Low-latency inter-domain communication on the Xen hypervisor

Fabian Lesniak
Karlsruhe Institute of Technology
Karlsruhe, Germany
lesniak@kit.edu

Tanja Harbaum
Karlsruhe Institute of Technology
Karlsruhe, Germany
harbaum@kit.edu

Jürgen Becker
Karlsruhe Institute of Technology
Karlsruhe, Germany
becker@kit.edu

***Abstract*—Designing mixed-critical systems that provide both safety-critical and best-effort functions is a major challenge. In embedded systems, hypervisors are increasingly used to separate sensitive real-time tasks from best-effort applications. To take advantage of hypervisors at the application level as well, there is an emerging trend to split large applications into multiple guest domains to make the software easier to analyze and maintain. However, having more partitions leads to increased communication overhead between guest domains, increasing the impact of latencies on performance. In this article, we propose a framework for Inter-Domain Communication (IDC) to tackle these issues by reducing IDC latencies. It is built upon established open-source implementations of the VIRTIO and RPMsg specifications, allowing to reuse existing drivers for guest OS or bare-metal software. We can show latency improvements of up to 3x compared to UDP communication between two guests, even faster than UDP communication using the loopback interface within one Linux guest. Splitting applications into separate guest domains is discussed, as a trace-off between communication latency and computational complexity exists. Further a service-oriented example application is analyzed, where our approach can lead to improvements through inherent concurrency. In addition, we demonstrate how to extend the original peer-to-peer scheme with a router component to allow multiple guests to exchange data.**

***Index Terms*—Hypervisor, inter-domain communication, low latency, VIRTIO, mixed-critical systems.**

I. Introduction

Throughout many technological domains, the requirements for embedded computing systems have risen tremendously in the last two decades, in terms of the number of implemented functions, their complexity and demands on communication [1]. As a result there is an increasing trend for mixed-criticality systems, where multiple components with different dependability and real-time characteristics (e.g. safety–critical and consumer functionality) are integrated into a shared computing platform [2]. The reasons behind this trend is lowering costs and power consumption as well as improving maintainability [3].

Virtualization technology can provide computational isolation among multiple applications on a common hardware platform [4]. It is increasingly used as a key technology in the middleware stack of embedded systems, whenever stronger isolation compared to an operating system is required. This includes mixed-criticality systems as well as cloud computing devices which execute applications from different ecosystems or with different requirements in terms of response time, computational performance or communication bandwidth in parallel.

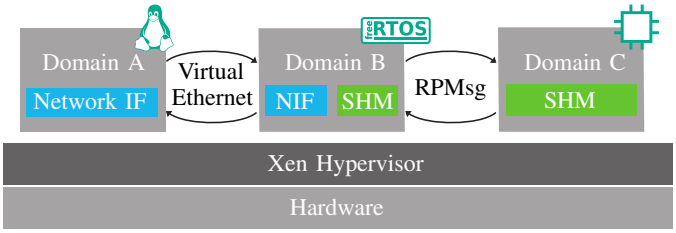

Fig. 1. A typical hypervisor setup with multiple heterogeneous guest domains. Domains A/B are communicating using ordinary virtual networking with high latency, whereas B/C use the proposed concept using RPMsg and shared memory with very low latency.

The benefits of virtualization do not only apply to independent, strongly isolated applications. By dividing components of a large, interconnected system (e.g. control loops or image verification chains) into multiple partitions, it is easier to service, update and monitor the individual components. However, such fine-grained subdivided modules require reliable and rapid communication with other modules and the outside world. In industrial practice, two technologies are commonly used: On the one hand, a virtualized network can be used, similar to hypervisor applications in datacenters. This is flexible and easy to use but the latency overhead caused by network virtualization has to be considered. Especially on less powerful embedded systems, this overhead becomes significant when a large number of guest domains is connected using virtualized network adapters. On the other hand, communication based on Shared Memory (SHM) can achieve higher performance, but is less flexible and typically more complex to implement. To enable efficient Inter-Domain Communication (IDC) on embedded systems, a solution is required that offers the performance of SHM but is similarly flexible and easy to use as virtualized networks.

