Adaptive Optics Hard and Soft Real-Time Computing Developments at ESO

Marcos Suárez Valles*[a], Nick Kornweibel[a], Enrico Marchetti[a], Poul-Henning Kamp[b], Niels Hald Pedersen[b], Robert Donaldson[a], David Barr[a], Calle Rosenquist[a]
[a]European Southern Observatory (ESO), Karl-Schwarzschild-Str. 2, 85748, Garching bei München, Germany; [b]FORCE Technology, Park Alle 345, DK-2605, Brøndy, Denmark

ABSTRACT

With the advent of the ELT First Light Instruments, the AO control problem paradigm is shifting towards networked, high-performance computing systems. This is particularly true for the auxiliary, soft real-time processing, due to the large increase in algorithm complexity and data volume, but applies partly also to hard real-time control, where a compromise may be found between the computing capability of the processing nodes and the complexity of the deterministic data distribution amongst them. These problems are not unique to AO but are present, to a variable extent, in several other Telescope subsystems.

ESO is conducting technology prototyping activities in both soft and hard real-time processing domains. The AO Real-Time Computing Toolkit standardizes soft real-time building blocks in key areas where long-term maintainability and homogeneity across instruments are deemed critical. In addition, it attempts to isolate components affected by roadmap uncertainty and/or rapid technology evolution. The intent is to streamline ELT Instruments development and operational costs by leveraging design re-usability and obsolescence management.

Along with the toolkit, the Hard Real-Time Core prototype, developed in the scope of an external contract with FORCE Technology, explores the application of mainstream CPU and networking hardware to the ELT-size MCAO, pseudo open-loop control problem under relevant performance requirements. This is framed within the development of standard, long-term support technologies for the Telescope Control System.

The motivation and current status of the above activities are presented, and the preliminary design and results from the corresponding prototypes discussed.

Keywords: Adaptive Optics, Real-Time Computer, ELT, RTC, Control Systems

1. INTRODUCTION

Drawing on the experience gained with the development and maintenance of the VLT Real-Time Computer (RTC) systems during the last decade, ESO has specified architectural requirements on the RTC of the ELT First Light Instruments. These requirements emphasize the RTC as an Instrument subsystem (deliverable by Consortia) and aim at facilitating long-term maintainability and obsolescence management, while preserving system performance and scalability. A key specification in this respect splits the RTC into two distinct components interconnected by a networked communications infrastructure –see Figure 1. Each component targets AO functions in a specific domain and computing timescale and is subject to its own technology roadmap.

The Hard Real-Time Core (HRTC) implements the main AO control loops, which perform demanding computations on every incoming Wave Front Sensor (WFS) pixel frame (up to kHz rate) and command actuators (e.g. deformable and Tip-Tilt (TT) mirrors) within tight timing constraints. To this effect, the HRTC is interfaced to the AO WFSs and actuators on board the Instrument via a dedicated AO RTC Real-Time Network. In addition, the HRTC commands both the ELT Quaternary Mirror (M4) and the ELT TT Stabilization Mirror (M5) via the ELT Deterministic Network, using the Telescope's Central Control System (CCS) as a broker.

*msuarez@eso.org; phone +49 89 3200 6699; fax +49 89 3200 6664; eso.org

The Soft Real-Time Cluster (SRTC) is an array of networked computing nodes in charge of the high-level supervision and optimization of the HRTC function. It is driven by requests from higher-level Instrument Control Software (ICS) layers, as well as by the reception and automatic processing of telemetry data (e.g. measurements, commands) from the HRTC. Computations elapse from seconds to minutes and involve complex algorithms operating on large data sets. A dedicated AO RTC Telemetry Network interconnects the HRTC and SRTC for telemetry data propagation. The injection of disturbance data into the HRTC (e.g. for calibration purposes) also makes use of this channel.

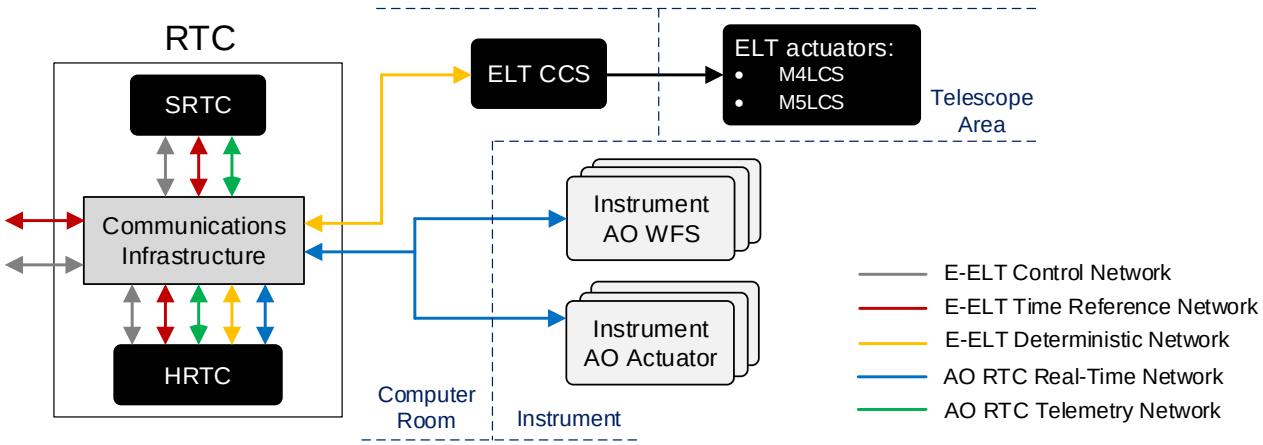

Figure 1. Main functional blocks and communication channels in the ELT RTC.

The AO auxiliary control loops, which involve commanding actuators at low-rate are (e.g. to bootstrap and maintain the optical configuration), are not implemented by the RTC. They are mapped to dedicated Instrument subsystems (typically using PLC-based technology) which operate on various parameters (including estimates computed by the SRTC).

Both the HRTC and the SRTC are connected to the standard ELT Control Network for system-wide coordination and configuration. They are synchronized with the rest of the Instrument and the Observatory via the ELT Time Reference Network, based on the Precision Time protocol (PTP[9]). In addition, the HRTC may implement internal networks for specific purposes (e.g. deterministic data propagation amongst its computing units). All the RTC interfaces are standardized on Ethernet to provide forward-compatible scalability. Depending on the application, link speeds of up to 10 Gbps are supported, with 40 Gbps currently under evaluation.

