Implications of non-constant clock drifts for the timestamps of concurrent events

Daniel Becker [#1,+], Rolf Rabenseifner [*2], Felix Wolf [#3,+]

[#] *Jülich Supercomputing Centre, Forschungszentrum Jülich*
52425 Jülich, Germany
{[1]d.becker, [3]f.wolf}@fz-juelich.de

[*] *High-Performance Computing-Center, University of Stuttgart*
70550 Stuttgart, Germany
[2] rabenseifner@hlrs.de

[+] *Department of Computer Science, RWTH Aachen University*
52056 Aachen, Germany

***Abstract*—To support the development of efficient parallel codes on cluster systems, event tracing is a widely used technique with a broad spectrum of applications ranging from performance analysis, performance prediction and modeling to debugging. Usually, events are recorded along with the time of their occurrence to measure the temporal distance between them and/or to establish a total event ordering. Obviously, measuring the time between concurrent events requires a global clock, which often, however, is not available on clusters. Assuming that potentially different drifts of local clocks remain constant over time, linear offset interpolation can be applied postmortem to map local onto global timestamps. In this study, we investigate the robustness of the above assumption using different timers and show that the error of timestamps derived in this way can easily lead to a misrepresentation of the logical event order imposed by the semantics of the underlying communication substrate. We conclude that linear offset interpolation alone may be insufficient for many applications of event tracing and discuss further options.**

I. Introduction

Driven by the availability of inexpensive commodity components produced in large quantities, clusters now represent the majority of parallel computing systems, exhibiting a vast diversity in terms of architecture, interconnect technology, and software environment. Exacerbated by the advent of multicore processors, parallel-application development on clusters faces the challenge of efficiently utilizing an increasingly complex hierarchy of latencies and bandwidths between cores on single chips, within SMP nodes, and across the network. This creates a demand for powerful software tools needed to increase the productivity of application development and to yield satisfactory runtime performance. However, to cope with cross-cluster diversity, tool developers can make only little assumptions with respect to the availability of non-standard features.

One technique widely used by cluster tools is event tracing with a broad spectrum of applications ranging from performance analysis [1], performance prediction [2] and modeling [3] to debugging [4]. In particular, event traces are helpful in understanding the performance behavior of message-passing applications, since they allow the in-depth analysis of communication and synchronization patterns. For instance, the Scalasca toolset scans event traces of parallel applications for wait states that occur when processes fail to reach synchronization points in a timely manner, for example, as a result of unevenly distributed workloads [5].

Usually, events are recorded along with the time of their occurrence to measure the temporal distance between them and/or to establish a total event ordering. Typical events being recorded include sending or receiving messages, entering or leaving functions, and events related to collective communication or synchronization. Most commonly, event traces are used to analyze MPI or hybrid MPI/OpenMP codes [6]. Obviously, measuring the time between concurrent events necessitates either a global clock or well-synchronized processor-local clocks. While some custom-built clusters such as IBM Blue Gene offer relatively accurate global clocks, most commodity clusters provide only processor-local clocks that are either entirely non-synchronized or synchronized only within disjoint partitions (e.g., SMP node). Moreover, external software synchronization via NTP [7] is usually not accurate enough for the purpose of event tracing. Assuming that potentially different drifts of local clocks remain constant over time, linear offset interpolation can be applied to map local onto global timestamps. In this study, we investigate the robustness of the above assumption across a range of timer technologies available on different platforms and show that the error of timestamps derived in this way can easily compromise the consistency of the logical event order imposed by message semantics. We conclude that linear offset interpolation alone may be insufficient for many applications of event tracing and discuss further options.

The outline of this article is as follows: After introducing the basics of processor clocks in Section II, we define the accuracy requirements of event tracing in Section III. In Section IV, we evaluate the effectiveness of linear offset interpolation using a range of timer technologies on different cluster systems and point out limitations. Then, in Section V, we discuss further options with emphasis on the retroactive correction

978-1-4244-2640-9/08/$25.00 © 2008 IEEE

of timestamps based on logical clocks as a means to remove residual inaccuracies that cannot be addressed by interpolation. Finally, we summarize our results in Section VI.

II. Processor Clocks

Processor clocks are used to obtain event timestamps and can be characterized in terms of their relative offset and drift. Figure 1 shows two clocks with both an initial offset and different but constant drifts. However, the rate at which the offset changes over time (i.e., clock drift) is usually time dependent. Here, we review the most common clock types, explain how they can be accessed, and discuss their accuracy.

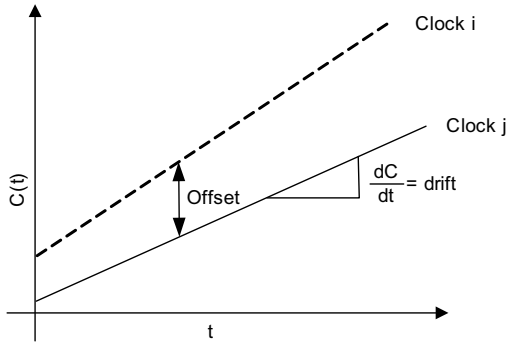


Fig. 1. Two clocks with both an initial offset and different but constant drifts.

Different types of clocks are used to measure and maintain the processor time. Clocks based on cycle counters use the processor clock signal to increment an internal counter on each tick. The step size, which depends on the clock rate, may change over time, as state-of-the-art power management may dynamically slow down or accelerate the signal. As a consequence, remote cycle counters are very hard to synchronize and therefore only useful to compare events happening on the same CPU chip. In contrast, hardware clocks, which are often called timestamp counters, use specialized hardware counters. Based on separate oscillators, their values are incremented on each tick of the oscillator, and thus, their step size does not depend on a potentially unstable clock rate. Although a single hardware clock can provide accurate relative timings, synchronization among multiple remote hardware clocks is usually not provided. As a further alternative, software clocks are realized in the form of user or library functions. Often, those clocks are implemented entirely in software without any underlying hardware support. Software-based synchronization among different software clocks (e.g., via NTP) may guarantee synchronized time values to a certain degree. Finally, system clocks are specializations of software clocks and managed by the operating system (e.g., `gettimeofday()`). Based on either cycle counters, hardware clocks, or software clocks, system clocks maintain the system-local time, which can be queried by calling a function.

