One Cluster, One OS:
Unified Distributed OS for Commodity Clusters

Arjun Vikram
Stanford Computer Science
Stanford, CA
arjvik@cs.stanford.edu

***Abstract*—Modern distributed compute clusters, such as research supercomputers, typically expose each physical node as an independent unit, with its own operating system, address space, and process space, along with some form of high-bandwidth remote memory access and communication interconnect linking them. While this design offers flexibility and failure isolation, it forces distributed applications to manage explicit inter-node communication and data movement, necessitating the use of programming models such as MPI or cluster management systems such as Kubernetes that lead to significant system complexity. In this paper, we present a unified distributed bare-metal Single-System Image operating system that virtualizes a cluster as a large-scale NUMA machine, requiring only RDMA support. By providing one global process space and one shared virtual address space across all nodes, it allows NUMA-aware applications to run with trivial modifications atop a commodity cluster. We demonstrate its effectiveness running on several low-powered Raspberry Pi Zero Single-Board Computers, with only software-emulated RDMA. We show this design greatly simplifies parallel programming on clusters while preserving efficiency, and we discuss opportunities for productionizing our approach.**

***Index Terms*—Bare Metal, Distributed Systems, Single-System Image**

I. Introduction

Today's "Supercomputers," found across research labs and industry and used for everything from large-scale simulation to machine learning jobs, are no longer individual machines, but rather large clusters of each reasonably sized server-class machines. Put together, these supercomputers can contain more than an exaflop of compute power, spread across up to tens of thousands of individual CPU cores. However, writing applications that can take advantage of this scale is still a challenge, and even the simplified linear algebra routines used by the TOP500 benchmark [1] of supercomputer potential cannot reach the theoretical peak performance of these machines due to the overhead associated with parallel programming.

Each individual node in these clusters runs its own operating system, typically Linux or some UNIX derivative. Each node operates with its own address space, process space, local filesyste, and network stack. The only indication that these nodes are part of a larger cluster is the presence of a high-bandwidth remote memory access (RDMA) [2] interconnect, such as InfiniBand [3] or RoCE [4], which allows for low-latency communication between nodes. Through the provided interconnect, these nodes can be given a unified network filesystem, job scheduling ability, and the promitives for message passing across nodes. This design has been the standard operating practice for the field of High-Performance Computing (HPC), with custom HPC-specific software stacks tailored for everything from network filesystems to job scheduling to message passing frameworks. While the HPC paradigm is effective for many workloads, it introduces significant complexity for developers who must manage the distributed nature of their applications.

In this paper, we present an approach to distributed computing on clusters that simplifies the programming model by unifying nodes of the cluster into a single programmable system, where the lines between machines are almost invisible to the programming model. We demonstrate that the only primitive required to achieve this is RDMA support, and implement a bare-metal prototype using just software-emulated RDMA on a cluster of Raspberry Pi Zero single-board computers.

A. Multi-System Clusters

The traditional approach to building a cluster of treating individual machines as independent systems requires a complex software stack to combine the computing power of each node into one "Supercomputer" or unified cluster.

One programming model frequently used in HPC is Message Passing, typically implemented via the Open Message Passing Interface (OpenMPI) [5]. Message Passing allows for communication between nodes by sending messages over the network, and frameworks like OpenMPI abstract away the details of message passing across the underlying partially connected graph of nodes. This programming model, however, draws sharp boundaries between nodes, requiring the programmer to explicitly manage the distribution of data and computation both within nodes and across nodes as two distinct levels.

Another programming model, used more frequently outside of the HPC space, is cluster management systems such as Kubernetes. Kubernetes allows for easy communication between nodes by placing processes on each node inside containers and connecting the containers through a virtualized network with centralized discovery and load-balancing support. This approach simplifies task orchestration, but still prevents containers from spanning across nodes. The Kubernetes ecosystem has developed a mechanism to migrate containers across nodes, called CRIU [6], aimed at fault tolerance and load balancing, but this is not a common use case and requires significant overhead to implement. We take inspiration from CRIU's snapshotting and migration design, but are able to make it far more efficient by leveraging unified address and resource spaces rather than needing to replicate independent an address space and resource namespace set.

B. Single-System Images

Alternatively, a Single-System Image (SSI) cluster [7] is a cluster that presents itself as one large machine to the user, with one global process space and one shared virtual address space across all nodes. Perhaps the best known attempt is the Plan9 operating system, which unifies the I/O space of all nodes in a cluster but doesn't allow for seamless process migration or memory sharing. The most popular full-fledged SSI operating system is the OpenSSI project, which attempts to extend the Linux kernel into an SSI cluster.

