508 Proposal - Disaggregated Simulations and extensions

Amanda Carbonari
acarb95@cs.ubc.ca
Stewart Grant
sgrant09@cs.ubc.ca
Fabian Ruffy
fruffy@cs.ubc.ca

KEYWORDS

Disaggregation, Network Simulation, Redundancy

ACM Reference format:
Amanda Carbonari
acarb95@cs.ubc.ca, Stewart Grant
sgrant09@cs.ubc.ca, and Fabian Ruffy
fruffy@cs.ubc.ca . 2016. 508 Proposal - Disaggregated Simulations and extensions. In *Proceedings of Operating Systems 508
, AndyLand UBC, Sept-Dec, 2017 (508 UBC),* 3 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Today's datacenters (DCs) are server-centric: users rent servers with specific hardware capabilities tailored to their needs (e.g., compute-intensive EC2 instances from Amazon). A *disaggregated* datacenter (DDC) disaggregates, or separates, the resources in a traditional DC into resource *blades*, with each resource connected directly to an interconnect (Figure 1). We use the term *blade* to describe a 1U server containing one resource type, and *resource* for an individual CPU, DIMM, SSD, etc. in a blade.

The modularity of DDCs benefits both operators and users [5]. By separating resources into blades, a DDC provides efficiency: the operator can upgrade specific hardware blades without impacting other resources types. Users also experience an efficiency win: in a DDC a user can provision the exact amount of resources they require and dynamically expand/shrink this set. This flexibility also increases the DC resource utilization.

There are two broad types of disaggregation: partial and full. In partial disaggregation compute resources have a small amount of local memory, memory and storage resources have CPUs, and a NIC is attached to each resource. In full disaggregation each resource is independent and is directly connected to the rack interconnect. Current research focuses on the practicality of partial disaggregation [1, 4, 7, 8].

In this paper, we discuss partial disaggregation at rack-scale: resources can only connect to other resources in the same rack (e.g., a CPU in rack1 cannot connect to a DIMM in rack2). The disaggregated rack is presented to the client as a single machine through a virtualization layer. Applications can request a number of VMs to allocate among the racks. Distributed applications might then run among several disaggregated racks, while a single-machine application will reside in one rack.

508 UBC, AndyLand UBC
2016. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

Figure 1: Simplified representation of a disaggregated datacenter with three communication points: (1) between the application and the virtualization layer, (2) between the virtualization layer and the physical resources, and (3) between the physical resources bundled as blades.

Disaggregation requires a re-thinking of failure assumptions core to existing application designs. We expect that DDC designs will make it possible for resource failures to no longer fate-share: a failure of a memory resource will not cause the failure of the CPU that was using that memory resource. In a traditional DC, complex applications take on the responsibility of dealing with server failures. But, such legacy applications are not disaggregation-aware, and should not be expected to handle individual resource failures in a DDC. So, how should the CPU respond when it cannot write to a memory resource? And, more importantly, should a developer deploying their code in a DDS be expected to reason about and account for failures of individual resources?

For example, distributed data processing platforms like Hadoop, Naiad, and Spark use system-specific fault tolerance schemes to recover from server failures. However, they cannot recover from resource failures [2, 9, 12]. In the case of resource failure, the

Amanda Carbonari
acarb95@cs.ubc.ca, Stewart Grant
sgrant09@cs.ubc.ca, and Fabian Ruffy
fruffy@cs.ubc.ca

entire application would fail, leading these applications to run inefficiently on Dec's. It's not unreasonable to expect legacy DC application to run in DDCs without modification but still take advantage of DDC modularity. Since such legacy, disaggregation-unaware, applications cannot efficiently handle resource failures, DDCs must provide infrastructure and system support for coping with such failures.

In this paper we explore the co-design of DDC fault tolerance with the network by expanding the role of software defined networking (SDN). An SDN controller has a global view of the network and can dynamically reconfigure routing [3, 6, 11]. Both features are useful in managing DDC resources. A controller and the switch can be implemented such that the controller is notified of new resources, the health of each resource, etc. Therefore, the controller can collect information about each resource in its rack. The dynamic flow rule changes will allow the network to react to events, such as failures or resource additions. The above is not difficult to achieve as many DCs already use SDN-capable ToR switches.

We believe that one key choice in designing fault tolerance for DDCs is the *granularity of resource fate sharing*. Since DDCs present an environment where no fate sharing exists, what should be exposed to the application? Should the failures be visible or transparent? And, if they are visible, should they be presented as a server failure? We believe that network-driven failure recovery in DDCs can be used to present legacy application with types of failure that can appear in a traditional DC (e.g., server failure). We break down possible fault tolerance designs based on the granularity of fate sharing (Figure **??**): VM-level (complete fate sharing), process-level (partial fate sharing), and no fate sharing. We believe that these three granularities, covered in more detail in Section **??**, are sufficient for most legacy applications.

Some applications, particularly those designed for a disaggregated environment may benefit from a fate sharing granularity that is different from the three types above. A file system that prioritizes strong consistency over availability may want to fate share a CPU resource that serializes operations with all but one disk resource. This system would retain strong consistency guarantees even if the serializing mechanism fails by retaining a single (serializing) disk resource, and failing the other disk resources. Considering a broad diversity of fate sharing models, how can DDCs provide fault tolerance? We argue that fate sharing granularity and failure mitigation should be *programmable*, allowing applications to choose the most appropriate fate sharing arrangement.

There are two places where this programmability can be implemented: the application and the (software defined) network. We argue that the network is the right place for this programmability. First, failure detection and mitigation can be more easily and more efficiently implemented by the network, especially in a DDC where all inter-resource communication is observable by the network. Second, the network is a natural place to enforce DDC resource fate sharing. Moving programmability into the network appears to be contrary to the end-to-end principle [10]. We are not, however, proposing to remove application-level failure handling entirely; for example, our proposed mechanisms will not handle a broad range of faults such as byzantine failures (these must still be handled by the application). But we do argue that, in line with the end-to-end principle, certain resource faults can be more efficiently dealt with by the network.

