Analysis of a modern distributed hypervisor: what we learn from our experiments

Mohamed Karaoui
first.last@ens-lyon.fr
ENS Lyon

Brice Teguia
brice.teguia_wakam@ens-lyon.fr
ENS Lyon

Bernabe Batchakui
bbatchakui@gmail.com
ENSPY Yaoundé

Alain Tchana
first.last@ens-lyon.fr
ENS Lyon

Abstract

GiantVM is the state of the art distributed hypervisor (VEE 2020) which is based on KVM. It allows to start a guest OS whose CPU and memory are provided by several physical machines. In this paper, we study the origin of performance overhead of GiantVM – which is the DSM, hence, in term of number of page faults. Then we propose several optimizations which allow to improve performances by about 39%. Moreover, based on our study and proposed optimizations, we laid out guidelines to build a distributed hypervisor that is independent and flexible – easy to evolve, from a distributed shared memory (DSM) system..

CCS Concepts: • **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks →** Network reliability.

Keywords: Virtualization, Distributed Shared Memory, Cloud, KVM

ACM Reference Format:
Mohamed Karaoui, Brice Teguia, Bernabe Batchakui, and Alain Tchana. 2022. Analysis of a modern distributed hypervisor: what we learn from our experiments. In *SPMA '22: Systems for Post-Moore Architecture, April 05, 2022.* Rennes, France, 6 pages.

1 Introduction

Despite the introduction of virtualization technology, data centers (DC) still face the problem of *server fragmentation*, which causes a considerable loss of money for operators. For example, Microsoft estimates in 2020 that a 1% reduction in fragmentation within its Azure cloud platform would result in hundreds of millions of dollars in savings [1]. This problem is fundamentally due to the classical server-centric architecture [2], which is massively adopted in today's DC. In this architecture, the DC is made of monolithic servers, each including all necessary hardware resources (mainly CPU, RAM, network and disk) to run applications. The latter are constrained to a single server boundaries.
Among the approaches that have been proposed to reduce fragmentation, *rack disaggregation*[1] [2–8] appears to be the most promising approach. It consists in consolidating a large pool of hardware resources (CPUs, RAM, disk, network, etc.) into a single large *rack-scale computer*. Hence, in this model, the DC can be seen as a set of very large machines, that can each be dynamically partitioned into applications with very flexible and diverse resource allocations. In order to become viable in practice, disaggregation requires (i) very efficient network communication between the elements/machines within the rack and (ii) specific support from the system software (OS, hypervisor).
Two forms of disaggregation, that we name *hard disaggregation* and *soft disaggregation*, have been proposed in the recent years. Hard disaggregation [8–11] is a more aggressive approach, as it requires a deep redesign of both hardware and software layers. In hard disaggregation, the rack is built as a cluster of specialized and independent resource boards (a resource-centric architecture) instead of a cluster of monolithic servers. Each resource board is a specialized machine that offers only one resource type to applications. Resource boards are interconnected with an ultra-fast network fabric [12]. Soft disaggregation [2, 4–7] consists in using traditional monolithic servers and adapting *only the software layers* to allow a VM to simultaneously use resources from several machines (within a rack). The high performance offered by current DC network technologies (e.g., Infiniband) makes soft disaggregation exploitable in the current server-centric architecture [3].
We focus on soft disaggregation solutions as it is viable with today's technologies compared to hard disaggregation. Hereafter we just use the term disaggregation to refer to soft disaggregation. Most research work in disaggregation focused only on memory disaggregation (far memory utilization), allowing a centralized VM (which vCPUs all run on a single machine) to remotely use free memory of other nodes. These solutions assume that memory is the limiting resource for server consolidation in the cloud. The recent paper published by Microsoft in 2020 at OSDI by Ambati et al. [1] showed that in their cloud, things are different: "*In more than 80% of cases where we could not allocate the hypothetical VM, the scarcest resource (i.e., the one that prevents the allocation) is cores...In the vast majority of remaining cases, disk space is the*

