Hardware Assisted Precision Time Protocol. Design and case study.

Patrick Ohly, David N. Lombard, Kevin B. Stanton

Intel GmbH
Hermülheimer Straße 8a
D–50321 Brühl, Germany

Abstract. Keeping system time closely synchronized among all nodes of a cluster is a hard problem. The Network Time Protocol reliably synchronizes only to an accuracy of a few milliseconds. This is too coarse to compare time stamps of fast events on modern clusters, for example, the send and receive times of a message over a low-latency network.
The Precision Time Protocol (PTP), defined in IEEE 1588, specifies a protocol which can substantially enhance the time accuracy across nodes in a local area network. An open source implementation of PTP (PTPd) relies on software time stamping, which is susceptible to jitter introduced by the non-realtime OS. An upcoming Ethernet NIC from Intel solves this problem by providing time stamping in hardware.
This paper describes our modifications which allow PTPd to make use of this hardware feature, and evaluates several approaches for synchronizing the system time against the PTP time. Without hardware assistance, PTPd achieved accuracy as good as one microsecond; with hardware assistance, accuracy was reliably improved and dependence on real-time packet time stamping in software was virtually eliminated.
Keywords: Cluster Clock, PTPd, IEEE 1588, Hardware Time Stamping

1 Introduction

1.1 Problem Statement

Keeping the system time closely synchronized among all nodes of a cluster is a hard problem. Each node typically has its own system time which is incremented based on clock ticks of some internal oscillator; these oscillators all run at slightly different frequencies; worse yet, that frequency will change in a non-deterministic way depending on environmental factors, in particular temperature.

In commodity clusters built from off-the-shelf components, these different nodes are only loosely connected. The Network Time Protocol (NTP) is typically used to synchronize the each node's clock by sending messages across the interconnect and measuring the delays. Because it is designed for wide area networks and only synchronizes infrequently, it only provides an accuracy in the range of milliseconds (see figure 5(a)). This is too coarse to compare time stamps of short events, like for example the send and receive of a message over a low-latency network—these have to be measured in microseconds or less.

Performance analysis tools like the Intel® Trace Analyzer and Collector compensate for clock offsets between nodes by transforming time stamps of events in a post-processing step based on clock comparisons typically done at the start and end of an experiment. This only allows for a linear interpolation, which is not sufficient to compensate for intermittent clock shifts: figure 1 shows clock offsets measurements done with a PingPong test between two machines. The lower part contains the visualization of some messages after about one fourth of the whole run. At that point the real offset deviated so much from the assumed linear one that after the transformation the send time stamp is greater than the receive time stamp. Messages in one direction (red, thick lines) seem to go back in time, while messages in the other direction (black, thin lines) seem to take much longer than they really did.

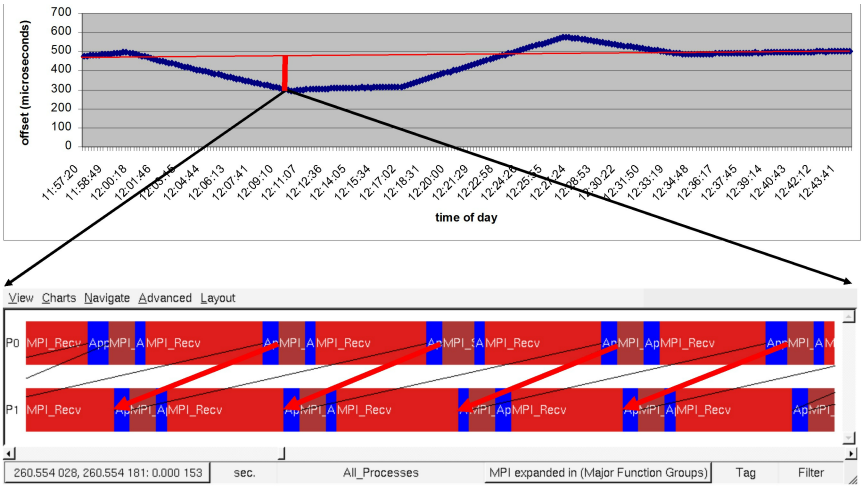


Fig. 1. Linear interpolation of time stamps does not fit the actual offsets, leading to messages apparently going backwards in time.

Non-linear interpolation can improve accuracy, but post-processing then still has the drawback that stamps taken on different hosts cannot be compared during data acquisition. Protocols which directly update the time stamp source (usually the host's system time) do not have this limitation.

1.2 Precision Time Protocol

The Precision Time Protocol (PTP, defined in IEEE 1588, [1]) specifies a standard protocol which can replace NTP within a local area network. It is based on measuring the send and receive time stamps of regular multicast messages sent from a master to all slaves (`Sync`) and of less frequent messages sent from each slave to the master (`Delay_Req`) (see figure 2). t' denotes a time measured

using the slave's clock; other values are either measured on the master or derived from both clocks. The time offset between master and slave is then calculated as follows.

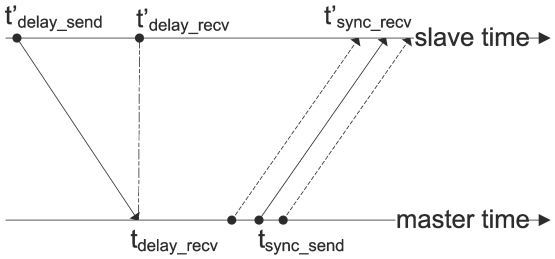


Fig. 2. PTP delay measurements.

