Making Time Make Sense in Robotic Simulation

James R. Taylor, Evan M. Drumwright, and Gabriel Parmer

Department of Computer Science, George Washington University, USA
{jrt,drum,gparmer}@gwu.edu

Abstract. Typical dynamic robotic simulators model the rigid body dynamics of robots using ordinary differential equations (ODEs). Such software libraries have traditionally focused on simulating the rigid body dynamics robustly, quickly, and accurately toward obtaining consistent dynamics performance between simulation and *in situ*. However, simulation practitioners have generally yet to investigate maintaining *temporal* consistency within the simulation: given that simulations run at variable rates, how does the roboticist ensure the robot's control software (controller, planners, and other user-level processes) runs at the same rate that it would run in the physical world? This paper describes an intersection of research between Robotics and Real-Time Operating Systems that investigates mechanisms for addressing this problem.

Introduction

Dynamic robotic simulators are one of the most widely used software tools in the field of Robotics today. Some of the recent focus on these simulations has been making them faster (*e.g.*, as with `Gazebo` in the DARPA Robotics Challenge), but one ongoing goal for rigid body simulations in general has been greater physical accuracy. The desire is that the simulations should evince physical behavior as close as possible to the real world, whether that closeness is measured quantitatively (as shown in recent experiments by Vose *et al.* [8]) or qualitatively (as can be seen in the unusual behavior observed in rigid "toys" like the Rattleback [4]). Clearly, Robotics will benefit as the physical accuracy of these systems becomes more faithful to reality: planning, optimization, and validation are just a few areas that can reap substantial improvements with better physical fidelity.

Our recent work [7] has broached an issue that has been safely ignorable for simulating dynamics for computer gaming and computer animation applications (the original focus of some popular simulation libraries like `Gazebo` and `ODE`): the *temporal consistency* between the simulation library and the "user level" software (controllers, planners, perception loops) for directing the robot. This issue arises because the simulation software does not simulate at the rate of "wall clock" time; as a result, the user level software—which may be expected to feed commands to and pull state from the simulation at roughly the same rate it would perform those operations on a physically situated robot—can not be expected to exhibit similar performance in simulation as *in situ*.

This paper continues investigating the issue of temporally consistent simulation. In addition to the general scheduling mechanisms and accurate timing of

D. Brugali et al. (Eds.): SIMPAR 2014, LNAI 8810, pp. 1–12, 2014.

a controller interfaced with dynamics that we explored in our previous work, we have further developed our infrastucture to ensure time consistent sharing of system resources among multiple controllers and planning processes. Additionally, we use a simple experiment to demonstrate the effect that neglect of this issue can have on robotic systems.

1 Background

1.1 Conventional Robotic Simulation Paradigm: The *Callback Model*

Current simulation architectures leverage a simple model whereby the dynamics executes for steps of time, and after each step the simulator invokes a call-back function which defines the planning and control for the system. This function can access dynamics state and it can actuate the system by adding virtual forces and torques. Importantly, the callback function can execute for an unbounded amount of time, without the time in the simulator progressing at all. The implication of this is that a significantly intelligent planner with correspondingly large computation times will execute on the *exact* environment it used as input, thus ignoring planner computation time. If this system were transplanted into a real environment the physical state of the system would diverge from the plan, possibly irreparably, by the time such a planner returned a sequence of controls for the system.

1.2 Simulation Components

Simulators integrate user defined client components together with dynamics libraries into a single framework such that a particular scenario may be evaluated. We define *client processes* to be the set of controllers and planners evaluated in such a system. Multiple robots might be simulated, each with sets of client processes. Collections of robots and environmental obstacles form a scenario. Each of the robots has a set of sensors and actuators that are controlled by the client processes in the system.

1.3 Temporal Requirements

Temporally consistent simulation attempts to ensure temporal consistency between different software components, and a simulation environment (*cf.* the notion of consistency for real-time aware, distributed shared memory accesses in Singla *et al.* [6]). The goal of temporally consistent simulation is not adhering execution to real-time, but rather ensuring consistency between the virtual progress of the dynamics, and the computational progress of the robotics software.

