mycelium
backendalpha

Creating a pseudo-instruction-set for scalable distributed computing

Zach Goethel

Introduction

Composability is the new hotness in the compute sphere. Admins find it convenient to isolate the finite resource groups of a computer, collect them in bulk, then slice them back up into VMs. Allocate rack space to high disk capacity. Create a section of the building for GPU compute. When the time comes, compose a virtual machine which has a slice of the storage and some of the compute. Every action is instant. A VM can be spun up or terminated in the scope of single moments or actions; capacity is scaled to match the demand of end users.

Companies like Liqid have brought us immeasurable levels of technological advancement. Their composable infrastructure offers unparalleled flexibility. Through bridging via systems' native PCI-Express interfaces, entire buildings of systems can communicate as if they were all within one server chassis. This approach strays from a more conventional system of hypervisor nodes, and is more rooted in hardware design and low-level resources sharing. Liqid has created a system of addressing hardware which transcends the boundaries of individual systems. It truly breaks a computing environment down into its raw resources. Then, provided only a simple BOM of required specs, a new computer can be conjured from the resource pool. This is the future of the server space—a future I hope home labs will soon share. In the meantime, I want for a legacy-compatible option. Something to run on my ever-growing collection of X86 dinosaurs.

Aging penguins rejoice. A solution may lie in an abstraction of the way we approach compute problems. Assuming an adequate instruction set is available, we can still break down and compile our applications into a distributed system of thinking. This theoretical instruction set could breath new life into compute clusters of old, who now lack some features of bleeding-edge systems.

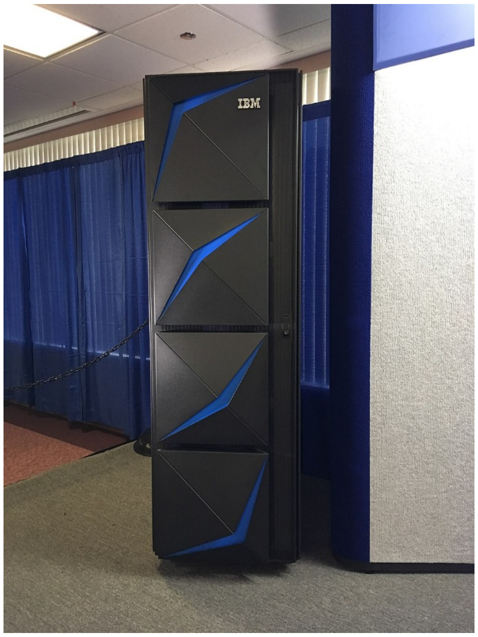
Table of Contents

Prior Developments

Mainframes

A centralized paradigm, mainframe systems from IBM and other business partners allowed one large shared computer to be used by several employees or students. Each user could create a new session with the mainframe, share its resource, then log out when completed. As the cost of individual computing systems was vast, a large central system was one of the most affordable options.

Sharing resources in such a way has obvious drawbacks. If the mainframe had downtime or needed repairs, entire businesses or communities could be without access to their work. During certain times of the day, employees may find it difficult to access the mainframe. They'll have to wait until after peak usage time to upload their deliverables.



(above: A modern IBM mainframe)

Personal Systems

To remediate some issues of having one large shared system, companies and schools opted for larger numbers of smaller and less powerful individual computers. These personal systems may still have shared resources (like company storage or printers)—or may be hooked into a managed company network—but each device feature its own capabilities and approaches problems from a much more individualistic view.

These devices could be any business laptops, school tablets, or other devices provided by an IT service to a student or employee.

In this case, as with all the others, issues will arise. Individual systems will age and need repair (then eventual replacement). Companies will often neglect their fleet, resulting in an entire network of outdated and insecure personal systems. Oversight of the fleet can also be time consuming.

While individual devices are convenient for employees and students, the upfront cost of registering a new device can be quite high. For an engineering firm, a new workstation and suite of engineering software can cost several thousands of dollars. Networked and mainframe solutions can offer better company-wide integration and stricter management of users. Personal systems can usually only be viable if connected to a shared network and closely maintained by IT personnel. With a centralized approach, users can be added and removed with improved ease and granularity.

Virtualization

VMs, a generalized term for several virtualization and containerization technologies in server and user space, are a method of centralizing and consolidating the assets of a company's compute infrastructure. One central system serves the fleet of users. Many of the restraints of mainframes had been overcome through brute compute force. Machines can be spun up and terminated with the flow of demand from employees. And, as technologies improve, virtual hosts can be upgraded and improved to give users an improved experience.

