machines.

- Finally, the ISA of the user's virtual machine and the ISA of each of the host computers determine the extent to which the performance of the system appears identical on the different sites. When the ISA of the guest is different from the ISA of the host, the virtual machine monitor is required to translate from one ISA to the other. We have already discussed various binary translation and optimization techniques that help in doing this.

In the next two subsections we examine the basic functioning of two systems that have implemented the environment-migration concept.

10.2.2 Using a Distributed File System: The Internet Suspend/Resume Scheme

Kozuch and Satyanarayanan (2002) proposed the use of virtual machines in migrating the state of a user environment from one computer to another via a technique they called *Internet Suspend/Resume* (ISR). The name attempts to capture the notion that the system essentially allows a user to suspend operation on one machine, travel to another machine, and resume execution on the other machine by migrating the state of the first machine over the Internet (**Figure 10.8**). The authors use the VMware GSX Server described in **Chapter 8** to carry out their experiments. The ISR proposal encapsulates the state of the virtual machine, including the contents of the virtual memory and the disk; the virtual machine monitor keeps this state in a file on a distributed file system accessible also from the destination computer.

One of the longstanding problems with the migration of state from one location to another is the need to change the IP address to access the network and all references to this address in the saved environment. MobileIP technology (**Perkins 1998**) solves this problem for mobile users of the Internet. The authors of the ISR scheme suggest that migration can be made completely transparent as long as the guest operating system is configured to use only such mobile technology.

The simplest approach for resuming the environment on a new machine is for this machine to access the state saved in the distributed file system and to load it as the state of a new virtual machine. If the system configuration and the virtual machine monitor on the destination machine are the same as those of the originating machine, the loading of the file may simply involve the loading of tables and data structures in the virtual machine monitor of the destination. In order to accommodate the variety of distributed file system definitions, Kozuch and Satyanarayanan suggest that the state file be written to an abstract file system interface, with machine-dependent aspects encoded in a small block that resides with the file.

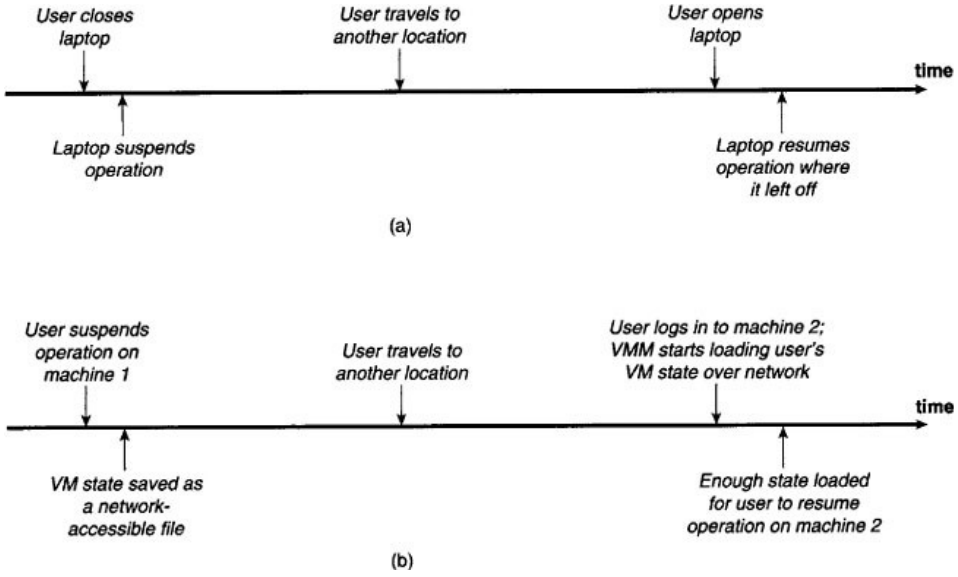


Figure 10.8 Internet Suspend/Resume. *(a) What happens on a laptop when the user travels from one place to another. (b) How a similar effect is obtained using VM migration. The state of the user's environment is saved on a distributed file system accessible to the machine in the user's destination location.*

Loading the entire state on the destination machine could take a considerably long time. In order to make the migration experience more responsive to the user, the authors propose the organization of the state information in modules that are loaded incrementally, with the environment resuming on the new virtual machine as soon as the needed elements have been loaded on the destination system, rather than waiting until the entire state has been loaded.

The ISR scheme uses a “pull” model, with the destination computer reading in the environment state file on demand, for example, when the user logs in to the destination machine. This could result in a long latency between the time the user logs in and the time when a sufficient amount of state is available on the destination to allow it to resume the environment. The authors suggest that there may be usage patterns that could be learned by a system in order to hide the latency of the resumption by proactively copying in the state to the local machine even before the user requests it. For example, the system could learn the pattern of travel between work and home for a particular user and bring in a part of the state of the system at the destination before the expected time at which the user arrives at the destination.

The amount of information that is part of the state of a virtual machine is considerably large. Copying the state, compressing it, transmitting it, uncompressing it, and then loading it back on another machine will be time consuming as well as bandwidth consuming. Techniques are needed to limit the amount of information that has to be transmitted in a burst, without compromising the ability to transmit the full state information from one host to another. For example, when a user frequently travels between two sites, the suspension of the environment at a site need not completely eliminate the environment at the site. It may be possible to keep information in some structured way so that when the environment is resumed at the site at a later time, only the changes to the environment need be loaded. Another possibility is to reuse portions of the environment from the environment of other users already on the system. Candidates for such reuse include files associated with the operating system, only a few of which are user specific.

The eventual adoption of such a scheme will depend on the security of information guaranteed by the migration system. Users would not want any information in their virtual machine to be accessible

by other users or even by the VMM. Kozuch and Satyanarayanan propose that the state saved in the distributed file system be maintained in an encrypted form. They also suggest that users be validated before being allowed use of a host machine.

10.2.3 State Encapsulation in the Stanford Collective

The Stanford Collective project (**Sapuntzakis et al. 2002**) takes a conceptually similar approach to environment migration. The principal difference from the Internet Suspend/Resume approach is that the capsule that holds the complete state of the user's virtual machine is sent to the destination computer to which the user is headed rather than to a distributed file system accessible by the destination. One advantage of this "push" model is that by appropriately planning the migration activity, the user can have instantaneous access to his or her environment at the destination site.

