ISPCS 2008 – International IEEE Symposium on Precision Clock
Synchronization for Measurement, Control and Communication
Ann Arbor, Michigan, September 22–26, 2008

Achieving Precise Coordinated Cluster Time in a Cluster Environment

Steven Froehlich, Michel Hack, Xiaoqiao Meng and Li Zhang
IBM T.J. Watson Research Center, Hawthorne, NY 10532-1569
Email: {stevefro, hack, xmeng, zhangli}@us.ibm.com

***Abstract*—A time-keeping mechanism is proposed for providing microsecond-level consistent time across a cluster of computers. The proposed mechanism is based on a new clock steering algorithm that uses piecewise linear mapping to align a local clock to an external reference clock in a smooth manner. We present two realizations of the algorithm: one is based on Pulse-Per-Second (PPS) and the other is based on low-latency timing message exchange. The time derived by the proposed mechanism is called CCT (Coordinated Cluster Time). It has a well-defined interface such that it can be used by applications with little overhead. Moreover, the interface deals completely with Leap Second issues. We implemented CCT on IBM BladeCenters and compared it to NTP. Experimental results demonstrate that the proposed mechanism achieves one microsecond precision.**

I. Introduction

Many applications running in a cluster environment demand high-precision clock synchronization at moderate cost. NTP, the *de facto* standard for clock synchronization, is a low-cost solution, yet its accuracy is mostly at millisecond level which is often not sufficient. On the other hand, IEEE 1588 gives superior performance by achieving sub-microsecond accuracy. Recently, in-depth analyses and extensions of IEEE 1588 have appeared in the literature [1]. A limitation of IEEE 1588 is that it requires additional hardware and may not be fully compatible with legacy cluster systems. Many researchers proposed time synchronization solutions which seek to strike a balance between cost and precision, e.g., Ridoux and Veitch [2] proposed to use the TSC (Time Stamp Counter) clock to ensure 10-microsecond accuracy on LANs.

In this paper we describe a time-keeping mechanism which provides a cluster-wide consistent time with a precision at one microsecond. We refer to the time derived by this mechanism as *CCT* (Coordinated Cluster Time). The core component is a new clock steering method. Clock steering occurs when a local clock wants to remain perfectly stable with respect to a reference clock. Generally, the local clock needs to estimate and compensate for the skew and offset between the two clocks by periodic resynchronization. By using the proposed clock steering method, each server in the cluster maintains a mapping of TSC cycle counter readings to the correct time *TAI* (International Atomic Time), which is the basis for deriving UTC (Coordinated Universal Time) and civil time. The mapping takes a piecewise linear form as mapping parameters are periodically updated by tracking timing signals from a reference time, either a local or a remote one. Therefore, the mapping constantly compensates for the offset between the cycle counter and the reference time, and it ensures that the clock steering is smooth and responsive to short-term skew and drift of the cycle counter.

We then present two concrete realizations of the new clock steering method. In the first realization, we built a Pulse-Per-Second (PPS) generator to distribute PPS signals to each server via serial port. Each server then uses PPS as reference time in running the piecewise mapping algorithm. This way, all such servers achieve mutual synchronization. The second realization is based on low-latency timing message exchange. Exchange Timestamp Protocol (XTP) is designed to determine the clock skew and offset between a local client and a remote server. XTP leverages low-latency communication fabrics such as InfiniBand and a robust filtering algorithm to accurately estimate the delay and jitter between the client and the server. Such estimations are then used for deriving the clock offset, so the client can steer its own CCT time by following the remote server's CCT clock.

We further design an API for a better usage of the derived CCT. The output CCT, as well as the underlying linear mapping information, is stored in a shared memory page. Thus applications in the user space can access CCT without making system calls (except possibly to obtain the initial mapping). This way, large overhead and possibly induced inaccuracy is avoided. Besides, the designed API is able to handle Leap Second events in a smooth manner such that discontinuity is avoided when CCT is used to measure intervals across a Leap Second. Besides measuring intervals, CCT can be used as an independent time-keeping mechanism to evaluate other synchronization mechanisms such as the Linux system clock and NTP. As a future direction, we are also experimenting with using CCT to steer the Linux system clock.

We implement and evaluate the CCT mechanism in IBM BladeCenter clusters running Linux OS. We compare CCT clock offset with Linux system clock offset under NTP. The experiments show that the presented mechanism indeed achieves a mutual synchronization tightness of one microsecond. This is four orders of magnitude better than NTP when the NTP reference clock is a remote one, or three orders of magnitude better when the NTP reference clock is within the same cluster.

