Evaluation of a Hypervisor-Based Smart Controller for Industry 4.0 Functions in Manufacturing

Florian Schade*, David Barton†, Jürgen Fleischer† and Jürgen Becker*
*Institute for Information Processing Technologies
†wbk Institute of Production Science
Karlsruhe Institute of Technology, Karlsruhe, Germany
Email: {florian.schade, david.barton, juergen.fleischer, becker}@kit.edu

***Abstract*—In machine tools, predictive maintenance and process monitoring can help optimize the efficiency of production systems by increasing machine availability, product quality, and by early fault detection. When integrating new sensors to provide data for such Industry 4.0 functions, the required interfaces and processing capacity are not always available, especially when upgrading existing production systems. An approach to face these issues is the smart controller architecture for sensor integration and data pre-processing. It uses a hypervisor to enable both fast reaction to incoming data and the flexibility and driver support of a Linux environment on a single platform. In this work, we implement and evaluate this concept on a commercial off-the-shelf (COTS) single-board computer in a real-world application. We show how the platform can be used for the integration of a state-of-the-art spindle monitoring sensor with an industrial edge device. Finally, we investigate the achievable end-to-end latency using the smart controller concept in comparison to a native Linux setup both in general and with respect to the application.**

***Index Terms*—edge computing, hypervisor, industrial control, intelligent manufacturing systems, machine tools, monitoring, predictive maintenance**

I. Introduction

The fourth industrial revolution (Industry 4.0) and the smart factory vision aim at enhancing automation, communication and monitoring of production systems to increase their efficiency and flexibility. Common approaches to enhance efficiency include predictive maintenance and process monitoring. Predictive maintenance is characterized by long-term monitoring of sensor data and the detection of patterns indicating the need for maintenance, for example by machine learning methods. While not being time critical, complex data processing software profits from the management features, drivers, and modularity of general-purpose operating systems (GPOS) such as Linux. Process monitoring, on the other hand, is the online monitoring of the machining process. While the computation often is less complex, a low latency between sensor data capture and the monitoring system response is highly relevant to enable functionality such as online process optimization or failure detection and emergency shutdown. When integrating the required sensors into new (greenfield) or existing (brownfield) production systems, the required hardware and software interfaces may not always be available.

To allow for low-effort adaption of interfaces, data preprocessing, and for the integration of low-latency monitoring functionality, a modular smart controller architecture for data communication and computing has been proposed in [1].

In this work, we present how the smart controller concept can be used on a commercial off-the-shelf single-board computer for sensor integration in an industrial use case. We give insight into the system and controller software architecture and determine the achievable latencies both in the use case and in a generic case. We evaluate *end-to-end latencies*, i.e. the delay between an external input to the controller and the corresponding output signal, thereby allowing for an estimate of the performance in real-world applications.

II. Related work

Many researchers in the field of production science have studied the integration of sensors into machine elements for predictive maintenance and process monitoring, thus sensing the condition of the machine and characteristics of the machining process. Recent examples include the use of an integrated camera to determine the condition of ball screw drives by Schlagenhauf et al. [2], the estimation of surface quality based on acceleration measurements in a milling tool by Möhring et al. [3], and the monitoring of tool wear using a combination of vibration, cutting force, and power data by Zhang et al. [4].

In machine tools and other equipment for discrete manufacturing, the full potential of Industry 4.0 is far from being achieved, as corresponding functions are often only found in pilot projects and isolated systems [5]. Therefore, scalability, ease of deployment, and reconfigurability are major challenges on the path towards a wide application of digitization in industrial manufacturing, especially in the context of retrofitting new functions to existing machines and within brownfield factories. These challenges can be mitigated by designing suitable architectures for data acquisition, communication, and processing. The data processing required for Industry 4.0 functions may be performed on an edge device [6], rely on cloud-based computing, or be integrated in a seamless architecture combining several computing domains as proposed by [7]. The choice depends on requirements including IT security, protection of data and intellectual property (IP), data volume and bandwidth, computing power, as well as latency. Barton et al. [8] designed a retrofitting kit for Industry 4.0 functions, in which the additional monitoring functions are installed as applications on a separate edge computer. These applications

access data such as positions and motor currents from the CNC unit (computer numerical control) via a local Ethernet network in order to derive insights into the machining process and display these in a web-based dashboard. Applications can be installed and configured remotely thanks to a cloud-based application management.

