PPSi – A Free Software PTP Implementation

Pietro Fezzardi
Università degli Studi di Pavia
Pavia, Italy

Maciej Lipiński
CERN, Geneva, Switzerland
Warsaw Univ. of Tech., Poland

Alessandro Rubini
Independent Consultant
Pavia, Italy

Aurelio Colosimo
Independent Consultant
Milano, Italy

Abstract—**This paper describes a new open source implementation of the Precision Time Protocol (PTP) [1] called PTP Ported To Silicon (PPSi) [2]. It was developed to fill in a niche in the free software world for a PTP daemon that is easily portable to a wide range of architectures and highly modular to enable protocol extensions — two key requirements of its driving force, the White Rabbit (WR) Project [3] [4]. PPSi's core protocol code is common for all the supported architectures ranging from a Linux PC to a soft–core processor running in a Field Programmable Gate Array (FPGA) — a feature minimizing code duplication, easing debugging, and facilitating new developments. This paper gives an overview of PPSi's internals describing design choices as well as the means of achieving portability and extensibility. A detailed example of a simulator architecture proves the design advantages. With an increasing number of supported architectures and a wide use in WR networks, PPSi is becoming an appealing PTP implementation also outside of the White Rabbit Community.**

I. Introduction

PTP Ported to Silicon (PPSi) [2] is a portable Precision Time Protocol (PTP) [1] implementation developed for the White Rabbit Project [3] [4] and licensed under the GNU Lesser General Public License (LGPL) [5].

White Rabbit [6] (WR) is an emerging technology designed for high–accuracy synchronization and based on existing standards. WR can be used in networks spanning several kilometers to synchronize thousands of nodes with sub–nanosecond accuracy and tens–of–picoseconds precision (mean offset and standard deviation, respectively). At the same time WR can guarantee deterministic and reliable data delivery with low–latency, without affecting synchronization [7].

The development of WR started at CERN as an effort to design the next generation replacement for the existing timing and control system of the accelerator facilities. Today, it has grown far beyond that: many research centers and private companies are taking part in WR development or considering its adoption.

WR is based on well known and existing standards and ideas, namely Ethernet (IEEE 802.3) [8], L1 syntonization (used also by ITU–T [9]) and the Precision Time Protocol (IEEE1588) [1], offering many potential scientific and commercial applications. A considerable effort was made to enhance the PTP protocol to achieve sub–nanosecond accuracy of synchronization while keeping it compatible with the standard and inter–operable with other implementations. WR defines a PTP Profile for high–accuracy applications, extending the protocol to what is called WRPTP [6][10]. The mechanisms used in this profile are now being evaluated by the IEEE P1588 Working Group [11] for adoption in the next revision of the standard.

PPSi is the result of a development effort aimed at providing full support for the newly defined WRPTP extension. The daemon is required to work in diverse environments, both hosted (e.g. the Linux–based WR switch) and freestanding (e.g. the WR end node with no operating system). This is achieved by using a modular design that separates the PTP protocol code from the required interactions with the specific run time environment. This approach eases further porting to new architectures and adding support for other PTP profiles.

In this paper we give a general overview of PPSi (section II), focusing on the modular design and stressing the technical choices aimed at portability (section III and IV). In section V we list and describe briefly all the architectures currently supported by PPSi. Finally in section VI we present an example of how PPSi's portable design was used to implement a self–test environment for performance tuning of the non–WR servo parameters; this simulation is supplemented with field–test results in section VII.

The discussion covers release 2014.07 of the code base, freely downloadable and distributable, that can be found in the public git repository of the project [12].

For a better comprehension of the text, some definitions are here provided: a *physical port* is the hardware performing the physical connection; a *PTP port* is a port as defined in the IEEE1588 protocol, i.e. a virtual input receiving PTP protocol packet; by instantiating a different data structure (a *PPSi instance*) for each PTP port, PPSi supports more than one PTP port running on the same physical port, e.g. one using UDP and another one on a raw Ethernet channel.

