, u^{\mathrm{b}}) \in \mathbb{R}^2$. We write the exogenous inputs to the system as $w := (v_{\mathrm{F}}, 1) \in \mathbb{R}^2$, where v_{F} is the velocity v_{F} of the front vehicle, and "1" in the second component is used to incorporate a constant term $(\gamma v_0^2 - \beta)/M$ arising from F. The resulting system is

$$\dot{x} = \begin{bmatrix}0 & -1\\ 0 & -2\gamma v_0/M\end{bmatrix}x + \begin{bmatrix}0 & 0\\ 1/M & -1/M\end{bmatrix}u + \begin{bmatrix}1 & 0\\ 0 & (\gamma v_0^2 - \beta)/M\end{bmatrix}w. \tag{10}$$

The continuous-time dynamics are discretized with a sample time $T := 0.1$ s, resulting in a discrete-time system we write as

$$x_{k+1} = A(v_0)x_k + B(v_0)u_k + B_d(v_0)w_k, \tag{11}$$

where x_k, u_k, and w_k are the values of x, u, and w, respectively, at $t = kT$. Note that $A(v_0)$, $B(v_0)$, and $B_d(v_0)$ depend on v_0, the center of the linearization.

4.1.1 MPC Problem Formulation. To generate control values at each discrete time $k_0 \in \mathbb{N}$, we apply MPC with a prediction horizon of $N_p \in \mathbb{N}$ time steps. Given any discrete time $k_0 \in \mathbb{N}$, let $\hat{x}_{k_0}$ be a measurement-based estimate of x at k_0, and let $\hat{v}_0$ be the velocity component of $\hat{x}_{k_0}$. For each $k \in \{k_0, k_0+1, \ldots, k_0+N_p\}$, let $k \mapsto u_{k|k_0}$ be planned control values starting at k_0, and let $k \mapsto x_{k|k_0}$ be the state prediction generated by (11) with initial condition $x_{k_0|k_0} = \hat{x}_{k_0}$ and using the control signal $u_{k|k_0}$.

The cost function of the MPC problem is a quadratic function that penalizes the deviance of $v_{k|k_0}$ from the desired velocity v_{des}, the control effort $u_{k|k_0}$, and changes to the control effort, which roughly corresponds to the vehicle's jerk ($\ddot{v}$). A positive definite matrix $R \in \mathbb{R}^{2\times 2}$ defines the control weight matrix and $\alpha \geq 0$ is a jerk penalization parameter.

The ego vehicle must always satisfy the following constraints:

$$\begin{aligned}\text{Headway:}\quad & h \geq h_{\min} := 6\,\mathrm{m}\\ \text{Velocity:}\quad & 0\,\mathrm{m/s} \leq v \leq v_{\max} := 20\,\mathrm{m/s}\\ \text{Acceleration force:}\quad & 0\,\mathrm{N} \leq u^{\mathrm{a}} \leq u^{\mathrm{a}}_{\max} := 4880\,\mathrm{N}\\ \text{Braking force:}\quad & 0\,\mathrm{N} \leq u^{\mathrm{b}} \leq u^{\mathrm{b}}_{\max} := 6507\,\mathrm{N}.\end{aligned}$$

To ensure the headway constraint $h \geq h_{\min}$ is satisfiable past the end of the MPC prediction horizon, we also must include a terminal constraint. In particular, h and v must satisfy

$$h \geq \frac{v^2}{2|a|} - \frac{v_{\mathrm{F}}^2}{2|a_{\mathrm{F}}|} + h_{\min}, \tag{12}$$

Problem 1 (Linear MPC).

$$\text{minimize}\quad J\left(x_{(\cdot)|k_0}, u_{(\cdot)|k_0}\right) := \sum_{k=k_0}^{k_0+N_p} \left(v_{k|k_0} - v_{\text{des}}\right)^2 + \sum_{k=k_0}^{k_0+N_p-1} u_{k|k_0}^{\top} R u_{k|k_0} + \alpha \sum_{k=k_0}^{k_0+N_p-2} \left|u_{k+1|k_0} - u_{k|k_0}\right|^2 \tag{14a}$$

with respect to

$$x_{k_0|k_0},\ x_{(k_0+1)|k_0},\ \ldots,\ x_{(k_0+N_p)|k_0} \in \mathbb{R}^2 \tag{14b}$$