As examples of hardware clocks, we consider IBM's real-time clock (RTC), IBM's time base register (TB), and Intel's timestamp counter register (TSC). All these clocks are 64-bit special-purpose registers. RTC counts seconds and nanoseconds, while TB and TSC return the number of ticks counted since processor reset. In contrast, `MPI_Wtime()` must be classified as a software clock that can be used to transparently query clock values on cluster systems. Open MPI [8], a widely used open-source MPI library, chooses among a rich set of implementations for `MPI_Wtime()` at configuration time. The default, however, is `gettimeofday()`. Support for emerging network timers that can be globally accessed is in progress. `gettimeofday()` often relies on network-based synchronization via NTP [7]. The general idea behind NTP is to synchronize distributed clocks before reading them. Distributed clocks query the global time from reference clocks, which are often organized in a hierarchy of servers. For this purpose, NTP uses widely accessible and already synchronized primary time servers. In addition, secondary time servers and clients can query time information via both private networks and the Internet. To reduce network traffic, the time servers are accessed only in regular intervals to adjust the local clock. Jumps are avoided by changing the drift while leaving the actual time unmodified. Unfortunately, varying network latencies limit the accuracy of NTP to about one millisecond compared to a few microseconds required to accurately trace MPI applications running on clusters equipped with modern interconnect technology.

Access to processor clocks is provided either locally or globally. Global accessibility implies that each processor has access to the same clock over an interconnection network within either the entire machine or only within a single partition (e.g., SMP-node or multicore-chip). Because every access either takes exactly the same amount of time regardless of the origin of the request or the exact amount of time is always known and can be used for local correction, global accessibility usually guarantees high accuracy. Even though each access introduces a certain and usually not negligible overhead, no further synchronization is required, which can even be counted against the initial overhead. For instance, the IBM Blue Gene/P system [9] offers a hardware clock that is globally accessible across the entire machine. In comparison, local accessibility means that each processor has only access to its own local clock. Of course, querying local clocks incurs less overhead because no data transfer over interconnection networks is required. On the other hand, the synchronization of remote clocks may create new overhead. Note that, in general, it cannot be assumed that processor-local clocks within the same SMP node are perfectly synchronized, as individual chips may provide their own timestamp counters.

Additionally, we distinguish between non-transparent and transparent access. Non-transparent access means a clock is queried directly, with all necessary calculations to yield the final time value left to the user. These calculations may include the multiplication with a scaling factor or the mapping onto a predefined start time. During a transparent clock access, in comparison, the user queries appropriate software functions that already incorporate these functionalities.

III. Requirements of Event Tracing

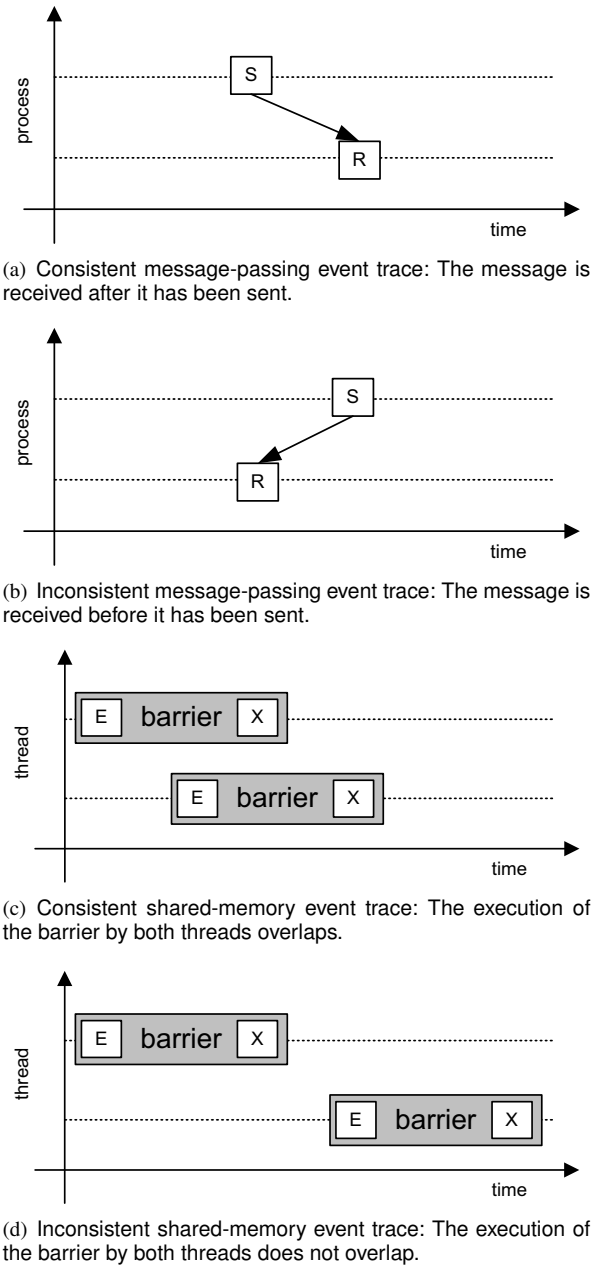
In this section, we formulate requirements distributed clocks must satisfy to allow the generation of event traces that are suitable for analyzing parallel applications. We start with explaining the basic scenario of event trace generation, present a straightforward method of compensating for missing synchronization among the clocks being used to assign the timestamps, and discuss limitations resulting from different sources of inaccuracy.

Before parallel applications can be traced, they usually must be linked to a tracing library. Instrumentation occurs either by inserting extra code into the program itself and/or by linking to wrapper libraries. MPI calls are commonly traced using PMPI interposition wrappers [10]. Whenever the running application generates an event, the tracing library takes the current time and writes an event record to a memory buffer. After program termination or if necessary already earlier while the program is still running, the buffer contents is flushed to disk. Events typically recorded by MPI and/or OpenMP applications include sending and receiving point-to-point messages, entering and leaving code regions, or events related to collective communication or synchronization. To minimize the intrusion overhead associated with timestamp creation, the clock is read locally, as querying a remote clock across the network would consume too much time, not to mention the inaccuracy that may be caused by an uncertain clock-reading latency. As a consequence, the timestamps taken on most cluster nodes stem from insufficiently synchronized local clocks.

a) Accuracy requirements: The accuracy of most trace analyses depends on the comparability of timestamps taken on different processors. Inaccurate timestamps can not only cause a given interval to appear shorter or longer than it actually was, but also change the logical event order, which requires that a message can only be received after it has been sent. This is also referred to as the *clock condition*. The clock condition, which is given in Equation 1, requires that a receive event occurs at the earliest l_{min} after the matching send event, with l_{min} being the minimum message latency.

$$t_{recv} \geq t_{send} + l_{min} \tag{1}$$

To avoid violations of this condition, the error of timestamps should ideally be smaller than one half of the message latency. Analogous requirements can be derived for alternative communication mechanisms such as collective communication or synchronization by mapping their semantics onto point-to-point communication. Inaccurate timestamps may lead to false conclusions during trace analysis, for example, when the impact of certain behaviors is quantified, or - even more strikingly - may confuse the user of trace visualization tools such as VAMPIR [11] by causing arrows representing messages to point backward in time-line views. Even worse, automatic trace-analysis tools such as KOJAK [6] that rely on the correct event order may break when encountering clock-condition violations.