II. PPSi Overview

PPSi is the PTP daemon that currently drives White Rabbit networks. It is written in C language to be able to support size–constrained environments, such as microcontrollers. The code base started as a completely new design based on modern programming techniques; we circulated an initial proof–of–concept document and prototype code in WR development circles [13], and after an approval the actual development started in December 2011. At the outset, PTP protocol code came from the PTPd project [14], the main working implementation available as free software when the White Rabbit project started. Later we migrated WR–PTPd to the new PPSi architecture, with a completely different design aimed at portability and modularity. In the process, we followed an

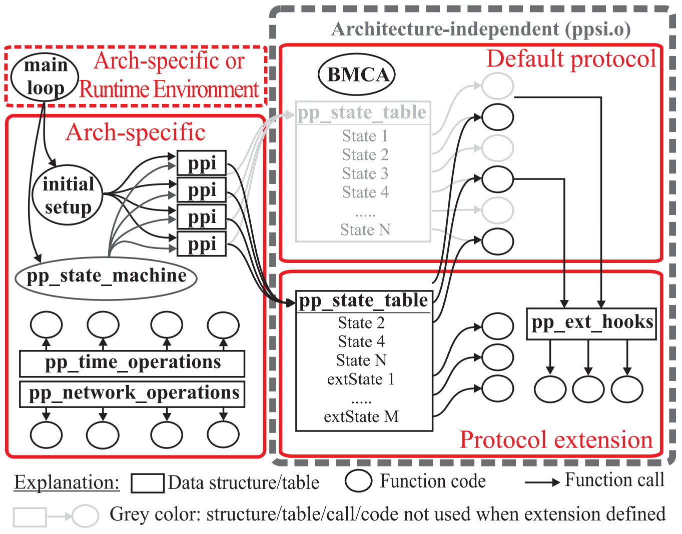

Fig. 1. PPSi architecture.

evolutionary rather than revolutionary process, to be able to quickly identify any regressions. Currently, very little remains of the original code base.

The source code is distributed under the GNU LGPL, which respects the original PTPd licensing terms, ensures the program's core will remain available to anyone, and allows support for new architectures even when the respective vendor does not want to disclose its own timestamping mechanisms.

PPSi can operate as an Ordinary Clock (OC) or as a Boundary Clock (BC) handling multiple ports. It supports PTP over both raw Ethernet or UDP, though currently only IPv4.

The user can configure different working modes for each port; PPSi supports several PTP ports on a single physical port which renders it useful to support both Ethernet and UDP mappings simultaneously. PPSi can be built for a number of different *architectures*, where the architecture code defines all interactions of the protocol code with the specific run time environment. Another major build–time option is a choice of the supported extensions, i.e. whether or not to include WRPTP. More fine–grained configurations can be selected at run time.

Diagnostics in PPSi provide different loglevels for each port and each code subsystem (e.g.: state machine, servo loop, frame I/O) that can be changed at run time. Each architecture provides the means for delivering diagnostics to the user. Frame diagnostics can go up to supporting complete dumps of sent and received data items — a feature which revealed useful when dealing with links between two WR nodes where no network sniffer is available.

III. PPSi Design Choices

The main design goal of PPSi, which differentiates it from other free software implementations on the market, is being self–contained. The top–level *Makefile* builds an object file, `ppsi.o`, that looks like a library to the calling code; the object file contains architecture–independent protocol code while the calling code is architecture–specific, as depicted in Figure 1. After initial setup, the *main* function of the calling code refers to a single entry point to the protocol code, `pp_state_machine()` — a procedure that operates synchronously and returns immediately: the caller passes a network frame as argument to the procedure, if available, and receives back a delay value to wait before calling the procedure again. Portability is thus achieved by removing any interaction with the outside world from the core of PPSi.

The code of PPSi can thus be divided into 3 areas depicted in Figure 1: the architecture–independent PTP "default protocol", the "architecture–specific" glue code, and profile–specific "protocol extension", such as WRPTP. The architecture–specific code provides timing and frame I/O capabilities by means of object methods associated to the PTP port (called `ppi`, PPSi instance). By filling in `pp_time_operations` and `pp_network_operations` structures, each architecture offers a complete link between the protocol engine and its actual run time environment (additionally, some architectures provide a main loop, as described later). Similarly, a custom PTP profile is implemented by providing function pointers within the `pp_ext_hooks` structure. Using these functions, by decoding Type–Length–Value (TLV) tuples in the announce message, profile code can choose to further participate in the communication and the Best Master Clock Algorithm (BMCA) decisions, or let the default PTP stack proceed unmodified.

