Incorporating Physical Dynamics Into Systems Mechanisms

David Ferry
Saint Louis University
College of Arts and Sciences
St. Louis, Missouri
dferry@slu.edu

ABSTRACT

Modern computer systems are time based systems. However, cyber-physical systems (CPS) reason about physical dynamics and time simultaneously. This suggests that physical dynamics could play a much larger role in implementation of systems mechanisms and operating systems for CPS. This paper posits two dynamics-based mechanisms that have potential use in CPS operating systems, and then imagines how the relationship between time and dynamics could inform an implementation of these mechanisms. First we consider applying dynamics to the traditional timer wheel algorithm to support the efficient creation and deletion of many physics-based event notifiers and error handlers. Second we consider using the physical state of a system to identify grace period completion in the context of the read-copy-update (RCU) mechanism used for highly efficient synchronization between many readers and few writers.

1 INTRODUCTION

Tight coupling of computation with physical dynamics (position, velocity, acceleration, etc.) is the distinguishing feature of modern cyber-physical systems (CPS). Yet, time is usually seen as the dominant driver of system behavior, because accurate time-based scheduling is ubiquitous and many CPS platforms are tightly coupled with control systems and real-time scheduling theory where the analysis of time is paramount. However, while time provides a convenient mechanism to quantify the operation of the system, we can ask whether there are cases where considering the physical dynamics of a system could lead to simpler or more efficient implementations of systems mechanisms.

As a position paper, this work asserts that the physical state of a system contains enough useful information to control the flow of computation in a system in meaningful ways. We are used to viewing the physical dynamics of a CPS as changing over time, but in Section 2 we can instead view time as the quantity that changes in response to changes in physical position, velocity, acceleration, etc. This leads us to a principle of "far in distance implies far in time" that helps determine when and how certain computations should take place. We are also used to physical dynamics modifying the control flow of a user program, such as through branching logic, but it is less used to inform the management of system resources, as we will do in Section 3. It is tempting to view cyber-physical systems as a physical system and a traditional computer glued together, but we instead should seek to understand where a system that is always aware of its physical state can go beyond traditional techniques.

In Section 2 we consider the implementation of efficient handling of dynamics based events, specifically the monitoring of distance thresholds between the system and points that should be avoided. When such a separation constraint is violated a software event is fired, but what if we want ubiquitous error checking and handling where we may have hundreds or thousands of processes all simultaneously monitoring different thresholds? A naive O(N) algorithm would be to simply check the distance periodically for each threshold, and if ever the physical quantity of interest achieves the desired value the event is triggered and handled. While this works, we draw inspiration from the timer wheel algorithm[9] to ask whether incorporating physical dynamics into our understanding of the situation would lead to a more efficient implementation. In particular, we observe that the formula "distance = rate × time" can be interpreted as implying that a large distance to cover implies a large time until event dispatch occurs, and this informs us as to which events may need to be handled in the near future.

In Section 3 we suppose we want efficient synchronization of a shared data structure. If we can connect data access patterns to the physical dynamics of the system, then could we use those dynamics to achieve efficient and scalable synchronization in the context of read-copy-update (RCU) synchronization[5]? We speculate about a dynamics-based RCU grace period for synchronizing shared data based on the physical state of the system.

These examples ask a more general question: where can physical dynamics be incorporated into existing system support for cyber-physical systems? Existing operating systems are essentially "physics agnostic," and yet achieve a high degree of function. This means that the current state of systems support forms a lower bound on how well mechanisms can be implemented. But, are there venerable algorithms that could be done better with the inclusion of dynamics? Incorporating the physics of a system into the OS or system support middleware only increases the information and resources we have available to us. How can we find the mechanisms that will surprise the entire research community by working better when included in a robot or a self-driving car than when implemented in a stationary desktop computer with no notion of dynamics? Conversely, what algorithms will fail to transfer efficiently to the realm of physical dynamics and why?

2 EFFICIENT DYNAMICS BASED EVENT DETECTION

How can event-based triggering be made to scale efficiently? We can envision CPS architectures where very many triggers must be handled simultaneously, or where event handlers are installed and removed frequently. In these cases simple data structures with reasonable performance exist, but can we do better than this? A similar problem arises in the maintenance of timers in traditional OS design.