(a) Consistent message-passing event trace: The message is received after it has been sent.

(b) Inconsistent message-passing event trace: The message is received before it has been sent.

(c) Consistent shared-memory event trace: The execution of the barrier by both threads overlaps.

(d) Inconsistent shared-memory event trace: The execution of the barrier by both threads does not overlap.

Fig. 2. Implications of inaccurate timestamps for message-passing (MPI) and shared-memory (OpenMP) event semantics.

Figure 2 exemplifies the potential implications of inaccurate timestamps for the semantics of message-passing and shared-memory events. The correct message-passing event order shown in Figure 2(a) is violated in Figure 2(b). The two diagrams show the time lines of two processes exchanging a message via a send (S) and a receive (R) event. In the second picture, the measurement suggests that the message has been received before it has been sent, which, of course, is impossible. The next two diagrams present a similar case that may occur in OpenMP programs when writing traces according to the POMP event model [12]. Shown is the execution of an OpenMP barrier by two threads involving two different

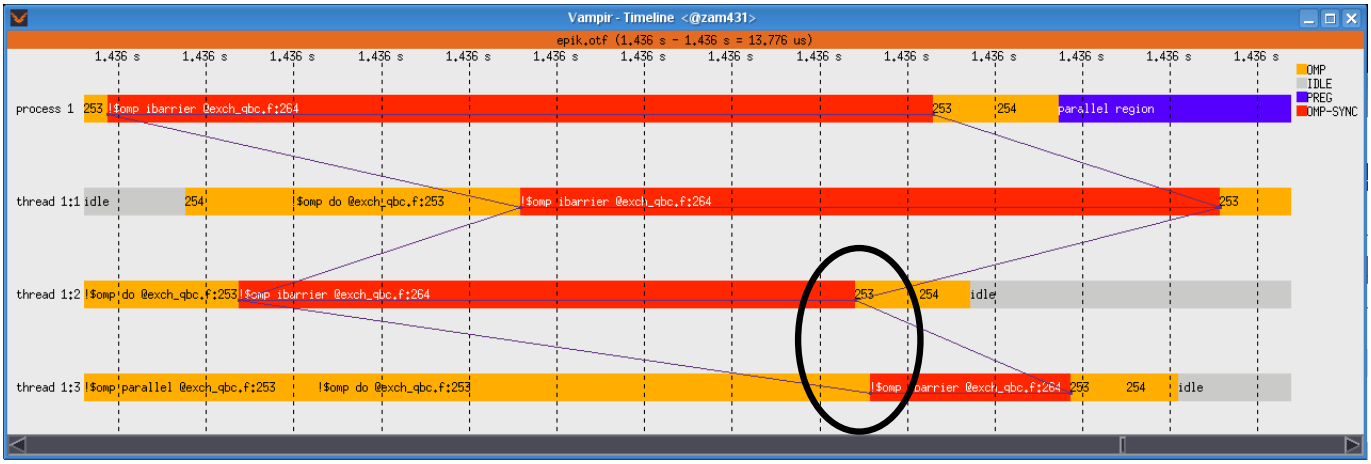


Fig. 3. A violation of OpenMP barrier semantics observed on an Itanium SMP node.

event types: entering (E) and exiting the barrier (X). Whereas in Figure 2(c) the event order is consistent, in Figure 2(d) one thread leaves the barrier before the other one has entered it, constituting a clear violation of barrier semantics. As the example in Figure 3 demonstrates, such violations can indeed occur in practice. The figure shows the time-line visualization of an event trace taken from an OpenMP benchmark program executed with 4 threads on an Intel Itanium node with 4 chips á 4 cores. As can be seen in the encircled area, thread 1:2 seems to have left the barrier (medium dark or red bars) before thread 1:3 had a chance to enter it. Besides violations of barrier semantics, OpenMP applications may also suffer from misrepresentations of other happened-before relationships specified in the POMP event model, such as the rule that all events belonging a parallel region must be temporally enclosed by fork and join events.

b) Linear offset interpolation: The most significant deviations between non-synchronized clocks result from differences in offset and drift. In a simple model assuming different but constant drifts, the local (i.e., worker) time can be mapped onto the global time of an arbitrarily chosen master postmortem via linear offset interpolation based on prior offset measurements. Offsets between master and workers can be determined using Cristian's probabilistic remote clock reading technique [13]. The master process sends a request to a remote worker process at time t_1, the worker responds by sending back its current local time t_0, which is received by the master at time t_2. Assuming that the two message delays have equal length, the offset can be calculated according to Equation 2.

$$o = t_1 + \frac{t_2 - t_1}{2} - t_0 \tag{2}$$

Since, contrary to our assumption, real message communication is prone to irregular delays, the process must be repeated several times to minimize the delay. Offset values among participating clocks are measured either at program initialization [14], [15] or at initialization and finalization [16], and are subsequently used as parameters of a linear correction function. Not to perturb the program, offset measurements in between are usually avoided, although a recent approach proposes periodic offset measurements during global synchronization operations while limiting the effort required in each step by resorting to indirect measurements across several hops [17]. Assuming two measurements, one at he beginning and one at the end, every remote worker has eventually two pairs (w_1, o_1) and (w_2, o_2) that contain its local worker time together with the offset to the matching master time when the two measurements were taken. The master time m can now be calculated from the worker time t using Equation 3.