This paper discusses an approach for low-latency IDC between guest domains on a virtualized platform. Figure 1 shows a typical hypervisor setup where two domains communicate using virtualized network adapters and another pair uses the improved IDC mechanism with minimal latency. While covering the technical details, evaluation and an example use-

case, the following chapters also focus in particular on existing open-source standards and tested implementations thereof, which are combined to build the proposed IDC system.

II. Related work

The Xen hypervisor offers basic integrated mechanisms for communication between partitions. Most common are virtual network devices, which allow a guest domain to participate in Ethernet networks. Depending on the virtual device model, it is also possible to use Virtual I/O Device (VIRTIO) as backend. Therefore, high bandwidths are possible, but not on par with physical Ethernet interfaces [5]. However, the necessity of networking stacks on both sides of the virtual Ethernet endpoint adds a considerable amount of latency [6].

Contrary to virtual devices for guests, one increasingly finds hardware with support for concurrent access by multiple virtualized guests [7]. Such integrated virtualization features allow for higher performance than a paravirtualized variant and reduced hypervisor overhead and complexity.

Further developments like PrioIDC [8] effectively improved latency by adapting the communication architecture to the schedule of the hypervisor. However, this concept does not offer advantages in terms of modularity and flexibility. By defining the scheduling, it is only suitable for mixed-critical systems that are statically defined at design time.

This is where the concept of CAN-to-Ethernet communication channels [6] comes into play. It connects the virtual network communication of Xen with existing CAN interfaces of legacy applications. This method, which is exclusively geared towards the automotive domain, offers a high degree of flexibility with sufficient performance. However, the latency does not come close to the order of magnitude of our concept, nor does it provide support for monitoring of guest partitions.

Monitoring is the main goal of VOSYSmonitor [9], a software layer based on ARM TrustZone. It can be used to run safety critical and non-critical software on the same device with isolation and conformity to automotive standards. The communication interfaces are implemented as network adapters, which introduces additional overhead for IP stacks on each guest domain.

The authors of TZ-VirtIO [10] present an communication mechanism which is partially similar on implementation level. Their method also combines VIRTIO with Remote Processor Messaging (RPMsg), but with different objectives: RPMsg is used for its original purpose of communicating with independent coprocessors, being a real-time capable processor in this case. Therefore, it can be used to extend our IDC mechanism for communication with external processors, but not for communication between domains on one platform.

Table I summarizes the aforementioned work and compares them with regard to different aspects. Some approaches do allow communication with low latency and high throughput. However, if both modularity and service-oriented communication is required in addition, none of the approaches will do. In this work we are targeting this gap, providing a low-latency IDC with modular and service-oriented design. The overhead per guest should be kept as low as possible so that the solution can be scaled to a particularly large number of guests.

TABLE I
State of the art approaches compared to our proposed architecture

	This work	Xen	PrioIDC [8]	C2E [11]	Vosys [9]	TZ-VirtIO [10]
latency	+	-	+	-	o	+
throughput	+	o	+	o	o	+
modularity	+	-	+	+	o	-
service-oriented	+	-	-	+	o	-
TEE	-	-	-	-	+	+

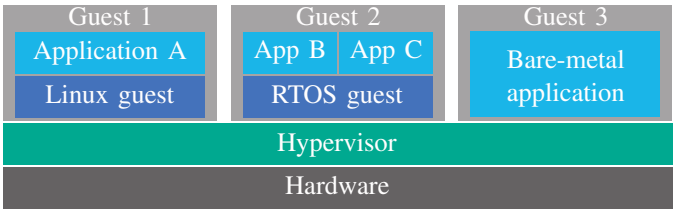

Fig. 2. Type-1 hypervisor with multiple guest domains

III. Inter-Domain Communication (IDC)

To realize IDC, the hardware features of the target platform have to be considered. In most modern systems, means for communication between hypervisor domains or between hypervisor and guest domains are limited to SHM, interrupts and hypercalls. Mechanisms on higher layers like virtualized devices (e.g. Ethernet) are using these features in the background.

Figure 2 shows a type-1 hypervisor and a choice of different guests running on top of it: a Linux domain, a real-time operating system (RTOS)-based application and bare-metal software. The hypervisor provides computing and memory resources to guests while establishing isolation between individual guests and scheduling their execution as configured. Input and output is usually realized using virtual devices created by the hypervisor (emulated or paravirtualized devices [12]), e.g. providing access to Ethernet networks [6]. It is also common to pass physical hardware to individual guests which then have exclusive access at near-native performance [7]. However, the number of physical interfaces is limited and even pass-through can introduce significant overhead [5] and is therefore out of the question for IDC.

