CHAPTER

Cloud hardware and software

4

This chapter presents the hardware and the software stack for cloud computing. In their quest to provide reliable, low-cost services, cloud service providers (CSPs) exploit the latest computing, communication, and software technologies to offer a highly available, easy-to-use, and efficient cloud computing infrastructure.

The cloud hardware infrastructure is built with inexpensive, off-the-shelf components to deliver cheap computing cycles. The millions of servers operating in today's cloud data centers deliver the computing power necessary to solve problems that in the past could only be solved by large supercomputers assembled from expensive, one-of-a-kind components.

The cloud system software is complex. There is not a single *killer application* for cloud computing thus, investing in large-scale computing systems can only be justified if the cloud infrastructure can effectively accommodate a mix of applications. In addition to scalability challenges, modern cluster management systems address the problems posed by a mix of workloads. Typical cloud workloads include not only coarse-grained, batch applications, but also fine-grained, long-running applications with strict timing constrains. Only strict performance isolation and sophisticated scheduling can eliminate the undesirable effects of long-tail distribution of latency-sensitive jobs response time.

Resource virtualization is widely used to hide the complexity of a physical system and facilitate system access. Virtual machines (VMs) and containers are key components of the cloud infrastructure as discussed in depth in Chapter 5. A VM abstracts the hardware and *exploits virtualization by multiplexing* allowing multiple virtual systems to share a physical system. A container exploits *virtualization by aggregation* and bridges the gap between a clustered infrastructure and assumptions made by applications about their environments. Containers abstract an OS and include applications or tasks, as well as all their dependencies. Containers are portable, independent objects that can be easily manipulated by the software layer managing a large virtual computer. This virtual computer exposes to users the vast resources of a physical cluster with a very large number of independent processors.

The management of large-scale systems poses significant challenges and has triggered a flurry of developments in hardware and software systems. Several milestones in the evolution of ideas in cluster architecture, along with algorithms and policies for resource sharing and effective implementation of the mechanisms to enforce these policies, are analyzed in this chapter. Our analysis of the cloud software stack is complemented by the discussion of software systems closely related to applications presented in Chapter 11 and by topics related to Big Data applications discussed in Chapter 12.

Section 4.1 examines challenges faced by cloud infrastructure and the benefits of virtualization and containerization. The next two Sections, 4.2 and 4.3, analyze Warehouse Scale Computers (WSCs) and their performance.

Then the focus switches to software and presents VMs and hypervisors in Section 4.4 and frameworks, such as Dryad, Mesos, and Borg, in Sections 4.5, 4.6, and 4.7, respectively. Dryad is an execution engine for coarse-grained data parallel applications, Mesos is used for fine-grained cluster resource sharing, and Borg is a cluster management system. Section 4.8 presents the evolution of cluster management systems and Section 4.9 covers Omega, a system based on state sharing. Section 4.10 analyzes Quasar, a system supporting QoS-aware cluster management. Resource isolation discussed in Section 4.11 is followed by an analysis of in-memory cluster computing with Spark and Tachyon in Section 4.12. Docker containers and Kubernetes are covered in Sections 4.13 and 4.14.

4.1 Cloud infrastructure challenges

Computing systems have evolved from single processors to multiprocessors, to multicore multiprocessors, and to clusters. The next step in this evolution, the Warehouse-scale computers (WSCs) with hundreds of thousands of processors are no longer a fiction, but serve millions of users, and are analyzed in computer architecture textbooks [52,232].

These systems are controlled by increasingly complex software stacks. Software helps integrate a very large number of system components and contributes to the challenge of ensuring efficient and reliable operation. The scale of the cloud infrastructure, combined with the relatively low mean-time to failure of the off-the-shelf components used to assemble a WSC, make the task of ensuring reliable services quite challenging.

At the same time, long-running cloud services require a very high degree of availability. For example, a 99% availability translates to 22 hours of downtime per quarter. Only a fair level of hardware redundancy combined with software support for error detection and recovery can ensure such a level of availability [232].

Virtualization. The goal of virtualization is to support portability, improve efficiency, increase reliability, and shield the user from infrastructure complexity. For example, threads are virtual processors, abstractions that allow a processor sharing among different activities thus, increase utilization and effectiveness. RAIDs are abstractions of storage devices designed to increase reliability and performance.

Processor virtualization, running multiple independent instances of one or more Operating Systems (OS), pioneered by IBM in early 1970s, was revived for computer clouds. Running multiple VMs on the same server enables applications to better share the server resources and lead to higher processor utilization. The instantaneous demands for resources by the applications running concurrently are likely to be different and complement each other, so the idle time of the server is reduced.

Processor virtualization is beneficial for both users and cloud service providers (CSPs). Cloud users appreciate virtualization because it allows better isolation of applications from one another than the traditional process-sharing model. Another advantage is that an application developer can chose to develop the application in a familiar environment and under the OS of choice. CSPs enjoy larger profits due to the lower cost for providing cloud services. Virtualization also provides more freedom for the system resource management because VMs can be easily migrated. VM migration proceeds as follows: the VM is stopped, its state is saved as a file, and the file is transported to another server where the VM is restarted. Nowadays, several hypervisors including KVM and Vmware ESXi fully support live migration, thus eliminating the need to stop the VM.

On the other hand, virtualization contributes to increased complexity of system software and has undesirable side effects on application performance and security. Processor sharing is now controlled by a new layer of software, the *hypervisor*, also called a Virtual Machine Monitor. It is often argued that a hypervisor is a more compact software with only a few hundred thousand lines of code versus the million lines of code of a typical OS, thus it is less likely to be faulty.

Unfortunately, though the footprint of the hypervisor is small, the server must run a management OS. For example, Xen, the hypervisor used by AWS until 2017, initially starts Dom0, a privileged domain that starts and manages *DomU* unprivileged domains. *Dom0* runs the Xen management *toolstack*, is able to access the hardware directly, and provides Xen with virtual disks and network access for guests.

Virtual Machines run applications inside a guest OS which runs on virtual hardware under the control of a hypervisor. One can build immutable VM images, but VMs are heavyweight and non-portable.

Containers. Containers are based on *operating-system-level virtualization rather than hardware virtualization*; they isolate an application running inside a container from another application running in a different container and isolate them both from the physical system where they run. Moreover, containers are portable, and the resources used by a container can be limited. They are more transparent than VMs, thus easier to monitor and manage. Containers have several other benefits including:

1. Ease creation and deployment of applications.
2. Applications are decoupled from the infrastructure; application container images are created at build time rather than deployment time.
3. Support portability; containers run independently of the environment.
4. Benefit from an application-centric management.
5. Have an optimal philosophy for application deployment; applications are broken into smaller, independent pieces and can be managed dynamically.
6. Support higher resource utilization.
7. Lead to predictable application performance.

Containers were initially designed to support the isolation of the *root* file system. The concept can be traced back to the *chroot* system call implemented in 1979 in Unix to change the root directory for the running process issuing the call and for its children, and to prohibit access to files outside the directory tree. Later, BSD and Linux adopted the concept, and in 2000, FreeBSD expanded it and introduced the *jail* command. The environment created with *chroot* was used to create and host a new virtualized copy of the software system.

Container technology has emerged as an ideal solution for application developers who no longer need to be aware of cluster organization and management details. Container technology is now ubiquitous and has a profound impact on cloud computing. Docker's containers gained widespread acceptance for ease of use, while Google's Kubernetes are performance-oriented.

Cluster management systems have evolved, and each system has benefited from the experience gathered from the previous generation. Mesos, a system developed at UC Berkeley, is now widely used by more than 50 organizations and has also morphed into a variety of systems, such as Aurora used by Twitter, Marathon offered by Mesosphere,[1] and Jarvis used by Apple. Borg, Omega, and Kubernetes are the milestones in Google's cluster management development effort discussed in this chapter.

[1] Mesosphere is a startup selling the Datacenter Operating System, a distributed OS, based on Apache Mesos.

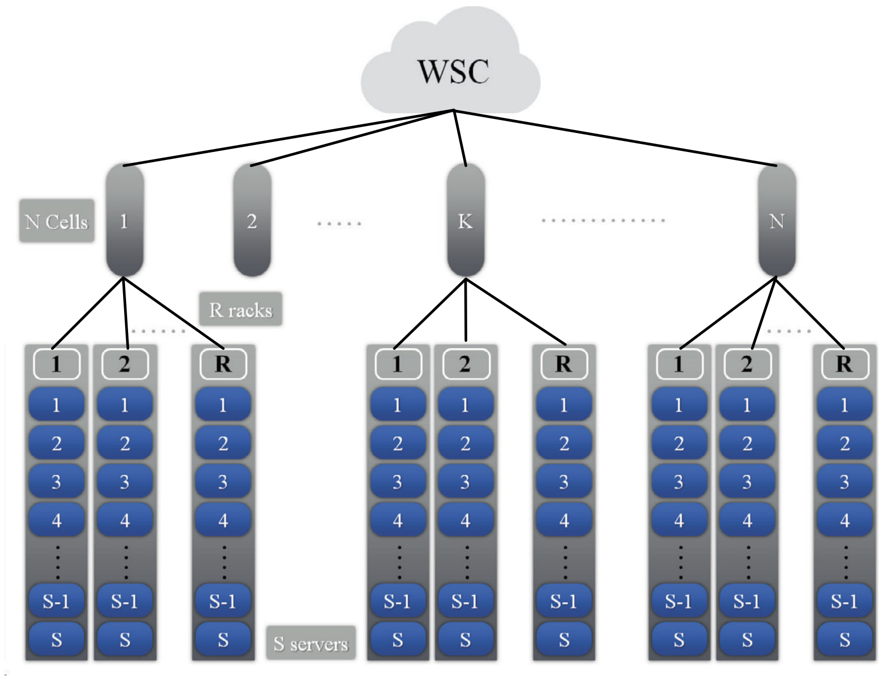

FIGURE 4.1

Organization of a WSC with N cells, R racks, and S servers per rack.

4.2 Cloud hardware; warehouse-scale computer (WSC)

Cloud computing had an impact on large-scale systems architecture. WSCs [52,232] form the backbone of the cloud infrastructure of Google, Amazon, and other CSPs. WSCs are hierarchically organized systems with 50 000–100 000 processors capable of exploiting request-level and data-level parallelism.

At the heart of a WSC is a *hierarchy of networks* which connect the system components, servers, racks, and cells/arrays together, as in Fig. 4.1. Typically, a *rack* consists of 48 servers interconnected by a 48 port, 10 Gbps Ethernet (GE) switch. In addition to the 48 ports, the GE switch has two to eight uplink ports connecting a rack to a cell. Thus, the level of *oversubscription*, the ratio of internal to external ports, is between $48/8 = 6$ and $48/2 = 24$. This has serious implications for the performance of an application; two processes running on servers in the same rack have a much larger bandwidth and lower latency than the same processes running on servers in different racks.

The next component is a *cell*, sometimes called an array, consisting of a number of racks. The racks in a cell are connected by an *array switch*, a rather expensive communication hardware with a cost two orders of magnitude higher than that of a rack switch. The cost is justified as the bandwidth of a switch with n ports is of order n^2. To support a 10 times larger bandwidth for 10 times as many ports, the cost increases by a factor of 10^2. An array switch can support up to 30 racks.

Table 4.1 The memory hierarchy of a WSC with the latency given in microseconds, the bandwidth in MB/sec, and the capacity in GB [52].

Location type	**DRAM**			**Disk**		
	Latency	***Bandwidth***	***Capacity***	***Latency***	***Bandwidth***	***Capacity***
Local	0.1	20 000	16	10 000	200	2 000
Rack	100	100	1 040	11 000	100	160 000
Cell	3 000	10	31 200	12 000	10	4 800 000

WSCs support both interactive and batch workloads. The communication latency and the bandwidth within a server, a rack, and a cell are different, thus, the execution time and the costs for running an application is affected by the volume of data, the placement of data, and by the proximity of instances. For example, the latency, the bandwidth, and the capacity of the memory hierarchy of a WSC with 80 servers/rack and 30 racks/cell is shown in Table 4.1 based on the data from [52].

DRAM latency increases by more than three orders of magnitude, while the bandwidth decreases by a similar factor from server, to rack, and to cell. The latency and the bandwidth of the disks follow the same trend, but the variation is less dramatic. To put this into perspective, the memory-to-memory transfer of 1 000 MB takes 50 msec within a server, 10 seconds within the rack, and 100 seconds within a cell, while disk transfers take 5, 10, and 100 seconds, respectively.

WSCs are expected to supply cheap computing cycles, but are expensive; the cost of a WSC is of the order of \$150 million, but the cost-performance is what makes them appealing. The capital expenditures for a WSC includes the costs for servers, for the interconnect and for the facility. A case study reported in [232] shows a capital expenditure of \$167 510 000 including \$6 670 000 for 45 978 servers, \$12 810 000 for an interconnect with 1 150 rack switches, 22 cell switches, 2 layer 3 switches, and 2 border routers. In addition to the initial investment, the operation cost of the cloud infrastructure including the cost of energy is significant. In this case study the facility is expected to use 8 MW.

We now take a closer look at the WSC servers and ask what type of processors are best suited as server components. Multicore processors are ideal components of WSC servers because they support not only data-level parallelism for search and analysis of very large data sets but also request-level parallelism for systems expected to support a very large number of transactions per second. *Data-parallel* and *request-parallel* applications are the two major components of the workloads experienced by cloud service providers such as Google.

There are two basic types of multicore processors often called *browny* and *wimpy* cores [242]. The single-core performance of a browny core is impressive, but so is its power dissipation. The wimpy cores are less powerful but consume less power. Power consumption is a major concern for cloud as we see in Section 1.2; for solid-state technologies the power dissipation is approximately $\mathcal{O}(f^2)$ with f being the clock frequency.

An application task needs to spawn a larger number of threads when running on wimpy rather than browny cores, and this has two major implications: First, it complicates the software development process because it requires an explicit parallelization of the application, thus increases the application development cost. The second, equally important implication is that running a larger number of threads increases the response time. Multi-phase algorithms use *barrier-synchronization* requiring all threads have to finish before the next phase of the computation. This means that all threads have to wait for the slowest one.