In summary, we describe existing DC fate sharing models ported to DDCs using SDN in the context of several sample applications. We then describe requirements for new fate sharing models made possible by DDCs and the applications that can benefit from these. Through our analysis, we noticed that different classes of applications require different fate sharing semantics. Some applications can benefit from previously unattainable fate sharing models with small changes. To achieve the full benefit of disaggregation for both legacy applications and new applications, we argue that fate sharing must be both *programmable* and *solved by the network*. We sketch out a design for how this might be achieved and discuss outstanding challenges.

2 PROPOSAL

In this section we propose experimental features of a disaggregated memory system, and extensions to an existing distributed shared memory system which will allow developers to write more expressive applications on top of it.

Disaggregated memory striping

In a disaggregated rack the memory allocated to a single application may be resident on separate machines. This begs the question, what are the operational semantics for a failure of part of a memory allocation? A variety of options are available; The traditional approach would see the application share the fate of its memory, and simply crash. In contrast the contents of the memory could be replicated, and in the face of a failure, a memory replica is transitioned to transparently. We propose to use an approach already common in disks, mainly the RAID architecture.

Our proposal is to implement main memory raiding at the page level. When an application page fault occurs, the page fault is intercepted by a raiding manager which implements a given raid algorithm, either (0, 1 or 6), and stripes the evicted page across memory blades. To retrieve pages, the page manager requests the striped page from memory blades and checks for the pages correctness. In the case of a failure, our proposed system has the same semantics as traditional raid. If a page is missing, it is reconstructed using parity raid 2-6. Or in the case of raid 0 the data is lost. In the face of truly lost data (raid 0 or 1) an entire application should share the fate of the memory blade.

Using main memory raid has multiple attractive properties. As already noted raid allows for correctness in the face of partial failures. In addition to failures, bit level memory correctness is also checked which allows for optimization's to be made at the network later. For example remote pages need not use TCP checksums. Finally this architecture has the advantage that memory bandwidth is multiplied almost linearly by the number of raided memory blades.

To evaluate our disaggregated memory raided system we will implement popular graph processing algorithms (label propagation, and pagerank) and measure their performance relative to benchmarks reached by other frameworks, such as Naiad, and GraphX. Additionally we will measure the rate of recovery of applications in the face of failures.

Multi-Threaded Applications

To date our distributed memory simulator only has support for single threaded applications, which share no resources. In order to investigate the appropriate OS semantics for applications written on a disaggregated architecture memory sharing and multi-threading are necessary. Adding these additional features requires re-engineering a portion of the Blue Bridge distributed shared memory system.

In the current architecture a single thread is allocated a local scratch space, when the scratch space is filled, and an application page faults, a request for distributed memory is made. The request has no information about the thread which issued the request or the scope of its distributed memory access. We propose an extension to the distributed memory allocation protocol, in which requests are identified by their memory offset, and the thread which issued the request. Additionally the system will require extensions to the memory manager so that multiple local scratch spaces can be maintained concurrently.

The addition of multi threaded applications raises the problem of isolation. Currently BlueBridge gives a single thread unlimited access to a 128 bit address space. In order to implement isolation, a manager must maintain an offset on which a given thread can operate on all memory blades. In essence this extension is simplified virtual memory for a distributed system.

With the addition of isolation comes the need to facilitate sharing. We propose the use of named buffers at known locations to allow sharing between isolated threads. Additionally two threads will have the ability to operate in the same address space for efficient computation.

DPDK -Real applications will require real benchmarks -Faster is always better -We can get actual wins using DSM

Switch MMU

Access to distributed memory is currently administered by a user space process collocated with an executing application. This application is trapped into when page faults occur, and it issues requests for remote memory. While practical, this approach does not take advantage of key aspects of disaggregated architecture at the scale of a rack.

In a pure disaggregated architecture, CPUfis do not maintain a cache of data, all memory requests are remote. This property causes all memory access requests to pass through a TOR. This routing property allows for memory management to be centralized on a TOR.

-This part is weak.

REFERENCES

[1] Intel, Facebook Collaborate on Future Data Center Rack Techologies, 2013. https://newsroom.intel.com/news-releases/intel-facebook-collaborate-on-future-data-center-rack-technologies/.

[2] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[3] N. Feamster, J. Rexford, and E. Zegura. The Road to SDN: An Intellectual History of Programmable Networks. *CCR*, 44, 2014.

[4] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, I. Osdi, P. X. Gao, A. Narayan, S. Ratnasamy, J. Carreira, and S. Shenker. Network Requirements for Resource Disaggregation. In *OSDI*, 2016.

[5] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network Support for Resource Disaggregation in Next-generation Datacenters. In *HotNets*, 2013.

[6] M. Jarschel, T. Zinner, T. Hoßfeld, P. Tran-Gia, and W. Kellerer. Interfaces, attributes, and use cases: A compass for SDN. *IEEE Communications Magazine*, 52(6):210–217, 2014.

[7] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *ISCA*, 2009.

[8] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch. System-level implications of disaggregated memory. In *HPCA*, 2012.

[9] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[10] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 1984.

[11] S. Sezer, S. Scott-Hayward, P. Kaur Chouhan, B. Fraser, D. Lake, C. Systems Jim Finnegan, N. Viljoen, N. Marc Miller, N. Rao, and S.-h. Layout. Are We Ready for SDN? Implementation Challenges for Software-Defined Networks. *Future Carrier Networks*, 51(7), 2013.

[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.