The architectural split of PPSi anticipates the new layering description of the IEEE1588 standard. The media–independent layer ("default protocol") defines clear *interfaces* with the media–dependent part ("arch–specific") as well as option– and profile–specific operations ("protocol extension"). The latter can also interface directly with the media–dependent layer, if necessary (e.g. control of Layer 1 syntonization in WR).

Build–time configuration of the architecture and extensions is performed using Kconfig. The tool, derived from the Linux kernel, is easily integrated into other software packages and is being widely adopted, becoming the de–facto standard for package configuration. Kconfig offers both interactive and non–interactive configuration, the latter by providing a pre–built `.config` file. PPSi uses the non–interactive feature to optionally compile for all supported architectures, thus giving a complete build–time coverage of the code, helping developers in identifying portability issues or new unsatisfied external dependencies.

Run–time configuration happens through "configuration strings" that the run time environment feeds to PPSi. Such configuration defines the mapping and profile of each port, clock quality, diagnostic loglevels of different subsystems and other variables. The configuration parser is a part of the PPSi core to ensure its availability in all the environments. Each of the 3 areas of PPSi can provide an array of configuration keywords, i.e. global items, as well as architecture–specific, and profile–specific keywords. Consequently, special needs can be addressed locally by the individual use case avoiding global changes to a centralized parser.

```
struct pp_state_table_item {
     int state;
     char *name;
     pp_action *f1;
};
```

Listing 1. Data structure for state table.

The main loop, providing the calling code, is not a proper part of PPSi — providing or not the main loop, as well as the associated linking rule, is up to the architecture–specific code. While hosted architectures usually provide a main loop and a `main` function, a typical freestanding implementation does not. This is because, in such environment, the object files for all the applications are linked together with an externally provided main loop, be it a Real–Time Operating System or a custom procedure to boot the processor. For example, the "White Rabbit PTP Core Software" [15], driving the WR node, deals with a number of management activities while running WRPTP. The PPSi build for `arch-wrpc` is a bare `ppsi.o` which acts as a function library within the main program of the soft processor.

IV. PPSi Internals

As said, the core of PPSi is the state machine which in turn relies on the architecture–specific network and time operations.

A. State Machine

PPSi runs one state machine for each PTP port being configured at start–up; each port is defined by a `struct pp_instance` object (*ppi*) which runs a global `pp_state_table[]`, as presented in Figure 1. The state table associates allowed states with functions called when the state is executed as depicted in Listing 1. The actual content of the state table, depends on whether PPSi is built for the default or profile–extended protocol — although a profile can refer to the default state table. Being compliant to IEEE1588 is up to the actual code: PPSi forces no policy in that respect. Actually, the implementation is a "pure" network–driven state machine, with nothing PTP–specific in the engine itself.

B. Time operations

One time operations structure is instantiated for each `pp_instance` (i.e. per PTP port). The structure collects all time–related interactions between the PPSi protocol code and its run time environment. The structure is shown in Listing 2.

Timestamps are hosted in a `TimeInternal` [1] structure which includes high–precision fields, although they are only used by the WR profile. The `set()` method is only used to implement time jumps when the slave finds its time offset is too big; adjustments can be offset–based or frequency–based, according to what the underlying architecture supports.

[1]We should note that `TimeInternal` follows a different naming style than anything else shown so far; that's because we chose to keep core items of WR–PTPd unchanged while redesigning the program's architecture; the same convention applies to all data–set items, where we follow official IEEE naming.

```
struct pp_time_operations {
    int (*get)();
    int (*set)();
    int (*adjust)();
    int (*adjust_offset)();
    int (*adjust_freq)();
    int (*init_servo)();
    unsigned long (*calc_timeout)();
};
```

Listing 2. Data structure for time operations.

The servo initialization method is there mainly to ask the hardware what is the currently applied frequency correction — in `arch-unix` this method uses `adjtimex` to query the current system settings. The retrieved value is used to prime servo parameters, whereas other implementations reset the clock to "no correction" at program startup. By using the previously–learned adjustment as its own starting point, our servo can avoid learning the frequency error of the individual host it runs on, a process taking several minutes. This trick ensures a smooth behavior when the program is restarted or is taking over servo control from NTP or another PTP implementation.

Timeouts in PPSi are purely–software entities, implemented by inline functions that rely on the `calc_timeout()` method which must offer millisecond resolution and be monotonic. The `get()` method does not return monotonic time, so we need specific architecture support for timeouts.

