The Case for Physical Memory Pools: A Vision Paper

Heather Craddock(✉), Lakshmi Prasanna Konudula, Kun Cheng, and Gökhan Kul

Division of Physical and Computational Sciences, Delaware State University, 1200 N DuPont Hwy, Dover, DE 19901, USA
{hcraddock,lkonudula,kcheng14,gkul}@desu.edu
http://delawaresec.com

Abstract. The cloud is a rapidly expanding and increasingly prominent component of modern computing. Monolithic servers limit the flexibility of cloud-based systems, however, due to static memory limitations. Developments in OS design, distributed memory systems, and address translation have been crucial in aiding the progress of the cloud. In this paper, we discuss recent developments in virtualization, OS design and distributed memory structures with regards to their current impact and relevance to future work on eliminating memory limits in cloud computing. We argue that creating physical memory pools is essential for cheaper and more efficient cloud computing infrastructures, and we identify research challenges to implement these structures.

Keywords: Cloud computing · Memory · Operating systems · Virtualization

1 Introduction

The growth of cloud computing is fuelled by the rise of big data and the global requirements for the inter-connectivity of computing resources. The cloud allows for data and resources to be shared on demand to connected hardware devices, reducing the need for reliance on purely local resources. According to the Data Never Sleeps report, 1.7 MB of data is expected to be created every second for every single person on earth by 2020 [6]. With such a vast amount of data being generated, it is imperative that hardware and software architectures are modified or created to manage modern requirements.

Virtualization plays a vital role in improving the utilization of distributed resources, and consequently cloud computing. Currently, servers are capable of hosting multiple operating systems (OS) with the help of multiple virtual machines (VM) resulting in enhanced network productivity, recoverability, and data migration. Virtualization works by combining hardware resources among multiple VMs. It is a challenging task to run an unmodified operating system on virtualized hardware. This led to development of several techniques to either

D. Da Silva et al. (Eds.): CLOUD 2019, LNCS 11513, pp. 208–221, 2019.
https://doi.org/10.1007/978-3-030-23502-4_15

modify the guest OS or combine application of virtualization techniques that help the system best use the available resources in a secure and efficient manner.

Despite developments in memory hardware that increase the availability of memory at lower cost, memory use and capacity is still limited by the ability of operating systems to manage the vast resources that are now available. It is particularly difficult to coordinate these memory resources over the cloud as the lack of systems and methods that have been developed to manage large, distributed memory is limiting.

The pervasiveness of the cloud makes it imperative to constantly improve cloud technologies and to understand the current state of cloud operating systems. This paper provides an overview of some recent innovations in operating system design, distributed memory management, and virtualization, while also noting where improvements may be further made upon these proposals. Suggestions are also made as to the future of cloud computing in general, and the direction in which future work may lie.

In this paper, we begin by discussing existing models and techniques, including operating system design proposals, improvements to memory virtualization, more efficient memory management, and memory protection methods in Sect. 2. We discuss the future direction of memory models for the cloud, with potential future work in operating system design, distributed memory, and virtualization, and also the direction of cloud computing in general in Sect. 3. We then conclude in Sect. 4.

2 Existing Models

2.1 Operating System Design

Operating system design is of paramount importance to any computing environment, but despite the increase in the popularity and necessity of the cloud there is little development in distributed OS design. Current models are outdated for cloud operations or ill-equipped to deal with the increased availability of resources. Vasilakis *et al.* [11] discussed a whole new OS design in a recent work, and Swift [10] suggested a technique for improving operating system memory management. Shan *et al.* [9] state that the direction of cloud computing leads away from monolithic servers and towards more disaggregated hardware components, and the authors propose a new operating system, LegoOS, that is designed for such a disaggregated system. LegoOS is the first operating system designed to manage these separate components and is a revolutionary method for approaching operating system design.