Paravirtualized devices are designed with the hypervisor in mind leading to reduced emulation overhead and simplified drivers on the guest. However, a paravirtualized Ethernet controller implies a lot of overhead for guest domains: In addition to the device driver, an Ethernet, IP and TCP/UDP stack is necessary to participate in common networks. While easy to use in a fully fledged guest OS like Linux, the complexity of bare-metal guests and some RTOSs increases greatly if such functionality is included. Therefore, we propose a lightweight IDC system with very little overhead. Designs

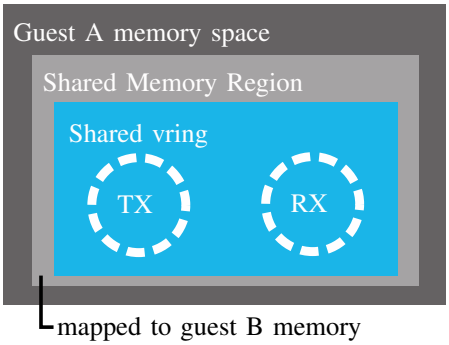

Fig. 3. Two virtqueue ring buffers (vring) shared between two guests

with many guest domains (e.g. in a microservice environment) will especially benefit from low latency and a small memory footprint of individual guests.

A. Shared Memory (SHM) management

From a guest perspective, SHM is the only resource allowing to exchange large amounts of data with other domains or the hypervisor. This SHM region needs to be managed in a uniform way by both communication partners. The VIRTIO [13] specification defines a standard interface for paravirtualized devices. Although the notion of frontend (guest driver interface) and backend (device model on host side) is tailored to paravirtualized devices, the VIRTIO mechanism can be used for IDC as well: To establish a peer-to-peer channel, one domain takes the role of the frontend domain, while the other domain acts like a device (backend). VIRTIO can define two ringbuffers (*virtqueue*) in the memory of the frontend domain, one serving as transmit (TX), the other as receive (RX) channel.

Several operating systems and run-time libraries include drivers for VIRTIO. These drivers are usually well tested and can be expected to be more stable than a new custom driver. Multiple backends are specified for VIRTIO, most important PCI Express (PCIe) and Memory Mapped I/O (MMIO). As a PCIe bus introduces additional overhead, the MMIO backend is preferred.

VIRTIO over MMIO is realized as follows: Both frontend and backend share a memory region containing virtqueues as well as configuration and status space (Figure 3). Therefore, one single SHM region is sufficient to set up the virtio connection. Notifications can be delivered using status bits in the status region, additional features like interrupts can also be used, although this is not explicitly defined in the VIRTIO standard.

To set up a VIRTIO channel, frontend and backend domains need to exchange information, which primarily consists of the shared memory location/size and the driver status on both sides. As VIRTIO does not define how such communication should be implemented, it is usually performed by the backend mechanism. When using PCIe, status information exchanged using a memory BAR. However, for the MMIO backend, such a channel needs to be set up independently. In our case, the Xen hypervisor provides the `xenstore`, a key-value store to exchange status information with the hypervisor and optionally other domains. It is accessible via a SHM region available to every Xen domain by default. The hypervisor manager is supposed to publish values in the `xenstore`, thus deciding which domains are allowed to communicate. Usually this is done by the privileged domain *Dom0*, but any domain with write permissions on the `xenstore` root can be used. The virtual device abstraction layer of Xen, `xenbus`, resides in the `xenstore`. The Xen project defines semantics for devices and drivers to exchange static information and state via the `xenbus` [14].

It is possible to assign static SHM regions between domains, but doing so is only recommended between trusted domains according to the Xen manual [15]. Thus, Xen *grants* are used to dynamically set up SHM, using the `xenstore` to negotiate with the hypervisor. Compared to static allocation, no memory needs to be reserved prior to guest startup.

B. RPMsg

RPMsg is a scheme for data exchange between processors [16]. It was originally designed for asynchronous multiprocessors and is commonly used to access independent coprocessors in a System on Chip (SoC). However, since RPMsg is based on VIRTIO, it maps seamlessly to SHM between guests. It uses two virtqueues, one serving as TX, the other as RX channel. On top of the virtqueues, multiple independent *endpoints* can be created, allowing to virtually divide the channel into multiple logical channels for individual purposes. As an additional benefit, the RPMsg header is very short with only 16 bytes in size. Implementations of RPMsg are available in the Linux kernel starting from version 3.4 as well as in the OpenAMP framework, which can be used in both bare-metal and RTOS systems.