$$u_{k_0|k_0},\ u_{(k_0+1)|k_0},\ \ldots,\ u_{(k_0+N_p-1)|k_0} \in \mathbb{R}^2 \tag{14c}$$

subject to

$$x_{k_0|k_0} = \hat{x}_{k_0}, \tag{14d}$$

and for each $k = k_0,\ k_0+1,\ \ldots,\ k_0+N_p-1$,

$$x_{k+1|k_0} = A(\hat{v}_0)x_{k|k_0} + B(\hat{v}_0)u_{k|k_0} + B_d(\hat{v}_0)\hat{w}(k|k_0), \tag{14e}$$

and for each $k = k_0,\ k_0+1,\ \ldots,\ k_0+N_p$,

$$0 \le v_{k|k_0} \le v_{\max}, \tag{14f}$$

$$0 \le u^{\mathrm{a}}_{k|k_0} \le u^{\mathrm{a}}_{\max}, \tag{14g}$$

$$0 \le u^{\mathrm{b}}_{k|k_0} \le u^{\mathrm{b}}_{\max}, \tag{14h}$$

$$h_{\min} \le h_{k|k_0}, \tag{14i}$$

and for $k = k_0 + N_p$,

$$h_{k|k_0} \ge \frac{v_{\max}}{2|a|} v_{k|k_0} - \frac{\hat{v}_{\mathrm{F}}^2(k|k_0)}{2|a_{\mathrm{F}}|} + h_{\min}. \tag{14j}$$

at the end of the prediction horizon, where $a_{\mathrm{F}} < 0$ is a lower bound on the rate of deceleration of the front vehicle (that is $\dot{v}_{\mathrm{F}} \ge a_{\mathrm{F}}$) and $a < 0$ which is an upper bound on the deceleration of the ego vehicle when maximum braking is applied (that is, $\dot{v} \le a$ when $u^{\mathrm{a}} = 0$ and $u^{\mathrm{b}} = u^{\mathrm{b}}_{\max}$).

Equation (12) is not suitable as a linear MPC constraint, however, because it includes a nonlinear term and depends on the future velocity of the front vehicle, which is unknown. By assuming that the front vehicle applies maximum braking, we estimate its worst-case future velocity as a sequence $k \mapsto \hat{v}_{\mathrm{F}}(k|k_0)$. Then, we use $k \mapsto \hat{w}(k|k_0) := (\hat{v}_{\mathrm{F}}(k|k_0),\ 1)$ as a worst-case prediction of the w. To remove the v^2 nonlinearity in (12), we replace v^2 with $vv_{\max} \ge v^2$, creating a more conservative terminal constraint:

$$h_{(k_0+N_p)|k_0} \ge \frac{v_{\max}}{2|a|} v_{(k_0+N_p)|k_0} - \frac{\hat{v}_{\mathrm{F}}^2(k_0+N_p|k_0)}{2|a_{\mathrm{F}}|} + h_{\min}. \tag{13}$$

The resulting MPC problem formulation is shown in Problem 1.

In Fig. 6, the results of simulating the ACC system are shown with a comparison between the results of serial and parallel simulation schemes. We see that at as the headway decreases, the system hits a point around $t = 3.5$ s when the delays significantly increase, rising above the sampling time. This increase causes the updated control values to be delayed by six time steps. The delays increase at this point because more MPC inequality constraints become active, making the optimization problem harder to solve. In this simulation, the vehicle recovers before colliding with the lead vehicle, but if the front vehicle brakes more aggressively, the computational delays could result in a collision.