The cost of systems using wimpy core may increase, e.g., the cost for DRAM will increase as the kernel and system processes consume more aggregate memory. Data structures used by applications might need to be loaded into memory on multiple wimpy-core machines instead of a single brawny-core machine with negative effects on performance. Lastly, managing a larger number of threads will increase the scheduling overhead and diminish performance.

Hölzle [242] concludes "Once a chip's single-core performance lags by more than a factor of two or so behind the higher end of current-generation commodity processors, making a business case for switching to the wimpy system becomes increasingly difficult because application programmers will see it as a significant performance regression: their single-threaded request handlers are no longer fast enough to meet latency targets."

4.3 WSC performance

The central questions discussed now are: how to extract the maximum performance from a warehouse-scale computer; what are the main sources of WSC inefficiency; and how these inefficiencies could be avoided. Even slight WSC performance improvements translate into large cost savings for the CSPs and noticeably better service for cloud users.

WSCs workload is very diverse; there are no typical, or "killer," applications that would drive the design decisions and, at the same time, guarantee optimal performance for such workloads. It is not feasible to experiment with systems at this scale or to simulate them effectively under realistic workloads. The only alternative is to profile realistic workloads and analyze carefully the data collected during production runs, but this is only possible if low-overhead monitoring tools that minimize intrusion on the workloads are available. Monitoring tools minimize intrusion by random sampling and by maintaining counters of relevant events, rather than detailed event records.

Google-Wide-Profiling (GWP) is a low-overhead monitoring tool used to gather the data through random sampling. GWP randomly selects a set of servers to profile every day, uses mostly Perf[2] to monitor their activity for relatively short periods of time, collects the *callstacks*[3] of the samples, aggregates the data, and then stores it in a database [413].

Data collected at Google with GWP over a period of 36 months is presented in [265] and discussed in this section. Only data for C++ codes were analyzed because C++ codes dominate the CPU cycle consumption, though the majority of codes are written in Java, Python, and Go. The data was collected from some 20 000 servers built with Intel Ivy Bridge processors.[4]

The analysis is restricted to 12 application binaries with distinct execution profiles: batch versus latency sensitive and low-level versus high-level services. The applications are: Gmail and Gmail-fe, the back- and front-end Gmail application; BigTable, a storage system discussed Section 7.11; *disk*, low-level distributed storage driver; *indexing1* and *indexing2* of the indexing pipeline; *search1, 2, 3*, application for searching leaf nodes; *ads*, an application targeting ads based on web-page contents;

[2] Perf is a profiler tool for Linux 2.6+ systems; it abstracts CPU hardware differences in Linux performance measurements.

[3] A *callstack*, also called execution stack, program stack, control stack, or run-time stack, is a data structure that stores information about the active subprograms invoked during the execution of a program.

[4] The Ivy Bridge-E family is made in three different versions with the postfix -E, -EN, and - EP, with up to six, ten, and twelve cores per chip, respectively.

video, a transcoding and feature extraction application; and *flights-search*, the application used to search and price flights.

In spite of the workload diversity, there are common procedures used by a vast majority of applications running on WSCs. Data-intensive applications run multiple tasks distributed across several servers, and these tasks communicate frequently with one another. Cluster management software is also distributed and daemons running on every node of the cluster communicate with one or more schedulers making system-wide decisions. It is not unexpected that common procedures accounting for a significant percentage of CPU cycles are dedicated to communication. This is the case of RPCs (Remote Procedure Calls), as well as serialization, deserialization, and compression of buffers used by communication protocols.

A typical communication pattern involves the following steps: (a) serialize the data to the protocol buffer; (b) execute an RPC and pass the buffer to the callee; and (c) caller deserializes the buffers received in response to the RPC. Data collected over a period of 11 months shows that these common procedures translate into an unavoidable "WSC architecture tax" and consume 22–27% of the CPU cycles. About one-third of RPCs are used by cluster management software to balance the load, encrypt the data, and detect failures. The remaining RPCs move data between application tasks and system procedures. Data movement is also done using library functions, such as *memmove* and *memcpy,* with descriptive names[5] which account for 4–5% of the "tax."

Compression, decompression, hashing, and memory allocation and reallocation procedures are also common and account for more than one fourth of this "tax." Applications spend about one fifth of their CPU cycles in the scheduler and the other components of the kernel. An optimization effort focused on these common procedures will undoubtedly lead to a better WSC utilization.

Most cloud applications have a substantial memory footprint, binaries of hundreds of MB are not uncommon, and some do not exhibit either spatial or temporal locality. Moreover, the memory footprint of applications shows a significant rate of increase, about 30% per year. Also, the instruction-cache footprints grow at a rate of some 2% per year. As more Big Data applications run on computer clouds, neither the footprint nor the locality of these applications are likely to limit the pressure on cache and memory management. These functions represent a second important target for the performance optimization effort.

Memory latency rather than memory bandwidth affects a processor ability to deliver a higher level of performance through Instruction Level Parallelism (ILP). The performance of such processors is significantly affected by stall cycles due to cache misses. It is reported that data cache misses are responsible for 50–60% of the stall cycles and, together with instruction cache misses, contribute to a lower IPC (Instructions Per Clock cycles).

There are patterns of software that hinder execution optimization through pipelining, hardware threading, out-of-order execution, and other architectural features designed to increase the level of ILP. For example, linked data structures cause indirect addressing that can defeat hardware prefetching and build bursts of pipeline idleness when no other instructions are ready to execute.

Understanding cache misses and stalls requires a microarchitecture-level analysis. A generic organization of the microarchitecture of a modern core is illustrated in Fig. 4.2. The core front-end processes

[5] In Linux, *memmove* and *memcpy* copy n bytes from one memory area to another; the areas may overlap for the former but do not overlap for the later.

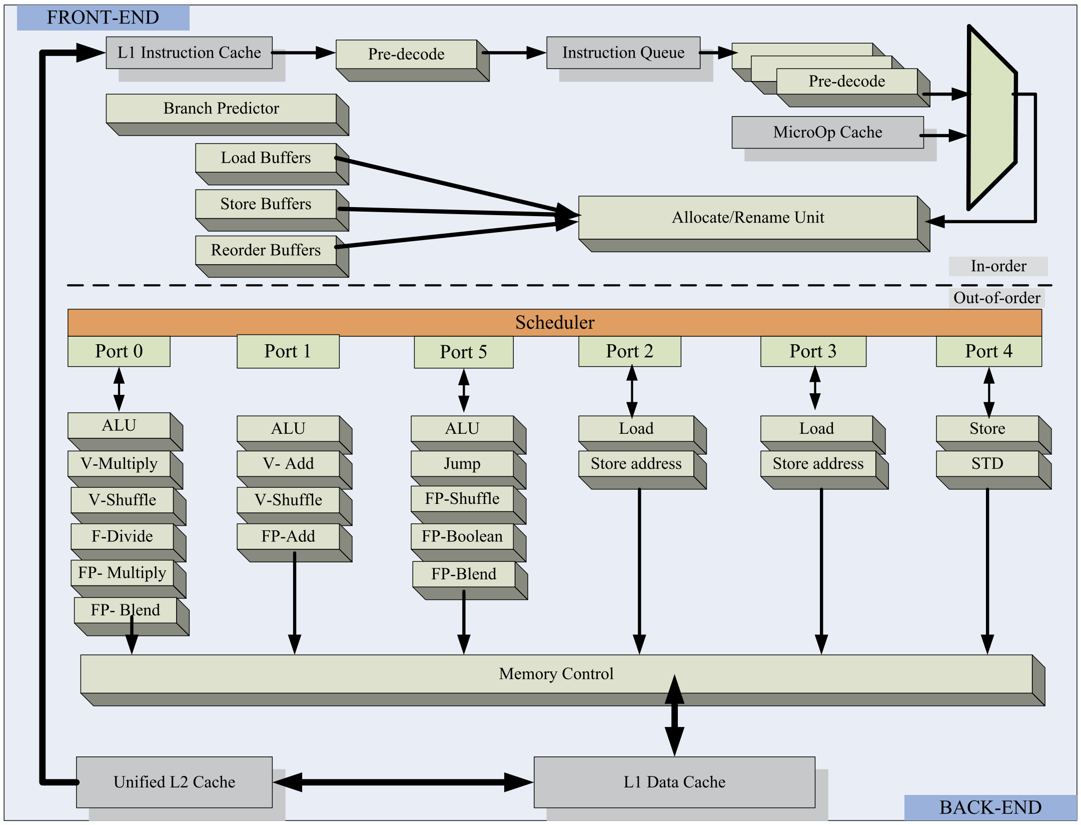

FIGURE 4.2

Schematic illustration of a modern processors core microarchitecture. Shown are the front-end, the back-end, L1 instruction and data caches, the unified L2 cache, and the microoperation cache. Branch prediction unit, load/store, and reorder buffers are components of the front-end. The instruction scheduler manages the dynamic instruction execution. The five ports of the instruction scheduler dispatch microinstructions to ALU and to load and store units. Vector (V) and floating-point operations (FP) are dispatched to ALU units.

instructions in order, while the instruction scheduler of the back-end is responsible for dynamic instruction scheduling and feeds instructions to multiple execution units including Arithmetic and Logic Units (ALU)s and Load/Store units.

The microarchitecture-level analysis is based on a top-down methodology [527]. According to https://software.intel.com/en-us/top-down-microarchitecture-analysis-method: “The Top-Down characterization is a hierarchical organization of event-based metrics that identifies the dominant performance bottlenecks in an application. Its aim is to show, on average, how well the CPU’s pipeline(s) were being utilized while running an application.”

This methodology identifies the *micro-op* (μop) queue as the separator between the front-end and the back-end components of a microprocessor core. The μop pipeline slots are then classified as *Retiring, Front-end bound, Bad speculation,* or *Back-end bound*, with only the first one doing useful work. *Front-end bound* includes overheads associated with fetching, instruction caches, decoding, and some other shorter-penalties and *Back-end bound* includes overheads due to the data-cache hierarchy and the lack of ILP; *Bad speculation* is self-explanatory.

In a typical SPEC CPU2006 benchmark, the front-end wasted execution slots are typically two–three times lower than those reported for the Google workload which account for 15–30% of all wasted slots. A reason for this behavior is that SPEC applications[6] do not exhibit the combination of low retirement rates[7] and high front-end boundedness of WSCs.

Data shows that the core back-end dominates the overhead and limit the ILP. The back-end and the front-end stalls limit the number of cores active during an execution cycle. To be more precise, only one or two cores of a six-core Ivy Bridge processor are active in 72% of execution cycles, while three cores are active during the balance of 28% of cycles.

The observation that memory latency is more important than memory bandwidth is a consequence of the low memory-bandwidth utilization at an average of 31% and a maximum of 68% with a heavy tail distribution. In turn, the low memory utilization is due in part to low CPU utilization. A surprising result reported in [265] is that the median CPU bandwidth utilization is 10%, while [52] reports a median CPU utilization in a much higher range, 40–70%. A low CPU utilization is also reported for the CloudSuite [170].

Several conclusions regarding optimal processor architecture can be reached from the top-down data analysis. Data analysis shows that cloud workloads display access patterns involving bursts of computations intermixed with bursts of stall cycles. Processors supporting a higher level of *simultaneous multithreading* (SMT) are better equipped to hide the latency by overlapping stall cycles than two-wide SMP processors. SMT is an architectural feature allowing instructions from more than one thread to be executed in any given pipeline stage at a time. SMT requires the ability to fetch instructions from multiple threads in a cycle; it also requires a larger register file to hold data from multiple threads.

The large working sets of the codes are responsible for the high rate of instruction cache misses. L2 caches show that MPKI (misses per kilo instructions) are particularly high. Larger caches would alleviate this problem but at the cost of higher cache latency. Separate cache policies that give priority to instructions over data or separate L2 caches for instructions and data could help in this regard.

[6] The following applications in the SPEC CPU2006 suite are used: *400.perlbench* application which has high IPC and the largest instruction cache working set; *445.gobmk*, an application with hard-to-predict branches; *429.mcf* and *471.omnetpp* memory-bound applications which stress memory latency; and *433.milc*, a memory-bound application which stresses memory bandwidth.

[7] In a modern processor, the Completed Instruction Buffer holds instructions that have been speculatively executed. Associated with each executed instruction in the buffer are its results in rename registers and any exception flags. The retire unit removes these executed instructions from the buffer in program order at a rate of up to four instructions per cycle. The retire unit updates the architected registers with the computed results from the rename registers. The retirement rate measures the rate of these updates.

4.4 Hypervisors

A hypervisor securely partitions resources of a computer system into one or more VMs. A *guest OS* is an operating system that runs under the control of a hypervisor rather than directly on the hardware. A hypervisor runs in kernel mode, while a guest OS runs in user mode; sometimes, the hardware supports a third mode of execution for the guest OS. Hypervisors allow several operating systems to run concurrently on a single hardware platform and enforce isolation among these systems, thus better security. A hypervisor controls how the guest OS uses the hardware resources; the events occurring in one VM do not affect any other VM running under the same hypervisor. At the same time, the hypervisor enables:

- Multiple services to share the same platform.
- The movement of a service from one platform to another called *live migration*.
- System modification while maintaining backward compatibility with the original system.

When a guest OS attempts to execute a privileged instruction, the hypervisor traps the operation and enforces the correctness and safety of the operation. The hypervisor guarantees the isolation of the individual VMs and, thus, ensures security and encapsulation, a major concern in cloud computing. At the same time, the hypervisor monitors the system performance and takes corrective actions to avoid performance degradation. For example, the hypervisor may swap out a Virtual Machine to avoid thrashing; to do so, the hypervisor copies all pages of the evicted VM from real memory to disk and makes the real memory frames available for paging by other VMs.

A hypervisor virtualizes the CPU and the memory. For example, the hypervisor traps interrupts and dispatches them to the individual guest operating systems; if a guest OS disables interrupts, the hypervisor buffers such interrupts until the guest OS enables them. The hypervisor maintains a *shadow page table* for each guest OS and replicates any modification made by the guest OS in its own shadow page table. This shadow page table points to the actual page frame, and it is used by the hardware component called the Memory Management Unit (MMU) for dynamic address translation.

Memory virtualization has important implications on the performance. Hypervisors use a range of optimization techniques; for example, the VMware systems avoid page duplication among different VMs, they maintain only one copy of a shared page, and use copy-on-write policies, while Xen imposes total isolation of the VM and does not allow page sharing. Hypervisors control the virtual memory management and decide what pages to swap out; for example, when the ESX VMware Server wants to swap out pages, it uses a *balloon process* inside a guest OS and requests it to allocate more pages to itself and, thus, swaps-out pages of some of the processes running under that VM. Then it forces the balloon process to relinquish control of the free page frames.