$$m(t) = t + \frac{o_2 - o_1}{w_2 - w_1} \times (t - w_1) + o_1 \tag{3}$$

c) Sources of inaccuracy: While the above scheme might prove satisfactory for short runs, measurement errors and time-dependent drifts may create inaccuracies and clock-condition violations during longer runs. Additionally, repeated drift adjustments caused by NTP may impede linear offset interpolation, as they deliberately introduce non-constant drifts. In general, inaccuracies in non-synchronized timestamps often show up as a result of either unstable clock drifts or measurement errors. Apparently, varying temperature and flexible power management provided by modern microprocessors may alter oscillation frequencies. As a result, clocks may gradually diverge as the time progresses. Moreover, insufficient timer resolution may introduce measurement errors, an effect exacerbated by OS jitter. Jitter interference is primarily caused by scheduling daemon processes or handling asynchronous events such as interrupts on the side of the operating system. Although all of the above influences may be predictable to some degree, modeling them correctly will require intimate knowledge of the underlying hardware and software infrastructure, which is usually not available to developers of generic cluster tools. From our perspective, this behavior can therefore be classified as non-deterministic. Finally, network topology and load may adversely affect the predictability of message latencies, an important prerequisite for network-based synchronization. As messages travel through various stages of the network, the processing time in each stage may vary depending on the current network load. Since messages exchanged between the same pair of locations may take differently long each time,

error correction based on assumptions about the message latency remains challenging.

IV. Clock Evaluation

The primary goal of our study is to evaluate the effectiveness of linear offset interpolation as an instrument for the postmortem synchronizations of timestamps in event traces of parallel applications. For this purpose, we conducted measurements using different timers on a selection of typical cluster architectures:

Xeon cluster: Located at the Center for Computing and Communication of RWTH Aachen University, the cluster consists of 62 computing nodes, each with 2 quad-core Intel Xeon processors running at 3.0 GHz. The computing nodes communicate primarily through an InfiniBand network.

PowerPC cluster: Located at the Barcelona Supercomputing Center, the cluster (aka MareNostrum) consists of 2560 JS21 blade computing nodes, each with 2 dual-core IBM 64-bit PowerPC 970MP processors running at 2.3 GHz. The computing nodes communicate primarily through a Myrinet network with Myrinet adapters integrated on each server blade.

Opteron cluster: Located at the National Center for Computational Sciences at Oak Ridge National Laboratory, the cluster (aka Jaguarcnl) consists of 3744 XT3 compute nodes, each with one dual-core AMD Opteron processor running at 2.6 GHz. Each node is connected to a distinct Cray SeaStar router through HyperTransport with all the SeaStars arranged in a 3-D-torus network topology.

In a first step, we measured residual clock deviations after applying (i) offset alignment only at program initialization so that all clocks started from zero and (ii) linear offset interpolation based on offset measurements both at program initialization and finalization, as described in the previous section. To reflect varying application runtimes, we performed short (300 s), medium (1800 s), and long (3600 s) measurement runs. All processes were located on different SMP nodes. In a next step, we measured the actual frequency of clock-condition violations in event traces of two realistic MPI applications. The first application we tested was the Parallel Ocean Program (POP), which is shipped with the SPEC MPI2007 1.0 benchmark suite [18]. The second application we tested was the MPI version of the ASC SMG2000 benchmark, a parallel semi-coarsening multigrid solver that uses a complex communication pattern and performs a large number of non-nearest-neighbor point-to-point communication operations. Finally, taking the hierarchical structure of modern multicore-based cluster architectures into account, we tried to estimate the chances of clock-condition violations in MPI or OpenMP codes when processes are placed on the same SMP node but on different chips or on the same chip, as shown in Table I for the Xeon cluster.

As described in Section III, the error of timestamps should ideally be smaller than one half of the message latency to generate traces suitable for parallel-program analysis. Since messages latencies between cores on a single chip, between chips on a single SMP node, and between different SMP nodes usually differ, we measured the message latency for all these cases. For the first case, we measured also the collective all-reduce latency. As Table II shows for the Xeon cluster, the latency exhibits significant variations depending on the relative location of processes, a fact that needs to be taken into account when designing postmortem synchronization algorithms.

TABLE I
Xeon cluster: Process pinning for measurements among SMP nodes, chips, and cores.

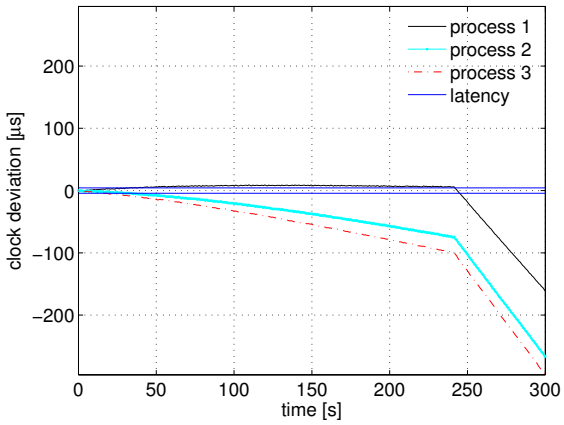
	Process pinning
Inter node	4 nodes 1 process per node
Inter chip	1 node 2 chips per node 1 process per chip
Inter core	1 node 1 chip per node 4 processes per chip

TABLE II
Xeon cluster: Measured message and collective latencies for different measurement setups.

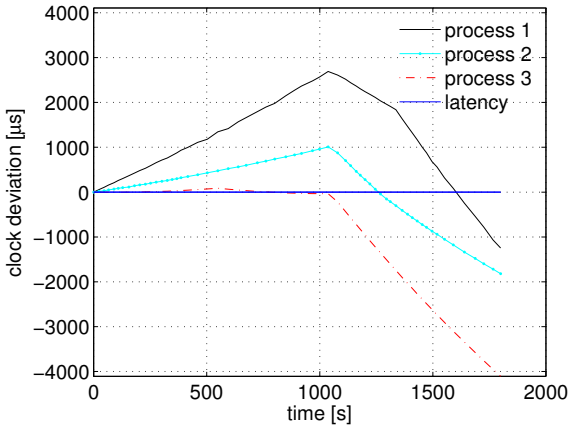
	mean [μs]	**std. dev. [μs]**
Inter node message latency	4.29	9.80E-04
Inter chip message latency	0.86	4.77E-05
Inter core message latency	0.47	6.94E-06
Inter node collective latency	12.86	1.68E-02

To evaluate the effectiveness of linear offset interpolation, we first measured clock deviations of `MPI_Wtime()`, `gettimeofday()`, and the Intel timestamp counter on the Xeon cluster during runs of increasing duration after an initial alignment of offsets. The results are shown in Figure 4. Obviously, `MPI_Wtime()` (Figure 4(a)) produces severe clock deviations of more than 200 μs already after a relatively short period. Interestingly, the deviation seems to grow roughly at a constant rate up to a turning point at which the slope abruptly changes. After this point, the affected processes continue striding away linearly but at a much higher rate. `gettimeofday()` (Figure 4(b)) exhibits a very similar drift pattern, again showing phases of roughly constant drift interrupted by sudden drift adjustments – albeit a little bit more curvy at least in one instance. The changes are presumably caused by the underlying NTP synchronization, which periodically corrects the drift to prevent the clocks from diverging too far. This, of course, is detrimental to linear offset interpolation, as it deliberately introduces non-constant drifts. In sharp contrast to the previous two measurements, however, the Intel timestamp counter (Figure 4(c)) appears to maintain an approximately constant clock drift even across a very long period of time. Obviously, hardware clocks seem much more appropriate when it comes to taking the timestamps of concurrent events.