The virtual machine of the Collective system is an Intel IA-32 platform running on the VMware GSX server (VMware), the basic operation of which was described in **Chapter 8**. As noted in that chapter, the GSX server is a hosted virtual machine system, with the host operating system being either Linux or Microsoft Windows. On such a virtual machine system, the VMM can take full advantage of the device drivers written for the wide range and types of I/O adapters and devices that already exist for the host operating system.

The Stanford Collective developers noted that there still remain challenges in supporting certain I/O devices, e.g., network adapters, because of the portability requirement. If a virtual machine is using a virtual network card to communicate over the Internet, the encapsulation will include an IP address that may not work on a physical computer different from the physical computer producing the capsule. The Collective system enhances the virtual machine monitor to

enable tunneling of network packets travelling to and from the capsule's old network over a virtual private network (VPN).

The Collective system developers performed extensive tuning of their system to be able to suspend a virtual machine and transmit it to a destination machine in about 20 minutes (a period they considered a typical commute time) using standard 384 kbps DSL lines. The system attempts to be oblivious of the details of the guest operating system. Hence many of the techniques they employ are applicable more generally than the ISR scheme described in the last subsection. Some of these are listed next.

Reducing Memory State Before Migration

Most physical machines today contain hundreds of megabytes of memory, and it is reasonable to expect virtual machines to have main memories that are similarly large. At any given instant, however, the part of main memory that is critical for good performance is relatively small. There is typically a large part of main memory that could be paged out to disk without noticeable difference in the system's response time. Unfortunately it is not possible to determine which pages do or do not belong to the current working set at suspend time without intruding into the guest operating system. The Collective system works around this by starting a *balloon* program on the virtual machine that requests a large number of pages. This program causes pages to be released from the other active processes on the system by the guest operating system—the expectation is that these pages are released from the currently inactive processes that do not need to respond instantaneously on a resume at the destination. The size of the memory state is now reduced by zeroing out the pages requested by the balloon program, thus enabling better compaction of the capsule.

There are several subtle policy issues associated with ballooning. While transmission of the state of the memory is more effective

when more of the pages are recovered through the balloon program, the response of the system during the resume operation becomes more sluggish unless the right number and type of pages belonging to the active processes remain mapped in memory. Pages holding cached data or those that are part of dirty buffers are ideally left untouched. On the other hand, active pages that are easily compressible need not be released to the balloon program; there is less benefit in their being zeroed out for compression and possibly greater overhead in their being restored during the resume process.

Note that the balloon program is a process running within the environment of the virtual machine, though it is triggered by the VMM when it receives a suspend request. The fact that this program runs on the guest operating system allows the program to be specialized for the characteristics of the operating system. Operating systems such as Microsoft Windows limit the size of the maximum working set for each process—the balloon program cannot recover more pages than the specified maximum number of pages that may be used by the balloon program itself. On the other hand, Linux does not set a limit to the size of the working set for a process—the Collective system monitors the swap space, while the balloon program keeps allocating and zeroing out new pages, until the free swap space reduces by a preset maximum.

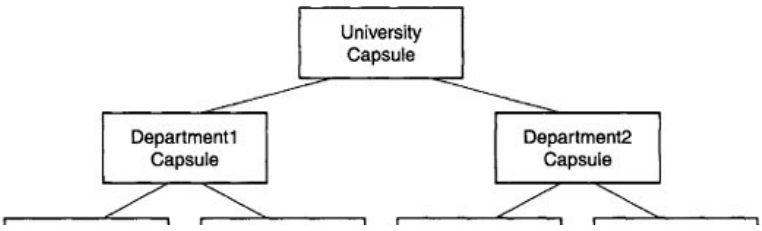
Reducing the Size of the Transmitted Packet

The balloon-program approach just mentioned goes a long way toward reducing the bandwidth required to transmit the information associated with a capsule. However, with very large disks becoming common, the state represented by the disk is rather formidable. Moreover, disks already hold many types of data, such as pictures and video, as compressed files, and additional compression is seldom effective. On the other hand, typical user migration involves just a few platforms, and the required disk image is usually not a big change from the image that was left on a platform when last visited.

The Collective exploits this observation by maintaining differences between capsules rather than the full disk image for each capsule, thus saving a considerable amount of time and resources.

Figure 10.9, taken from **Sapuntzakis et al. (2002)**, shows a sample capsule hierarchy, where each node represents a capsule state. The state of the disk in a node is saved in its entirety only in the capsule corresponding to the root node. Each node in this path inherits a base disk image corresponding to its parent node. Only the incremental difference between the states of the parent and the child is stored in each child node. The state of the disk at any node can be derived from the disk information contained in the nodes along the path from the root down to that node.

Each incremental disk uses the *copy-on-write* concept. The idea here is that whenever a copy of a disk is needed, the entire disk contents are not copied. Rather, the image of the disk is represented by a set of pointers to unique copies of various segments of the disk. Associated with each segment is a reference counter that counts the number of pointers to that segment. When a change needs to be made and a segment must be written into, the reference counter is checked. If the count is 1, then the original copy of the segment can be written into. If not, the segment is copied, the pointer is made to point to the new copy, and its reference counter is decreased by 1. This new copy can now be modified without affecting the earlier versions. Copy-on-write allows many different versions to share a single copy of the actual data; this is particularly effective when these versions are largely the same and differ in only some small aspects.