[1]For practicability purposes, DC disaggregation is always studied at the rack scale.

constraint.. Far memory utilization based solutions is inefficient in this context. Therefore, the need for investigating distributed VMs, while still providing a single system image (SSI) to cloud users, is becoming crucial.
Zhang et al. [7] introduced GiantVM, a Linux/KVM-based and open source distributed hypervisor which allows to start distributed VMs. To this end, GiantVM implements a distributed shared memory (DSM) based on Ivy protocol [13]. A distributed VM is made of a set of centralized VMs which share the same memory using the DSM. While some of the challenges have been discussed in the GiantVM paper (such as interrupt distribution and distributed timer synchronization), GiantVM includes the following limitations: (1) it does not distribute virtual disks; (2) it does not allow live migration; (3) it is closely linked with the x86 implementation of KVM, which has two consequences. First, it is architecture dependent (Intel). Secondly, linking the two code bases (KVM sources and GiantVM ones) is error prone, making it very difficult to maintain and scale.
The work presented in this paper focuses on the two latter limitations. First, we study the origin of performance overhead of GiantVM, then we propose several optimization ideas which improve GiantVM by up to 39%. Moreover, based on our study and the proposed optimizations we lay out the fundations to build distributed hypervisors that have generic and flexible (easy to evolve) DSM design for distributed VMs.
The rest of the paper is organized as follows. Section 2 introduces GiantVM. Section 3 presents our contributions and their performance impact. Section 5 presents the related work. Section 6 concludes the paper.

2 GiantVM

This section provides a summary of the key concepts and architecture of GiantVM, for further details refer to [7] . GiantVM is a QEMU-Linux/KVM based distributed hypervisor which allows to launch distributed guest OSes, as shown in Fig. 1. To run a distributed virtual machine, we basically start multiple sub VMs, which resides on different physical nodes and need to be synchronized and then synchronize the sub VMs to obtain a global entity. To build GiantVM, the authors modified both QEMU (user space application) and KVM (kernel module). In the following we describe how each resource is virtualized and distributed by GiantVM.

vCPU. They are QEMU threads. They are one of the easiest resources to distribute since their state (set of registers) does not need to be shared. A vCPU state needs to be accessed (read and write) only locally by the vCPU/thread. GiantVM starts on each node all the vCPUs of the distributed VM except that the vCPUs that are not local are paused waiting for an exit event (see below). These vCPUs are called *shadow vCPUs*.

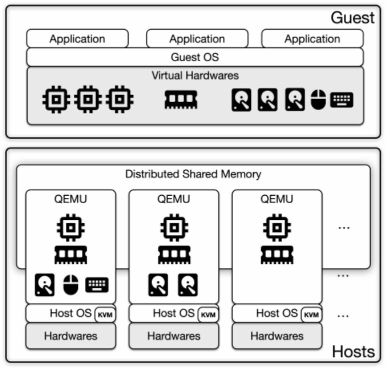

Figure 1. GiantVM architecture (taken from [7]).

Memory. GiantVM relies on the Distributed Shared Memory (DSM) approach [13] to ensure memory coherence. This approach is generic and can be applied to any memory portion as long as there is an access protection mechanism. On modern architectures, access to memory is controlled through the page table. GiantVM leverages nested paging [14] in KVM, which uses two page table layers for each VM. The first page table is called extended page table (EPT) and is used to map guest physical addresses to host physical addresses. The second page table is the one of the QEMU process (which represents the VM) from the point of view of the host kernel. The first page table is managed by KVM while the second one is managed by the host OS. In GiantVM, the DSM is only applied to the EPT. Thus only the accesses performed by the guest OS (vCPUs) to its memory are managed by DSM. Accesses performed by other QEMU threads (mainly peripheral ones) to the guest memory may not be coherent, thus need to be managed carefully.
Concerning the DSM, a page (when using the DSM) can have one of the following three states, also called the MSI protocol [2] : *Modified*, *Shared*, or *Invalid*. An Invalid page is not available locally and it needs to be fetched from the owner node. The page can either be requested in Shared or Modified mode. Shared mode is used when the page is to be accessed in read mode, and Modified mode is used when the page is to be accessed in read and write mode.
The DSM mechanism ensures that a page is accessible in the write mode on only one node. This is done by invalidating access rights to all other nodes. However, if a page is only accessed in read mode then multiple copy can coexist concurrently on multiple nodes. Thus a page within a node can either be in shared state (read mode), modified state (write mode) or invalid.