Revamping OS Design. Vasilakis *et al.* [11] state in their paper that in the 40 years since the original UNIX system was developed, new features or developments have just been stacked upon previous features, resulting in an overly-complex and unwieldy model. Having grown beyond the original simplistic intent, the model is not suitable for scaling or decentralization. The authors found four key issues:

1. Data Cataclysm: Distributed system developments require more than minor tweaks
2. Reliance on Commodity Hardware: Software must manage assurance on fault-intolerant hardware
3. Rise of the Personal Cloud: Original model is not designed for personal micro-clouds
4. Data Processing Shift: Data processing is now performed by ordinary people

The authors propose a distributed system design built from the bottom up. The design includes a large number of components such as file systems more tolerant to scaling, a new type of execution primitive, and sandboxing for software fault isolation. The system was designed based on a programming language, like UNIX is based on C, but the language does not yet exist. For this design to be proved to be useful, the design must be practically implemented using an adapted or created programming language. This system was important in beginning the discussion about revitalizing OS design. Finally, although the authors proposed automatic memory management as garbage collection to save on the bugs and vulnerabilities inherent in manual memory management, garbage collection can have its own issues; it can be difficult to analyze memory and performance, and this garbage collection only deals with memory resources and not all resources. These issues would need to be addressed in a practical implementation.

OS Design for Near Data Processing (NDP). The emergence of NDP architecture required a revision to the traditional memory structure. The proximity to the memory modules has proven to enhance the throughput and exhibit low power consumption. In NDP, however, heterogeneity and parallelism cannot be solely handled, hence requiring the support of OS support to deal with problems like locality, protection and low-latency. To address these issues, Barbalace *et al.* [5] proposed a new OS called Shadowgraphy based on a number of design principles. A multiple kernel OS design where CPU heterogeneity can be backed on the same machine, enabling the services and IO devices to interact individually in NDP and CPU while exhibiting the same protection and privileges across all kernels. NDP enforces user privilege protections across all the kernels, so applications running in the NDP processing unit can maintain different levels of user privilege. Scheduling is done locally at every kernel by tracking the data access pattern. Instead of moving the data, the code is moved. Data migration is made efficient by caching it in both hardware and software at different levels of the memory hierarchy. The CPU and NDP topology takes a new shape enabling a transparent environment for the users to review.

Barbalace *et al.* [5] state that it is time to redesign system software for NDP starting from the OS. They address the drawbacks of offloading while multiple applications are running; however, it isn't discussed how Shadowgraphy OS solves this issue. In order to achieve transparency for application developers, the system should both support asymmetricity in the processing units and should be able to provide multiple levels of OS interfaces. A new, multi-kernel, multi-server design should be considered to accommodate multiple users with concurrent accesses in the system.

Distributed Memory Techniques. Distributed Shared Memory (DSM) is a memory architecture that makes it possible to share computing tasks over multiple different hardware components. In DSM, separate physical address spaces can be logically addressed as if they were one space, and in this way multiple processing nodes with individual memory components can be connected over a network to create a larger pool of memory resources. There are many issues with current Distributed Shared Memory techniques that recent papers attempt to address.

Scalability in Data-Intensive Applications. Many systems in use for data-intensive applications are not easily scalable, particularly over the current hardware configuration of many nodes connected over a high-bandwidth network. To attempt to solve this problem, Nelson *et al.* [7] propose Grappa, a software distributed shared memory system for use over clusters to improve performance over data-intensive applications. Grappa's key improvement was the implementation of parallelism to ensure the use of maximum process resource while also disguising communication costs and message-sending latency. Although previous methods for implementing distributed shared memory relied on locality of data and caching to be able to scale effectively, implementing distributed computing in parallel allowed the authors to disguise the high-bandwidth network costs inherent in the hardware system. To create this scalable system, Grappa was designed with three key components:

1. Distributed Shared Memory - Allows access to data anywhere in the system, where local data can be exported to the global address space to be accessible to other cores. Operations are performed at the data's home node to prevent unnecessary retrieval or sending of data over the network, guaranteeing memory consistency and global order.
2. Tasking System - Multi-threading and work-stealing allow for functioning parallelism and load-balancing to better utilize system resources. All tasks are allocated to worker threads to execute, and threads performing long-latency operations yield their core so the processor can still be utilized.
3. Communication Layer - Smaller messages are aggregated into larger ones to limit the use of network bandwidth

Scalability in Grappa comes at the cost of fault-tolerance, as it was deemed cheaper to restart after failure than recover. This is an area for potential improvements. It may be interesting to explore in future how to improve the system so that recovering from failure is cheaper than restarting entirely. In addition, sending small messages using this method is somewhat limited by current hardware; Grappa should be revisited as hardware innovations could lead to improvements in network latency.

Improving Memory Access Speed. Improving the speed of memory access is always crucial in distributed memory. Constructing large-scale clusters with vast memory resources is cheaper now than it ever has been, but network latency is caused by separated hardware elements. [4] suggests methods to improve memory access speeds over clusters.

Memory access using TCP/IP protocol in cluster systems is slow. Existing techniques like Remote Direct Memory Access (RDMA) allow direct memory access from one machine into another without involving the OS, but to achieve this the NICs bypass the kernel and remote CPU providing direct access to data. FaRM utilizes one-sided RDMA reads and directly accesses the memory, enhancing the speed of message passing and thus improving the performance of the apps. FaRM allows lock-free reads which ensures that the transactions are in order and utilizing a single RDMA read.

RDMA allows high throughput and low latency. RDMA also achieves remote memory access through the NIC instead of the remote CPU which means that the I/O operations will not go through the CP. This may cause the CPU to lose control of the data in some cases. If a transaction occurs and there is a failure to determine if there is enough storage space for the transaction, there could be data loss or other serious error. This can be mitigated by reserving enough space in the preparation step, but there is room for improvement.

Server Load Imbalance. The RackOut memory pooling technique suggested in [8] utilizes direct access to improve access speeds across clusters. Novakovic *et al.* noted that server load imbalance limited performance. While most large-key value stores keep data in the memory of memory servers in order to provide both low latency and high throughput, skew limits performance capabilities and there are currently no methods to reduce the skew that do not involve incurring other overheads. As skew can cause load imbalance which correlates to poor utilization of data centers, the authors note that it was important to develop a system which could meet all of its objectives while managing load imbalance.

RackOut is implemented on a group of servers that have internal high bandwidth, a low-latency communication fabric, and direct access to other nodes' memory through one-sided operations. Using this method, nodes in the rack can access the memory of other server nodes without using the remote CPU, thus minimizing server load imbalance. Furthermore, as memory access within a rack is fast and data is only replicated when needed outside the rack, speed of memory access is improved and sharing operations between the nodes balanced the workload more evenly across the rack. The RackOut method is limited by the communication fabric latency, although technology is trending towards lower-latency fabrics. While this system discusses its scalability, the study was limited to research resources; it would improve on the theoretical nature of the paper for tests of this system to be conducted on larger or commercial-scale data centers.

Limited Discussion on Remote Memory. Without discussion of the issues facing areas of computing, it can be difficult to establish the direction of a field or the potential for future work. In [3], the authors enumerate a number of different areas in cloud computing that could face challenges as they believed that discussion on the subject was out of date. The idea of remote memory was proposed nearly 20 years ago when network technology made it difficult to find and implement remote memory solutions. Though current networks are a great deal faster than before, there is still limited discussion on efficiently realizing remote memory. There are still a number of challenges that we need to address, and some potential solutions:

1. Remote host crashes: (1) expose failures to the application by allowing it to provide failure handlers; (2) use replication or erasure coding to mask the failure through redundancy.
2. Slow or Congested Network: (1) prioritize network traffic and pre-allocate network bandwidth for remote memory; (2) give each application different regions of memory.
3. Virtual Memory Overheads: check and rebuild the subsystem of the virtual memory.
4. Virtual Machine Indirection: Based on the mechanisms of reducing virtual memory overheads, find ways for the hypervisor to extract information about applications.
5. Transparency Level: design the remote memory in different cases.
6. Sharing Model: Limit remote memory to private data, where sharing is prohibited.
7. Lack of Write Ordering Across Hosts: (1) DSM enforces ordering with appropriate protocols to solve the problem, although this is costly; (2) allow applications to use remote hosts for memory; (3) allow reordering for applications whose semantics support it.
8. Non-uniform latency: (1) use the existing operating system mechanisms for NUMA; (2) expose the memory speed to applications, which can use appropriate data structures and layouts to optimize the performance.
9. Remote Host Compromised: (1) encrypt the data in remote memory; (2) strengthen the security of the larger system to compensate for the larger attack surface.
10. Local vs. Cluster Memory: adopt a static allocation which reserves a reasonable amount of local memory and leaves the rest for cluster memory.
11. Remote Memory Allocation: centralize the problem by requiring allocations across the cluster to go through a host that manages memory.
12. Memory Placement: the simplest mechanism is to centralize the decision of placement.
13. Local Memory Management: the machine hosting the physical memory should manage it, but this may add overheads on modern RDMA-based NICs.
14. Control Plane Efficiency: use off-the-shelf solutions for control planes, although experiments testing their performance for this use are required.
15. Memory Metadata Overhead: manage remote memory in slabs that are much larger than the page size, so that the system need only keep one set of metadata for each slab.

Although [3] introduces a number of issues and the potential solutions, the authors overlook some key issues. Firstly, the authors do not deeply discuss security problems. When the data is stored in remote memory, not only do we need to consider the local machine security state, but we also need to think about the network and remote machine security. Implementation of suitable security measures may be more costly in terms of computing resources or finances; there are two possible ways to improve security: firstly, the remote machines may have the highest security priority because the bulk of the data is stored there; secondly, access and the network needs to be appropriately secured and protected; finally, distributing the data over a larger number of different remote machines with their own security could prevent an attack on one host from compromising the whole system.

2.2 Improving Memory Virtualization

Virtual memory plays a vital role in modern computing as it can deliver various benefits like improved security and increased productivity for programmers. The operating system and page table play a crucial role in memory management but can cause high execution-time overheads. As an attempt to solve the problem, Aguilera *et al.* [3] proposed a hardware/software co-design called Redundant Memory Mapping (RMM).

The paper proposes a hardware/software co-design called Redundant Memory Mapping (RMM). Range translation can map contiguous virtual pages to physical pages. The authors address the primary problem of using a page in their paper. A TLB miss can be overcome by using 124 range translations from the range table. The paper presented few evaluations showing that RMM works for all configurations and workloads.

Although RMM eliminates vast majority of page walks, using eager paging may increase latency, which in turn can induce fragmentation. Latency and fragmentation can have heavy impact on the performance. Implementing RMM also relies on additional hardware and software which may involve future development. Retrieving data in parallel during translation can be a potential solution to the stated problem. This allows storage of huge data sets with low-latency for real time data analysis.

2.3 Efficient Memory Management

Memory capacity is a key limitation in system design. The recent breakthrough 3D XPoint memory in non-volatile memory technology has given the world abundant memory capacity at much lower costs. These technologies provided memory in larger magnitudes than DRAM at lower power and prices. Existing system designs, however, are incapable of handling such large memories.

The essential design principle of Order(1) operations is proposed in [10] to manage vast memories. This principle aims to complete memory management operations in constant time, independent of the size of the operand. It applies a file-system technique to memory management. Instead of operating on individual pages, the operations are enabled on the whole file and thus providing Order(1) performance. It is less complicated to expose data directly to the programs instead of the kernel as the data already exists in the memory. Using the file systems to manage memory is convenient as they can maintain gritty meta data, and are capable of translating large addresses and handling large memories. In Towards O(1) memory, the memory layer above the files is removed and the user mode memory is allocated as files with tmpfs as a backup. Only the references to the files are counted, ignoring the references to pages. Memory can only be reclaimed when a process terminates or unmaps. Pointers are used to improve the efficiency in memory mapping through sharing between the processes. Overheads in tracking and cleaning bits is not required as the memory itself is too large which eliminates the need for swapping between disks. This system uses range translations to trim the cost of memory access.