The SRTC will concentrate most of the long-term maintenance effort in the form of continuous upgrade and alignment with the evolving Observatory (hardware and software) standards. From the VLT experience, this is only sustainable if well-established technologies with a clear upgrade path are in place. Along these lines, the SRTC is standardized as a cluster of mainstream, server-class computers running the (CentOS-based) ELT Linux operating system and interconnected via Ethernet switched fabrics. The multiple CPU paradigm may, however, not suffice for all the RTC use cases. In which case, hardware acceleration (by way of GPU offload processing) is supported to a limited extent –i.e. restricted to those design patterns and hardware realizations less harmful to system maintainability.

ESO is not only standardizing the SRTC technology, but will also provide key software building blocks (i.e. the AO RTC Toolkit) in this domain. This is critical for a consistent operational scenario across instruments, ensuring that the SRTC is the primary interface between the RTC and the rest of the Instrument and Observatory.

The HRTC is historically on the cutting edge of technology at the time of its design, and is therefore subject to multiple performance vs maintainability trade-offs. This limits its upgrade path and makes it likely to undergo major changes (even re-implementation) during the Instrument's lifetime. Rather than standardizing HRTC technology, ESO will specify its interfaces to achieve full replaceability as mitigation of obsolescence. Nonetheless, prototyping activities (i.e. the HRTC Prototype) are being conducted which adapt mainstream technologies from the SRTC to the HRTC domain, aiming at improved maintainability.

2. AO RTC TOOLKIT

The AO RTC Toolkit is a suite of software tools, libraries and reference implementations supporting the development of the SRTC by the Instrument Consortia. It addresses common RTC functions, decoupling them from the particular HRTC realization. In this respect, the Toolkit is also used by the Telescope's RTC –and potentially other real-time subsystems.

Unlike the VLT case, where the pre-existing software framework was not powerful enough at the time of RTC development, the Toolkit is built on top of the ELT ICS framework and the ELT Core Integration Infrastructure (CII). These provide the required middleware, distributed ecosystem, application paradigm and basic services. Amongst the latter are Configuration, Logging and Alarm facilities, an Online Database (OLDB) for e.g. the storage of computing products, a Data Display Toolkit (DDT) for visualization purposes, support for (Qt-based) graphical user interfaces, and the supervisory control system interfaces to connect with other subsystems.

The Toolkit is delivered as an extension to the ICS framework and CII, adding AO domain-specific features and performance improvement where required. To a great extent, its product breakdown is driven by the architectural requirements specified to the ELT Instruments.

2.1 Telemetry Data Distribution Infrastructure

One of the Toolkit's central use cases is the propagation of telemetry data from the HRTC to the final applications in the SRTC. In this respect, the Toolkit standardizes the RTC telemetry data flow by defining the relevant interfaces and data paths, and providing the software components to be deployed on the SRTC computing nodes –see Figure 2.

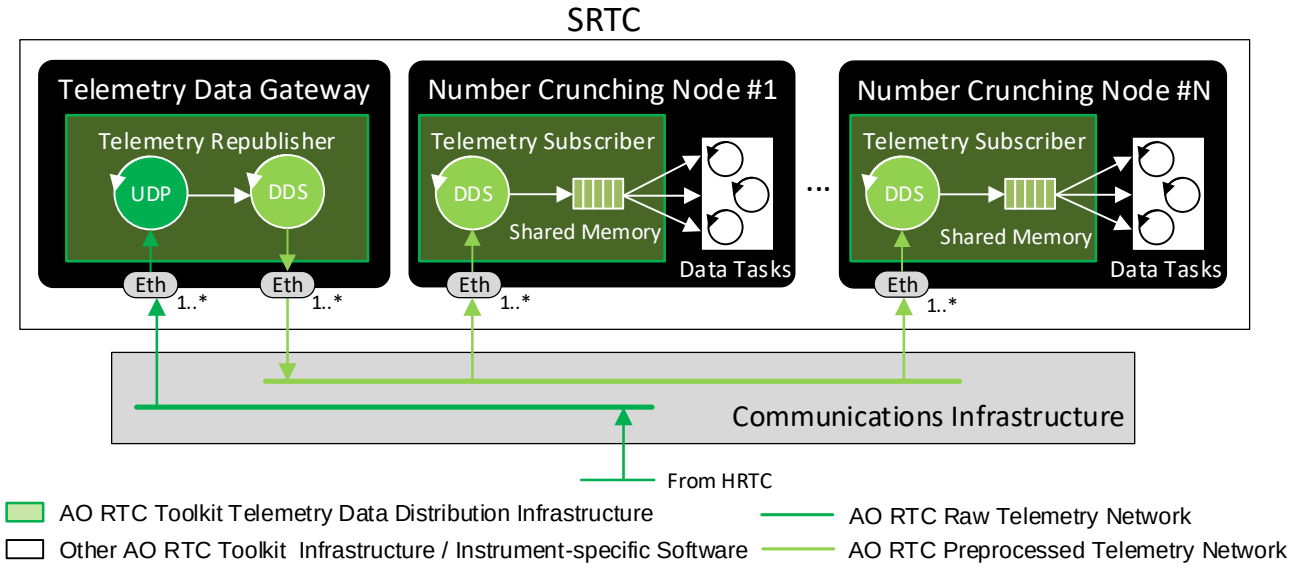

Figure 2. Standardized SRTC telemetry data flow and related computing nodes and software components.

Pixel frames are tagged with a monotonically increasing counter when transmitted by the WFS. This information is preserved throughout internal HRTC processing, to keep track of the current loop cycle and likewise associate it with intermediate computing products, including the telemetry data.

Telemetry data is organized in messages (i.e. *topics*) comprised of a standardized header and Instrument-specific payload. An instance of any given topic (i.e. a *sample*) carries data produced by the HRTC at a certain loop cycle. Several topics whose samples may be correlated via loop cycle number (i.e. originated by the same or synchronized HRTC control loops) are said to form a *data cloud*.

The HRTC publishes telemetry data as UDP topics over the AO RTC Raw Telemetry Network, without acknowledgment, nor flow control. This enables HRTC implementations with limited network stack resources –e.g. FPGA-based. In exchange, careful design and dimensioning of this network is required to ensure reliable data propagation –limited by the link's bit error rate.

At least one Telemetry Data Gateway Node in the SRTC is interfaced to the AO RTC Raw Telemetry Network. The Telemetry Republisher components therein capture UDP topics and forward them as multicast DDS[4] topics to the AO RTC Preprocessed Telemetry Network, using strict reliability. This prevents telemetry data loss within the SRTC at the expense of retransmissions.

