Proceedings of the Fourth HPI Cloud Symposium "Operating the Cloud" 2016

Stefan Klauck, Fabian Maschler, Karsten Tausche

Technische Berichte Nr. 117

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 117

Stefan Klauck | Fabian Maschler | Karsten Tausche

Proceedings of the Fourth HPI Cloud Symposium "Operating the Cloud" 2016

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2017
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Druck: docupoint GmbH Magdeburg

ISBN 978-3-86956-401-2

Zugleich online veröffentlicht auf dem Publikationsserver der Universität Potsdam:
URN urn:nbn:de:kobv:517-opus4-394513
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-394513

Preface

Every year, the Hasso Plattner Institute (HPI) invites guests from industry and academia to a collaborative scientific workshop on the topic ***Operating the Cloud***. Our goal is to provide a forum for the exchange of knowledge and experience between industry and academia. Co-located with the event is the HPI's Future SOC Lab day, which offers an additional attractive and conducive environment for scientific and industry related discussions. ***Operating the Cloud*** aims to be a platform for productive interactions of innovative ideas, visions, and upcoming technologies in the field of cloud operation and administration.

On the occasion of this symposium we called for submissions of research papers and practitioner's reports. A compilation of the research papers realized during the fourth HPI cloud symposium ***Operating the Cloud*** are published in this proceedings. We thank the authors for exciting presentations and insights into their current work and research. Moreover, we look forward to more interesting submissions for the upcoming symposium later in the year.

Contents

Scalability, Availability, and Elasticity through Database Replication in Hyrise-R

Stefan Klauck

Enterprise Platform and Integration Concepts
Hasso Plattner Institute
stefan.klauck@hpi.de

The growing analytical demand increases the importance of scalability and elasticity for mixed workload in-memory databases. Data replication is a way to cope with the growing demand and entails increased availability. In this paper, we describe different replication mechanisms, balancing query performance and availability. In addition, we outline how we implemented the cloud-ilities scalability, availability, and elasticity in Hyrise-R, a replication extension of the in-memory database Hyrise. Finally, we summarize further current research activities within the Hyrise project, i. e., data tiering, self-adaption and non-volatile RAM.

1 Introduction

Increasing main memory sizes and parallelism in computer architectures boost the development of in-memory database systems, i. e., systems that store and process data solely in main memory. Some of these systems, e. g., SAP HANA [3], HyPer [7], Peloton [1], and Hyrise [6], are well suited for enterprise workloads, consisting of transactional and analytical queries. A growing number of users and an increasing demand for enterprise applications can saturate or even overload single-node database systems at peak times. Better performance can be achieved by improving a single machine's hardware, but it is often cheaper and more practicable to follow a scale-out approach and use additional machines [2]. Kruger et al. [8] showed in an analysis of a modern enterprise system that more than 80 % of OLTP and more than 90 % of OLAP requests are reading queries. The large amount of reading queries and the possibility to distribute them among several nodes make the concept of database replication desirable for enterprise applications. Data replication has many dimensions, describing which data items are replicated, which nodes can update data items, or how to keep data items in sync.

SAP HANA [11] and HyPer's scale-out version ScyPer [10] propose master replication to cope with the growing OLAP demand. Replica instances can execute reading queries on snapshots in parallel without violating any consistency or isolation requirements. We implemented master replication for the in-memory database Hyrise and call the extension Hyise-R(eplication) [14]. Besides performance, data replication on additional nodes can increase the availability by redundancy in case of failures. An important property for replicated systems is elasticity, i. e., the ability of expanding and shrinking the system dynamically without having to disrupt

its availability. An elastic implementation enables the database to react to changing workloads and resource requirements, and saves costs by running on minimal resources with no violations of service level agreements. This paper contributes:

- A classification of database replication.
- An architectural blueprint of an OLAP scale-out columnar in-memory database.
- A description how we implemented the cloud-ilities scalability, availability, and elasticity in Hyrise-R.

After this introduction, the paper continues with a classification of replication approaches. Following, Section 3 gives an overview of the in-memory database Hyrise and current related research topics. We describe the implementation of scalability, availability, and elasticity in Hyrise-R in Section 4. Section 5 concludes this paper.

2 Replication Approaches

Replication approaches can be classified by the way replicas are updated. Gray et al. [5] distinguished two replication models: on the one hand eager and lazy replication, on the other hand master and group replication. In addition, the information how to update replicas can be expressed logically or physically. Finally, we distinguish full and partial replicas.

2.1 Eager vs. Lazy

Eager replication propagates updates to all replicas as part of the transaction. When a transaction is committed, it is executed on every replica atomically. Following, all data items in the cluster, i. e., on all nodes, are at the same state after the end of a transaction. The performance to keep replicas in sync is important, because it directly influence the transaction latencies for eager replication. In contrast, lazy replication postpones the updates of replicas. The propagation of changes to the other nodes is handled asynchronously. Lazy replication delivers better transaction latencies than eager approaches, because it does not wait to return until all nodes are synchronized. In addition, lazy replication can optimize the communication between master and replica nodes by combining the log information of multiple transactions in a single message without sacrificing transacting latencies.

2.2 Master vs. Group

Master and group replication differentiate where data-altering transactions can be issued. The first approach allows them only on a dedicated node, called master or primary, which is responsible to propagate changes to the other nodes, called

replicas or respectively secondaries. Contrary to master replication, group replication, a so-called update everywhere strategy, allows writing queries on every node and propagates the changes from there to the other nodes. Resulting, there is no designated master node within the cluster. Master replication avoids coordination between nodes for transaction handling. However, master replication cannot scale beyond on node. To achieve highest transaction rates for workloads with little conflicts, group replication is necessary. Partitioning and clever workload distribution can avoid transactional conflicts between nodes by assigning update responsibilities to nodes based on key ranges and route queries accordingly.

2.3 Logical vs. Physical Updates

The kind of information how to update replica instances can be logical or physical. Logical logging describes updates on higher level, such as SQL statements, whereas physical logging provides information on lower level with regards of the used data structures, for example specifying the offsets where to insert or change values. The size of physical logs depends on the amount of changed data, but the speed of replaying it is usually faster than for logical logs, because the queries do not have to be reexecuted [16].

2.4 Full vs. Partial Replicas

The majority of analytical requests query a limited set of tables and attributes. Partial replicas only hold copies of frequently accessed data. This allows them to answer frequent analytical queries while not requiring the capacity that a full copy of the database would. Also, they can be used to store highly specific indices, such as full-text search indices, which are not created on the master node for capacity reasons.

3 Hyrise

This section introduces the database project Hyrise, describes its read-only scale-out extension Hyrise-R, and sketches a columnar database blueprint with extensions we are currently researching.

3.1 Overview

Hyrise is an open source in-memory database research project, initiated by the investigation of an optimal storage layout for data records. On the one side, a columnar arrangement is well suited for attributes which are often accessed sequentially, e. g., via scans, joins, or aggregations. On the other side, attributes accessed in OLTP style queries, e. g., projections of few tuples, can be stored in a row-wise manner. Hyrise supports flexible hybrid table layouts, i. e., storing

attributes corresponding to their access patterns to optimize cache locality [6]. Over time, many in-memory database concepts were developed, evaluated, and integrated into Hyrise (see Figure 1).

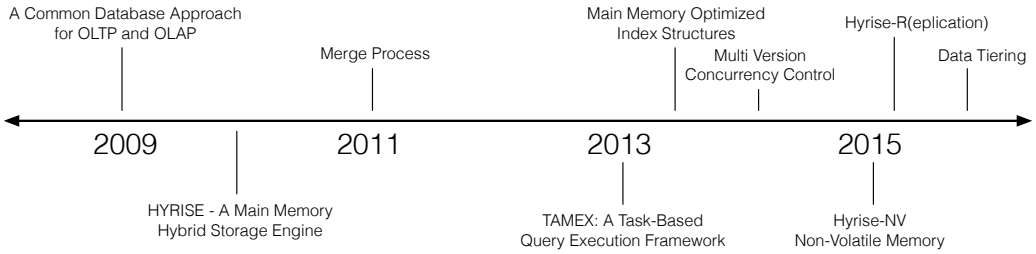

Figure 1: Hyrise research history

By exploiting a main-delta architecture [8], Hyrise is well suited for mixed workloads. Tuples in the main partition are stored dictionary compressed with a sorted dictionary. This allows efficient vector scanning and supports range queries without decoding complete columns. Data modifications are inserted in the write-optimized delta partition. Using an unsorted dictionary for the delta is a trade-off for better write and reasonable read performance. The periodic merge process moves tuples from the delta to the main partition. Hyrise exploits an insert-only approach and multi-version concurrency control with snapshot isolation as default isolation level [13]. That is why, Hyrise can process writing queries without delaying reading ones. Hyrise features a task-based query execution framework to execute dynamically arriving transactional queries in a timely manner, even while complex analytical queries are executed [17]. Current Hyrise research activities focus on leveraging non-volatile(NV) RAM [12, 15], dynamic and transparent data tiering [9], and scale-out systems [14].

3.2 Hyrise-R Overview

Hyrise-R [14] is a scale-out extension for Hyrise, which implements master replication. Figure 2 shows the architecture of a Hyrise-R cluster, consisting of a query dispatcher, a single Hyrise master node, and an arbitrary number of replicas. Users submit their database requests to a query dispatcher, which acts as a load balancer for reading queries. The dispatcher parses the queries for data-altering operations, i. e., inserts, updates, and deletes. The master node processes all writing transactions. Data changes are written into a local log, implemented as a ring buffer [13]. Besides storing the log entries to persistent memory, the master sends them to the replicas, which update their data accordingly.