However, the most time critical functions, for example an emergency stop on detecting a fault, are not covered by the architectures described above. These require a dedicated solution close to the field level of the Industry 4.0 architecture. To allow for the integration of low-latency data processing with a full-featured Linux OS environment on a single CPU, a smart controller software architecture is proposed in [1]. It is based on the Jailhouse hypervisor [9] to run both a Linux and a real-time operating system (RTOS) in parallel and to isolate them from each other. The proposed architecture is used and evaluated in this work.

Previous performance evaluations involving hypervisors on the Banana Pi were done by Toumassian et al. [10] and Danielsson et al. [11]. Toumassian et al. evaluated the performance overhead of hypervisors compared to a native Linux setup. In their evaluation, the Jailhouse hypervisor shows the lowest overhead at $0.04\,\%$ while Xen hypervisor setups show significantly higher values ($21.6\,\%$ and $7.4\,\%$, depending on the scheduler). Danielsson et al. proposed a methodology for quantifying the performance isolation between processor cores. They use it to compare a Jailhouse-partitioned setup to a Linux setup. Both investigate the overhead and isolation, respectively, of hypervisors using Linux guests only and did not consider mixed OS systems. In contrast, our work evaluates the achievable latencies of a hypervisor-based mixed-OS setup and compares them to a native Linux setup. Also, this work is based on an industrial use case while the referenced works use artificial benchmark applications.

While the smart controller concept proposes static separation of RTOS and GPOS, there are other approaches focusing more on the dynamic and tight integration of real-time tasks and best-effort Linux tasks. The Preempt-RT Linux patches aim at making the Linux kernel itself real-time capable [12], other approaches put all Linux threads under the control of a supervising microkernel and thereby avoid extensive modification of the Linux kernel itself. Examples are the RTAI [13] and Xenomai [14] projects. An exemplary evaluation of such a microkernel approach is presented in [15] where the interrupt latency of an L4-microkernel-based system is shown. In these cases, schedulers control the mixed execution of real-time and best-effort tasks. Since we believe that a latency comparison between static and dynamic approaches yields interesting results, we evaluate the smart controller concept against a Preempt-RT approach in Section IV-C.

III. Use case and proposed architecture

A. Industry 4.0 Testbed

Our approach is demonstrated on a Deckel Maho DMC 60H, a machining center used for milling and drilling operations in the manufacturing of metal parts, for example in the machine building industry. Electric drives power the feed axes responsible for moving the parts being machined and the cutting tool. These drives and the necessary peripheral equipment (e.g. tool change, clamping, coolant) are controlled by a CNC unit, in this case a Siemens Sinumerik 840D sl. The retrofitting kit for Industry 4.0, described briefly in Section II and in more detail by Barton et al. [8], was implemented on this machine. A Sinumerik Edge is used to run the retrofitted local monitoring applications.

B. Retrofitting Industry 4.0 Functionality

To retrofit an overload detection and predictive maintenance functionality to the main spindle of the machining center, a new sensor shall be integrated, measuring spindle displacement and tilt. This data can be used to derive the force affecting the tool attached to the spindle.

The overload detection mechanism is intended to protect the machine, especially the main spindle, and the workpiece from damage. Overload may be caused by unintended contact with obstacles (i.e. collision), unfavorable process parameters, or previous damage to the tool. These conditions occur especially frequently in the manufacturing of single parts or small series, due to errors when programming tool paths or physically setting up the machine (manual positioning of fixtures and workpiece). Overload is characterized by the force affecting the tool exceeding a defined threshold. Since this force is related to the spindle displacement, for this evaluation this can be reduced to a threshold comparison of the displacement values. When an overload situation is detected, a machine emergency stop shall be triggered. Since overload situations are unexpected operating conditions which may cause damage, the emergency stop needs to be triggered as soon as possible after the beginning of the overload situation. As described in [1], the acceptable latency is in the order of few milliseconds.