Implementing Towards O(1) memory has its own limitations: operations that depend on page level mappings cannot be easily supported and are difficult to optimize; also, the system becomes complex when we try to store the volatile data in the persistent memory, breaking the isolation between user and kernel space and leading to memory leakages. In order to avoid this situation the memory should be zeroed before being reused.

2.4 Memory Protection

The emergence of large non-volatile main memories and rack-scale computers running large 'micro services' creates significant challenges for memory protection based solely on MMU mechanisms. Optimization for translation performance has put the protection at stake. Some challenges include stale locations leading to memory corruption, hypervisor calls, and nested pages.

Achermann *et al.* [2] propose Matching Key Capabilities(MaKC), a new architecture which is capable of scaling memory protection at both the user and kernel level. MaKC divides the memory hierarchies into equal sized blocks associated with Block Protection Key (BPK) and Execution Protection Key (EPK). Memory access is allowed only when there is a match between BPK and EPK, and blocked otherwise. HMACs (Hash Message Authentication Codes) ensure that messages cannot be forged nor manipulated in transit. The authors proposed that key matching could be implemented with protection tables that contain BPKs which the hardware can read and cache. Proposed MaKC has a capability-based system to handle authorization and protection in complex memory hierarchies. The MaKC approach also allows enabling huge pages without compromising the security of small page sizes.

MaKC has not been fully designed yet, and there are still decisions to be made on which features need to be implemented on the CPU-side or memory-side. Implementing MaKC could lead to some potential issues:

1. Storing large number of keys and HMACs can result in space overhead; in order to access them, one needs to enter supervisor state which can be expensive

2. The use of MaKC will add complexity
3. Using cryptographic keys in MaKC to authenticate the fingerprints could increase security management complexity
4. The model is proposed for fixed size blocks which may leave memory unused and thus poor memory management

3 Research Directions for Disaggregated Memory in the Cloud

In this section, we first make a case for physical memory pools. Then we discuss the potential improvements in OS design, distributed memory, and virtualization, as well as considerations regarding the direction of cloud computing as a whole.

3.1 Physical Memory Pools

We argue that the cloud systems need to migrate to a hardware resource disaggregation scheme that uses *physical memory pools*. A physical memory pool (PMemP) is a cluster of memory units that do not belong to any monolithic server, and can be demanded and used by connected monolithic machines on a need-based basis as seen in Fig. 1. This component is governed by a *governor* that prioritizes resource requests from participating servers, and allocates memory to these machines accordingly. An example of this approach in the literature is LegoOS [9] where networked *mComponents* are used as memory units. In this sense, an mComponent is a type of physical memory pool. There are four challenges that this hardware resource disaggregation scheme has to tackle: (1) Current OS architecture limitations (Sect. 3.2), (2) Plug-n-play use of the component (Sect. 3.3), (3) Adoption to existing virtual systems (Sect. 3.4), and (4) Network speed limitations to use memory as a networked device (Sect. 3.5).

