A Power Cap Oriented Time Warp Architecture

Stefano Conoci*
Sapienza, University of Rome
conoci@diag.uniroma1.it
conoci@lockless.it

Davide Cingolani*
Sapienza, University of Rome
cingolani@diag.uniroma1.it
cingolani@lockless.it

Pierangelo Di Sanzo*
Sapienza, University of Rome
disanzo@diag.uniroma1.it
disanzo@lockless.it

Bruno Ciciani*
Sapienza, University of Rome
ciciani@diag.uniroma1.it
ciciani@lockless.it

Alessandro Pellegrini*
Sapienza, University of Rome
pellegrini@diag.uniroma1.it
pellegrini@lockless.it

Francesco Quaglia*
University of Rome "Tor Vergata"
francesco.quaglia@uniroma2.it
quaglia@lockless.it

ABSTRACT

Controlling power usage has become a core objective in modern computing platforms. In this article we present an innovative Time Warp architecture oriented to efficiently run parallel simulations under a power cap. Our architectural organization considers power usage as a foundational design principle, as opposed to classical power-unaware Time Warp design. We provide early experimental results showing the potential of our proposal.

CCS CONCEPTS

• **Computing methodologies** → **Discrete-event simulation**; • **Computer systems organization** → *Multicore architectures*; • **Software and its engineering** → *Power management*;

ACM Reference Format:
Stefano Conoci, Davide Cingolani[1], Pierangelo Di Sanzo[1], Bruno Ciciani[1], Alessandro Pellegrini[1], and Francesco Quaglia[1]. 2018. A Power Cap Oriented Time Warp Architecture. In *SIGSIM-PADS '18 : SIGSIM-PADS '18: SIGSIM Principles of Advanced Discrete Simulation CD-ROM, May 23–25, 2018, Rome, Italy*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3200921.3200930

1 INTRODUCTION

Power usage has become a major concern in software applications. In this context, the objectives of power oriented design of simulation systems can be disparate. They range from the elongation of the lifetime of mobile devices involved in the simulation—as for on-line simulations or volunteer computing on mobile devices [2]—to the usage of power governors to optimize the execution of the simulation model—as for the case of Time Warp parallel simulations [4] where the CPU-core frequency is dynamically controlled in order to throttle the execution of simulation objects out of the critical path [8]. Within this panorama, we target the orthogonal objective of efficiently running Time Warp simulations under a power budget constraint. This problem is generally known in the literature as *power capping*, and is essentially related to the fact that infrastructure/system owners may decide to limit the power consumption of the employed machines for various reasons, including data-center cooling costs

A trivial way of imposing a power cap to Time Warp simulations would consists in running the Time Warp platform on top of a group of CPU-cores with properly tuned down performance states (i.e. operating frequency and voltage). This approach would deliver a scaled down computing power, resembling a scenario where the Time Warp application is executed on less performing hardware. However, we do not consider such an approach satisfactory, since it does not consider power efficiency as a core aspect in the design of the Time Warp architecture.

In this paper we take the different perspective of devising a new Time Warp architectural organization which is by design oriented to power capping. It allows us to control the power usage—hence the speed—of operations performed by the threads *selectively* exploiting different software paths for different classes of threads.

Typical Time Warp architectures are generally based on a unique control flow graph, which characterizes the execution of all the involved threads. Along this graph each thread typically executes both housekeeping operations and event processing. In our new design, by imposing different control flow graphs to the different threads, we propose an asymmetric scenario where a few threads run tasks that are more time critical, while other threads run less critical ones. Separating the tasks in such different classes leads to the possibility of lowering down the power state and/or the frequency of operations of a given CPU-core (running a specific thread)—which allows meeting the power cap—while still enabling more critical Time Warp tasks to be executed timely.

Our power cap oriented Time Warp architecture has been implemented as a variation of the architectural organization of the open source ROOT-Sim package [10]. In the concluding part of this paper we report early experimental results that show the potential of our proposal.

2 RELATED WORK

Common power capping techniques in the literature (e.g. [9]) are application-agnostic—they enforce power budgets at the level of server machines, without accounting for workload features of the

[*] Also with Lockless s.r.l.

SIGSIM-PADS '18 , May 23–25, 2018, Rome, Italy

ACM ISBN 978-1-4503-5092-1/18/05…$15.00
https://doi.org/10.1145/3200921.3200930

hosted applications. Contrariwise, we explicitly optimize the execution of Time Warp-based application under a power budget.