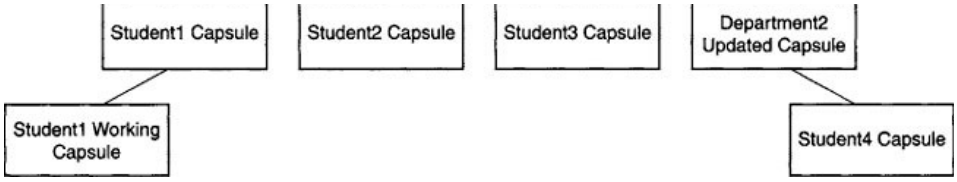


Figure 10.9 Example Capsule Hierarchy for a University. *The root capsule contains all the software that will be needed by all students. Each department customizes the capsule to suit its requirements. The department administrator does this by deriving a child capsule from the parent capsule. As mentioned in the text, only changes to the university capsule are saved in each department capsule. Students derive capsules from the department capsule, perhaps having one capsule per course in a department. Alternatively, like Student1 depicted earlier, students may derive their own private capsule. The leaf node, e.g. Student1 Working Capsule, is what migrates when the student moves from place to place.*

Before changes are made to the disk image, all nodes in the path from the root to the leaf corresponding to the image have to be transferred. Modifications can be made only to leaf nodes in a hierarchy. For example, in **Figure 10.9**, if Student4 wishes to make a change to his or her disk on a location that does not already have any of the information corresponding to nodes in the path from the root up to the leaf node, this information is brought over. A new child node is created with Student4 as the parent, and changes are made strictly to the incremental disk at this node. When the student moves to another computer, the currently open incremental disk image is transferred to the new computer. The images corresponding to the other nodes are essentially read-only images and need not be deleted from the computer from which execution was suspended. As students move from computer to computer, various nodes in the hierarchy remain behind, increasing the probability of finding a copy of a desired node and decreasing the incremental traffic that will be incurred when the user moves from place to place.

Reducing Start-Up Time on a Resume

On a resume operation, a lot of time can be saved if the information in the capsule disks is fetched on demand rather than being prefetched completely, because the working set typically needed by a user in a session is small. On the Collective system this is implemented as follows. All accesses to disks are intercepted by the VMM

and forwarded to a disk server. The disk server translates the request to an access in one of the nodes in the hierarchy. If it is a write, the access must be to the leaf node, as mentioned earlier. An access to information residing in one of the intermediate nodes may need a remote access to the node that owns the information. On such an access, local shadow copies of the desired blocks are made from the remote incremental disk image. Thus the desired blocks of the remote disks are incrementally built up on demand on the local system. Note that disk blocks in a node can be shared by all capsules derived from this node. In **Figure 10.9**, for example, if some blocks of the departmental copy are brought over on demand, they need not be brought over again when another student attempts to access the same blocks.

Reducing Transmission Time and Bandwidth by Exploiting Redundancy in Disk Blocks

It is not uncommon to find disk blocks that are identical, because the nature of activity on most systems tends to promote similarity among data blocks. Here are some sample situations when a needed block is already available on the system.

- A user moving back and forth between two systems leaves behind a trail of disk blocks that can potentially be reused the next time the site is visited.
- Blocks of program files are often already resident in memory. Most programs do not modify their own code, and hence it is generally possible to copy these disk blocks directly from memory.
- The memory of a machine often contains disk blocks other than program files, for example, blocks that are part of a disk cache.
- Often different users (and hence different capsules) utilize the same programs or data files. If a copy of a block already present in the capsule of a different user could be employed, it eliminates the need to transmit the copy from a remote site.

The Collective makes use of the *hashed copy* scheme. Each block on disk is associated with a hash value that aims to identify uniquely the contents of the block. The hash scheme employed in the

Collective is a strong cryptographic hash, SHA-1, which has a very low probability of a collision (**NIST 2002**). Instead of transferring the actual data blocks between computers, the Collective transmits the computed hash values of the data blocks. When a disk block is needed by a computer, it checks to see whether a block with the same hash value is available locally. It then uses this block directly or generates a copy of the block if the block is to be written into (copy-on-write). This avoids the need to transmit the block from a remote location. If the block is not available locally, the computer broadcasts a request for a copy. Other providers on the system receive this request and check against a table of hash values of already-available blocks and satisfy the request. The authors claim that the scheme is reliable because the probability of a collision, i.e., of having the same hash value for different data, is less than the probability that a requestor of a disk block receives a bad block because of an error in a TCP connection or an error in memory. The scheme is also claimed to be secure because SHA-1 makes it practically impossible to manufacture data that produce a given hash value.

Results of experiments (see **Figure 10.10**) using the Collective system (**Sapuntzakis et al. 2002**) indicate that these optimizations significantly reduce the traffic and hence the time for migrating a user environment from one machine to another on a typical DSL network. The authors simulated the migration of a user between home and work machines using snapshots based on the Business Winstone 2001 benchmark. The various techniques described earlier appeared to reduce the amount of data transferred to typically around 50MB, which can then be reduced further to less than 22MB through compression using the gzip program. Hashing reduces the transferred amounts even more. Needless to say, there were situations, for example, at the beginning of program execution sessions, where the raw amount of data needed to be transferred was as high as 500MB. However, even in these cases, compression brought down the size by a factor of 2 or more, with hashing providing further re-

ductions. The authors conclude that the experiments indicate that in typical situations of commuting between home and work, the entire environment could be transmitted during the commute. This makes it possible for a user to suspend execution at one site and resume execution at the other without noticing any change in the computing environment.

