A Low-Overhead Constant-Time LTF Scheduler for Optimistic Simulation Systems

Tiziano Santoro and Francesco Quaglia
DIS, Sapienza Università di Roma

Abstract—We present an implementation of the Lowest-Timestamp-First (LTF) algorithm for the identification of the next Logical Process (LP) to be dispatched in context where the optimistic simulation kernel conforms the best-practice of keeping separate event lists for the hosted LPs. The implementation provides low-overhead, constant-time dispatching. We release our implementation within the open source ROOT-Sim optimistic simulation platform. Experimental data are also reported supporting the effectiveness of our proposal.

I. INTRODUCTION

A traditional way to achieve high performance simulations is the employment of parallelization techniques. They are based on the partitioning of the simulation model into Logical Processes (LPs) that can execute events in parallel on different CPUs and/or different CPU-Cores, and rely on synchronization mechanisms to achieve causally consistent execution of simulation events at every LP.

In this article we cope with optimistic synchronization (based on rollback for recovering possible timestamp order violations due to the absence of block until safe policies for event processing), and address the selection of the next LP to be dispatched in scenarios where multiple LPs are hosted by the same instance of the simulation kernel. This is the typical case for (very) large models, where the number of LPs can be (significantly) greater than the number of available computing elements.

As for the above issue, the most commonly adopted CPU scheduling policy is Lowest-Timestamp-First (LTF) [7], according to which the next LP to be dispatched is the one whose next event has the minimum timestamp across all the LPs hosted by the same instance of the simulation kernel. LTF is attractive for its simplicity, and especially because it does never cause out-of-order event processing across LPs hosted by the same simulation kernel process.

Actually, advanced kernels for optimistic simulation systems typically maintain different event lists, one for each hosted LP (see, e.g., [2], [3], [8]). This is because the identification of past and future events, and the move of events from, e.g., past to future upon rollback, needs to be actuated on a per LP basis for efficiency reasons. In such a scenario, the algorithm for identifying the future event with minimum timestamp according to LTF, requires adequate design/implementation in order to prevent the CPU scheduling task to become a bottleneck. A classical solution according to which the timestamps of the next events of different LPs are kept within an array requires complete array scanning upon the scheduling operation, with linear cost vs the number of LPs. On the other hand, organizing those timestamps within an ordered data structure, such as an heap, requires logarithmic cost for both scheduling and/or update upon variation of the next event timestamp for an LP.

We present the design and implementation (based on C technology) of an LTF scheduler that statistically provides constant-time operativity, with very limited actual overhead. Scheduling operations requiring non-constant time may only occur upon a significative variation of the locality of the next-events timestamps along the simulation-time axis. The statistical significance of such a situation clearly depends on the event pattern along the simulation-time axis for the specific application. However, there is an empirical evidence (see, e.g., [4], [5]) that LP event patterns are, at any time, characterized by greater density of events in the near future of the actual Global-Virtual-Time (GVT) \(^1\), and significantly reduced event density in the far future. As a consequence, the locality of LP next-events along the simulation-time axis is likely to change slowly.

Our proposal can be seen as a variation of the priority queue for event sets known as Calendar-Queue [1]. However, the calendar queue (statistically) offers constant-time access with actual limited overhead for element insertions/extractions in cases where the amount of elements is similar to (and is evenly distributed on) the amount of time-buckets within the calendar. In cases where this assumption does not hold, constant time is provided with non-minimal actual overhead. With our variation, we (statistically) keep the constant-time property by jointly ensuring minimal overhead without requiring a balanced distribution with a single element for each time bucket, thus increasing the efficiency of the scheduler vs the actual distribution of the next-event timestamps of the LPs along the simulation-time axis. We have integrated our LTF scheduler within the open source ROOT-Sim (ROme OpTimistic Simulator) package, and report the results of an experimental study demonstrating the effectiveness of our proposal, especially for large scale models.

The remainder of this paper is structured as follows. In Section II we discuss related work. In Section III we present the design/implementation of the scheduler. Experimental data are provided in Section IV.