4.5 Execution of coarse-grained data-parallel applications

The distinction between fine-grained and coarse-grained paralellism introduced in Chapter 3 is important for understanding cloud software organization. Application developers have used the SPMD (Same-Program-Multiple-Data) paradigm for several decades for exploiting coarse-grained parallelism. The name SPMD illustrates perfectly the idea behind the concept—a large dataset is split into several segments processed independently, and often concurrently, using the same program.

For example, converting a large number of images, e.g., 10^9, from one format to another, can be done by splitting the set into 1 000 thousand segments with 10^6 images each, and then running concurrently the conversion program on 1 000 processors. In this example, 1 000 processors cut the computing time by almost three orders of magnitude.

It is easy to see that such applications are ideal for cloud computing since they need a large computing infrastructure and can keep the systems busy for a fair amount of time. To CSP's delight, such jobs increase system utilization because no scheduling decisions have to be made until the time-consuming job has finished. This was noticed early on and, in 2004, the MapReduce idea was born [129].

MapReduce and Apache Hadoop, an open-source software framework consisting of a storage part, the Hadoop Distributed File System (HDFS), and the processing part called MapReduce discussed in Chapter 11 expand on the SPMD concept. MapReduce is a two-phase process. Input data is first segmented, then the computations on data segments are carried out during the Map phase; partial results are then merged during the Reduce phase.

This extends the scope of SPMD for computations that are not totally independent, the so-called *embarrassingly parallel* applications. In the previous example, instead of converting the 10^9 images, we now search during the Map phase for an object that could appear in any of them; after the search in each segment is completed, during the Reduce phase, we combine the partial results and further refine the information about the object from the set of images selected during the Map phase.

Dryad is a general-purpose distributed execution engine developed in 2007 by Microsoft for coarse-grained data-parallel applications. Microsoft wanted to use Dryad for running Big Data applications on its clustered server environment as a proprietary alternative to Hadoop, a widely used platform for coarse-grained data-parallel applications. A Dryad application combines computational *vertices* with communication links to form a dataflow graph [255]. Then, it runs the application by executing the vertices of this graph on a set of available computers, communicating through files, TCP pipes, and shared-memory.

The system is centrally controlled by a *Job Manager* (JM) running either on one of the nodes of the cluster or outside the cluster, on the user's computer. The JM uses a *Name Server* (NS) to locate the nodes of the cluster where the work is actually done. The JM uses an application-specific description to construct the dataflow graph of the application. A *daemon* running on each cluster node communicates with the JM and controls the execution of the code for the vertex of the graph assigned to that node. Daemons communicate directly among themselves without the intervention of the JM and use the information provided by the dataflow graph to carry out the computations.

A detailed description of the Dryad dataflow graph given in [255] presents a set of simpler graphs used to construct more complex one. The graph nodes are annotated to show the input and output datasets. Two connection operations are the point-wise composition and the complete bipartite composition.

The DryadLINQ (DLNQ) system is the product of a related Microsoft project [530]. It exploits the Language INtegrated Query (LINQ), a set of .NET constructs for performing arbitrary side effect-free transformations on datasets to automatically translate data-parallel codes into an workflow for distributed execution. The distributed execution is then executed on the Dryad platform. The development of DryadLINQ was motivated by the fact that parallel databases implement only declarative sides of SQL queries and do not support imperative programming. Dryad is not scalable and to no one's surprise, soon after announcing plans to release Windows Azure- and Windows Server-based implementations of open source Apache Hadoop, Microsoft discontinued the project.

4.6 Fine-grained cluster resource sharing in Mesos

Mesos is a light-weight framework for fine-grained cluster resource sharing developed in late 2010s at UC Berkeley. Mesos consists of only some 10 000 lines of C++ code [240]. Mesos runs on Linux, Solaris and OS X and supports frameworks written in C++, Java, and Python. Mesos can use Linux containers to isolate task CPU cores and memory.

A novelty of the system is a two-level scheduling strategy for large clusters with workloads consisting of a mix of frameworks. The term "framework" in this context means a large consumer of CPU cycles and widely-used software systems such as Hadoop, discussed in Chapter 11, and MPI (Message Passing Interface), a standardized and portable message-passing system used by the parallel-computing community since the 1990s. Another novelty is the concept of *resource offer*, an abstraction for a bundle of resources that a framework can allocate on a cluster node to run its tasks.

The motivation for Mesos is that *centralized scheduling is not scalable due to its complexity* and cannot perform well for fine-grained resource sharing. Framework jobs consisting of short tasks are mapped to resource *slots,* and fine-grained matching has a high overhead and prevents sharing across frameworks. Mesos allows multiple frameworks to share resources in a fine-grained manner and achieve data locality. It can isolate a production framework from experiments with a new version undergoing testing and experimentation. Each framework has a *scheduler* that receives resource offers from the master and an *executor* on each machine to launch the tasks of the framework. Scheduler functions are:

- *callbacks*,[8] such as *resourceOffer, offerRescinded, statusUdate,* and *slaveLost*, and
- *actions*, such as *replyToOffer, setNeedsOffers, setFilters, getGuaranteedShare,* and *killTask*.

The executer functions are *callbacks*, such as *launchTask* and *killTask*, and *actions*, such as *sendStatus*.

Framework requests differentiate mandatory from preferred resources. A resource is *mandatory* if the framework cannot run without it, e.g., a GPU is a mandatory resource for applications using CUDA.[9] A resource is *preferred* if a framework performs better using a certain resource but could also run using another one.

This two-level scheduling strategy keeps Mesos simple and scalable and, at the same time, gives the frameworks the power to optimally manage a cluster. The system is flexible and supports pluggable *isolation modules* to limit CPU, memory, and network bandwidth, and the I/O usage of a process tree. Allocation modules can select framework-specific policies for resource management. For example, the killing of a task of a greedy or buggy framework is aware whether the tasks of a framework are interdependent, as in case of MPI, or independent, as in case of MapReduce.

Mesos is organized as follows: A master process manages daemons running on all cluster nodes, while frameworks run the tasks on cluster nodes. The master implements the fair-sharing of resource offers among frameworks and lets each framework manage resource sharing among its tasks. The system is robust, i.e., there are replicated masters in hot-standby state. When the active master fails, a ZooKeeper service[10] is used to elect a new master. Then, the daemons and the schedulers reconnect to the newly elected master.

[8] A callback is executable code passed as an argument to other code; the callee is expected to execute the argument either immediately or at a later time for synchronous and, respectively, asynchronous callbacks.
[9] CUDA is a parallel-computing platform supporting graphics processing units (GPUs) for general-purpose processing.
[10] ZooKeeper is a distributed coordination service implementing a version of the Paxos consensus algorithm, see Chapter 11.

The limitations of the distributed scheduling implemented by Mesos are also discussed in [240]. Sometimes, the collection of frameworks is not able to optimize bin packing and a centralized scheduler. Another type of fragmentation occurs when the tasks of a framework request relatively small quantities of resources and, upon completion, resources released by the tasks are insufficient to meet the demands for large quantities of resources by tasks from a different framework. Resource offers could increase the complexity of framework scheduling, but centralized scheduling is not immune to this problem, either.

Porting Hadoop to run on Mesos required relatively few modifications. Indeed, the *JobTracker* and the *TaskTrackers* components of Hadoop map naturally as Mesos framework scheduler and executor, respectively. Apache Mesos is an open-source system adopted by some 50 organizations including Twitter, Airbnb, and Apple (see http://mesos.apache.org/documentation/latest/powered-by-mesos/).

Several frameworks based on Mesos have been developed over the years. Apache Aurora was developed in 2010 by Twitter and now is open-source. Chronos is a cron-like[11] system, elastic and able to express dependencies between jobs. Apple uses a Mesos framework called Jarvis[12] to support Siri. Jarvis is an internal PaaS cloud service to answer IOS user's voice queries. A fair scheduler at Facebook allocates the resources of a 2 000 node cluster dedicated to Hadoop jobs. MPI and MapReduceOnline (a streaming of MapReduce discussed in Chapter 11) jobs for ad targeting need the data stored on the Hadoop cluster, but the frameworks cannot be mixed.

The utilization analysis of a production cluster with several thousand servers used to run production jobs at Twitter shows that the average CPU utilization is below 20% and the average memory utilization is below 50% [240]. Reservations improve the CPU utilization up to 80%. Some 70% of the reservations overestimate the resources they need by one order of magnitude, while 20% of the reservations underestimate resource needs by a factor of five.

In summary, *one can view Mesos as the opposite of virtualization. A VM is based on an abstraction layer encapsulating an OS together with an application inside a physical machine. Mesos abstracts physical machines as pools of indistinguishable servers and allows a controlled and redundant distribution of tasks to these servers.*

4.7 Cluster management with Borg

A computer cluster may consist of tens of thousands of processors. For example, a cell of a WSC is in fact a cluster consisting of multiple racks, each with tens of processors, as shown in Fig. 4.1. There are two sides of cluster management; One reflects the views of application developers who need simple means to locate resources for an application and then to control the use of resources; the other is the view of service providers concerned with system availability, reliability, and resource utilization.

These views drove the design of Borg, a cluster management software developed at Google [495]. Borg's design goals were:

- Manage effectively workloads distributed to a large number of systems; be highly reliable and available.

[11] *Cron* is a job scheduler for Unix-like systems used to periodically schedule jobs; often, it is used to automate system maintenance and administration.
[12] Jarvis is short for *Just A Rather Very Intelligent Scheduler.*

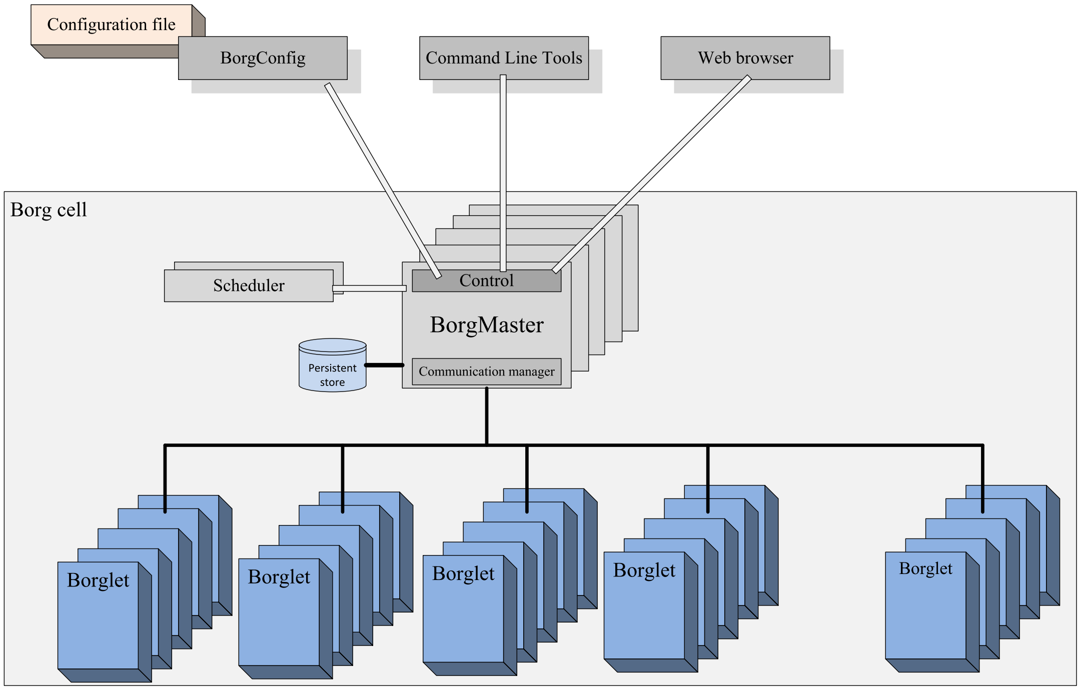

FIGURE 4.3

Borg architecture. A replicated *BorgMaster* interacts with *Borglets* running on each machine in the cell.

- Hide the details of resource management and failure handling and allow users to focus on application development. This requirement is important because the servers of a cluster differ in terms of processor type and performance, number of cores per processor, RAM size, secondary storage, network interface, and other capabilities.
- Support long-running, highly-dependable applications, including interactive production jobs and batch non-production jobs.

A Borg cluster consists of tens of thousands of machines co-located and interconnected by a data center-scale network fabric. A cluster managed by Borg is called a *cell*. The architecture of the system shown in Fig. 4.3 consists of a logically centralized controller, the *BorgMaster*, and a set of processes running on each machine in the cell, the *Borglets*. All Borg components are written in C++.

The main *BorgMaster* has five replicas, and each replica maintains an in-memory copy of the state of the cell. The state of a cell is also recorded in a Paxos-based store on local disks of each replica. An elected master serves as the Paxos leader and handles operations that change the state of a cell, e.g., submit a job or terminate a task.

The master process performs several actions: (i) handles client RPCs that change state or looks-up data; (ii) manages state machines for machines, tasks, allocs, and other system objects; (iii) communicates with *Borglets*; and (iv) responds to requests submitted to a web-based user interface. *Borglets*

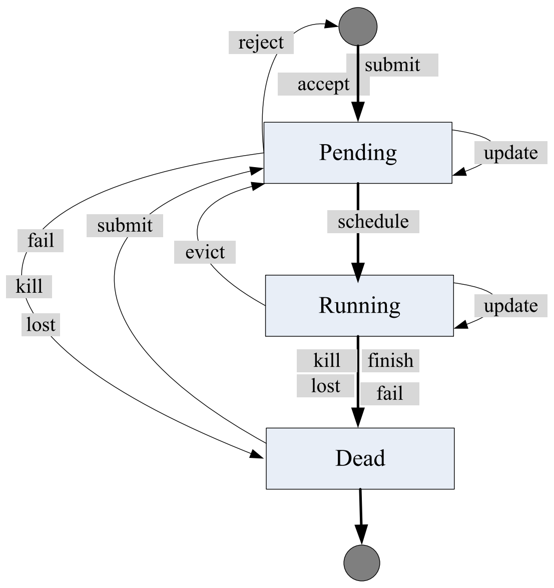

FIGURE 4.4

The states of a Borg task. Task state changes as a result of either user requests or system actions.

start, stop, and restart failing tasks, manipulate the OS kernel setting to manage local resources, and report the local state to the *BorgMaster*.