C. Network operations

The network operations are shown in Listing 3. One such structure is instantiated for every PPSi instance: this allows an architecture to differentiate between ports offering hardware timestamps and ports without this capability. Moreover, in case of a device having more physical ports (e.g. a BC), each port may have its own *driver*, thus handling different hardware interfaces on the same device.

At initialization time, architecture–specific code is expected to initialize each PTP port using information in the instance of the `pp_network_operations` data structure. Each architecture knows whether it can support UDP or Ethernet mapping and can pre–set the default. Where the architecture supports both mappings, configuration can set up two PTP ports on a single physical interface.

The `send()` and `recv()` methods are responsible for frame I/O but are also in charge of retrieving a timestamp for the operation. The `TimeInternal` values are those used in the PTP calculations.

Finally, the `check_packet()` method is used to poll hardware for frame arrival, waiting for no more than a specified timeout.

D. PTP Profiles and Extensions

PPSi supports PTP profiles and extensions by means of protocol "hooks" and a customized state machine table, where needed ("Protocol extension" in Figure 1). The available hooks are shown in Listing 4. The implementation of the "default protocol" (Delay Request–Response Default PTP profile) calls the defined hooks at appropriate places, so the custom profile can override (or not) the default behavior while avoiding massive code duplication. A profile may want to define a customized state machine table if it requires specific steps to establish communication with a peer. For example, WR requires a sequence of specific actions to establish a WR link whenever two WRPTP daemons complete a successful Announce handshake.

The current set of hooks covers three different roles:

1) managing the extension: `init()`, `open()` and `close()`. These are concerned with initialization and clean up of any additional data structures or run time options.
2) extending the PTP protocol: from `listening()` to `handle_followup()`. These hooks can change the protocol behavior in specific places to manage specific TLV tuples or other needs, especially when transitioning from one state to another.
3) handling packets for profile–specific handshake: `pack_announce()` and `unpack_announce()`. These hooks provide and identify profile–specific TLV tuples to either activate profile–specific communication with a new peer or fall back to default PTP synchronization.

The hooks shown are designed to match the needs of White Rabbit, but if different profiles are added to the code base and new hooks are needed, the structure can be extended. This approach is a practical solution to limit code duplication and still keep the profile detached from the core PTP engine of the default protocol.

V. Supported Architectures

PPSi's main use case is WR networks and Linux host computers. The arch–independent design, though, brought us to support a number of additional architectures allowing to better help code development and testing.

The current code base includes support for the following architectures:

- `arch-unix`: the default architecture chosen for an unconfigured build. It represents a UNIX hosted environment, which we usually run on a Linux system with GNU standard libraries. It runs either as an Ordinary Clock or a Boundary Clock.
- `arch-wrs`: the "White Rabbit Switch" build. The WR switch includes special hardware for frame timestamping and phase detection, so it needs its own set of network operations. Also, the main loop of the process handles inter–process communication (IPC) with other processes that run on the WR switch. Modeling the switch as a separate architecture allows to share protocol code with the other WR architecture implementation (i.e. `arch-wrpc`, below) while exploiting the special WR switch software and hardware, and still being able to fall back on default PTP and `arch-unix` operations when running in a non–WR network. The switch is a 18–port Boundary Clock.
- `arch-wrpc`: the "White Rabbit PTP Core" [15] architecture. This is the port of PPSi that runs as an Ordinary Clock in White Rabbit I/O peripherals, e.g. [16], [17]. The run time environment of this build is a soft–core CPU running within an FPGA.
- `arch-bare-i386` and `arch-bare-x86_64`: these two architectures build Linux processes but do not rely on standard libraries: both process startup and the few required system calls are implemented in assembly language by the ports themselves. In practice, these ports allow to run freestanding versions of PPSi without the need to actually test on a microcontroller: if programmers can build and run either of the *bare* ports for their host, surely no new unexpected dependency on host features was introduced.
- `arch-sim`: this is a simulator, recently added to the project for testing purposes. When built for `arch-sim`, PPSi implements two PTP instances that communicate: one of them is a master and the other is a slave. By means of special time and network operations, the two instances exchange frames via a software–only channel and their perceived time flows at a much faster pace. This "architecture" runs the default protocol engine, and helped us tune the PPSi non–WR servo. It is explained in section VI.