Small, irregular messages and continuous data streams can be transmitted in the same way, making it suitable for different applications. If the drivers are set up properly for the respective target platform, almost zero-copy behavior is possible. At minimum, a copy between userspace and the kernel-managed virtqueue is necessary on Linux. On bare-metal, true zero-copy is possible.

C. Setup and teardown of VIRTIO via `xenbus`

The process of setting up and tearing down a communication channel is based on a state machine in both the frontend and backend driver. This state field resides in the `xenstore`, reusing the `xenbus_state` definition by Xen. After domain startup, both frontend and backend wait for configuration keys to appear in the `xenstore`. The process is visualized in Figure 4 and consists of the following steps:

1) Both frontend and backend wait for configuration keys to appear on the `xenbus`.
2) As soon as the frontend detects a matching root entry, it publishes vendor/device IDs and virtqueue information.

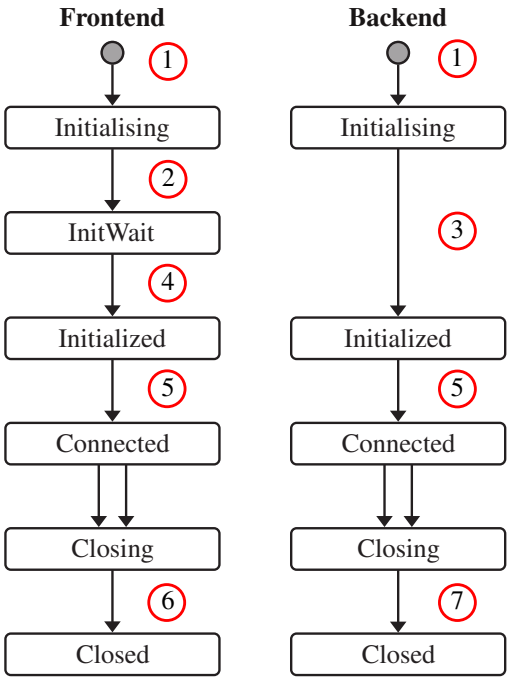

Fig. 4. State machine life cycle on frontend and backend; numbers on state transitions refer to the detailed description in section III-C

3) The backend initializes virtqueues and allocates memory grants and event channel ports. It also creates the local VIRTIO and RPMsg devices, if the device ID matches.
4) The frontend can now mount the grants using the published information and create the VIRTIO and RPMsg device counterparts.
5) The communication channel is now ready for use. Both sides wait for the other endpoint to switch to the *Closing* state.
6) To close the channel, the frontend releases the device and SHM resource first, signaling completion by entering the *closed* state.
7) The backend can release all resources after the frontend has disconnected, the channel is now closed.

After the channel is closed, the `xenstore` values can be cleaned up by the hypervisor manager. We do not provide any measures for error handling during channel setup, as the cause must be an implementation or configuration error. In case of an error, the drivers stay in their current state and need to be reset manually.

D. Driver stack

Figure 5 shows the full software stack on both frontend and backend. Basically, the stack is identical for both sides, except for initialization differences between backend and frontend. Parts of the stack can reside in kernel space when using an OS, but boundary between kernel and user mode can be chosen freely to fit implementation needs. In our reference implementation, a channel between a Linux domain implementing the frontend and a bare-metal domain in the backend role is

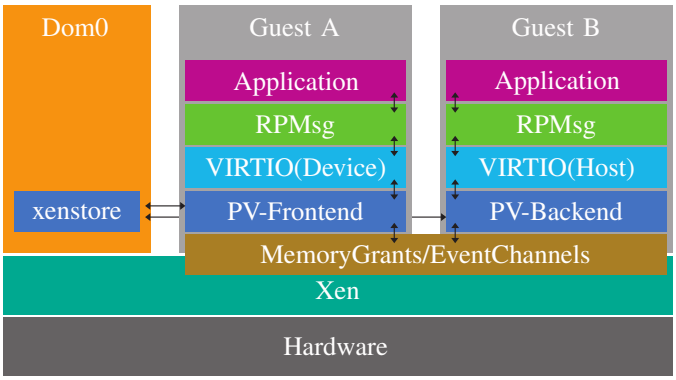

Fig. 5. The full communication stack between two domains including xenstore for management