Devices. In GiantVM, devices are not distributed. They are all managed by a single node, called node 0. GiantVM implements an *event router* that is responsible of routing events from other nodes to the node 0.

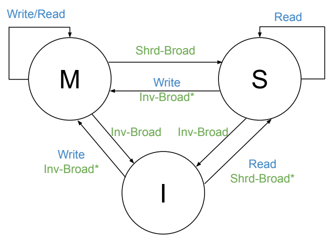

Figure 2. Description of the state diagram of the MSI protocol. In blue is the access of the process which provoke a transition, in green the actions of the protocol either to a local access or a remote one.

3 Performance Analysis

In this section we study the performance of GiantVM, and then provide optimizations based on our findings.

Experimental setup. It is important to notice that we want to measure the performance/impact of the software design, to determine how expensive the DSM used in GiantVM is, assuming that the network latency (requests latency) is small as possible. For this reason, we carried out the experiments on a single machine on which VMs communicate using a shared memory. This is the fastest communication mechanism that we can have. The used machine is DELL Latitude 5490 with 4 CPUs and 8 GB of RAM. The GiantVM virtual machine has 2 vCPUs, 2 GB RAM and runs a small debian distribution with Linux kernel 4.9.217. It is composed of two VMs, each having one vCPU and 1 GB of RAM. The benchmark is the startup followed by the shutdown of GiantVM. We consider this as a good starting point to evaluate GiantVM because of the following reason. When booting or terminating, the VM's guest OS accesses several memory pages, which may lead to several page faults due to DSM. This may lengthen boot and termination times. The need to start or terminate very quickly a VM in the cloud is very important for scalability [15].
We are interested in two metrics: the number of page faults (internal metrics) and the benchmark execution time. Each result presented in this paper is the average of five executions. The baseline is the execution of the VM without distribution.

Results. We counted 973,542 page faults generated during the execution of GiantVM vs 63,927 page faults with the baseline. As a consequence, it takes 17.19 seconds to boot and stop the distributed VM and 5.2 seconds for the baseline, which corresponds to about 3x performance degradation (note: GiantVM boot phase requires a time to synchronize the different instances which can go up to 4 seconds – can be further optimized). Fig. 3 shows how often guest page numbers (GPN) are subject to page faults. We observed that 91,773 distinct GPN are at the origin of the 973,542 page faults. 74% of GPN generate only one fault. 10 GPN generate

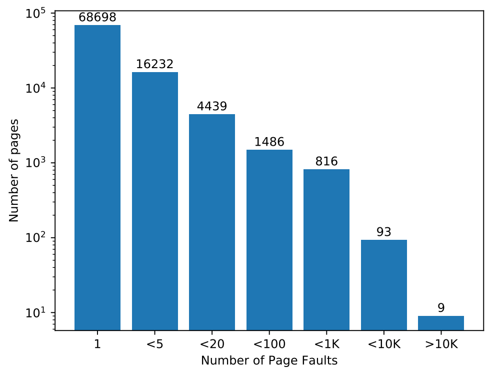

Figure 3. Number of page faults per group of pages

30% of all faults. The latter observation makes these 10 GPN critical for performance, thus justifying our attention below. Table 1 presents the number of faults for each of these 10 GPN. We can see that GPN *7685* is the most critical one, its proportion of faults is about 16% of the total faults generated by the 10 GPN. By analysing the kernel binary we found that GPN address 7685 holds the following variables:

```
ffffffff81e051c0 d bit_wait_table
ffffffff81e05040 d hpet
ffffffff81e05000 D jiffies
ffffffff81e05000 D jiffies_64
ffffffff81e05080 D mmlist_lock
ffffffff81e05180 d pidmap_lock
ffffffff81e05100 d softirq_vec
ffffffff81e050c0 D tasklist_lock
```

