Radium: Transparent Distributed Execution via Process Virtualization

Aidan Cully, Dusan Veljko, Hyojong Kim, Vance Miller, Husheng Zhou, Joel Zambrano, Mazhar Memon
Cloud Infrastructure Business Group, VMware, Inc.
{acully,dveljko,khyojong,vmiller,zhusheng,jzambrano,mmemon}@vmware.com

Abstract

The soaring demand for AI has led to a surge in specialized computation hardware, which poses challenges in sharing resources through conventional virtualization methods among end users. Moreover, the extensive data required by AI often cannot be conveniently co-located with the compute resources, resulting in costly and unsuitable migration attempts. To address these issues, Radium offers a userspace framework employing process virtualization, thread execution migration, and distributed shared memory. By leveraging Radium, an unmodified application binary operates in an encapsulated virtualized environment and its execution can be transparently distributed among nodes where resources locate. Radium enables resource aggregation with little performance penalty over high latency network connectivity. By virtualizing Linux processes, Radium supports novel hardware by nature without modifying existing infrastructure or applications.

1 Introduction

The soaring demand for AI needs tremendous data and computing power. It is unaffordable for end users to purchase dedicated powerful computing resources for a single task. The easiest way is to rent computing resources [4–7]. However, a common challenge arises when the required data is located far from the computing resources. Typically, two approaches are adopted: bringing computation to the data or bringing data to the computation. Presently, distributed execution frameworks bring computation to data by distributing code to nodes where the data is located [1, 13]. Nonetheless, this approach proves infeasible for hardware-accelerated computing, as data machines often lack accelerators. On the other hand, bringing data to computation necessitates copying data to computing resources or remotely mounting data using NFS. However, there are constraints associated with data copying: confidential or private data may not be allowed to be copied, large data sizes may result in unaffordable copying time and storage, and streamed or frequently updated data requires frequent synchronization with its copy. Using NFS also requires the data and computing machines to be co-located within a low-latency network. Otherwise, AI and ML-style data access models will introduce significant overhead. Preliminary experiments demonstrate that a network latency of 5ms can result in model training with Tensorflow [10] being more than 4 times slower.

API-remoting [9, 12] presents an alternative approach to utilize remote computing resources with local data without explicitly copying data. However, it is not universally applicable since it heavily relies on understanding targeted libraries and APIs. It requires considerable developer effort to navigate undocumented APIs and update the API remoting framework with each new targeted vendor release. To tackle real-world resource aggregation challenges, we introduce Radium, a universal solution that seamlessly aggregates local and remote resources with minimal performance penalty over high-latency network connections. Radium enables unmodified application executable binaries to operate within an encapsulated virtualized environment, and their execution can be transparently distributed among nodes where resources are available. By encapsulating Linux processes, Radium readily supports novel hardware and remains unaffected by vendor library changes.

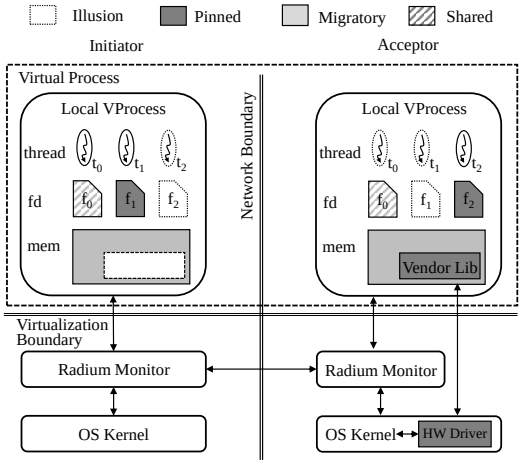


Figure 1. Architecture of Radium Virtualization.

2 Design and Implementation

Terminologies used in Radium are borrowed from elsewhere in VMware [2]: where VMware has the concepts of virtual machines, physical machines, VMExits, and the VMMonitor, Radium has the analogous concepts of virtual processes (***VProcess***), physical processes (***PProcess***), virtual process exits (***VPExit***), and the virtual process monitor (***VPMonitor***). Fig. 1 shows an example of Radium deployed on two nodes connected via a network. The ***Initiator*** node (on the left of the figure) runs the PProcesses that the user directly interacts with, while the ***Acceptor*** node runs Radium PProcesses in response to requests from remote Initiators. The Initiator and Acceptor run on different physical machines without sharing any physical resources: the only means of communication between them is the Radium communication channel.