NGOSCPS, 2019, CPS-IoT Week, Montreal Canada
.

There are two types of timer-based operations supported by the Linux kernel. The first are timers, which are used when a user process wants to be notified when a certain time has elapsed. These are generally expected to expire and usually require handling. The second is timeouts, which are commonly used for error-checking blocking operations that are expected to succeed but should not block indefinitely if they fail (e.g. network related calls such as establishing a TCP connection or sending a TCP packet). This second class of timeouts is interesting because (1) it is the far more common use case of timing in user software and consequently (2) at runtime there are far more of them. Ingo Molnar described this as the "millions of network timers" problem[7]- many timeouts are created and handled, but almost all of them never expire. If the blocking call succeeds, then the timeout timer is canceled, and the operation is finished. Thus, it makes sense to optimize for the case where non-expiration is the expected outcome.

In our hypothetical CPS architecture we may have hundreds or thousands of processes that want to use triggers to do fault detection, and the triggering criteria change frequently. In such a system each process would register an event handler that triggers when an error condition occurs, but because the use case is error handling, these triggers are not actually expected to fire. This gives rise to an analogous "millions of error handlers" situation. How can this be handled efficiently?

In the Linux kernel, timer expirations are handled through a data structure called a timer wheel[1]. Timers are sorted into buckets based on what jiffy they will expire in the future. The first bucket contains all those timers that will expire soon, the second bucket contains timers that will expire shortly after that, and so on until we get to a final bucket, which is all those timers that expire in the far future. These buckets are also arranged by jiffy, so that all jiffies for the short term future have a specific place in each bucket. The result is that all jiffies have a known location, so adding and removing timers are O(1) operations with this data structure, and the most common use case of adding and removing a timer without ever expiring it is fast. Each jiffy the time keeper for the first bucket is advanced by one, and if there are any timers to expire they are done so then. When the first bucket is exhausted, the second bucket cascades down to the first position, and subsequent buckets cascade down as well[1], resulting in an O(N) cascade. Maintenance of this data structure is amortized O(1) with the O(N) cascades happening periodically.

Such an approach does not apply directly to the case of dynamics event notification. Imagine a system that wants to maintain distance separation between itself and other points in the world. Whenever a new point becomes tracked the various processes in the system register their event handlers that trigger when the separation falls below a specific value. We must track these event handlers, but none of the classic data structures seem reasonable for the "millions of event handlers" case where insertion, deletion and maintenance are hopefully O(1). Lists result in O(N) time, priority heaps result in O(log(N)) time, and trees also result in O(log(N)) time for the common non-expiration case, so there is no obvious candidate. A timer wheel analogue would be nice, but there are several key assumptions not stated about timer wheels that break when instead we want a distance-measure wheel:

- Time progresses smoothly and predictably.
- Time progresses only in one direction.
- Time progresses equally for all timers.

The timer wheel algorithm depends on a predictable progression which is lost when we are tracking distance rather than time. The distance between the system and a point can decrease or increase, the rate of change of distance varies based on speed and direction, and when we're tracking multiple points the distance between these points and the system may be increasing, decreasing, or unchanging over time. However, we can still take inspiration from the timer wheel notion of binning various timeouts by their future expiration. For firing events the direct analogy would have us assume that all our tracking points lie on a line in front of us, and then in order to fire events we could say that each slot on our distance wheel represents a 1m interval, so the current position of the wheel indicates events less than 1m distant, while the second slot are those between 1m and 2m distant, then the third is those events 2m to 3m distant, etc. When the system moves forward by a meter then the wheel pointer advances one slot, bringing all those events 1m closer.

However, it's the maintenance of points that we are more concerned with. We know the quantity of distance changes continuously, which implies that event triggers far away in distance are also far away in time. By separating those points that we could physically reach from those points we could not possibly reach we need not consider the unreachable points during regular maintenance. Binning points by their distance from the system, or by their relative direction from the system, or by their location over the Earth would all provide potential avenues to categorically ignore those points that are impossible to reach. After some interval (dependent upon velocity, location, etc.) some of those unreachable points will become reachable, and some reachable points would become unreachable. This could be done with an O(N) maintenance step so we get similar amortized maintenance time to the original timer wheels, or perhaps a more clever algorithm reasoning on physical quantities exists.