Predictive maintenance approaches such as a remaining lifetime prognosis have more relaxed latency requirements. Data may be acquired over long periods of time and then analyzed offline or off-site (e.g. using cloud services).

The sensor to be integrated in our use case is the Schaeffler SpindleSense sensor [16]. It measures axial and radial displacements between a sensor ring attached to the spindle housing and a measurement ring attached to the spindle shaft. Its resolution is approx. $1\,\mu\mathrm{m}$, measurement data is sampled at $1\,\mathrm{kHz}$ and provided via CAN bus (Controller Area Network). Since neither the CNC unit nor the edge unit offers CAN bus interfaces directly, a protocol conversion unit is needed. While the sensor may provide internal data processing for spindle monitoring, this is not used in our setup to maximize flexibility.

C. Proposed System Architecture

Fig. 1 shows the overall system architecture of the machining center. The machine is controlled by a CNC unit as described in Section III-A. It is connected to an industrial edge unit via a local Ethernet network within the machine. The edge device is connected to the shopfloor network and can be used

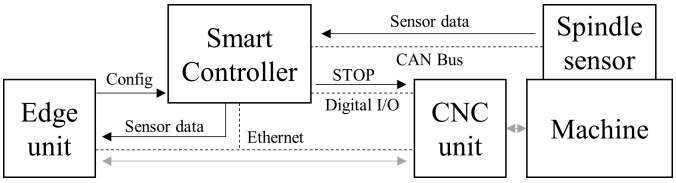

Fig. 1. Overall system architecture. Physical communication lines and buses are depicted as dashed lines, logical communication links using arrows. Communication links unrelated to the spindle sensor are marked gray.

to forward information to manufacturing execution systems (MES) and cloud services.

The newly added spindle sensor is installed in the machine. It is connected to a smart controller unit via CAN bus, which in turn is connected to the machine-local Ethernet network and to the machine control via a dedicated digital signal line to signal an overload condition and thereby trigger an emergency stop.

The smart controller implements the protocol conversion and data pre-processing to efficiently provide spindle sensor data to the edge unit and runs low-latency data processing for spindle overload detection. It checks for overload conditions and sends a stop signal to the machine control when necessary. In parallel, it filters and aggregates sensor data according to a configuration received from the edge unit and then forwards the aggregated data to that unit.

D. Proposed Smart Controller Architecture

Fig. 2 shows the smart controller hardware and software components used to realize machine monitoring and data pre-processing as well as their relation.

The Lemaker Banana Pi M1 single-board computer [17] is used as hardware platform for the smart controller unit. It features an Allwinner A20 SoC comprising two ARM Cortex-A7 CPU cores, 1 GB DDR3 RAM, and an integrated CAN MAC peripheral among others. To interface the CAN bus, an SN65HVD230 CAN transceiver is used.

As proposed in the smart controller concept, the Jailhouse hypervisor is used to enable the concurrent execution of a Linux OS (operating system) and the FreeRTOS real-time OS (RTOS) and to control their access to system resources. The OS run inside Jailhouse partitions, which are exclusively assigned one CPU core each. Jailhouse is configured to grant the RTOS partition exclusive access to the CAN controller, the GPIO (general-purpose input/output) registers to issue the stop signal, and to the respective registers of the clock control units. For debugging and evaluation it is also granted access to a UART peripheral (Universal Asynchronous Receiver Transmitter) and further GPIO registers. Relevant CAN and UART interrupts are routed to the RTOS partition. The remaining hardware resources are assigned to the Linux partition. A hardware timer module and memory for data exchange are shared between the Linux and RTOS partitions.