$$masterToSlaveDelay := t'_{sync\_recv} - t_{sync\_send} \tag{1}$$
$$slaveToMasterDelay := t_{delay\_recv} - t'_{delay\_send} \tag{2}$$
$$oneWayDelay := (masterToSlaveDelay + slaveToMasterDelay)/2 \tag{3}$$

offsetFromMaster is the amount of time that the slave is in front of the master:

$$offsetFromMaster := t' - t \tag{4}$$
$$\Leftrightarrow t = t' - offsetFromMaster \tag{5}$$

$$\begin{aligned} slaveToMasterDelay &= t_{delay\_recv} - t'_{delay\_send} \\ &\overset{5}{=} t'_{delay\_recv} - offsetFromMaster - t'_{delay\_send} \\ &= \underbrace{t'_{delay\_recv} - t'_{delay\_send}}_{\text{corresponds to } slaveToMasterDelay'} - offsetFromMaster \end{aligned} \tag{6}$$

$$masterToSlaveDelay \overset{5}{=} \underbrace{masterToSlaveDelay'}_{\text{same reasoning as in 6}} + offsetFromMaster \tag{7}$$

$$\begin{aligned} &masterToSlaveDelay - oneWayDelay \\ &\overset{3}{=} (masterToSlaveDelay - slaveToMasterDelay)/2 \\ &\overset{7,6}{=} \underbrace{(masterToSlaveDelay' - slaveToMasterDelay')/2}_{\text{all time stamps from same host, assumed symmetric delays} \Rightarrow 0} + offsetFromMaster \end{aligned} \tag{8}$$

The substitution with zero in the last step is based on the assumptions that packet transmission times are perfectly symmetric and that there are no mea-

suring errors; in reality, varying transmission delays lead to inaccuracy. Asymmetry in the packet processing can lead to a constant offset between master and slave clocks. With equation 8 it is possible to recalculate the offset for each `Sync` packet; the *oneWayDelay* can be updated less frequently, e.g. after each `Delay_Req`.

Although not specified by the PTP standard, the usual reaction to the observed offset is to speed up or slow down the slave's clock so that the offset converges against zero over time. To cope with noise in the measurements and to ensure a smooth regulation of the system time, the open source PTPd ([2]) uses a *clock servo* which takes raw delay measurements as input, transforms them into filtered *oneWayDelay* and *offsetFromMaster* values, and then derives a frequency adjustment from these values.

PTPd relies on time stamping of network packets in software on the host. This is done inside the Linux or BSD kernel as follows: *incoming packets* are time stamped using the `SO_TIMESTAMP` feature of the IP stack. *Outgoing packets* are looped back via `IP_MULTICAST_LOOP`, with the time stamp generated for the looped packet used as the send time stamp. This assumes that the requisite processing in the IP stack happens close to the point in time when the packet leaves the host.

This time stamping is susceptible to varying delays and therefore reduced accuracy of the delay measurements:

In the hardware: modern Ethernet NICs can reduce interrupt rates by combining multiple incoming packets and only generating one interrupt for several of them

In the OS: before time stamping can take place, the OS needs to react to the interrupt; outgoing packets are also queued (Nagel algorithm) and take different code paths, so the approximation of the send time stamp via the looped packet can be very much off.

Section 3.1 shows that network load considerably impacts the accuracy of the software-only PTPd implementation to the point that it is barely better than traditional NTP.

1.3 Hardware Assistance in the NIC

An upcoming (but not yet announced) Intel Ethernet Network Interface Controller (NIC) will provide time stamping in hardware: the NIC itself has its own oscillator and maintains a NIC time driven by that oscillator. Similar to the `adjtimex()` system call, the NIC time can be sped up or slowed down to compensate for varying oscillator speeds. The host can read the current NIC time by accessing NIC registers.

Incoming PTP packets are detected by the hardware, triggering time stamping which copies the current NIC time and some relevant attributes (source ID, sequence number) of the packet into registers. As there is only one set of registers, and they are are locked until read, the PTP daemon must read them

before the next packet arrives or that packet will be passed through without time stamping. The PTP daemon can detect such an overrun by examining the additional attributes.

Outgoing packets are time stamped when requested by the PTP daemon; the daemon avoids the time stamp register overrun by sending a packet and waiting until it has left the host and was time stamped.

1.4 Hardware Assistance in the Network

Highly accurate time stamping inside the NIC is one component of good overall accuracy. PTP offers several concepts which address varying delays caused by the network itself:

PTP v2 transparent end-to-end clocks: Network equipment, e.g., a switch, adds the delays introduced by itself to a field in the transmitted PTP messages. This allows the PTP master and slaves to remove that varying delay from their measurements.

PTP v2 peer-to-peer clocks: In this mode, time is synchronized directly between node and the switch or router it is connected to. The network device then gets its time via PTP from some other master clock.

PTP v1 boundary clocks: The switch prevents multicasting the UDP packets; instead it acts as PTP master itself–the effect is the same as with peer-to-peer clocks.

With full support for PTP v2, accuracy at the hardware level as good as $20ns$ has been demonstrated before ([8]). The same kind of accuracy was achieved by another group at Intel using the Intel NIC prototypes. Major vendors of Gigabit Ethernet (GigE) and 10 Gigabit Ethernet network equipment are working on PTP v2 support. Although products are not announced or available yet, it is clear that support for PTP v2 in the network will be ubiquitous soon.

However, hardware support for PTP in the NIC and the network requires corresponding software support to achieve two additional goals:

- synchronizing against world time obtained e.g., via traditional NTP
- synchronizing the hosts' system times; reading the NIC time is slow (over $1\mu s$) and causes a potentially varying delay itself, therefore it is not practical to time stamp events occurring on the CPU using NIC time

The focus of our work was also on leveraging this technology for system time synchronization. The next subsections present the two different methods implemented to achieve that. Section 3 analyzes their effectiveness. Results had to be obtained without using PTP v2 features because PTPd has not been adapted yet and no suitable network switch was available. Many real-world clusters will be in the same situation, therefore it is worthwhile to investigate such a setup.

1.5 Assisted System Time

In this mode, PTP packets are still time stamped by the NIC, but the clock which is controlled by PTPd is the system time. For this to work, the time stamps generated inside the NIC are transformed into system time before feeding them into the normal PTPd machinery: each time a NIC time stamp is extracted from the NIC, the NIC–system time offset is measured and added to the NIC time stamp to obtain the corresponding system time stamp.