Users interact with running processes by means of RPCs to the *BorgMaster* and trigger task state transitions. Users request actions such as *submit, kill*, and *update*. The task state is also changed by system actions such as: *reject, evict,* and *lost*, see Fig. 4.4.

The *scheduler* is the other main component of the *BorgMaster*. The scheduler scans periodically in a round-robin order a priority queue of pending tasks. The *feasibility* component of the scheduling algorithm attempts to locate systems where to run tasks. The *scoring* component identifies the machine(s) to actually run the task.

Alloc and *alloc sets* reserve resources on a machine and, respectively, on multiple machines. Jobs have priorities: Distinct *priority bands* are defined for activities such as monitoring, production, batch, and testing. A *quota system* for job scheduling uses a vector including the quantity of resources such as CPU, RAM, and disk for specified periods of time. Higher-priority quota cost more than lower-priority ones. To simplify resource management and balance the load, a system similar with the one described in [22] is used to generate a single cost value per vector and minimize the cost, while balancing the workload and leaving room for spikes in demand.

Production jobs are allocated about 70% of CPU resources and 55% of the total memory [495]. A Borg job could have multiple tasks and runs in a single cell. The majority of jobs do not run inside a VM. Tasks map to Linux processes running in containers.

To manage large cells, the scheduler spawns several concurrent processes to interact with the *BorgMaster* and *Borglets*. These processes operate on cached copies of the cell state. Several other design decisions are important for system scalability. For example, to avoid frequent time-consuming machine and task scoring, Borg caches the score until the properties of the machine or task change significantly.

To avoid determining the feasibility of each pending task on every machine, Borg computes feasibility and scoring per equivalence classes of tasks with similar requirements. Moreover, this evaluation is not done for every machine in the cell but on random machines until enough suitable machines have been found.

The state of the *BorgMaster* can be saved as a *checkpoint* file and later used for studies of system performance and effectiveness, or to restore the state to an earlier point in time. A simulator, the *Faux-Master*, designed to identify system errors and performance problems by replaying checkpoint files, facilitated the effort to improve the Borg system.

Results collected for a 12 000-server cluster at Google show an aggregate CPU utilization of 25–35% and an aggregate memory utilization of 40%. A reservation system raises these figures to 75% and 60%, respectively [412].

4.8 Evolution of a cluster management system

It is interesting to examine the evolution of cluster management systems over the years and see if cloud service providers, such as Amazon, Google, and Microsoft, have optimized their scheduling strategies to increase the resource utilization and efficiency of large data centers. Such studies are possible only if trace data covering extended periods of time are made available to researchers from industry and academia. Fortunately, in 2011, Google published a one-month trace from a cluster managed with Borg [411]. Eight years later, in May 2019, Google published again detailed job scheduling information from eight Borg cells [510], enabling a group of researchers to draw some conclusions regarding changes in workloads, scheduling strategies, and resource utilization.

The analysis in [476] starts with a comparison of hardware used by the two traces, 2019 versus 2011: eight versus one cell, 96 400 servers versus 12 600, 7 versus 3 hardware platforms, 21 versus 10 server configurations, and about the same number of servers per cell, 12 000 versus 12 000. As we can see, trace data collected in 2019 involves a considerably larger number of more diverse servers. The software is also different; it provides schedulers with a wealth of information to optimize resource utilization and better support high priority jobs. The 2019 Borg supports alloc sets, i.e., resources reserved for a job and distributed across multiple machines, job dependencies, batch queuing, and vertical scaling; the system also allows a much larger range of priority values 0–450 instead of 0–11. Vertical scaling is now supported by the Autopilot discussed in [426].

2018 workloads differ from those for 2011. The mean job-arrival rate increased from 964 to 3 360 jobs per hour, and the job submission rate was 3.7 times higher. The median task-scheduling rate increased 3.6 times; many task scheduling events are for rescheduling indicating that there is more “churn” in the system, i.e., more task completion events.

The 2019 scheduler is more agile and has to work harder, the number of jobs to be scheduled is *seven* times higher, and most of the workload has moved into the high-priority best-effort batch tier. The heavy-tailed distribution of job sizes and the power law Pareto distribution with heavy tail of computing cycles and memory add to the scheduling challenges of 2019 Borg. Significantly improved scheduling decisions led to an increase of both average CPU and memory utilization: 20–40% for CPU and 3–30% for memory. The utilization variance is lower, and there were fewer servers with utilization larger than 80%. Median scheduling delays decreased, but the tail is longer for the last 28% of jobs.

The large variability of resource consumption is reflected by the spread of the coefficient of variation C^2 of two measures, Normalized Compute Unit-hours (NCU-hours) and Normalized Memory Unit-hours (NMU-hours) used per job. $C^2 = 23\,000$ for compute consumption because the variance is about 33 300 with a mean of 1.2 NCU-hours and $C^2 = 43\,000$ for memory consumption. Both values are extremely large, $C^2 = \text{variance}/\text{mean}^2$ is invariant to data normalization and $C^2 = 1$ for an exponential distribution, and a study of workloads in several supercomputing centers found that C^2 ranged from 28 to 256, depending on the supercomputing center. Trace data analysis shows that compute consumption and memory consumption follow almost the same distribution, and the two metrics are correlated [476]. One of the conclusions of this research is that resource-prediction algorithms appear more efficient than users manually defining their own resource requirements.

4.9 Shared state cluster management

Could multiple independent schedulers do a better cluster management job than monolithic or two-level schedulers? The designers of the Omega system understood that efficient scheduling of large clusters is a very hard problem due to the scale of the system, combined with the workload diversity, and that only a novel approach should be considered [442].

The workload of Google systems targeted by Omega was the mix of production/service and batch jobs discussed in Section 4.7. More than 80% of the workload are short batch jobs, spawning a large number of tasks. A larger share of resources, 55–80%, is allocated to production jobs running for extended periods of time and with fewer tasks than the batch jobs. The scheduling requirements are: short turnaround time for batch jobs and strict availability and performance targets for production jobs.

The scheduler workload increases with cluster size and with the granularity of the tasks to be scheduled. The finer the task granularity, the more scheduler decisions have to be made; thus the likelihood of spatial and temporal resource fragmentation and lower resource utilization is higher. The solution adopted in Omega's design is to allow multiple independent schedulers to access a *shared cluster state* protected with a lock-free optimistic concurrency control algorithm.

In Omega, there is no central resource allocator, and each scheduler has access to all cluster resources. A scheduler has its own private and frequently updated copy of the *shared cluster state*, a resilient master copy of the state of all cluster resources. Whenever it makes a resource allocation decision, a scheduler updates the shared cluster state in an *atomic transaction*.

In case of conflict among tasks, at most one commit succeeds, and the resource is allocated to the winner. Then, the shared cluster state re-syncs with local copies of all schedulers. The losers may then retry at a later time to gain access to the resource and can be successful after the resource has been released by the task holding it.

Multiple schedulers may attempt to allocate the same resource at the same time, thus there is the possibility of conflict. An optimal solution to this problem depends upon the frequency of conflicts. An optimistic approach increases parallelism and assumes that conflicts seldom occur and, when detected, they can be resolved efficiently. A pessimistic approach used by Mesos is to ensure that a resource is available to only one framework scheduler at a time.

Table 4.2 A side-by-side comparison of four types of schedulers. The schedulers differ in terms of: (a) scope, the set of resources controlled; (b) possibility of conflict and conflict resolution method; (c) allocation granularity, gang scheduling versus task-by-task; and (d) scheduling policy.

Scheduler	Resources	Conflict	Granularity	Policy
Monolithic e.g., Borg	All available resources	None	Global	Priority & preemption
Static partition e.g., Dryad	Fixed subset of resources	None	Per-partition policy	Scheduler-dependent
Two-level e.g., Mesos	Dynamic subset of resources	Pessimistic	Gang scheduling	Strict fairness
Shared-state e.g., Omega	All available	Optimistic	Per-scheduler policy	Priority & preemption

Jobs typically spawn multiple tasks, and the next question is whether all tasks of a job should be allocated all the resources they need at the same time when the job starts execution, a strategy called co-scheduling or gang scheduling. The alternative is to allocate resources only at the time when a task needs them. In the former case, resources end up being idle until the tasks actually need them, and the average resource utilization decreases. In the latter case, there is a chance of deadlock as some tasks need resources allocated to other tasks, while those holding these resources need the resources held by the first group of tasks.

Several metrics are useful to compare the effectiveness of large cluster schedulers. If we view scheduling as a service and a scheduling request as a transaction, then the time elapsed from the instance a job is submitted until the scheduler attempts to schedule the job is the waiting time; the time required to schedule a job is the service time. The waiting time and the service time are two important metrics for scheduler effectiveness. Conflict resolution is a component of the scheduler service time for a shared-state scheduler like Omega. The *conflict fraction* is the average number of conflicts per successful transaction.

The service time has two components, t_{sch_job}, the overhead for scheduling a job, and t_{sch_task}, the time to schedule a task of the job; thus, the total time to make a scheduling decision for a job with n tasks is:

$$t_{sch} = t_{wait} + t_{sch\_job} + n \times t_{sch\_tks}. \tag{4.1}$$

A monolithic scheduler using a centralized scheduling algorithm does not scale up. Another solution is to statically partition a cluster into sub-clusters allocated to different types of workloads. This policy is far from optimal due to fragmentation because the balance between different types of workload changes over time. A two-level scheduler has its own limitations, as we have seen in Section 4.6. Table 4.2 provides a side-by-side comparison of monolithic schedulers, schedulers for static-partitioned clusters, two-level schedulers, and multiple, independent, shared-state schedulers.

A light weight simulator was used for a comparative study of Omega and the other schedulers. The two-level scheduling model in this simulator emulates Mesos and achieves fairness by alternately offering all available cluster resources to different schedulers. It assumes low-intensity tasks, thus resources become available frequently and scheduler decisions are quick. As a result, a long scheduler decision time means that subsets of cluster resources are unavailable to other schedulers for extended periods of time. The simulation results show that Omega is scalable, and at realistic workloads, there is little interference among independent schedulers. The simulator was also used to investigate gang scheduling [34] useful for MapReduce applications.

A more accurate, trace-driven scheduler can be used to gain further insights into scheduler conflicts. Trace-driven simulation is quite challenging; moreover, a large number of simplifying assumptions limit the accuracy of the simulator results. Neither the machine failures, nor the disparity between resource requests and the actual usage of those resources in the traces, are simulated by the trace-driven simulator designed for Omega. The results produced by the trace-driven simulator are consistent with the ones provided by the light-weight simulator.

4.10 QoS-aware cluster management

Quality of Service (QoS) guarantees are important for the designers of cloud applications and for the users of computer clouds who wish to enforce a well-defined range of response time, execution time, or other significant performance metrics for their cloud workloads. Enforcing workload constraints is far from trivial, ergo few cluster management systems could legitimately claim adequate QoS support.

Two aspects of cluster management, resource allocation and resource assignment, play a key role for supporting QoS guarantees. *Resource allocation* is the process of determining the amount of resources needed by a workload, while *resource assignment* means identifying the location of resources that satisfy an allocation. Both aspects of resource management require the ability to classify a given workload as a member of one of several distinct classes. Once classified, the steps are to allocate to the workload precisely the amount of resources typical for that class, assign the resources, monitor the workload execution, and adjust this amount if needed.

QoS and workload classification. Workload classification is a challenging problem due to the wide spectrum of system workloads. Thus, an effective filtering mechanism is needed to support a classification algorithm capable to make real-time decisions. Classification is widely used by recommender systems such as the one used by Netflix. A recommender system seeks to predict the preference of a user for an item by filtering information from multiple users regarding the item. Such systems are used to recommend research articles, books, movies, music, news, and any other imaginable item.

A short diversion to the Netflix Challenge [56,135] could help understanding the basis of the classification technique for a QoS-aware cluster management. The Netflix Challenge uses Singular Value Decomposition (SVD) and PQ-tree[13] reconstruction [53,501]. SVD input is a sparse matrix A of rank r describing a system of n viewers and m movies:

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{pmatrix} = U \Sigma V^t \tag{4.2}$$

[13] A PQ tree is a tree-based data structure that represents a family of permutations on a set of elements, that is used to solve problems where the goal is to find an ordering that satisfies various constraints.

with

$$U_{m,r} = \begin{pmatrix} u_{1,1} & u_{1,2} & \dots & u_{1,r} \\ u_{2,1} & u_{2,2} & \dots & u_{2,r} \\ \vdots & \vdots & \vdots & \vdots \\ u_{m,1} & u_{m,2} & \dots & u_{m,r} \end{pmatrix} \quad \text{and} \quad V_{r,n} = \begin{pmatrix} v_{1,1} & v_{1,2} & \dots & v_{1,n} \\ v_{2,1} & v_{2,2} & \dots & v_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ v_{r,1} & v_{r,2} & \dots & v_{r,n} \end{pmatrix}. \tag{4.3}$$

Σ, the diagonal matrix of singular values of matrix A, is:

$$\Sigma_{r,r} = \begin{pmatrix} \sigma_{1,1} & 0 & \dots & 0 \\ 0 & \sigma_{2,2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_{r,r} \end{pmatrix}. \tag{4.4}$$

The ratings are the matrix elements $a_{ij} \in A$, $1 \le i \le m$, $1 \le j \le n$. SVD decomposes matrix A as $A = U \cdot \Sigma \cdot V^t$ with U being the matrix of left singular vectors representing correlation between rows of A and the similarity values, Σ is the matrix of similar values, and V is the matrix of right singular vectors representing the correlation between columns of A and the similarity values.