Without reliable data delivery, the most data-intensive SRTC computations would be frequently aborted due to incomplete telemetry data sets, hindering AO loop optimization. Alternatively, the end processing nodes would need to be highly deterministic, further stressing their computing resources –already under significant load. The proposed design limits deterministic networking and real-time properties to the raw telemetry domain and gateway nodes, relaxing the constraints on the selection of the SRTC computing technology.

For scalability reasons, no correlation of topics takes places at the time of republishing. This would result in large size, aggregate DDS super-topics propagated unconditionally to all SRTC nodes, regardless of their data needs. Instead, each topic is treated as an independent HRTC logical channel (e.g. WFS#i measurements, actuator#j commands) up to the end nodes, which may subscribe to different subsets of (smaller) DDS topics each. Multiple republisher components may be present in the gateway node, each typically in charge of a data cloud –or a part of it.

The republishing process is payload-agnostic. The default implementation preserves the original UDP payload into the resulting DDS sample and enforces pre-defined fields (e.g. loop cycle, time stamp) to support later correlation. However, instrument-specific data wrangling (e.g. changes to encoding, endianness adaptation, data unscrambling, removal of data padding) is foreseen via user software extensions.

The AO RTC Preprocessed Telemetry Network interconnects the SRTC Number Crunching Nodes, in charge of the numerical processing of telemetry data. Inside them, one or more Telemetry Subscriber components collect DDS samples and make them available in shared memory, for consumption by the end Data Tasks –see Figure 3 (left).

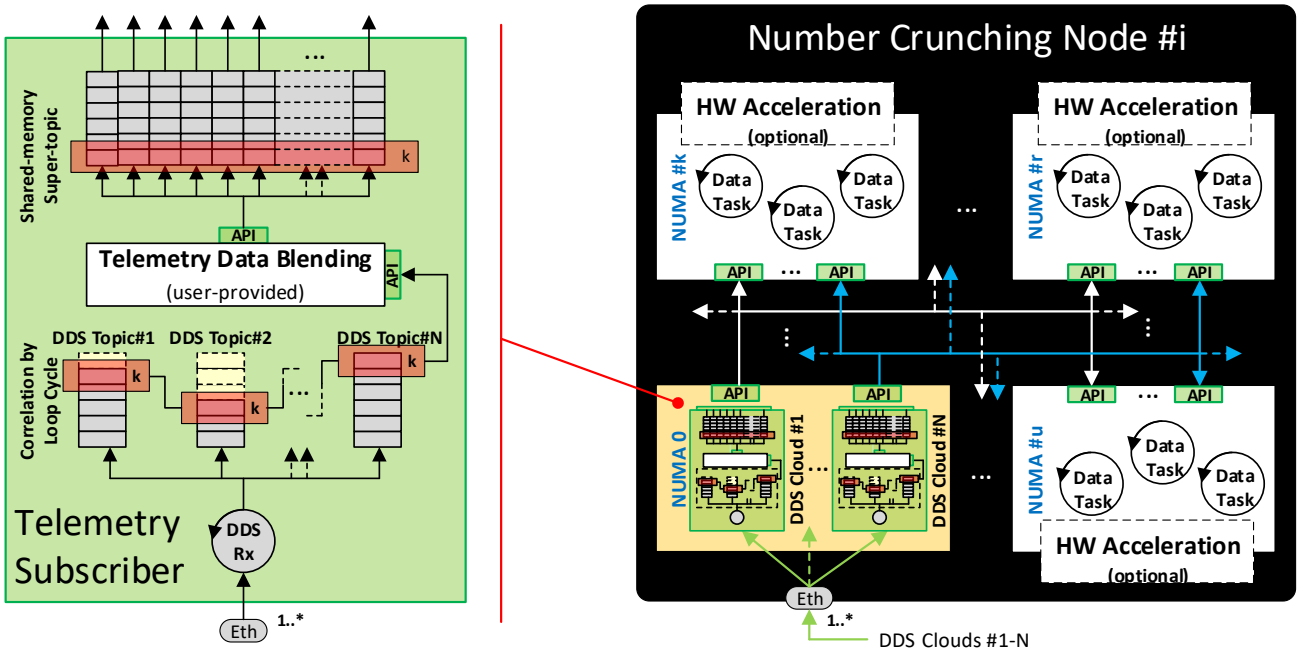

Figure 3. Telemetry subscriber (left) and deployment per node and data cloud (right).

Each subscriber operates on a subset of DDS topics belonging to the same data cloud and correlates them attending to their loop cycle. The correlated payloads are enqueued in shared memory as the individual fields of a super-topic. To allow Instrument-specific data blending (e.g. separation/merging of payloads, removal of redundant information, restriction of ranges, reordering), the actual enqueuing operation is delegated to user software extensions.

A single shared-memory queue per node serves multiple, co-located data tasks. No reliable data distribution is provided (i.e. the queue may roll over and override unread data), but a lockless read/write interface with data consistency check is

implemented instead. A data task failing to read samples fast enough (i.e. overrun by the writer) will detect this condition by itself, which confines this failure mode. In a reliable implementation, the writer would be blocked and propagate the malfunction upstream the DDS infrastructure, resulting in a (much more difficult to trace) failure at the gateway node.

The shared-memory queue is not intended for in-place data processing. A standardized API allows data tasks to extract individual samples (or groups of them) into their internal application buffers with minimum overhead, implement typical *read N – skip M* work cycles and detect data loss. To enable selective copy and Instrument-specific data blending (e.g. separation/merging of payloads, architecture-dependent memory layout) the actual copy operation is delegated to user software extensions. This facilitates offloading the computation to (optional) hardware accelerators without additional data movement –e.g. by pinning the data task buffers in memory.

The resulting DDS deployment scales with the number of SRTC nodes and data clouds (rather than with the number of data tasks) and is thus minimized –along with any retransmissions. In addition, the Toolkit provides the means to isolate the subscriber infrastructure into few cores in a NUMA node, thus reserving the rest of the computing resources for the Instrument-specific data processing –see Figure 3 (right).

2.2 Component Framework

The Toolkit extends the ICS framework with a component infrastructure which standardizes the state machine engines available to the SRTC applications (thus guaranteeing that a common, basic interface is offered to higher-level software for coordination purposes). Building on top of these engines and the CII middleware-agnostic facilities, base components are provided, along with the mechanisms to extend them. These support, amongst others, asynchronous implementation of requests and state activities (via thread pools), a common publisher-subscriber event channel (based on a standardized topic with parsable payload) and the handling of static and dynamic configuration data.

All Instrument-specific SRTC applications (e.g. supervisory as well as data tasks) are to be derived from the above base components. In addition to them, a portfolio of reusable applications is provided by the Toolkit, including e.g. data recording, disturbance injection and collection of data during an exposure. These still require customization by the user, since they operate on telemetry and/or event channel topics, whose payload is Instrument-specific.