Regarding studies in the area of parallel simulation, the works in [1, 3] provide evidence of how using different algorithms to run specific simulation models can impact power and energy overheads. One main outcome is that parallel and distributed simulations suffer from power and energy overhead more intensely than sequential simulation. This demands for innovative designs making parallel simulation systems more prone to energy and power efficiency. Our work is exactly in this direction since our objective is to devise a Time Warp architecture specifically designed to deliver optimized performance under a power cap.

The proposal in [8] controls the processor speed—via DVFS (Dynamic Voltage and Frequency Scaling)—for optimizing Time Warp performance. Lowering down the power usage can lead to emulate a throttling scheme where excessively optimistic simulation objects are slowed down by slowing down the speed of the CPU-core/thread they are bound to. Our work is orthogonal to [8] since we do not tune the performance state of threads based on their degree of speculation, but based on the different type of tasks they execute. Moreover, [8] does not address performance optimization under a power budget, as instead we do via our Time Warp architecture.

Asymmetry—in the form of the master/worker paradigm—has been exploited in [5] to process distributed simulations on public resources and desktop grid infrastructures. This work does not cope with power budgets, thus our proposal is fully orthogonal to it, although we share some baseline system design concepts such as the idea of pipelined interactions across the asymmetric threads.

3 THE ARCHITECTURE

Our power-cap oriented Time Warp architecture is based on the idea of exploiting asymmetric thread operations to carry out different tasks. In particular, we discriminate two classes of tasks, and hence of threads:

Class-1 Forward mode processing of simulation events;

Class-2 All other tasks, namely GVT (Global Virtual Time) computation, fossil collection, state saving, rollback (including coasting forward), scheduling events to be processed in forward mode, message exchange, and so on.

Threads running **Class-2** tasks are referred to as Controller Threads (CTs). Threads executing simulation events in forward mode, namely **Class-1** tasks, are instead referred to as Processing Threads (PTs). In our architecture, threads are pinned to different CPU-cores, so that we can control the performance states of the CPU-cores—with the aim of matching the power budget—which reflects into the speed of operations performed by the different threads. In this scenario, PTs play a core role in controlling how to spend the overall power budget assigned to the Time Warp system. More in detail, running PTs on top of CPU-cores configured with lower performance states generates the scenario where the execution of the overall application workload (the actual simulation events to be processed while moving forward along the simulation time) is slowed down. However, slowing down those threads does not lead to slowing down CTs, which can be hosted on other CPU-cores, which can then be run at a relatively higher power state. This enables all **Class-2** tasks to be carried in a timely manner, which is crucial to the goodness of the runtime dynamics. In fact, literature studies have shown that fast completion of housekeeping tasks, such as rollback (including state reconstruction via, e.g., coasting forward) or GVT computation (see, e.g., [7]), is fundamental in order not to impair synchronization dynamics, and not to incur the risk of higher incidence of wasted speculative computation.

In this paper we focus on shared-memory multi-core machines, so that a CT and the PTs bound to it always have access to the same data related to the simulation execution. In any case, our approach could be generalized by adopting it on top of each individual machine within a distributed memory system and making a CT and its controlled PTs reside on the same machine. On the other hand CTs residing on remote machines may interact just like traditional threads running a non-power cap oriented Time Warp platform—as an example, they might exploit message passing in case of event-communication between simulation objects managed by two PTs, each of which associated with remote CTs.

Indicating with N_{cores} the number of available CPU-cores for running the Time Warp system, and with N_{CT} and N_{PT} the number of used CTs and PTs, respectively, we have $N_{cores} = N_{CT} + N_{PT}$.

A CT controls at least one PT, thus in our architecture the inequality $N_{CT} \leq N_{PT}$ holds. This is perfectly aligned with the idea of having fewer threads running more critical tasks in a timely manner, via higher power demand, and more threads running the normal forward workload, via lower power demand.

Each CT_j is in charge of managing the execution of a subset of all the simulation objects. It manages their event queues, by taking care of incorporating into the proper queues any new event destined to these objects, or canceling a previously inserted event in case of an incoming anti-event. CT_j also manages the state queues of the simulation objects, by taking checkpoints of their states and logging them into the state queue of the corresponding object.