Real-time operating systems (RTOS) could be used to provide the same functionality as our temporally consistent simulation design on Linux, but would surprisingly not provide many additional assurances. The focus of RTOSes is

often to control and bound latency for I/O. As temporally consistent simulation replaces traditional I/O with interfaces for coordinator and dynamics interactions, such RTOS facilities are superfluous.

A difficulty in providing a temporally consistent simulation infrastructure is that the timing requirements for the client processes vary significantly, and that they require accurate timing from the system. A system's sensors and actuators often have a natural frequency at which they provide environmental data or take actuation commands. Correspondingly, controllers are often executed at a rate closely tied to those sensor and actuator frequencies (i.e. their rates of input/output), and thus execute in a *periodic* manner. Specifically, they are activated by the OS every N milliseconds, at which points they do their computation, and block waiting for the next periodic activation. Note that if a controller *overruns* its computation, it might finish execution only after an activation. This is often called *missing a deadline*, and can result in instability. In contrast, planners often execute irregularly, do as much computation as is required, and provide their output as fast as they can, but not on a tight schedule as with controllers. These computations are often called *best-effort*. Planners often also provide higher-level sets of commands to the controllers via Inter-Process Communication (IPC) channels.

In a temporally consistent simulation, the activations of periodic controllers must be as time-accurate as possible, and the computation for the planners must not interfere adversely with the controller's ability to meet deadlines. The most difficult timing requirements, however, derive from the interactions between client processes and the external world, and between client processes. If the simulated time within the dynamics gets too far ahead of the amount of time that the client processes have executed, then sensor data will reference data "in the future". Alternatively, if the simulated time in the dynamics lags behind computation, then actuator commands will be sent to stale dynamics state. Comparably, the planner and controllers must be temporally kept in sync for the same reason. This is at the core of temporal consistency: all aspects of the simulation environment must proceed at rates that are realistic, and in sync.

1.4 System Scheduling toward Temporally Consistent Simulation

In attempting to address temporally consistent simulation, this research requires an infrastructure that can control not only the rate of progress of a dynamics engine (which is often provided naturally by it's API), but also of the *execution progress* of multiple client process computations. Put another way, the temporally consistent system infrastructure requires control over the scheduling of the system. Unfortunately, there is a *semantic gap* between what scheduling facilities the kernel of the system provides (often a black-box), and what is required by the simulation architecture. Many previous research projects in the area of operating systems have attempted to solve this problem. For example, [5] and [3] provide system infrastructures that are extensible, enabling normal user-level processes to define their own scheduling policies. However, they both require drastic system changes to provide these infrastructures. Alternatively,

[2] and [1] attempt to create an environment in existing systems in which some of the timing characteristics can be controlled by user-level code. Our research continues with this trend by creating a multi-process simulation environment in which the *coordinator* plays the role that the kernel traditionally takes: it controls system scheduling (i.e. the interleaving of different client processes, and the dynamics), and communication. However, it is designed to execute at user-level using only the facilities and APIs provided by a POSIX-compliant OS such as Linux. Thus, users need not modify their underlying systems at all.

The coordinator must address three main challenges: (1) how can Linux's POSIX-like API be leveraged to control scheduling and communication; (2) how can abstractions be provided to client processes so that they read sensor data, send actuator commands, and communicate via IPC normally; and (3) how can client computation be scheduled alongside the dynamics engine's virtual time?

2 OS Facilities for Temporally Consistent Simulation

2.1 Timing Facilities

The goal of the temporally consistent system is to ensure that for a number of client processes, and the dynamics, time progresses consistently for all. A key concept for temporally consistent systems is the maximum deviation in this progress, which we define in the following. We denote each of the client processes and dynamics as $\{p_0, \ldots, p_n\} \in P$. $p_i \in P$ has executed (or had simulated time progress) an amount of time e_i^t by time t within a given simulation infrastructure. Each periodic process blocks waiting for its next activation, thus essentially moving its own computational progress forward by the amount of time it waits, w_i^t. This wait time is common for controllers that activate periodically, but don't use all execution time until their next activation. Thus we define the temporal drift of the system, Δ, as such:

$$\Delta = \max_{\forall t}\{\max_{\forall p_i \in P}(e_i^t + w_i^t) - \min_{\forall p_i \in P}(e_i^t + w_i^t)\}$$

Intuitively, Δ is the maximum deviation in temporal progress between any two parts of a simulation. A perfectly temporally consistent system is one in that $\Delta = 0$, while a traditional callback-driven simulator with a planner that always executes for more time than a step in the dynamics has $\Delta = \infty$ as $t \to \infty$.