We plan porting PPSi to bare–metal ARM7 and Cortex–M; an ARM7 prototype was demonstrated in 2013 [18]. A user already ported to STM32 [19].

```
struct pp_network_operations {
    int (*init)(ppi);
    int (*exit)(ppi);
    int (*recv)(ppi, frame, len, *tstamp);
    int (*send)(ppi, frame, len, *tstamp);
    int (*check_packet)();
};
```

Listing 3. Data structure for network operations.

```
struct pp_ext_hooks {
    int (*init)();
    int (*open)(ppi, run_time_options);
    int (*listening)(ppi, frame, len);
    int (*master_msg)(ppi, frame, len, mtype);
    int (*new_slave)(ppi, frame, len);
    int (*handle_resp)(ppi);
    void (*s1)(ppi, MsgHeader, MsgAnnounce);
    int (*execute_slave)(ppi);
    void (*handle_announce)(ppi);
    int (*handle_f_up)(ppi, tstamp, correction);
    int (*pack_announce)(ppi);
    void (*unpack_announce)(buf, MsgAnnounce);
};
```

Listing 4. Data structure for PTP extension hooks.

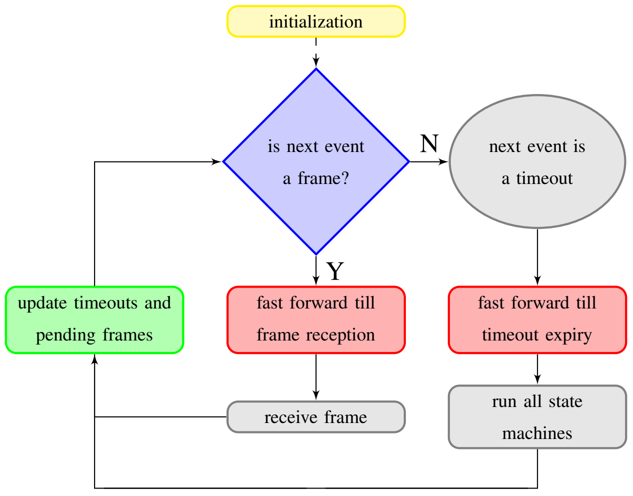
initialization
is next event a frame?
N
Y
next event is a timeout
update timeouts and pending frames
fast forward till frame reception
fast forward till timeout expiry
receive frame
run all state machines

Fig. 2. Overall functioning of the simulator.

VI. An Example: the PPSi Simulator

We now see an example of how the modularity of PPSi was used to implement a simulator (mentioned in section V) and tune the performance of a non–WR servo controller.

We designed the simulator as a new architecture (`arch-sim`) running as a Unix process and featuring its own network operations and time operations. The main program allocates two PPSi state machines (i.e. two *ppi* instances of `pp_instance`) that simulate two PTP peers. At run time, the peers act as PTP master and slave that communicate exchanging PTP frames within the simulator itself.

The PPSi core runs unmodified: only the time and frame exchange are simulated; this ensures that the convergence patterns shown by the simulator reflect how PPSi behaves in real networks.

The user can specify latency and jitter of the simulated link at run time in the configuration file; adding fault–injection is the next planned step.

PPSi calls the PTP state machine either when it receives a frame or when its timeout expires. The simulator takes advantage of this knowledge of timeouts by fast–forwarding the time of both peers whenever no frame is being delivered and both peers are waiting for a timeout. Thus, the simulator, on current PC hardware, can run one thousand complete PTP exchanges in one second of CPU time.

Figure 2 represents the simulator's loop, which is repeated for a user–defined number of iterations. If PPSi is run with proper diagnostic levels, developers can use the log files to gather useful information.

```
struct pp_sim_time_instance {
  int64_t current_ns;      // ~300 years
  int64_t freq_ppm_real;   // simulated hw err
  int64_t freq_ppm_servo;  // correction drift
};
```

Listing 5. Data structure for clock simulation.