Radium defines three classes of resources that a process interacts with: ***Shared***, which are usable from any location at any time; ***Migratory***, which are usable from any location but only one location at a time; and ***Pinned***, which are usable only from certain locations. In Fig. 1, file f_1 is a pinned resource on the Initiator; file f_2, hardware driver, and its userspace library are pinned resources on the Acceptor; private read/write memory mappings, as well as execution threads, are considered migratory resources; and file f_0 (which represents a file known to be identical on both nodes, such as `/dev/null`) is considered shared between nodes (each node uses a private copy of the resource).

SC'23, Nov 12–17, 2023, Denver, CO
2023.

When Radium running an executable binary, the very first step is to do *bootstrapping*, which creates a new VProcess sharing an address-space with the VPMonitor. Then the VProcess code proceeds to load the application binary using an embedded dynamic loader which works similarly to the kernel – mapping application binary, getting PT_INTERP, loading dynamic linker, preparing initial stack, and beginning execution. After the virtualization boundary of VProcess is established, every syscall will be intercepted by Radium to either directly pass to kernel or forward it to remote nodes. Radium maintains memory coherency among intiator and acceptors by using *Distributed Shared Memory (DSM)* which can be concurrently accessed from VProcess on any number of nodes. Radium uses *Overlay File System* to provide an illusion of a common filesystem across all the nodes, which includes regular files, device files, and special files.

3 Evaluation

We employed popular AI frameworks and models to conduct a preliminary evaluation of Radium, aiming to address the following key questions: 1) How well does Radium perform compared to native run and existing techniques? 2) How does Radium respond to varying network latency and different data and computation localities? For this evaluation, we utilize Tensorflow [10] for training models on both the mnist [3] and imagenet [8] dataset. Our setup involve one initiator and one acceptor, with data residing on the initiator and computational resources (GPUs) located at the acceptor.

To address the first question, we conduct a performance test, the specifics of which are detailed in Table 1. We compare the performance metrics (measured in images per second) of Radium against both Bitfusion and the native run as the baseline. We choose Bitfusion because it represents a state-of-the-art solution capable of migrating CUDA computing to remote GPU machines while leveraging local data. Additionally, Bitfusion is optimized for PVRDMA connections, improving data transfer efficiency.

The results, displayed in Fig. 2, depict the normalized performance relative to the baseline on the y-axis, with the benchmark name on the x-axis, consisting of the model name and GPU type. For instance, "Resnet56-V100x2" signifies training the Resnet56 model on two NVIDIA V100 GPUs. Fig. 2 demonstrates that Radium achieves over 75% of the baseline performance, often exceeding 85% and even reaching up to 99%. Notably, Radium outperforms Bitfusion significantly in all cases, particularly when dealing with intensive computation and data loads. Furthermore, Radium supports AMD GPUs, a feature absent in Bitfusion.

To address the second question, we conducted two tests with varying network latency. In the first test we run pytorch-dcgan [11] and observe the end-to-end execution time compared to Bitfusion and native run, as shown in Fig. 3 (a). We observe that as network latency increases, the execution time for both Radium and Bitfusion grows linearly, with Radium exhibiting a more gradual increase. For instance, when network latency reaches 50ms, Radium's execution time is 232s compared to the native execution's 176s. In the second test, we loaded a 64MB dataset located on the remote acceptor from the initiator to observe the data loading time under various network latency. We compared the data load time of Radium with a highly-tuned network file system (NFS), which sends the entire 64MB file in response to a single read. The results are presented in Fig. 3 (b), where the x-axis represents latency in milliseconds, and the y-axis represents data load time in seconds. We observed

Table 1. Experiment Setup

Components	Initiator	Acceptor
OS	Ubuntu 22.04	Ubuntu 22.04
CPU	Intel Xeon Gold 6230R	Intel Xeon Platinum 8260
RAM	128GB	128GB
Network	Low latency TCP/IP/PVRDMA	
GPU	N/A	2xV100 / 1xMI100