Commodity hardware features timing facilities that are based on somewhat granular units of time. For example, timer ticks provide preemptive execution to prevent system starvation from a single process, which occur at a minimum fixed interval (for example, 100 or 1000 times a second). POSIX-based operating systems provide APIs for accessing timers and execution accounting facilities. Thus, the granularity for executing processes on modern systems is somewhat large. If this is bounded by 10ms (the timer inter-arrival on our system), then $\Delta \geq$ 10ms. Thus our temporally consistent system attempts to minimize Δ within the confines of the hardware and OS provisions. However, we have found

that the choice of the OS facility used for this timing has a large impact on the accuracy of timing in the system.

Accounting for Execution Time. To track the execution progress of a client process, traditional OS facilities (such as those used in the `time` and `top` programs) have a very large granularity, and an unbounded error. Instead of relying on these very coarse grained mechanisms, we use the cycle-accurate *time stamp counter* register that is available on most processors. It is a 64 bit value that counts the number of cycles elapsed in the processor, and is accessible on x86 and x86-64 processors through the `rdtsc` instruction. To maintain accurate time using `rdtsc`, the system must (1) know the processor speed; (2) maintain a consistent processor speed (or, alternatively use "invariant time stamps" in modern processors); and (3) all client processes and the coordinator must remain active on only one, shared processor core. For (1), we read the processor speed from the `/proc` file-system, for (2), we disable all power saving and throttling features, and for (3), we confine the measured process to run on a single processor core using the `sched_setaffinity(.)` system-call family. Thus our system can cycle-accurately account for the execution time (e_i^t) of each client process.

Accounting for Wait Time. To track the wait-time (w_i^t) for a process, our system provides an API similar to `setitimer` which enables recurring, periodic activations. Controllers use our API to schedule periodic activations, and after their computation for a specific activation is complete, they become inactive waiting for the next activation. The system tracks this elapsed wait time until the process is again executed. Notably, this wait time does signal some temporal progress for those controllers, even though it does not include computation time, thus why we consider it in the calculation of Δ.

Granularity of Preemption. For any scheduling system to control the execution of unknown computation (that might, for example, contain an infinite loop), preemption is required. A significant flaw of the callback model is that it executes the planner non-preemptively – the simulator cannot stop the planner when it has executed for too long. The hardware provides timer interrupts as its basic mechanism for preemption. POSIX provides a number of facilities for notification of timer interrupts. Our coordinator uses signals associated with the timer to receive these notifications. When the hardware causes a timer tick, the OS vectors it into a user-level signal that switches away from the previously executing client process (*e.g.*, planner), and to the coordinator, where scheduling decisions can be made. This, combined with the accurate execution time explained above, provides the temporally consistent system with the main facilities it requires to manage timing.

2.2 System Scheduling

The default scheduling policy in Linux is `SCHED_OTHER` that makes no guarantees on when any thread in the system will make progress. In contrast, it also includes two real-time policies for first-in-first-out, non-premptive, fixed priority scheduling, and preemptive (round-robin), fixed priority scheduling—`SCHED_FIFO` and

`SCHED_RR`, respectively. These policies are *predictable* in that if two processes both want to run on the CPU, the higher priority one will always be chosen to execute. A process can be set to be scheduled using any of these policies via the `sched_setparam` system call.

Context Switching. We take advantage of the predictable behavior afforded by these kernel scheduling policies to implement our own scheduling policy in the coordinator. The coordinator itself always executes at the highest priority. The client process it wants to switch to will be given the next highest priority. To finish the switch to that process, the coordinator will block waiting for timer interrupts, actuator commands, blocking notifications, and IPC (blocking on multiple sources in POSIX can be done with `select`). If any of these are detected, it will wake, and immediately activate (as it is highest priority). Whenever the coordinator executes, it makes a scheduling decision about which client process/dynamics should run next to optimize for a minimal Δ.