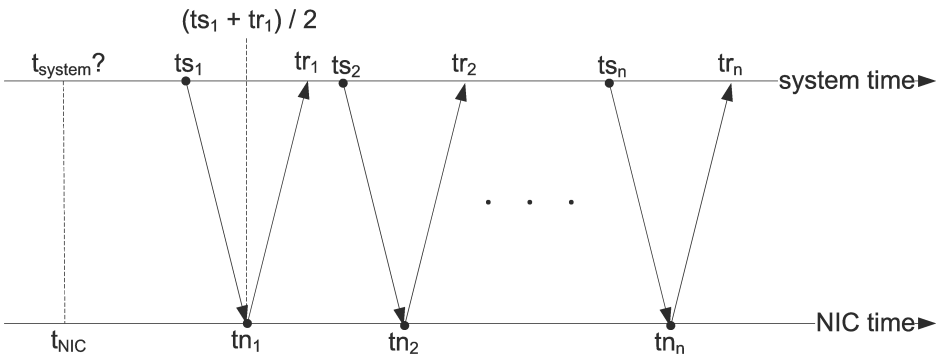


Fig. 3. Hardware assisted system time stamping: the offset which has to be added to t_{NIC} to obtain the corresponding t_{system} is approximated at a later time.

Figure 3 illustrates this process. The NIC–system time offset is measured by reading system time ts_1, NIC time tn_1, and another system time tr_1 and by assuming that $tn = (ts + tr)/2$. This measurement is repeated a handful of times (to average the result) in kernel space (to avoid disruptions). Outliers, recognized by long deltas $tr - ts$, are removed before averaging.

On a test machine (3), such a sequence of 100 measurements resulted in an average delta $ts - tn$ of $1741ns$ with a standard deviation of $43.2ns$. Durations were evenly spread between $1680ns$ and $1825ns$ with no obvious outliers. Load on the CPU, even when it involved many calls into the kernel, had no obvious impact. Because the measurements were so stable, the number of samples was reduced to 10 for real runs.

One possible drawback of this method is that NIC and system time typically run at slightly different speeds: therefore the NIC–system time offset at the time when the NIC time stamped a packet, and the offset when PTPd reads that time stamp, are slightly different. One possible improvement is to keep track of the overall skew s between NIC and system time, then calculate the system time stamps as $t_{system} = tn + offset + s * (tn - t_{NIC})$

This improvement was implemented as part of the "assisted" mode. It turned out that the skew was very low: during a 20 minute period when both system

time and NIC time ran without any correction factor, they diverged by $190.4ms$, which corresponds to $160ppm$. $tn - t_{NIC}$ was in the order of $50\mu s$, so the error incurred was insignificant at $8ns$ (and usually less) per sample.

1.6 Two-Level PTP

To avoid this clock offset approximation problem, the two-level PTP method runs two levels of clock synchronization (figure 4): at the network layer, PTP synchronizes NIC times across the network. Inside each node regular NIC–system time offset measurements drive another instance of the PTP clock servo which then adjusts the NIC time (on the master) or system time (on PTP slaves).

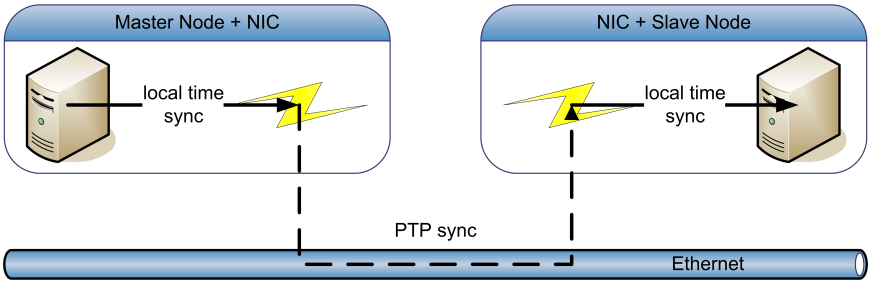


Fig. 4. Two-level clock synchronization: between system time and NIC time and between NICs. The arrows indicate how world time is pushed into the slave's system time.

The drift in the NIC–system time offset does not affect the PTP protocol when using this method. The drawback is an overall more complex system and potentially slower reaction to changes of the master system time because the PTPd clock servo is involved multiple times (twice between system and NIC and once between NICs).

2 Modifications to PTPd and Device Driver

2.1 PTPd Abstraction Layer

PTPd has an abstraction layer that hides system internals like sending and receiving packets and the time source. This abstraction assumed that the same time source is controlled by the PTP protocol as well as used for scheduling regular events inside the PTPd daemon.

This abstraction had to be extended: now the *PTP clock* is the one which time stamps PTP packets and is (optionally) controlled by PTPd, the *system time* is used by PTPd to sleep for certain amounts of time (typically one second). In the default software-only mode, these two clocks are identical. In combination with

the Intel NIC, the PTP clock is the NIC's internal time, and the host provides the system time.

There is a patch for PTPd which adds supports for hardware time stamping with the Freescale MPC831x family of chips, but it is based on a lot of `#ifdef` statements in the main PTPd code. The new architecture is modular, so the actual access to NIC specific APIs is hidden from the core PTPd engine. Adapting the modified PTPd to other hardware should be straight-forward.

2.2 Implementing the new Modes

A new command line option `-z` chooses different modes of using the Intel NIC. The default ("system") is the traditional mode without hardware support. In the other two modes ("assisted" and "both") packet loop back is disabled. Instead, the time stamp of an outgoing packet is retrieved from the hardware directly after sending it. The sender has to poll for the hardware time stamp because there is no notification mechanism; a while loop which sleeps for short intervals between checking the hardware avoids wasting CPU cycles. During testing under load, it was sometimes found that no time stamp was generated, most likely because the packet was dropped. To cope with that, the sending function returns after half a second. The PTP protocol then continues by resending packets.

Information about incoming packets is copied into a circular buffer in the hardware layer of PTPd each time the hardware is accessed. This tries to ensure that the NIC time stamp registers are cleared as soon as possible for the next packet. The protocol layer then asks for the time stamp matching the packet it is currently processing. The time stamp might not be available, for example if the hardware was not queried in time to free the incoming time stamp registers. Again this is treated as if the packet itself had been lost.

Implementing the assisted mode was easy because it simply replaces the system time stamp with the hardware time stamp plus NIC–system time offset. The two-level mode required further changes so that the PTPd clock servo could be instantiated twice, once for NIC-to-NIC synchronization and once for NIC-to-system synchronization. Regular NIC–system offset measurements drive the NIC-to-system clock servo.