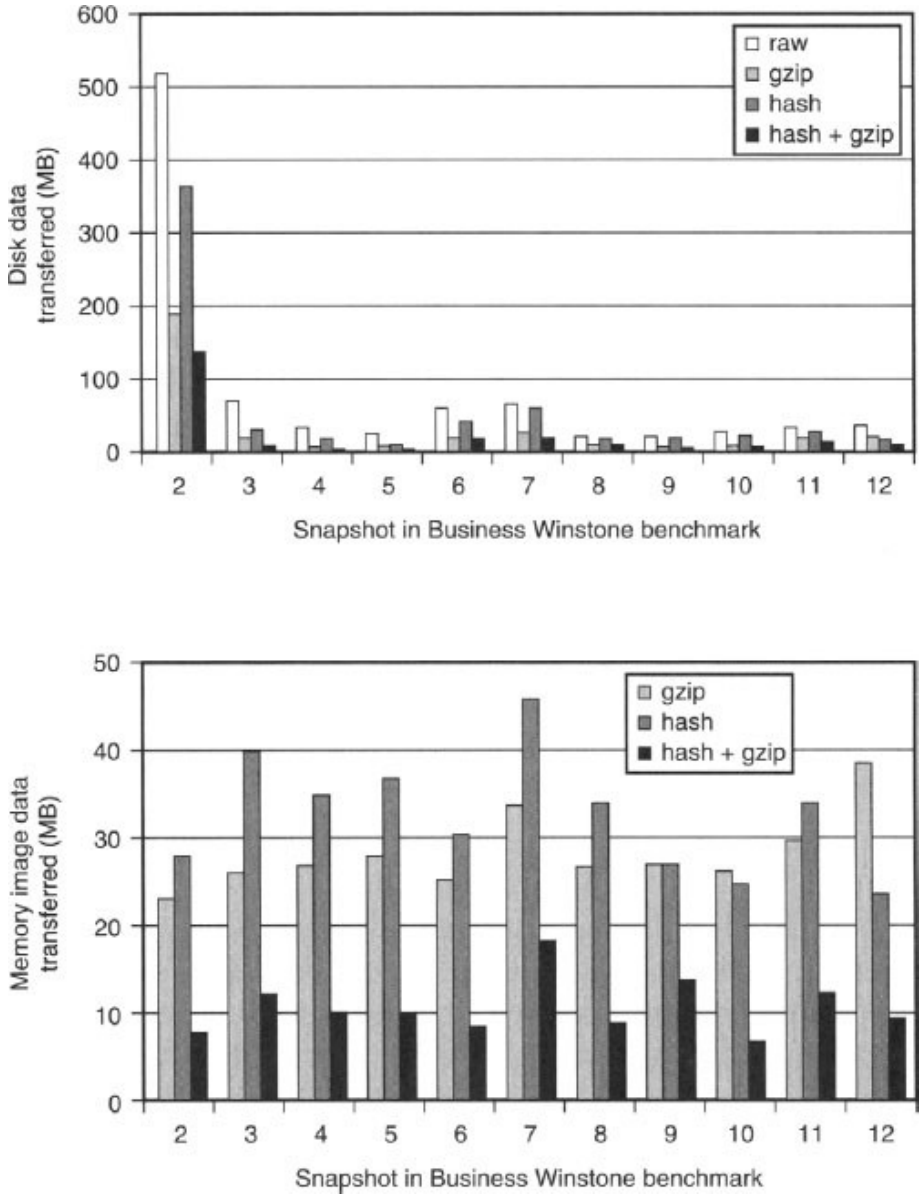


Figure 10.10 Performance of Optimization Techniques Used in the Collective (Sapuntzakis et al. 2002). *The raw memory size in the experiments is 256MB. Snapshots were taken while running the Winstone benchmark every minute after the first three minutes. The migration of each snapshot was simulated assuming the destination had all information up to the previous snapshot. The graphs show the effectiveness of the hashing scheme combined with compression using gzip in reducing the traffic from the originating host to the destination host.*

situation need.

10.2.4 Migration of Virtual Machines in VMotion

The Internet Suspend/Resume scheme and the Collective scheme are research projects, and it may be a while before all the issues involving the migration of virtual machines over the Internet are sorted out. Meanwhile, there are aspects of such migration that already exist in VMotion, developed by VMware.

VMotion is part of the VirtualCenter (VirtualCenter) infrastructure management software that manages a cluster of Intel IA-32 virtual machine systems connected in a local-area network, as shown in Figure 10.11. The management functions of VirtualCenter include the deployment and monitoring of virtual machines running on a native virtual machine system called the ESX server. Under certain circumstances VirtualCenter may need to migrate a running virtual machine from one host in the cluster to another. Examples of such circumstances include:

- **Load balancing,** to improve the response time of the system through better utilization of resources
- **Security,** to quarantine a virtual machine that has been attacked
- **Collocation,** to bring communicating virtual machines closer together
- **Fault tolerance,** to move a failing host to another processor
- **Power management,** to move the load away from an overheated processor
- **Maintenance,** to move the load away from some processor while it is upgraded

The migration of a virtual machine from one node to another in VirtualCenter is done using VMotion. This migration involves copying the state of the virtual machine, including the processor state, the memory state, and the state of other virtual hardware resources, such as BIOS, devices, MAC address for Ethernet cards, and chip set states. The issues here are quite similar to the issues we discussed in connection with the ISR and Collective schemes. However, VMotion, being a product, has to grow cautiously—it is currently restricted in

its capabilities in several respects.

- The source and destination computers must be in the same server cluster managed by the same VirtualCenter manager.
- The file systems on the source and destination computers must be identical and located on shared disks in a storage-area network (SAN). This avoids the complexities of migrating disks encountered by the Collective.
- The processors running on the two computers must have the same architecture and be provided by the same vendor. This avoids potential incompatibilities that could prevent the state saved on one machine from being directly loaded into the other machine.

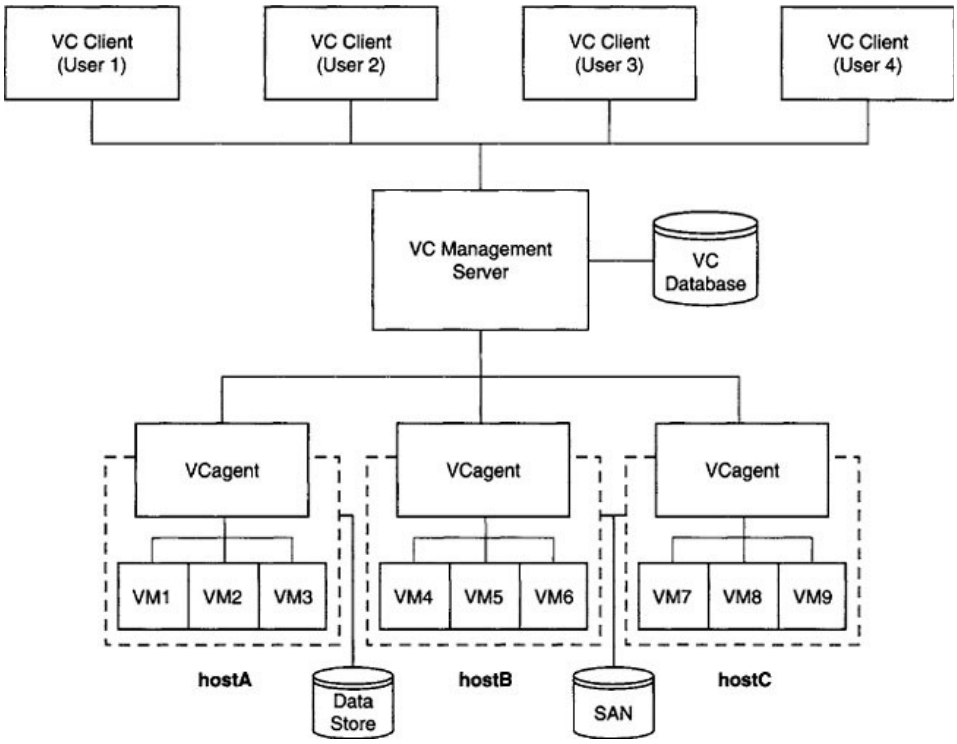


Figure 10.11 Overview of the VMware VirtualCenter. *VC stands for VirtualCenter. A cluster of three nodes is shown, with the nodes named hostB and hostC sharing a file system on a storage-area network (SAN).*

- The virtual machine supports a Gigabit Ethernet adapter.
- The virtual machines must not be running multiprocessor cluster applications; they should be running only stand-alone applications.