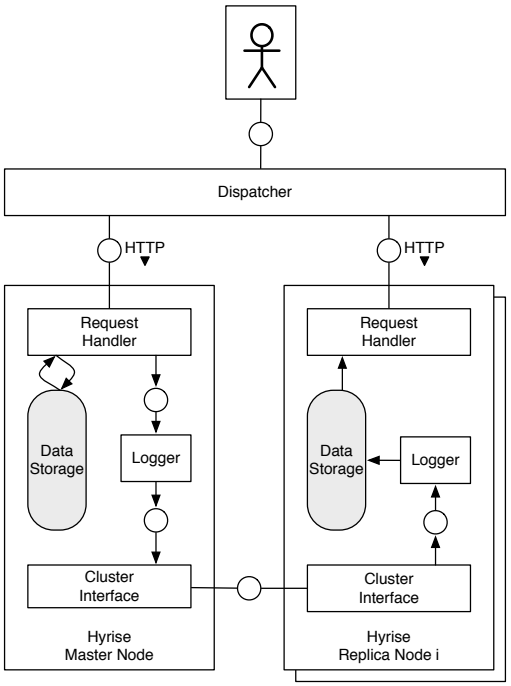

Figure 2: Hyrise-R architecture

3.3 OLAP Scale-Out DB Blueprint

Besides replication, there are related and orthogonal concepts to improve in-memory databases. In this section, we summarize in-memory data management concepts we investigate at the Hasso Plattner Institute. Further, we show how these concepts fit into Plattner's blueprint [11] of a columnar in-memory database (see Figure 3).

- Distributed Log. A distributed log encapsulates data durability, fault tolerance, and the replica synchronization in a dedicated cluster of nodes [4]. Only the master writes to the distributed log, which stores and replicates database logs and checkpoints for k-safety. Replicas decide how quickly they are synchronized (pull the newest log entries or get the newest log entries pushed). In this way, full replicas can synchronize more often than partial replicas, reducing the overall number of messages in the network. The distributed log can exploit NVRAM to store the newest log entries to speed up writing queries of the master node.

- Data Tiering. Data tiering is based on the idea of identifying, reorganizing, and splitting tables into hot and cold data partitions according to access criteria. Hot data comprises relevant data that is required to process the major portion of the workload. Hence, hot data should be stored on fast storage.

Cold data is accessed infrequently and can be stored on slower storage to save costs.

Besides saving costs, we can speed up queries which do not access data on the cold partition(s). One way to do this is letting the application specify explicitly whether a query has to be executed on both (hot and cold) or only the hot partition. Another way is using pruning filters, e. g., implemented by Bloom filters, to let the database system decide whether the cold partition(s) can be skipped [9].

- Self-Adaption. There are many optimization problems when tuning databases, e. g., which indices to create, how to sort and partition data, how often to replicate single data items. In the past, database administrators were responsible for achieving the best database performance. By knowing the workload, internal data organization, and query execution plans, the database system manages enough information to adopt itself.
- NVRAM. Besides the usage of NVRAM as fast persistent memory for logs, we investigate how to exploit NVRAM for in-memory data structures. NVRAM is directly attached to the CPU's memory controller, where it behaves like DRAM. It is expected to come with larger capacities and lower costs than DRAM and could be used to build bigger and cheaper in-memory systems. However, NVRAM will suffer lower bandwidth and higher latencies (especially for writing queries). We investigate a database system in which both types of memory are used side-by-side. Data structures are placed either on faster DRAM or cheaper NVRAM depending on the access frequency and pattern.

4 Cloud-ilities in Hyrise-R

This section explains the implementation of the cloud-ilities scalability, availability, and elasticity in Hyrise-R.

4.1 Scalability

Scalability describes the ability of a system to use additional resources to process more work. There are two common approaches for scaling (database) systems [2]. Scale-up or vertical scaling describes the strategy of adding more resources, such as CPU or memory, to a single-machine setup. Scale-out or horizontal scaling denotes a system extension by adding more machines. Hyrise-R implements scale-out by replicating the data of a Hyrise master node to replica nodes. In the following, we discuss the scalability of read workloads, write workloads, and the communication between cluster nodes.

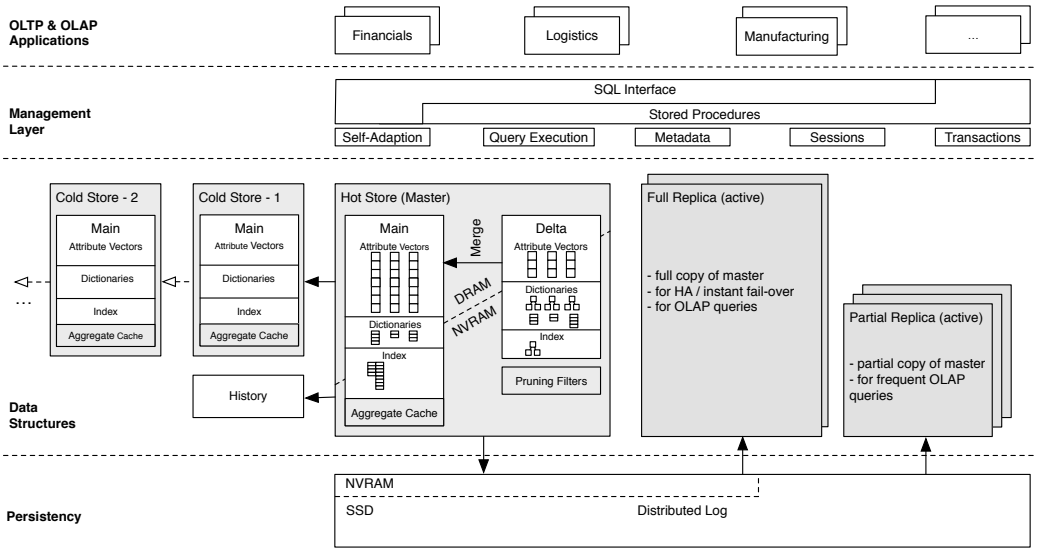

Figure 3: Blueprint of a columnar database

- Reads. Hyrise-R scales linearly for read-only workloads, because reading queries can be routed to arbitrary nodes (master or replica) and executed in parallel on snapshots without violating consistency or isolation.
- Writes. Implementing master replication, the master node processes all writing transactions. Hence, write-only workloads can only be scaled up. For mixed workloads, consisting of reads and writes, the database can be scaled out until no read-only transactions are executed on the master node anymore.
- Communication. Communication is needed between master and replica nodes to keep them in sync. The master node sends log information to the replica nodes. Replica nodes have to acknowledge the reception or must have the possibility to request previous logs to be able to stay in sync. Sending logs of multiple commits into a single message reduces the number of messages and can, thus, improve the throughput, but it can increases transaction latencies. To reduce the latency, it can be sufficient to replicate updates to a subset of cluster nodes eagerly, and lazily to the rest.

  A straightforward messaging approach is using point-to-point connections between the master and each replica. With this approach, the number of messages increases with the number of replicas linearly. Better scalability is promised by using multi-/broadcast communication, peer-to-peer messages, or using a dedicated system to keep the replicas in sync, e. g., a distributed log (see Section 3.3). Broadcast is the most scalable and efficient solution, because a single message is sufficient to keep all replicas in sync. Peer-to-peer networking reduces the number of messages per node, but it can increase the number of hops and, thus, the latency to update replicas. A distributed log can scale independently from the database cluster. In comparison to direct

messages between master and replica nodes, the indirection via the shared log can increase the time until the replicas receive the log.

4.2 Availability

Availability is the measure describing whether the system is operational at a point in time, i. e., the ratio of uptime and lifetime. Availability is reduced in case of a failure which stops the operation of the database system. Failover is the process of detecting the failure and switching to a backup to proceed the service. Depending on the time to fail over and expected response time of a system, the availability from the user's perspective could be 100 percent even in the case of failures.

Schwalb et al. [14] discussed the heartbeat protocol of Hyrise-R to detect failures. In the following, we explain how the choice between eager and lazy replication (see Section 2.1) influences the failover and, thus, the availability. Replica failures can be handled by not using the replica for load balancing anymore. As a result, the remaining nodes have to take over the load of the failed one. In case of a master node failure, a replica has to become the new master. When implementing eager replication, the replicas have always the same transactional state as the master. An arbitrary replica, which is voted, takes over the master's role. Using lazy replication, the replica nodes are either exactly in the same state or almost. When missing the last committed transactions, the log must be readable for the replicas for a fast failover. Using a distributed log supports fast failover for lazy replication.

4.3 Elasticity

Elasticity is the capability to shrink and extend the database cluster depending on the current system load. This property is desirable to optimize resource utilization in order to save costs. Especially in cloud computing, tenants pay for those resources which are allocated for them. Tenants can use a cloud service interface to acquire and release resources directly, or define policies indicating under what condition resources should be scaled automatically. For both monitoring the resource utilization is necessary to adjust the currently allocated cluster resources. Additionally, elasticity requires starting and stopping Hyrise instances on multiple machines, and adding and removing them to a running Hyrise-R cluster.

When starting new Hyrise instances and adding them to the cluster, they have to load the table data with the newest transactional changes. Checkpoints, which are regularly created by the master node, are used as basis for table loads. They reduce the number of log entries which have to be examined by the joining database node. Nonetheless, loading all table data can take a while for large databases. To exploit started Hyrise instances for load balancing before they have loaded the complete data set into their main memory, instances could report their load status, i. e., the completely loaded table data, such as columns and indices, to the dispatcher. If the dispatcher receives a query for which the newly started Hyrise instance already finished loading the related table data, it can propagate the query to the node.