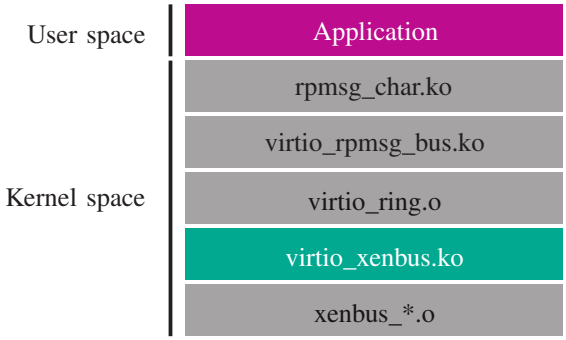

Fig. 6. Driver stack on Linux side (frontend)

shown. First, the Linux driver stack is discussed, following a description of a driver stack built into a bare-metal domain using OpenAMP.

The Linux kernel includes drivers for RPMsg, providing a character device to userspace applications as well as a backend driver binding to VIRTIO (`virtio_rpmsg_bus`). Naturally, Linux also provides a VIRTIO driver infrastructure, whereof our design uses the `virtio_ring` module primarily. A framework to read, write and received notifications via the `xenbus` is also available, providing an interface to allocate SHM grants and event channels as well. Here, our main contribution is the missing link between these frameworks: a driver called `virtio_xenbus`, which allows to create VIRTIO devices (and subsequently RPMsg sub-devices) based on `xenbus` entries. Figure 6 shows the stack of drivers and the location of the `virtio_xenbus.ko` driver in it, connecting `xenbus` and VIRTIO interfaces. Here, the Linux side implements the frontend role, allowing to use the Linux kernel interfaces to allocate grants and event channels as well as to initialize virtqueues and shared structures within the SHM. However, this is not an inherent limitation of the proposed architecture: frontend/backend roles can be exchanged easily and communication between Linux/Linux or bare-metal/bare-metal are possible in the same way.

In contrast to the Linux implementation, the bare-metal stack uses VIRTIO and RPMsg implementations of OpenAMP.

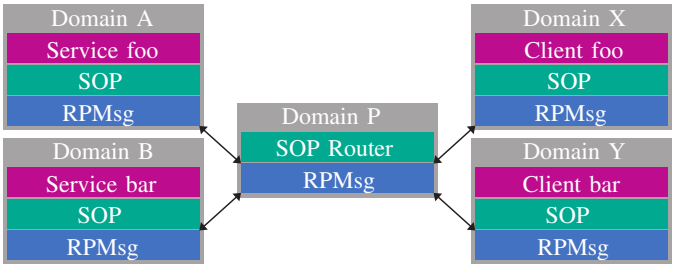

Fig. 7. Structure of multiple domains communicating with SOP over the RPMsg connection

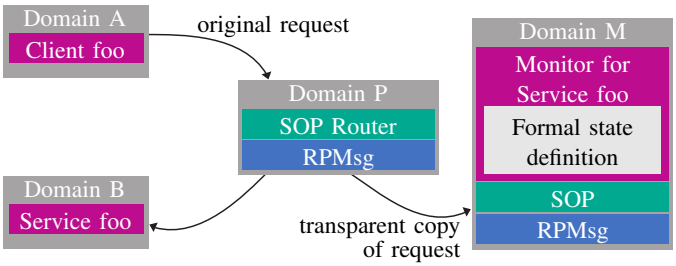

Fig. 8. Architecture with a dedicated partition to monitor a service using IDC. The router component transparently forwards communication to the monitor.

They turned out to be well tested and stable for this use case and no modifications to the OpenAMP code are necessary. However, OpenAMP does not provide drivers to access the `xenstore` and manage `xenbus` data, and also lacks interfaces for the grant table. We contribute these features as part of an additional library, alongside the state machine logic (Section III-C). It forms the functional equivalent to the `virtio_xenbus` Linux driver, but as an OS-independent library. This library makes it easy to realize our proposed scheme within other frameworks as RTOSs.