The rest of this paper is structured as follows. In Section II we briefly describe the piecewise linear mapping based clock steering method . In Section III we discuss two realizations of the method: one is based on PPS and the other based on

978-1-4244-2275-3/08/$25.00 ©2008 IEEE

Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on March 5, 2009 at 13:26 from IEEE Xplore. Restrictions apply.

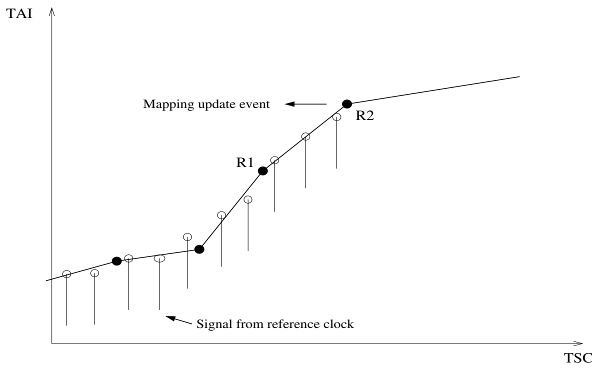

Fig. 1. Piecewise linear mapping

timing message exchange. In Section IV we describe the API and how CCT handles the Leap Second issue. Section V shows experimental results and Section VI concludes the paper.

II. Clock steering algorithm based on piecewise linear mapping

Our method leverages TSC, a CPU cycle register existing in most modern CPU chips (the register may carry a different name on architectures other than x86, e.g., it is named Time Base register on PowerPC). TSC counts from zero after the CPU reboots. Given the fact that TSC is a relatively stable clock source with high resolution, one can map TSC readings to TAI. Specifically, the mapping is as follows:

$$TAI \quad = \quad \text{slope} * (TSC - \text{tsc}_0) + \text{offset}$$

where *slope* is the linear mapping parameter which includes the unit conversion factor (ns per cycle) as well as the skew of the TSC-based clock. Since the TSC clock may still drift due to factors such as temperature change, the above linear mapping does not guarantee a stable estimate of TAI. Our solution is to continuously update the slope parameter by steering towards an external reference clock. The reference clock continuously emits signals (see Figure 1). Upon the arrival of each signal, the TSC reading and the time for the signal arrival (measured by the reference clock) are recorded. Now suppose R_1 is a leg corresponding to the current mapping with parameter slope_1. At periodic intervals - usually a couple of seconds - all the signal arrivals in one interval are used to determine the linear mapping of TSC readings to TAI. If one signal arrival deviates significantly from the linear mapping constructed by the rest of the arrivals, the arrival is labeled as an outlier and automatically removed in computing the linear mapping parameters. The computation proposes a new slope for the linear mapping. If the new slope is sufficiently different from the current slope a new leg of the piecewise-linear mapping R_2 will appear in Figure 1.

In the implementation, the piecewise mapping information is stored in a circular buffer which consists of a fixed number of entries, each corresponding to one leg. Each entry consists of a triple *(TSCval, TAIns, Slope)* indicating that the mapping Slope should be used after the time point *TSCval/TAIns*. All the entries are in circular order based on *TSCval*. The last entry in the buffer must always be an EOT (End-Of-Time) entry which has the largest possible *TSCval* of the field.

The mapping update is conducted by a daemon, which is implemented as an independently running program that wakes up from time to time. When a new mapping leg with slope *newSlope* is created, the update daemon selects a time point *newTAIns* in the near future, and updates the circular buffer according to Algorithm 1.

Algorithm 1 Mapping update

1: Find the EOT leg in the circular buffer.
2: Verify that the preceding entry is the current leg with respect to newTAIns, i.e. newTAIns is larger than the field TAIns in the current leg. If not, the update program ran too soon, and it should skip this update.
3: Compute newTSCval by reverse linear interpolation from newTAIns using the current leg (identified in Step 2 above). No special precautions are needed since we assume that this is the only updater.
4: Mark the next entry following the current EOT (in circular order) with a second EOT mark. When every entry has been written, this is effectively overwriting the oldest entry.
5: Fill in newTAIns and newSlope in the current EOT leg. At this point the new information is still effectively invisible to a concurrent reader, as the EOT mark is still there.
6: Overwrite the current EOT mark with newTSCval computed in Step 3 above. This is what instantiates the new leg. The second EOT mark (written in Step 4 above) becomes the new current EOT mark.