5 Conclusion

This paper presents how data replication implements the cloud-ilities scalability, availability, and elasticity. It classifies and explains replication approaches, describing how and which data is replicated, and how the replicated data is kept in sync. Further, we summarized Hyrise-R, an implementation of full master replication, which uses physical logs to synchronize replica nodes. A distributed log can increase the availability for lazy replication and partial replicas reduce scale-out costs. Besides replication, our future work in the context of in-memory data management focuses on partition pruning, self-adaption, and NVRAM.

6 Acknowledgments

We have received funding from the European Unions Horizon 2020 research and innovation program 2014–2018 under grant agreement No. 644866 (SSICLOPS). This document reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] J. Arulraj, A. Pavlo, and P. Menon. "Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26-July 01, 2016*. 2016, pages 583–598.

[2] D. DeWitt and J. Gray. "Parallel database systems: the future of high performance database systems". In: *Communications of the ACM* 35.6 (1992), pages 85–98.

[3] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. "The SAP HANA Database – An Architecture Overview". In: *IEEE Data Eng. Bull.* 35.1 (2012), pages 28–33.

[4] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengießer, C. Mathis, T. Bodner, and W. Lehner. "Towards Scalable Real-time Analytics: An Architecture for Scale-out of OLxP Workloads". In: *PVLDB* 8.12 (2015), pages 1716–1727.

[5] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. "The Dangers of Replication and a Solution". In: *SIGMOD*. 1996, pages 173–182. DOI: 10.1145/233269.233330.

[6] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden. "HYRISE - A Main Memory Hybrid Storage Engine". In: *PVLDB* 4.2 (2010), pages 105–116.

[7] A. Kemper and T. Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots". In: *27th International Conference on Data Engineering (ICDE)*. 2011, pages 195–206. DOI: 10.1109/ICDE.2011.5767867.

[8] J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. "Fast Updates on Read-Optimized Databases Using Multi-Core CPUs". In: *PVLDB* 5.1 (2011), pages 61–72.

[9] C. Meyer, M. Boissier, A. Michaud, J. O. Vollmer, K. Taylor, D. Schwalb, M. Uflacker, and K. Roedszus. "Dynamic and Transparent Data Tiering for In-Memory Databases in Mixed Workload Environments". In: *ADMS@VLDB*. 2015, pages 37–48.

[10] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. "ScyPer: Elastic OLAP Throughput on Transactional Data". In: *Proceedings of the Second Workshop on Data Analytics in the Cloud*. DanaC '13. ACM, 2013, pages 11–15. ISBN: 978-1-4503-2202-7. DOI: 10.1145/2486767.2486770.

[11] H. Plattner. "The Impact of Columnar In-Memory Databases on Enterprise Systems". In: *PVLDB* 7.13 (2014), pages 1722–1729.

[12] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner. "nvm malloc: Memory Allocation for NVRAM". In: *ADMS@VLDB*. 2015, pages 61–72.

[13] D. Schwalb, M. Faust, J. Wust, M. Grund, and H. Plattner. "Efficient Transaction Processing for Hyrise in Mixed Workload Environments". In: *IMDM@VLDB*. 2014, pages 16–29.

[14] D. Schwalb, J. Kossmann, M. Faust, S. Klauck, M. Uflacker, and H. Plattner. "Hyrise-R: Scale-out and Hot-Standby through Lazy Master Replication for Enterprise Applications". In: *IMDM@VLDB*. 2015, 7:1–7:7. DOI: 10.1145/2803140.2803147.

[15] D. Schwalb, G. Kumar, M. Dreseler, A. S., M. Faust, A. Hohl, T. Berning, G. Makkar, H. Plattner, and P. Deshmukh. "Hyrise-NV: Instant Recovery for In-Memory Databases Using Non-Volatile Memory". In: *21st International Conference of Database Systems for Advanced Applications (DASFAA)*. 2016, pages 267–282. DOI: 10.1007/978-3-319-32049-6_17.

[16] J. Wust, J. Boese, F. Renkes, S. Blessing, J. Krüger, and H. Plattner. "Efficient logging for enterprise workloads on column-oriented in-memory databases". In: *21st ACM International Conference on Information and Knowledge Management (CIKM)*. 2012, pages 2085–2089.

[17] J. Wust, M. Grund, and H. Plattner. "TAMEX: a Task-Based Query Execution Framework for Mixed Enterprise Workloads on In-Memory Databases". In: *43. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*. 2013, pages 487–501.

Scalable and Secure Infrastructures for Cloud Operations

Fabian Maschler, Jan-Henrich Mattfeld, Norman Rzepka

Operating Systems and Middleware Group
Hasso Plattner Institute, University of Potsdam
Fabian.Maschler@student.hpi.uni-potsdam.de,Jan-Henrich.Mattfeld@student.hpi.uni-potsdam.de,Norman.Rzepka@student.hpi.uni-potsdam.de

Within the context of the EU-funded project Scalable and Secure Infrastructures for Cloud Operations (SSICLOPS), we present a virtual private cloud testbed. With the help of nested virtualization, we create various instances running a distributed in-memory database. We adopt a central management tool to display the current cloud status, add or remove instances and define their database roles. We also perform load tests and examine dynamic resource adjustments. This paper includes not only test results and a detailed overview of our test setup, but also insights into the challenges of automatic private cloud deployments

1 Introduction

This paper contains the documentation for the master project Scalable and Secure Infrastructures for Cloud Operations (SSICLOPS) of the Operating Systems and Middleware group at the Hasso Plattner Institute (HPI) in the summer term 2016.

The SSICLOPS master project is part of a same-named EU-wide project[1], which is co-funded by the European Union as part of the Horizon-2020 program. The larger project focuses on techniques for the management of federated private cloud infrastructures, in particular cloud networking techniques within software-defined data centers and across wide-area networks.

Within this context, we present a virtual private cloud testbed. With the help of nested virtualization, we create various instances running a distributed in-memory database. We adapt a central management tool to display the current cloud status, add or remove instances and define their database roles. We also perform load tests and examine dynamic resource adjustments. The project's source code is available on GitHub. [2]

In the following subsections, we describe the technological foundations of the project as well as previous master projects. We then give some insights into the challenges of automatic cloud deployment with OpenStack-Ansible and finally provide a detailed overview of our resulting setup using DevStack. We conclude with test results and suggested future work.

[1] https://ssiclops.eu.
[2] https://github.com/SSICLOPS/openstack-testbed-vm.

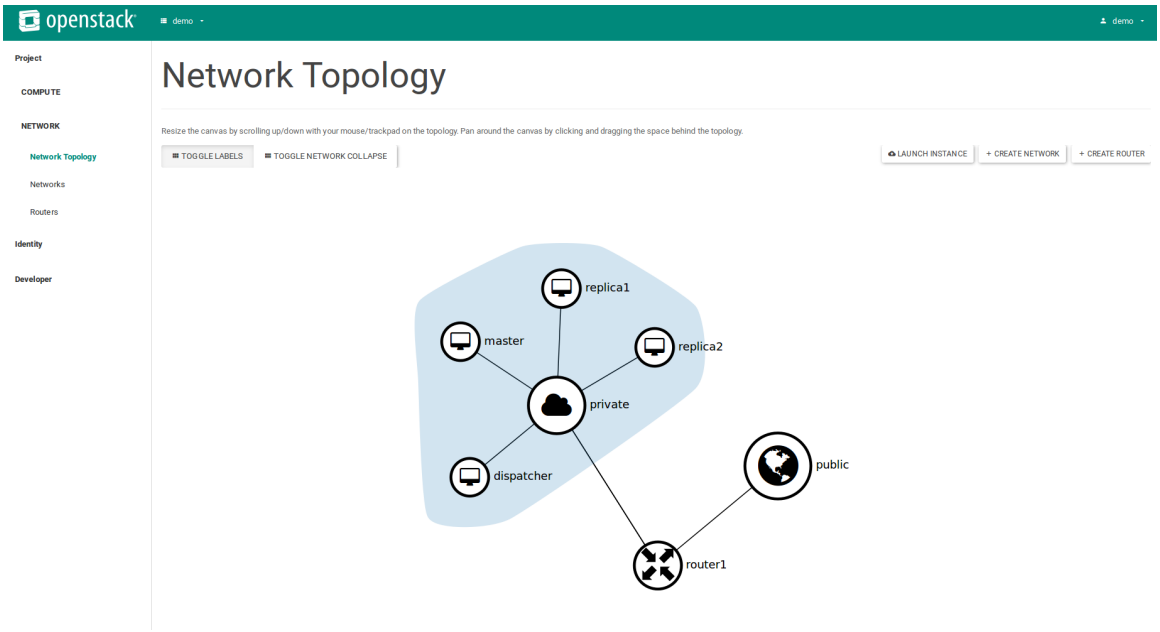

Figure 1: OpenStack Mitaka dashboard, showing a private cloud topology

1.1 OpenStack

OpenStack is a free and open-source Infrastructure as a Service (IaaS) project. It provides several modules to share hardware resources, create software-defined networks, compute nodes and storages. Users can manage the framework via a graphical dashboard (see Figure 1), a REST API or command line tools. OpenStack development is supported by large corporations such as *HPE*, *IBM*, *AMD*, *Intel*, *Canonical*, *Red Hat*, *SUSE* and more.