CT_j associates the managed objects with its bound PTs according to a partitioning scheme. More in detail, a partition p_i of all the objects managed by CT_j is bound to an individual PT_i, meaning that the object belonging to the partition p_i can only be scheduled for forward execution on PT_i. This leads to the scenario where no two different PTs can work simultaneously on the state of a same object, thus preventing data conflicts. On the other hand, CT_j and its controlled PT_i might need to work on the state of a same simulation object, given that they carry out disjoint classes of tasks that may anyhow lead to operate on the same object memory image. More in detail, PT_i is in charge of manipulating the state of the object when an event is being processed in forward mode, while CT_j, beyond taking checkpoints, may also access the object state for reloading a previous checkpoint and reprocessing coasting forward events in case of a rollback. We recall that both CT_j and PT_i live on a same shared-memory machine so that they can both directly access the state image of a same object by relying on address space sharing. Given such a sharing of the accesses to the object state, a CT and all its controlled PTs need to put in place a scheduling mechanism to determine which of them can operate on the object state at any time, guaranteeing isolated access to prevent inconsistencies.

The scheduling of the actions on the objects' states is put in place in our architecture via the notion of *port* between CT_j and PT_i. A port is a bidirectional communication structure—still exploiting

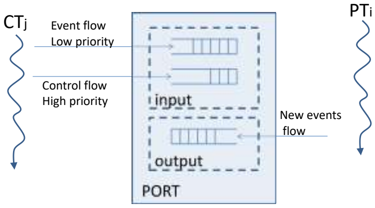

Figure 1: Structure of the port between CT and PT.

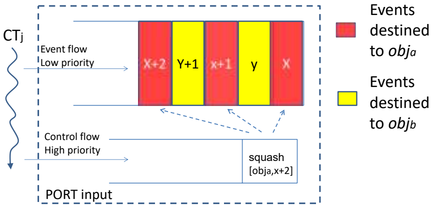

Figure 2: Example flow of event and control records.

shared memory support—based on the multiplexing of different data flows. Data that flows from CT_j to PT_i are multiplexed along two channels of the port, having different priorities. We refer to this flow-direction as *input*. Conversely, data that flows from PT_i to CT_j travels along a single channel, and we refer to this flow-direction as *output*. Figure 1 shows the port scheme.

The low priority input channel is used by CT_j to post to PT_i the events to be processed in forward mode, which are destined to the objects belonging to the p_i partition. CT_j extracts unprocessed events from the objects' event queues following the Lowest-Timestamp-First (LTF) rule, and posts them to the input channel of the port. Hence it creates into the port a pipeline of events that PT_i can extract and process, accessing the state of the corresponding objects. We note that LTF guarantees that, for each individual simulation object, the extracted events from the pipeline respect timestamp-ordering, unless (1) causal inconsistencies are revealed due to the arrival of some straggler event at that object—possibly injected by another PT_k—or (2) the cancellation of some event that passed through the pipeline, or (3) the objects produces for itself some new event with timestamp lower than another one already filled into the pipeline, which gets eventually processed. Once an event is extracted and then processed, newly produced events (if any) are posted by PT_i to the output flow of the port. These are in their turn extracted by CT_i and are incorporated into the event queues of the destination objects, if they belong to the p_i partition. Otherwise these events are sent towards the CT instance to which the corresponding objects are bound.

The accesses by CT_j and PT_i to the port are asynchronous, meaning that there is no blocking synchronization between the two threads. This allows CT_j to switch to a different PT_k it is managing whenever a port of some other PT_i—previously filled with event records—does not yet provide in output new events to handle.