When a request is received to migrate a virtual machine using VMotion, software in the VirtualCenter performs the following actions.

1. It first ensures that the virtual machine is in a stable state on the current host.
2. It then copies the contents of the memory belonging to the virtual machine to the destination host. Data associated with the guest operating system as well as the data of applications running on the virtual machine are copied. This is referred to as the *baseline copy.* It is not the final copy because the virtual machine on the original host continues to run during this process.
3. The virtual machine on the original host is suspended. VirtualCenter then copies the last changes to memory along with the rest of the state of the virtual machine to the destination host. The information sent to the destination host at this stage is a capsule containing changes to the information sent earlier, similar to the incremental capsule as described in the Collective.
4. It then activates the virtual machine on the new host.

It is almost certain that more migration technology that has been developed in various research projects will continue to find its way into systems such as the VirtualCenter. What we observe is that there are many forms in which the problem of migration of virtual environments presents itself. We have reported here the early work in this area. As research progresses and as vendors get more experience dealing with products that allow migration, many new techniques will be developed. The eventual popularity and success of this paradigm may, in fact, mark the coming of age of virtual machine technology, particularly of system virtual machines.

10.3 Grids: Virtual Organizations

In the past couple of decades, the world has seen a rapid increase in the total amount of available computation power. Continual advances in lithographic techniques have allowed silicon feature sizes to shrink to incredibly small dimensions and hence have enabled the incorporation of billions of transistors on a fingernail-size die. This has been accompanied by the development of microprocessors, systems, and applications available at low cost to the common person and by the availability of easy-to-use programs that exploit the Internet. The economics of the computer industry are such that it is more profitable for companies to provide only a few types of micro-

processors and a few types of systems rather than to spend the enormous amount of money needed to develop a wide variety of microprocessors, to manufacture them at different locations, and to develop different operating systems for each type of system. The result is the wide availability of fairly high-powered general-purpose computing engines that sit on the desks of millions of people around the world or are being carried by millions of people virtually everywhere they go.

Yet the nature of tasks that most users perform on their computers is such that only a tiny fraction of the available compute power is actually used. Whether it is browsing the Internet to shop for an appliance or editing a term paper for a history class, the typical user does not exploit anywhere close to the full capabilities of the computer he or she is working on. The immediate impact of this is an environmental one—there is some energy consumed by a system as long as it is turned on—even when it is not performing any useful computation. This impact is being addressed in many ways, especially through the use of automatic power-management techniques. However, as computation and storage keep getting cheaper and as these computers get faster, there is more energy consumed by each of them. Thus, despite the incorporation of sophisticated power-management techniques, there is a sharp increase worldwide in the energy used by computers.

Ironically, there are occasions when users would like even more computation power than what they have available on their own systems. Scientists working with real-time data often do not have the processing capability to manipulate the large volumes of data they gather. They may need such processing capability either simply for visualization or for actually making decisions based on the data in real time. Their ability to perform such computation may be restricted due to the lack of compute capability, but it could also be due to the sophisticated software needed for the computation. Such users may not be willing to spend the large sums of money needed

to buy systems that give them the desired response time or to buy the relatively expensive software, because of the infrequent use they make of such systems or software. This scenario has an analog in other areas, such as transportation, where a user generally owns a vehicle that is adequate for most but not all of his or her needs; for the occasional extraordinary need, the user rents an appropriate vehicle, borrows it from an acquaintance, or takes advantage of a service provider such as a public transportation company.

Applying the vehicle rental model to the computer world seems a plausible thing to do. In fact, in the early days of personal computers, computer rentals were not uncommon. However, as the complexity of configuring software increased and as the cost of owning hardware decreased, short-term rental of computers fell out of favor.

With the emergence of ubiquitous connectivity and the Internet, rental is no longer the right model for making more efficient use of compute power. Instead, the public utility is beginning to look like a more appropriate model. Rather than own or lease the physical resources needed to carry out an occasional task, the emerging model for the future is for the user to seek these resources over a communication network and to have the task performed remotely. The computation world is, as suggested in **Foster and Kesselman (1998)**, in a state analogous to the electricity world in 1910, when local generators were the norm. Over time the electrical system evolved toward power grids, where suppliers of electricity pooled their resources for use by a much larger number of consumers. The design of the power utility system allows each user as much or as little power as he or she may need at any given time, with charges based on the amount consumed. The application of this model to the computing world would allow a user as much computation power as he or she may need to get an acceptable response time for a task, without having to own all the required resources.

The utility model for computation is already being driven by the needs of the high-end scientific computation community. The problems in understanding fundamental forces in nature, in the development of new drugs for common diseases, and in the prediction of weather all have one aspect in common—they rely on the ability to conduct large simulations involving massive amounts of computation. The physics community has long exploited large-scale collaboration across the globe ("big science") as an alternative to dividing up the available funding among smaller, less effective projects. This community has been quietly spearheading the adoption of this collaborative model for computing, by championing the utility model under the name of *grid computing*. Just as "big science" converted individual projects and laboratories into one large laboratory aimed at solving one fundamental problem after another, grid computing is bringing together many small and big computers in a heterogeneous "grid," both to make available to others unused computational resources on one person's computer, as well as to allow a bigger group of users to share a variety of specialized, often expensive, computing resources, such as supercomputers, in the manner of a utility.

About a decade ago, the World Wide Web was developed by the scientific computing community as a means for more effective collaboration among scientists to share both data and experimental results. Later, the same model proved to be useful for the commercial world and for the layperson. It is possible that models of grid computing originating with the scientific community will similarly prove to be functional to a much wider set of users in harnessing the enormous amounts of computation power already available and that will become available in the future.