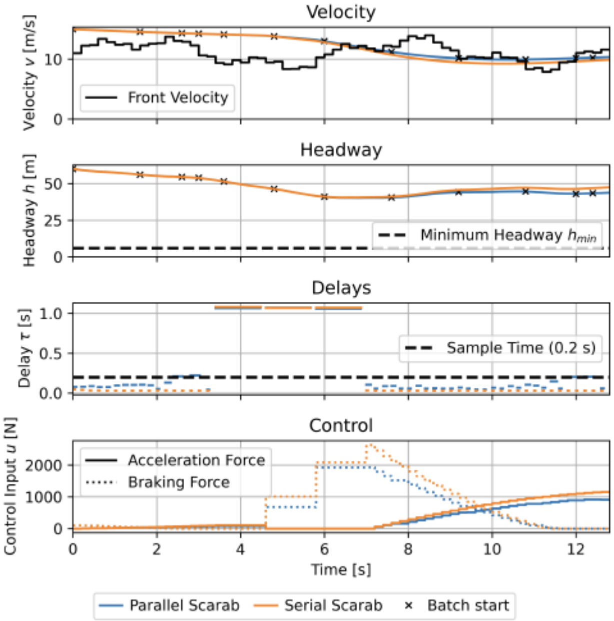

Figure 6: Comparison of trajectories for the ACC system from Section 4.1 simulated using the serial and parallel modes. In the Delays plot, the horizontal lines extend from the start time of each computation to its completion time.

Figure 7 shows a comparison of Sharc simulations using instruction caches of size 1 KB, 8 KB, and 1 MB. We see that computation times increase as the size of the instruction cache shrinks, producing significant deviation in v and h between simulations.

4.2 Inverted Pendulum System

We examine the inverted pendulum system as an example of a system that requires significantly more computational resources than the previous example. In contrast to the linear MPC used in the ACC example, the inverted pendulum system is controlled using nonlinear MPC because the dynamics are nonlinear. Since nonlinear MPC is much more computationally demanding, the serial approach takes impractically long to simulate in Sharc, motivating the parallel approach. Figure 8 illustrates the inverted pendulum system. The system is initiated with a slight deviation from the upright position, and the objective is to maintain the pole in an upright orientation by controlling the horizontal force applied on the cart. This control problem is substantially more challenging than the ACC problem because of the nonlinear nature of the dynamics and the instability of the upright position. Additionally, maintaining the pole in a vertical position requires the application of control inputs at a much higher frequency, further compounding the complexity of the problem.

4.2.1 Nonlinear MPC Formulation. Nonlinear Model Predictive Control (MPC) addresses the solution of a generic nonlinear optimization problem formulated as in Problem 2. The natural number N_c is the control horizon, $N_p \ge N_c$ is the prediction horizon of the

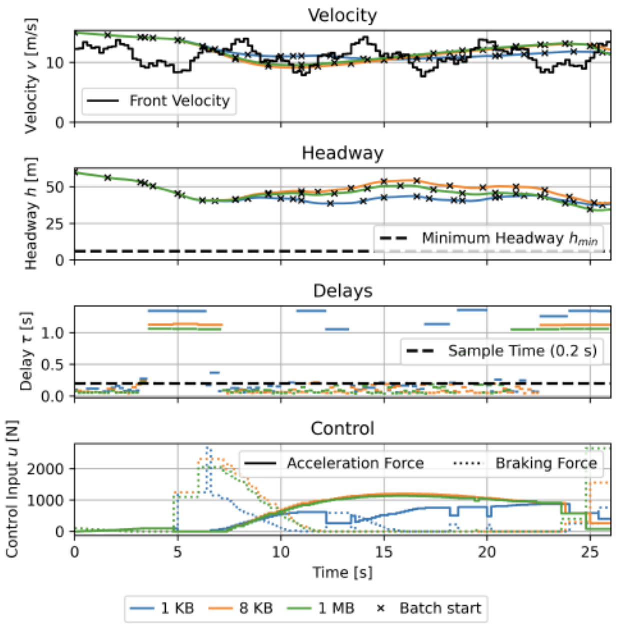

Figure 7: Comparison of Sharc simulations for the ACC system in Section 4.1 using various sizes of instruction cache.

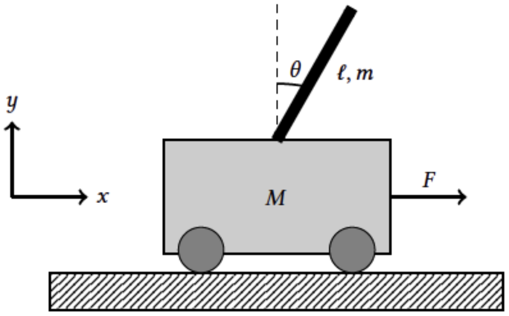

Figure 8: Inverted pendulum system consisting of a pole attached to a controlled cart. The goal is to balance the pole vertically by controlling the horizontal force F to the cart.