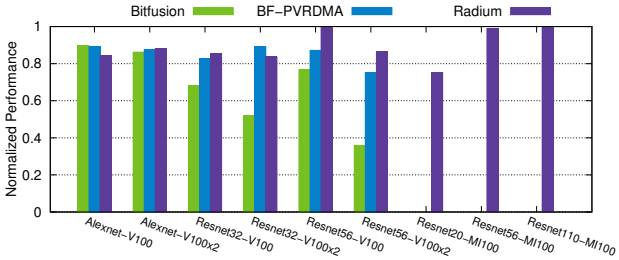


Figure 2. Normalized performance of Radium and Bitfusion

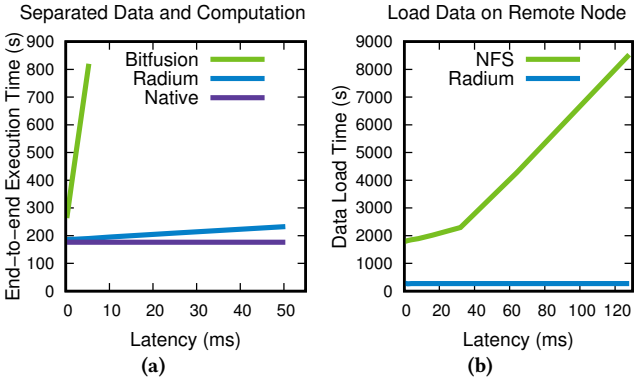


Figure 3. Network latency resiliency and breakdown

that the load time of NFS increases significantly with latency, while Radium's performance remains consistent. This is because Radium loads the data into memory on the node containing the data and can hide communication overheads for later portions of the dataset while the GPU processes earlier portions.

These network-resilient tests essentially illustrate the network round-trips in the critical path under different scenarios: Bitfusion requires round-trips at every CUDA synchronization point, while NFS demands round-trips when reading data from the disk. Radium showcases commendable latency resilience in both scenarios.

4 Conclusion

In this paper, we introduce Radium, a userspace framework that leverages process virtualization to facilitate seamless distributed execution across both local and remote locations. Radium empowers resource aggregation by establishing virtual process connections among nodes housing these resources. With Radium, cutting-edge hardware can be swiftly deployed and seamlessly integrated into existing hardware and software stacks. This is made possible because Radium maintains a virtualization boundary at public syscalls, thus remaining impervious to vendor-specific proprietary APIs. Experimental results underscore Radium's robust performance, particularly for AI benchmarks characterized by disaggregated data and computation, even in scenarios with high-latency network connectivity.

References

[1] Dhruba Borthakur. 2007. The hadoop distributed file system: Architecture and design. *Hadoop Project Website* 11, 2007 (2007), 21.

[2] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y Wang. 2012. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Transactions on Computer Systems (TOCS)* 30, 4 (2012), 1–51.

[3] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. 2017. EMNIST: Extending MNIST to handwritten letters. In *2017 international joint conference on neural networks (IJCNN)*. IEEE, 2921–2926.

[4] Amazon Corporation. 2006. Amazon Elastic Compute Cloud (EC2). https://aws.amazon.com/ec2/

[5] Google Corporation. 2008. Google Cloud. https://cloud.google.com/

[6] Microsoft Corporation. 2010. Microsoft Azure. https://azure.microsoft.com/en-us/

[7] Oracle Corporation. 2016. Oracle Cloud Infrastructure. https://www.oracle.com/cloud/

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[9] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing & Simulation*. IEEE, 224–231.

[10] Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association.

[11] Wei Fang, Feihong Zhang, Victor S Sheng, and Yewen Ding. 2018. A Method for Improving CNN-Based Image Recognition Using DCGAN. *Computers, Materials & Continua* 57, 1 (2018).

[12] VMware, Inc. 2020. VMware vSphere Bitfusion. Retrieved Augest 5, 2023 from https://docs.vmware.com/en/VMware-vSphere-Bitfusion/index.html

[13] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*.