2.3 Hardware Acceleration

The Toolkit extends the standard ELT Development Environment (based on the *Waf*[6] build automation tool) with support for the NVIDIA CUDA[7] technology. This extension enables compilation of GPU kernels using the *nvcc* version shipped with CUDA, offloading non-GPU translation units to a compatible version of the *gcc* compiler. Shared libraries are thus generated which fully encapsulate the GPU computations and provide a simple, standard C++ API to them. The final data task application is linked against these libraries, using the standard (e.g. newer) *gcc* version in the ELT Development environment, relying on the compiler's ABI forward-compatibility. This prevents CUDA dependencies (eventually frozen due to GPU obsolescence) from dictating the ELT Linux compiler version in the long-term.

In addition to native CUDA development, the use of NVIDIA libraries (where applicable) is encouraged. For legal reasons, the usage of the Linux real-time kernel in servers hosting NVIDIA GPU units is currently not supported, which may constraint the kind of operations to be carried out in these servers, as opposed to fully CPU-based hosts.

3. HRTC PROTOTYPE

The HRTC Prototype addresses the Laser Guide Star (LGS) hard real-time data paths and numerical computations of a Multi-Conjugate Adaptive Optics (MCAO) RTC using Pseudo Open-Loop Control (POLC)[1], as per the dimensions and requirements in Table 1.

Table 1. HRTC Prototype major dimensions and requirements.

6 x LGS Shack-Hartmann WFS	**Control algorithm**
800x800 pixels, 80x80 subapertures	Pseudo Open-Loop Control (POLC)
4,616 used subapertures	Correlation matched filter
500 Hz frame rate (goal:700 Hz)	Full MVM reconstruction (32-bit floating-point)
Rolling Shutter readout	Individual IIR filtering per mode
3 x DM	**Performance requirements**
M4+M5: 5,316 modes	RTC end-to-end latency < 1.6ms
INS-DM1: 500 modes	RTC control jitter < 100 µs std. dev. (goal: 40 µs)
INS-DM2: 500 modes	POLC matrices update period < 30 s

The intent is to validate mainstream CPU and Ethernet technologies for the most demanding (realistic) HRTC use cases, prioritizing long-term maintainability and ease of development over system compactness and performance per watt considerations. A full-scale prototype is pursued which implements an unabridged network infrastructure, with data propagation and computations following the exact time pattern expected for the ELT systems. To this end, simulated WFS and actuators have been developed by ESO which mimic the timing of the actual devices to µs precision.

The prototype avoids major algorithmic simplifications not yet sufficiently proven on sky for the ELT size. It therefore employs single-precision arithmetic, computes explicitly the incoming wavefront in slope space (thus requiring two full matrix-vector-multiplications (MVM) per loop cycle) and exploits no matrix sparsity. Compared to less conservative implementations, this strategy can easily accommodate involved (e.g. non-linear) WFS modelling in the future.

The RTC latency specification is compatible with ~5 ms overall AO loop delay at 500 Hz frame rate (~4 ms at 700 Hz) when the actual dynamic response of the ELT LGS WFS and M4 (plus the pure CCS internal delays) are considered.

3.1 Hardware Platform

The two MVM operations computed by POLC at 500 Hz (spanning all WFS and actuators) require a minimum memory bandwidth of 1.4 TB/s. Not relying on cache locality, this dictates a server architecture with multiple memory buses. The AMD EPYC family has been chosen as the best compromise between memory bandwidth and cost –see Table 2.

Table 2. Technical specifications of the two types of HRTC Prototype servers.

Big Box (Dell R7425)	**Small Box (Dell R6415)**
Dual-socket AMD EPYC 7501 (Naples)	Single-socket AMD EPYC 7251 (Naples)
4 silicon dies per socket	4 silicon dies per socket
2 DDR-2666 memory buses per die	2 DDR-2400 memory buses per die
2 core complexes (CCX) per die	2 core complexes (CCX) per die
4 Zen cores + 8 MiB L3 cache per CCX	1 Zen core + 4 MiB L3 cache per CCX
16 DDR-2666 memory buses per server	8 DDR-2400 memory buses per server
64 Zen cores + 128 MiB L3 cache per server	8 Zen cores + 32 MiB L3 per server

A Dell R7425 (hereafter *Big Box*) theoretically delivers in excess of 340 GB/s and is used for memory-bound computations. A less powerful Dell R6415 (hereafter *Small Box*) performs less demanding and/or CPU-bound tasks. Configured for memory channel interleaving[2], each EPYC CPU package exposes its four dies as separate NUMA nodes.

Standard 10 GbE networking infrastructure has been adopted for the prototype. This is compatible with the ~4 Gbps aggregate telemetry data throughput estimated for the various HRTC sources. It also accommodates easily the ~1 Gbps bandwidth required to download all POLC matrices (~2.6 GiB) from a single port (worst-case) every 30 s. The expected network serialization delays are small compared to the RTC latency requirement and do not immediately justify higher speeds. Along the same lines, optimization via cut-through switching has not been pursued.

3.2 Software Platform

The Linux CentOS 7.5 distribution is used, as a compromise between proper AMD EPYC architecture support by the system compiler (gcc 4.8.5) and easy later replacement with the upcoming release of ELT Linux (based on CentOS 8).

The HRTC Prototype servers are configured as diskless computers booting from the network via PXE[8]. The boot process employs a RAM disk hosting a standard (non-real-time) Linux kernel 3.10 image and a custom *init* script, which loads the drivers for the network cards in use and spawns a housekeeping application –i.e. *domus*. This approach drops the default *systemd* process on behalf of tighter hardware and software configuration control. The result is a single-user (i.e. *root*) run-level where only those devices essential for HRTC operation are enabled and no system services run, which limits the sources of non-deterministic behavior (e.g. interrupts, re-scheduling).

Furthermore, the lack of a Linux userland dramatically reduces software dependencies, thus aiding maintainability. Any Linux kernel with support for the essential devices will successfully boot in the HRTC servers. Conversely, replacing a hardware device with a new model (e.g. due to obsolesce) comes down to finding a compatible driver for the latter.

The *domus* housekeeping process in each server provides a command-line interface, through which the binary application specific to the server's role is deployed and its lifecycle controlled.

3.3 Implementation

The HRTC Prototype is interfaced to six simulated WFS and four simulated actuators via the AO RTC Real-Time Network. Each of these external connections employs a dedicated 10 Gbps physical link between the corresponding device and a network switch. A cluster of servers interconnected by internal networks (realized by the switch using the same technology) implement the HRTC computing pipeline –see Figure 4.