Today, the OpenStack project is the foundation of multiple large private cloud environments. These include business (e. g. *Open Telekom Cloud*, *Rackspace*) as well as non-profit (e. g. *Wikimedia*) and research (e. g. *CERN*) solutions.

1.2 Hyrise-R

Hyrise is an in-memory research database developed by the Enterprise Platform and Integration Concepts (EPIC) group at Hasso Plattner Institute Potsdam.[3] It shares several features with SAP HANA. These include a delta store, column-based storage, dictionary encoding, several compression techniques as well as an insert-only approach. Hyrise also features a remarkable OLAP performance and optimizations for OLTP tasks [2].

Following the scale-out paradigm, Hyrise-R extends the base project to a cloud-based in-memory solution. Figure 2 shows the architecture consisting of a Hyrise

[3] http://hpi.de/plattner/projects/hyrise.html.

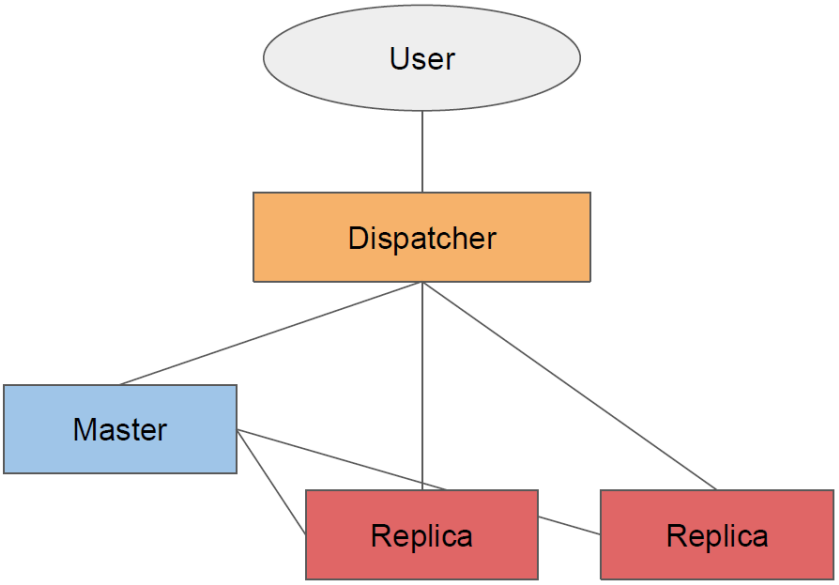

Figure 2: Hyrise-R installation, including a dispatcher, a master node and two replicas

master node, several replicas and one dispatcher [4]. Within SSICLOPS, privacy and security concerns as well as performance and reliability of such setups will be evaluated [1].

We provide an automated base setup of the underlying OpenStack infrastructure to run Hyrise-R and also offer an modified version of the Hyrise-R cluster management tools (shown in Figure 3), compatible with DevStack and the OpenStack Python framework.

1.3 Previous Projects

Former master projects at HPI evaluated the possibilities to automatically deploy a private cloud testbed based on OpenStack Kilo. They conclude that – while still open-source – business solutions like HPE Helion OpenStack[4] are hard to customize and sparsely documented. They focused on evaluating multi-node/multi-region setups, as shown in Figure 4, and therefore excluded basic test frameworks like DevStack [3].

However, they built a foundation for our application-level testbed by introducing nested virtualization and running dependability tests on OpenStack itself [3]. These include crashes of compute and control nodes as well as the object storage. They

[4]http://www8.hp.com/us/en/cloud/hphelion-openstack.html.

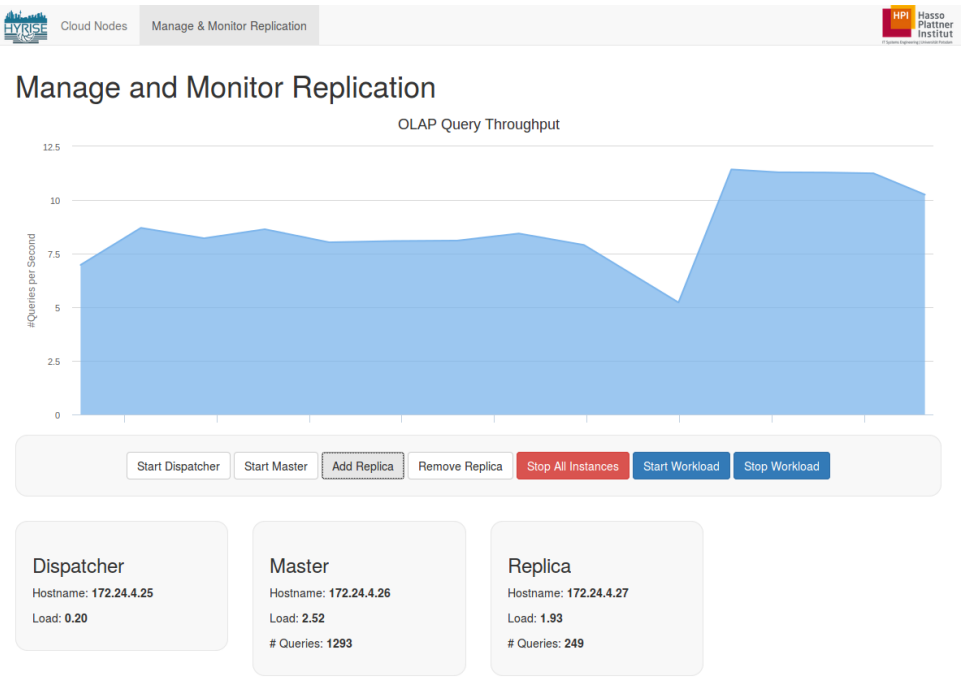

Figure 3: Hyrise-R dashboard, managing replicas and enabling distributed load tests

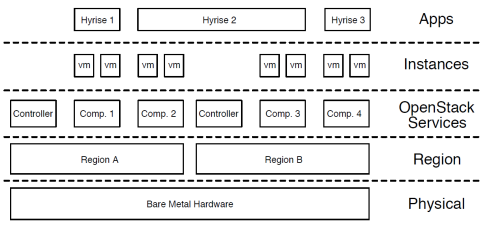

Figure 4: The previous project's OpenStack testbed uses nested virtualization to resemble individual data centers, servers and VMs. Box sizes represent available memory and compute resources [1].

report minor flaws (e. g. a user needs to login again) and the need for additional software (e. g. to check instance and vm statuses) for better reliability.

Hyrise-R is a variant of the in-memory database Hyrise that supports snapshot-based and continuous replication [4]. Keller and Klauck created an infrastructure testbed for Hyrise-R that sets up the dispatcher, master node and replicas with Docker containers. [5] That existing setup includes a web-based cluster management tool to manage the replicas, perform basic load tests and monitor the query performance.

[5]https://github.com/DukeHarris/hyrise_rcm.

2 Findings/Discarded Attempts

The previous projects focused on creating a set of Bash and Ansible scripts for a complex OpenStack setup, including multiple regions. Since then, two major changes occurred:

1. The introduction of OpenStack Mitaka renders the previous work incompatible. Due to its complexity and prototype-like implementation, a migration seems impractical.
2. In contrast, the official OpenStack-Ansible project[6] gained a lot of attention and development.

2.1 OpenStack-Ansible on Ubuntu 16.04

Instead of building on the previous sources, our first attempt was a clean start using the official OpenStack-Ansible playbooks. The project claims to deploy a production-ready OpenStack environment, providing scalability, simple operations and a clear upgrade path. Neither of these claims were fulfilled by previous projects.

Following the all-in-one quick start guide[7], we aimed to setup OpenStack Mitaka on Ubuntu 16.04. However, the new Ubuntu version introduced lots of incompatibilities which are documented in our project wiki. These include – but are not limited to – missing version strings and the introduction of *systemd*:

- Missing Ansible package for Ubuntu 16.04
- *mongodb* service not restarting during setup
- Missing identifier for LXC host creation
- Unsupported locale settings
- Missing *modprobe* commands
- etc.

While we developed workarounds for most of these problems, the number of failing playbook tasks lead us to a downgrade from Ubuntu 16.04 to the previous LTS version 14.04.

2.2 OpenStack-Ansible on Ubuntu 14.04

The all-in-one setup runs flawlessly on Ubuntu 14.04, providing an OpenStack environment containing router, network and running virtual machines (instances). The setup is – in theory – reproducible and we automated it even more by providing a Vagrantfile. However, this solution has some downfalls:

[6] https://github.com/openstack/openstack-ansible.
[7] https://docs.openstack.org/developer/openstack-ansible/developer-docs/.

- Bare metal setup on our workstation (32 cores, 64GB RAM, SSD) takes roughly 2 hrs.
- A reboot requires additional manual steps
- *iptables* rules and network setup are error-prone
- The system is unreliable when starting instances

Despite Ansible's theoretical idempotence, the setup failed occasionally on rebuild. Even though the system is generally working, we look for a more stable and lightweight solution, thus introducing the final architecture built on DevStack in the next section.

3 Final Setup

In our final setup, we use DevStack to deploy a single-node installation of OpenStack and run Hyrise-R instances inside. The images used to create the VMs are provisioned with KVM and a custom script, thus can be loaded in OpenStack using the QCOW2 format. In this section we describe how the basic installation, shown in Figure 5, works. Detailed setup instructions can be found in our GitHub repository's readme file and the wiki.[8]

3.1 Image Creation

Having a single-node deployment of OpenStack given by DevStack, we created images to run the example application Hyrise-R. To prepare base images, we created a provision script for each of the following VM types:

- Hyrise-R – Hyrise Instances
- Hyrise-R Dispatcher – Orchestration
- Hyrise-R Clustermanager – Dashboard and Benchmark

As base image we use an image from Ubuntu which is prepared for cloud usage.[9]. We worked with a custom image as well and did the cloud-init configuration manually.[10]. It turns out that we achieve the same result more easily with the prepared image.