In FreeRTOS, we implemented a CAN driver controlling the Allwinner A20 CAN peripheral. A data acquisition task uses it to receive CAN messages sent by the sensor. It parses the sensor data, runs the overload detection, and then forwards

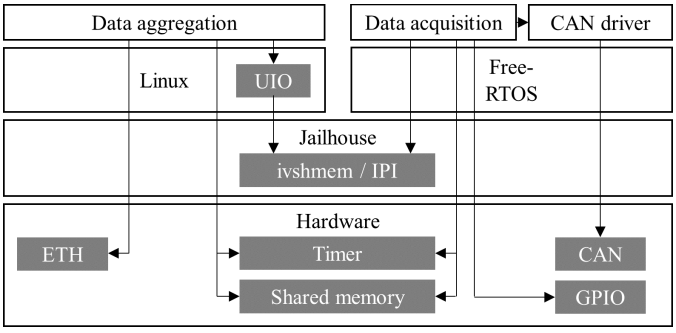

Fig. 2. Smart controller hardware/software stack. Arrows indicate component usage.

it to the Linux partition. The overload detection signals an overload condition if the spindle displacement in any direction exceeds a defined threshold. In this case, a GPIO pin is set to indicate the overload condition. This signal can be interpreted by the machine control to stop the machine. Here, voltage level shifting from the Banana Pi's 3.3 V levels to the machine control digital I/O levels may be necessary. If the displacement values along all axes are below the threshold, the GPIO pin is reset, indicating normal operation.

Communication between the RTOS and Linux is realized using a simplified variant of the *RPMsg* protocol. Our implementation is based on the RPMsg-Lite component [18]. It uses shared memory for data exchange and the *ivshmem* interface provided by Jailhouse to signal new data availability. The latter is a virtual-PCI-based interface which can be used to trigger inter-partition interrupts (IPI). On Linux, the data aggregation application contains the RPMsg implementation and receives these interrupts via the Linux Userspace I/O (UIO) drivers. It then filters and aggregates the sensor data and then sends it to the edge unit via Ethernet (ETH). The aggregation period as well as filtering parameters can be defined by the edge unit using configuration commands.

When combining information from multiple sources, e.g. multiple sensors, data synchronization is important. Under Linux, this can be achieved using clock synchronization protocols such as NTP or PTP. In our setup, a common NTP server is used for time synchronization between the edge unit and the smart controller. However, since the sensor data is acquired by the RTOS, time synchronization between Linux and RTOS is needed. This is achieved by a shared hardware timer unit serving as a common time base between the partitions. Incoming sensor data is annotated with the timer value on reception and then forwarded to Linux. There, the system-wide sensor data timestamp is determined by considering the offset between the current and annotated timer value.

IV. Evaluation

One objective of this work is to determine the achievable end-to-end latency of the smart controller concept on the Banana Pi M1 platform. We used two scenarios for evaluation:

- **Use-case-based evaluation:** We measured the reaction time of the smart controller unit in an industrial use case.

- **Generic evaluation:** We measured the reaction in a simple, generic case and investigated the effect of system load.

In both scenarios, the smart controller architecture is compared to a native Linux approach where Linux is running directly on the CPU without a hypervisor or RTOS.

A. Measurement Setup

To measure the reaction time of the smart controller, an external measurement unit is used which generates input signals to the smart controller and receives its output signal. We use a Xilinx Zynq UltraScale+ ZCU102 evaluation board comprising an FPGA, processor cores, hardware timer units, CAN MAC peripherals, and a CAN transceiver, among others. It supports $3.3\,\mathrm{V}$ I/O and can thereby be connected directly to the Banana Pi. The measurement unit is connected to the smart controller CAN interface as well as to two digital I/O lines as shown in Fig. 3.