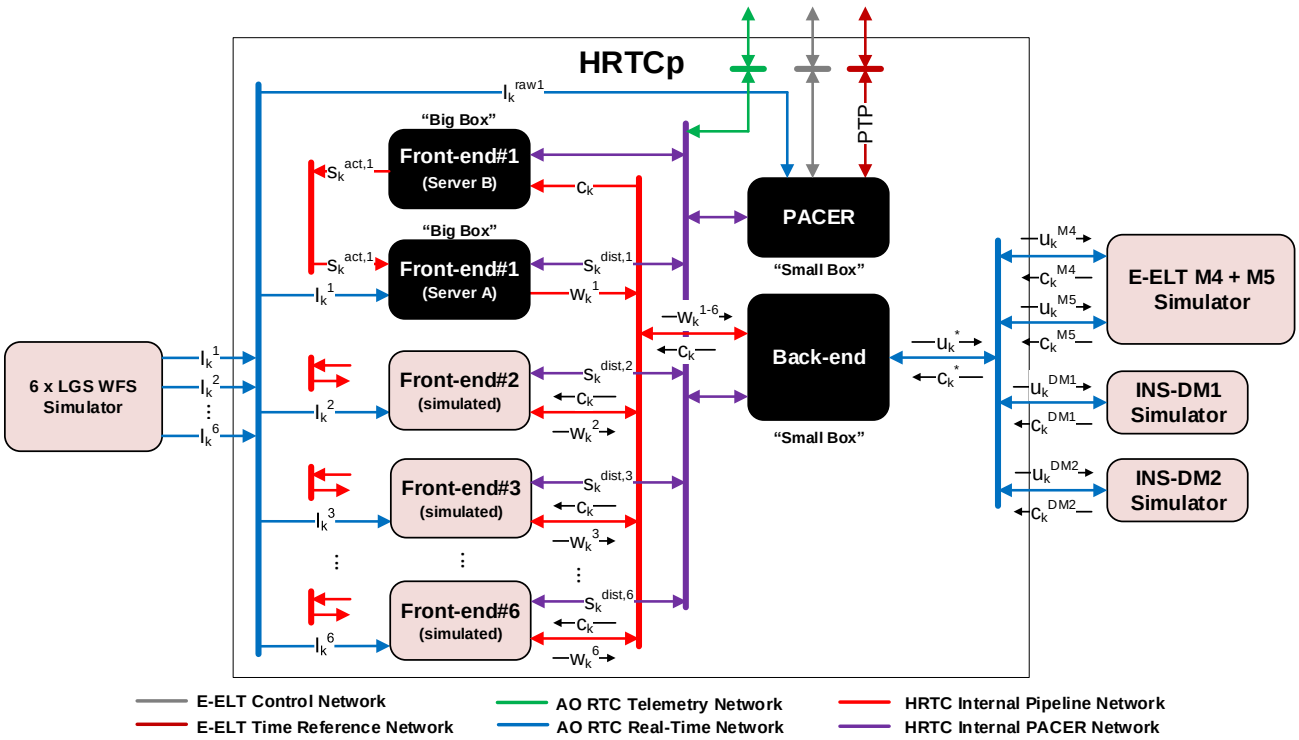

Figure 4. HRTC Prototype topology and main data paths. The thin connecting lines indicate individual network links, whereas the thick ones represent aggregate data paths internal to the network switch.

Real-time computation results are shared amongst the servers via an HRTC Internal Pipeline Network. Time synchronization, telemetry, configuration, disturbance injection and coordination make use of a separate HRTC PACER Network, further described in section 3.4. These networks are internally arranged in subnets –not shown in Figure 4.

Each WFS is processed by a dedicated Front-end stage (see Figure 5, left) comprised of two Big Box instances (i.e. Servers A and B). Server A computes the incoming, open-loop wavefront estimate w_k^i from the pixel frame I_k in parallel with WFS readout, via a Correlation Matched Filter followed by an MVM Tomographic Projection. This process requires the slopes $s_k^{act,i}$ induced by past actuations c_k, which are computed by Server B in advance of loop cycle k (via MVM back-projection) and forwarded to Server A. There, they are added to the residual slopes $s_k^{res,i}$ to form the pseudo open-loop (POL) slopes $s_k^{pol,i}$. If slope disturbance $s_k^{dist,i}$ is available for loop cycle k, it is also added to the result.

The open-loop wavefront estimates $w_k^{1\text{-}6}$ are received by a single Back-end stage (see Figure 5, right), implemented in terms of a Small Box. They are linearly combined and subtracted from the last wavefront realized by the actuators to estimate the wavefront residual error e_k. This is further processed by an array of temporal IIR filters (one per controlled mode) to produce a corrective modal command u_k, which is further split into the individual actuator commands u_k^{M4}, u_k^{M5}, u_k^{DM1} and u_k^{DM2}. Any disturbance terms $u_k^{dist,M4}$, $u_k^{dist,M5}$, $u_k^{dist,DM1}$ or $u_k^{dist,DM2}$ applicable for loop cycle k are added to them. The echo from the actuators c_k^{M4}, c_k^{M5}, c_k^{DM1} and c_k^{DM2}, containing the commands actually realized, are processed offline to be used during the error estimation of loop cycle $k+1$.

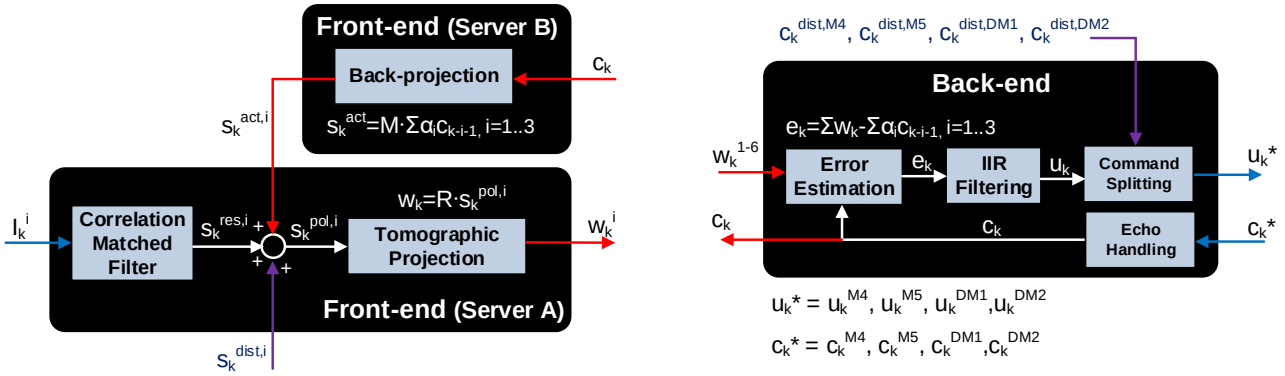

Figure 5. Detail of the HRTC prototype Front-end (left) and Back-end (right) computations. Vectors $s_k^{res,i}$, $s_k^{act,i}$, $s_k^{dist,i}$ and s_k^{pol} have size 9,332, whereas the dimension of vectors w_k^i, c_k, u_k and e_k is 6,316.