The mapping structure must be initialized, that is, we must provide an entry with a TSCval smaller than the current TSC reading. The initialization must be done before the updating procedure is started. The mapping update algorithm is lightweight because the circular buffer is pretty short, e.g. four to eight entries, so a full-length scanning would in fact be fast. Also, the update program is the only one that updates the map, so it can remember where the last leg is in the circular buffer, and avoid a scan to locate it.

As the circular buffer stores the piecewise linear mapping information, any program that requests CCT can locate the latest mapping entry in the circular buffer and map a TSC reading to CCT. This procedure is described in Algorithm 2.

Algorithm 2 Reading CCT

1: Read the current TSC and remember as TSCnow.
2: Scan the circular buffer until an entry is found whose TSCval is less than or equal to TSCnow, but which is followed by an entry whose TSCval is strictly greater than TSCnow (or is the last entry in the circular buffer). Remember the TSCval used in the comparison as myTSCval. This scan must succeed in at most N steps if the buffer holds N entries, and is in an initialized state. If no applicable entry has been found after N steps, report an error ”map not initialized”.
3: Pick up TAIns and Slope from the selected entry (as myTAIns and mySlope).
4: Verify that myTSCval still matches TSCval in the selected entry; if not, go back to Step 1. This can happen if the reader was interrupted, and in the meantime the update program has performed several updates, wrapping around the circular buffer. It is not likely that this would happen repeatedly.
5: Use linear interpolation to convert TSCnow to TAInanoseconds: *TAInanoseconds* = *mySlope**(*TSCnow* - *myTSCval*) + *myTAIval*

It may worth noting that the above algorithms are entirely lock-free, thus avoiding the kinds of lock delays often en-

Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on March 5, 2009 at 13:26 from IEEE Xplore. Restrictions apply.

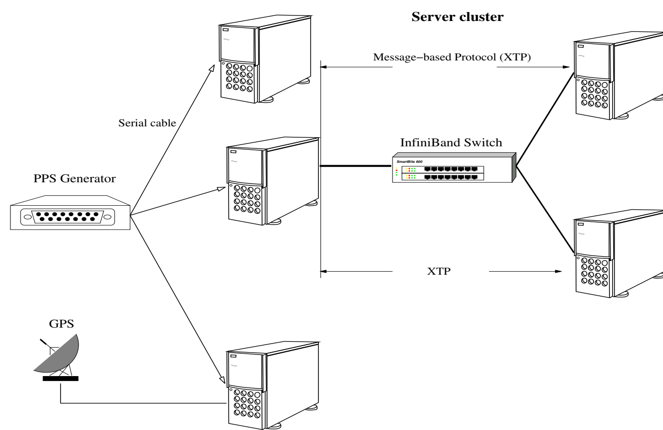

Fig. 2. System model

countered when a structure is accessed by one writer and concurrent readers. In a system with strong memory ordering no additional instructions are needed, though in system with weak ordering it would be necessary to insert so-called memory barriers at appropriate points. This would not affect the overall nature of the protocol.

There are two caveats with using the TSC as a means to derive time: (1) On some multiprocessor platforms running certain OS, TSCs on different CPUs are not synchronized with each other. E.g., when Linux kernel 2.6.22 runs on a server with a dual-core AMD Opteron processor, we observe that the TSC readings across two CPUs have an approximately 8-millisecond offset. (2) Most modern CPUs are able to adaptively change CPU frequency to lower CPU power consumption (e.g., the SpeedStep technology from Intel). For some processors this technology will further affect the TSC frequency. We believe both problems will go away on newer processors, because manufacturers have realized that a uniform and reliable TSC is valuable (and some other performance-monitoring register could be used to count actual CPU cycles). The multiprocessor issue can be dealt with if there is a way to know on which CPU the TSC is read (e.g. *rdtscp* instruction on recent AMD processors). The power management issue can be dealt with by providing a handshake between power management software and the CCT update daemon, to permit the map to be recomputed before user processes are dispatched after a change of frequency. In our experiments we have simply disabled power management.

III. Two realizations

We have designed and implemented two realizations based on the piecewise linear mapping. One is a hardware approach that uses PPS. The other is a software approach that uses a Timestamp Exchange Protocol (XTP). We will describe these two realizations and some practical issues in the rest of this section.