B. Use-case-based Evaluation

To determine the end-to-end latency of the smart controller in an industrial use case we connected the smart controller unit to the measurement unit instead of the real spindle sensor and machine control. The measurement unit simulates the spindle sensor by sending CAN messages identical to the sensor's. For each measurement, the measurement software first triggers the measurement unit's CAN peripheral to send a sensor message indicating displacement beyond the threshold. The CAN peripheral confirms successful transmission by the TXOK interrupt. In the corresponding interrupt handler of the measurement software, a hardware timer is started. The measurement software then polls the feedback line and stops the timer when the signal value indicating overload is seen. Based on the timer tick count, the delay is calculated.

To compare the smart controller concept against a native Linux implementation, two setups were investigated:

- **Smart Controller:** The smart controller is implemented as described in Section III-D.
- **Native Linux:** A native Linux implementation is used. To make full use of the Linux features, a user space application controls the GPIO pins, i.e. the stop signal, using the libgpiod library. It uses the SocketCAN library to interact with the CAN peripheral. Upon reception of a CAN message, the threshold check is done and the stop signal is set based on the result.

Table I shows the results of 1000 measurements. The smart controller setup shows significantly lower latencies than the Linux setup. This can be explained by the fact that the RTOS task has lower overhead when accessing GPIO and CAN peripherals due to less context switches and lean drivers as well as less other threads waiting to be scheduled when compared to a Linux user space application. For the Linux application, these aspects together with context switches between kernel and user mode, among others, lead to a high latency.

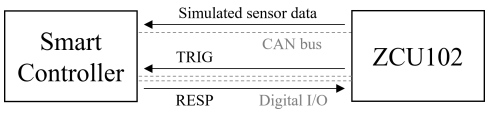

Fig. 3. Measurement setup. Physical connections are depicted as gray dashed lines, arrows indicate the data flow.

TABLE I
END-TO-END LATENCY FOR SENSOR CAN BUS MESSAGES, 1000 MEASUREMENTS PER SETUP

Evaluation Setup	Latency [µs]		
	mean* ± *SD	***max***	***min***
Smart Controller	130 ± 8.3	182	125
Native Linux	$12\,107 \pm 104$	12 917	12 013

C. Generic Evaluation and the Effects of System Load

We compare the smart controller implementation to different native Linux implementations concerning achievable latency when reacting to external events and the influence of varying load situations on this latency. A basic functionality is used for this evaluation, serving as a generic, minimal example of input data processing: As soon as a positive edge is received on an external digital input line (TRIG line, e.g. from a light barrier), a positive edge shall be output on an output line (RESP line, e.g. the stop signal). While this functionality would be redundant in most real-world applications, it serves as a benchmark for the end-to-end latency caused by OS overhead, scheduling effects, and hardware.

To measure the latency, the measurement unit sends a positive edge on the TRIG line which is received by the Banana Pi's GPIO peripheral, triggering an interrupt. When the edge is detected by the application software, i.e. the interrupt handler is executed, it outputs a positive edge on the RESP line. This is detected by the measurement unit, which determines the delay between sending and receiving an edge.

For this evaluation a high precision measurement system was used, realizing the measurement functionality in the FPGA of the measurement unit. The measurement circuit consists of a state machine, which is directly interfacing the measurement lines and is controlling a counter. It is clocked at $500\,\mathrm{MHz}$ and therefore achieves a temporal resolution of $2\,\mathrm{ns}$. To determine the latency of the measurement system itself, the measurement lines were connected directly to each other. This resulted in a measurement of $18\,\mathrm{ns}$ and therefore can be neglected given the measurement results. Using FPGA-based setups is an established method for latency measurements in similar applications, e.g. the OSADL Latency Box [19].

The following setups were evaluated:

1) **Smart Controller:** The smart controller implementation as described in Section III-D. No CAN messages were received during the measurement. An RTOS task is used to monitor the TRIG signal and set the RESP signal.
   a) The RESP signal is set directly in the IRQ handler.
   b) The RESP signal is set by a high-priority RTOS task which is woken by the IRQ handler.