II. RELATED WORK

The processor scheduling problem in optimistic simulation systems has been deeply investigated in literature. In particular, several works have proposed solutions aimed at bounding the negative performance effects related to

\(^1\)The GVT represents the simulation-time commitment horizon. Its period calculation allows memory recovery of obsolete information, e.g. saved LP states, which is no more relevant for synchronization purposes.
the occurrence of rollbacks within the parallel/distributed execution of the simulation model. The solutions in [9], [11], [12], [14], [16] target the reduction of the amount of rollback by assigning priorities to the LPs according to some statistical criterion aimed at reducing the likelihood of timestamp order violations within the system. The work in [13] extends such a view point by providing a scheduling framework targeting both the reduction of the amount of rollback, and the reduction of the waste of time associated with rollback operations. However, due to its simplicity, the standard solution for the scheduling problem is still represented by LTF [7]. As hinted, this algorithm has the advantage of avoiding causality violations across the LPs hosted by the same instance of the simulation kernel, which anyway contributes to keep low the amount of rollback within the simulation model execution. Compared to the above works, in this paper we are not targeted to define any new algorithm. Instead, we provide a highly efficient implementation of the LTF standard algorithm.

Our approach has also relations with the literature on priority queues, and in particular with works addressing the maintenance of timestamp ordered event sets (see [15] for a comparative analysis). Among these works, the closest one to our approach is [1], where, as hinted, a so called Calendar-Queue algorithm is presented, which statistically provides constant-time insertions of events with generic timestamps, and constant-time extraction of the event with the minimum timestamp (i.e. the highest priority event). This solutions is based on a linear data structure (an array) whose entries are associated with different time buckets, and the constant-time property is (statistically) guaranteed with low actual overhead only in cases where the distribution of the event timestamps (in combination with dynamic resize of the linear data structure) allows having a number of events similar to the number of time buckets, with even distribution of the events on the different buckets. Instead, in our proposal we use a combination of a linear data structure and a hierarchical bitmap, which allows us to provide constant-time scheduling with low actual overhead for all the cases in which the distribution of the timestamps of the next events of the LPs to be scheduled (in combination with the resize of the data structures) is such that each time bucket is associated with at most one element, and at least one element is currently registered within a significantly large bunch of buckets. This is a statistical property less strict than the one required for the efficiency of the Calendar-Queue in [1]. Also, in the Calendar-Queue, insertion of elements with timestamp above the current upper-limit time bucket registered within the linear data structure is supported via a list with linear access cost. Instead, such an overflow situation is handled in our scheduler via an overflow-table with \( O(1) \) access cost. This can be achieved since our scheduler is targeted to a problem intrinsically different from the one tackled by traditional priority queues for event sets. These priority queues target situations where the number of elements registered within the priority queue can significantly change over time, while our scheduler is targeted to cases in which the number of registered elements is fixed, or stable in time, since it corresponds, at any time, to the number of LPs hosted by underlying the instance of the simulation kernel (2).

III. THE SCHEDULER

A. Overview

Our LTF scheduler works by maintaining a tree-like data structure. Each leaf keeps information related to fixed size simulation-time buckets \( \delta_j \). This information consists of a list of entries, each one keeping the identity of an LP and the value of the timestamp of its next to be executed event. At any time, a single entry is registered within the data structure for any LP. The structure is made up by several blocks, each one containing a representation of a tree with the same fixed arity and depth. Therefore, the i-th block keeps information about next-event timestamps falling within a fixed simulation-time window \( \Delta_i \) of size \( |\Delta_i| = N \times |\delta_j| \), \( N \) being the number of leaves within the block.

The tree associated with each block holds the information about the occupancy of its leaves by means of nodes forming a hierarchical bitmap. In particular, each node within the tree holds a summary of the status of each of its children (those being either other bitmap nodes, or leaves). This is done by logically linking the i-th bit of the bitmap of each bitmap node to the status of the i-th child, thus setting the bit to 1 if the child contains any element in its direct children or in its offspring, or to 0 if the branch does not contain any elements in the associated leaves. In this way, the index of the branch that contains the element with minimum timestamp within the simulation time covered by a block can be quickly retrieved via a simple operation of iteratively finding the first (most significant) bit set along the hierarchical bitmap nodes organization starting from the root node. This operation requires constant time vs the number of elements. Also, by checking for a bitmap node to entirely be 0, which can be done via efficient instructions already present in most of the hardware architectures, allows to quickly skip large parts of empty trees during the search operation.