E. Service-oriented communication

RPMsg provides a simple interface to send and receive unstructured binary data, allowing to easily adapt it to different use cases. To demonstrate its versatility, we implemented a library for service-oriented communication on top of RPMsg. This library called Service Oriented Protocol (SOP) allows for domains to issue method calls to services implemented in other guest domains, handling serialization of parameters and results as well as asynchronous completion transparently. Since the underlying communication channel is a point-to-point connection, an extension is necessary to communicate with multiple domains. Therefore, we introduce a *router* component running on a Linux domain. Figure 7 shows an example architecture of multiple bare-metal domains where two clients on domains X and Y access services running in domains A and B.

It is also possible to establish multiple RPMsg connections per domain. However, the router component allows to enforce communication rules, for example for security purposes. It also opens up a possibility to monitor system behavior based on communication patterns. Using SOP, it is easy to build service-oriented systems which are split in multiple guest domains on a low level, up to distinct domains for each available service.

F. Real-time monitoring support

Monitoring is a common technique to verify if a component is operating in a specified environment. Especially for safety-critical functions, monitoring the execution state helps to improve the reliability of such a system [17]. The basic idea is to define valid execution states of a software component during development. At execution time, a component called *monitor* verifies that the application as well as the environment are within the specified, valid states of execution at all time. If a violation is detected, the system designer has multiple options to take action depending on the critical of the failing component:

- Reporting: In case a misbehavior of a component is not critical to the system or the user, reporting the incident to the developer can be sufficient.
- Rollback: If the system was previously updates to a newer version of the component, returning to a previously known-good version can solve the issue. The known-good version can optionally be kept running (*hot-standby*) after an update to be able to perform a faster rollback.
- Degraded functionality: If feasible, the system can enter a state of degraded functionality where the violated state of execution does not occur. Manual intervention is required to return to a state of full functionality.

It can easily be seen that hypervisors fit very well into the monitoring concept. When large applications are split into multiple domains, it is very easy to replace single components (i.e. domains) to perform updates or rollbacks. However, monitoring on functionclass can be difficult to achieve under safety–critical conditions, as the performance impact can not be neglected [18]. Thus, we propose monitoring on domain level, where IDC takes place. Figure 8 shows an example architecture for a dedicated partition monitoring another partition.

The router component of SOP (Section III-E) is dynamically configured to forward specific requests to be monitored, as defined by the monitor partition, to said monitoring partition. The original communication remains unchanged and the communication partners are not influenced by the monitoring. This approach is limited to monitoring of the state and parameters which are exchanged during communication, as internal values of each component can not be observed. However, this limitation is less significant in cases where applications are divided into domains on a very low level of granularity, where monitoring takes place close to the class or module level. The impact on communication performance can be minimized by configuring the hypervisor to execute the monitor on a separate processor. As a copy of the message data is created, memory bandwidth is the primary performance factor then.

IV. Evaluation

The performance of our approach is evaluated in this chapter. First, the rount-trip latency is measured using synthetic

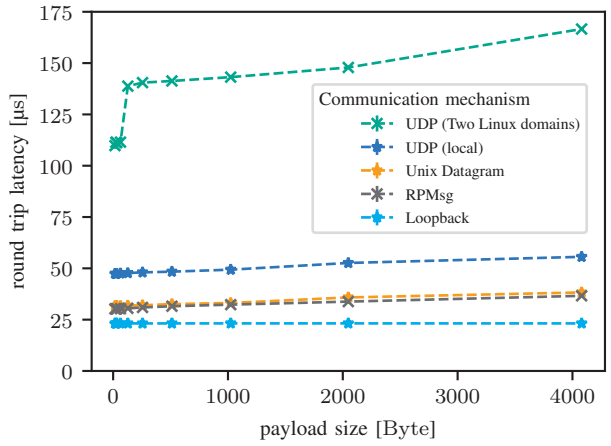

Fig. 9. Round trip latency over payload size, compared to other common communication mechanisms