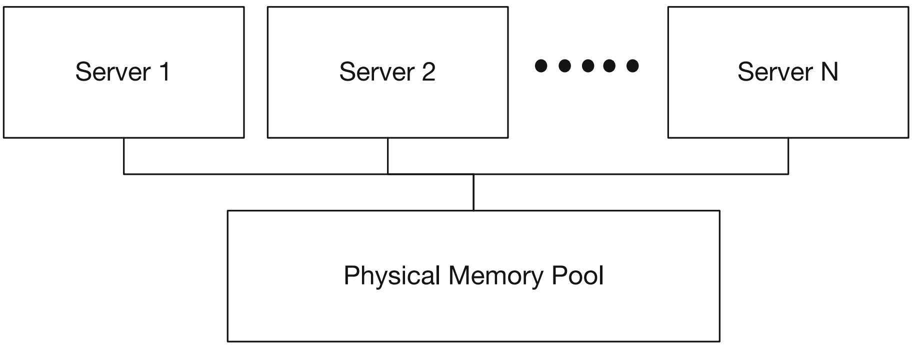


Fig. 1. Physical memory pool overview

Example. Java Virtual Machine (JVM) uses a memory space that is allocated to itself when a Java program is run on the system. The JVM users can define a minimum amount of heap memory as well as a maximum amount of heap memory for the program. Initially, the program is given the minimum amount. If the program requires more heap memory than this amount, JVM increases the allocated memory, up to the maximum heap amount [1]. We expect that the physical memory pools work as efficient and in a similar manner to this logic. To show how this would work on JVM, we designed the following simulation.

First, we create a `List`. In an infinite loop, in each iteration, we add 100 elements to the list, and randomly search for 100 elements by index in the list until we run out of memory. This enforces utilization of each part of the list, hence preventing the use of swapping function efficiently. When the system runs out of the initial heap memory, it has to make the next block of memory available, just like PMemPs would, as described in Sect. 3.1.

In the simulation run, we used an Apple MacBook Pro with 2.2 GHz Intel Core i7processor, 16 GB 2400 MHz DDR4 RAM, and 256 GB SSD. The OS is MacOS High Sierra 10.13.6, and we used Java 1.8.0. To show the efficiency of the mechanism and difference between block allocation and directly using maximum available heap memory, we compared initial allocations of 128 MB, 256 MB, 512 MB and 1024 MB, respectively, and maximum allocation of 1024 MB. The results can be seen in Fig. 2. We share all the code and documentation on GitHub[1].

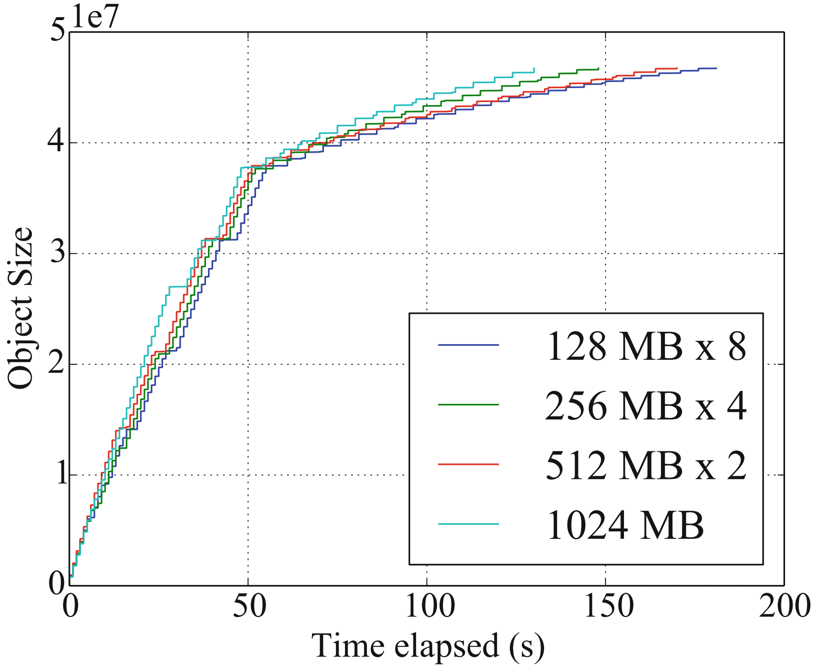
Figure 2a shows the growth of the List over time. As expected, the runs with larger initial heap size reach to the maximum number of elements faster. However, the increase pattern is still comparable and the difference is due to the time JVM spends on adding new memory blocks and time spent on garbage collection before doing so. The garbage collection behavior can be better seen in Fig. 2b. The garbage collector runs more aggressively when nearing the initial heap size, but it still needs to add new memory blocks to be able to continue operating. In a memory resource disaggregation based system, this is the ideal behavior. The advantage of memory resource disaggregation is, instead of building monolitic servers with large memory sizes, we can share a common pool of memory accordingly.