Modern multi-chip systems already employ a heirarchical memory architecture, with multiple chips connected by a high-bandwidth interconnect, and each chip containing multiple CPU cores with their own local memory. This architecture is known as Non-Uniform Memory Access (NUMA) [8], and it allows for efficient communication between cores on the same chip while still allowing for communication between cores on different chips. In a typical NUMA system, each chip can access the memory controllres of other chips without going through the chip most local to the controller, at the cost of higher latency and lower bandwidth; NUMA-aware applications are responsible for placing data and processes on the appropriate chips to minimize latency and maximize bandwidth. NUMA-unaware applications can still run unmodified on NUMA systems, and a NUMA-aware operating system can ensure that single-chip programs allocate physical memory assigned to the most local memory controller.

In this paper, we take inspiration from NUMA's approach to a distributed compute API. We treat a Single-System Image cluster as a large-scale NUMA machine, with each node being a NUMA node. Where possible, we employ RDMA for non-local memory access, and we provide a coherent single-system view for applications with transparent process migration between hosts wherever needed. We also provide NUMA-like semantics to allow SSI-aware applications to effectively distribute their resources across the cluster while allowing unaware applications to run unmodified.

We begin with our approach to the most compelling feature of an SSI cluster:

II. Process migration

In an SSI system, kernelspace and userspace are differentiated in that each node has a kernelspace with identical but distinct kernel address spaces, while userspace operates with a unified address space across all nodes. A highly desirable feature of an SSI cluster is the ability to transparently migrate userspace processes between nodes.

In the context of a userspace process, the only relevant ephemeral process state is contained in the processors' registers. That is, a process can be snapshotted by persisting the contents of every register, and resumed on the same or another machine by restoring the contents of the registers. Because the instruction pointer/program counter is contained as a register, the process is resumed at the same point it was snapshotted, and thus can continue exactly where it left off with no loss of state.

More difficult to handle restoration of is the process' memory state. Upon process migration, we choose not to migrate any pages of memory at all —instead, the newly migrated process is given a freshly empty virtual address space. The process will immediately page fault (instruction prefetch fault) before running its first instruction, as the page containing its executable code will not be mapped yet. Upon this page fault, the kernel, recognizing that the process was migrated, will copy the page of memory from the original node to the new node over RDMA, placing it at some physical address in the new node's memory but mapping it to the same virtual address as before. The process can then resume running, only page faulting again for further data or code pages that need to be accessed.

Modern speculative CPUs perform page table walks in parallel with instruction execution, though they do not execute a page fault handler until actually arriving at an instruction falling in an unmapped page or one that accesses an unmapped page [9]. Unfortunately, this means that the RDMA accesses cannot be performed in parallel with instruction execution, and so each new page of memory used by the process requires a page's worth of RDMA latency to access. However, RDMA accesses are not blocking [10], and thus the kernel can schedule a new thread to run while the RDMA transfer is in flight, resuming the requesting process only once the RDMA request's completion triggers an interrupt.

A. Supertable Bookkeeping

In this design, the source and destination node of a migration need to be aware the "page supertable" of each process, which maps virtual pages to tuples containing a node ID, physical page address, and both a valid and dirty bit). Multiple nodes can hold copies of the same page while it remains clean, but once a page becomes dirty by being modified on one node, it must be invalidated on all other nodes (for an example sequence of this, see Figure 1).

The bookkeeping database of memory state is a simple distributed key-value store, and there are numerous algorithms for maintaining a consistent key-value store across distributed nodes [11], [12].