2.3 Access to Hardware

The Ethernet driver was extended with several `ioctl()` calls to grant a process access to the new hardware features if it was sufficiently privileged to send raw packets. NIC–system time offset measurements are done inside the driver; this additional code is not part of the packet processing and thus does not affect normal packet handling at all. On the other hand, integrating it with the packet processing would give the driver the possibility to clear the incoming time stamp registers itself without waiting for PTPd to ask for them.

The current hardware only allows measurement of an offset, as shown in figure 3; the API was defined in terms of two delays because that is the more general concept.

3 PTP and Test Setup

In a typical PTP setup, one master node would coordinate its system time with some external source via NTP and broadcast the calibrated time via PTPd inside the cluster. For such a setup, PTPd on the master node with NTP has to be started with additional options that designate the clock provided by this PTPd daemon as a "secondary standard reference clock (`-s`) derived from NTP (`-i`)" and forbid PTPd to update the system time (`-t`):

```
ptpd -s 2 -i NTP -t
```

NTP must not be used on the slave nodes. Instead PTPd is started without any special parameters and then runs with the default settings: local clocks are considered unsynchronized, system time is modified by PTPd.

PTP chooses the master clock dynamically based on clock properties; in this setup the clock on the master node is used because it is better than the other nodes. If the PTPd on the master nodes fails, then the remaining normal nodes pick one master node among themselves—the time would still be synchronized within the cluster, just not with the outside world.

Master and slave nodes must be connected via Ethernet so that UDP multicast packages reach all nodes; it is not necessary that the master node is connected to the same high-speed interconnect used inside the HPC cluster. PTPd has very minimal CPU requirements, but it is sensitive to varying packet processing delays; ideally the Ethernet should not be used for traffic which delays the UDP multicasts. A PTPd user reported better results when using traffic shaping ([4]).

A four-node cluster (Intel SR1500AL 1U systems with dual Intel Xeon® 5100-series dual-core processors) was equipped with prototype Intel NIC cards which contain the PTP support. GigE (for PTP) and InfiniBand (for independent clock comparison) were used. The nodes ran Linux 2.6.23 in 64 bit mode. That kernel has a high resolution system time and grants userspace access to it via a *virtual dynamic shared object (VDSO)*. On systems where the `rdtsc` instruction returns synchronized results on all cores (which is normally the case on Intel systems) the `gettimeofday()` and `clock_gettime()` implementations provided in the VDSO are executed without entering kernel space. Older glibc version do not yet know about `clock_gettime()` in the 2.6.23 VDSO, so the routine was called directly to get system time with *ns* resolution and low overhead. The following experiment was repeated with different parameters:

1. Shut down NTPd and NTP on all nodes.
2. Invoke `ntpdate` on each node to synchronize system time against the same NTP server in a (for PTP relatively) imprecise way.
3. Run NTP on the master node.
4. Set up PTPd as describe above on slaves #0-#2 or (in non-PTP configurations) run NTP also on those nodes.

5. Measure and log the clock offset between each pair of nodes via PingPong test over Infiniband throughout the run.
6. Log the data gathered by PTPd itself and the NTP daemon.
7. Stop the run after 2 hours.

All PTPd instances in each run used the same synchronization mode. It would have been possible to combine different modes or even PTPd with some other PTP implementation because the different modes do not change the packets that are sent, only the time stamping inside the nodes. This experiment was done with the following modes:

ntp: all nodes use NTP for time synchronization,
system: PTPd synchronizes system time without hardware support.
assisted: PTPd synchronizes system time with hardware assistance.
both: two-level synchronization with PTPd.

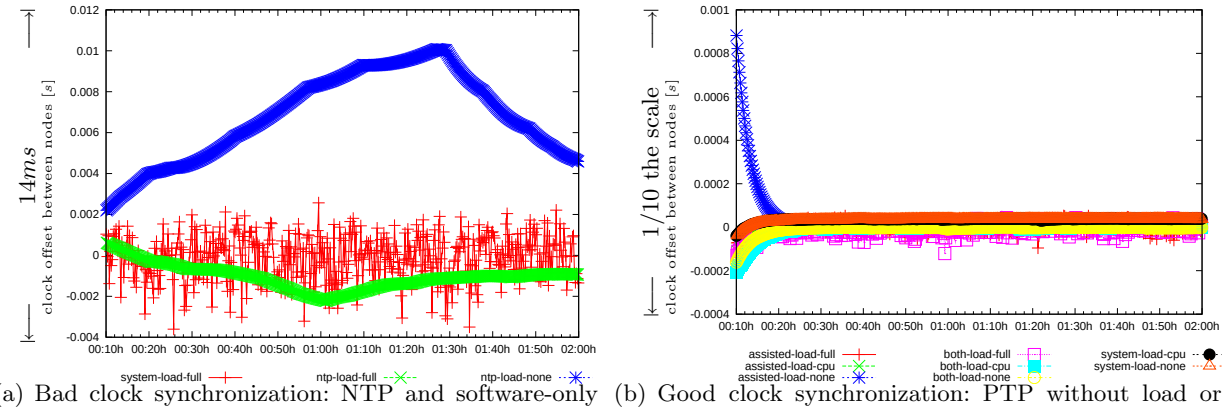
load-none: network and CPU are idle
load-cpu: the network is idle and the CPUs are busy.
load-full: the interface and the switch over which PTP packets are sent is kept busy by setting up the `netperf` TCP test in a ring so that each node sends and receives at maximum rate; all CPU cores are fully occupied by processes that run a busy loop.

Process priorities were chosen so that the PingPong test over Infiniband runs at higher priority than PTPd and the processes generating CPU load. The test gathers multiple time samples as quickly as possible in a short time interval, sorts them by their round-trip time, then discards outliers before using the remaining samples to calculate the offset. That way the offset measurement should not be affected when the CPUs are under load.

3.1 Comparison

Figure 5 shows the system time offsets between one slave and the master node measured via the Infiniband PingPong test during one run per configuration. The left graph shows the runs which did not converge towards a small clock offset: NTP is smooth, but inaccurate. PTP works much better in most configurations: the y-axis of the right graph covers *one tenth* of the y-axis of the left graph. PTP without hardware assistance produces worse results under load than NTP: with NTP, linear interpolation works reasonably well, but with PTP the clock jumps too much back and forth.