3.2 OS Design

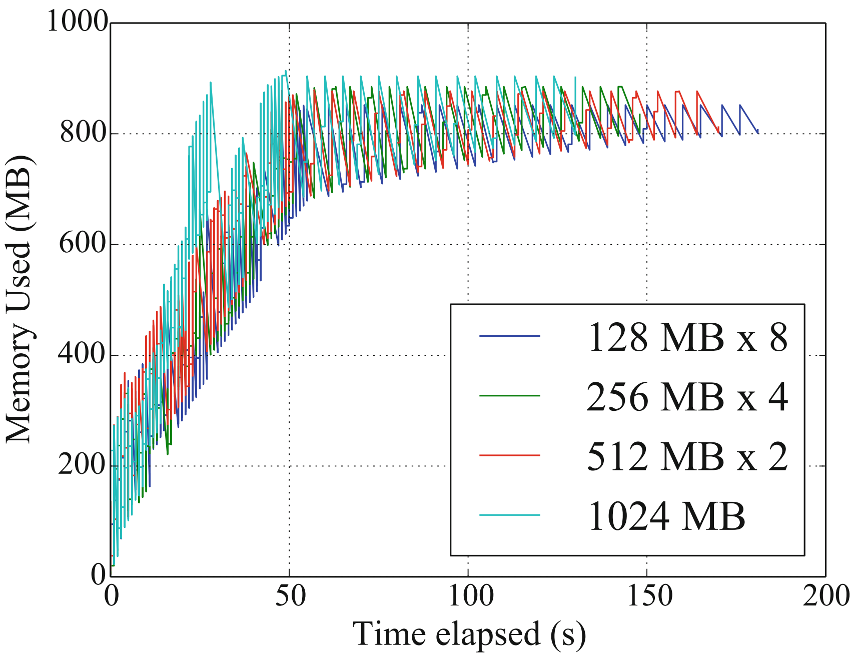
As in [11], OS design may be constrained by the availability of an appropriate language to code it. The OS should provide a transparent environment to the application developer which can allow users to inspect information about the platform, such as the topology of the CPU and NDP.

Future work may also include the design of a multiple-kernel, multi-server model that can allow concurrent access to multiple users on the system and

[1] Code repository located at https://github.com/PADLab/MemorySwapExperiment.



Object count growth in the List



Memory utilization growth

Fig. 2. Comparison of allocation under varying memory configurations

support the asymmetricity in the processing units. A future cloud OS should be able to be as multi-functional and dynamic as possible to meet ever-developing requirements.

3.3 Distributed Memory

In distributed systems, networking hardware, address translation and NICs often act as performance bottlenecks to the system. To prevent this bottleneck, developments must continue to be made in the hardware that the discussed systems are designed upon. Meanwhile, these limitations could be curtailed by increasing the node processing and network capacity, building bigger and more capable nodes, aggregating multiple server nodes into larger entities, and implementing in-memory computing where the translations and data fetch can go hand in hand.

In systems such as Grappa proposed in [7], improving fault-tolerance is important, especially as systems are constantly being implemented on fault-prone hardware. Efforts must be made to design systems where recovering from faults is simpler than restarting.

3.4 Virtualization

Virtualization provides flexibility, scalability, and cost advantages to cloud computing. Although virtualization breaks the line between the hardware and the applications running on dedicated servers, adding a virtualization layer has downsides such as slashing the application performance, adding processing overheads for memory, translations and inducing security vulnerabilities in the system. To prevent these bottlenecks, the operating system should be considered as the principal design for virtualization. Temporarily, we can downsize these limitations by adapting towards in-memory computing and providing isolation between the kernel and user space.