Process Blocking. In attempting to override the scheduling policies of the kernel, the coordinator must consider the case when a client process blocks. For simplicity, in this work, we assume that the client processes only block waiting for timeouts (periodic controllers), or waiting for sensor data. Without accounting for blocking, the coordinator might lose control of the system: if the process that is supposed to be executing instead blocks, then the kernel will take over and choose the next highest thread to execute, which might not be in the time consistent system, thus invalidating the coordinator's execution accounting.

2.3 Interprocess Communication

Communication between client processes (*e.g.*, the planner sending commands to the controller), sensor data requests, and actuator commands require the coordinator to mediate the communication. Each client process is given access to two pipes that are used to (1) block the process waiting for an event (*e.g,*. IPC, sensor data), or (2) send a notification to the coordinator that the process is sending data (*e.g.*, IPC, actuator commands). The coordinator is awakened by such notifications and can decide where to copy the data (it is in shared memory) or how to manipulate the dynamics.

3 Time Consistent Simulator Design

The time consistent simulator must manage multiple conflicting goals. On the one hand, the timing requirements of controllers require the meeting of deadlines (*i.e.*, an accurate activation time), and on the other, temporal consistency is required to have a clear mapping between actual system execution, and simulated execution. As the simulated environment must support multiple robots and varied software infrastructures with rich communication structures, it must be highly flexible. This section covers the implementation of the infrastructure, and details how it integrates with the OS facilities from Section 2.

3.1 Hierarchical Scheduling and Threads

To handle the required generality, we use a hierarchical scheduling framework [5]. Such systems define a tree of schedulers. The leaves of the tree are client processes, and the dynamics. When activated, the *root* scheduler determines from its children which to execute (*i.e.* it makes a scheduling decision), and does so using a polymorphic method invocation to `dispatch` the child. If the child is a leaf, then the dispatch function will either (1) use the context switch mechanism described in Section 2.2, or (2) make an invocation into the dynamics to step the simulated time forward. However, as the system is hierarchical, the dispatched child could be a scheduler itself. The key insight here is that each of the schedulers in the hierarchy can define different scheduling policies.

Scheduling Policies. There are two policies we use in the system, one to maintain minimal temporal consistency, and the another to do strict priority-based scheduling. As a general rule, the schedulers close to the root are concerned with temporal consistency, while those that represent an actual scheduler on a robot are concerned with maintaining accurate timing for the system controllers, thus placing them at a higher priority than the planners. Though more scheduling policies could be added, we found that these are sufficient.

Example Use of Hierarchical Scheduling. The system set-up we use in Section 4 includes a dynamics engine and two robots, one with both a planner and a controller, and the other with a simple controller. All of the three threads for the robots, and the dynamics must be scheduled. Thus, we organize the system with a single root scheduling for consistent timing between each robot, and the dynamics. The robot with both the planner and the controller has each under a scheduler with the fixed-priority policy, with priority going to the controller. The hierarchical arrangement of schedulers and the dynamics and client processes is essential to properly schedule given the different goals of different parts of the system, and to enable the simulation of complex, possibly multi-robot systems.

Maintaining Proper Accounting in the Hierarchy. Just as scheduling decisions follow a chain from the root to a leaf, the accounting for execution time and progress must go from leaves down toward the root, so that scheduling decisions can be made at each scheduler based on an accurate rendition of how much temporal progress all of its children have made. For this, we aggregate the execution times of all children, unless they are all blocked waiting for sensor data, in which case we determine that child's progress to be the minimum of those block times.