For the purposes of this consideration, I have grouped containerization in with virtualization. Many will refer to containers as "light-weight" VMs. Usually this will manifest as a Linux hypervisor host (such as a VMWare or ProxMox instance) with Linux containers. Certain kernel features and resources of the hypervisor are shared with the containers to ensure they can run as lightweight as possible.

Via VMs and containers, tens or hundreds of systems can be run within one single server chassis. Several users can log in and access a seemingly dedicated machine. Nobody has to know they are sharing a single computer via virtualization and RDP.

Compute Clusters

These clusters are aptly named. They are usually groups of peer machines or virtual machines. The group of peers can be referred to as a cluster. Sometimes each individual system or virtual system is called a node. For the purposes of this writing, I will refer to individual systems as nodes.

Compute clusters enable the technologies associated with high-availability (HA) systems. Since the compute solution is split across multiple nodes, there can exist failsafes and redundancies to allow near-perfect uptimes. When it comes time to service or upgrade a server node, changes can often be made with reduced downtime.

Individual nodes in a cluster may feature specialties. For example, some nodes may feature increased graphical compute capability; others may focus on providing storage or database resources. As demand for different resources changes, nodes with different specialties (or offering different services to users) can be added or removed.

Composability

(above: Liquid's logo)

As previously mentioned, composability shares common use-cases with virtualization. The difference here is that composable infrastructures are bare metal. There needn't be the overhead of a virtual machine. Tech media poster-child Liqid's approach is to do everything over PCI-Express. This effectively creates a computer spanning the whole floor of a datacenter.

Due to the sprawling nature of a composed system, there exist implicit power and hardware redundancies which can make this infrastructure more reliable and maintainable.

Is this technology a hyperconverged enterprise solution for the technically elite? Hobbyists need a comparable solution to serve the home.

Similarly to how Liqid has abstracted computer hardware to be composable, I think it could be possible to abstract clusters of Linux nodes to perform distributed tasks. In order to approach an implementation, we need to abstract the compute interface of a Linux system.

Which elements to take?

We can take a collection of some of the best qualities of all of the above technologies. These can be the central design elements of the Mycelium/Polymer application design.

- Isolate Mycelium Spore instances with containerization
- Dynamic load response with task kernels
- Work in compute clusters for reliability and redundancy
- Split capacity between nodes for availability
- Give users intuitive and lightweight controls
- Make application deployment easy via Polymer management
- Keep the average user safe from technical jargon

Resource Abstraction

Logic Procedures

Due to volatility of logic values and computational complexity of business or application logic, operations relating to one application process should be consolidated to the same node. This will allow the most seamless and least latent communication between that application's logical processes.

However, duplicate logical procedures can be created on separate nodes.

These procedures are the business logic workloads which drive the application as a whole. These processes will decide what compute tasks should be run, what data will be inputted, and where to send the processed data.

Compute Tasks

The computations performed on nodes in the cluster must have a defined input and output schema. Compute tasks are provided as a collection of instruction sets contingent upon a provided input data range. Each instance of the compute execution will provide its output to the invoker.

A creator of a compute task should be able to define how their task is executed. For instance, if they know the task contains a large number of floating point operations, the programmer may choose to target systems with graphical compute acceleration. If a task is dependent on high memory capacity, that may affect deployability to graphics or CPU compute nodes.

Each task's instructions are defined as a "kernel," similar to how CUDA and OpenCL approach instances of compute. Each unit of work is a kernel; it will be duplicated in parallel across the number of nodes necessary to meet the task's demand.

These types of tasks could include a distributed render of a video project. Utilize both the CPU and GPU capacity of Polymer's network to perform compute efforts.

Data Storage

As with many other cluster configurations, bulk data storage will likely be centralized to "storage nodes." Each node should always feature data redundancy, as well as an offsite backup. Optionally, each deployed node could feature a smaller and faster storage solution—this could be a cache or scratch drive in each server to store in-flight compute results.

In a database case, where multiple nodes must sync states and tables of data, a database would have a central host which is persistent (cold storage). In a large mesh configuration, there may need to be a delivery network—a number of write-through cache nodes which can accelerate delivery of database entries. Implementations would have to be careful with synchronization to avoid lost or corrupt data.

Communication

Instances of a compute kernel running within a common node will communicate via Mycelium's simulated network in local memory. In a case where compute is performed across several machines, the data is routed via the network and over IP. Low-latency interconnection of nodes is crucial for effective compute deployment. Both compute data and task executable data will be transferred between nodes. Input and output data will be routed to and from the compute nodes. The compute instructions themselves will be delivered to each compute node as well.