A. PPS-based clock steering

The first realization is based on a PPS generator and distributor. Most of our blade servers are located in a chassis which has a high density connector attached to the serial port of each blade. We built a 1U PPS distributor that provides PPS signals to multiple chassis. The transmission delay for PPS signal on the serial cable is about tens of nanoseconds. This factor is negligible considering that our targeted precision is one microsecond. The PPS distributor supports daisy-chaining, so multiple units can be used to extend reachability to multiple frames from a single PPS source. In order to accurately capture the PPS signal on each blade, we developed a serial port driver that uses interrupt reflection to capture rising pulses. The interrupt handler introduces less than 1 microsecond delay on modern blades under native (not virtual) Linux.

When PPS is used as the reference time for running our clock steering algorithm, we create a virtual character device which provides a public interface for accessing a memory page which stores TSC/TAI mapping information. This memory page is shared by the PPS driver and an asynchronous daemon. While the PPS driver records the TSC value for each captured reference signal, the daemon is responsible for updating the mapping by using the latest recorded TSC values. The shared page mechanism ensures that the two modules access the mapping context without requiring system calls (after setup).

When we experimented with virtualized environment (in this case, Xen), we observed that the delay incurred by capturing PPS pulses can be significant and erratic, which affects the accuracy of using PPS interrupts. We plan to address this issue by revising the Hypervisor.

B. Low-latency timing message exchange

In contrast to using local hardware as the reference clock, another realization of our clock steering method is purely based on software by using a remote server's clock as the reference clock. A protocol called Exchange Timestamp Protocol (XTP) is designed to determine the clock offset between a local client and a remote server. By using XTP, both the client and the server run physically concurrently. This makes it possible to exploit a stretch of time that happens to be free from system noise. The client and the server exchange low-latency timing messages in order to estimate their clock offset. The exchange procedure starts when the client sends a "doorbell" request to the server. Upon receiving the request, the server optionally sends an acknowledgement to the client. The client and the server then both proceed to concurrently exchange their respective timestamps with each other multiple times (say, four to ten times), in order to (a) warm up lookaside structures such as caches and TLBs and (b) to permit selection of the shortest round-trip time, which is least likely to be polluted by system noise. The captured best apparent forwards and backwards delays are then subject to convex-hull filtering which produces the clock offset and skew used to update the piecewise-linear map so as to track the reference clock.

XTP exploits two techniques for an accurate estimation of the delay and jitter between the client and server. First, XTP

Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on March 5, 2009 at 13:26 from IEEE Xplore. Restrictions apply.

leverages InfiniBand commonly existing in high-performance clusters. InfiniBand is a consolidated interconnect fabric for I/O and inter-process communication (IPC). Its attractive features are high bandwidth ($\geq$ 2.5 Gbps), low latency and moderate cost. In particular, InfiniBand supports RDMA (Remote Direct Memory Access) mode which allows message exchange without system calls. The RTT (Round-Trip Time) is about 10 microseconds and the jitter can be as low as 100 nanoseconds. Thus, by using RDMA mode, XTP effectively reduces the delay and the variance of jitter during the timestamp exchange, Also, XTP applies the convex-hull algorithm proposed in our earlier work [3]. This algorithm can effectively filter out longer-than-usual delays when deriving clock offset and skew.

When we employ XTP in the experiments which will be described in later sections, we use a static timing hierarchy, i.e., each client has a fixed set of neighbors to acquire reference time from. This is clearly not good considering single node failure and load balancing. Later we shall use our AP2P algorithm (Almost Peer-to-Peer) [4] which allows a dynamic topology and is thus significantly more fault-tolerant.

IV. API, Leap Second handling and applications

We have constructed an API that allows CCT to be used in a unified, convenient form. The piecewise mapping information, which is necessary for reading CCT time, is stored in a shared memory page. This design choice allows access without a system call, except possibly on first reference. One important advantage of the shared page method is that it requires no kernel changes. By using a shared page, we are able to define a user-level function that directly accesses the mapping information. Applications requesting CCT simply call this user-level function without the overhead caused by system call. Moreover, because the API is independent of the clock steering mechanism, the underlying algorithm is fully transparent to applications.