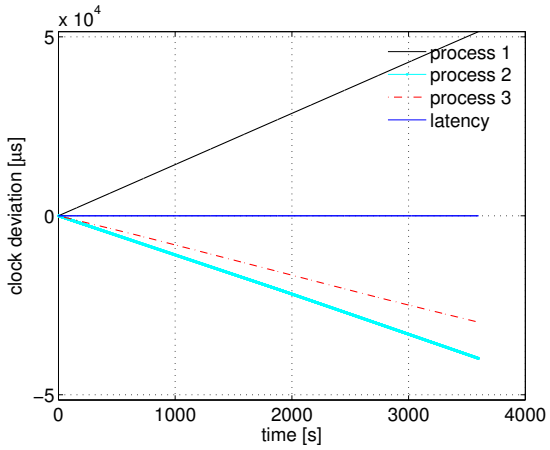
Having reached this conclusion, subsequent experiments



(a) MPI_Wtime(): Clock deviations during a short run.

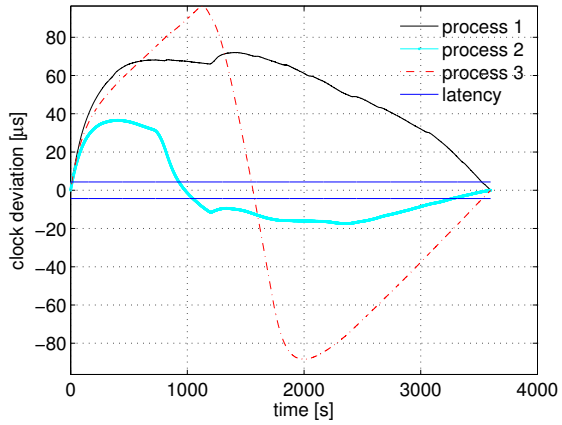


(b) gettimeofday(): Clock deviations during a medium run.

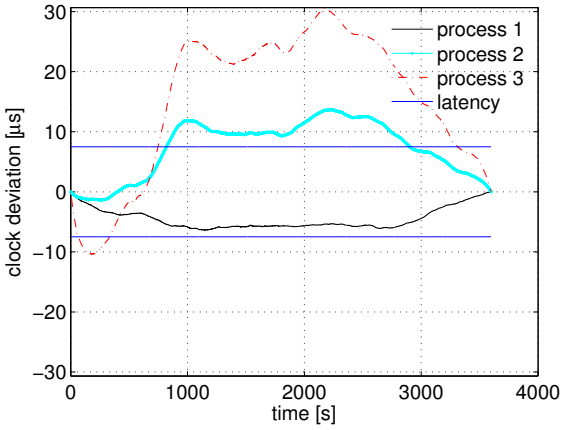


(c) Intel's timestamp counter: Clock deviations during a long run.

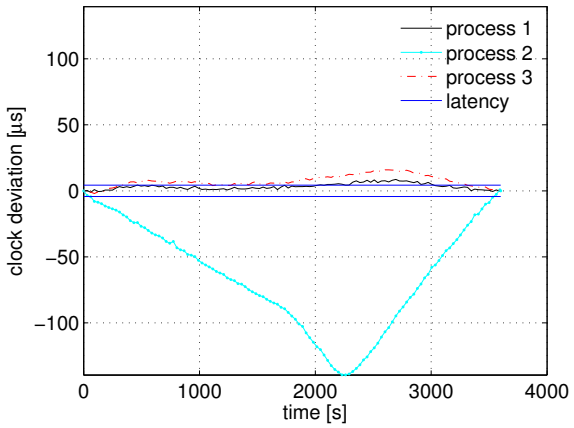
Fig. 4. Xeon cluster: Measured clock deviations of different timers during short, medium, and long measurement runs after an initial offset alignment.



(a) Xeon cluster: Clock deviations using Intel's timestamp counter.



(b) PowerPC cluster: Clock deviations using IBM's time base register.



(c) Opteron cluster: Clock deviations using gettimeofday().

Fig. 5. Measured clock deviations of two different hardware clocks and gettimeofday() during long runs after linear offset interpolation.

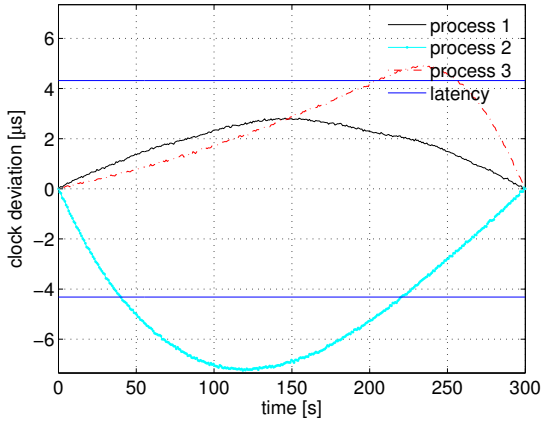


Fig. 6. Measured clock deviations after linear interpolation during a short run on the Xeon cluster using the Intel timestamp counter. The deviations slightly exceed the latency.

therefore focused on the evaluation of hardware clocks on different cluster platforms. We conducted tests on the three clusters with Intel's timestamp counter, with IBM's time base register, and for comparison with `gettimeofday()` always using a duration of 3600 s. Figure 5 shows residual clock deviations after performing linear offset interpolation with an expected convergence of offsets at the end of the run. As can easily be seen, linear interpolation already accounts for the most severe differences in offset and drift, although significant deviations can still be observed, the highest occurring when using `gettimeofday()` on the Opteron system. In fact, measured deviations exceeded the message latency already after a few minutes or even earlier, rendering linear interpolation alone insufficient to guarantee the absence of clock-condition violations during longer runs. Since shorter runs also use a shorter interpolation interval, linear interpolation may still be adequate in those cases, although our results on the Xeon cluster suggest that even then violations may occur (Figure 6).