Page align optimization. Variable colocation within the same memory page may lead to false sharing in the DSM. This is a well known problem in the research topic of coherent processor cache implementation on SMP machines. It is generally handled by dedicating a cache line (the coherence unit) to each critical variable. Here is an illustration for `jiffies` variable in Linux source code

```
... unsigned long jiffies __cacheline_aligned_in_smp ...
```

In our context, we propose to dedicate a memory page to such variables. To do so, we adopt a straightforward approach which consists in just configuring the kernel to consider that a cache line is 4KB size (a page size). The cumulated size of these pages is about 32KB, which is very negligible compared to the total size of the VM (order of GB). By this way, all variables in the kernel which where parametrized by Linux developers to be aligned on a cache line will be assigned a dedicated memory page. This simple implementation reduces the total number of page faults by about 21%, which corresponds to about 39% in the reduction of the benchmark execution time.
Although this first optimization reduces the number of faults, we still have four GPN which generate several faults as shown in Fig. 4. Table 2 presents the number of faults for

GPN	507422	8843	507897	508947	1309719	508951	7753	7686	8842	7685
# PF	9592	11279	11713	11934	12432	13697	15165	20020	22218	164703

Table 1. The 10 more critical GPN. # PF means the number of page faults.

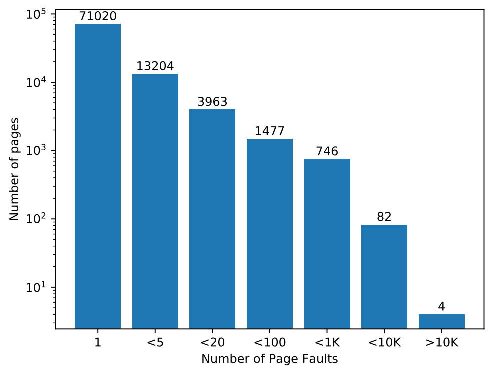

Figure 4. Number of page faults when the cache line size if configured to 4KB.

GPN	7685	7804	9013	7686
# PF	10611	13203	20755	100188

Table 2. The 4 more critical GPN when the cache line size if configured to 4KB.

each of these four GPN. By analysing Linux kernel code, we found that except GPN *9013*, the three others hold only one variable (as a consequence to our first optimization). The most faulted GPN (*7686*) holds `hpet`, GPN *7804* holds `rcu_sched_state`, and GPN *7685* holds `jiffies`. In the current state of progress of our work, we focused on GPN *7686*/`hpet` and GPN *7685*/`jiffies`, which are variables used by Linux to provide timer and clocking.

GPN* 7686/`hpet` *optimization. By analysing the piece of code that accesses the `hpet` timer, we found that there are two available implementations. The first one is simple as it consists in directly reading the raw value of the timer each time the kernel requests it. The second implementation, which is more scalable, uses a cache which is protected by a lock. We found that the latter is the root cause of faults generation in DSM. 64-bit Linux systems uses this second version by default. By switching to the first implementation, we eliminate this issue.

GPN* 7685/`jiffies` *optimization. `jiffies` is a variable which is frequently modified for clocking purposes in Linux. We observed that the Ivy synchronization algorithm (which is *write invalidate*) used by GiantVM is not appropriate for memory pages which hold such variables. Each modification by a node of the distributed VM generates a fault and invalidates the memory page on other nodes. We decided to investigate a different synchronisation algorithm, that we call *write update*, for this specific GPN. Note that the latter is known in advance as the kernel binary is known. `jiffies` is modified by `do_timer` function in Linux. We modify `do_timer` to realize a hypercall in order to ask the hypervisor to realize the modification of `jiffies`. The hypercall parameters include the GPN of the page which holds jiffies, the offset of the variable in that page, and the new value of `jiffies`. The handler of the hypercall, which runs inside the hypervisor, first locally updates the memory page, then asks the other nodes to realize the same operation. As a result, the guest OS (its vCPUs) never directly modifies `jiffies`, thus eliminating faults and invalidations in the DSM. The cost of the hypercall is negligible compared to the sum of the cost of the faults and invalidations generated by the DSM. This optimization reduces the number of page faults by about 17.05%