In the future, range translations should pave the way for emerging workloads by utilizing in-memory computing, which can leverage the growth in physical memory to store huge data sets for low-latency and real-time data analysis.

3.5 Network

Using current network technology introduces some challenges that are still heavily researched. Firstly, network components and protocols include their own communication overheads. We believe that in a hardware communication focused environment this problem can easily be addressed, especially if the hardware network is isolated from the communication network. Secondly, network bandwidth limits the speed and it may not be as fast as a motherboard bus. It should be noted, however, that network technology has developed drastically in the last decade and this should soon become a minor issue.

3.6 Direction of Cloud Computing

There are a number of questions to contemplate:

1. How can system complexity be managed? With movement towards heavily interconnected microservices provided by a number of different service providers, it is important to maintain performance and security through all the different structures. There is also a strong degree of trust in these services, as other teams or companies may run vital infrastructure. It is important to find methods to maintain and ensure trust and reliability of outsourced services.
2. How can failure tolerance be guaranteed? Distributed systems are responsible for a number of critical societal components, and there is a huge reliance on the cloud for health, safety, productivity, business, and more. Ensuring network systems are failure-tolerant should be a top priority.
3. How can virtualization be improved and supported? Sharing resources between different and distant hosts is a key component of cloud computing, so how can operating systems continue to support more efficient virtualization of services and guarantee their security?
4. How can the environmental and financial impact of distribution be limited? Increased distribution leads to increased energy cost as giant data centers are always providing their services. Virtualization can help this as the resources can be used from many different physical locations. Potential work for the future may be in improving workload distribution on virtualized software so that the most demanding tasks are completed on the most energy-efficient hardware environments.

4 Conclusion

We discussed the contribution of a number of works that proposed methods of improving or innovating the cloud, from specific models such as LegoOS, Grappa, Shadowgraphy OS and RackOut to theoretical discussions of the issues cloud computing presents as in [3]. The core contribution of this paper lies in offering future research directions around the vision of independent physical memory pools that the servers in a data center can share based on their current need, and de-allocate when the resources are no longer needed.

Although some improvements or ideas for potential future work were suggested in this paper, the survey is by no means comprehensive; the consistent theme from the reviewed papers is that the discussion is only recently beginning and there is still so much to explore in the realm of cloud computing.

Acknowledgements. This material is based in part upon work supported by the funding provided by the State of Delaware to Delaware State University. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the University or the State.

References

1. Tuning java virtual machines. docs.oracle.com. Accessed 22 Mar 2019
2. Achermann, R., et al.: Separating translation from protection in address spaces with dynamic remapping. In: HotOS (2017)
3. Aguilera, M.K., et al.: Remote memory in the age of fast networks. In: ACM SoCC (2017)
4. Hodson, O., Dragojević, A., Narayanan, D., Castro, M.: FaRM: fast remote memory. In: USENIX NSDI, 4 (2014)
5. Barbalace, A., Iliopoulos, A., Rauchfuss, H., Brasche, G.: It's time to think about an operating system for near data processing architectures. In: HotOS (2017)
6. James, J.: Data Never Sleeps 6.0. Technical report, Domo Inc, 06 2018
7. Nelson, J., et al.: Latency-tolerant software distributed shared memory. In: USENIX ATC (2015)
8. Novakovic, S., Daglis, A., Bugnion, E., Falsafi, B., Grot, B.: The case for rackout: Scalable data serving using rack-scale systems. In: ACM SoCC (2016)
9. Shan, Y., Huang, Y., Chen, Y., Zhang, Y.: Legoos: a disseminated, distributed OS for hardware resource disaggregation. In: USENIX OSDI (2018)
10. Swift, M.M.: Towards O(1) memory. In: HotOS (2017)
11. Vasilakis, N., Karel, B., Smith, J.M.: From lone-dwarfs to giant superclusters: rethinking operating system abstractions for the cloud. In: HotOS (2015)