problem, and the control input u is kept constant after the control horizon N_c steps. The nonlinear system underlying the MPC is

$$\begin{aligned} x_{k+1} &= f(x_k, u_k) \\ y_k &= h(x_k, u_k). \end{aligned}$$

A key feature of Model Predictive Control is the *receding horizon control* strategy where the Problem 2 is solved over the finite prediction horizon N_p, resulting in an optimal sequence of control inputs $u_{k_0|k_0}, u_{(k_0+1)|k_0}, \dots, u_{(k_0+N_c-1)|k_0}$. However, only the first control input $u_{k_0|k_0}$ is applied to the system. Once $u_{k_0|k_0}$ is applied, the state of the system is updated, and the optimization problem is solved again using the new state as the initial condition.

Problem 2 (Nonlinear MPC).

$$\begin{aligned} \text{minimize} \quad J(x_{(\cdot)|k_0}, u_{(\cdot)|k_0}) &:= \sum_{k=k_0}^{k_0+N_c-1} C(x_{k|k_0}, u_{k|k_0}) \\ &\quad + \sum_{k=k_0+N_c}^{k_0+N_p-1} C(x_{k|k_0}, u_{(k_0+N_c-1)|k_0}) \end{aligned} \tag{15a}$$

with respect to

$$x_{k_0|k_0},\ x_{(k_0+1)|k_0},\ \dots,\ x_{(k_0+N_p)|k_0} \in \mathbb{R}^{n_x} \tag{15b}$$

$$u_{k_0|k_0},\ u_{(k_0+1)|k_0},\ \dots,\ u_{(k_0+N_c-1)|k_0} \in \mathbb{R}^{n_u} \tag{15c}$$

subject to

$$x_{k_0|k_0} = \hat{x}_{k_0}, \tag{15d}$$

and for each $k = k_0,\ k_0+1,\ \dots,\ k_0+N_p-1$,

$$x_{k+1|k_0} = f(x_{k|k_0}, u_{k|k_0}), \tag{15e}$$

and for each $k = k_0,\ k_0+1,\ \dots,\ k_0+N_p$,

$$\ell_{\mathrm{i}}(x_{k|k_0}, y_{k|k_0}, u_{k|k_0}) \le 0, \tag{15f}$$

$$\ell_{\mathrm{e}}(x_{k|k_0}, u_{k|k_0}) = 0. \tag{15g}$$

To solve Problem 2 in numerical experiments, the `libmpc++` open-source C++ MPC library is used [23]. The optimization problem is solved using the Sequential Least Squares Quadratic Programming (SLSQP) method [18] iteratively for various initial states $\hat{x}_{k_0} \in \mathbb{R}^{n_x}$, aiming to minimize the user-defined cost function $C(x_{k|k_0}, u_{k|k_0})$.

4.2.2 Inverted Pendulum System Dynamics. Consider the system $x := (p, v, \theta, \omega) \in \mathbb{R}^4$, where p denotes the cart's position, v its velocity, θ the pole's angular deviation from the upright position, and ω the pole's angular velocity. Assuming no friction, the nonlinear continuous-time dynamics are:

$$\dot{x} = \begin{bmatrix} \dot{p} \\ \dot{v} \\ \dot{\theta} \\ \dot{\omega} \end{bmatrix} = \tilde{f}(x, u) = \begin{bmatrix} v \\ \dfrac{F + m\omega^2 \ell \sin\theta - m\ell\dot{\omega}\cos\theta}{m+M} \\ \omega \\ \dfrac{(m+M)g\sin\theta - (F + m\omega^2\ell\sin\theta)\cos\theta}{\ell\big(4(m+M)/3 - m\cos^2\theta\big)} \end{bmatrix}. \tag{16}$$

In this formulation, $u = F$ denotes the control force applied to the cart. The system parameters are the pole mass m, the cart mass M, the pole length ℓ, and the gravitational constant g. The continuous-time system $\dot{x} = \tilde{f}(x, u)$ is discretized with sample time $T := 0.1$ s to produce a discrete-time system as in Problem 2. No explicit constraints are imposed on the inputs, states, or outputs. Instead, the desired behavior is achieved by tailoring the cost function, defined as a quadratic function of the state and input:

$$C(x_{k|k_0}, u_{k|k_0}) = x_{k|k_0}^\top Q x_{k|k_0} + u_{k|k_0}^\top R u_{k|k_0},$$

where $Q \in \mathbb{R}^{n_x \times n_x}$ and $R \in \mathbb{R}^{n_u \times n_u}$ are weighting matrices that penalize deviations from the desired state and control effort, respectively.