4 Discussion

In the previous section we learn that there are several factors affecting the performance of a distributed hypervisor, those can be improved by simple optimization in the guest OS software. However, we learned from our study that the design and architecture of distributed VMs should be changed, especially in term of DSM management. This is explained in this section.

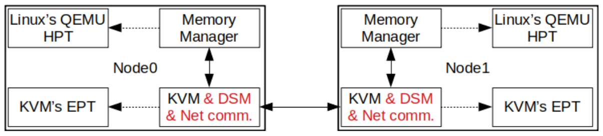

Figure 5. GiantVM simplified view

GiantVM simplified view: the DSM is linked with KVM. Fig. 5 presents a simplified view of the way GiantVM handles memory distribution. As the VM runs in QEMU, its memory is managed by *Linux Memory Manager*. Meanwhile, as KVM is in charge of memory virtualization, it manages EPT and does the necessary translations using the page table of QEMU. GiantVM implements the DSM directly into KVM. This makes GiantVM difficult to debug and improve. GiantVM hangs when the number of nodes is greater than two. We found that the main cause of GiantVM programming errors and the difficulty to handle them is the fact that GiantVM's DSM implementation is intertwined with KVM. This complicates the evolution of both code base: KVM is developed by the Linux community while GiantVM is developped by Zhang et al. [7].

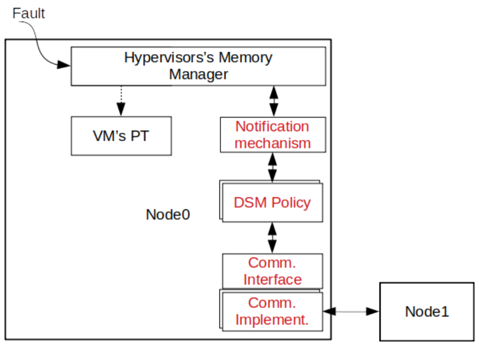

Figure 6. Independent DSM management

We have analyzed the distribution of modifications made by GiantVM authors in the KVM files. It appears that their code is distributed in many files in a not contiguous way. There are files that contain minor modifications(return values of functions for instance) and some others (like `mmu.c`) that has the highest number of non contiguous modified blocs. This study shows how much GiantVM and KVM are intertwined. Moreover, since GiantVM uses the version `4.9.76` of the Linux Kernel, it will be complicated to make it evolve.
To summarize, GiantVM's design is not generic: (1) it is linked to KVM, more precisely its Intel implementation; (2) it can only work on machines with EPT (Extended Page Technology) features; (3) GiantVM applies Ivy DSM protocol to the whole VM while we see that *write update* could improve the system for some kernel variables.

Use a file system. To address these challenges, we suggest that we extract the DSM management from the hypervisor, for it to be easier to improve and make the system more modular. This is depicted in Fig. 6. Here, when a page fault rises, the handler transfers the fault handling to the memory manager of the hypervisor. To handle the fault, the memory manager can call the DSM. To this end, we propose that hypervisor's memory manager should allow the ability to register callbacks for DSM implementations. After its execution, the DSM uses a communication library to contact other nodes. Several implementations of the communication interface can be used by the DSM. We can cite Ethernet, Infiniband RDMA, shared memory, etc.
The DSM is implemented as a file system which is used to back all memory sections representing the guest OS memory. Thus, the memory manager calls the DSM using the Linux Virtual File System API. To realize the reverse path, the DSM registers with the Linux memory manager using its MMU notifier framework. It also allows the implemention and utilization of several DSM synchronization policies.
The usage of such a system would help GiantVM to be improved with regards to its design.