3.2 Coordinator Design

The coordinator is the heart of the system and orchestrates all execution. The hierarchical scheduling policy is executed in the coordinator, and when a dispatch is made to a client process, the coordinator goes through the following steps: (1) swap the priorities of the previously active client process, and the one we want to switch to (Section 2.2), (2) take a time-stamp reading (Section 2.1), and (3) block the coordinator (on `select`) waiting for an event (timer tick or

request for sensor data). As the coordinator (which is highest priority) blocks, the system naturally switches to the new process, thus completing a context switch. Unblocking the coordinator (and subsequent blocking of the active client) is accomplished by writing notifications to the pipes that will wake the main coordinator thread of execution. Client processes explicitly notify the coordinator of servicing needs and scheduling demands by sending `read`, `write`, and `idle` notifications including timestamp to the coordinator which result in yielding by the client process and rescheduling per their scheduling policy. The coordinator also implements a real-time monotonic timer via `timer_create` that periodically sends a `timer` notification including timestamp. The timer notification ensures the coordinator interrupts a long running (typically *best-effort*) client process such that all client processes (especially *periodic*) are given fair access to the processor on a regular basis and no client process can starve all others.

Table 1. Process Priority Assignment

Priority	Level	Process	Description
p_c	ℓ_0	coordinator	highest real-time priority for the OS
$p_c - 1$	ℓ_1	active client	the currently dispatched client
$p_c - 2$	ℓ_2	block detection	reserved for a block detection process
$p_c - 3$	ℓ_3	waiting clients	all other waiting (or blocked) clients

Coordinator Initialization. System boot-up is a delicate process that we detail here. Upon initialization, the coordinator is bound to the CPU, set as a real-time process with highest priority ℓ_0 (Table 1), opens pipes for IPC, opens the shared memory, intializes the dynamics system, creates all client processes, and initializes the timer. Creation of a client process involves wrapping the process with a client thread, forking a new process, scheduling with the system as a real-time process and with real-time priority ℓ_3, binding to the same CPU as the coordinator, disabling console interaction, and launching the executable file via `execl`. Console interaction is disabled after the fork by forwarding `stdin`, `stout`, and `stderr` to `/dev/null` to minimize blocking system calls within these processes, which might disturb the coordinator's control over timing. When a client process is dispatched, the coordinator raises the client system priority from ℓ_3 to ℓ_1, and yields to the dispatched child by blocking via `select`. When the client process publishes any notification to the coordinator, the coordinator unblocks, preempts the client process, and lowers the client system priority from ℓ_1 to ℓ_3.

3.3 Client Process

A client process is an external main function program that must provide facilities for opening the shared buffer, for sending read, write, and idle notifications on prescribed channels, and for executing its own computation code. A client process must be preregistered with the simulation such that it is linked to the

corresponding dynamic body, is described as a controller or planner, is classified as either periodic or best-effort, and has IPC facilites prepared. Forking the coordinator and executing the external program, inserts the external program into the process space of the coordinator as a child process and inherits the established IPC channels. Notifications of process `read`s indicate requests for simulation state (*e.g.*, reading sensor data). The coordinator services these `read`s using shared memory to pass data. Client process `write`s correspond to either a controller sending commands to actuators, which are correspondingly interpreted to manipulate dynamics state, or they correspond to a planner sending the plan to the controller via the coordinator. Finally, client processes send `idle` notifications to the coordinator to yield until the next activation (*e.g.*, for periodic controllers).

4 Experimental Validation

Our experiments aim to illustrate the performance discrepancy between systems using callback functions and our time consistent system. The experiments have been designed to reflect our experience with building software for both simulated and physically situated robots; this decision results in a few discrepancies between the time consistent and callback-based systems that will be noted below.

Our experimental scenario uses a predator-prey scenario with two identical "space ships" (*i.e.*, rigid bodies moving freely in SE(3) via application of forces). The ships are constrained to move within a cubic region of space; when a ship attempts to move out of this region, a spring-like penalty force pushes it back toward the free region.

4.1 Predator and Prey Behavior

The prey is driven by a simple control policy, which enacts either a random walk (we save the seed so that we can reproduce the walk across trials) or a fleeing behavior, depending on the distance of the predator. The prey flees by moving directly away from the predator using limited force.

The predator uses kinodynamic planning to chase the prey by exploiting the latter's deterministic behavior when the predator gets sufficiently close. Indeed, given ample planning time, the predator should be able to plan to intercept the prey by using the prey's deterministic movement model and an inverse dynamics model (that determines the requisite forces to achieve a target acceleration).