Grid computing is a form of virtualization. It behaves like a system virtual machine by automatically directing a user application to a machine that has available resources and that matches the require-

ments of the application. Taking the system virtual machine concept to a higher level, grid computing creates a *virtual organization* by allowing a systematic sharing of resources between disparate systems that may be geographically distributed (**Figure 10.12**). Grid computing thus extends the virtual machine concept from instruction-level dynamic optimizers through process-level and system virtual machines to what we will call *organization-level virtual machines.* Unlike the other types of virtual machines discussed in this book, the real machine that is virtualized does not have a well-defined, formal architecture. Rather, its architecture is embodied in the procedures and protocols that govern the relationships between the collaborators. Many of these procedures and protocols are often nothing more than a verbal understanding between human beings. The creation of the virtual organization using a federation of computer systems will hopefully provide the impetus to understand and document the interfaces that define these relationships in a formal manner.

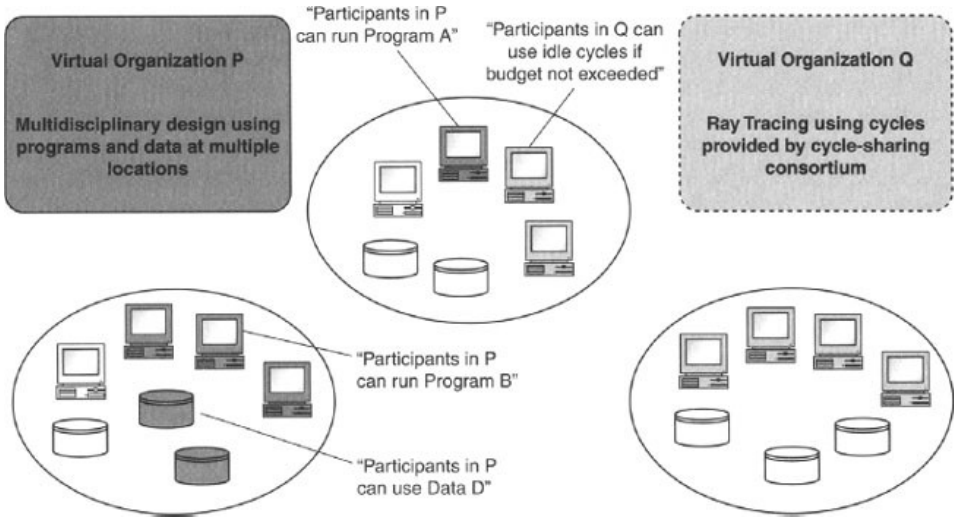


Figure 10.12 A Grid as a Set of Organizations, Both Real and Virtual (**Foster, Kesselman, and Tuecke 2001**). *Pictured are three real organizations (ovals) and two virtual organizations. Virtual Organization P is a collaborative project with distributed resources, as shown by the dark grey shading. Virtual Organization Q is a project that exploits idle cycles in resources on the grid (shown in light grey shading) to perform ray tracing. Sample policies that govern access to resources are shown in quotes.*

The development of such interfaces will facilitate the deployment of new services and allow the grid concept to be exploited by a

broader community. Thus, rather than the virtual organization being an attempt to faithfully reproduce the functioning of a real organization, the virtual organization enables the creation of function that did not exist before. The enablement of new function is a property of all virtual machines, as we have seen throughout this book.

10.3.1 Characteristics of an Ideal Grid

In this section we attempt to describe the salient characteristics of a grid. These characteristics are a combination of what exists on grids today and a wish list of features necessary for grid computing eventually to become a utility. The discussion here is an adaptation of the discussion of the computational grid described in **Foster and Kesselman (1998)** and in **Foster, Kesselman, and Tuecke (2001)**. In the former, the authors describe a computational grid as a software and hardware infrastructure that provides dependable, consistent, inexpensive, and pervasive access to high-end computational capabilities. In their subsequent paper, Foster and his colleagues extended this definition, suggesting that a grid is an infrastructure that enables flexible, secure, and coordinated resource sharing among dynamic collections of individuals, institutions, and resources. These latter characteristics led them to refer to the grid as a *virtual organization.* We will now examine each of the characteristics of a grid in more detail.

Infrastructure

A computing grid is fundamentally a hardware and software infrastructure that enables sharing of resources. Hardware infrastructure components include computational resources, storage resources, sensors, and instruments. Software infrastructure components include programs to monitor use of resources, programs to schedule resources to requestors, and programs to turn resources on and off.

A computational resource on a grid may be a workstation, a mainframe, a supercomputer, or virtually any other type of computer. A shared storage resource may be a part of a disk on a personal workstation, a portion of a file system, or an entire network-attached storage. One important characteristic of a grid is that the systems comprising a grid need not be homogeneous—there can be a variety of processors and a variety of operating systems running on the grid.

Dependability of Service

A desirable characteristic of a grid that follows from its utility nature is that it be dependable. A true grid utility must be designed in such a way that all users of the grid either have equal access to its resources or can be guaranteed a quality of service specified in a contract with the service provider. This is sharply in contrast with most existing grids, which usually comprise a loose collection of resources provided by willing individuals or collaborating groups with a loose understanding of what is provided and what is expected. A grid participant who has volunteered the idle cycles on a workstation may not have any obligation today to provide the cycles that are pledged. As the concept of a grid evolves to that of a service, users will start to depend on the services of the grid for their day-to-day activities, and such informal understanding between participants will inevitably have to be replaced by more formal specifications and contracts. There are already efforts of this nature under way, especially with the establishment of the World Wide Grid Forum and the work being carried out in the Open Grid Services Architecture (OGSA).

Dependability of the grid system also implies fault tolerance—when a computational node in a system fails, the system must be able to recover from the failure and reallocate the resources in a manner transparent to the end user. Dependability also implies guarantee of security—users of the system must be guaranteed that their assets

security—users of the system must be guaranteed that their assets are not accessible to others and that others cannot maliciously prevent their access to resources.

Note that dependability and guarantee of quality of service are important in other forms of virtualization we have already seen. In the system virtual machines of **Chapter 8**, many VMs access a single complex of hardware, and it is important for the virtual machine monitor to allocate resources in a fair way among the various virtual machines. Similarly, the different forms of partitioning we encountered in **Chapter 9** must ensure that resource allocation among the partitions is done both in a fair way and in a way that maximizes the performance of the system.