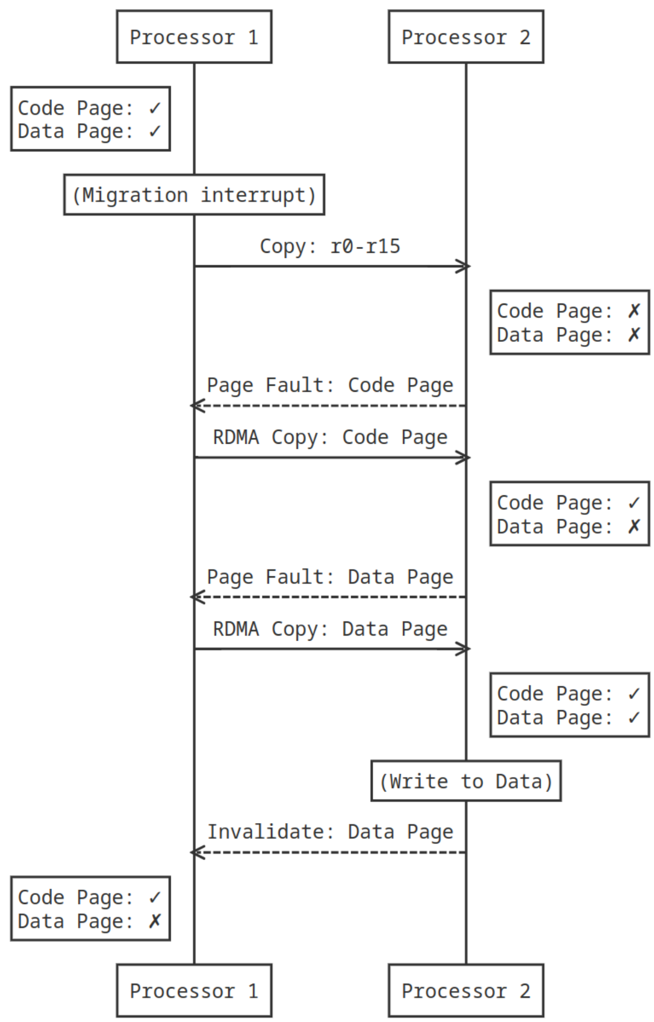

Fig. 1. Sequence diagram of an example process migration. The migrated process first obtains clean copies of its code and data pages from the source node, then modifies the data page memory, resulting in invalidating the page stored on the source node in case the process is ever migrated back.

Removing any requirements on fault tolerance allow for an even simpler leader-follower implementation.

III. Scheduling

Once process migration is possible, it becomes possible to create a cooperative threading system that allows threads to be scheduled across nodes. The goal of such a system is to complete threads $\min(N, T)\times$ faster (where N is the number of nodes and T is the number of threads) by distributing threads across all nodes.

We implement a traditional cooperative threading system, maintaining a ready queue of threads to be scheduled. The mechanism for yielding and resuming threads is almost identical to that of process migration, meaning that we can employ the same state save and state restore code by "migrating a process to the same node" in order to thread switch. To maintain liveness, we employ a distributed linked list as the unified ready queue shared by the entire cluster, sharding the values of the queue (process state) such that they are only stored on the node enqueueing them. Thus the only communication for an enqueue is the update of a "next node" pointer on the node with the previous queue tail entry, and a global update of the tail entry pointer. Dequeueing a thread from this queue can require copying the register state struct from the node that enqueued it, but this can be done quickly over RDMA.

A. Distributed Locks

Thread synchronization is a major challenge in a system like this, and we provide a distributed lock primitive to address this. To simplify conflict resolution, each lock is assigned a master node, and all requests to aquire and release it are arbitrated by that node. If a process attempts to aquire a lock that is already held, it will be placed into a wait queue for that lock on its master node, and enqueued into the ready queue by the master node once the lock is released.

B. Pinned Threads

Certain operating system resources, such as memory-mapped IO (MMIO), have physical addresses that are only valid on the node they were created on. To support these resources, we pin any thread that maps these memory addresses to the node they were created on, preventing them from being unintentionally migrated. To allow for node-agnostic memory-mapped IO, we also map a copy of the MMIO pages at a second virtual address on each node, and do not consider accesses to these pages a request for pinning a thread. This allows for the traditional demo of an LED blinking on one node and then resuming blinking on another node upon migration.

C. Scheduling Policy

As the scheduling policy is not the focus of this paper, we implement a simple round-robin scheduling policy that distributes threads evenly across all nodes. This policy is sufficient for most workloads, but more sophisticated scheduling policies could consider affinity of threads to the nodes they were last run on in order to minimize RDMA latency.

IV. Implementation

To demonstrate the feasibility of our design, we implement a prototype bare metal operating system on a two-system cluster of ARMv6 Raspberry Pi Zero W v1.1 Single-Board Computers (SBCs). Since the Raspberry Pi Zero does not have hardware support for RDMA, we implement a software-emulated RDMA protocol in the form of an interrupt handler running in kernelspace on

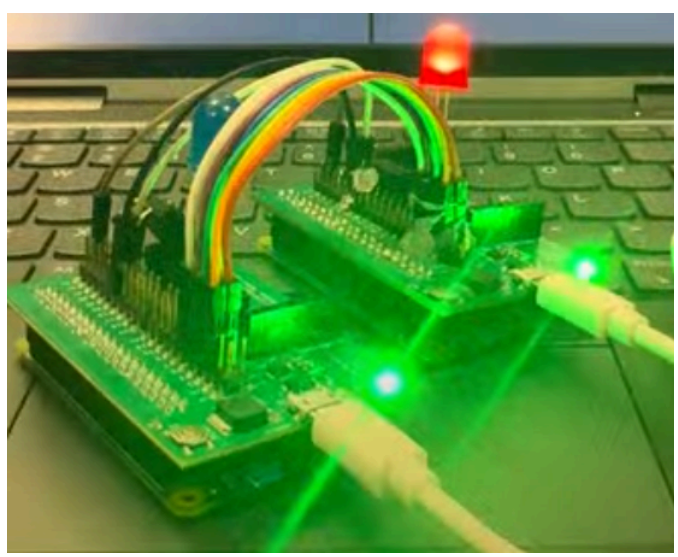

Fig. 2. A two-node cluster of Raspberry Pi Zero SBCs connected by a parallel cable. Pictured is a LED blink program after it has been migrated from one node to the other.