The parameters used in our experiments for the inverted pendulum system are shown in Table 1. The system is initialized with a 2-degree angular deviation from the upright position, and the goal

Parameter	Notation	Value
Sampling time	T	0.1 s
Control horizon	N_c	4 time steps
Prediction horizon	N_p	20 time steps
State cost matrix	Q	diag(1, 1, 1, 1)
Input cost matrix	R	1
Mass of cart	M	1.0 kg
Mass of pole	m	0.1 kg
Length of pole	ℓ	1 m
Initial state	x_0	$\begin{bmatrix}0 & 0 & 2^\circ & 0\end{bmatrix}^\top$
Number of CPU cores	c	16
Experiment Horizon	K	64 time steps (6.4 s)

Table 1: Parameters used for the inverted pendulum system.

of the controller is to make the upright position asymptotically stable. Figure 9 shows the plots of the state norm, computational delays, and control inputs across three distinct processor clock frequencies. At 625 MHz, the system achieves stability as the computational delays consistently remain below the sample time ($T = 0.1$ s), albeit approaching this threshold. However, as we slightly decrease the clock frequency to 600 MHz, we start to see delays longer than sample time, causing the control input to be repeated for one or more time steps. This suboptimal control action forces the inverted pendulum system to move further from the equilibrium, causing the computational difficulty to increase, leading to progressively longer computation times. The cumulative effect of these cascading delays ultimately prevents stabilization of the upright equilibrium. As anticipated, further reduction of the clock frequency to 500 MHz aggravates this behavior by ending up with more delays. These results underscore the critical importance of minimizing computational delays in control systems to maintain stability and optimal performance.

The reason we utilized the parallel approach for this problem can be understood by looking at the simulation durations. We utilized $c = 16$ CPU cores and ran the experiments for $K = 64$ time steps. Thus, if there are no delays the experiment is expected to be finished in four batches as in the case with the blue curve in Fig. 9. The start of each batch is denoted by a cross (×) on the plot of $\|x\|$. Since each simulation step requires approximately 15 minutes, the serial approach would require roughly 16 hours to complete. In the parallel mode, however, simulations terminated in 4 batches with a 625 MHz clock speed, 17 batches with 600 MHz, and 20 batches with 500 MHz. The parallel approach yielded significant speedup, with speedup factors of 16×, 3.76×, and 3.20× compared to the serial method, which illustrates how the parallelization speedup is larger when fewer time steps have long computational delays as predicted by Theorem 3.1.

5 CONCLUSION

In this paper, we present SHARC as a tool to simulate user-specified control algorithms on a given processor microarchitecture, evaluating how computational constraints affect the performance of the control algorithm and the safety of the physical system. We illustrated the power and usefulness of SHARC via two examples: an adaptive cruise controller implemented with linear MPC and an inverted pendulum system controlled by nonlinear MPC. By providing insight into the impact of computing hardware on the performance of a CPS, SHARC allows for the co-design of control algorithms and the computational hardware on which they are run. Future work includes 1) using SHARC to identify common bottlenecks in particular classes of control algorithms and computational hardware and 2) developing an automated framework for jointly optimizing the parameters of the hardware and the control algorithm.