PQ-reconstruction with Stochastic Gradient Descent (SGD) [53] uses matrices U and V to reconstruct the missing entries in matrix A. The initial reconstruction of A uses $P^t = \Sigma \cdot V^t$ and $V = U$. SGD iterates over the elements $R = [r_{ui}]$ of $R = Q \cdot P^t$ until convergence. The iteration process uses two empirically determined values, η and λ, the *learning rate* and the SGD *regularization factor*, respectively. Two parameters μ and b_u the average rating and a user bias that account for the divergence of some viewers from the norm, respectively, are also used. The error ϵ_{ui} and the values q_i and p_u at each iteration are computed as:

$$\begin{aligned} \epsilon_{ui} &\leftarrow r_{ui} - \mu - b_u - q_i \cdot p_u^t \\ q_i &\leftarrow q_i + \eta(\epsilon_{ui} p_u - \lambda q_i) \\ p_u &\leftarrow p_u + \eta(\epsilon_{ui} q_i - \lambda p_u) \end{aligned} \quad . \tag{4.5}$$

The iteration continues until the L2 norm of the error becomes arbitrarily small

$$|\,\epsilon\,|_{L2} = \sqrt{\sum_{u,i} |\,\epsilon_{ui}\,|^2} \le \epsilon. \tag{4.6}$$

The convergence speed of stochastic gradient descent is limited by the noisy approximation of the true gradient. When the gains decrease too slowly, the variance of the parameter estimate decreases equally slowly. When the gains decrease too quickly, the expectation of the parameter estimate takes a very long time to approach the optimum. SVD complexity is $\mathcal{O}(\min(n^2 m, m^2 n))$, and the complexity of PQ-reconstruction with SGD is $\mathcal{O}(n \times m)$.

Quasar. A performance-centric approach for cluster management is implemented by two systems developed at Stanford University, Quasar [136] and its predecessor Paragon [134]. While Paragon handles only resource assignment, Quasar implements resource allocation, as well as resource assignment.

Quasar is implemented in C, C++ and Python, runs on Linux and OS X, and supports applications written in C/C++, Java, and Python. Applications run unaltered, there is no need to modify them for running under Quasar.

Quasar is based on several innovative ideas. One of them regards reservation systems and the realization that users are seldom able to accurately predict the resource needs of their applications. Moreover, performance isolation, though highly desirable, is hard to implement. As a result, the execution time and the resources used by an application can be affected by other applications sharing the same physical platform(s). This is the reason why reservation systems often lead to underutilization of resources and why QoS guarantees are seldom offered.

Quasar presents a high-level interface to allow users, as well as schedules integrated in widely used frameworks, to express constraints for the workloads they manage. These constraints are then translated into actionable resource-allocation decisions. Classification techniques are then used to evaluate the impact of these decisions on all system workloads.

Performance constraints differ for different types of workloads. For transaction processing systems, the system bandwidth, expressed as number of queries per second, represents a meaningful constraint because it reflects the response time experienced by users. For large batch workloads, e.g., the ones involving frameworks such as *Hadoop*, the execution time captures the expectations of the end-users.

To operate efficiently, the classification algorithms is based on four parameters: resources per node, number of nodes for allocation, server type, and degree of interference for assignment. The results of the four independent classifications are combined by a greedy algorithm used to determine as accurately as feasible the set of resources needed to satisfy the performance constraints. The system constantly monitors the workload performance and adjusts the allocations when feasible.

Rather than relying exclusively on the user's characterization of the workload, the classification system combines information gathered about the workload from prescreening with information from a database about past workloads. Once accepted in the system, a workload is profiled during a short execution on a few servers with two randomly selected scale-up allocations.

For example, *Hadoop* workloads are profiled for a small number of map tasks and two configurations of parameters, such as the number of mappers per node, the size of the Java AM heap, the block size, the amount of memory per task, the replication factor, and the compression factor. A configuration includes all relevant data for the workload. The configuration data is uploaded into a table with workloads as rows and scale-up configurations as columns. To constrain the number of columns, the entries are quantized to integer multiples of cores and blocks of memory and storage.

The classification engine and the scheduler. The classification engine distinguishes between the allocation of more servers to a workload, called *resource scale out*, and additional resources from servers already allocated to the workload, called *resource scale up*. The Quasar classification engine carries out, for each workload, four classifications for scale up, scale out, heterogeneity, and interference. Some workloads may require both types of scaling, others one type or another. For example, Quasar may monitor the number of queries per second and the latency for a web server and apply both types of scaling. Initially, Quasar was focused on compute cores, memory, and storage capacity with the expectation to soon also cover the network bandwidth.

In addition to scale up and scale out, there are two other types of classifications, heterogeneity and interference. To operate with a matrix of small dimensions and, thus, reduce computational complexity, the four types of classifications are done independently and concurrently. The greedy scheduler combines data from the four classifications.

The scale-up classification evaluates how the number of cores, the cache, and the memory size affect performance. The scale-out classification is only applied to several types of workloads that can use multiple servers, and profiling is done with the same parameters as for the scale-up classification. The heterogeneity classification is the result of profiling the workload on several randomly selected servers. Lastly, interference classification reflects the sensitivity and tolerance of other workloads using shared cores, cache, memory, and communication bandwidth.

The objective of the greedy scheduler is to allocate to each workload the least amount of resources enabling it to meet its SLO (Service Level Objectives).[14] For each allocation request, the scheduler ranks available servers based on the resource quality, e.g., the sustainable throughput combined with minimal interference. First, it attempts vertical scaling, allocating more resources on each node, and then, if necessary, switches to horizontal scaling allocating additional nodes, while keeping the total number of nodes as low as feasible.

Resources are allocated to applications on a FCFS basis. This could led to sub-optimal assignments, but such assignments can be easily detected by sampling a few workloads. The scheduler also implements admission control to prevent oversubscription.

In summary, Quasar provides QoS guarantees and, at the same time, increases resource utilization. The process starts with an initial profiling of an application with short runs. The information from the initial profiling is expanded with information regarding four factors that can affect performance, scale up, scale out, heterogeneity, and interference. Then, the greedy scheduler uses the classification output to allocate resources that enable SLO compliance and maximize resource utilization.

4.11 Resource isolation

A recurring theme of this chapter is that a cluster management systems must perform well for a mix of applications and deliver the performance promised by the strict Service Level Objectives (SLOs) for each workload. The dominant components of this application mix are *latency-critical* workloads, e.g., web search, and *best-effort* batch workloads, e.g., Hadoop. The two types of workloads share the servers and compete with each other for their resources.

The resource management systems discussed up to now act at the level of a cluster, but cannot be very effective at the level of individual servers or processors. First, they cannot have accurate information simply because the state of processors changes rapidly and communication delays prohibit a timely reaction to these changes. Secondly, a centralized or even a distributed system for fine-grained, server-level resource tuning would not be scalable.

Each server should react to changing demands and dynamically alter the balance of resources used by co-located workloads. A system with feedback is needed to implement an *iso-latency policy*, in other words, to supply sufficient resources so that SLOs are met. More bluntly, this means allowing Latency Critical (LC) workloads to expand their resource portfolio at the expense of co-located best-effort workloads.

This is the basic philosophy of the Heracles system developed at Stanford and Google [312]. In this section, we discuss the real-time mechanisms used by Heracles controller to isolate co-located

[14] A service level objective is a key element of a SLA. SLOs are agreed upon as a means of measuring the performance of the CSP and are a way of avoiding disputes between the users and the CSP based on misunderstanding.

workloads. In this context the term “isolate” means to prevent the best-effort workload to interfere with the SLO of the latency-critical workload.

Latency-critical workloads. A closer look at three Google latency-critical workloads, *websearch, ml_cluster*, and *memkeyval,* helps us better understand why resource isolation is necessary for co-located workloads. The first, *websearch*, is the query component of the web search service. Every query has a large fan-out to thousands of leaf nodes, each one of them processing the query on a shard of the search index stored in DRAM. Each leaf node has strict SLO of tens of ms. This task is compute intensive because it has to rank search hits and has a small working set of instructions, a large memory footprint, and a moderate DRAM bandwidth.

The second, *ml_cluster*, is a stand-alone service using machine-learning for assigning a snippet of text to a cluster. Its SLO is also of tens of ms. It is slightly less CPU intensive, requiring a larger memory bandwidth and lower network bandwidth than *memkeyval*. Each request for this service has a small cache footprint, but a high rate of pending requests puts pressure on the cache and DRAM.

The third, *memkeyval*, is an in-memory, key-value store used by the back-end of the web service. Its SLO latency is of hundreds of ms. The high request rate makes this service compute intensive mostly due to the CPU cycles needed for network protocol processing.

Sharing resources of individual servers is complicated because the intensity of the LC workloads at any given time is unpredictable; therefore, their latency constrains are unlikely to be satisfied at times of peak demand unless special precautions are taken. Resource reservation at the level needed for peak demand of LC workloads may come to mind first. But this naive solution is wasteful: It leads to low or extremely low resource utilization and, thus, the need for better alternatives.

Processor resources. Processor resources subject to dynamic scaling and the mechanisms for resource isolation for each one of them are discussed next. Physical cores, cache, DRAM, the power supplied to the processor, and network bandwidth are all resources that affect the ability of an LC workload to satisfy the SLO constraints. Individual resource isolation is not sufficient; cross-resource interactions deserve close scrutiny. For example, contention for cache affects DRAM bandwidth; a large network bandwidth allocated to query processing affects CPU utilization because communication protocols consume a large number of CPU cycles.

Processor cores are the engine delivering CPU cycles and an obvious target of dynamic rather than static allocation for co-located workloads. This problem is complicated by Hyper-Threading (HT) in multicore Intel processors. HT is a proprietary form of SMT (simultaneous multi-threading) discussed in Section 4.3. HT takes advantage of superscalar architecture and increases the number of independent instructions in the pipeline. For each physical core the OS uses two virtual cores and shares the workload between them when possible. This interferes with the instruction execution, shared caches, and TLB (translation look-aside buffer)[15] operations.

Dynamic frequency scaling is a technique for adjusting the clock rate for cores sharing a socket. The higher the frequency, the more instructions are executed per unit of time by each core, and the larger is the processor power consumption. Clock frequency is related to the operating voltage of the processor. The *dynamic voltage scaling* is a power conservation technique often used together with frequency scaling, thus the name *dynamic voltage and frequency scaling* (DVFS).

[15] TLB can be viewed as a cache for dynamic address translation; it holds the physical address of recently used pages in virtual memory.

The *overclocking* techniques based on DVFS opportunistically increase the clock frequency of processor cores above the nominal rate when the workload increases. To allow the cores of an Intel processor to adjust their clock frequency independently, the *Enhanced Intel SpeedStep* technology option should be enabled in the BIOS.[16] Heracles reacts to lower best-effort workloads by reducing the number of cores assigned to best-effort tasks.

The cycle stalls limit the effective IPC (instructions per clock cycle) of individual cores. This means that the shared *last-level cache*[17] is another critical resource shared by the LC and best-effort co-located workloads and should be dynamically allocated. Lastly, the DRAM bandwidth can greatly affect the performance of applications with a large memory footprint.

One answer to the question of how to implement an isolation mechanism allowing LC workloads to scale up could be to delegate this task to the local scheduler. Why not use existing work-conserving[18] real-time schedulers such as SCHED_FIFO or CFS, the Completely Fair Scheduler? A short detour into the world of real-time schedulers used by most operating systems should convince us that these schedulers are designed to support data streaming and cannot satisfy the SLO requirement of LC tasks.

The SCHED_FIFO scheduler allocates the CPU to a high-priority process for as long it wants it, subject only to the needs of higher-priority realtime processes. It uses the concept of "real-time bandwidth," *rt_bandwidth*, to mitigate the conflicts between several classes of processes; once a process exceeds its allocated *rt_bandwidth,* it is suspended. The bandwidth of LC tasks changes so SCHED_FIFO scheduler cannot accomplish what we need.

CFS uses a red–black tree[19] in which the nodes are structures derived from the general *task_struct* process descriptor with additional information. CFS is based on the "sleeper fairness" concept enforcing the rule that interactive tasks that spend most of their time waiting for user input or other events get a comparable share of CPU time as other processes, when they need it. As threads are spawned for every new query request, we see that this approach is not satisfactory either.

Communication bandwidth sharing inside the server is controlled by the OS. Linux can be configured to guarantee outgoing bandwidth for the latency-critical workload. For incoming traffic it is necessary to throttle the core allocation until flow-control mechanisms of communication protocols are triggered. Communication among servers is supported by the cluster interconnection fabric and can be ensured by communication protocols that give priority to short messages typical for the latency-critical workloads.

Architectural support is needed for workload isolation at the microarchitecture level. Newer generations of Intel processors, such as the Xeon E5-2600 v3 family, provide the hardware framework to manage a shared resource, like a last-level cache through Cache Monitoring Technology (CMT) and Cache Allocation Technology (CAT). CMT enables an OS or a hypervisor to determine the usage of

[16] The basic input/output system (BIOS) is invoked after a computer system is powered up to load the OS and later to manage the data flow between the OS and devices such as keyboard, mouse, disk, video adapter, and printer.

[17] Multicore processors have multiple level caches. The last level cache (LLC) is the cache called before accessing memory. Each core has its own L1 I-cache (instruction cache) and D-cache (data cache). Sometimes, two cores share the same unified (instruction+ data) L2 cache and all cores share an L3 cache. In this case, the highest shared LLC is L3.

[18] A work-conserving scheduler tries to keep the resources busy, if there is work to be done, while a non-work conserving scheduler may leave resources idle while there is work to be done.

[19] A red–black tree is a self-balancing binary search tree where each node has a "color" bit (red or black) to ensure the tree remains approximately balanced during insertions and deletions.