In general all clock correction mechanisms seem to have stabilized after 30 minutes, if they stabilize at all. Whether this convergence rate can be increased requires more experiments. In this paper we were more interested in accuracy of the stable phase, so all of the following statistics ignore the initial 30 minutes of each run.



(a) Bad clock synchronization: NTP and software-only PTP under load.

(b) Good clock synchronization: PTP without load or with hardware assistance.

Fig. 5. system time offsets during one run for each of the configurations

A picture says more than thousand words, but looks can also be deceiving... Therefore several statistics were calculated to get quantifiable results. Table 1 shows the values for all runs with "system-load-none". The standard deviation is the commonly used measure of how much samples varied. Different distributions can lead to the same standard deviation (e.g., samples equally distributed over a broad interval vs. samples well centered around the average with some far outliers). Therefore the percentage of samples in certain symmetric intervals around the median are also shown. This answers the more intuitive question of "how likely is it that the next sample falls within a certain error range?".

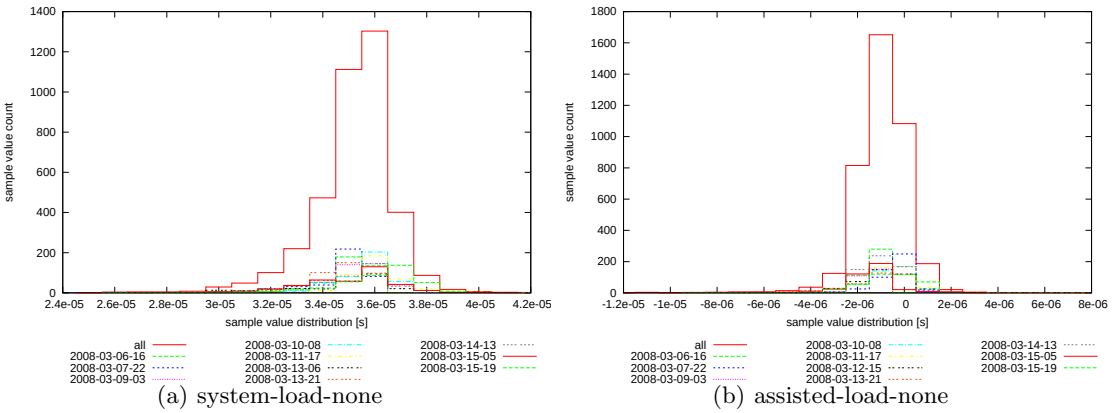
The last row calculates the same statistics over samples from *all* runs on the same machine with the same configuration, i.e., as if the data had come from one run of multiple times the duration of the actual 2 hour runs. This is based on the assumption that any difference in the stable phase of different runs is more likely to be caused by constant aspects of the hardware, the underlying synchronization method, and the test setup rather than the random start condition. The histograms of two different configurations (figure 6) support that assumption.

Table 2 compares the different configurations. It shows the statistics over the combined data of the stable phases of all runs. From it one can draw several conclusions:

1. NTP always resulted in much larger offsets than with the PTP based methods, regardless of CPU or network load. At least it always regulated the system time very smoothly, in contrast to "system-load-full".
2. Average and median are usually close to each other, with NTP being the exception.
3. Load on the CPU had some influence, but not much: configurations with CPU load are always worse than the corresponding configuration without, but the minor differences might not be significant.

Table 1. Statistics for all runs with configuration "system-load-none"; each run produced more than 400 samples. They converge against the same offset and thus can be combined.

median offset [s]	average offset [s]	std dev [s]	% of samples in +/- range around median			
			$1\mu s$	$10\mu s$	$100\ \mu s$	$1ms$
3.52e-5	3.49e-5	1.02e-6	76	100	100	100
3.50e-5	3.47e-5	1.11e-6	73	100	100	100
3.49e-5	3.49e-5	9.08e-7	75	100	100	100
3.56e-5	3.56e-5	9.34e-7	68	100	100	100
3.56e-5	3.57e-5	1.05e-6	76	100	100	100
3.51e-5	3.43e-5	2.05e-6	57	100	100	100
3.56e-5	3.53e-5	1.08e-6	74	100	100	100
3.47e-5	3.42e-5	1.68e-6	56	100	100	100
3.53e-5	3.48e-5	2.06e-6	51	99	100	100
3.63e-5	3.58e-5	2.07e-6	56	99	100	100
All samples combined:						
3.53e-5	3.50e-5	1.55e-6	63	99	100	100



(a) system-load-none (b) assisted-load-none

Fig. 6. Histograms of multiple different runs (low curves) and all runs combined (high curve); bin size always $1\mu s$.

Table 2. Combined statistics over all runs with a certain configuration, sorted by configuration and standard deviation. Each configuration is covered by at least 5 runs.

configuration	median offset [s]	average offset [s]	std dev [s]	% of samples in +/- range around median			
				$1\mu s$	$10\mu s$	100 μs	$1ms$
ntp-load-full	-2.26e-3	-2.33e-2	3.66e-2	0	0	0	10
ntp-load-none	-1.96e-3	-2.68e-3	7.35e-3	0	0	5	17
ntp-load-cpu	-3.63e-3	-3.60e-3	3.21e-3	0	0	5	31
system-load-full	1.94e-5	-9.10e-5	1.18e-3	0	0	5	59
system-load-cpu	3.85e-5	3.83e-5	1.86e-6	53	99	100	100
system-load-none	3.59e-5	3.57e-5	1.52e-6	63	99	100	100
both-load-full	6.69e-6	2.38e-6	2.06e-5	3	39	99	100
both-load-cpu	-1.73e-6	-2.54e-6	5.15e-6	21	95	100	100
both-load-none	-1.44e-6	-1.68e-6	2.88e-6	39	99	100	100
assisted-load-full	2.42e-6	4.07e-7	2.78e-5	2	29	99	100
assisted-load-cpu	2.47e-7	2.41e-8	1.97e-6	59	99	100	100
assisted-load-none	-4.79e-7	-4.02e-7	1.19e-6	66	99	100	100

4. The accuracy of all PTP based methods suffers considerably when the network is fully loaded. The load affects the delays inside the network, which is a common problem of all methods (section 3.2 below investigates that in more detail).
5. The network load also affects processing inside the host; this negative effect is successesfully avoided when using hardware time stamping in the NIC: the "assisted-load-full" and "both-load-full" have a 42x resp. 57x smaller standard deviation than "system-load-full".
6. The "assisted" mode provides better results than "both" mode without network load; it is slightly worse under load. Before drawing further conclusions it is necessary to retest with hardware suppport in the network to make results more stable.