To limit the cost of the prototype, only one Front-end instance and the Back-end are fully realized. The remaining Front-end units are simulated using a single, less performant server each. These execute no numerical computations but transmit pre-defined results, consume configuration and disturbance data, etc. following the real application timing. Since a fully-dimensioned network infrastructure is provided, the simulated units can be replaced with actual ones with minimum impact –e.g. to test side-by-side implementations based on different CPU technologies.

Each server executes a single-process application (launched via *domus*), which in turn spawns all the required computing threads, together with its own command-line interface for coordination by the supervisory software in the SRTC. Sharing the virtual memory amongst the various tasks allows efficient data access and synchronization. Determinism is further achieved by isolation of CPU cores, pinning threads to them via affinity settings and exploiting NUMA locality.

As an example, most of the resources in Front-end Server A are isolated from the operating system, which is confined to one core in NUMA node 0. A thread in this node receives WFS data packets and forwards the pixels to node 1, where separate threads implement pixel calibration and POL slope computation, proceeding incrementally as data completion rate (e.g. subaperture) allows. The MVM projection (size 6,316x9,232) is implemented concurrently by 48 threads, which fully span nodes 2 to 7. Slope values are copied to these nodes and the computation triggered on individual blocks of optimum size as they become available. Upon MVM completion, the results are copied back to node 0 for transmission. All threads are pinned to pre-defined CPU cores and spin on counters indicating the completion level of their input data stream. Additional threads in the same node are dedicated to coordination, configuration and telemetry.

A similar MVM setup in Front-end Server B is not limited by the incoming data rate (the back-projection can be started as soon as the actuator echoes are received) and may therefore proceed faster by using additional NUMA nodes and/or relying on L3 cache optimization techniques. E.g. the traversing of the matrix may be alternated between top-to-bottom and bottom-to-top every other loop cycle, such that a fraction of the computation always operates from cache.

A full implementation of the Back-end computation is not yet available. As of today, this server is exercised with the actual network data flow and timing, but dummy computations. Its final workload is not expected to be limited by memory bandwidth, but actually I/O bound.

3.4 Time Driven Operation

A special server (i.e. PACER) in the HRTC Prototype is interfaced to the ELT Control Network, ELT Time Reference Network and ELT Deterministic Network and subtends the HRTC Internal PACER Network, which provides the equivalent services over a time-sliced communication channel –see Figure 3.

The PACER server has its own clock controlled via PTP and distributes the time reference to the rest of HRTC servers using a custom protocol. It interfaces to the AO RTC Real-Time Network and keeps track of the start of every loop cycle using a PLL, which locks on the WFS packet train. It determines the reference time for the current loop cycle and distributes it via a *ticktock* packet to the rest of servers, which use it to synchronize their own clocks. No PTP service is run by these servers, thus removing a potential source of unpredictable jitter.

In addition, PACER acts as sequencing node, (optionally) restricting non real-time operations (e.g. configuration updates) to a pre-defined maintenance window within the loop cycle, in order to further limit computing jitter. Owning the time reference for the loop cycle, PACER can schedule operations in the HRTC servers with sub-cycle resolution.

PACER transmits to each HRTC server every loop cycle a fixed train of packets carrying disturbance and configuration data, with the content of the configuration packets dynamically planned based on priority queues. Configuration data are retrieved by PACER from TCP endpoints and the highest priority one at any time scheduled for transmission over as many contiguous loop cycles as required. This maximizes network bandwidth utilization while avoiding transmission bursts. All packets are always present in the train (with dummy content when unused), to reduce jitter.

Sometime within each loop cycle, the HRTC servers propagate back to PACER a *tocktick* packet, containing status and diagnostic information. This is used for situational awareness and possibly self-tuning of timing parameters –e.g. the transmission window for the fixed packet train. As long as the diagnostics include internal pipeline timestamps, the supervisory software in the SRTC can (non-invasively) monitor HRTC performance through PACER.

Since an absolute time reference is available, it is possible to trade HRTC latency in exchange for reduced control jitter, by commanding the actuators at pre-defined instants with respect to the start of each loop cycle. If the time to command is chosen to match the upper limit of the measured HRTC processing time, the computing and control jitter get decoupled, the latter mainly affected by jitter sources in the network stack and physical layer. Note that the PLL in PACER may be used to stabilize lowly-deterministic WFS signals, thus decoupling as well the measurement jitter.

4. RESULTS

4.1 SRTC Prototyping

Various throughput, computational load and scalability tests have been conducted using a prototype of the Telemetry Data Distribution Infrastructure described in section 2.1. These tests involve a Dell R730 server (Intel Xeon E5-2680 v4, 128 GB DDR4) configured as a raw telemetry data source. By means of a DPDK[5]-based application, UDP telemetry topics of ~55 kiB (each consisting of seven UDP packets) are published at 700 Hz loop rate simultaneously over seven dedicated network interfaces, according to a (configurable) deterministic time sequence. An identical server (not using DPDK) incarnates the Telemetry Data Gateway and can be interfaced to the Raw and Pre-processed Telemetry Networks using a variable number of network interfaces, depending on the specific test.

As part of the scalability tests, the gateway node was configured to receive the UDP topics from the seven sources on one dedicated network interface. The retransmission of the DDS topics was done on a single, separate interface. Four additional servers of different types were used as Number Crunching Nodes. In each of them, a Telemetry Subscriber component received the seven DDS topics and copied them to shared memory. Several affinity settings were exercised, trying to identify the minimum resources required per subscriber. With the application confined to one (isolated) CPU core and the interrupts from the receive interface also routed to it, a single-core occupancy of ~40% was observed together with a rate of DDS negative acknowledgements below 0.06%.

Given the use of multicast and the low incidence of DDS retransmissions, the computational effort on the gateway node remained around ~3% (of a single core) throughout the various tests, independently of the number of crunching nodes.