each node. For the purpose of demonstration, we connect eight GPIO pins on each node to each other to form a parallel cable (Figure 2), and add two additional connected pins to act as ready and valid bits, resulting in an AXI-like communication bus between the two Pi nodes. For simplicity sake, we disabled all caching at the hardware level instead of worrying about cache coherence. However, we're confident that cache coherence awareness is possible to implement in the same way we performed page invalidation upon modification.

We implement a simple kernel that supports process migration, cooperative threading, and distributed locks. We are able to successfully run programs that spawn tens of threads across the two nodes and blink LEDs on each. As a cute demo, we spawn an odd number of threads that blink LEDs in discernable patterns, and alternatingly yield between them so each pattern flashes on both nodes!

V. Conclusion and Future work

Our major contribution in this work is a RDMA-based framework for virtualizing individual nodes in a cluster as NUMA nodes in a large-scale NUMA machine. While we presented a rudimentary implementation of this in order to demonstrate its feasibility, we believe that there is significant room for improvement in building a production-ready system. In future work, we plan to incorporate cache coherence awareness, more sophisticated scheduling policies, and a more robust distributed key-value store for process state management. We also plan to explore the use of hardware DMA and RDMA support in order to further push the performance of our system.

Acknowledgements

Thank you to Professor Dawson Engler, for developing and teaching the course CS140E Operating Systems from Scratch that inspired this project, and for all the advice and support in working on it. As my CS140E final project, I developed the process migration compnent of this project. Thank you to the teaching staff of CS140E, especially Joe Tan, for their invaluable support in designing and debugging my final project, without which this paper would not have been possible. Finally, thank you to the Stanford Computer Science Department for supporting me as a TA for the current iteration of CS140E as I worked on this project, and for supporting my work on this as my Senior Project.

References

[1] E. Strohmaier, J. J.~Dongarra, H. D.~Simon, M. Meuer, and H. W. Meuer, "TOP500: The List of the World's Fastest Supercomputers." Accessed: Jun. 09, 2025. [Online]. Available: https://top500.org/

[2] J. Hu, H. Shen, X. Liu, and J. Wang, "RDMA Transports in Datacenter Networks: Survey," *IEEE Network*, vol. 38, no. 6, pp. 380–387, 2024, doi: 10.1109/MNET.2024.3397781.

[3] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, "High performance RDMA-based MPI implementation over InfiniBand," in *Proceedings of the 17th Annual International Conference on Supercomputing*, in ICS '03. San Francisco, CA, USA: Association for Computing Machinery, 2003, pp. 295–304. doi: 10.1145/782814.782855.

[4] C. Guo *et al.*, "RDMA over Commodity Ethernet at Scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, in SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 202–215. doi: 10.1145/2934872.2934908.

[5] E. Gabriel *et al.*, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, Sep. 2004, pp. 97–104.

[6] S. Pickartz, N. Eiling, S. Lankes, L. Razik, and A. Monti, "Migrating LinuX Containers Using CRIU," in *High Performance Computing*, M. Taufer, B. Mohr, and J. M. Kunkel, Eds., Cham: Springer International Publishing, 2016, pp. 674–684.

[7] R. Buyya, T. Cortes, and H. Jin, "Single System Image," *The International Journal of High Performance Computing Applications*, vol. 15, no. 2, pp. 124–135, 2001, doi: 10.1177/109434200101500205.

[8] C. Lameter, "NUMA (Non-Uniform Memory Access): An Overview: NUMA becomes more common because memory controllers get close to execution units on microproces-

sors.," *Queue*, vol. 11, no. 7, pp. 40–51, Jul. 2013, doi: 10.1145/2508834.2513149.

[9] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: a mechanism for speculative address translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, in ISCA '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 307–318. doi: 10.1145/2000064.2000101.

[10] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance {RDMA} systems", in *2016 USENIX annual technical conference (USENIX ATC 16)*, 2016, pp. 437–450.

[11] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.

[12] L. Lamport, "The Part-Time Parliament," *ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169. Also appeared as SRC Research Report 49. This paper was first submitted in 1990, setting a personal record for publication delay that has since been broken by [60].*, May 1998, [Online]. Available: https://www.microsoft.com/en-us/research/publication/part-time-parliament/