Each tree is pre-allocated (in terms of bitmaps and leaves) within a structured record. Also, a top level indexing array \( tli[] \) is kept with an entry for each block, providing a reference to the record associated with the tree related to that block. Also, the index of the block currently containing the element with the minimum timestamp across all the LPs is explicitly kept in an apposite variable, which we refer to as \( \text{min_index} \).

When a new element is to be inserted into the data structure, it is first hashed (using its timestamp as the hash key) to determine the time interval \( \Delta_j \) and the time bucket \( \delta_j \) within which it falls, uniquely corresponding to the i-th block and to the j-th leaf within that block. Then, the event is inserted in the list associated with that leaf, by directly accessing the appropriate tree via the top level reference array \( tli[] \). The hierarchical bitmap nodes associated with

\( ^2 \)Variations may occur over time in case of, e.g., dynamic load balancing schemes migrating the LPs from one kernel instance to another.
the corresponding branch within the tree are checked, and possibly updated, in order to provide a correct representation of the presence of such a new element within the tree-like data structure. Also, in case the corresponding leaf already keeps other elements, a simple (non-ordered) head-insertion is left unchanged. In this case, the leaf is set to the index of the array entry pointing to that block.

Figure 1 shows a schematization of the data structure for the case of (i) two blocks, each one having a single occupied (and hence valid) leaf, with a collision on the leaf with index 5 (corresponding to \( \delta_5 \)), (ii) trees organized as two level bitmaps of size 2, and (iii) \( |\Delta_j| \) set to 12; \( |\delta_j| \) set to 3.

Upon element insertion, the \texttt{min\_index} variable is updated as follows. In case an element is inserted within the tree-like data structure having timestamp falling within the interval of simulation time \( \Delta_j \) associated with the current minimum (i.e. \texttt{min\_index} is currently set to \( j \)) or in a subsequent interval, then \texttt{min\_index} is left unchanged. In case the insertion falls within an interval \( \Delta_k \) preceding the one associated with the current minimum, \texttt{min\_index} is simply set to the index \( k \) of the block corresponding to that \( \Delta_k \) interval.

Upon element extraction we have instead the below update rules. For an extraction associated with an LP schedule operation, the extracted element (according to LTF) is the one with the current minimum timestamp within the whole tree-like data structure. By using \texttt{min\_index}, the corresponding block is determined, and its hierarchical bitmap nodes are used to determine the time bucket \( \delta_j \), and hence the leaf, currently keeping the element with that minimum timestamp. In case of collision on that leaf, a linear search is performed on the corresponding non-ordered list to determine the element with the actual minimum. This operation can be performed in constant time in case collisions are statistically non-relevant. If the block indexed by \texttt{min\_index} gets empty after the element extraction, the top level referencing array is scanned for accessing the root bitmap node associated with subsequent blocks in order to identify the next non-empty block, if any. Similarly to the Calendar-Queue algorithm [1], where a scan across an array of time buckets is operated in similar situations, this operation requires constant time vs the number of elements within the priority queue (i.e. the LPs to be scheduled in our case). However, our approach is expected to exhibit reduced overhead since our scan operation checks at each step the presence of at least one event over \( N = |\Delta_i|/|\delta_j| \) time buckets, which is achieved thanks to the presence of the hierarchical bitmaps within the scheduler data structure. Then \texttt{min\_index} is set to the index of the array entry pointing to that block.

For an extraction associated with a change of the timestamp of the next event of a given LP, and hence a change of the priority of that LP (3) we have to perform two different tasks: (A) The element currently registered within the tree-like data structure for that LP needs to be removed; (B) The new element must be inserted. Task B gets performed in constant time exactly as discussed above for the treatment of insertions. As for task A, we have augmented the scheduler data structure with a so called \texttt{LP\_info} array, used as a lookup data structure, such that the i-th entry keeps information about the timestamp of the element currently registered within the scheduler for the i-th LP, if any (see again Figure 1). Hence task A is performed by retrieving this timestamp, hashing it to determine the corresponding block and leaf within which the element is currently registered, and removing it from the list associated with that leaf. This might also require updating the hierarchical bitmap nodes associated with the representation of the current state of the branch toward that leaf. As for the aforementioned schedule operation, this can be done in constant time in case each leaf keeps at most a single element (i.e. collisions are statistically non-relevant). After, the \texttt{LP\_info} array is updated to reflect the new priority for that LP.