3.2 Low-level PTPd Synchronization

As seen in the previous section, network load directly affects accuracy. This section examines how PTPd deals with the measured data.

Figures 7 and 8 show the raw delays measured via hardware time stamping (*masterToSlaveDelay*, *slaveToMasterDelay*) and the *offsetFromMaster* derived from them by PTPd via the clock servo. Compared to the other values the derived one-way delay changes very little and is therefore only shown in a separate graph (figure 8(b)) together with the corresponding raw one-way delay. The raw one-way delay and offset can be calculated directly from the raw delay measurements: directly after each *slaveToMasterDelay* update they are fairly accurate because both this *slaveToMasterDelay* and the most recent *masterToSlaveDelay* are likely to be close to each other.

It is more instructive to look at the statistics of the raw values rather than their graphs: the raw delay has an average value of $4.2\mu s$ (GigE with no OS involved and no load on the switch) and $0.09\mu s$ standard deviation over the large time interval. Under full load the average increases to $66.2\mu s$ and the deviation to $43.8\mu s$: the deviation is several orders of magnitude worse. This degradation of the measurement quality directly affects the clock synchronization quality: the deviation of the offset is much worse with load (figure 8) compared to the deviation without load (figures 7(c), 7(a), 7(b)). It is obvious that improving the quality of delay measurements at the network layer is essential for good clock synchronization quality.

When looking at the operation of PTPd in more detail for a small time interval where clock adjustment was stable (figure 7(c)) one finds that both *masterToSlaveDelay* and the adjustment basically alternate between two values. The differences are very small, well below $1\mu s$ for the offset. It seems like the adjustment factor is always a bit too large or too small. Tweaking the filtering parameters of the clock servo might help.

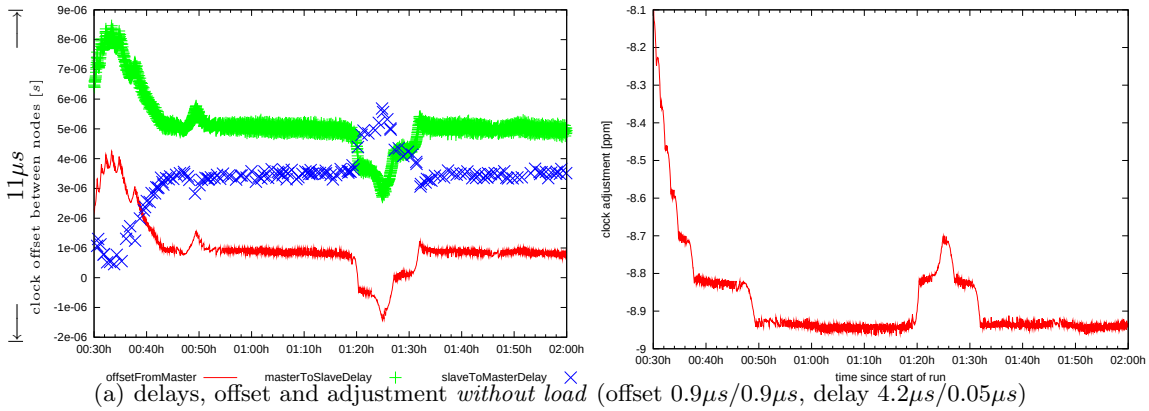
In the larger time interval (figure 7(a)) there are periods in the first slave where the offset is larger than average, in particular at the beginning and again at around 1:25h. Without better support for latency measurements inside the network it is impossible to tell what exactly caused these spikes: when network delays change, the measured *masterToSlaveDelay* changes first because it is sampled more frequently. This leads to a change in the *offsetFromMaster*, just *as if* the clocks had started to drift apart. PTPd assumes that latencies do not change, so it adapts the clock frequency. Some time later, the *slaveToMasterDelay* also gets updated. Now it is obvious that the change in the *offsetFromMaster* was in fact caused by a change of network latency, but the clock already ran for a while with the wrong correction factor; now the clocks have *really* drifted apart.

Figure 9 illustrates this negative effect of varying network traffic on clock synchronization. During those runs the netperf TCP benchmark between slave #0 and master was started for 20 seconds every 5 minutes to generate non-constant load on the network. The offsets were measured with the InfiniBand Ping Pong test and thus are not affected by changes of Ethernet latency. The offsets vary quite a lot more over the duration of each run than during runs without load.

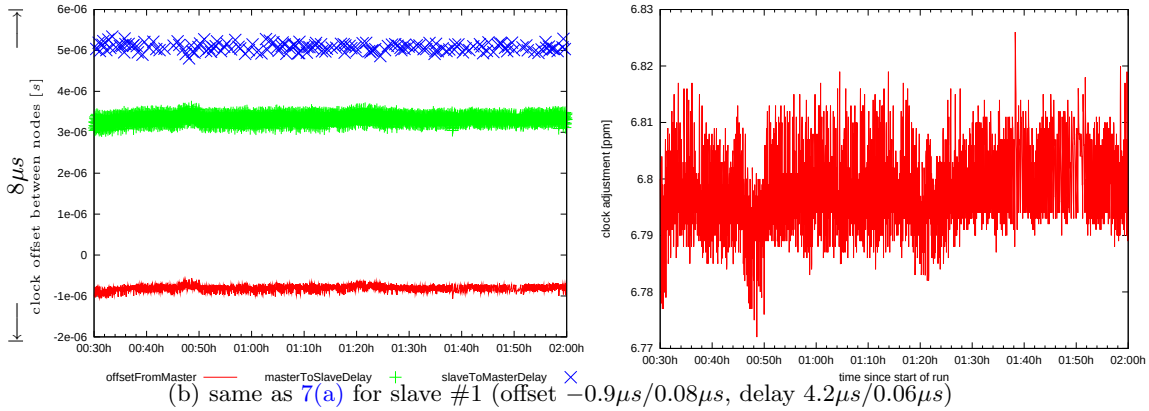
3.3 One-way Synchronization and Slave-to-Slave Offsets

Measuring *slaveToMasterDelay* introduces an additional source of errors. One could disable measuring it completely and replace it with a fixed value, e.g., zero. This will lead to a constant offset between master and slaves, but all slaves would have the same offset and thus could still be compared against each other. As seen before, sampling the *slaveToMasterDelay* is important to detect varying latencies and thus was always used for these experiments.