To quantify the extent of clock-condition violations in traces of real applications, we performed experiments with POP and SMG2000 on the Xeon cluster, each time using 32 processes. Emulating a realistic scenario, we refrained from using a specific process pinning. Instead, we kept the default setting and let the scheduler choose the pinning automatically. Traces were obtained using the Scalasca toolset, which performs linear offset interpolation based on offset measurements taken during `MPI_Init()` and `MPI_Finalize()`. We ran POP with the mref input data set, causing it to execute 9000 iterations in roughly 25 min. Since tracing the full run would consume a prohibitively large amount of storage space, we traced iterations 3500 to 5500. This "partial" tracing corresponds to the recommended practice of tracing only pivotal points of long-running applications that warrant a more detailed analysis. For SMG2000, a problem size of $16 \times 16 \times 8$ per process with five solver iterations was configured. We emulated a longer run of SMG2000 by inserting sleep statements immediately before and after the main computational phase so that it was carried out ten minutes after initialization and ten minutes before finalization. This corresponds to a scenario similar to POP, in which only distinct intervals of a longer run are traced with tracing being switched off in between. For our purposes, the artificial chronological distance to the offset measurements on either end of the run adjusted the interpolation interval to roughly twenty minutes execution time. However, with many realistic codes running for hours, the execution times of both POP and SMG2000 in our experiments can still be regarded as an optimistic assumption.

Figure 7 shows the frequency of clock condition violations for both applications on the Xeon cluster. The numbers represent averages across three measurements for each application because he number of violations varied between runs. The front row shows the percentage of messages with the order of send and receive events being reversed, while the back row shows the fraction of message transfer event in relation to the total number of events in the trace. We also counted logical messages that can be derived by mapping collective communication onto point-to-point semantics. The results underline our hypothesis that linear interpolation alone is insufficient to produce traces free of clock condition violations and that such violations may adversely affect a significant percentage of message events.

Finally, we examined relative deviations of clocks co-located on the same SMP node of the Xeon cluster without any correction, after aligning only initial offsets, and after applying linear offset interpolation. We distinguished between processes located on different chips and processes located on the same chip. In all cases, the deviations we measured

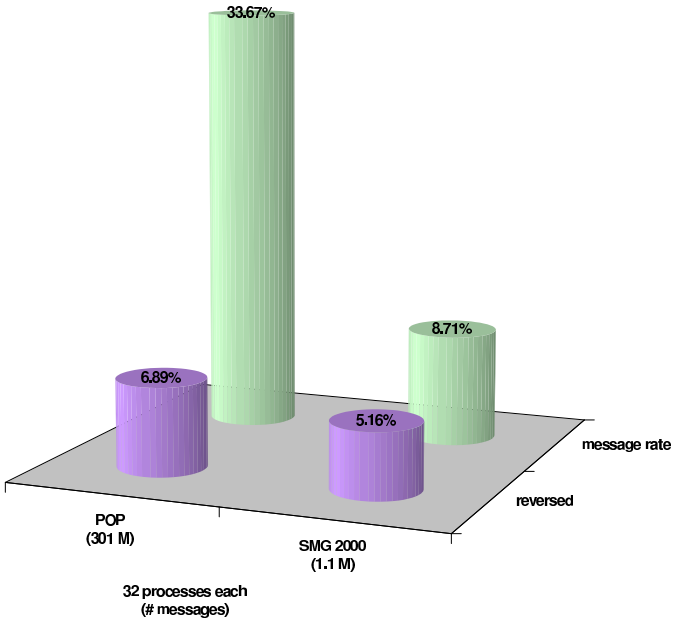


Fig. 7. Xeon cluster: Percentage of messages with the order of send and receive events being reversed and of message transfer events in relation to the total number of events for SMG2000 and POP.

essentially constitute "noise" oscillating around zero with a maximum difference of roughly 0.1 μs between any two clocks in our ensemble. One conclusion is that on this system MPI message semantics can be easily preserved without further postprocessing of timestamps.

However, as experiments on an Intel Itanium SMP node with 4 chips á 4 cores suggest, on some systems the semantics of OpenMP constructs can be easily violated: We took traces from a simple OpenMP benchmark program that executes a loop whose body contains a single parallel-for construct, which is a short cut for an OpenMP for construct enclosed by a parallel region. The tests were conducted with varying numbers of threads, ranging from 4 to 16. All events were recorded according to the POMP event model. Neither offset alignment nor linear offset interpolation was applied to the timestamps, which were taken using the Intel timestep counter. Figure 8 shows the fraction of parallel regions exhibiting clock-condition violations. Again, the numbers represent averages across three measurements for each configuration. The row in the back gives the percentage of parallel regions with violations of any kind, while the three rows in the front give the percentages of parallel regions with specific violations: at the region entry (i.e., fork event not the first event), at the region exit (i.e., join event not the last event), and during the implicit barrier. Notably, when using only four threads, more than three quarters of the regions (83 %) were affected, with violations at the region exit occurring most frequently. However, the fraction of affected regions drops sharply as the number of threads is increased, with 12 threads causing only very few violations and 16 threads none at all. OpenMP synchronization latencies rising with an increasing number of threads offer a potential explanation. Interestingly, some of the traces showed violations at the region entry but not at the exit and vice versa, which may back the assumption that a more systematic clock deviation can be held responsible. Unfortunately, the test system did not support the pinning of individual OpenMP threads to specific cores so that have been unable to distinguish between inter- and intra-chip effects. Whether offset alignment or interpolation can alleviate the errors remains to be evaluated and also depends on the question to which extent the mapping of threads onto cores remains stable during the execution of longer programs.

Summarizing the insights we have gained so far, we can state that linear offset interpolation is insufficient at least for message-passing and hybrid applications spawning more than one SMP node. As a consequence, the logical event order imposed by the semantics of the underlying communication substrate may be misrepresented. Still lacking more appropriate timer technologies such as network clocks on many cluster systems, we now have to look for alternatives. In the next section, we therfore review several approaches aiming at correcting such inconsistencies, most of them usually applied postmortem.