B. Memory Layout of Scheduler Blocks

As said, our objective is to design a constant-time scheduler which also provides low actual overhead. To this aim, some choices have been made in order to improve the access performance of the scheduler data structure. In particular, the tree associated with each block is represented in memory in a very convenient way, both for its memory layout and the related memory access operations. Let \( h \) be the arity of the tree, and \( d \) its depth, the tree will have a total number of \( (h^d - 1)/(h-1) \) bitmap nodes and \( h^d \) leaves. The tree is allocated in our scheduler via a couple of contiguous arrays of the above sizes, respectively. The first array is a contiguous memory chunk storing bitmap nodes, and is organized in a classical packing way of trees onto a linear data structure by letting the root node of the tree be in the first position of the array (index=0), and by identifying the index of the j-th children of the i-th node as \( \text{index}(i,j) = i \times h + 1 + j \). By this organization, the operation of rising up along the tree towards the root is immediate.

In optimistic simulation systems, a change of the priority of an LP may also occur in case an event of that LP currently registered within the tree-like data structure needs to be cancelled due to the arrival of the corresponding anti-event in a rollback phase.
In fact, it is sufficient to make an integer division to obtain the index of the parent of a node. A similar operation can be carried out to find the position of the children relative to its parent, which is especially useful for the purpose of updating the bitmaps. We simply calculate the modulo of the index of the node minus 1, over h.

The second array, storing the pointers to the lists associated with each block, allows immediate access to the list of interest in both the cases of element insertion and extraction, as depicted in the previous section. Given the timestamp associated with the element to be inserted/extracted, the hashing mechanism for the identification of the correspondence bucket (and hence of the corresponding entry in this array) is simply implemented as a couple of divisions, first between the timestamp value and |Δi|, then between the remainder of this division and |δj|.

Bitmap nodes are actually represented in memory as words, so their length is proportional to the maximum word length achievable with the particular architecture the software is run on (it will most likely be either 32 or 64 bits). A trivial way of finding the first set bit within a bitmap node, which is exactly the way followed by the ffs library function, would be to bitwise rotate the word itself by one bit at a time, until a bit in a particular position is found to be set. However, this solution relies on an approach requiring to iterate, in the worst case, over all the bits within the bitmap node. Therefore, we opted for an optimized different solution which takes advantage of the floating-point capabilities offered by modern architectures. Specifically, in our implementation, the bitmaps are actually stored in memory with the most-significant-bit referring to child #0, and the least-significant-bit referring to the highest-level child. Also, via assembly blocks nested within the C code, we exploited the fyl2x instruction offered by the x87 instruction set, which calculates the base 2 log of the argument multiplied by a scalar. By setting the scalar value to 1, a simple base 2 log is calculated, which is the position of the most significant bit set. This value can then be used to traverse the tree by directly accessing the child with the requested index.

C. Handling Overflows

At any time, the tree-like data structure covers an interval of simulation time of size ΔST = |Δi| × K, where K is the current size of the top level indexing array tli[]. Given the finite size of this data structure, it is possible that the next-event timestamp, to be registered by the scheduler for some LP falls beyond the covered interval. This overflow condition is managed in our implementation by extending the LP_info[] lookup array. In particular, each entry of this array has been augmented with a flag indicating whether some valid element for this LP is currently registered within the overflow region (therefore not being present in the tree-like data structure but only within the lookup table). The scheduler also keeps a counter indicating the number of LPs for which the corresponding elements are registered within the normal (non-overflowed) interval. If this number goes to zero, then the extraction of the element with the minimum timestamp according to the LTF rule boils down to a linear-cost search operation of such a minimum value within the overflow information records kept by the lookup table. This is the only case in which non-constant time is experienced. However, as hinted, there is typically a strong locality of not yet executed events close to the current GVT [4], [5]. Therefore, by reusing the entries of the tli[], and the corresponding trees according to a circular buffer policy when the GVT is computed and memory recovery is performed, the likelihood of actual scheduling performed by linear scan of the lookup table should have non-relevant statistical significance in general settings. To cope with such an issue, our scheduler offers an appropriate API that can be invoked by the simulation kernel upon GVT calculation. Upon invocation, every block associated with an interval of simulation time Δi whose upper extreme is lower than GVT is recovered according to the circular buffer policy since there is a guarantee that no element will never eventually have timestamp falling within that interval (where simulation execution is already committed).