TABLE II
END-TO-END LATENCY [µs] FOR DIGITAL INPUT SIGNALS, 1000 MEASUREMENTS PER SETUP

		system idle			system under load		
		mean* ± *SD	***max***	***min***	***mean* ± *SD***	***max***	***min***
Smart Controller	RTOS IRQ handler	89.3 ± 9.0	105.1	73.4	90.8 ± 14.4	264.2	70.4
	RTOS task	136.3 ± 9.1	152.7	117.1	139.1 ± 20.3	543.3	118.1
Linux	IRQ handler	90.9 ± 9.9	213.7	73.0	101.8 ± 10.2	144.0	81.4
	Threaded IRQ handler	138.0 ± 17.4	574.4	117.3	278.7 ± 139.4	1097.7	123.7
	libgpiod	$11\,993 \pm 287$	20 430	11 888	$179\,289 \pm 18\,171$	240 255	99 191
Linux (Preempt-RT)	Threaded IRQ handler	158.6 ± 56.9	1928	121.3	165.5 ± 62.8	2093	107.9
	libgpiod	$27\,593 \pm 82.2$	29 311	27 450	$27\,950 \pm 208.9$	29 882	27 510

2) **Native Linux:** Linux is running directly (natively) on the CPU.
   a) A kernel driver is used to monitor the TRIG signal, the RESP signal is set directly in the IRQ handler.
   b) A kernel driver is used to monitor the TRIG signal, the RESP signal is set in a threaded IRQ handler.
   c) A user space application is used to monitor the TRIG signal and set the RESP signal. It uses the libgpiod library to interact with the kernel's GPIO framework and its event loop to detect TRIG edges.
3) **Native Linux (Preempt-RT):** Linux with Preempt-RT patches is running directly on the CPU. Implementations evaluated are the same as in setup 2. However, implementation 2a could not be used with Preempt-RT patches. It does not match the concept of Preempt-RT, which includes handling IRQ handlers as threads.

We measured the latency with and without load on the Linux OS for all setups. Load was induced using the `stress` application [20]. To cause load on the CPU, memory, and I/O, 10 workers spinning on a square root function, 10 workers allocating and freeing $80\,\mathrm{MB}$ memory each, and 10 workers causing I/O load by spinning on `sync()` were launched using the parameters `stress -c 10 -m 10 --vm-bytes 80000000 -i 10`.

Table II shows the results of 1000 measurements per setup and load siuation. In idle state the smart controller implementations and Linux kernel module implementations show similar latencies. For IRQ handler implementations they are about $90\,\mu\mathrm{s}$, for task/threaded IRQ handler implementations about $137\,\mu\mathrm{s}$. At the same time, the maximum latencies seen for the smart controller setup are significantly lower than for native Linux implementations ($49\,\%$ of the Linux latency for IRQ, $27\,\%$ for task/threaded IRQ).

Under high load, the mean latencies in the smart controller setups increase just by few microseconds while for the Linux kernel driver setups they increase by $12\,\%$ (IRQ handler) and $102\,\%$ (threaded IRQ). This may be caused by Linux kernel code and modules which run in an IRQ context or disable interrupts for some time, thereby postponing the reaction to the TRIG signal. For threaded IRQs, delays may be explained by additional scheduling overhead and other threads competing for CPU time. Using Preempt-RT, a lower average latency can be achieved under load.

Regarding the maximum latencies encountered during our measurements, there where cases the smart controller setup shows higher latencies than the native Linux setup. This was seen for the IRQ handler implementations under load and is reproducible. It may be caused by hardware specifics of the Allwinner A20 chip. Follow-up experiments indicate that this effect is related to the memory subsystem since it only occurred when memory stress was involved. Without memory stress, similar maximum latencies were measured.

As discussed in Section IV-B, there is a large difference in latency when comparing user space implementations to smart controller or Linux kernel module implementations. While smart controller and kernel implementations lead to mean latencies of approx. $90\,\mu\mathrm{s}$ to $160\,\mu\mathrm{s}$ without load and $90\,\mu\mathrm{s}$ to $280\,\mu\mathrm{s}$ under load, user space implementations result in latencies in the range of tens of milliseconds without load and hundreds of milliseconds under load. Our experiments show that using Preempt-RT patches significantly reduces this impact of system load on user space implementations' latencies, e.g. reducing their effect on the mean latency from $167\,\mathrm{ms}$ to $357\,\mu\mathrm{s}$.