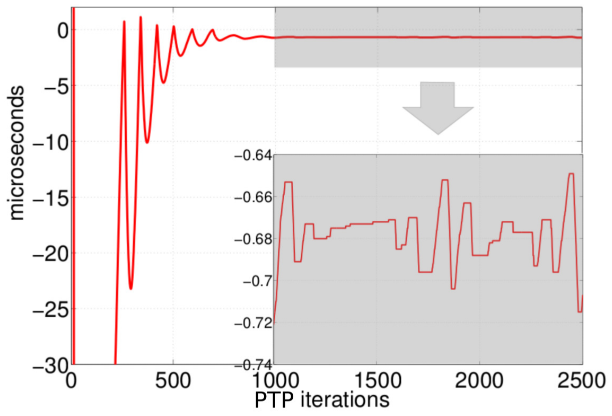

Fig. 3. The simulated synchronization performance, master–to–slave offset, before performance tuning.

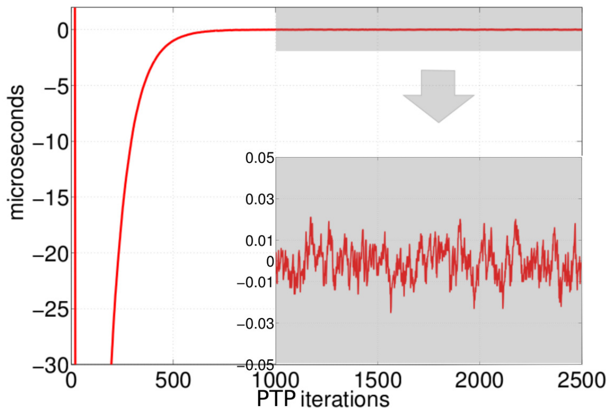

Fig. 4. The simulated synchronization performance, master–to–slave offset, after performance tuning.

The simulator architecture enabled to stress–test PPSi and isolate errors in the non–WR servo controller implementation, so we could improve its performance. The results of a simulated synchronization between two PTP nodes with an initial frequency offset of 10ppm are presented in Figure 3 and Figure 4. The simulation was configured to have an average propagation delay of 1 ms, an initial master–to–slave offset of 10 ms and introduce a random transmission jitter in the range $[-100\text{ns}, 100\text{ns}]$.

Figure 3 shows simulation performance before servo–tuning: poor convergence and constant steady–state offset can be observed. Different simulation runs showed a dependency of the time offset with respect to the initial frequency offset between master and slave. After using the simulator to identify and fix the main reasons for such performance and further tuning the controller, the results presented in Figure 4 were obtained. The synchronization accuracy improved significantly; now the main contributor to the performance is the jitter which stays within tens–of–nanoseconds.

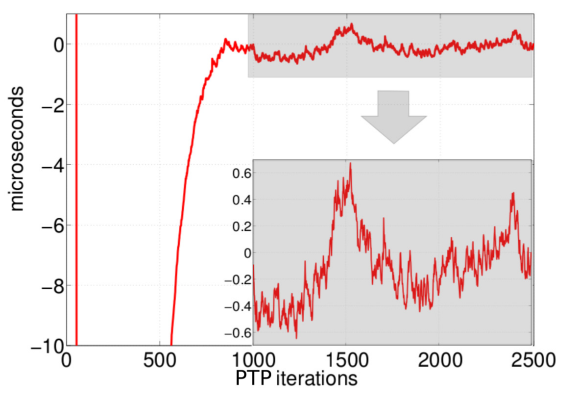

Fig. 5. Measured synchronization performance using software time stamps.

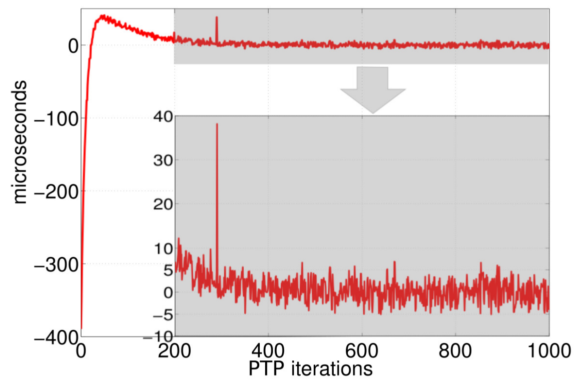

Fig. 6. Recovering from a time jump: raw data of offset values calculated by PPSi from software time stamps.

VII. Field–Test Results of the Simulation–Based Improvements

Real tests on the field confirmed the slave converged much better with these code changes, matching the data we collected in the simulations and proving the usefulness of the simulator.