Synchronization

Semaphores and synchronization primitives should be held by a single node. That single node should provide API access to increment and decrement the semaphores. Local machines will operate like normal; over the network, a secondary message will have to be dispatched when access to the resource is available.

One approach could be to host all synchronization at the closest router node. Then it is locally available to pools of nodes which will cooperate to compute data.

Optimization

Out-of-order Execution

Temporal coupling occurs within a computation when tasks must be completed in a certain specific order. If executed temporally incorrect, the output may result in a crash or incorrect data. The executable format of the abstract instruction set should calculate a topological graph of which tasks are reliant on the result of previous tasks, grouping mutually exclusive operations into frames which can be executed in parallel.

This can allow extra system resources to be used to accelerate a singly threaded operation. The singly threaded logic task can be broken down into sequences of buckets of mutually exclusive elements. Executed in order, the end result of the accelerated computation should be equal to the un-accelerated output. I recognize a similarity to AVX extensions, which use parallelized vector silicon to unroll loops in the code and accelerate them.

Speculation

As the topology of the program's execution is evaluated, we can also use that data to speculate about possible branch results from the running task. Both possible paths of execution should be cached. If necessary, they can also be speculatively executed—their results may prove useful, but if not just become wasted CPU cycles and energy.

This could put undue pressure on compute nodes to evaluate branch possibilities. It might not be practical to evaluate these branches for large results (if both branches result in high compute usage).

Instruction Set

Required Functions

There is a set amount of required functionality for a useful instructions set to function.

- ### Memory allocation functions

  An application must be able to instruct allocation or freeing of memory.

- ### Mathematical operations

  Arithmetic such as addition, subtraction, division, multiplication, and remainder.

- ### Bit Operations

  Bit-wise logical operations used for common tasks like bit masking.

- ### Comparison/Branch

  Allow logical decisions to be made based on input data; speculative execution hinges on caching or evaluation of these branches before they are chosen.

- ### Stack Management

  Efficient and debuggable runtime stack to power effective problem solving.

Additional Requirements:

- Memory copies
- Memory reallocation
- System IO
- Call function
- Synchronization primatives
- Loop definition

Instruction Specification:

Instruction	Index	Description	Input Schema	Output Schema
malloc	0x00	Allocates a memory buffer to the provided handle	**size_t** size	**char** **buffer
free	0x01	Frees the buffer associated with the provided handle	**char** *buffer	-
realloc	0x02	Reallocates a buffer to increase or decrease its size	**size_t** size	**char** **buffer
memcpy	0x03	Copies a range of data from one buffer to another	**char** *from, **uint** offset, **size_**t size	**char** *to
add	0x04	Adds the two provided values, storing the sum in a third provided location	**int** a, **int** b	**int** *result
add_l	0x05	Adds the two provided values, storing the sum in a third provided location	**long** a, **long** b	**long** *result
add_f	0x06	Adds the two provided values, storing the sum in a third provided location	**float** a, **float** b	**float** *result
add_d	0x07	Adds the two provided values, storing the sum in a third provided location	**double** a, **double** b	**double** *result
add	0x08	Adds the two provided values, storing the sum in a third provided location	**int** a, **int** b	**int** *result
add_l	0x09	Adds the two provided values, storing the sum in a third provided location	**long** a, **long** b	**long** *result
add_f	0x0A	Adds the two provided values, storing the sum in a third provided location	**float** a, **float** b	**float** *result

Instruction	Index	Description	Input Schema	Output Schema
add_d	0x0B	Adds the two provided values, storing the sum in a third provided location	**double** a, **double** b	**double** *result
sub	0x0C	Subtracts the second argument from the first, storing the difference in a third provided location	**int** a, **int** b	**int** *result
sub_l	0x0D	Subtracts the second argument from the first, storing the difference in a third provided location	**long** a, **long** b	**long** *result
sub_f	0x0E	Subtracts the second argument from the first, storing the difference in a third provided location	**float** a, **float** b	**float** *result
sub_d	0x0F	Subtracts the second argument from the first, storing the difference in a third provided location	**double** a, **double** b	**double** *result
mod	0x10	Subtracts the second argument from the first, storing the difference in a third provided location	**int** a, **int** b	**int** *result
mod_l	0x11	Subtracts the second argument from the first, storing the difference in a third provided location	**long** a, **long** b	**long** *result
cast_f	0x12	Casts the provided float to an integer	**float** value	**int** *result
cast_f_l	0x13	Casts the provided float to an integer	**float** value	**long** *result
cast_d	0x14	Casts the provided float to an integer	**double** value	**int** *result
cast_d_l	0x15	Casts the provided float to an integer	**double** value	**long** *result