V. DISCUSSION

Our evaluation shows that functionality realized in the RTOS of the smart controller architecture shows significantly lower reaction latencies than the same functionality implemented as a native Linux user space application. This can be explained by the overhead introduced by Linux. To avoid this overhead, the functionality could be implemented as a Linux kernel module. This allows for direct access to peripherals without context switches between kernel and user mode and leads to comparable latencies when the system is idle.

However, implementing functionality in the kernel allows for full access to all memory in a native Linux setup, which breaks memory isolation between applications running on the CPU. This can be undesirable for security or IP protection reasons, e.g. if software from different suppliers is integrated in one system. Using a hypervisor enables the integration of low-latency software in the RTOS partition while isolating it from software running on Linux in kernel mode.

Under load, the smart controller concept leads to lower average latencies when compared to Linux kernel module implementations ($50\,\%$–$89\,\%$ of the latencies measured using native Linux). Soft real-time applications thereby profit from

the isolation provided by the hypervisor. However, RTOS latencies are still influenced by the load on the Linux partition. This is caused by interference between the CPU cores of the hardware platform due to shared hardware ressources (e.g. caches and buses). Follow-up experiments indicate that this interference mainly coincides with memory load, indicating that shared caches and memory buses may be a major cause. This could be tackled at hypervisor level by applying cache-aware techniques in memory partitioning, such as the cache coloring approach presented in [21].

Regarding the industrial application, we presented an architecture which we successfully used to integrate a spindle sensor into an existing Industry 4.0 testbed (Section III). We demonstrated in Section IV-B that the hypervisor-based architecture for the smart controller achieved an end-to-end latency of $130\,\mu s$ on average and at most $182\,\mu s$, while the native Linux implementation had a latency of up to $12.9\,ms$. Therefore, the hypervisor-based architecture enables the requirements of the application (reaction within few milliseconds) to be met, unlike the Linux-based solution. The machining center considered in this use case has a maximum travel speed of $40\,m/min$, under these conditions the maximum latency corresponds to a traveled distance of up to $0.121\,mm$ in the first setup and $8.61\,mm$ in the second setup.

VI. Conclusion

In this work we demonstrated how the smart controller architecture proposed in [1] can be integrated in an industrial use case. We then evaluated it both in this real-world application and in a generic test to determine end-to-end latencies to external events and compared this to native Linux implementations. While the smart controller concept shows significantly lower latencies when compared to native Linux user space implementations of the same functionality, we found that Linux-kernel-module-based implementations can achieve similar mean reaction times as long as the system is mostly idle. When the system is under load, however, the smart controller approach shows lower mean latencies compared to Linux kernel module implementations, making it profitable for soft-real-time applications. Besides, while the realization of functionality in kernel modules may break application isolation requirements in native Linux setups, the isolation between low-latency applications and others may still be achieved when applying the smart controller concept. For hard real-time applications, maximum latencies are relevant. Here, the results are inconclusive concerning a potential benefit of the smart controller concept over Linux kernel module implementations. Therefore, next steps include further investigation of the causes of interference between the CPU cores on the hardware platform and the applicability of cache-aware partitioning techniques.

Acknowledgment

This research and development project was funded by the German Federal Ministry of Education and Research (BMBF) under grant number 02P17X000. The I4TP project is managed by the Project Management Agency Karlsruhe (PTKA). The authors are responsible for the contents of this publication.

References

[1] D. Barton, P. Gönnheimer, F. Schade, C. Ehrmann, J. Becker, and J. Fleischer, "Modular smart controller for industry 4.0 functions in machine tools," *Procedia CIRP*, vol. 81, pp. 1331–1336, 2019.