V. Clock Synchronization

Error estimation allows the retroactive correction of clock values in event traces after assessing synchronization errors among all distributed clock pairs. First, difference functions among clock values are calculated from the differences between clock values of receive events and clock values of send events (plus the minimum message latency). Second, a medial smoothing function can be found and used to correct local clock values because for each clock pair two difference functions exist. Regression analysis and convex hull algorithms have been proposed by Duda [19] to determine the smoothing function. Using a minimal spanning tree algorithm, Jezequel [20] has adopted Duda's algorithm for arbitrary processor topologies. In addition, Hofmann [21] has improved Duda's algorithm using a simple minimum/maximum strategy. Babaoğlu and Drummond [22], [23] have shown that clock synchronization is possible at minimal cost if the application makes a full message exchange between all processors in sufficiently short intervals. However, jitter in message latency, nonlinear relations between message latency and message length, and one-sided communication topologies limit the usefulness of error estimation approaches.

In contrast, logical synchronization uses happened-before relations among send and receive pairs to synchronize distributed clocks. Lamport has introduced a discrete logical clock [24] with each clock being represented by a monotonically increasing software counter. As local clocks are incremented after every local event and the updated values are exchanged at synchronization points, happened-before relations can be exploited to further validate and synchronize distributed clocks. If a receive event appears before its corresponding send event, that is, if a clock-condition violation occurs, the receive event is shifted forward in time according to the clock

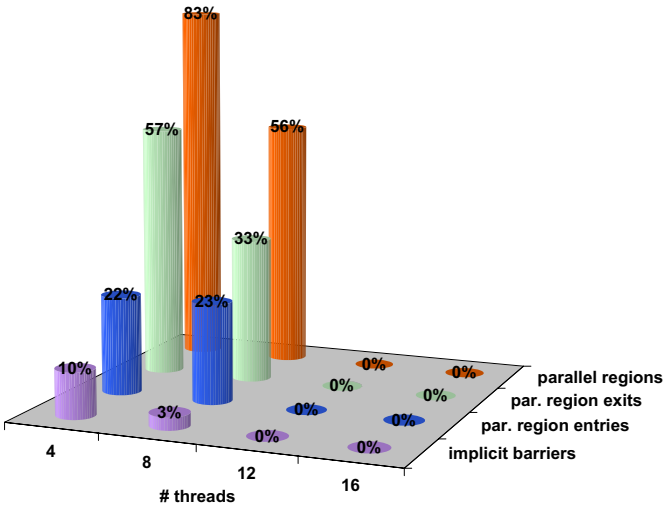


Fig. 8. Intel Itanium SMP node: Percentages of parallel regions in an OpenMP benchmark program exhibiting clock-condition violations across a range of thread counts.

value exchanged. As an enhancement of Lamport's discrete logical clock, Fidge [25], [26] and Mattern [27] have proposed a vector clock. In their scheme, each processor maintains a vector representing all processor-local clocks. While the local clock is advanced after each local event as before, the vector is updated after receiving a message using an element-wise maximum operation between the local vector and the remote vector that has been sent along with the message.

Finally, the *controlled logical clock* (CLC) algorithm [28], [29] developed by one of the authors retroactively corrects clock condition violations in event traces of message-passing applications by shifting message events in time while trying to preserve the length of intervals between local events. The algorithm restores the clock condition using happened-before relations derived from message semantics. If the clock condition is violated for a send-receive event pair, the receive event is moved forward in time. To preserve the length of intervals between local events, events following or immediately preceding the corrected event are moved forward as well. These adjustments are called forward and backward amortization, respectively. Note that the accuracy of the adjustment depends on the accuracy of the original timestamps. Therefore, the algorithm benefits from weak pre-synchronization such as the linear-offset interpolation discussed earlier. Since the original CLC algorithm takes only point-to-point messages into account, it has recently been extended to make it suitable for realistic MPI applications that perform not only point-to-point but also collective communication [30]. The basic idea behind this extension is to map collective onto point-to-point communications by considering a single collective operation as being composed of multiple point-to-point operations, taking the semantics of the different flavors of MPI collective operations into account (e.g. *1-to-N*, *N-to-1*, etc.). Additionally, the algorithm has been efficiently parallelized so that it can be applied to traces from large numbers of processes [31].

VI. Conclusion

In this study, we have evaluated different options for obtaining event timings when tracing parallel applications on cluster systems. Because the danger of perturbation complicates offset measurements in the middle of the run, simple (as opposed to piecewise) linear offset interpolation has been introduced as an established instrument used by state-of-the-art tracing tools such as Scalasca for an initial correction of timestamps and as a yardstick to assess the appropriateness of timer technologies.

Since software clocks such as `MPI_Wtime()` or `gettimeofday()` often leverage network synchronization via NTP, which can lead to sudden drift adjustments, hardware clocks such as IBM's time base register (TB) or Intel's timestamp counter register (TSC) have been identified as alternatives with at least approximately constant clock drifts. However, as a more detailed analysis revealed, even those suffer from drift deviations that may compromise the accuracy of linear offset interpolation - especially when the application runs longer than a few minutes. As a consequence, many traces of MPI applications spawning multiple SMP nodes of a cluster system will exhibit violations of the clock condition, potentially misrepresenting the logical event order imposed by message semantics and therefore harming further analyses. Evidence of frequent violations in real codes has been presented.

Moreover, inaccuracies of timestamps within single SMP nodes in combination with the low latency of shared-memory synchronization in OpenMP may lead to infringements of OpenMP semantics on some systems. As our experiments further indicate, smaller numbers of OpenMP threads tend to be more easily affected than larger numbers – potentially due to lower OpenMP synchronization latencies when using only a few threads.

Obviously, when using hardware clocks, linear offset interpolation can significantly increase the accuracy of timings when tracing across several SMP nodes, but is still insufficient when applied in isolation. A viable option for removing remaining inconsistencies is the CLC algorithm, as it can be efficiently applied to realistic message-passing traces even at larger scales. Current limitations, which still need to be addressed, include the non-observance of shared-memory clock conditions related to OpenMP constructs and the algorithm's inability to account for synchronized clocks within single SMP nodes. The latter may become important because if the timestamp of a process is modified in the course of applying the algorithm, timestamps of processes co-located on the same SMP node that are close to the modified time may need to be modified as well.

Acknowledgment

The authors thankfully acknowledge the computer resources, technical expertise and assistance provided by the Barcelona Supercomputing Center - Centro Nacional de Supercomputacion and the National Center for Computational Sciences at Oak Ridge National Laboratory (ORNL). The resources used at ORNL are supported by the Office of Science of the Department of Energy under Contract DE-ASC05-00OR22725. Finally, we would like to express our gratitude to the following individuals for their generous help: Judit Gimenez, Jesus Labarta, and Patrick Worley.