A problem arising from all available base images is the minimum size of 40 GB. It is very difficult to shrink the virtual disk size, even the actual usage (after provisioning) is less than 3 GB. We circumvented this problem with increasing the

[8] https://github.com/SSICLOPS/openstack-testbed-vm.
[9] https://cloud-images.ubuntu.com/vagrant/trusty/current/.
[10] http://docs.openstack.org/image-guide/openstack-images.html.

disk size of the DevStack machine to 800 GB, even though the host's disk is smaller. Since our images will never exceed their available disk space the virtually reserved space within OpenStack does not cause any trouble.

Some dynamic information, such as IP addresses of the Hyrise-R master and dispatcher, cannot be included in the image at provision time. Thus, we need to connect to a newly started image in OpenStack and inject this information. Both, OpenStack and cloud-init, do provide interfaces for that. Unfortunately, we could not see positive results for any of the following methods:

- Shell script given through the Horizon dashboard
- Cloud-init script[11]
- *user_data* parameter in commandline tools and REST API

Considering these issues, we decided to assign a floating IP during instance creation in the Hyrise RCM dashboard application (see subsection 3.3) and inject the dynamic information over ssh.

3.2 DevStack

DevStack assists in setting up an OpenStack development and testing installation. It builds directly from source and provides command-line tools for system operations and maintenance. In the original OpenStack project, it is an essential part for integration testing of the various modules. However, it can also be used to experiment with different OpenStack configurations.

While previous groups discarded DevStack because of the lacking multi-node capabilities, we believe it fits this project's needs very well. The default single-node installation of DevStack provides the following services:

- Dashboard (horizon)
- Compute (nova)
- Networking (neutron)
- Identity (keystone)
- Image Service (glance)
- Object Storage (swift)
- Block Storage (cinder)
- Orchestration (heat)

[11] http://cloudinit.readthedocs.io/en/latest/topics/examples.html.

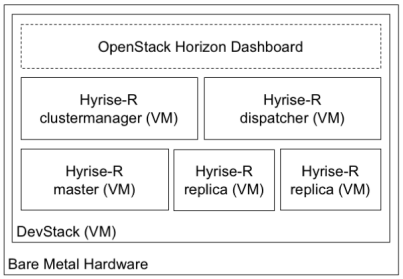

Figure 5: The components of the Hyrise-R installation run in separate VM instances which are nested in a DevStack VM.

In addition to the original DevStack sources[12], we built a Vagrantfile and an Ansible playbook to automate the deployment even more. The command *vagrant up* is then enough to create an OpenStack node, exposing the horizon dashboard to the local host, as well as ssh connections to possible instances. These can easily be created using the Python command line tools.

The setup requires *Vagrant*, *Ansible* and the Python packages *python-novaclient*, *python-neutronclient* and *python-openstackclient*. Moreover, NAT should be enabled on the host to allow internet access to the instances. We use *libvirt* to virtualize the DevStack VM. In contrast to VirtualBox it supports nested hardware-accelerated virtualization that greatly speeds up DevStack instances. A complete rebuild on the latest OpenStack sources now takes only 15 minutes and results in the nested VM architecture shown in Figure 5.

3.3 Hyrise-R

Hyrise-R [4] is a variant of the in-memory database Hyrise that supports replication. A cluster installation of Hyrise-R consists of a single master node that can be used for reads and writes, several read-only replica nodes and a dispatcher that distributes the queries. Every node has to register itself with the dispatcher. Therefore, each node needs to know the address of the dispatcher. Likewise the replica nodes have to register themselves with the master node.

The master node is the only node that accepts writes. It passes changes to the database asynchronously on to connected replicas via a distributed log. Hence, the replica nodes can be used for fulfilling read queries and therefore to greatly speed up the entire system.

We modified the Hyrise-R clustermanager[13] for OpenStack. Each component of the application (i. e. clustermanager, dispatcher, master and replica nodes) is

[12]https://github.com/openstack-dev/devstack.
[13]https://github.com/DukeHarris/hyrise_rcm.

deployed in an individual VM instance. Dispatcher, master and replica nodes can be launched through the graphical user interface of the clustermanager. The clustermanager will automatically assign the correct IP addresses, so that the components can communicate effectively. Internally, the clustermanager spawns new instances using the OpenStack Python framework and configures the VMs using ssh.

For Hyrise-R to run inside an OpenStack VM instance, it needs to be compiled for the correct architecture requirements. We found that the virtual CPUs inside a nested *libvirt* VM only support a subset of the instructions the host CPU might support. Therefore, we compile the application with the following *GCC* compiler flags:

-march=x86-64 -mcx16 -msahf -mno-movbe -mno-aes -mno-pclmul -mpopcnt -mno-abm -mno-lwp -mno-fma -mno-fma4 -mno-xop -mno-bmi -mno-bmi2 -mno-tbm -mno-avx -mno-avx2 -mno-sse4.2 -mno-sse4.1 -mno-lzcnt -mno-rtm -mno-hle -mno-rdrnd -mno-f16c -mno-fsgsbase -mno-rdseed -mno-prfchw -mno-adx -mfxsr -mno-xsave -mno-xsaveopt –param l1-cache-size=32 –param l1-cache-line-size=64 –param l2-cache-size=4096 -mtune=generic

We conducted simple load testing benchmarks on the system. As shown in Figure 3, we can clearly observe an increase in query throughput when adding an additional replica to the setup.

4 Conclusion

In this paper, we presented a virtual private cloud testbed. We documented how to reproducibly set up such an environment. Also, we demonstrated that a distributed application, such as Hyrise-R, can be effectively deployed and studied in our presented testbed.

We used DevStack to implement our testbed, because it is lightweight and easy to install. However, it only supports single-host installations. In future work, it could be interesting to expand this setup to a multi-host installation. For multi-host installations we recommend using OpenStack-Ansible, because it is fairly well documented and supported.

To test our virtual private cloud testbed, we created a Hyrise-R cluster that serves as sample application. We implemented tools that allow the scaling of the application by adding and removing replicas, by modifying previously existing tools. Our experiments show that adding new replicas to a cluster greatly improves the query throughput of the database.

As the result of our project, we present a tool for developing and studying distributed applications in cloud enviroments. This will be useful for further research in the SSICLOPS EU project. Detailed installation instructions as well as more

information on the discarded attempts can be found in the GitHub repository's readme file and the wiki.[14]

In future work, usability improvements such as reboot durability of the testbed could be added. Also, multi-host or multi-region installations and reliability features would make the testbed more similar to a production-like environment. In future implementations, more lightweight VM images could be used to enable faster boot times.

References

[1] F. Eberhardt, J. Hiller, O. Hohlfeld, S. Klauck, M. Plauth, A. Polze, M. Uflacker, and K. Wehrle. *Design of Inter-Cloud Security Policies, Architecture, and Annotations for Data Storage*. Technical report. Hasso Plattner Institute, University of Potsdam, 2016.

[2] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. "HYRISE: a main memory hybrid storage engine". In: *Proceedings of the VLDB Endowment* 4.2 (2010), pages 105–116.

[3] N. K. J. Eschrig S. Knebel. *Dependable Cloud Computing with OpenStack*. Master Project Report. 2015.

[4] D. Schwalb, J. Kossmann, M. Faust, S. Klauck, M. Uflacker, and H. Plattner. "Hyrise-R: Scale-out and Hot-Standby through Lazy Master Replication for Enterprise Applications". In: *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*. ACM. 2015, page 7.

[14] https://github.com/SSICLOPS openstack-testbed-vm.

dOpenCL – Evaluation of an API-Forwarding Implementation

Karsten Tausche, Max Plauth, Andreas Polze

Hasso Plattner Institute, University of Potsdam, Germany
karsten.tausche@student.hpi.de,max.plauth@hpi.de,andreas.polze@hpi.de

Parallel workloads using compute resources such as GPUs and accelerators is a rapidly developing trend in the field of high performance computing. At the same time, virtualization is a generally accepted solution to share compute resources with remote users in a secure and isolated way. However, accessing compute resources from inside virtualized environments still poses a huge problem without any generally accepted and vendor independent solution.

This work presents a brief experimental evaluation of employing *dOpenCL* as an approach to solve this problem. *dOpenCL* extends OpenCL for distributed computing by forwarding OpenCL calls to remote compute nodes. We evaluate the *dOpenCL* implementation for accessing local GPU resources from inside virtual machines (VM), thus omitting the need of any specialized or proprietary GPU virtualization software. Our measurements revealed that the overhead of using *dOpenCL* from inside a VM compared to utilizing OpenCL directly on the host is less than 10 percent for average and large data sets. For very small data sets, it may even provide a performance benefit. Furthermore, *dOpenCL* greatly simplifies distributed programming compared to, e. g., MPI based approaches, as it only requires a single programming paradigm and is mostly binary compatible to plain OpenCL implementations.