We define the API so that it deals completely with Leap Second issues, i.e., returns both UTC and TAI in a convenient form, and permits the actual Leap Second to be recognized, e.g, if the application wants to display 23:59:60, as required by the International Organization for Standardization (ISO). Since the API returns both UTC and TAI, users can either apply TAI for measuring intervals or using UTC to translate to their Locale regional time. The following is the data structure that is passed by reference to the ReadCCT() interface:

```
struct CCT {
  int  s;   // UTC seconds since 1970
  int  ns;  // nanosecond part
  int  nls; // Next/last Leap Second (TAI)
  int  lso; // Leap Second Offset
  long tsc; // 64-bit raw TSC value
}
```

To obtain TAI from the above struct, one only needs to add the Leap Second Offset (LSO) to the seconds field. TAI is incremented regularly, but UTC repeats a positive Leap Second (or skips a negative Leap Second), so it cannot be used unchanged to measure intervals that cross a Leap Second event.

The Leap Second information, i.e., *nls* and the applicable *lso*, is also maintained in the shared page; it is mostly static and would be updated when a new Leap Second event has been announced by the IERS (the International Earth Rotation Service), which happens at least every six months (announcing a new event, or declaring that no event is scheduled for the next few months). The information that is recorded consists of the current Leap Second Second offset, the next LSO, and nls, the TAI second of the next event. The piecewise mapping computes TAI, whose "seconds" component is then compared to nls. The applicable LSO is then subtracted in order to get UTC seconds. When no Leap Second event is planned, the applicable LSO can be the "next" one; the previous "current" having become stale when the last Leap Second event happened.

Note that ours is not the first attempt to handle Leap Seconds properly. In 1998 Markus Kuhn in [5] proposed a revised <time.h> that did two things: (a) it provided for explicit access to TAI (separate from UTC) and (b) it handled positive Leap Seconds in the UTC interface by running the fractional second past 999,999 microseconds up to 1,999,999 microseconds (or 1,999,999,999 nanoseconds, as applicable), thus avoiding duplicate UTC timestamps and permitting the recognition of the actual Leap Second. This is pretty elegant, and simply different from our solution – though we do offer one additional piece of information, namely the next (or previous) Leap Second Event, all the time, not just on Leap Second day.

Speaking of the usefulness of our derived CCT, many synchronization-critical and time-keeping applications can benefit from using CCT. For example, one can use CCT to evaluate NTP in a cluster environment (as we described in our later experiment section), or use CCT to measure the Linux system clock accuracy. As a future direction, we are planning to use CCT to steer the Linux system clock by using the system call *adjtimex*.

V. Experiments

We have implemented our design in IBM BladeCenter environments and compared it to NTP. We first evaluate our CCT implementation on 40 blade servers. These 40 servers are located in several chassis on the same rack. All the servers are running Linux with kernel 2.6.19 in Xen and have InfiniBand connections. By default all these servers enable NTP service and their common NTP clock source is outside the domain. The Round-trip Time (RTT) between a blade and the NTP clock source is up to 50 milliseconds. All these servers are connected to a common PPS generator and maintain both PPS-based CCT and XTP-based CCT. We choose one blade as a probing agent that runs native Linux (not under Xen) from its local disk. The probing agent continuously polls every other node requesting its Linux system time and CCT (both PPS-based and XTP-based). XTP uses the probing node's PPS-based CCT as the reference clock. The probing agent then

Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on March 5, 2009 at 13:26 from IEEE Xplore. Restrictions apply.

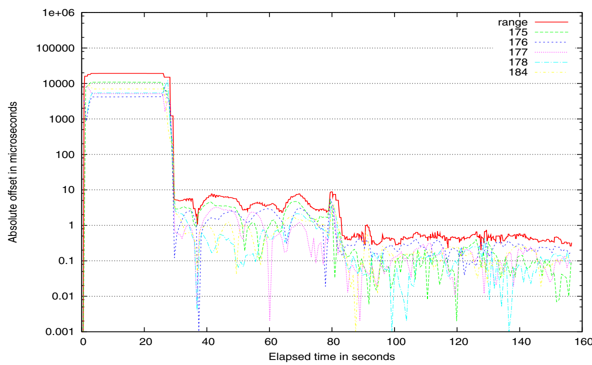

Fig. 3. Clock offset for testing CCT in a 40-blade cluster. The offset is relative to the reference time on the probing agent. "175"-"184" are blade IDs. The curve "range" (on the top) shows the maximum mutual offset between any pair of monitored clocks. Note that the Y axis uses logarithmic scale.