As an example, Figure 5 shows the time offset of a slave PPSi instance running on a general purpose computer from a White Rabbit master; the computer uses software time stamps and had an initial offset of 1ms. The PPSi running on the WR Switch is running the default PTP profile because it identified its own peer as not WR–capable. Despite running on a conventional office network, including a general purpose Ethernet switch in the PTP path, the current PPSi servo, with the improvements we developed using the simulator, can keep the slave well within 1 microsecond from the master, symmetrically around zero.

Figure 6 shows how PPSi recovers from a jump in slave time by 500 microseconds. The Y axis represents the offset from master actually measured by PPSi in each PTP iteration, without any filtering: the figure shows all the jitter you may expect with software time stamps. During recovery we see an "outlier" event at iteration 290: the measured offset from master is quite different from the nearby ones.

PPSi considers outliers all measurements that are too different, in magnitude, from the current averaged value, but relaxes its threshold as more outliers are detected, to eventually react if a real change in network latency occurred. This outlier value is not discarded because its magnitude is not much different from the current running average, but no such events occur after the slave is synchronized to the master.

VIII. Conclusions

This paper describes the implementation of the PPSi PTP daemon. We designed PPSi as a Free Software package that supports the White Rabbit PTP Profile, while keeping it interoperable with the default profile. Our need to support White Rabbit in both Linux environments and standalone FPGA nodes led us to abstract interactions between the actual protocol and the time/network software primitives. As a result, PPSi is currently the only PTP implementation whose portability spans both hosted and freestanding environments.

A thorough review of PPSi by an external expert in early 2014 provided us with clear guidelines and enabled to set the path for further developments and improvements to the current code base, mainly in the area of management.

Overall, we think PPSi is a promising PTP implementation, able to support a wide range of devices and platforms, without penalizing performance and leaving the path open to future PTP profiles and new versions of the standard.

References

[1] *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, IEEE Std. 1588-2008, 2008.

[2] PPSi Manual. www.ohwr.org/attachments/download/1952/ppsi-manual-130311.pdf.

[3] White Rabbit. http://www.ohwr.org/projects/white-rabbit.

[4] J. Serrano, M. Cattin, E. Gousiou, E. van der Bij, T. Wostowski, G. Daniluk, and M. Lipiński, "THE WHITE RABBIT PROJECT," *Proceedings of IBIC2013*, 2013.

[5] GNU Lesser General Public License. www.gnu.org/licenses/lgpl.html.

[6] M. Lipiński, T. Włostowski, J. Serrano, and P. Alvarez, "White Rabbit: a PTP application for robust sub-nanosecond synchronization," *Proceedings of ISPCS*, 2011.

[7] M. Lipiński, J. Serrano, T. Włostowski, and C. Prados, "Reliability In A White Rabbit Network," *Proceedings of ICALEPCS*, 2011.

[8] *IEEE Standard for Information Technology–Telecommunications and Information Exchange Between Systems–Local and Metropolitan Area Networks–Specific Requirements Part 3: Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications - Section Three*, IEEE Std. 802.3-2008, 2008.

[9] *Timing characteristics of a synchronous Ethernet equipment slave clock (EEC)*, ITU-T Std. G.8262, 2007.

[10] E. Cota, M. Lipiński, T. Włostowski, E. Bij, and J. Serrano, "White Rabbit Specification: Draft for Comments," www.ohwr.org/documents/21, july 2011, v2.0.

[11] IEEE P1588 Working Group. ieee-sa.centraldesktop.com/1588public.

[12] PPSi's Public Git Repository. git://ohwr.org/white-rabbit/ppsi.git.

[13] White Rabbit PTP Proposal. www.ohwr.org/documents/92.

[14] PTPd project. www.ptpd.sourceforge.net.

[15] White Rabbit PTP Core. www.ohwr.org/projects/wr-cores/wiki/Wrpc_core.

[16] Simple PCIe FMC carrier (SPEC). www.ohwr.org/projects/spec.

[17] Simple VME FMC Carrier (SVEC). www.ohwr.org/projects/svec.

[18] A. Rubini, "PPSi," *Better Embedded Conference*, July 2013.

[19] White Rabbit Developers Mailing List. lists.ohwr.org/sympa/arc/white-rabbit-dev/2014-03/msg00015.html.