1 Introduction

Since the emergence of big data in virtually all research and business fields, developments in high performance computing are focusing more and more on data parallel algorithms. For satisfying the resulting demand for processing power, GPUs and accelerators have become much more popular compared to traditional CPU-based approaches. This development is not yet reflected well in the field of parallel and distributed programming paradigms. Software developers are mostly forced to use combinations of techniques (Figure 1). For example, MPI is used to distribute compute calls to multiple machines, whereas locally on each machine, APIs such as OpenCL are required to access compute devices. On the one hand, MPI itself has no means to directly access compute devices. On the other hand, compute APIs such as OpenCL and CUDA do not allow access to remote devices. However, using a combination MPI and a local compute APIs is unnecessarily complex and error prone [8]. Furthermore, MPI requires users to deploy their application code to all compute nodes, which might additionally introduce security risks.

dOpenCL[8] proposes a solution to these problems by extending the original OpenCL standard with means of distributed computing, without requiring any changes of the employed programming paradigm. Applications using dOpenCL

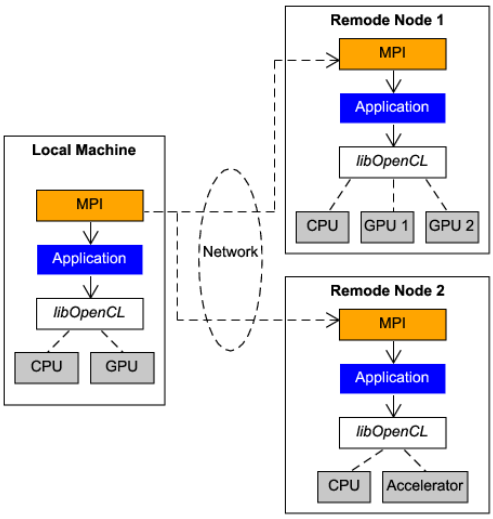

Figure 1: MPI/OpenCL combination

are still deployed on a single machine only, but the underlying dOpenCL implementation allows to execute OpenCL kernels on any OpenCL device available in the network (Figure 2). From a programmers point of view, all available devices appear in a single OpenCL platform and are transparently usable as traditional OpenCL devices.

When combining high performance computing with virtualization, accessing compute devices from within virtual machines (VMs) in a performant and flexible way is still quite difficult to realize. Virtualization solutions are available that directly assign compute devices to the VM[13] using PCI passthrough. However, these are highly specialized solutions that are limited to specific operating systems, platforms or vendors. Furthermore, such solutions generally exclusively lock compute devices for the entire lifetime of the VM, no matter if there is an application running that currently uses it. We found that dOpenCL is quite useful in this situation: The user's application code is deployed only to the VM, so that it is always isolated from the host system. OpenCL kernels, however, can be executed on the host or other compute nodes in the network with little overhead. At the same time, compute devices are only locked as long as they are actually used by an application. This enables utilizing compute devices on demand by a varying number of applications in VMs.

In this paper, we evaluate dOpenCL for accessing GPUs attached to the host from inside a VM. We found that dOpenCL currently is the only implementation that is entirely based on open standards and implementations. Furthermore, our evaluation shows that using dOpenCL leads to little overhead, both in terms of runtime and deployment, compared to a native OpenCL setup.

2 Related Work

Many GPU API-forwarding implementations have been proposed that enable the use of compute APIs from inside virtual machines (VMs). In this section, we compare implementations based on OpenCL and NVIDIA CUDA, as these are the most widely spread compute APIs. GPU API-forwarding implementations generally consist of a front-end and a back-end. The front-end provides access to a compute API within VMs and redirects API calls to the back-end running on the virtualization host.

2.1 CUDA Forwarding Implementations.

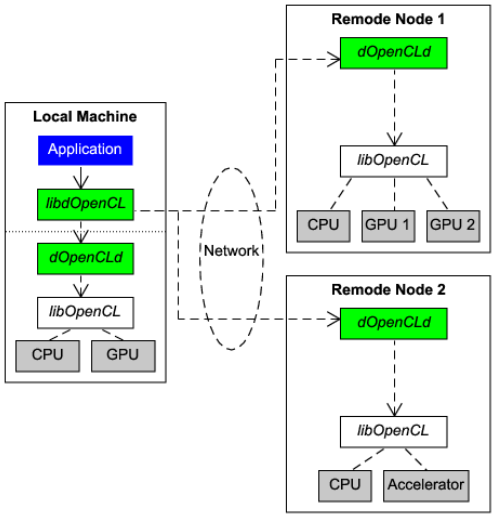

Figure 2: dOpenCL

Figure 3: Distributed computing using a traditional combination of MPI and OpenCL (Figure 1) compared to the new dOpenCL implementation (Figure 2). MPI requires explicit knowledge of all compute nodes and deployment of the application to all nodes. With dOpenCL, the application is deployed only on a single machine. The distribution of OpenCL kernels to compute nodes is handle by the underlying implementation.

CUDA based implementations are only compatible with NVIDIA GPUs, and are thus not portable enough for our objectives. However, these approaches are still interesting for comparison. *GViM*[6] provides CUDA forwarding based on the Xen hypervisor. It additionally implements resource sharing between multiple VMs. Although its experimental implementation is limited to Xen, its approach might be

generalization for other virtual machine monitors (VMMs). *vCUDA*[12] and *gVirtuS*[5] are VMM-independent implementations. Besides simple API-forwarding, vCUDA implements a full GPU-virtualization, including suspend-and-resume, and multiplexing capabilities. All three publications include experimental implementations that provide only a limited set of CUDA functions of outdated CUDA API versions. In contrasts, recent proprietary releases of *rCUDA*[4] provide complete CUDA 8.0 runtime API support. rCUDA was originally intended to provide remote GPU access in HPC clusters, but proved to be efficiently usable for local GPU access from within virtual machines, too[4].

2.2 OpenCL Forwarding Implementations

As OpenCL is less popular compared to CUDA, fewer publication exist that focus on OpenCL API-forwarding. However, the OpenCL API is completely open, which significantly alleviates creating portable implementations. *Hybrid OpenCL* [1] extends OpenCL by a network layer that enables to access remote OpenCL devices without using MPI or any additional RPC protocols. Its experimental implementation, based on a specific Linux-based OpenCL runtime for Intel CPUs, and demonstrated that its networking overhead amortizes for long running compute tasks. *SnuCL*[10] generalizes the OpenCL approaches for heterogeneous small and medium scale HPC clusters. Additionally, it extends the OpenCL API by collective operations, similarly to those included in MPI. The SnuCL runtime makes the computing devices available in a set compute nodes accessible from a single compute host. Its implementation relies on OpenCL kernel code transformations to correctly execute data accesses on remote machines.

Finally, *dOpenCL*[8, 7] generalizes OpenCL for distributed, heterogeneous computing, comparable to SnuCL. However, it introduces a central device manager that distributes available compute devices (on compute nodes) in a network to multiple compute hosts in the network. Compute devices are exclusively locked on demand. This allows to flexibly share a large number of devices with a varying number of users in the network. Our experiments with dOpenCL are based on an unpublished prototype that replaces the proprietary *Real-Time-Framework* (RTF)[14] used in the original implementation by a *Boost.Asio*[11] based implementation. This dOpenCL version does not include the device manager, but it uses the same API-Forwarding implementation as the RTF-based version. In our experiments, we measure the performance of dOpenCL when used in a VM on a single machine. Thus, we do not need the device manager and can rely on the open Boost.Asio based prototype.

3 Concept

In this section we describe the main components, concepts and implemented OpenCL API features of dOpenCL. Furthermore, we present the Rodinia benchmark suite that we use to evaluate the performance of dOpenCL.

3.1 dOpenCL

We follow dOpenCL's original naming convention[8]: *Host* denotes the machine where user applications are running. These applications make use of computing hardware that is part of the *compute nodes*. In our case, the host is a virtual machine, whereas the local bare metal machine is the compute node.

Components. The dOpenCL middleware consists of three components as depicted in Figure 4. Client applications are linked against the dOpenCL *client driver*. This library provides binary compatibility with the OpenCL client libraries, so that any application linked against an OpenCL library can use the dOpenCL client driver without recompilation. Compute nodes in the network need to run the dOpenCL *daemon* in order to be accessible for dOpenCL clients. The daemon sets up an OpenCL context on its local machine to execute OpenCL calls on the actual hardware. Both client driver and daemon rely on the *communication library* that implements network transfers between host and compute node.

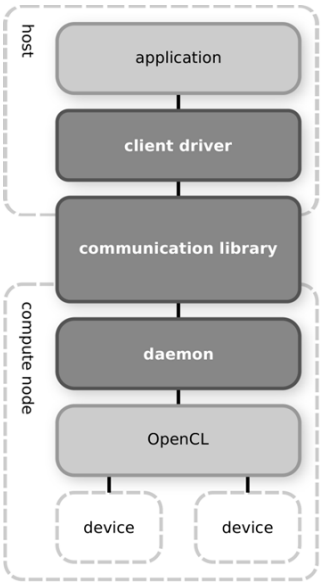

Figure 4: Stack of dOpenCL components. The user application is linked against the dOpenCL-library (client driver) that redirects all OpenCL calls through the communication library to dOpenCL daemons. Image source: [8]

Remote Device Discovery. dOpenCL introduces a central *device manager* that dynamically assigns compute nodes available in the network to applications. Using a device manager for remote device discovery is called *managed mode* in dOpenCL (Figure 5). Initially, dOpenCL daemons register themselves and their local compute devices at the device manager. An application using dOpenCL is configured with a

set of properties that it requires for its OpenCL tasks. Based on this configuration, the client driver requests currently idle compute devices from the device manager (1). The device manager then assigns appropriate devices to the application (2). Relevant compute nodes are now informed of the device assignment (3a), and the list of nodes is sent back to the requesting application. In the last step, the client driver in the application requests its assigned devices from their respective compute nodes (4, 5). On the host side, this process is implemented transparently in the client driver. That way, contexts on remote OpenCL devices are set up in the same way as traditional local OpenCL contexts.