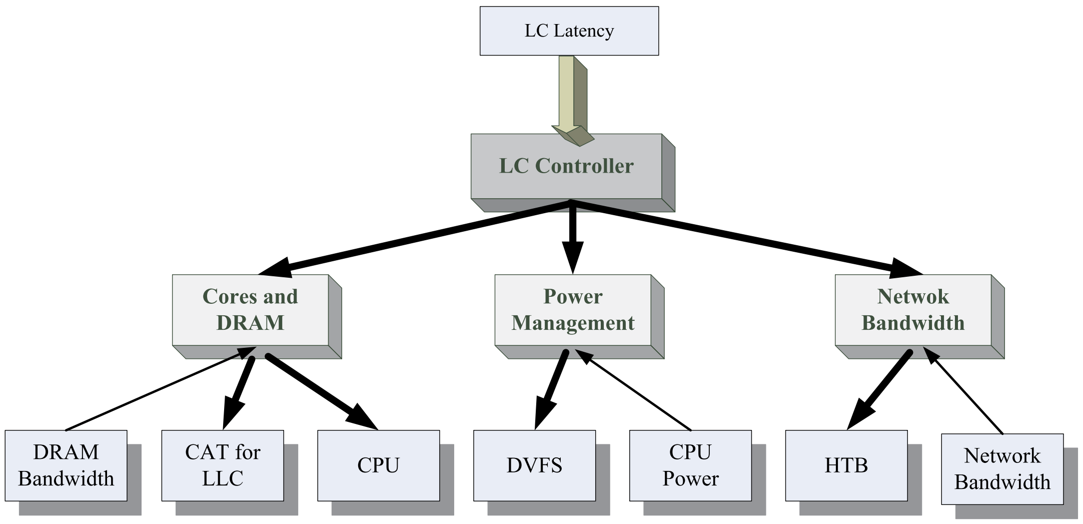

FIGURE 4.5

Heracles organization. The system runs on each server and controls isolation of a single-LC workload and multiple best-effort jobs. LC controller acts based on information regarding latency constraints of LC workload and manages three resource controllers for: (a) Core and DRAM—uses CAT for LLC management and acts based on DRAM bandwidth data; (b) Power management—acts on DVFS using information on CPU power consumption; and (c) Network bandwidth—enforces bandwidth limits for outgoing traffic from best-effort tasks using *qdisc* scheduler with HTB (token bucket queuing) discipline with information provided by network bandwidth-monitoring system.

cache by applications running on the platform. It assigns a Resource Monitoring ID (RMID) to each of the applications or VMs scheduled to run on a core and monitors cache occupancy on a per-RMID basis. CAT allows access to portions of the cache according to the *class of service* (COS).

Heracles organization and operation. Heracles runs as a separate instance on each server and manages the local interactions between the latency-critical and best-effort jobs. Fig. 4.5 shows the latency-critical controller and the three resource controllers for cores, caches, and DRAM for power management and for communication bandwidth. The controller uses the *slack*, the difference between the SLO target and the tail of the measured performance index. A negative value of the latency slack means that the time-sensitive workload has increased in intensity and is getting close to exceeding its SLO latency, and, thus, it requires more resources.

The latency-critical controller is activated every 15 units of time and uses as input the latency-critical application latency and its load. When the slack is negative or when the latency-critical load is larger than 80% of capacity, the best-effort application is disabled. If the slack is less than 10%, the best-effort task is not allowed to grow, and when the slack further decreases to 5%, then two cores allocated to best-effort tasks are removed. Best-effort is enabled when the latency-critical load is less than 80%.

The operation of the latency-critical controller is described by the following pseudocode:

```
while True
     latency = Poll_Latency-critical - AppLatency
     load  = Poll_latency-critical -AppLoad
     slack = (target - latency)/target
     if slack < 0
          Disable Best-effort
          EnterCoolDown()
     elseif load > 0.85
          Disable best-effort
     elseif load < 0.80
          Enable Best-effort
     else if slack < 0.10
          Dissallow Best-effort Growth
          if slack < 0.05
          Best-effort_core.RemoveTwoCores
sleep{15}
```

There is a strong interaction among the number of cores allocated to a workload, its LLC cache, and DRAM requirements. This strong correlation explains why a single specialized controller is dedicated to the management of cores, LLC, and DRAM. The main objective of this controller is to avoid memory bandwidth saturation. The high-water mark that triggers action is 90% of the peak streaming DRAM bandwidth, measured by the value of hardware counters for per-core memory traffic. When this limit is reached, cores are removed from best-effort tasks.

When the DRAM bandwidth is not saturated, the gradient descent method is used to find the largest number of cores and cache partitions by alternating between increasing the number of cores and increasing the number of cache partitions allocated to best-effort tasks subject to the condition that the SLO of latency-critical tasks are satisfied. The power controller determines if there is sufficient power slack to guarantee latency-critical SLO with the lowest clock frequency. The system was evaluated with three latency-critical workloads, and the results show an average 90% resource utilization without any SLO violations.

4.12 In-memory cluster computing for Big Data

The distinction between system and application software is blurred in cloud computing. The software stack includes components based on abstractions that combine aspects of applications and system management. Two systems developed at UC Berkeley, Spark [533] and Tachyon [302], are perfect examples of such elements of a cloud software stack.

It is unrealistic to assume that very large clusters could accommodate in-memory storage of petabytes or more in the foreseeable future. Even if storage costs will decline dramatically, the intensive communication among the servers will limit the performance. There are iterative and other classes of Big Data applications where a stable subset of the input data is used repeatedly. In such cases, dramatic performance improvements can be expected if a *working set* of input data is identified, loaded in memory, and kept for future use.

Obvious examples of such applications are those involving multiple databases and multiple queries across them and interactive data mining involving multiple queries over the same subset of data. Another example of an iterative algorithm is the *PageRank* algorithm [72], where data sharing is more complex. At each iteration, a document with *rank* $r^{(i)}$ and n neighbors sends a contribution of $\frac{r^{(i)}}{n}$ to each one of them and updates its own rank as

$$r^{i+1} = \frac{\alpha}{N} + (1-\alpha)\sum_{j=1}^{n} c_j \tag{4.7}$$

with α as the dumping factor, and N is the number of the documents in the database, and the sum over all contributions it received.

A distributed shared-memory (DSM) is a solution to in-memory data reuse. DSM allows fine-grained operations, yet access to individual data elements is not particularly useful for the class of applications discussed in this section. DSM does not support effective fault-recovery and data distribution and does not lead to significant performance improvements. Ad hoc solutions to in-memory data reuse for various frameworks have been implemented, e.g., HaLoop [77] for MapReduce.

The question is whether a data sharing abstraction suitable for a broad class of applications and use cases can be developed for supporting a restricted form of shared memory based on *coarse-grained* transformations. This abstraction should provide a simple, yet expressive, user-interface allowing the end-user to describe data transformations, as well as powerful behind-the-scene mechanisms to carry out the data manipulations in a manner consistent with the system configuration and the current state of the system.

A data sharing abstraction. The concept of *Resilient Distributed Dataset* (RDD), for fault-tolerant, parallel data structures was introduced in [532]. RDD allows a user to keep intermediate results and optimizes their placement in the memory of a large cluster. The user interface of RDD exposes: (1) partitions, atomic pieces of the dataset; (2) dependencies on the parent RDD; (3) a function for constructing the dataset; and (4) metadata about data location.

Spark provides a set of operators to efficiently manipulate such persistent datasets using a set of coarse-grained operations such as *map, union, sample,* and *join*. *Map* creates an object with the same partitions and preferred locations as its parent, but applies the function used as an argument to the call to the *iterator* method applied to the parent's records. *Union* applied to two RDDs returns an RDD whose partitions are the union of the partitions of the two parents. *Sampling* is similar to *map*, but the RDD stores for each partition a random number generator to deterministically sample parent records. *Join* creates an RDD with either two narrow, two wide, or mixed dependencies.

The driver program created by the user launches at run-time multiple workers that read data from a distributed file system such as HDFS and distributes the data across multiple RDD partitions. Tasks locality is ensured by the delay scheduling algorithm [532] used by the *Spark's* scheduler. If a task fails, the system restarts it on a different node if the parent is available. Partitions too large to fit in memory are stored on the secondary storage, possibly on solid-state disks, if available.

Spark [533] and RDDs are restricted to I/O intensive applications performing bulk writing. Spark and Tachyon [302], discussed later in this section, share the concept of *lineage* to support error recovery without the need to replicate the data. Lineage means to trace back a descendent from a common ancestor and, in the context of this discussion, it means that lost output is recovered by again carrying out the tasks that created the lost data in the first place.

Spark driver programs. Imagine that an application wants to access a large log file stored in HDFS as a collection of lines of text. We wish to create a persistent dataset called *errors* of lines starting with the prefix "ERROR" distributed over the memory of the cluster, count the number of lines in this persistent dataset, and then carry out two more actions: (1) count errors containing the string "MySQL"; and (2) return the time of the errors, assuming that time is the third field in a tab-separated format of an array called HDFS. The following Spark code will do the job:

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
errors.count
errors.filter(_.contains("MySQL")).count()
errors.filter(_.contains("HDFS")).map(_.split('\t')(3)).collect()
```

Notice that the *lines* dataset is not stored, only the much smaller *errors* dataset is stored in memory and used for the three actions. Behind the scenes, the Spark scheduler will dispatch a set of tasks carrying the last two transformations to the nodes where the cached partitions of *errors* reside.

The Spark code for the *PageRank* algorithm summarized in Eq. (4.7) is:

```
// Load graph as an RDD of (URL, outlinks) pairs
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
// Build RDD of (targetURL, float) pairs with contributions sent by each page
val contribs = links.join(ranks).flatMap {
   (url, (links, rank)) =>links.map(dest => (dest, rank/links.size))
   }
// Sum contributions by URL and get new ranks
ranks = contribs.reduceByKey((x,y) => x+y).mapValues(sum => a/N + (1-a)*sum)
}
```

The *map, reduce*, and *join* operations and the lineage datasets of the graph of the several iterations of the *PageRank* algorithm are shown in Fig. 4.6.

A new *ranks* dataset is created at each iteration, and it is wise to call the *persist* action to save the dataset on secondary storage using the *RELIABLE* argument and reduce the recovery time in case of system failure. The *ranks* can be partitioned in the same way as the *links* to ensure that the *join* operation requires no additional communication.

Spark dependencies. An important design decision in Spark was to distinguish between *narrow* and *wide* dependencies. The former allow only one partition of RDD to use the parent RDD and enable pipelined execution on one cluster to compute all parent partitions, for example, enable the application of a *map* followed by a *filter* on an element-by-element basis. In addition, recovery after a node failure is more efficient for narrow dependencies because only the lost parent needs to be recomputed and this can be done in parallel.

On the other hand, wide dependencies enable multiple children to depend on a single parent. Data from all parent partitions must be available and shuffled across the nodes for MapReduce operations.

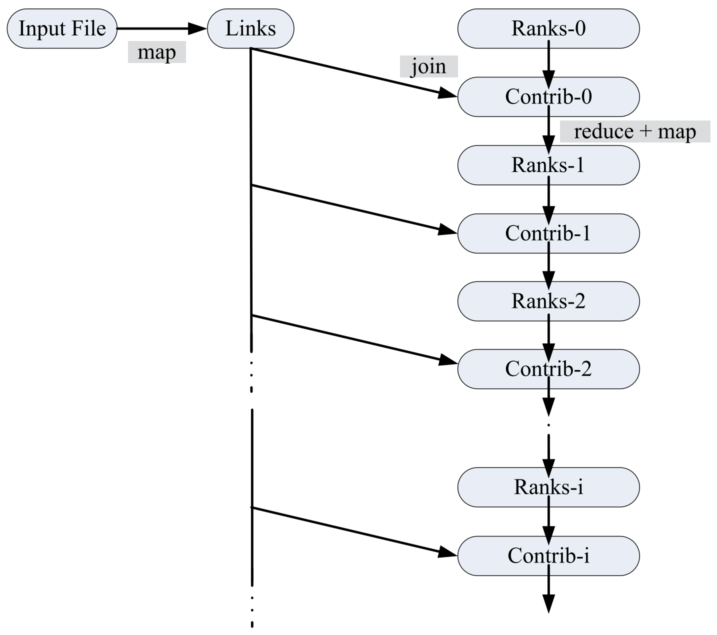

FIGURE 4.6

The lineage dataset for several iterations of the *PageRank* algorithm and the *map, reduce*, and *join* operations.

This also complicates recovery after a node failure. Fig. 4.7 shows the effect of dependencies on RDD partitions as a result of *map, filter, union, join,* and *group* operations.

Persistent RDD can be stored in memory either as deserialized Java objects, or as serialized data, and can also be stored on disk. Lineage is a very effective tool to recover RDDs after a node failure. Spark also supports checkpointing, and this is particularly useful for large lineage graphs.

According to [533] "Spark is up to 20 times faster than Hadoop for iterative applications, speeds-up a data analytics report 40 times and can be used interactively to scan a 1 TB dataset with 5–7 seconds latency." It is also very powerful; only 200 lines of Spark code implement the HaLoop model for MapReduce applications. HaLoop [77] extends MapReduce with programming support for iterative applications and improves efficiency by adding various caching mechanisms and by making the task scheduler loop-aware.

These results show that caching improves dramatically Big Data applications performance running on storage systems that support only append operations, such as HGFS. Lost data can be recovered by lineage; there is no need for replication of immutable data. On the other hand, fault-tolerance of applications involving write operations is more challenging. Fault tolerance based on data replication incurs a significant performance penalty when a data item is written on multiple nodes of a cluster.

The write bandwidth throughput for both hard disks and solid-state disks is three orders of magnitude lower than the memory bandwidth. Only, the random access latency of solid-state disks is much lower than the latency of hard disks, and their sequential I/O bandwidth is not larger. The network

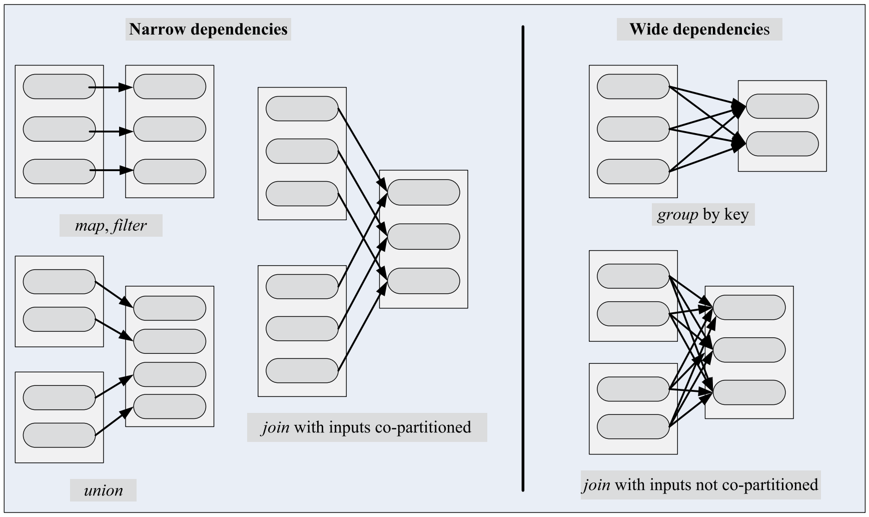
Narrow dependencies
Wide dependencies
map, filter
union
join with inputs co-partitioned
group by key
join with inputs not co-partitioned

FIGURE 4.7

Narrow and wide dependencies in Spark. Arrows show the transformation of the partitions of one RDD due to *map, filter*, two RDDs for *union*, and two RDDs for *join* with inputs co-partitioned for narrow dependencies, when only one partition of RDD is allowed to use the parent RDD. For wide dependencies, when multiple children depend upon a single parent, two transformations are presented: a *group* by key of one RDD and *join* with input not co-partitioned for two RDDs.

bandwidth is also orders of magnitude lower than the memory bandwidth. The system discussed next addresses precisely the problem of supporting fault-tolerance for in-memory datasets where both read and write operations must be supported.

Tachyon. The system was designed for high throughput in-memory storage for applications performing both extensive reads and writes [302]. The name of the system targeting Big Data workloads discussed in Chapter 12, "Tachyon,[20]" was most likely chosen to reflect the performance of the system—in Greek "tachy" means "fast." To recover lost data, the system exploits the lineage concept used also by Spark and avoids data replication, which could dramatically affect its performance.

Fault-tolerant, in-memory caching of datasets, while supporting both read and write operations, requires answers to several challenging questions:

1. How to recover the lost data due to server failures?
2. How to limit the resources and the time needed to recover lost data?
3. How to identify frequently used files and recover them with high priority when lost?
4. How to avoid recovering temporary files?

[20] Tachyon is also the name given to a hypothetical particle that moves faster than the speed of light. In modern physics, "tachyon" refers to imaginary mass fields rather than to faster-than-light particles.

5. How to share resources among the two activities, running the jobs and re-computations?
6. How to ensure the system's fault-tolerance?
7. How to manage the storage for binaries necessary for data recovery?
8. How to choose files to be evicted when their cumulative size exceeds the available storage space?
9. How to deal with file-name changes?
10. How to deal with changes in the cluster's run-time environment?
11. How to support different frameworks?

The answers to these questions given by the designers of the system are discussed next. Checkpointing alone is not a solution for re-computation of lost data because periodic checkpointing leads to an unbounded recovery time. Lineage alone is also not feasible because the depth of the lineage graphs keeps growing, and the time to recompute the entire path to the leaf of the graph is prohibitive. The solution adopted by Tachyon is based on combined checkpointing and lineage.

The Edge algorithm introduced in [302] checkpoints only the leaves of the lineage directed acyclic graph (DAG). This strategy reduces the number of checkpointed files and limits the resources needed for recovery. The implicit assumption of this approach is that data is immutable between consecutive checkpoints. Data should be versioned if modified between two consecutive checkpoints, and different files produced from the same parent should have different IDs.

A read counter associated with every file is used as the file priority. Frequently, read files have a high priority, while temporary files with a low read count are avoided. Re-computation is done asynchronously in the background using the linage, thus, the interference with jobs running on the cluster is minimized and their SLOs can be guaranteed. The system is controlled by a *Tachyon Master* (TM) with several stand-by replicas ready to take over if the current master fails. If the master fails, a Paxos algorithm is used to elect the next master.

Computing the lineage of a file requires re-running the binaries of all applications executed from the instance the parent was created until the lost data was created. The workflow manager, a component of the TM, uses a DAG for each file and does a depth-first search (DFS) of nodes to reach the targeted files; it stops as soon as it reaches a node representing a file already in storage. So, it is fair to ask how much storage is necessary for the binaries of all jobs that can potentially be executed during the recovery process. Data gathered by Microsoft shows that a typical data center running some 1 000 jobs daily needs about 1 TB of storage for all the binaries executed over a period of one year [214].

Data eviction is based on a LRU (Least Recently Used) policy. This policy is justified by the access frequency and the by temporal locality. According to a cross-industry study [103], the file access in a large data center often follows a Zipf-like distribution, and 75% of re-accessing occurs within six hours. A file is uniquely identified by an immutable ID in the lineage information record to address file name changes. This ensures that the re-computation done according to the ordered list prescribed by lineage reads the same files and in the same order as in the original execution.

Among the most frequent changes in the cluster's runtime environment are changes of the version of a framework used to re-compute lost data and changes of the OS version. To address this problem, the system runs in a *Synchronous mode* before any such change when all un-replicated files are checkpointed and the new data is saved. Once this is done, this mode is disabled. Lastly, Tachyon requires a program written in a framework to provide information before writing a new file. This information is used to decide if the file should be in memory only and to recover a lost file using its lineage.

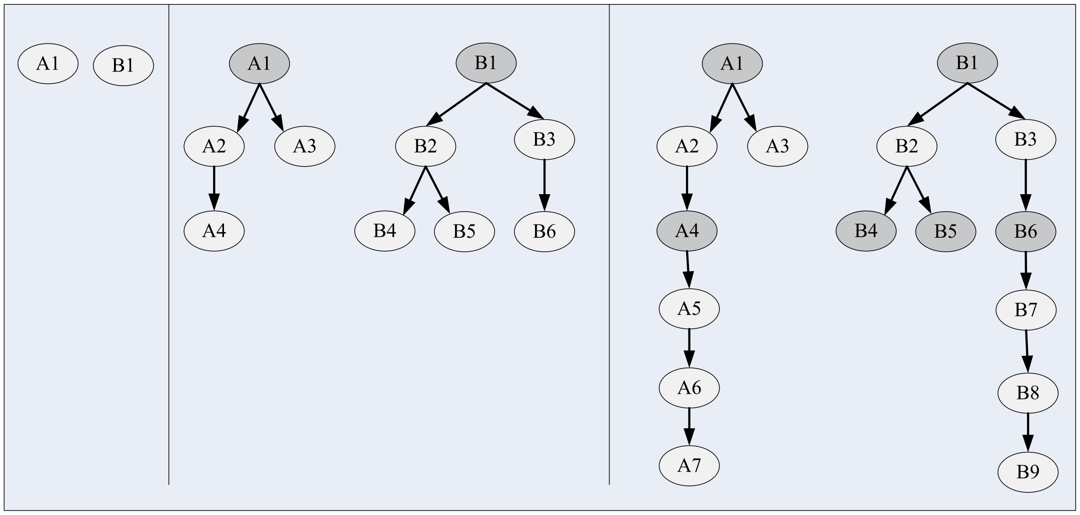

FIGURE 4.8

Edge algorithm. Dark-filled ellipses represent checkpointed files and light-filled ones represent un-checkpointed files. Lineage is represented by a DAG with files as vertices and edges representing the transformation of a parent to all of its descendants. Shown are the lineage DAGs of two files, $A1$ and $B1$. At each stage, only the leaf nodes of the lineage DAG are checkpointed. $A1$ and $B1$ are checkpointed first, then $A4$, $B4$, $B5$, and $B6$. In the next stage, not shown in the figure, only $A7$ and $B9$ will be checkpointed.

Edge algorithm. The algorithm assumes that the lineage is represented by a DAG with files as vertices and edges representing the transformation of a parent to all of its descendants. The algorithm checkpoints the leafs of the graph. For example, assume a lineage chain for a file A_0 including files $\{A_0, A_1, A_2, \dots A_i, \dots A_j, \dots\}$. Then, if there is a checkpoint of A_i and A_j is lost, the re-computation starts with the latest checkpoint, in this case A_i, rather than A_0. Fig. 4.8 shows the lineage DAGs of two files $A1$ and $B1$ and the leafs checkpointed at several instances of time; $A1$ and $B1$ are checkpointed first, then $A4$, $B4$, $B5$, and $B6$.

The Edge algorithm does not take into account priorities. A balanced algorithm alternates edge-driven checkpointing with priority-based checkpointing, allocating a fraction c of time the former and a fraction $(1-c)$ of time to the latter. To guarantee applications SLOs the recovery time for any file must be bounded.

Call W_i the time to checkpoint an edge i of the DAG and G_i to generate edge i from its ancestors. Then two bounds on the recovery time for any file are proven in [302]: (1) Edge checkpointing alone leads to the following bound of the recovery time

$$\mathcal{T}^{edge} = 3 \times M \quad \text{with} \quad M = \max_i \{T_i\}, \quad \text{and} \quad T_i = \max(W_i, G_i)^9. \tag{4.8}$$

This shows that the re-computation time is independent of the DAG's depth. (2) The bound for the recovery time for alternating edge and priority checkpointing is

$$\mathcal{T}^{edge,priority} = \frac{3 \times M}{c} \text{ with } M = \max_i\{T_i\}, \text{ and } T_i = \max\left(W_i, G_i\right). \tag{4.9}$$

Resource management. Resource management policies should address several concerns. When several files have to be recovered at the same time the system should consider data dependencies to avoid recursive task launching. Dynamic file checkpointing priorities are also necessary; low priority assigned to a file requested by a low priority job should be automatically increased when the same file is requested by a high priority job. The lineage record should be deleted after checkpointing to save space.

All resources should be dedicated to normal work when no recovery is necessary. Typical average server utilization rarely exceeds 30%, so most of the time, there are sufficient resources available for data recovery. But what to do when the system is near its capacity? Then the priority of the jobs and of the recovery come into play and is used by the cluster scheduler.

Tachyon could accommodate the two most frequently used scheduling policies for cluster resource management, priority based and fair-shared based scheduling. In case of *priority-based* scheduling, all re-computation jobs are given the lowest priority by default. Deadlock may occur unless precautions are taken.

For example, assume that a job $\mathcal{J}_i$ has a higher priority than the file $\mathcal{F}$ it needs to recover and that job J is scheduled to run. At the time when $\mathcal{J}_i$ needs access to $\mathcal{F}$, the job $\mathcal{R}_\mathcal{F}$ needed to recover $\mathcal{F}$ cannot run as it inherits the lower priority of $\mathcal{F}$. The solution is *priority inheritance*. In this case, $\mathcal{J}_i$ and, implicitly the job $\mathcal{R}_\mathcal{F}$, should inherit the priority of $\mathcal{J}_i$ that needs $\mathcal{F}$. If another job with an even higher priority needs the file $\mathcal{F}$ its priority, hence the priority of $\mathcal{R}_\mathcal{F}$, should increase again.

Assume now a *fair-share scheduler*. In this case, $W_1, W_2, \ldots W_i, \ldots$ are the weights of resources allocated to jobs $\mathcal{J}_1, \mathcal{J}_2, \ldots \mathcal{J}_i, \ldots$, respectively. The minimal share unit is $W_g = 1$. When files $\mathcal{F}_{i,1}$, $\mathcal{F}_{i,2}$, $\mathcal{F}_{i,3}$ of job $\mathcal{J}_i$ are lost the scheduler allocates the minimum weight W_g from the W_i to the three recovery jobs $\mathcal{R}_{\mathcal{J}_i,\mathcal{F}_1}$, $\mathcal{R}_{\mathcal{J}_i,\mathcal{F}_2}$ and $\mathcal{R}_{\mathcal{J}_i,\mathcal{F}_3}$. At the time job $\mathcal{J}_i$ needs to access the file $\mathcal{F}_{i,2}$ the scheduler allocates the fractions $(1-\alpha)$ and α of W_i to $\mathcal{J}_i$ and $\mathcal{R}_{\mathcal{J}_i,\mathcal{F}_2}$, respectively.

Tachyon implementation. Tachyon has two layers: the *lineage layer* tracks the sequence of jobs that have created a file; the *persistence layer* manages data in storage, can be any replication-based storage such as HDFS, and is used for asynchronous checkpoints. Tachyon has a master-slave architecture with several passive replicas of the master and worker daemons running on every cluster node and managing local resources. The lineage information is tracked by a workflow manager running inside the master. The workflow manager computes the order of checkpoints and interacts with the cluster resource manager to get resources needed for re-computations.

Each worker uses a RAM disk for storing memory-mapped files. The concept of wide and narrow dependences inherited from Spark is used to carry out the operations discussed earlier in this section. The system was implemented in about 36 000 lines of Java code and uses the *Zookeeper* for election of a new master when one fails.

Tachyon lineage can capture the requirements of MapReduce, and SQL, as well as Hadoop and Spark can run on top of it. According to [302] Tachyon has a 110 times higher write throughput and for realistic workloads, improves end-to-end latency four times compared with in-memory HDFS. It

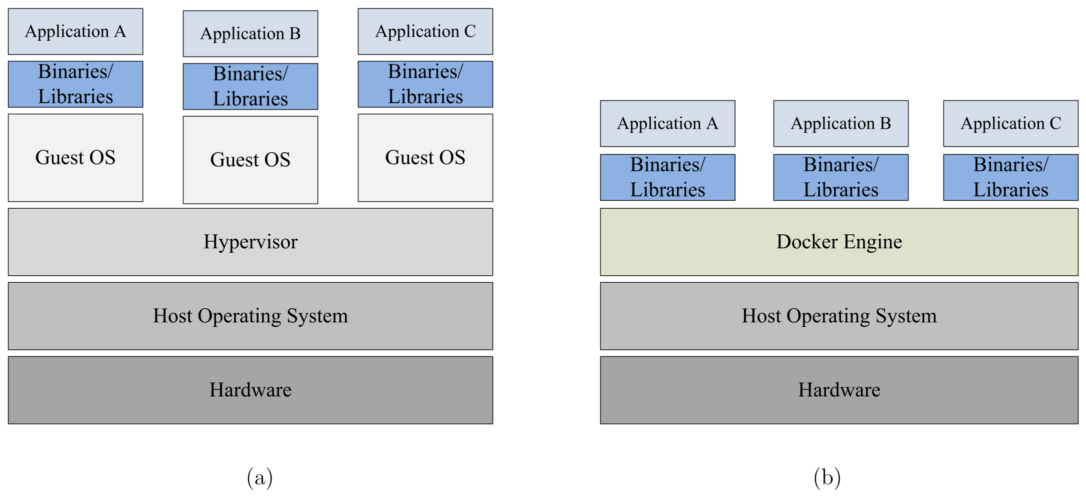

FIGURE 4.9

System organization for Virtual Machines and Docker containers. (a) VM; (b) Multiple Docker containers running on the same system share the OS kernel thus, have a smaller memory footprint and a shorter start-up time than VMs.

can also reduce network traffic by up to 50% because many temporary files are deleted before being checkpointed. Data from Facebook and Bing show that it consumes not more than 1.65% of cluster resources for re-computations.

4.13 Containers; Docker containers

This idea of containers was extended from file systems supported by *chroot* to other namespaces including the process IDs. Initially, the *cgroups* concept was implemented at Google in 2006 in the Linux kernel to isolate, control, limit, prioritize, and account for the resources (CPU, memory, disk I/O, networks) available to a set of processes, by freezing groups of processes and managing checkpointing and restarting. Resource limiting enforces the target set for resource utilization, while prioritization allows a group of processes to get a larger share of CPU cycles or a higher disk I/O throughput.

Docker created containers along with a set of tools with standard APIs; it also made the same container portable across all environments. “Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, system libraries—anything that can be installed on a server. This guarantees that the software will always run the same, regardless of its environment,” according to Docker. Fig. 4.9 depicts the organization of both VM- and container-based systems.

Containers are light weight, cost-effective in a public cloud environment, provide better performance, and require fewer hardware resources for a private cloud. Containers isolate an application from the underlying infrastructure and from other applications and support performance and security isolation. Multiple containers running on the same machine share the OS kernel, have a smaller memory footprint, and a shorter start-up time than VMs.