Clearly, we need to include the possibility to squash portions of the current pipeline at low cost as soon as some inconsistency is detected along the flow of event records that were previously inserted, and to manage state restoration if requested because of erroneous speculation involving already processed events at some simulation object. This is the case of the arrival of a straggler event for some object, leading to the need to retract event records destined to that object which still stand into the pipeline, and to the need to rollback the object state if out-of-order processing already happened at that object. The same is true for the arrival of an anti-event annihilating some previously processed event, or one that currently stands into the pipeline. To manage these scenarios, we exploit the high priority input channel of the port, together with a mechanism that tags event records. Each event record that is inserted into the pipeline is tagged with a unique per-object identifier, which in our case is a monotonically increasing counter value. In the example shown in Figure 2, tags for events of the simulation object obj_a range from x to $x + 2$, while those destined to the object obj_b range from y to $y + 1$. If the pipelined events destined to obj_a need to be undone then CT_j inserts into the high priority input channel a squash control record bound to obj_a, and carrying the same counter value of the last event record that was inserted into the pipeline. For the example in Figure 2, the control record is structured as squash$[obj_a, x+2]$. PT_i extracts such a control records as soon as it finishes processing its last event record—given the higher priority of the control flow information in input to the port—and switches to a state where, upon extracting from the pipeline event records destined to obj_a and tagged with counter value up to the one of the squash control record, it simply discards them, thus avoiding to carry out processing tasks touching the state of the destination object. Essentially, squash tells to PT_i to ignore events destined to obj_a that still stand into the pipeline, which are no longer consistent in terms of timestamp ordering at the destination object. Note that event timestamps are uncorrelated from the counter value used to tag event records.

To indicate to CT_j that the squash message has been processed, and that the target object will be not accessed by PT_i till any new valid event record—tagged with a larger counter value with respect to the squash tag—will be posted, PT_i simply routes the squash control record to the output flow of the port. Upon detecting the presence of this control record, CT_j can safely act on the state of the target object in order to possibly restore a correct state snapshot, if requested. We note that when the squash control record is inserted by CT_j it is possible that the out-of-timestamp-order events for the destination object were only those standing into the pipeline. This is the scenario where the last event processed by PT_i for that object had a timestamp still compliant with causality—and CT_j does not need to perform any state restore action for the simulation object. To detect this condition, CT_j accesses a meta-data table, with one entry for each managed object, which is updated by PT_i with the timestamp of the last event it processed on any object. If the table-value associated with the object indicates that the last processed event had timestamp lower that the one associated with the causality violation that generated the squash, then no state restore operation is carried out by CT_j, which simply resumes filling the pipeline with event records destined to that object in renewed correct timestamp order.

Another important aspect in the separation of the tasks performed by CTs and PTs is the one related to checkpointing for

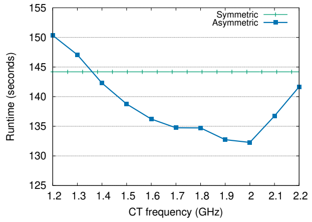

Figure 3: Performance with power cap = 30 Watt.

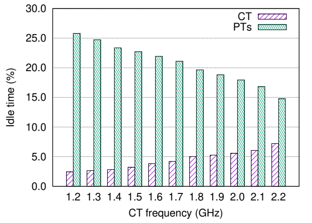

Figure 4: Percentage of idle time for CT and PTs.

creating correct snapshots for state restoration. As hinted, in our power capping oriented Time Warp architecture checkpointing is a **Class-2** task, namely one to be carried out by CTs. In order for CT_j to detect that it can safely access the state of some object to perform checkpointing, with no interference caused by forward event processing carried out by PT_i, we rely on the concept of "bubbles", similar to those used in conventional pipelined CPUs. More in detail, when CT_j determines that time has come to take a checkpoint for a given object, it inserts into the pipeline a bubble event-record, tagged with the object identifier. Upon extracting this bubble, PT_i does not carry out any processing action, rather it simply forwards the bubble towards the output flow of the port. When CT_j detects that the bubble has been posted on the port output flow it gains information that any other event that was posted before the bubble to the pipeline, which was destined to the same object, has been already processed by PT_i. Hence, CT_j can safely access in isolation the state of the object in order to take a checkpoint. Clearly, any existing policy that selects when (and of which LPs) the checkpoints should be taken can be adopted in our scheme to determine when to introduce the bubble event. The assumption for the correctness of this approach is that, once the bubble is posted to the port, no other event is posted to the port input flow for the same object till the time the bubble is observed along the output flow, and the checkpoint of the object state is taken. To achieve this, we devise a management of the objects—inspired by [6]—such that some objects can be temporarily "unschedulable' thus being not considered by the LTF scheduler.