D. Run-Time Optimization

The run-time optimization of the scheduler entails: (1) Dynamic resize of the tli[] array; (2) Dynamic variation of the scheduler granularity, in terms of size |δj| of the used time buckets. As we pointed out, the scheduler exhibits constant time when scheduling operations refer to elements located within the non-overflowed simulation-time region, covered by the blocks registered within tli[]. Therefore, depending on the application-specific event pattern, and on the density of the events along the simulation-time axis, we need to dynamically alter the size of the scheduler data structures in order to maintain the statistical property that a compile-time defined fraction of scheduling operations must refer to the non-overflowed region. In our dynamic resize mechanism, if the actual fraction goes down the compile-time defined value, the tli[] array is doubled in size.

The second optimization is related to the size of the time-span |δj| covered by each bucket, and is aimed at reducing such a span in case collisions were statistically significant. Actually, the time span |δj| is proportional to |Δi| by a factor of h^d defined at compile time. This in turn directly influences the span covered by the tli[] array as a whole. In order not to lose the benefits obtained by resizing the tli[] array as a result of the above optimization, whenever the bucket span is reduced, the size of the array is increased by the same amount, thus leaving the total simulation-time span covered by tli[] unchanged. For this optimization, we keep an accumulator, which is increased by the value of the number of elements examined each time a scan is performed due to collisions upon insertion/extraction operations, and a counter, which is simply increased by 1 for each scan. Hence the ratio between the two values expresses the average scan length, and the aim of the optimization is to provide value of at most one for such an average scan (i.e. constant-time scan of the single element of the list associated with each non-empty bucket). Assuming that the events are more or less uniformly distributed within each bucket, by increasing
the precision of the structure, thus narrowing the interval covered by each block, there is a higher probability that the size of each bucket will be likely to accommodate at most one event. For this, the precision can be increased by scaling down the span covered by each block $|\Delta t|$ by a factor that is the average scan length, and increasing the number of items in the $\Sigma_i$ array by the same amount in order to preserve its total span, in terms of covered simulation-time interval.

We have introduced within the scheduler a proper API to perform the two aforementioned dynamic optimizations on demand. Calculation of a new GVT is an ideal trigger for invoking this API since, as explained above, housekeeping operations for supporting the circular buffer policy for the $\Sigma_i$ array are already required upon adoption of a new GVT value by the kernel.

IV. EXPERIMENTAL DATA

To assess the effectiveness of our scheduler design, we have integrated it within ROOT-Sim (ROme OpTimistic Simulator). This is an open source, advanced, general purpose platform developed using C technology, which is based on a simulation kernel layer that ultimately relies on MPI for data exchange across different kernel instances. The platform transparently supports all the mechanisms associated with parallelization (e.g., mapping of LP on different kernel instances) and optimistic processing. In Figure 2, we show a schematization of the architecture of ROOT-Sim. The simulation kernel interacts with the overlying application-level software via an interface based on an application level event-handler callback, implementing the event execution logic, and an event-schedule service offered by the simulation kernel, which can be used to inject new events with in the system, destined to whichever LP. The simulation kernel exploits the event-handler callback every time the scheduler determines, according to the LTF policy, that any new event needs to be processed by some locally hosted LP.