References

[1] W. Nagel, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer*, vol. 12, no. 1, pp. 69–80, 1996.

[2] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris, "DiP: A parallel program development environment," in *Proc. of the European Conference on Parallel Computing (Euro-Par)*. Lyon, France: Springer, August 1996, pp. 665–674.

[3] G. Rodriguez, R. Badia, and J. Labarta, "Generation of simple analytical models for message passing applications," in *Proc. of the European Conference on Parallel Computing (Euro-Par)*, ser. Lecture Notes in Computer Science, vol. 3149. Pisa, Italy: Springer, August - September 2004.

[4] S. Huband and C. McDonald, "A Preliminary Topological Debugger for MPI Programs," in *Proc. of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, R. Buyya, G. Mohay, and P. Roe, Eds. IEEE Computer Society, 2001, pp. 422–429.

[5] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr, "Scalable parallel trace-based performance analysis," in *Proc. 13th European PVM/MPI Conference*. Bonn, Germany: Springer, September 2006.

[6] F. Wolf and B. Mohr, "Automatic performance analysis of hybrid MPI/OpenMP applications," *Journal of Systems Architecture*, vol. 49, no. 10-11, pp. 421–439, November 2003.

[7] D. L. Mills, "Network Time Protocol (Version 3)," The Internet Engineering Task Force - Network Working Group, March 1992, rFC 1305.

[8] Open MPI, http://www.open-mpi.org/.

[9] IBM Blue Gene Team, "Overview of the IBM blue gene/p project," *IBM Journal of Reserach and Development*, vol. 52, no. 1/2, 2008.

[10] Message Passing Interface Forum, "MPI: A message passing interface standard, version 1.1, chapter 8," June 1995, http://www.mpi-forum.org/docs/mpi-11-html/node152.html.

[11] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer 63*, vol. XII, no. 1, pp. 69–80, January 1996.

[12] B. Mohr, A. Malony, S. Shende, and F. Wolf, "Design and prototype of a performance tool interface for OpenMP," *The Journal of Supercomputing*, vol. 23, no. 1, pp. 105–128, August 2002.

[13] F. Cristian, "Probabilistic clock synchronization," *Distributed Computing*, vol. 3, no. 3, pp. 146–158, 1998.

[14] T. H. Dunigan, "Hypercube clock synchronization," Oak Ridge National Laboratory, TN, Technical Report ORNL TM-11744, February 1991.

[15] ——, "Hypercube clock synchronization," ORNL TM-11744 (updated), September 1994, http://www.epm.ornl.gov/∼dunigan/clock.ps.

[16] E. Maillet and C. Tron, "On efficiently implementing global time for performance evaluation on multiprocessor systems," *Journal of Parallel and Distributed Computing*, vol. 28, pp. 84–93, 1995.

[17] J. Doleschal, A. Knüpfer, M. S. Müller, and W. E. Nagel, "Internal timer synchronization for parallel event tracing," in *Proc. 15th European PVM/MPI Conference*. Dublin, Ireland: Springer, September 2008.

[18] M. S. Müller, M. van Waveren, R. Lieberman, B. Whitney, H. Saito, K. Kalyan, J. Baron, B. Brantley, C. Parrott, T. Elken, H. Feng, and C. Ponder, "SPEC MPI2007 - An application benchmark for clusters and HPC systems," in *Proc. of the 23rd International Supercomputing Conference (ISC'07)*, Dresden, Germany, June 2007.

[19] A. Duda, G. Harrus, Y. Haddad, and G. Bernard, "Estimating global time in distributed systems," in *Proceedings of the 7th International Conference on Distributed Computing Systems, Berlin, September 21-25, 1987*. IEEE Computer Society Press, 1987, pp. 299–306.

[20] J.-M. Jézéquel, "Building a global time on parallel machines," in *Proceedings of the 3rd International Workshop on Distributed Algorithms,* LNCS 392, J.-C. Bermond and M. Raynal, Eds. Springer-Verlag, 1989, pp. 136–147.

[21] R. Hofmann, "Gemeinsame Zeitskala für lokale Ereignisspuren," in *Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen, 7. GI/ITG-Fachtagung, Aachen, 21.-23. September 1993*, B. Walke and O. Spaniol, Eds. Springer-Verlag, Berlin, 1993.

[22] O. Babaoğlu and R. Drummond, "(Almost) no cost clock synchronization," in *Proceedings of 7th International Symposium on Fault-Tolerant Computing*. IEEE Computer Society Press, July 1987, pp. 42–47.

[23] R. Drummond and O. Babaoğlu, "Low-cost clock synchronization," *Distributed Computing*, vol. 6, no. 4, pp. 193–203, July 1993.

[24] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[25] C. J. Fidge, "Timestamps in message-passing systems that preserve partial ordering," in *Proceedings of 11th Australian Computer Science Conference*, February 1988, pp. 56–66.

[26] ——, "Partial orders for parallel debugging," *ACM SIGPLAN Notices*, vol. 24, no. 1, pp. 183–194, January 1989.

[27] F. Mattern, "Virtual time and global states of distributed systems," in *Proceedings of International Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, France, October 1988*, M. Cosnard and P. Quinton, Eds. Elsevier Science Publishers B. V., Amsterdam, 1989, pp. 215–226.

[28] R. Rabenseifner, "The controlled logical clock - a global time for trace based software monitoring of parallel applications in workstation clusters," in *Proc. 5th EUROMICRO Workshop on Parallel and Distributed (PDP'97)*, London, UK, January 1997, pp. 477–484.

[29] ——, "Die geregelte logische Uhr, eine globale Uhr für die tracebasierte Überwachung paralleler Anwendungen," Ph.D. dissertation, Universität Stuttgart, March 2000.

[30] D. Becker, R. Rabenseifner, and F. Wolf, "Timestamp synchronization for event traces of large-scale message-passing applications," in *Proceedings of the 14th European PVM/MPI Conference*. Paris, France: Springer, October 2007, pp. 315–325.

[31] D. Becker, J. C. Linford, R. Rabenseifner, and F. Wolf, "Replay-based synchronization of timestamps in event traces of massively parallel applications," in *Proc. of the First International Workshop on Simulation and Modelling in Emergent Computational Systems (SMECS-2008, in conjunction with ICPP-2008)*, Portland, OR, September 2008.