4 EARLY EXPERIMENTAL RESULTS

To asses our proposal we have ran the widespread PHOLD benchmark, configured with 1024 objects in a bi-dimensional mesh, interacting with each other with probability set to 0.8. We set the event granularity to the coarse grain value of about 1 msec, so as to not be adverse to classical Time Warp. In fact in such a scenario most of the computation resides in forward event processing, so that housekeeping operations, which are the most critical ones when all the threads are slowed down at the same manner to meet the power cap, represent a reduced percentage of the overall computational cost of the simulation. All runs have been executed on a 10 CPU-core machine equipped with an Intel Xeon E5-2630 v4, 256 GB of ECC memory running Debian 9 with kernel release 4.9.0. The CPU frequency ranges from 1.2 GHz at P-state 11 to 2.2 GHz at P-state 1. We do not consider turbo boosting (P-state 0) in this evaluation since it cannot be easily controlled from software and it is generally power inefficient. In Figure 3 we show the variation of the execution time with power cap set to 30 Watt for classical Time Warp[1] (named Symmetric) with all CPU-cores slowed uniformly for meeting the power budget, and our architecture with differentiated slow-down of CT and PTs (named Asymmetric). In the latter case we plot the curve as a function of the power assigned to the CT, which determines the residual power budget to be assigned to the PTs. By the plot we see that our architecture allows reducing the completion time, with increased gain when we fine tune the respective power budgets to be assigned to the asymmetric threads operating within the platform. On the other hand, the Asymmetric architecture pays the cost of leading both CT and PTs to remain sometimes idle—when no work to be carried out is posted to the opposite side of the port—as shown in Figure 4 [2]. Future work will focus on the runtime optimization of the pipelined interaction across threads and on the dynamic reallocation of the power budget based on the evolution of the simulation.

REFERENCES

[1] Aradhya Biswas and Richard Fujimoto. 2017. Energy Consumption of Synchronization Algorithms in Distributed Simulations. *J. Simulation* 11, 3 (2017), 242–252.

[2] Richard M. Fujimoto and Aradhya Biswas. 2015. On Energy Consumption in Distributed Simulations. In *Proceedings of the 3rd ACM Conference on SIGSIM-Principles of Advanced Discrete Simulation, London, United Kingdom, June 10 - 12, 2015.* 99–100.

[3] Richard M. Fujimoto, Michael Hunter, Aradhya Biswas, Mark Jackson, and Sabra Neal. 2017. Power Efficient Distributed Simulation. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM-PADS 2017, Singapore, May 24-26, 2017.* 77–88.

[4] David R. Jefferson. 1985. Virtual Time. *ACM Transactions on Programming Languages and Systems* 7, 3 (jul 1985), 404–425. http://portal.acm.org/citation.cfm?doid=3916.3988

[5] Alfred J. Park and Richard M. Fujimoto. 2012. Efficient Master/Worker Parallel Discrete Event Simulation on Metacomputing Systems. *IEEE Trans. Parallel Distrib. Syst.* 23, 5 (2012), 873–880.

[6] Alessandro Pellegrini and Francesco Quaglia. 2014. Transparent Multi-core Speculative Parallelization of DES Models with Event and Cross-state Dependencies. In *Proceedings of the 2014 ACM/SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS).* ACM Press, 105–116. https://doi.org/10.1145/2601381.2601398

[7] Bruno R. Preiss, Wayne M. Loucks, and Ian D. MacIntyre. 1994. Effects of the Checkpoint Interval on Time and Space in Time Warp. *ACM Trans. Model. Comput. Simul.* 4, 3 (1994), 223–253. https://doi.org/10.1145/189443.189444

[8] Patrick Putnam, Philip A. Wilsey, and Karthik Vadambacheri Manian. 2012. Core Frequency Adjustment to Optimize Time Warp on Many-core Processors. *Simulation Modelling Practice and Theory* 28 (2012), 55–64.

[9] Sherief Reda, Ryan Cochran, and Ayse Coskun. 2012. Adaptive Power Capping for Servers with Multithreaded Workloads. *IEEE Micro* 32, 5 (Sept. 2012), 64–75. https://doi.org/10.1109/MM.2012.59

[10] The High Performance and Dependable Computing Systems Research Group (HPDCS). 2012. ROOT-Sim: The ROme OpTimistic Simulator. https://github.com/HPDCS/ROOT-Sim. (2012). https://github.com/HPDCS/ROOT-Sim

[1] This is the native ROOT-Sim implementation.

[2] In our experimental setup the CT posts to the port a batch of up to 128 events at a time towards each PT.