Recoverability of the LP states, to enable rollback based synchronization, is supported via an advanced state management subsystem named Di-DyMeLoR [17], [10] offering both full and incremental log/restore capabilities. The latter are based on a light software instrumentation approach, allowing low-cost tracking of every memory write instructions (and the corresponding addresses) performed by an LP. This subsystem can be seen as wrapper of allocation/deallocation ANSI-C standard services, interposed at linking time between the application-level software and the standard malloc library. This approach allows the application programmer to use dynamic allocated memory within the simulation software in a transparent way to lower level memory management tasks, such as log/restore operations, supported within the ROOT-Sim kernel.

The original CPU scheduler within ROOT-Sim relied on an $O(n)$ approach, where a scan operation of an array of pointers to the next event of the LPs within the respective queues determines the LP to be scheduled according to LTF. This scheduler version will be used as a baseline for the evaluation of the innovative constant-time version.

The test-bed application software is a parameterizable cellular system simulator, explicitly modeling fading and channel interference phenomena [6]. Each LP models a single cell, by tracking, via dynamically allocated data structures, channel allocation and power management information for ongoing calls. Upon the start of a call destined to a mobile device currently hosted by the cell, the LP allocates a new call-setup record via a couple of dynamically allocated data structures, and links it to a list of already active records. Each record gets released when the corresponding call ends or is handed-off towards a different cell. In the latter case, a similar call-setup procedure is executed at the destination cell. Upon call-setup, power regulation is performed, which involves scanning the aforementioned list of records for computing the minimum transmission power allowing the current call setup to achieve the threshold-level SIR value, according to GSM technology. Data structures keeping track of fading coefficients are also updated while scanning the list. We have simulated micro-cells, each one managing up to 200 wireless channels, using classical settings such as exponential distribution of the call inter-arrival time, and average call duration of 2 minutes. Also, the call inter-arrival time to each cell has been set in a way to provide normal (far from saturation) operativity of the cellular system, with channel utilization factor on the order of 10%. The choice of micro-cells (each one managing a relatively limited amount of channels) plus normal operativity has been done in order to provide a simulation model configuration with relatively fine event granularity. Medium/large granularity applications would tend to mask the relative cost of housekeeping operations (such as CPU scheduling) thus not allowing a significative study of related optimizations. We have varied the size of the simulation model between 128 and 2048 LPs. At least 1000 completed calls per cell have been simulated in each configuration.

The hardware platform used in this experimental study is a QuadCore machine, equipped with an Intel Core 2 Quad Q6600 (64bit execution support, 2.4GHz, 4MB L2 Cache per couple of cores, 32KB L1 Cache per core, 1GHz Front Side...
Bus speed) and 4GB of RAM memory. As for the software environment, the running Operating System is GNU/Linux (kernel 2.6.22-31 64bit, distribution OpenSUSE 9.2), the used gcc version is 4.2.1, the used binutils version (ld and gas) is 2.17.50 and the used MPI version is OpenMPI 1.2.4.

Four instances of the ROOT-Sim kernel have been run on this platform (one per CPU-core), with even distribution of the LPs forming the simulation model onto each kernel instance. Regarding ROOT-Sim run-time parameters, the GVT period (namely, the interval for memory recovery of obsolete logs) has been set to 1 sec. With this value, RAM usage never exceeds 60/70%, thus avoiding swapping phenomena that would alter the reliability of the reported measures. Also, for each simulation model size, the state log frequency (namely, the interval for memory recovery of obsolete logs) has been set to 1 sec. With this value, RAM usage never exceeds 60/70%, thus avoiding swapping phenomena that would alter the reliability of the reported measures.

For each simulation model size, the state log frequency (namely, the interval for memory recovery of obsolete logs) has been set to 1 sec. With this value, RAM usage never exceeds 60/70%, thus avoiding swapping phenomena that would alter the reliability of the reported measures. For each simulation model size, the state log frequency (namely, the interval for memory recovery of obsolete logs) has been set to 1 sec. With this value, RAM usage never exceeds 60/70%, thus avoiding swapping phenomena that would alter the reliability of the reported measures.

V. Summary

In this paper we have addressed the CPU scheduling problem in optimistic simulation systems, by providing the design/implementation of a constant-time LFT scheduler ensuring low actual overhead. It is based on proper data structures that dynamically adapt themselves to the properties of the events distribution. We have also shown the effectiveness of our proposal via an experimental study on a mobile simulation application.