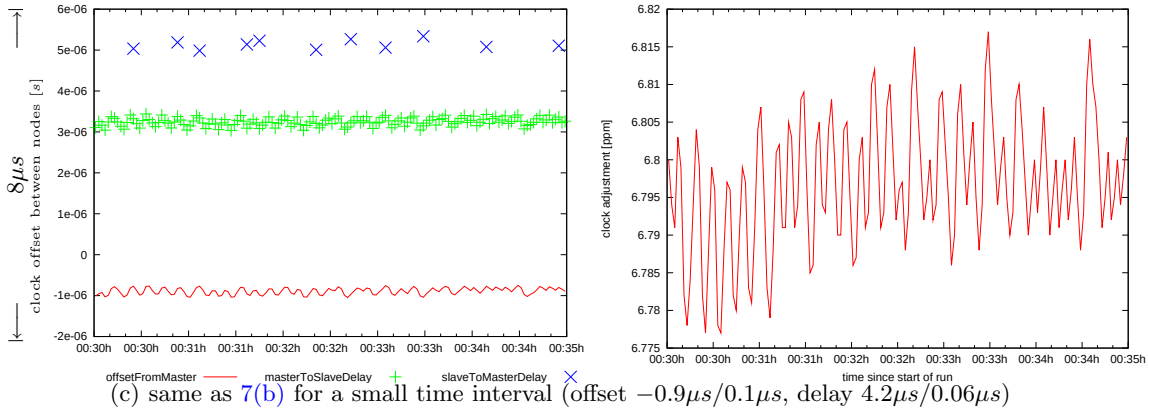
Table 3 compares the clock synchronization between slave/master and slave/slave for the two most promising synchronization methods with and without load. *All*



(a) delays, offset and adjustment *without load* (offset $0.9\mu s/0.9\mu s$, delay $4.2\mu s/0.05\mu s$)

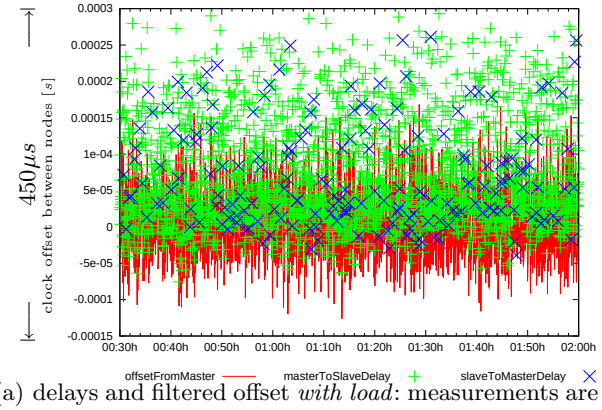


(b) same as 7(a) for slave #1 (offset $-0.9\mu s/0.08\mu s$, delay $4.2\mu s/0.06\mu s$)

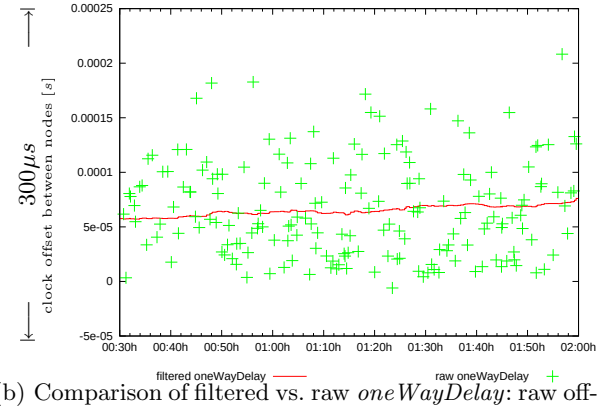


(c) same as 7(b) for a small time interval (offset $-0.9\mu s/0.1\mu s$, delay $4.2\mu s/0.06\mu s$)

Fig. 7. Data gathered by PTPd via hardware time stamping *without load*. The numbers in brackets are the average raw offset and its standard deviation, plus the same for the one-way delay. They were calculated from the raw measurements and not the filtered *offsetFromMaster* shown in the graphs (figure 8(b) illustrates the difference).

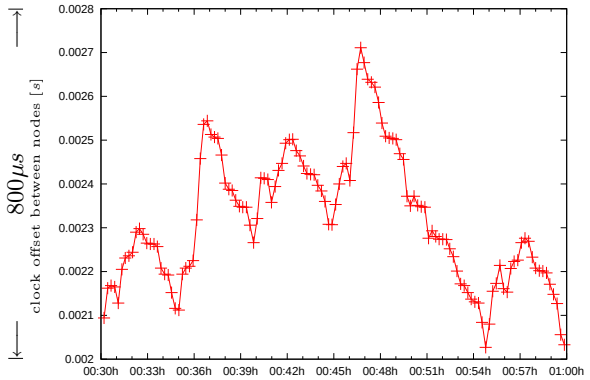


(a) delays and filtered offset *with load*: measurements are too noisy to be useful

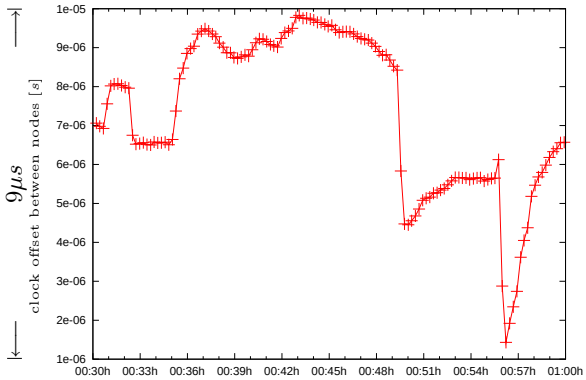


(b) Comparison of filtered vs. raw *oneWayDelay*: raw offset $-5.5\mu s/52.9\mu s$, raw delay $66.2\mu s/43.8\mu s$.