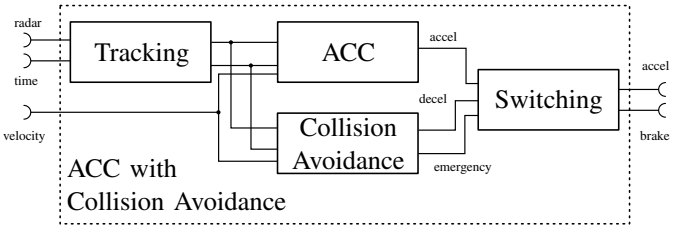

Fig. 10. Block diagram of the Adaptive Cruise Control example use case

TABLE II
ACC PERFORMANCE COMPARISON (SINGLE APP/SPLIT DOMAINS)

Application type	1 step	1000 steps	ratio
monolithic application	6 µs	5243 µs	~873
split on block level	4036 µs	4313 µs	~1.069
resulting overhead factor	~672	0.823	

benchmarks and compared to common alternatives. Second, we show a real-world use case implementing an Adaptive Cruise Control (ACC) algorithm splitted across several guest domains.

A. Round-trip latency

The test setup is as follows: an unprivileged bare-metal guest is connected to a Linux guest. The bare-metal guest is configured to return incoming messages unmodified as soon as they are received. On the Linux guest, we use the `benchmark` library by Google [19] to run tests of different block sizes, 10000 iterations each. Figure 9 shows the results and compares them with other common approaches. Note that the *Unix Loopback* and *TCP Loopback* cases do not involve data transmission between different guest domains. These cases use loopback mechanisms within the Linux guest only and are provided for reference.

We can show that the full round-trip latency is about $48\mu s$ for 512 byte payload. Due to the zero-copy impementation, latency does not increase greatly for even larger payloads. UDP communication set up between two Linux guest domains for comparison has at least $110\mu s$ round-trip latency. With the additional overhead of a second Linux guest compared to our bare-metal domain in mind, it can be seen that our approach is faster by a factor of $x2.3$. Additionally, we discovered that UDP communication between applications on the same Linux guest using the loopback interface is even slower than our IDC approach and on par with Unix datagram sockets on the same system.

B. Adaptive Cruise Control use case

Figure 10 shows a block diagram of a ACC algorithm used in the automotive sector. Each block is realized as a bare-metal guest domain implementing the respective function. All bare-metal guests are connected to one central Linux domain running the router application of our service-oriented communication framework discussed in section III-E. This leads to a number of domains for this use case: the *Tracking*, *ACC*, *Collision Avoidance (CA)* and *Switching* domains representing blocks in the data flow diagram, an unprivileged Linux guest running the router component and an additional bare-metal domain to create the test stimuli.

We compare the duration of a full run of the ACC algorithm using inter-domain communication as described with a monolithic version including all modules in a single bare-metal application. Table II shows the average duration of both a single iteration of the ACC algorithm and 1000 iterations of the same step to illustrate increased algorithmic complexity. Total runtime is measured starting with the first call into the *tracking* service until final values are returned by the *switching* service.

We observe a very fast execution of a single step in the monolithic application due to the low complexity of the algorithm. When executing 1000 steps at once, the runtime increases as expected, although slightly lower due to cache effects and measurement overhead. The single step for a split application is much slower, which can be explained by the round trip latency and scheduling impact. Each call to the four guest domains adds a round-trip of ~50 µs, which does contribute to the total runtime. The remaining overhead is caused by scheduling of the different guest domains, as no fixed schedule is used. The high standard deviation of 640 µs matches this explaination. However when running 1000 iterations per step, we can even observe lower runtime than with the monolithic application and only low additional overhead compared to a single iteration. The split implementation incidentally executes the ACC and CA services in parallel, presumed the hypervisor schedules both guests on different CPUs. This is a welcome side effect of the service-oriented communication framework. We can conclude that the feasibility of splitting an application across domains depends on the algorithmic complexity. The round-trip time is almost independent of the payload size and should not be significantly larger than the runtime of the algorithm.

V. Conclusion

In this work, we introduce a mechanism for communcation between unprivileged guests on the Xen hypervisor. The design makes use of multiple standard technologies, as VIRTIO, RPMsg and the `xenbus`, where existing driver implementations can be re-used. While these technologies introduce multiple layers in software, the run-time overhead on latency is very low. Since SHM is used for data exchange, security benefits of the hypervisor through memory isolation remain, except for domains sharing a communication channel. We also introduced an example use case for this IDC mechanism, a simple service-oriented abstraction layer. It allows to separate large applications into fine-grained guest domains providing individual services, profiting from communication between guests with very low latency. Additionally, we discussed how to implement networks on top of the peer-to-peer channel, allowing to monitor and enforce communication behaviour. Compared to standard mechanisms like paravirtualized ethernet network, our approach allows for fast communication at the expense of flexibility with respect to interoperability with existing network protocols. This is a promising approach to implement extensively partitioned applications on hypervisor-based systems with mixed-critical workload. Especially large applications which profit most from partitioning can be enhanced using this IDC approach, similar to the ACC use-case presented in this work.