4.2 HRTC Prototyping

Figure 6 (left) shows the results of inter-core communication tests carried out on the AMD EPYC 7501. The basic test runs two threads in two different CPU cores which write to each other's memory in a *ping-pong* fashion, while time stamping their actions using the local CPU time stamp counter (via the assembly instruction *rtdsc*). This creates two tables of interleaved time stamps, from which any offset between both cores can be calibrated out. As a side effect, the attainable *ping-pong* frequency is related to the memory latency between the two cores. By deploying the test on all possible CPU core combinations, Figure 6 (left) is formed. The worst-case latency corresponds to cores in different CPU sockets and is only slightly above 1 µs.

To further validate the thread synchronization strategy implemented by the Front-end Server A (see section 3.3), a test was conducted in which a supervisory thread in NUMA node 0 measures the time required to wake up 56 worker threads pinned to each CPU core in nodes 1-7. The (eight) worker threads deployed in the same node spin on a shared sentinel

flag (local to it). In each iteration, the supervisory thread takes a timestamp and writes to the seven remote sentinel flags. Each worker thread time stamps the detection of the write and acknowledges it via a dedicated flag, so that the supervisory thread can wait for all of them and restart the cycle. Figure 6 (right) shows the signaling time measured for different NUMA distances. The worst case results (i.e. 1.5 µs) are well in-line with those from the above test.

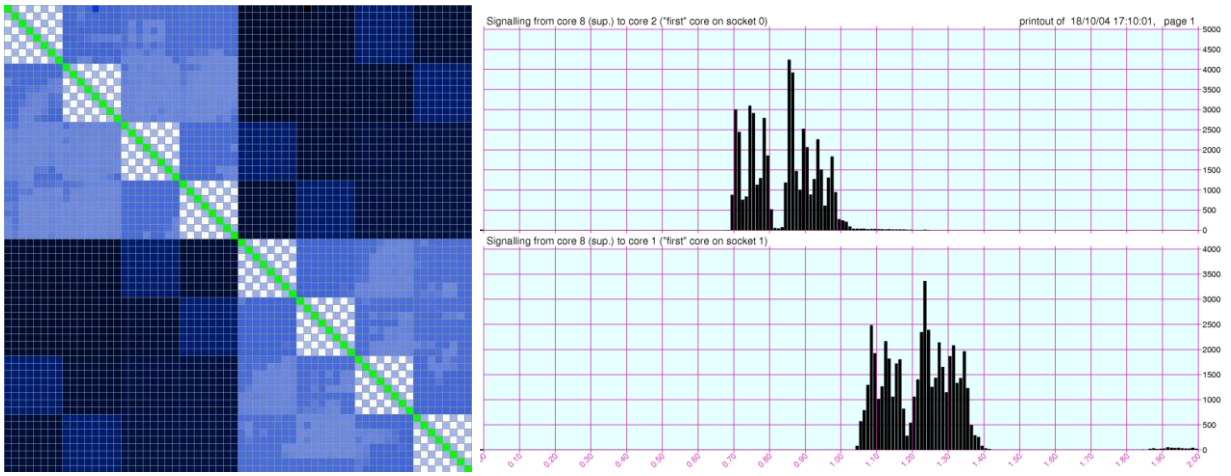

Figure 6. AMD EPYC 7501 inter-core latency map (left) and time to signal 56 threads in remote NUMA nodes (right). Left: green (same core, latency not defined), white (80 ns), medium blue (425 ns), dark blue (1,150 ns). Right, upper graph: signaling between cores in the same NUMA node. Right bottom graph: signaling between cores in different NUMA nodes. The CPU core numbering for the left and right figures follows the AMD and Dell schemes[3] respectively.

Figure 7 shows a time sequence diagram of the main computing steps in the Front-end Server A, as per section 3.3. The various tasks progress concurrently and are complete within less than 100 µs from the reception of the last WFS network packet. The implementation under test includes so far neither the reception of the actuator slopes term from Server B, nor the transmission of results to the Back-end server. Telemetry data is also not being published. Still, the test involved the continuous update of the reconstruction matrix (~222 MiB) every ~1 s in parallel with the computations (i.e. no maintenance window was enforced) to maximize potential jitter contributions.

Despite the foreseeable subsequent networking delays (e.g. transmission of the result, serialization with those from the rest of units at the network switch) and Back-end computing time, the end-to-end HRTC latency is expected to be well below the specification in Table 1. This provides significant margin for minimizing jitter at the expense of latency see – section 3.4.

Tests have been conducted having a thread in a Small Box server periodically send fixed trains of network packets with configurable inter-packet delay. The thread spins on the system clock and transmits packets at pre-computed times, including these in the packets as a timestamp. Packets are collected by a thread in another Small Box server, the timestamp extracted, compared with the time of reception and statistics computed. The transmit are received threads are isolated in dedicated cores and the operating system kernel threads removed from them.

Figure 8 (left) shows the observed time interval between the start of successive trains of packet under nominal operating conditions (i.e. the test setup used 8 kB packets with fixed inter-packet delay, repeating the transmission every 2 ms). The observed standard deviation is 0.66 µs, proving that the transmission time is controllable within the 1 µs range. A single packet during the test experienced a delay of approximately 13.7 µs, which pushes the two inter-packet intervals surrounding it to plus and minus 13.7 µs respectively, as seen in the magnified plot in Figure 8 (right).