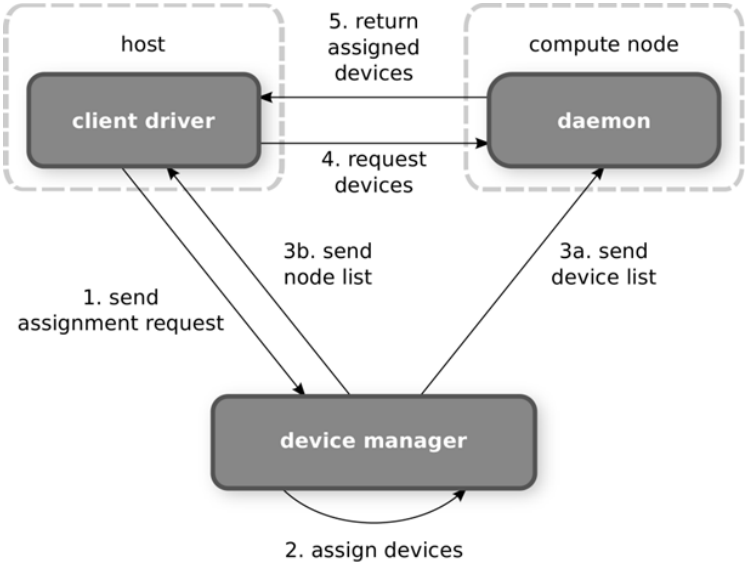

Figure 5: The managed mode allows the client driver to automatically discover available compute nodes in the network. For that, a central device manager exclusively assigns OpenCL devices to clients on demand. Image source: [8]

Besides the managed mode, dOpenCL provides a simpler setup where dOpenCL nodes are configured in a *dcl.nodes* file. This file is comparable to the *hosts* file that is used along with MPI. In this mode, no device manager is used, thus there is also no dynamic assignment of devices to multiple clients. However, for our current experiments, this simpler operation mode suffices. We run a single benchmark application at a time in a virtual machine, and its respective dcl.nodes file refers to the dOpenCL daemon running on the underlying bare metal machine (see also Section 2.2).

Comparison with the OpenCL API. dOpenCL implements a subset of the OpenCL API version 1.2[9]. Currently, it does not implement image and sampler APIs, sub-buffer APIs, vendor extensions and a few other functions. Besides, it is not meant to ever support interoperability with OpenGL or DirectX. Additionally to the OpenCL

API, dOpenCL includes some experimental implementations for collective operations comparable to MPI. When using these function, application code is no longer compatible with OpenCL libraries anymore. However, especially for larger clusters, collective operations can potentially handle many operations more efficiently.

3.2 Rodinia Benchmark Suite

We use the *Rodinia Benchmark Suite*[2] for performance measurements. Rodinia is a set of benchmark applications designed for heterogeneous systems, and provides implementations for OpenCL, CUDA, and OpenMP. For our current tests, we only used the *Gaussian Elimination* benchmark included in the suite. Limiting to a single benchmark generally does not allow for a comparable performance evaluation[3]. However, in our case we only need to measure the overhead that is introduced by redirecting OpenCL calls – the executing hardware is the same, no matter whether the benchmark is started from inside a VM or directly on a bare metal machine.

During our experiments, we noticed a highly different degree of optimization in the set of benchmarks included in Rodinia. It also seems that most benchmarks are optimized primarily for specific NVIDIA GPUs, whereas we were using integrated and dedicated AMD GPUs. Furthermore, the Gaussian elimination benchmark we use for our evaluation shows irregularities that are probably not caused by the hardware or OpenCL/dOpenCL implementation, but rather by the benchmark itself (see Section 4). For our purpose, however, these effects are not critical as we do not need to compare benchmarking results of different compute devices.

4 Evaluation

We evaluate the performance of dOpenCL on a desktop computer equipped with an integrated (AMD Radeon R7 “Spectre”, APU: A10-7870K) and a dedicated (AMD FirePro W8100 “Hawaii”) GPU. The detailed specifications of the test system are denoted in table 1. We performed the Gaussian elimination benchmark included in Rodinia with different matrix sizes to measure the impact of task size on the resulting total runtime. We executed the benchmark on the bare metal machine using plain OpenCL to determine reference runtime. Additionally, we performed the same benchmark from within a KVM-based virtual machine using dOpenCL, with a dOpenCL daemon running on the bare metal machine. We did not succeed in running an applications using dOpenCL on the bare metal machine that is also running the dOpenCL daemon, as this caused deadlocks in the daemon.

In our benchmarks, we observed a generally small overhead introduced by dOpenCL compared to plain OpenCL (Figure 6). This behavior remains largely the same, irregardles of wether the integrated or the dedicated GPU are used. Also, both GPUs demonstrate an exceptional long runtime for a matrix size of $3200 \cdot 3200$ values, both using plain OpenCL and dOpenCL. We assume that this amplitude is issue caused by the interaction between the benchmark (see Section 3.2) and the GPU hardware we employed, but we did not investigate the issue any further.

Table 1: Specifications of the test systems

CPU	AMD A10-7870K (Kaveri)
Memory	2 × 8GB PC3-17066U (DIMM)
Integrated GPU	AMD Radeon R7 Graphics (Spectre)
Dedicated GPU	AMD FirePro W8100 (Hawaii)
Disk	240GB Intel 535 Series (SATA III)
Operating system	Ubuntu Linux 15.10

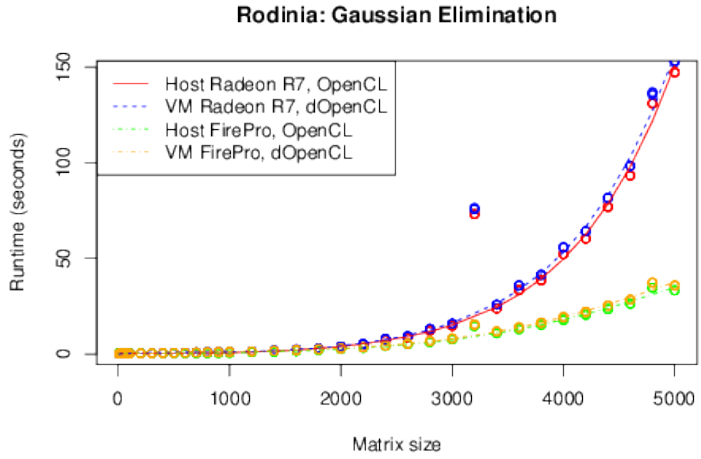

Figure 6: Runtimes of the Gaussian elimination benchmark on the integrated and dedicated GPU, using OpenCL (bare metal) and dOpenCL (VM). Note that the fitted lines do not take into account the high amplitudes at matrix size 3200.

Our measurements revealed, that for very small data set sizes, task execution through dOpenCL may lead to even faster execution times compared to plain OpenCL calls (Figure 7). Firstly, the benchmark running in a VM accesses hardware on the local physical machine, thus the networking latencies should be very low. Secondly, we assume that the dOpenCL daemon caches OpenCL contexts and states, so that consecutive executions of a benchmark may reuse a previously created OpenCL states. Consequently, the benchmark application effectively only initializes a dOpenCL platform and device, which transparently represents OpenCL objects. However, when running the benchmark application using plain OpenCL, it has to initialize the OpenCL platform and context in each run.

For larger data sets, the overhead of dOpenCL compared to plain OpenCL remains constantly small (Figure 8). This is even the case for the high amplitude at a matrix size of 3200. Therefore, it can be assumed that the native execution time of an OpenCL task has little influence on the overhead induced by dOpenCL.

A relative comparison of the runtimes produced by using dOpenCL and plain OpenCL exposes three general trends (Figure 9). Firstly, as already noted, for very small data sets dOpenCL allows for shorter execution times compared to

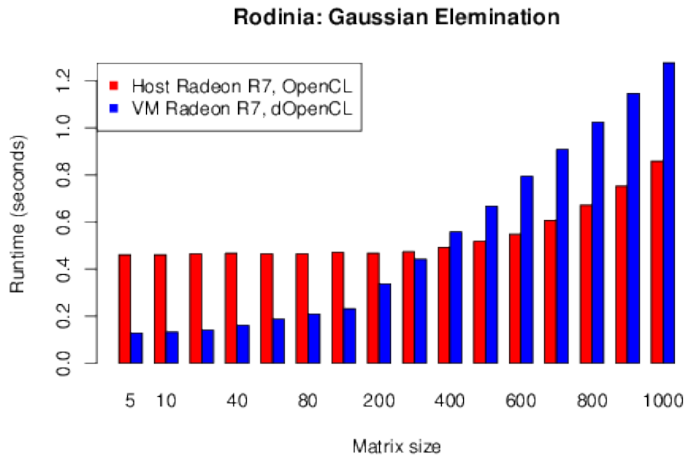

Figure 7: Detailed plot of runtimes for smaller matrices

plain OpenCL. Secondly, for medium sized data sets, the overhead introduced by dOpenCL seems to exceed dOpenCL's initial speedup. In this case, using dOpenCL leads to an approximately doubled runtime in the worst case. Thirdly, for larger data sets, the asymptotic overhead of dOpenCL remains continuously smaller than 10 %.