Planning and Control. We instituted our own kinodynamic planning mechanism which applies controls, integrates its models of the predator and prey forward in time, and finds a plan that brings the predator closer to the prey. Our initial efforts used `OMPL`, but the inherent multi-threaded nature of the library and its use of wall-clock time for determining when the planner should terminate confounded our system's efforts to schedule the planning process. Using wall-clock for process timing assumes that the process will not be scheduled-out,

so the planning time parameter in running `OMPL` in the context of a real-time system can only be considered an idealized upper bound. The planner is allowed to execute for a maximum time (1.0s) and the resulting plan is not executed beyond a maximum duration (0.1s); beyond this point the open loop execution of the plan by the predator tends to lead to it becoming dynamically unstable.

The planner is called differently on the time consistent and callback-based systems. On the callback-based system, the planner is called only when all of the commands from a previous plan have been executed (or the plan has become stale by going over the maximum allowable duration). The time consistent system calls a planner in a manner analogous to operation on a real robot: (1) before a plan arrives, the predator executes "no-op" commands (*i.e.*, it applies no force and no torque); (2) the planner attempts to find a plan (the predator continues to execute "no-op"s at this time); (3) when a plan is found, the predator begins executing the plan and immediately calls the planner to begin planning again; (4) the planner keeps executing that plan until the sequence of commands is complete or the planner notifies the controller that a new plan is available.

The predator controller uses a composite feedforward (*i.e.*, the planned commands) and negative-feedback controller to account for error between its current state and the desired outcome. The prey uses a simple control policy only.

4.2 Time Consistent System and Callback-Based-Systems

The time consistent system was built on top of an otherwise unmodified version of `Moby`. For comparison we used two callback-based systems, "vanilla" `Moby` and `Gazebo/ODE`. Each simulation was run with a 0.01s dynamics time step, a maximum planning time of 1.0s, a planning step size of 0.01s, and both controllers running at a frequency of 100Hz. Our experiments were run on Linux kernel 3.2.0 ("vanilla" Ubuntu 12.04) using a 2.80GHz Intel Xeon quad-core processor.

4.3 Experimental Specifications

All scenarios start in the same configuration with the predator and prey halted and separated by ten meters and a flee triggering distance of five meters. Scenarios were simulated for twenty seconds of simulation time, and each experimental trial consisted of running the scenario with identical random seed for the prey using the three systems: `Gazebo`, "vanilla" `Moby`, and modified `Moby` (the time consistent system).

4.4 Experimental Results

The results of our experiments, depicted in Figure 1, show that the simulations based on the callback model yield virtually identical statistical behavior while the time consistent simulation exhibits dissimilar behavior. Because the simulation state is frozen during planning in the callback model simulations, the predator is consistently able to plan from its current state to the current state

of the prey, which allows it to maintain close proximity at nearly all times. The statistical distributions for the callback-based systems are centered within the flee triggering distance with a maximum distance equal to the initial distance. In the time consistent simulations, the predator is able to approach the prey for only short durations and the statistical distribution is centered more closely to the starting distance and exhibits high variance. Animated renderings of the simulations show that the predator tracks the prey very closely in the `Gazebo` and "vanilla" `Moby` simulations while the predator generally undershoots or overshoots the prey's position in the time accurate system.