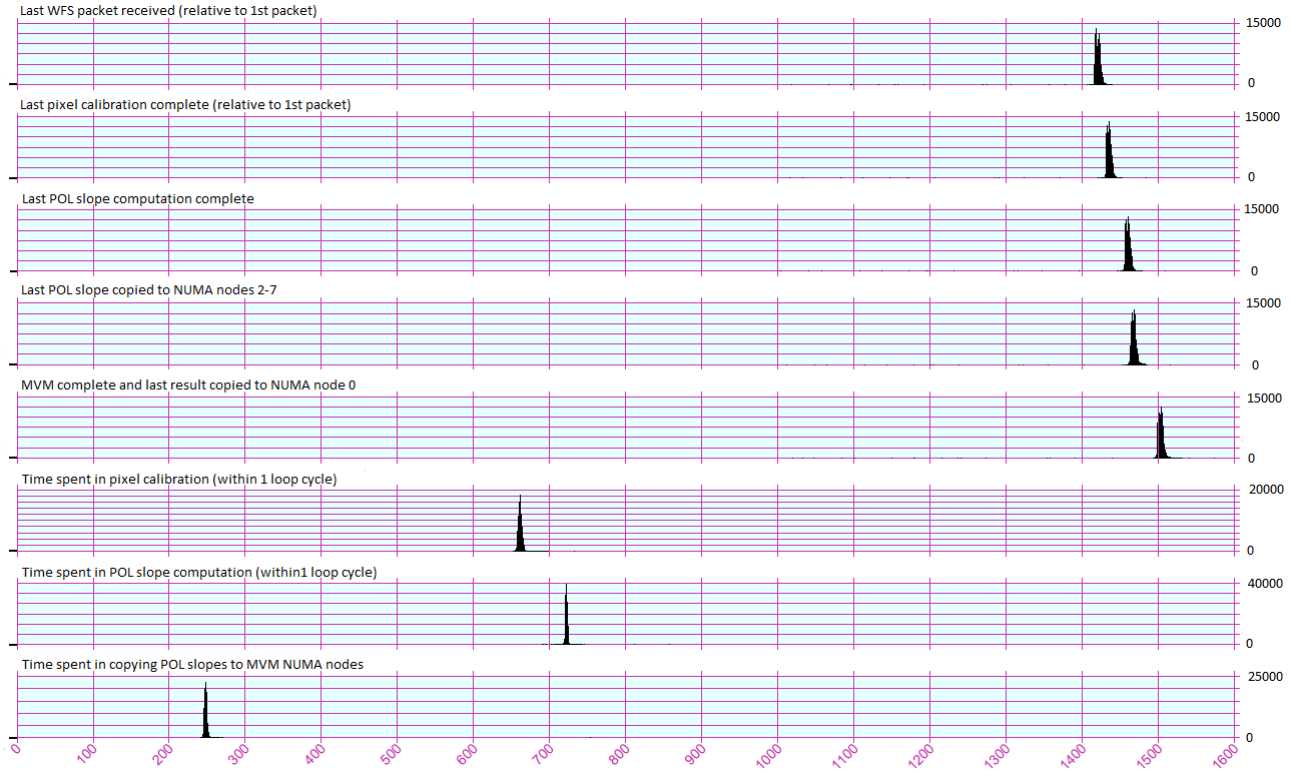

Figure 7. Timing results from the main computations in Front-end Server A. Upper five graphs: histograms of the observed duration of the various Front-end computing steps wrt. the first WFS packet received. Bottom three graphs: histograms of the cumulative time spent in the execution of key computing steps within one loop cycle.

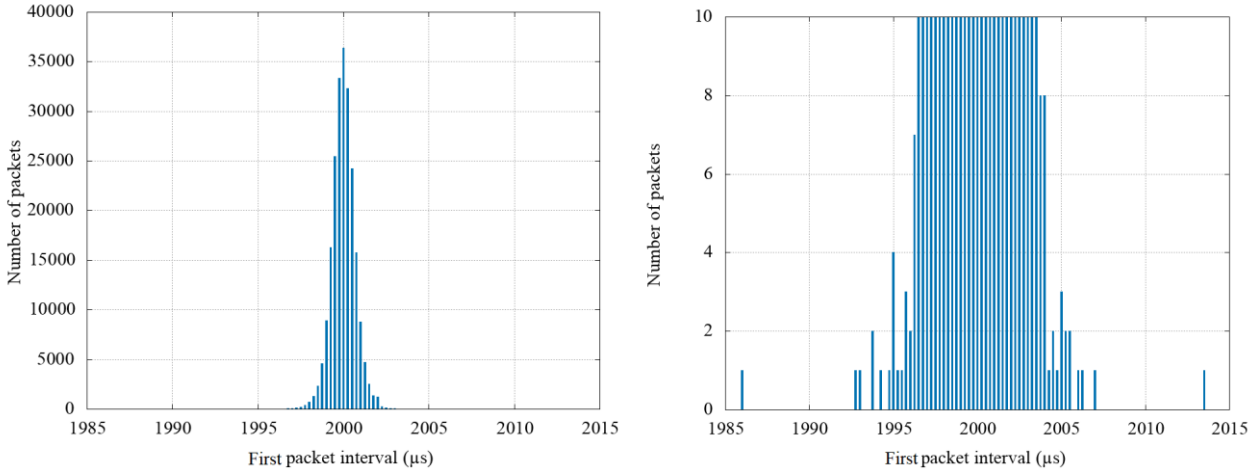

Figure 8. Results from packet transmission test. Left: histogram of the observed interval between the first packet of successive packet trains transmitted with 2 ms period. Right: magnification of the above.

5. CONCLUSIONS

The SRTC prototyping phase at ESO is being concluded to move forward into the construction of the final product. Scalability has been demonstrated for a reference design, where one or more gateway nodes propagate telemetry data reliably to a potentially large number of number crunching nodes. The infrastructure to be deployed in the latter for data

reception and shared memory distribution can be easily confined to few CPU cores, leaving the rest of computing resources to Instrument-specific data tasks, potentially using hardware acceleration. GPUs can be supported in this respect via development environment extensions without constraining the ELT Linux distribution in the long term.

Work on the HRTC Prototype is still required to produce a full implementation –currently planned for the end of 2019. The results so far indicate that CPU technology can deliver latency in the low hundreds of µs for an ELT-size MCAO system, whilst limiting control jitter to few µs standard deviation. This is achieved isolating computing resources and pinning to them tasks which run in non-blocking fashion. Unpredictable sources of non-determinism are removed by having the (diskless) servers boot from the network with a minimum set of drivers and no userland. Tight synchronization amongst these servers allows time-slicing the work cycle to minimize crosstalk between the various tasks. Commanding the actuators within one such pre-defined time-slice prevents stringent determinism specifications from being artificially propagated to every other server. Very little performance is entrusted to the operating system and tools (e.g. neither preemption patch, nor network stack bypassing are used).

REFERENCES

[1] Gilles, L., "Closed-loop stability and performance analysis of least-squares and minimum-variance control algorithms for multiconjugate adaptive optics", Applied Optics, Vol. 44, Issue 6, 993-1002 (2005)

[2] [NUMA Topology for AMD EPYC Naples Family Processors], Advanced Micro Devices (AMD), publication #56308, rev. 0.70, May 2018.

[3] [HPC Tuning Guide for AMD EPYC Processors], Advanced Micro Devices (AMD), publication #56420, rev. 0.7, December 2018.

[4] Data Distribution Service (DDS), https://www.dds-foundation.org/

[5] Data Plane Development Kit (DPDK), https://www.dpdk.org/

[6] Nagy, T. "The Waf book", https://waf.io/

[7] Compute Unified Device Architecture (CUDA), https://developer.nvidia.com/cuda-zone

[8] [Preboot Execution Environment (PXE) Specification], Intel Corporation, Version 2.1, September 20, 1999

[9] [IEEE 1588-2008 – IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems], https://standards.ieee.org/standard/1588-2008.html