Consistency of Service

The behavior of the service provided by a grid must be predictable and consistent. A grid may contain several types of machines and systems, each of which may be able to satisfy the requirements of a particular task. While it would be impossible to ensure that the response time of a task is exactly the same each time it is executed, users will expect consistent results each time a task is executed. They may also expect some reasonable bound on the response time.

Pervasive Access

An important aspect of any utility is that it be accessible from virtually anywhere. Most current grids, as well as those that are being proposed, use the Internet as their communication medium. Thus, as long as the Internet is accessible from some part of the world, a user in that part of the world should have access to all resources made available by the grid.

Inexpensive Access

A utility service becomes popular only when the cost of a task to the

end user is attractive as compared to the alternative of performing the task on resources owned by the user. Like a utility, a grid should satisfy the needs of both the large organization wishing to perform data-mining tasks on a massive database as well as the individual who would like to have a video rendered.

Coordinated Resource Sharing

All the attributes of a grid mentioned so far view the grid as a utility. A grid is envisioned to be more than just a utility, however. Users of a utility may not have much in common with each other—they form a community only in the sense that they happen to be using the same service. In its role as a virtual organization, however, the grid is expected to allow diverse groups to pool their resources and work collaboratively toward a common goal. This view of the grid requires the development of conventions for access privileges to various resources, including not only hardware but also content and programs. It also requires the development of protocols for communication between various clusters or even various grids.
Coordinated resource sharing requires the precise identification of which resources are shared, who is allowed to share these resources, and in what manner they may be shared. Moreover, the underlying infrastructure must provide the ability to discover appropriate resources that a task may need anywhere on the grid. Development of protocols that address these sharing and communication issues, called *intergrid protocols*, are key to the widespread adoption of the grid model, just as Internet protocols were key to the widespread adoption of the Internet.

Dynamic Communities

As the grid model evolves, it is likely that a single grid will support not just one community but, rather, several different communities. An example of a community is a group of researchers collaborating to design a new drug. The resources needed by this community may

be similar to those needed by another community that is collaborating on a high-energy physics problem. Both these communities should be able to coexist on the same grid. Once again the infrastructure of the grid must ensure that adequate means are provided to identify, monitor, and control the needs of each community and to isolate one community from another.

Communities need not be static. Communities may spring up for the purpose of solving a problem and dissolve soon after this goal is achieved. A user may enter a community at some point during the lifetime of the community and exit at some later point. A user may even be a member of two different communities at the same time. All these scenarios require that adequate provision be made in the architecture and the interfaces while designing a grid.

Whether or not the grid concept evolves to embrace one unified standard protocol, such as the Internet, remains to be seen. Today there are several different grids, each with its own protocols and each having its own suite of tools and management software. It is quite possible that the model that eventually emerges will be that of multiple grids, one for each community, working on a larger physical grid that pools an appropriate fraction of its resources for each of the community grids. Each community grid, with its own set of protocols, may actually be viewed as a *virtual grid,* much the way virtual machines with different operating systems coexist on a virtual machine system.

10.3.2 Evolution of the Grid Computing Model: The Globus Toolkit

A computing grid can take on several different forms, ranging from a simple network of workstations taking advantage of idle cycles on user machines to a complex commercial server that satisfies the requirements of users who may wish to contract out a single application or an entire virtual machine. It is likely that no single paradigm

tion of an entire virtual machine. It is likely that no single paradigm will emerge as the winner. Rather, the grid is likely to evolve ultimately as a service that customizes the solution to the needs of the user.

The number of grid systems being deployed is growing by leaps and bounds. Most of the current efforts are aimed at bringing together researchers in specific communities. **Table 10.2** contains different classes of users of the grid ranging from system administrators to the end user. Each of these classes has a purpose distinct from the others and it is important that any toolkit developed to facilitate the use of the grid satisfies the needs of all of these classes of users. The grid developers provide the basic services required to construct the grid. Tool developers construct the programming models and associated tools which are used by the application developers, who construct grid-enabled applications for the end user.

Today there is an active effort to provide basic infrastructure tools to enable users to deploy and manage grids of their own. The Globus project (Globus), for example, is an open-source effort that aims to provide core services, interfaces, and protocols that enable users to create a new class of applications accessing remote resources seamlessly while allowing resources to remain under local control. The Globus project distributes a toolkit that includes software services and libraries to perform discovery, monitoring, and management of resources, to enhance security of transactions on the grid, and to perform more routine functions, such as file management. The expectation is that it will provide a substrate not only for collaboration among the science and engineering communities, but also for businesses and corporations. The effort was originally funded by DARPA but today is sponsored by several federal agencies as well as by corporations.

TABLE 10.2 Classes of Grid Users (**Foster and Kesselman 1998**)

Class	Purpose	Makes Use of	Concerns
End users	Solve problems	Applications	Transparency, performance
Application developers	Develop applications	Programming models, tools	Ease of use, performance
Tool developers	Develop tools, programming models	Grid services	Adaptivity, exposure of performance, security
Grid developers	Provide basic grid services	Local system services	Local simplicity, connectivity, security
System administrators	Manage grid resources	Management tools	Balancing local and global concerns

10.3.3 Comparison with Conventional Virtual Machines

In this section we try to highlight the similarities and differences between the concept of a grid and that of other virtual machines we have been describing in this book.

Efficient Utilization of Resources

The eventual goal for the first grid systems was the desire to put to use the idle cycles available on the hundreds of millions of computers in the world today. There are many fundamental problems that require a lot of computation power but whose solutions do not need the large, expensive supercomputers commonly employed for such problems. Harnessing the unused resources toward solving such large problems would be an efficient use of the world's computational resources. This is similar to the motivation that led to the development of system virtual machines and to multiprocessor virtual machines, including logical partitioning.

Sharing of Resources

System virtual machines for both uniprocessors and multiprocessors support sharing of resources. However, unlike grid computing, the sharing in a conventional system VM is usually limited to the physical hardware resources—the VMM in such systems is typically not concerned with the sharing of content, for example.

A central concept in grid computing is the sharing of resources of disparate systems by a community of users working toward a common goal. The discovery of resources needed to solve a problem and the negotiation for use of such resources with their owners play a big role on a grid.