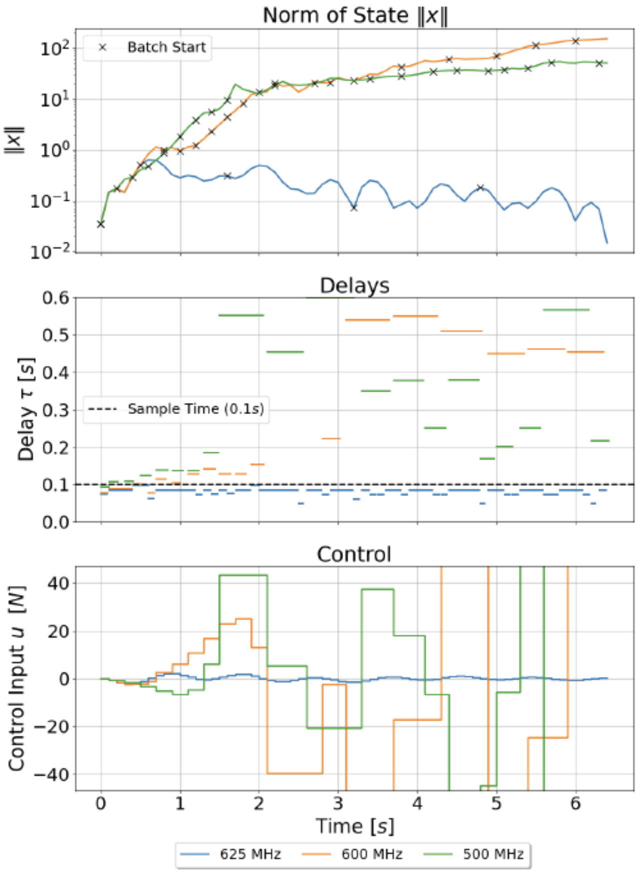

Figure 9: Effect of the clock frequency on the inverted pendulum system presented in Section 4.2.

ACKNOWLEDGMENTS

This research was supported by NSF grants CNS-2039054, CCF-1942754, and CNS-2111688; by AFOSR grants FA9550-19-1-0169, FA9550-20-1-0238, FA9550-23-1-0145, and FA9550-23-1-0313; by AFRL grants FA8651-22-1-0017 and FA8651-23-1-0004; by ARO grant W911NF-20-1-0253; and by DOD grant W911NF-23-1-0158.

REFERENCES

[1] [n. d.]. Scarab. https://github.com/Litz-Lab/scarab
[2] 2023. Tesla Full Self-Driving Chip (FSD chip). https://en.wikichip.org/wiki/tesla_(car_company)/fsd_chip
[3] Daniel Arnström. 2023. *Real-Time Certified MPC: Reliable Active-Set QP Solvers.* Department of Electrical Engineering, Linköping University, Linköping.

[4] Daniel Arnström, David Broman, and Daniel Axehill. 2024. Exact Worst-Case Execution-Time Analysis for Implicit Model Predictive Control. *IEEE Trans. Automat. Control* 69, 10 (Oct. 2024), 7190–7196. https://doi.org/10.1109/TAC.2024.3395521

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[6] Andrea Bonci, Sauro Longhi, Giacomo Nabissi, and Giuseppe Antonio Scala. 2020. Execution Time of Optimal Controls in Hard Real Time, a Minimal Execution Time Solution for Nonlinear SDRE. *IEEE Access* 8 (2020), 158008–158025. https://doi.org/10.1109/ACCESS.2020.3019776

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *Proc. International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization.* IEEE Computer Society, USA, 265–275.

[8] M. Caccamo, G. Buttazzo, and Lui Sha. 2002. Handling Execution Overruns in Hard Real-Time Control Systems. *IEEE Trans. Comput.* 51, 7 (July 2002), 835–849. https://doi.org/10.1109/TC.2002.1017703

[9] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM Trans. Archit. Code Optim.* 11, 3 (Aug. 2014), 28:1–28:25. https://doi.org/10.1145/2629677

[10] Robert N. Charette. 2009. This Car Runs on Code. *IEEE Spectrum* (Feb. 2009). https://spectrum.ieee.org/this-car-runs-on-code

[11] Cyril Faure, Mongi Ben Gaid, Nicolas Pernet, Morgan Fremovici, Grégory Font, and Gilles Corde. 2011. Methods for real-time simulation of Cyber-Physical Systems: application to automotive domain. 1105–1110. https://doi.org/10.1109/IWCMC.2011.5982695

[12] Rolf Findeisen, Lars Grüne, Jürgen Pannek, and Paolo Varutti. 2011. Robustness of Prediction Based Delay Compensation for Nonlinear Systems. *IFAC Proceedings Volumes* 44, 1 (Jan. 2011), 203–208. https://doi.org/10.3182/20110828-6-IT-1002.03090

[13] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. https://doi.org/10.48550/arXiv.2210.14324 arXiv:2210.14324 [cs]

[14] Lars Grüne and Jürgen Pannek. 2011. *Nonlinear Model Predictive Control: Theory and Algorithms.* Springer London, London. https://doi.org/10.1007/978-0-85729-501-9