5 Conclusion

We evaluated dOpenCL as an OpenCL forwarding implementation to utilize OpenCL capable devices from within a local virtual machine. Compared to alternative approaches, dOpenCL is beneficial as it introduces little overhead, both in terms of deployment and in terms of execution time. It does not depend on a specific device vendor or virtualization product and is conceptually independent from a specific operating system. Also, dOpenCL allows for a dynamic assignment of compute devices to applications. Compared to other solutions, compute devices are not linked to specific VMs for the lifetime of the VM.

Compared to plain OpenCL, dOpenCL introduces an asymptotic runtime overhead of less than 10 %, which makes it efficiently usable in productive environments. In the worst case, for a limited range of task sizes, dOpenCL resulted in a roughly doubled runtime. When using dOpenCL, care must be taken to omit this effect for data set sizes and task runtimes that are employed. When spawning a large amount of quickly finishing OpenCL tasks, dOpenCL can be beneficial when used as back-end. Its daemon internally caches OpenCL states, so that the platform and device setup can significantly accelerate a shortly running kernels.

In further studies, the impact of the OpenCL task runtime and size of corresponding data sets could be evaluated separately. Furthermore, in our experiments we

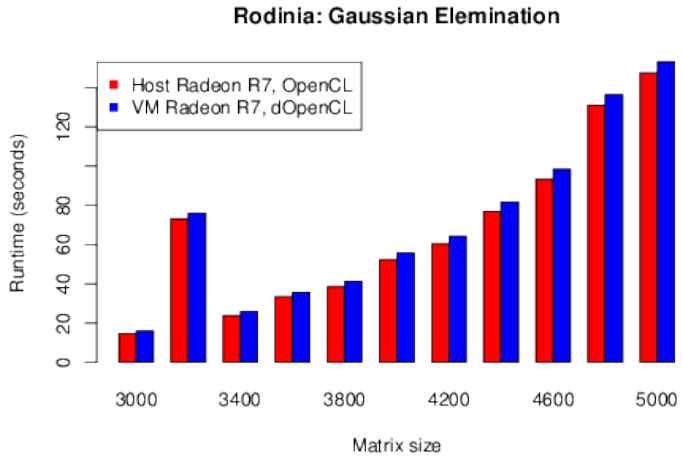

Figure 8: Even for larger data sets, forwarding OpenCL calls using dOpenCL is relatively inexpensive. The high amplitude for a matrix size of $3200 \cdot 3200$ is probably caused by the benchmark implementation.

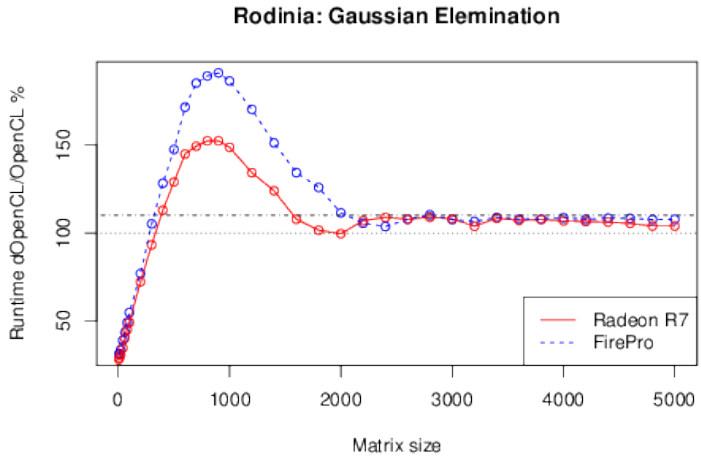

Figure 9: Average quotients of dOpenCL and OpenCL execution times. For very small data sets, dOpenCL is even faster than direct OpenCL calls, probably due to caching of OpenCL contexts in the dOpenCL daemon. For larger data sets, dOpenCL's overhead remains lower than 10 % (dash-dot line).

noticed unexpectedly high runtimes for specific parameter sets, both when using plain OpenCL and dOpenCL. These effects should be evaluated by analyzing the implementations of dOpenCL and Rodinia more thoroughly.

Acknowledgement

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866.

Disclaimer

This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] R. Aoki, S. Oikawa, R. Tsuchiyama, and T. Nakamura. "Hybrid OpenCL: Connecting Different OpenCL Implementations over Network". In: *Computer and Information Technology, International Conference on* 0 (2010), pages 2729–2735.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. "Rodinia: A benchmark suite for heterogeneous computing". In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Oct. 2009, pages 44–54.

[3] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads". In: *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)*. IISWC '10. Washington, DC, USA: IEEE Computer Society, 2010, pages 1–11. ISBN: 978-1-4244-9297-8.

[4] J. Duato, F. D. Igual, R. Mayo, A. J. Peña, E. S. Quintana-Ortı, and F. Silla. "An efficient implementation of GPU virtualization in high performance clusters". In: *European Conference on Parallel Processing*. Springer. 2009, pages 385–394.

[5] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. "A GPGPU Transparent Virtualization Component for High Performance Computing Clouds". In: *Euro-Par 2010 - Parallel Processing: 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part I*. Edited by P. D'Ambra, M. Guarracino, and D. Talia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 379–391. ISBN: 978-3-642-15277-1.

[6] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. "GViM: GPU-accelerated Virtual Machines". In: *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*. HPCVirt '09. Nuremburg, Germany: ACM, 2009, pages 17–24. ISBN: 978-1-60558-465-2.

[7] P. Kegel, M. Steuwer, and S. Gorlatch. *dOpenCL*. 2016. URL: `http://dopencl-uni-muenster.de/`.

[8] P. Kegel, M. Steuwer, and S. Gorlatch. "dOpenCL: Towards uniform programming of distributed heterogeneous multi-/many-core systems". In: *Journal of Parallel and Distributed Computing* 73.12 (2013). Heterogeneity in Parallel and Distributed Computing, pages 1639–1648. ISSN: 0743-7315.

[9] Khronos Group. *The OpenCL Specification. Version: 1.2*. Nov. 14, 2012.

[10] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. "SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters". In: *Proceedings of the 26th ACM International Conference on Supercomputing*. ICS '12. San Servolo Island, Venice, Italy: ACM, 2012, pages 341–352. ISBN: 978-1-4503-1316-2.

[11] C. M. Kohlhoff. *Boost.Asio - 1.61.0*. 2016. URL: `http://www.boost.org/doc/libs/1_61_0/doc/html/boost_asio.html`.

[12] L. Shi, H. Chen, J. Sun, and K. Li. "vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines". In: *IEEE Trans. Comput.* 61.6 (June 2012), pages 804–816. ISSN: 0018-9340.

[13] J. Song, Z. Lv, and K. Tian. *KVMGT: a Full GPU Virtualization Solution*. Oct. 2014.

[14] A. P. u. V. S. University of Muenster. *The Real-Time-Framework*. 2016. URL: `http://www.uni-muenster.de/PVS/en/research/rtf/index.html`.

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
116	978-3-86956-397-8	**Die Cloud für Schulen in Deutschland : Konzept und Pilotierung der Schul-Cloud**	Jan Renz, Catrina Grella, Nils Karn, Christiane Hagedorn, Christoph Meinel
115	978-3-86956-396-1	**Symbolic model generation for graph properties**	Sven Schneider, Leen Lambers, Fernando Orejas
114	978-3-86956-395-4	**Management Digitaler Identitäten: aktueller Status und zukünftige Trends**	Christian Tietz, Chris Pelchen, Christoph Meinel, Maxim Schnjakin
113	978-3-86956-394-7	**Blockchain : Technologie, Funktionen, Einsatzbereiche**	Tatiana Gayvoronskaya, Christoph Meinel, Maxim Schnjakin
112	978-3-86956-391-6	**Automatic verification of behavior preservation at the transformation level for relational model transformation**	Johannes Dyck, Holger Giese, Leen Lambers
111	978-3-86956-390-9	**Proceedings of the 10th Ph.D. retreat of the HPI research school on service-oriented systems engineering**	Christoph Meinel, Hasso Plattner, Mathias Weske, Andreas Polze, Robert Hirschfeld, Felix Naumann, Holger Giese, Patrick Baudisch, Tobias Friedrich, Emmanuel Müller
110	978-3-86956-387-9	**Transmorphic : mapping direct manipulation to source code transformations**	Robin Schreiber, Robert Krahn, Daniel H. H. Ingalls, Robert Hirschfeld
109	978-3-86956-386-2	**Software-Fehlerinjektion**	Lena Feinbube, Daniel Richter, Sebastian Gerstenberg, Patrick Siegler, Angelo Haller, Andreas Polze
108	978-3-86956-377-0	**Improving Hosted Continuous Integration Services**	Christopher Weyand, Jonas Chromik, Lennard Wolf, Steffen Kötte, Konstantin Haase, Tim Felgentreff, Jens Lincke, Robert Hirschfeld
107	978-3-86956-373-2	**Extending a dynamic programming language and runtime environment with access control**	Philipp Tessenow, Tim Felgentreff, Gilad Bracha, Robert Hirschfeld
106	978-3-86956-372-5	**On the Operationalization of Graph Queries with Generalized Discrimination Networks**	Thomas Beyhl, Dominique Blouin, Holger Giese, Leen Lambers
105	978-3-86956-360-2	**Proceedings of the Third HPI Cloud Symposium "Operating the Cloud" 2015**	Estee van der Walt, Jan Lindemann, Max Plauth, David Bartok (Hrsg.)

ISBN 978-3-86956-401-2
ISSN 1613-5652