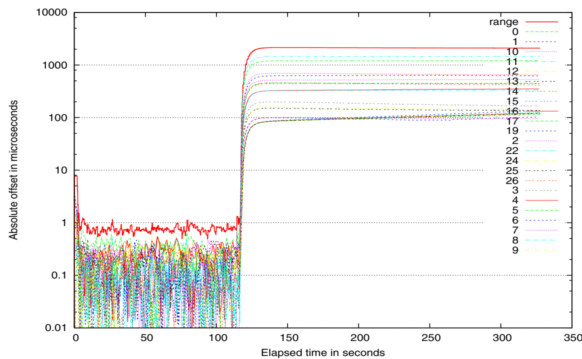

Fig. 4. Clock offset for testing CCT in a 31-blade cluster.

calculates the offset with respect to its own PPS-based CCT, which is treated as a common reference time, and derives from this a bound on the mutual offsets between all nodes other than the probing node.

For evaluation purposes, we consecutively run three time-keeping schemes: Linux system time synchronized by NTP, CCT synchronized by PPS running in Xen, and CCT synchronized by XTP. The experiment results are shown in Figures 3 where the offsets for five arbitrarily chosen individual blades and the overall offset range are shown. In the first 30 seconds, the measured offset is for the Linux time between the probing agent and every other individual blade. Such an offset ranges from a few ms to nearly 20ms. (In fairness, the remote NTP source used in that experiment accounts for the large offsets.) In the next time period (30 sec, 85 sec), the measured offset is for the PPS-based CCT on blades running in Xen. The magnitude of offsets lies in the 1 to 10 microsecond range. Starting from 85 seconds, the measured offset is for XTP-based CCT. This time, offsets fall below 1 microsecond. Our experiment demonstrates that both PPS and XTP exceed NTP in terms of synchronization precision. Compared to NTP, the proposed CCT mechanism reduces the mutual clock offset by up to four orders of magnitude.

In another scenario, we evaluate CCT on 31 servers in another BladeCenter cluster. Every blade runs local Linux with kernel version 2.6.9. One major difference between this scenario and the previous one is that all the 31 blades are configured to follow a common NTP clock source within the same cluster. The RTT between a blade and the NTP clock source is several millisecond at most. There are no PPS connections in this BladeCenter, thus every blade only maintains XTP-based CCT. Similar to the previous experiment, we select one blade as the probing agent and the provider of a reference time. In the first 110 seconds, we let the CCT on every other blade follow the CCT on the probing agent. As we see from Figure 4, the offset is mostly below 1 microsecond. Starting from 120 seconds, we let the CCT on each blade follows its own Linux system clock. This way, the measured offset, which is between 1 and 3 milliseconds, essentially reflects the offset for each blade's own Linux clock. Overall, NTP gives a nice synchronization in this case since the NTP clock source is in the same cluster. However, the CCT mechanism still reduces mutual clock offset by three orders of magnitude.

VI. Conclusion

Cluster-wide clock synchronization is important for many high-performance and critical applications. In this work we present a time-keeping mechanism for providing microsecond-level consistent time (referred to as CCT) in a cluster environment. The core component of the presented mechanism is a new clock steering method which maps CPU cycle counter readings to TAI. The mapping is in a piecewise form so that it steers the local clock to an external reference clock in a timely and smooth manner. We further introduce two designs for realizing the clock steering method. One is to use Pulse-Per-Second (PPS) as the reference clock. The other design is a XTP protocol that implements a remote reference clock based on low-latency timing message exchange. The derived CCT has an API which not only offers little overhead for usage but also handles Leap Seconds conveniently. Experiments on IBM BladeCenter clusters demonstrate that the proposed design achieves synchronization precision in the one-microsecond range. This is three orders of magnitude improvement compared with NTP solutions,

References

[1] C. Na, D. Obradovic, R. Scheiterer, G. Steindl, and F.-J. Goetz, "Synchronization performance of the precision time protocol," in *IEEE ISPCS*, Austria, October 2007.

[2] J. Ridoux and D. Veitch, "Ten microseconds over lan, for free," in *IEEE ISPCS*, Austria, October 2007.

[3] L. Zhang, Z. Liu, and C. Xia, "Clock synchronization algorithms for network measurements," in *IEEE INFOCOM 2002*, 2002.

[4] A. Sobeih, M. Hack, Z. Liu, and L. Zhang, "Almost peer-to-peer clock synchronization," in *IEEE IPDPS 2007*, 2007.

[5] M. Kuhn, "Proposed new <time.h> for ISO C 200X," Online paper, September 1998, `http://www.cl.cam.ac.uk/~mgk25/time/c`.

Authorized licensed use limited to: King Fahd University of Petroleum and Minerals. Downloaded on March 5, 2009 at 13:26 from IEEE Xplore. Restrictions apply.