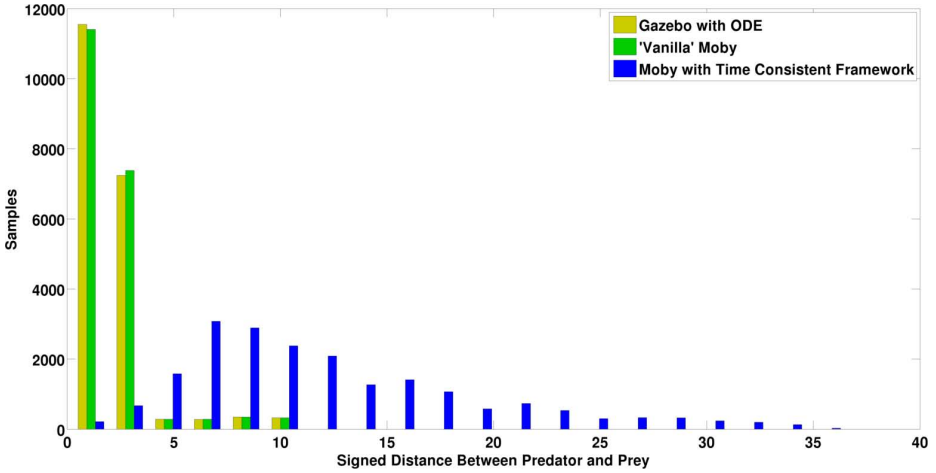

Fig. 1. Histogram showing the instantaneous distances between predator and prey over ten trials (2,000 samples per trial). Systems based on the callback model are effectively able to plan while the predator and prey are frozen in time, while the time consistent system must plan and act in real time. Consequently, the predator is able to stay much closer to the prey in the systems based on the callback model.

The predator in the callback-based systems is able to maintain much smaller distances to the prey solely because the predator's planner is able to execute proportionally for much longer without the danger of plans becoming stale. At the end of 20 simulated seconds, the planner in the traditional callback-based systems consumed on average 188 seconds, thus yielding $\Delta = 168.01$ seconds. The ratio of planner execution time (1.0s) to planner frequency (0.1s) indicates that the planner runs for 10 times longer per second than the simulated time progresses and approximates $\Delta = 10t$. In contrast, $\Delta = 0.003$ seconds for the time consistent system; though Δ is non-zero (due to intrinsic hardware limitations), its value is independent of the time that the simulation runs.

In a follow-up experiment, we measured the overhead of our components for enforcing temporal consistency. For a single trial, the overall system ran twenty seconds of simulation time in a real-time of 21.22 seconds during which the

system spent 0.42 seconds coordinating, 20.21 seconds running planners and controllers, and 0.59 seconds stepping dynamics. From this result, we estimate our framework adds 2% of overhead, which we argue is acceptable.

5 Future Work

For future work, we will investigate augmenting our system to increase time accounting accuracy by detecting unconstrained blocking system events (which most non-real-time software triggers) and client process unblocking events, which will allow us to better support existing libraries like `OMPL` without requiring modifications to the libraries themselves. To take full advantage of current system architectures, we will also scale the coordinator scheduling system to utilize multiple cores and multiple processors. We will support simulations for which time does not proceed monotonically (like those that use adaptive integration).

Acknowledgements. This work was partially supported by NSF CMMI-110532.

References

1. Anderson, J.H., Mollison, M.S.: Bringing theory into practice: A userspace library for multicore real-time scheduling. In: Proc. IEEE Real-Time and Embedded Technology and Applications Symp (RTAS), pp. 283–292 (2013)
2. Aswathanarayana, T., Niehaus, D., Subramonian, V., Gill, C.: Design and performance of configurable endsystem scheduling mechanisms. In: Proc. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS), pp. 32–43 (2005)
3. Ford, B., Susarla, S.: Cpu inheritance scheduling. In: Proc. USENIX Symp. on Operating Systems Design and Implementation (OSDI), pp. 91–105 (1996)
4. Mirtich, B.: Impulse-based Dynamic Simulation of Rigid Body Systems. PhD thesis, University of California, Berkeley (1996)
5. Parmer, G., West, R.: HiRes: A system for predictable hierarchical resource management. In: Proc. IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS) (2011)
6. Singla, A., Ramachandran, U., Hodgins, J.: Temporal notions of synchronization and consistency in beehive. In: Proc. ACM Symp. on Parallel Algorithms and Architectures, SPAA 1997, pp. 211–220. ACM, New York (1997)
7. Taylor, J.R., Drumwright, E.M., Parmer, G.: Temporally consistent simulation of robots and their controllers. In: Proc. ASME Intl. Design Engr. Tech. Conf. and Comput. and Inform. in Engr. Conf., Buffalo, NY (2014)
8. Vose, T., Umbanhowar, P., Lynch, K.M.: Friction-induced velocity fields for point parts sliding on a rigid oscillated plate. Intl. J. of Robotics Res. (June 2009)