Another advantage of Docker containers is increased productivity. Containerization allows developers to choose the most suitable programming languages and software systems and eliminates the need to make copies of production code and install the same configuration in different environments. It also supports efficient up and down scaling of an application.

Docker ecosystem is built around a few concepts discussed next. An *image* is a blueprint of an application. A *container* consists of one or more images and runs the actual application. A *daemon* is a background service running on the host that manages building, running, and distributing Docker containers. The daemon is the process that runs under the operation system to which clients talk to. A *client* is a command line tool used by a user to interact with the daemon. A *hub* is a registry of Docker images, a directory of all available Docker images.

There are two types of images, base and child. *Base images* have no parent image, usually images with an OS like Ubuntu, or Busybox.[21] *Child images* are build on base images with additional functionality. *Official images* are maintained and supported by Docker, and have one word name; for example, *python, ubuntu* are base official images. *User images* are created and shared by users. A *Dockerfile* is a facility to automate the image creation, a text-file including Linux-like commands invoked by clients to create an image.

Docker Swarm exposes standard Docker API. Docker tools including Docker CLI, Docker Compose, Dokku, and Krane, work in this native Docker clustering. The distribution of it is packed as a Docker container and to set it up one only needs to install one of the service discovery tools and run the *swarm* container on all nodes, regardless of the OS.

Cloud computing has embraced containerization. Containers-as-a-Service (CasS) is geared toward efficiently running a single application. Several CSPs including Heroku, OpenShift, dotCloud and CloudFoundry use containers to support PaaS delivery model. Amazon, Google, Microsoft, OpenStack, Cloudstack and other CSPs offering IaaS implementations support containers. Container support by Amazon, Google, and Microsoft is overviewed next.

ECS is Amazon EC2 Container Service. It is straightforward for AWS users to create and manage ECS clusters, as ECS is integrated with existing services such as IAM for permissions, CloudTrail to get data regarding resources used by a container, CloudFormation for cluster lunching, and other services. AWS uses a custom scheduler/cluster manager for containers. Container hosts are regular EC2 instances. To deploy a containerized application on AWS one has to first publish the image on an AWS accessible registry e.g., the Docker Hub. Amazon ECS is free, while Google Container Engine, discussed next, is free up to five nodes. The charge granularity at AWS is one hour while at Google or Microsoft the charge is for the actual time used.

Google Container Engine (GKE). GKE is based on Kubernetes, an open source cluster manager available to Google developers and the community of outside users. Google's approach to containerization is slightly different, its emphasis is more on performance than ease of use as discussed in Section 4.14. Two billion containers are started at Google every week according to http://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/. GKE is integrated with other

[21] BusyBox is software providing several stripped-down Unix tools in a single executable file and running in environments such as Linux, Android, FreeBSD, or Debian.

services including Google Cloud Logging. Google users have access by default to private Docker registry and a JSON-based declarative syntax for configuration. The same syntax can be used to define what happens with the hosts. GKE can launch and terminate containers on different hosts as defined in the configuration file.

Microsoft Azur Container Service. The Azure Resources Manager API supports multiple orchestrations including Docker, and Apache Mesos.

In recent years the Open Container Initiative (OCI) was involved in an effort to create industry-standard container format and runtime systems under the Linux Foundation. The 40+ members of the OCI work together to make Docker more accessible to the large cloud users community. OCI's approach is to break Docker into small reusable components. In 2016 OCI has announced Docker Engine 1.11 which uses a daemon, *containers* to control *runC*[22] or other OCI compliant run-time systems to run containers. Users access the Docker Engine via a set of command and a user interface.

4.14 Kubernetes

Kubernetes is a cluster manager for containers. The origin of the word Kubernetes is in ancient Greek language: "kubernan" means to steer and "kubernetes" is helmsman. Kubernetes is an open source software system developed and used at Google for managing containerized applications in a clustered environment. Kubernetes bridges the gap between a clustered infrastructure and assumptions made by applications about their environments. Kubernetes is written in *Go*[23] and it is designed to work with operating systems that offer lightweight virtual computing nodes. The system is lightweight, modular, portable and extensible. Mesos is augmenting Kubernetes and expected to support Kubernetes API.

Kubernetes is an open-ended system and its design allowed a number of other systems to be build atop Kubernetes. The same APIs used by its control plane are also available to developers and users who can write their own controllers, schedulers, etc., if they choose so. The system does not limit the type of applications, does not restrict the set of runtimes or languages supported and allows users to choose the logging, monitoring, and alerting systems of their choice.

Kubernetes does not provide built-in services including message buses, data-processing frameworks, databases, or cluster storage systems and does not require a comprehensive application configuration language. It provides deployment, scaling, load balancing, logging, and monitoring. Kubernetes is not monolithic, and default solutions are optional and pluggable.

Kubernetes organization. A master server manages a Kubernetes cluster and provides services to manage the workload and to support communication for a large number of relatively unsophisticated *minions* servers that do the actual work. *Etcd* is a lightweight, distributed key-value store to share configuration data with the cluster nodes. The master also provides an API service; an HTTP/JSON API is used for service discovery. The Kubernetes scheduler tracks resources available and those allocated to the workloads on each host.

[22] *runC* is an implementation of the Open Containers Runtime specification and the default executor bundled with Docker Engine. Its open specification allows developers to specify different executors without any changes to Docker itself.

[23] *Go* or *Golang* is open source compiled, statically typed language like Algol and C, has garbage collection, limited structural typing, memory safety features, and CSP-style concurrent programming.

Minions use a *docker* service to run encapsulated application containers. A *kuberlet* service allows minions to communicate with the master server and with the *etcd* store to get configuration details and update the state. A proxy service of the minions interacts with the containers and provides a primitive load balance.

In Kubernetes, *pods* are groups of containers scheduled onto the same host and serve as units of scheduling, deployment, horizontal scaling, and replication. Pods share fate and share resources such as storage volumes. The *run* command is used to create a single container pod and the *Deployment* which monitors that pod. All applications in a pod share a network namespace, including the IP address and the port space thus can communicate with each other using *localhost*.

Pods manage co-located support software including: content management systems, controllers, managers, configurators, updaters, logging and monitoring adapters, and event publishers. Pods also manage file and data loaders, local cache managers, log and checkpoint backup, compression, rotation, snapshotting, data change watchers, log trailers, proxies, bridges, and adapters.

Arguably, pods are preferable to running multiple applications in a single Docker container for several reasons: (i) efficiency—containers can be lightweight as the infrastructure takes on more responsibility; (ii) transparency and user convenience—containers within the pod are visible to the infrastructure and allow it to provide process management, resource monitoring, and other services; (iii) users do not need to run their own process managers, or deal with signal and exit-code propagation; and (iv) decoupling software dependencies—individual containers may be versioned, rebuilt, and redeployed independently.

Kubernetes *replication controllers*—handle the lifecycle of containers. The pods maintained by a replication controller are automatically replaced if they fail, get deleted, or are terminated. A replication controller supervises multiple pods across multiple nodes. *Labels* provide the means to find and query containers and *services* identify a set of containers performing a common function.

Kubernetes has different CLI, API, and YAML[24] definitions than Docker and has a steep learning curve. Kubernetes setup is more complicated than Docker Swarm and its installation differs for different operating systems and service providers.

4.15 Further readings

There is little doubt that Amazon has a unique position in the cloud computing world. There is a wealth of information on how to use AWS services on the AWS web site, but little has been published about the algorithms and the mechanisms for resource allocation and the software used by AWS. The effort to maintain a shroud of secrecy probably reflects the desire to maintain Amazon's advantage over its competitors. There are only a few papers, including [255,530], describing research results related to Microsoft's Azur cloud platform.

In stark contrast to Amazon and Microsoft, Google's research teams publish often and have made a significant contribution to understanding the challenges posed by very large systems. The evolution of ideas and Google's perspective in cloud computing is presented in [131]. An early discussion of Google

[24] CLI stands for Command Line Interface and provides the means for a user to interact with a program; YAML is a human-readable data serialization language.

cluster architecture is presented in [50]. The current hardware infrastructure and the Warehouse Scale Computers are analyzed in a 2013 book [52] and in a chapter of a classical computer architecture book [232]. A very interesting analysis of WSC performance is due to a team including Google researchers [265]. The discussion of multicores best suited for typical Google workloads is presented in [242].

There is a wealth of information regarding cluster management and the systems developed at Google: Borg [495], Omega [442], Quasar [136], Heracles [312], and Kubernetes [80]. Controlling latency is discussed in [130]. Performance analysis of large-scale systems using Google trace data is reported in [412].

Important research results related to cloud computing have been reported at UC Berkeley, where Mesos was designed [240], Stanford [134,136,311,312], and Harvard [265]. Cloud energy consumption is analyzed in many publications, including [9,30,45,51,52,331,494,499].

A very positive development is the dominance of open software. Open software is available from Apache, Linux Foundation, and others. Docker software and tutorials can be downloaded from https://www.docker.com/ and https://www.digitalocean.com/community/tags/docker?type=tutorials and http://prakhar.me/docker-curriculum/, respectively.

Detailed information about Kubernetes is at http://kubernetes.io/ and https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes. The Open Containers Initiative of the Linux Foundation has developed technologies for container-based applications, see https://www.opencontainers.org/news/news/2016/04/docker-111-first-runtime-built-containerd-and-based-oci-technology. A distributed main memory processing system is presented in [263].

4.16 Exercises and problems

Problem 1. The average CPU utilization is an important measure of performance for the cloud infrastructure. [52] reports a median CPU utilization in the 40–70% range, while [265] reports a median utilization around 10% consistent with utilization reported for the CloudSuite [170]. Read the three references. Discuss the results in [265] and explain the relation between CPU utilization and memory bandwidth and latency.

1. Discuss the effects of cycle stalls and of the ILP (Instruction Level Parallelism) on processor utilization. Analyze the data on cycle stalls and ILP reported in [265].
2. Identify the reasons for the high ratio of cache stalls and the low ILP for data-intensive WSC workloads reported in [265].
3. Why do WSC workloads exhibit the high ratio of cache stalls and the low ILP?
4. What conclusions regarding memory bandwidth and latency can be drawn from the results reported in [265]? Justify your answers.

Problem 2. Discuss the results regarding simultaneous multithreading (SMT) reported in [265].

1. For what type of workloads is SMT most effective? Explain your answer.
2. The efficacy of SMT can be estimated by comparing specific per-hyperthread performance counters with ones aggregated on a per-core basis. This is very different from measuring the speedup that a single application experiences from SMT. Why?
3. Why is it difficult to measure the SMT efficiency in a cloud environment?

Problem 3. Mesos is a cluster management system designed to be robust and tolerant to failure. Read [240] and answer the following questions:

1. What are the specific means to achieve these design goals?
2. It is critical to make the Mesos *master* fault-tolerant because all frameworks depend on it. What are the special precautions to make the *master* fault-tolerant?

Problem 4. Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters, each with up to tens of thousands of machines. Read [495] and answer the following questions:

1. Does Borg have an admission control policy? If so, describes the admission control mechanism.
2. What are the elements that make the Borg scheduler scalable?
3. How does the job mix on Borg cells affect the CPI (Cycles per Instruction)?

Problem 5. Omega is a scalable cluster management system based on a parallel scheduler architecture built around shared state, using lock-free optimistic concurrency control. Read [442] and answer the following questions:

1. What scheduler performance metrics are used for Omega, why is each one of them relevant, and how it is actually measured?
2. Trace-driven simulation was used to gain insights into the system. What are the benefits of trace-driven simulation, and how was it used to investigate conflicts?
3. One of the simulation results refers to gang scheduling. What is gang scheduling, and why it is beneficial for MapReduce applications?

Problem 6. Quasar is a QoS-aware cluster management system using a fast classification techniques to determine the impact of different resource allocations and assignments on workload performance.

1. Read [135] to understand the relationship between the Netflix challenge and the cluster resource allocation problem.
2. Quasar [136] classifies resource allocation for scale up, scale out, heterogeneity, and interference. Why are these classification criteria important, and how are they applied?
3. What are stragglers, and how does Quasar deal with them?

Problem 7. Cluster management systems must perform well for a mix of applications and deliver the performance promised by the SLOs for each workload. Resource isolation is critical for achieving strict SLOs.

1. What are the mechanisms used by Heracles [312] for mitigating interference?
2. Discuss the results related to latency of Latency Sensitive workload (LS) from Heracles experiments.
3. Discuss the results related to Effective Machine Utilization (EMU) from Heracles experiments.

Problem 8. Effective cloud resource management requires understanding the interaction between the workloads and the cloud infrastructure. An analysis of both sides of this equation uses a trove of trace data provided by Google. This analysis is reported in [412].

1. What conclusions can you draw from the analysis of the trace data regarding the schedulers used for cluster management?
2. What are the reasons for scheduler behavior revealed by the trace analysis?
3. What characteristics of the Google workloads are most notable?

Problem 9. Tachyon is a distributed file system enabling reliable data sharing at memory speed across cluster computing frameworks. Read [302] and answer the following questions:

1. Does the evolution of memory, storage, and networking technologies support the argument that cloud storage systems should achieve fault-tolerance without replication?
2. What is file popularity, and what is the distribution of file popularity for big data workloads?
3. How is this distribution used?
4. For what type of events is this distribution particularly important?