5 Related work

Providing the illusion of a single system image (SSI) to a distributed OS has been a hot topic in the past. This can be achieved either in the hardware, the hypervisor, or the OS. The survey made by Healy et al. [16] in 2016 provides a good historical perspective. Unlike our work, most prior solutions have been realized in the context of non virtualized machines.

Hardware-level solutions. We found both commercials [17–19] and academia solutions [20–22]. These systems provide to the processors transparent access to several memory units and peripherals of the platform. The main disadvantage of these systems is that they are highly expensive making them more suitable for HPC-like environments rather than the cloud.

OS-level solutions. The 90s have seen the development of most SSI systems [23–28]. These systems implemented only a limited transparency level due to the tremendous number of components to distribute (process, file descriptors, sockets, proc file system, ...). More recent works took into account heterogeneous systems [29, 30] which are more commonly found in cloud-like environments. Finally, other works [31] envisioned SSI for disaggregated racks. The disadvantage of these solutions is that they are OS specific and they do not take into account virtualization.

Hypervisor-level solutions. GiantVM [7], the system that we study in this paper, is the most recent SSI implementated at the hypervisor-level. Among other projects [32–35], we found vSMP from ScaleMP [36], an active commercial solution on which no documentation is publicly available. The main limitation of these solutions is that they are highly rigid (specific architecture, OS, static number of sub virtual machines, etc.). Our proposed design is generic and flexible.

6 Conclusion and future works

We analyzed the performance and the design of giantVM, a KVM-based distributed hypervisor. About performance, we proposed several optimizations that allow to accelerate VM boot and termination times. These optimizations share the same goal which is the minimization of the number of page faults caused by critical kernel's data structures. We are currently extending this approach to user space applications. In fact, critical data structures of user space applications can be identified during the pre-production phase, where the application is tested with several typical workloads.
Concerning the design contribution, we proposed to extract the DSM management from KVM, such that the debugging, and the evolution of both systems would be smoother. To achieve that, we proposed that the hypervisor provides an interface with callbacks that the DSM manager would register to in order for them to be linked and to communicate.

As a future work we plan to build a KVM-based distributed hypervisor, that shall not manage the DSM of the distributed virtual machines.

7 Acknowledgements

This work was funded by the **ScaleVisor** project of Agence Nationale de la Recherche, number ANR-18-CE25-0016, and IDEX IMPULSION SprinkledVM OPE-2020-0018

References

[1] Ambati Pradeep et al. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *Proc. of USENIX OSDI*, 2020.

[2] Vlad Nitu et al. Welcome to Zombieland: Practical and Energy-Efficient Memory Disaggregation in a Datacenter. In *Proc. of ACM EuroSys*, 2018.

[3] Peter X. Gao et al. Network Requirements for Resource Disaggregation. In *Proc. of USENIX OSDI*, 2016.

[4] Emmanuel Amaro et al. Can Far Memory Improve Job Throughput? In *Proc. of ACM EuroSys*, 2020.

[5] Juncheng Gu et al. Efficient Memory Disaggregation with INFINISWAP. In *Proc. of USENIX NSDI*, 2017.

[6] Zhenyuan Ruan et al. AIFM: High-Performance, Application-Integrated Far Memory, 2020.

[7] Jin Zhang et al. GiantVM: A Type-II Hypervisor Implementing Many-to-One Virtualization. In *Proc. of ACM VEE*, 2020.

[8] Yizhou Shan et al. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proc. of USENIX OSDI*, 2018.

[9] Kevin Lim et al. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proc. of ISCA*, 2009.

[10] Chenxi Wang et al. Semeru: A Memory-Disaggregated Managed Runtime. In *Proc. of USENIX OSDI*, 2020.

[11] Alain Tchana et al. Rebooting Virtualization Research (Again). In *Proc. of ACM APSys*, 2019.