Other physical quantities such as velocity and acceleration may not all fit into the same framework. Acceleration and velocity both change much more rapidly over time, with tremendous spikes possible in both cases. In these cases we cannot nearly so easily say that a given physical state is unreachable, and perhaps the strongest assumptions we are willing to admit only permit a best case O(N) maintenance time.

Lastly, the above discussion is done in the context of error handling and fault detection, but what about control? Many recent papers have examined non-periodic control schemes, such as event-triggered or self-triggered control[4]. Event-triggered control relies on continuous monitoring of the physical state of a system to dispatch events, similar to the objective described here. Self-triggered control has the controller compute the next control update time whenever it is invoked. It must be observed that in current systems any control process is far more likely to require a timer-style

[1]The implementation of the timer wheel was changed in version 4.8 in 2016 [2] so that cascading no longer happens, but introduces tradeoffs between expiration accuracy and length of timeout. The precise differences between these two timer wheel variations are immaterial for this discussion, but it is worth noting that the new implementation avoids the O(N) cascade.

mechanism rather than the timeout-style mechanism described above, meaning that it is expected that every timer event will eventually fire, and thus the goal of optimizing for non-expiration is not applicable. However, if this were beneficial, for example if a system had many controllers that were expected to change frequently, then we would want a data structure capable of supporting this action. Self-triggered control, being dependent on self-computed timer intervals, could be efficiently supported through the existing timer wheel infrastructure (though the new version of the timer wheel described in [2] would introduce a tradeoff between how distant a timer update could be and the accuracy of that timeout expiration). Event-triggered control, being dependent on the physical state of the system, is not supported via the timer subsystem and a distance-based version of the timer wheel could benefit this approach. Indeed, existing systems with fast periodic control rates (greater than 1000Hz) must manage hardware input as a perishable and expensive resource [3], so such a mechanism could be a way to multiplex many dependent event-triggered control processes on one data source.

3 DYNAMICS BASED READ-COPY-UPDATE SYNCHRONIZATION

Read-Copy-Update (RCU) is widely used in the Linux kernel to support fast, scalable synchronization in the case of many readers and relatively few writers. The bulk of the effort in RCU is placed onto the writer-path, so that readers can safely process large data structures efficiently, while writers do extra work to ensure the safety of the reads. Under RCU there may be many simultaneous copies of a piece of shared data, and if a copy needs to linger as long as it is referenced by a reader. A major question in an RCU implementation is how the system knows when a copy of shared data is no longer necessary and it may be reclaimed. There are of course many correct solutions to this problem, such as reference-counting pointers or reader-writer locks, but there is an extremely efficient solution that exploits a clever observation of system mechanics, and in this section we question whether a similarly suitable solution exists based on the physical dynamics of a system.

Readers under RCU are said to subscribe to a specific version of the data being read. The reader acquires a pointer to the data to be read and notifies the RCU system that it is starting a read-side critical section with an RCU read lock. The RCU system must guarantee that this pointer and copy of the data are valid until the critical section finishes. The only restriction placed on RCU readers is that they may not block or sleep[2]. Writers wanting to make modifications to shared data first make a copy of the shared data. The updater will modify that copy, and then the new version of the data with updates is published atomically. Support for atomically updating large data structures is usually achieved with an atomic pointer write. After an RCU update happens new readers will see the newly published version of the data, but the original (now outdated) copy of the data still exists, and it cannot be removed immediately. Instead, the writer must wait for all outstanding readers to finish, and only then is it permitted to reclaim the old copy of the data.

The central problem in designing an efficient RCU system is the question of how one can ensure that all outstanding readers have finished their critical sections and performed the RCU read unlock action, even though there may be many such critical sections executing on many cores. As said above, solutions such as reference counting pointers or locking exists, but the RCU reader mechanism needs to be particularly efficient and scalable, and such approaches would require readers to perform expensive and contentious operations. To reason about such a solution we define a *grace period* for each old version of data which starts at the moment it was superseded, and ends once all outstanding reader critical sections (i.e. RCU read locks) that began before publication have finished (i.e. called RCU read unlock). Thus, saying that an RCU writer grace period has finished is equivalent to saying that the RCU writer can destroy the previous copy of the data that it modified, as all subsequent readers would be looking at a newer copy.