References

[1] M. Broy, "Challenges in automotive software engineering," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 33–42. [Online]. Available: https://doi.org/10.1145/1134285.1134292

[2] S. Fürst, "Challenges in the design of automotive software," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, 2010, pp. 256–258.

[3] 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, 52nd CIRP Conference on Manufacturing Systems (CMS), Ljubljana, Slovenia, June 12-14, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212827119306365

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 164–177. [Online]. Available: https://doi.org/10.1145/945445.945462

[5] S. Alonso, J. Lázaro, J. Jiménez, L. Muguira, and A. Largacha, "Analysing the interference of xen hypervisor in the network speed," in *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)*, 2020, pp. 1–6.

[6] D. Reinhardt, M. Güntner, M. Kucera, T. Waas, and W. Kühnhauser, "Mapping can-to-ethernet communication channels within virtualized embedded environments," in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2015, pp. 1–10.

[7] O. Sander, T. Sandmann, V. V. Duy, S. Bähr, F. Bapp, J. Becker, H. U. Michel, D. Kaule, D. Adam, E. Lübbers, J. Hairbucher, A. Richter, C. Herber, and A. Herkersdorf, "Hardware virtualization support for shared resources in mixed-criticality multicore systems," in *Design, Automation Test in Europe (DATE)*, 2014, pp. 1–6.

[8] S. Xi, C. Li, C. Lu, and C. Gill, "Prioritizing local inter-domain communication in xen," in *2013 IEEE/ACM 21st International Symposium on Quality of Service (IWQoS)*, 2013, pp. 1–10.

[9] P. Lucas, K. Chappuis, B. Boutin, J. Vetter, and D. Raho, "Vosysmonitor, a trustzone-based hypervisor for iso 26262 mixed-critical system," in *2018 23rd Conference of Open Innovations Association (FRUCT)*, 2018, pp. 231–238.

[10] A. Oliveira, J. Martins, J. Cabral, A. Tavares, and S. Pinto, "Tz- virtio: Enabling standardized inter-partition communication in a trustzone-assisted hypervisor," in *2018 IEEE 27th International Symposium on Industrial Electronics (ISIE)*, 2018, pp. 708–713.

[11] D. Reinhardt, M. Güntner, M. Kucera, T. Waas, and W. Kühnhauser, "Mapping can-to-ethernet communication channels within virtualized embedded environments," in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2015, pp. 1–10.

[12] H. Fayyad, L. Perneel, and M. Timmerman, "Full and para-virtualization with xen: A performance comparison," *Journal of Emerging Trends in Computing and Information Sciences*, vol. Volume 10, pp. 719–727, 09 2013.

[13] "Virtual i/o device (VIRTIO) version 1.1." [Online]. Available: https://docs.oasis-open.org/virtio/virtio/v1.1/cs01/virtio-v1.1-cs01.pdf

[14] Xen Project. XenStore paths. [Online]. Available: https://xenbits.xen.org/docs/4.13-testing/misc/xenstore-paths.html

[15] ——. xl.cfg - xl domain configuration file syntax. [Online]. Available: https://xenbits.xen.org/docs/4.13-testing/man/xl.cfg.5.html

[16] OpenAMP Project. RPMsg messaging protocol. [Online]. Available: https://github.com/OpenAMP/open-amp

[17] Y. Bebawy, H. Guissouma, S. V. Maelen, J. Kröger, G. Hake, I. Stierand, M. Fränzle, E. Sax, and A. Hahn, "Incremental contract-based verification of software updates for safety-critical cyber-physical systems," in *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2020, pp. 1708–1714.

[18] L. Gao, M. Lu, L. Li, and C. Pan, "A survey of software runtime monitoring," in *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, 2017, pp. 308–313.

[19] Google, Inc. Google benchmark – a microbenchmark support library. [Online]. Available: https://github.com/google/benchmark