[15] Makoto Ishikawa, D.J. McCune, George Saikalis, and Shigeru Oho. 2007. CPU Model-Based Hardware/Software Co-design, Co-simulation and Analysis Technology for Real-Time Embedded Control Systems. In *IEEE Real Time and Embedded Technology and Applications Symposium.* IEEE, 3–11. https://doi.org/10.1109/RTAS.2007.9

[16] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (Jan. 2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[17] Larry M. Kinnan. 2009. Use of Multicore Processors in Avionics Systems and Its Potential Impact on Implementation and Certification. In *2009 IEEE/AIAA 28th Digital Avionics Systems Conference.* IEEE, 1.E.4–1–1.E.4–6. https://doi.org/10.1109/DASC.2009.5347560

[18] D. Kraft. 1988. *A Software Package for Sequential Quadratic Programming.* Wiss. Berichtswesen d. DFVLR. https://books.google.com/books?id=4rKaGwAACAAJ

[19] Patrick Leteinturier, Simon Brewerton, and Klaus Scheibert. 2008. *MultiCore Benefits & Challenges for Automotive Applications.* SAE Technical Paper 2008-01-0989. SAE International, Warrendale, PA. https://doi.org/10.4271/2008-01-0989

[20] Franc Mihalič, Mitja Truntič, and Alenka Hren. 2022. Hardware-in-the-Loop Simulations: A Historical Overview of Engineering Challenges. *Electronics* 11, 15 (2022). https://doi.org/10.3390/electronics11152462

[21] Surim Oh, Mingsheng Xu, Tanvir Ahmed Khan, Baris Kasikci, and Heiner Litz. 2024. UDP: Utility-Driven Fetch Directed Instruction Prefetching. In *ACM/IEEE 51st Annual International Symposium on Computer Architecture.* IEEE, 1188–1201. https://doi.org/10.1109/ISCA59077.2024.00089

[22] Yash Vardhan Pant, Houssam Abbas, Kartik Mohta, Truong X. Nghiem, Joseph Devietti, and Rahul Mangharam. 2015. Co-Design of Anytime Computation and Robust Control. In *IEEE Real-Time Systems Symposium.* IEEE, San Antonio, TX, USA, 43–52. https://doi.org/10.1109/RTSS.2015.12

[23] Nicola Piccinelli. [n. d.]. Libmpc++: A library to solve linear and non-linear MPC. https://github.com/nicolapiccinelli/libmpc

[24] Dev Pradhan. 2010. *Multicore Processors Bring Innovation to Medical Imaging.* Technical Report. Texas Instruments.

[25] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13).* Association for Computing Machinery, New York, NY, USA, 475–486. https://doi.org/10.1145/2485922.2485963

[26] André Seznec and Pierre Michaud. 2006. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction-Level Parallelism* 8 (2006), 23.

[27] Stanley W. Smith, Yeojun Kim, Jacopo Guanetti, Ruolin Li, Roya Firoozi, Bruce Wootton, Alexander A. Kurzhanskiy, Francesco Borrelli, Roberto Horowitz, and Murat Arcak. 2020. Improving Urban Traffic Throughput With Vehicle Platooning: Theory and Experiments. *IEEE Access* 8 (2020), 141208–141223. https://doi.org/10.1109/ACCESS.2020.3012618

[28] R. M. Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development* 11, 1 (Jan. 1967), 25–33. https://doi.org/10.1147/rd.111.0025

[29] Martin Torngren, Dan Henriksson, Karl-Erik Arzen, Anton Cervin, and Zdenek Hanzalek. 2006. Tool Supporting the Co-Design of Control Systems and Their Real-Time Implementation: Current Status and Future Directions. In *IEEE Conference on Computer Aided Control System Design.* IEEE, Munich, Germany, 1173–1180. https://doi.org/10.1109/CACSD-CCA-ISIC.2006.4776809

[30] Kenneth Vollmar and Pete Sanderson. 2006. MARS: An Education-Oriented MIPS Assembly Language Simulator. *SIGCSE Bull.* 38, 1 (March 2006), 239–243. https://doi.org/10.1145/1124706.1121415

[31] Victor M. Zavala and Lorenz T. Biegler. 2009. The advanced-step NMPC controller: Optimality, stability and robustness. *Automatica* 45, 1 (Jan. 2009), 86–93. https://doi.org/10.1016/j.automatica.2008.06.011

Revised 10 March 2025