[12] Madeleine Glick et al. Silicon Photonics Enabling the Disaggregated Data Center. In *Advanced Photonics*, 2018.

[13] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):334–341, November 1989.

[14] Nested paging. http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf. Accessed: 2021-08-5.

[15] Vlad Nitu, Pierre Olivier, Alain Tchana, Daniel Chiba, Antonio Barbalace, Daniel Hagimont, and Binoy Ravindran. Swift birth and quick death: Enabling fast parallel guest boot and destruction in the xen hypervisor. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, page 1–14, New York, NY, USA, 2017. Association for Computing Machinery.

[16] Philip Healy, Theo Lynn, Enda Barrett, and John P. Morrison. Single system image: A survey. *Journal of Parallel and Distributed Computing*, 90-91:35–51, 2016.

[17] Tom Lovett and Russell Clapp. Sting: A cc-numa computer system for the commercial marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 308–317, 1996.

[18] James Laudon and Daniel Lenoski. The sgi origin: a ccnuma highly scalable server. *ACM SIGARCH Computer Architecture News*, 25(2):241–251, 1997.

[19] Michael Woodacre, Derek Robb, Dean Roe, and Karl Feind. The sgi® altixtm 3000 global shared-memory architecture. *Silicon Graphics, Inc*, 44, 2005.

[20] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, W-D Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The stanford dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[21] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, et al. The stanford flash multiprocessor. In *Proceedings of 21 International Symposium on Computer Architecture*, pages 302–313. IEEE, 1994.

[22] Bishop C Brock, Gary D Carpenter, Eli Chiprout, Mark E Dean, Philippe L De Backer, Elmootazbellah N Elnozahy, Hubertus Franke, ME Giampapa, David Glasco, James L Peterson, et al. Experience with building a commodity intel-based ccnuma system. *IBM Journal of Research and Development*, 45(2):207–227, 2001.

[23] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The locus distributed operating system. *ACM SIGOPS Operating Systems Review*, 17(5):49–70, 1983.

[24] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, et al. Overview of the chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, 1992.

[25] David Cheriton. The v distributed system. *Communications of the ACM*, 31(3):314–333, 1988.

[26] Sape J. Mullender, Guido Van Rossum, AS Tanenbaum, Robbert Van Renesse, and Hans Van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, 1990.

[27] Ang Goscinski, Michael Hobbs, and Jackie Silcock. Genesis: an efficient, transparent and easy to use cluster operating system. *Parallel Computing*, 28(4):557–606, 2002.

[28] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, Gaël Utard, Ramamurthy Badrinath, and Louis Rilling. Kerrighed: a single system image cluster operating system for high performance computing. In *European Conference on Parallel Processing*, pages 1291–1294. Springer, 2003.

[29] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, volume 27, 2008.

[30] Antonio Barbalace, Binoy Ravindran, and David Katz. Popcorn: a replicated-kernel os based on linux. In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2014.

[31] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 69–87, 2018.

[32] Stefan Lankes, Pablo Reble, Carsten Clauss, and Oliver Sinnen. The path to metalsvm: Shared virtual memory for the scc. In *Proceedings of the 4th many-core applications research community (MARC) symposium*, volume 55, pages 7–14, 2011.

[33] Jinbing Peng, Xiang Long, and Limin Xiao. Dvmm: A distributed vmm for supporting single system image on clusters. In *2008 The 9th International Conference for Young Computer Scientists*, pages 183–188. IEEE, 2008.

[34] Kenji Kaneda, Yoshihiro Oyama, and Akinori Yonezawa. A virtual machine monitor for providing a single system image. *URL http://web. yl. is. su-tokyo. ac. jp/ kaneda/dvm*, 2005.

[35] Matthew Chapman and Gernot Heiser. Implementing transparent shared memory on clusters using virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 383–386, 2005.

[36] Scalemp vsmp release notes. .https://files01.scalemp.com/downloads/vSMP_Foundation/8.1/8.1.1145.4/vsf-8.1.1145.4.rel_notes.txt.. Accessed: 2021-08-06.