In the "RCU Classic" implementation in the Linux kernel[6] there is a simple trick that allows indirect determination of whether or not a grace period has finished, with constant time and actually zero overhead to the readers. As mentioned above, threads are not allowed to sleep or block while in an RCU read side critical section. In a non-preemptible kernel (and only non-preemptible) this means that any such critical sections will start and run to completion without interruption, so determining whether these critical sections have finished is equivalent to being able to run a new piece of test code on all cores in the system. If this test code is able to start on a core, then it means that any previous reads prior to running the test code must have finished. Simply by observing that a context switch has occurred we can verify that a grace period has ended, and thus transform a heavyweight operation (verify that all outstanding reads are finished) into a comparatively lightweight operation (run any short bit of code on all processors in the system).

It is this clever trick that tells us when an RCU grace period must have ended that inspires the usage of physical dynamics as a method of synchronization. Suppose we are using RCU for synchronization, and that the computations currently running in the system are determined by the physical state of the system. The system might be in physical state A with some set of associated computations and a set of outstanding RCU grace periods. If the system transitions to another physical state B, and such a transition guarantees that the computations started in state A have finished, then this transition to state B is equivalent to saying that all RCU grace periods started in state A have finished and the associated data may be reclaimed. Moreover, we can imagine that each grace period is associated with arbitrary physical quantities that determine when their data becomes irrelevant. Here we have used information about the physical state of the system not to do branching control flow, but to meaningfully control a system mechanism.

Importantly, the original clever trick described above only works with a non-preemptible kernel, and its cost scales with the number of cores in the system. The new clever trick proposed here has neither of these restrictions. Note that there are many ways of determining when RCU grace periods end, including scalable time-based methods suitable for use in real-time software and with preemptive kernels[8, 10].

[2]There are many flavors of RCU, and some do allow blocking and sleeping.

4 CONCLUSION

The consideration of physical dynamics in cyber-physical systems should be seen as an opportunity to develop new mechanisms and re-think existing systems approaches. Physical dynamics are more than just quantities that must be controlled- they might encode a wealth of information useful to a computational platform tasked with managing computational resources. This physical data could tell us what activities are more urgent than others, what tasks are currently executing, what data in the system is important, and possibly other things as well. Ultimately, these physical quantities could inform many aspects of system operation, and the combination of our traditional view of systems combined with this new data gives is a richer view of the world than we had before.

REFERENCES

[1] J. Corbet. 2005. A new approach to kernel timers. https://lwn.net/Articles/152436/.

[2] J. Corbet. 2015. Reinventing the timer wheel. https://lwn.net/Articles/646950/.

[3] D. Ferry, G. Bunting, A. Maqhareh, A. Prakash, S. Dyke, K. Aqrawal, C. Gill, and C. Lu. 2014. Real-time system support for hybrid structural simulation. In *2014 International Conference on Embedded Software (EMSOFT)*. 1–10. https://doi.org/10.1145/2656045.2656067

[4] W. P. M. H. Heemels, K. H. Johansson, and P. Tabuada. 2012. An introduction to event-triggered and self-triggered control. In *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*. 3270–3285. https://doi.org/10.1109/CDC.2012.6425820

[5] P. McKenney. 2004. *Exploiting Deferred Destruction: An Analysis of Read-copy-update Techniques in Operating System Kernels*. Ph.D. Dissertation. AAI3139819.

[6] P. McKenney. 2007. What is RCU, Fundamentally? https://lwn.net/Articles/262464/.

[7] I. Molnar. 2005. kernel/timer.c design. https://lwn.net/Articles/156329/.

[8] Y. Ren, G. Liu, G. Parmer, and B. Brandenburg. 2018. Scalable Memory Reclamation for Multi-Core, Real-Time Systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 152–163. https://doi.org/10.1109/RTAS.2018.00025

[9] G. Varghese and T. Lauck. 1987. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP '87)*. ACM, New York, NY, USA, 25–38. https://doi.org/10.1145/41457.37504

[10] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer. 2015. SPeCK: a kernel for scalable predictability. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 121–132. https://doi.org/10.1109/RTAS.2015.7108434