Fig. 8. Data gathered by PTPd via hardware time stamping under *constant network load*. Load considerably increases latency and variation of the measurements.



(a) software-only PTP



(b) assisted PTP

Fig. 9. Varying load on the network prevents accurate clock synchronization: the offsets do not converge as they do without load. In this example hardware assisted PTP is affected, but still about 100x better than without assistance.

slaves must have been affected the same way at the same time by varying network delays, because the effect cancels itself out when comparing the clocks of two slaves. The results of the slave/slave comparison are usually better than those of the slave/master comparison.

Table 3. Clock comparison slave #0–master and slave #0–slave #1.

configuration	peer	std dev [s]	% of samples in +/- range around median			
			$1\mu s$	$10\mu s$	100 μs	$1ms$
assisted-load-none	master	1.04e-6	70	100	100	100
assisted-load-none	slave #1	3.67e-7	96	100	100	100
assisted-load-full	master	2.36e-5	2	31	100	100
assisted-load-full	slave #1	3.41e-5	1	23	99	100
system-load-none	master	1.10e-6	72	100	100	100
system-load-none	slave #1	6.62e-7	87	100	100	100
system-load-full	master	1.19e-3	0	0	6	59
system-load-full	slave #1	2.65e-4	0	2	28	99

4 PTP in HPC Clusters: Benefits and Obstacles

The most obvious benefit of having a highly accurate synchronized clock in the whole cluster is for performance analysis with tracing tools: the problem illustrated in figure 1 is avoided as soon as the accuracy is considerably below the one-way latency of small messages.

One problem with running another daemon on each node of the cluster is that it adds to the *system noise*, which can affect the scaling of parallel applications ([3], [7]). With plain PTP v1, all daemons receive all multicast packages, including the slave–to–master packets that the other slaves do not have to process. PTP v1 boundary clocks or v2 peer-to-peer clocks would overcome this problem and reduce interruptions to handling of the `Sync` packet, which is by default sent every $2s$. An in-kernel implementation of PTP would be possible and should avoid this effect.

A synchronized clock might also be useful to enhance performance: *gang scheduling* ensures that all processes of a parallel application run on their respective nodes at the same time ([5], [6]). A well-synchronized cluster clock makes this scheduling more accurate. When scheduling the application is not possible, then perhaps scheduling the interrupting background activities so that they occur at the same time might be feasible instead ([7]).

5 Conclusions

Hardware support for time stamping of PTP packets proved to be essential for high clock synchronization accuracy: under constant network load the hardware

support in the Intel NIC increased the accuracy by a factor of around 50 (table 2).

Without load PTP managed to keep the majority of the offset measurements inside an interval of $+/-1\mu s$, with and without hardware support. These measurements were done via a different interconnect using the normal system time of the hosts; therefore they are representative of real-world use cases in HPC clusters.

With load the accuracy is considerably lower. To achieve sub-μs accuracy and to become resilient against load, it will be necessary to use network equipment which supports PTP. Major vendors are adding this feature to their switches and have already demonstrated $+/-20ns$ accuracy.

While switches that support PTP are anticipated, none was available for these experiments. Colleagues who have measured link delay through PTP-enabled switches found that low-level PTP accuracy was much better than 1us and (importantly) was independent of network traffic. The new "assisted" mode of PTPd directly translates NIC time stamps into system time stamps with very little additional noise; therefore we hope to reproduce their results also at the system level using PTP switches once available.

6 Acknowledgements

We thank all our colleagues who made this work possible and/or contributed to it. Chris Hall wrote the initial patch for the kernel driver and a simple PTP daemon using it. He shared that code and patiently answered all ensuing question. The two-level PTP method is based on his suggestions.

Radheka Godse and Debbie Sullivan helped to get access to documentation and up-to-date driver sources, which were necessary to work with Linux 2.6.23. Mike Cooper and Tim Walker provided the necessary NIC prototype cards.

Various people answered questions or participated in discussions around PTP (Nathan Mather, Ido Dinerman, Elad Benedict, Roy Larsen, Hans-Christian Hoppe) or reviewed this paper (Gerd-Hans Krämer, Peter Post, Thomas Kentemich, Georg Bisseling)–all remaining mistakes are of course entirely our own.

Credits are due to Kendall Correll for developing PTPd and making its source code freely available. Finally, this work would not have been possible without Joe Throop's, Bill Magro's and Karl Solchenbach's permission to spend time on it.

References

[1] http://ieee1588.nist.gov/. 2

[2] http://ptpd.sourceforge.net/. 4

[3] Rahul Garg and Pradipta De. Impact of noise on scaling of collectives: An empirical evaluation. In *Lecture Notes in Computer Science*, volume 4297/2006, pages 460–471, Berlin/Heidelberg, 2006. Springer. 17

[4] Gertjan Hofman. vlan prioritizationn & filtering results. http://sourceforge.net/forum/forum.php?thread_id=1937491\&forum_id=469207. 9

[5] José E. Moreira, Hubertus Franke, Waiman Chan, Liana L. Fong, Morris A. Jette, and Andy Yoo. A gang-scheduling system for asci blue-pacific. In *HPCN Europe '99: Proceedings of the 7th International Conference on High-Performance Computing and Networking*, pages 831–840, London, UK, 1999. Springer-Verlag. 17

[6] Ronald Mraz. Reducing the variance of point to point transfers in the ibm 9076 parallel computer. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 620–629, New York, NY, USA, 1994. ACM. 17

[7] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 55, Washington, DC, USA, 2003. IEEE Computer Society. 17

[8] Michael D. Johas Teener, Benjamin Hur, and Dawey (Weida) Huang. Timing performance of a network of low-cost ieee 802.1as network elements. In *International IEEE Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, Vienna, Austria, 2007. http://www.johasteener.com/files/IEEE_ISPCS07_802_1AS_Performance_d6.pdf. 5