[2] T. Schlagenhauf, C.-P. Feuring, J. Hillenbrand, and J. Fleischer, "Camera based ball screw spindle defect classification system," in *Production at the leading edge of technology*, J. P. Wulfsberg, W. Hintze, and B.-A. Behrens, Eds. Springer Berlin Heidelberg, 2019, pp. 503–512.

[3] H. C. Möhring, S. Eschelbacher, and P. Georgi, "Fundamental investigation on the correlation between surface properties and acceleration data from a sensor integrated milling tool," *Procedia Manufacturing*, vol. 52, pp. 79–84, 2020.

[4] X. Y. Zhang, X. Lu, S. Wang, W. Wang, and W. D. Li, "A multi-sensor based online tool condition monitoring system for milling process," *Procedia CIRP*, vol. 72, 2018.

[5] C. Schmitz, A. Tschiesner, C. Jansen, S. Hallerstede, and F. Garms, "Industry 4.0: Capturing value at scale in discrete manufacturing."

[6] B. Chen, J. Wan, A. Celesti, D. Li, H. Abbas, and Q. Zhang, "Edge computing in iot-based manufacturing," *IEEE Communications Magazine*, vol. 56, no. 9, pp. 103–109, 2018.

[7] H. Mueller, S. V. Gogouvitis, A. Seitz, and B. Bruegge, "Seamless Computing for Industrial Systems Spanning Cloud and Edge," in *2017 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2017, pp. 209–216.

[8] D. Barton, R. Stamm, S. Mergler, C. Bardenhagen, and J. Fleischer, "Industrie 4.0 Nachrüstkit für Werkzeugmaschinen: Modulare Lösung für zustandsorientierte Instandhaltung und Prozessüberwachung [Industry 4.0 retrofitting kit for machine tools: Modular solution for condition-based maintenance and process monitoring]," *WT Werkstattstechnik*, vol. 110, no. 7-8, pp. 491–495, 2020.

[9] Jailhouse project. [Online]. Available: https://github.com/siemens/jailhouse

[10] S. Toumassian, R. Werner, and A. Sikora, "Performance measurements for hypervisors on embedded arm processors," in *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2016, pp. 851–855.

[11] J. Danielsson, T. Seceleanu, M. Jägemar, M. Behnam, and M. Sjödin, "Testing performance-isolation in multi-core systems," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, 2019, pp. 604–609.

[12] The RTL Collaborative Project [Wiki]. [Online]. Available: https://wiki.linuxfoundation.org/realtime/start

[13] RTAI - the RealTime Application Interface for Linux. [Online]. Available: https://www.rtai.org/

[14] Xenomai project. [Online]. Available: https://www.xenomai.org

[15] A. Lackorzynski, C. Weinhold, and H. Härtig, "Predictable low-latency interrupt response with general-purpose systems," in *Proceedings of OSPERT2017, the 13th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications OSPERT 2017*, 2017, pp. 19–24.

[16] Schaeffler Technologies AG & Co. KG, *Schaeffler SpindleSense (TPI 258)*. [Online]. Available: https://www.schaeffler.de/remotemedien/media/_shared_media/08_media_library/01_publications/schaeffler_2/tpi/downloads_8/tpi_258_de_en.pdf

[17] Shenzhen LeMaker Technology Co., Ltd. Banana Pi specifications. [Online]. Available: http://www.lemaker.org/product-bananapi-specification.html

[18] Rpmsg lite component. [Online]. Available: https://github.com/NXPmicro/rpmsg-lite

[19] F. Gottschling and C. Emde, *The OSADL Latency Measurement Box*, Open Source Automation Development Lab (OSADL) eG. [Online]. Available: https://www.osadl.org/uploads/media/OSADL-Latency-Box.pdf

[20] stress project. [Internet Archive]. [Online]. Available: http://web.archive.org/web/20190707055144/http://people.seas.harvard.edu/~apw/stress/

[21] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic memory hierarchy and virtualization for modern multi-core embedded systems," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 1–14.