Distributed Versus Centralized Control

Compared to traditional virtual machines, the grid has a global scope, with resources spread over large areas, perhaps across the world. Control of shared resources therefore cannot be done in any centralized manner. As with the Internet, users of a grid may collaborate with each other to determine appropriate sharing and use of resources. This is quite unlike a conventional virtual machine monitor, which must have a centralized view of resources and must maintain strict control over the use of resources in order not to compromise the privacy and security of users.

Heterogeneous Nodes

Like operating systems, virtual machine monitors and hypervisors generally do not span nodes of a cluster. A VMM works best when it has the flexibility to reallocate virtual machines among a set of similar real processors. The components in a node may be somewhat dissimilar in terms of their sizes and perhaps in the nature of devices attached to them, but they are similar in terms of the ISA supported by the processors.

On a grid, the expected degree of heterogeneity in the types of components available is much higher. Each cluster could have a processor, memory, and I/O configuration quite different from other clusters on the grid. Moreover, the processors in a cluster could be running a different type of operating system or even a different ISA from those in other clusters. The set of programs that can be run on one cluster may be considerably different from the set that can be

run on another. A program that performs a function on one cluster may be completely different from the program that performs the same function on another cluster. Such an environment will breed programs that may not directly specify the instructions needed to solve a problem, but will, rather, specify the function needed to solve the problem. The software infrastructure of the grid will assess the availability of various different types of nodes that could perform the function, choose the best cluster and node on which to run the function, and configure the appropriate program that must be run on that node.

Adaptation of Applications

One of the requirements imposed on virtual machines in the past was the need to run applications unchanged. The fact that the application was running on a virtual machine, as opposed to a real machine matching an expected configuration, was unknown to the application. In system virtual machines this is true not only at the application level but also at the operating system level.

This level of transparency was not manifested in early examples of the grid. A case in point is the SETI@home project (**Anderson et al. 2002**), where the solution is tailored especially to the physical characteristics of the grid. In particular, the solution is aware of the latency of communication between nodes on the grid and breaks up the problem in such a way that a given computer can operate on one of the pieces for a sufficiently long time without needing to communicate with other computers on the grid. The solution also accounts for the potential unreliability of nodes on the grid by sending each problem chunk several times to different computers.

The hope is that someday these functions can be taken care of in the infrastructure of the grid so that the solution developer will need to be concerned only with the aspects of the problem at hand rather than the physical characteristics of the platform on which the prob-

lem is being solved. There are significant challenges in reaching that point, but efforts are already under way. One example is the development of libraries that will ease the porting of applications meant to run originally on message-passing clusters or on distributed shared-memory systems or even on tightly coupled SMP systems so that they can exploit the computational resources available on a grid (**Foster and Karonis 1998**).

Portability of Applications

As noted in the discussion in **Chapter 5** on high-level language virtual machines, there is increasing emphasis on enabling applications to be portable so that they can run everywhere. Thus applications are being written with portability in mind. Targeting applications to run on virtual machines, such as the Java virtual machine, rather than on processors with specific ISAs is a step in that direction. Enabling applications for the grid is another step in that direction. Applications that can exploit not only heterogeneous system configurations but also heterogeneous ISAs are likely to be portable on a variety of platforms, possibly even those that emerge in the future. Further, the potential unreliability of components on the grid as well as the challenges of maintaining security and privacy on such an open structure could lead to the development of infrastructure for creating applications that produce the right results even under adverse circumstances.

10.3.4 Coming Full Circle: Implementing the Grid on Classic Virtual Machine Systems

The administration of the grid, as we have noted, involves several pieces of software, often referred to as *middleware,* that perform various operations, including those of allocating the available resources to tasks, scheduling the tasks, and ensuring security and privacy. Middleware supports the process level of abstraction, just as a

traditional operating system supports processes. However, the analogy between the grid and an operating system should not be carried too far. For example, like an operating system, the grid has to perform user accounting; unlike the operating system, however, the grid relies on the accounting services provided by multiple administrative domains. Legacy jobs, ordinarily running under the accounting policy of a traditional operating system, may find an inconsistency in the manner they are dealt with because different administrative domains may employ different accounting policies.

An interesting solution to this multipolicy problem, proposed in **Figueiredo, Dinda, and Fortes (2003)**, is to change the level of abstraction of jobs from the user process level to the full machine, which includes the hardware and the operating system. With this approach, the unit of work with which the grid middleware deals becomes a system virtual machine rather than a process. Accounting can now be done conveniently in a hierarchical manner—the operating system running on the virtual machine performs accounting in a traditional manner, while the grid accounts for the resources at the virtual machine level.

Utilizing the conventional system virtual machine as a unit for the grid is suitable from other points of view, too. It automatically addresses the two important aspects of a grid—isolation of jobs from each other and platform independence. Beyond these, it adds the important ingredient of environment flexibility. It is not necessary for the user application to be recast for an OS environment supported at some given node—the unit of work includes the full system environment in which the application executes. Let us examine these aspects in more detail.

- **User isolation:** As we saw in **Chapter 8**, system VMs were originally designed as an alternative to multiprogramming, with each user isolated from the physical system and from other users by running on a separate virtual machine. This feature of a system VM protects a user sharing resources with other users. This is of particular importance on a grid, where it may not be prudent to trust a provider

of physical resources. Moreover, the forced user isolation in the system VM paradigm ensures greater system integrity. Compared to a typical multiprogrammed environment, it is less likely that the actions of a user on a virtual machine will either crash the system or bring down other users. Moreover, as discussed in **Section 10.1.3**, there is less likelihood that a malicious attack on a single user will compromise the security of the system as a whole.

- **Platform independence:** The platform independence that can be provided by a system VM also helps in its use for grid computing. For example, the user simply has to specify the hardware resources needed—for the most part, it is not necessary for there to be a physical system that matches the exact configuration specified by the user. Through various emulation techniques, such as those described in **Chapter 8**, the virtual machine monitor can virtualize those devices that do not physically exist. Also, it is not necessary to recompile or relink an application, because the virtual machine monitor ensures compatibility right up to the ISA level.
- **Task management and accounting:** There is a fundamental difference in the way resource control is managed in a virtual machine—based computing grid as compared to a more conventional grid. In process-oriented grid deployments, resource control must be managed for each process, similar to that in traditional multiprogramming systems. In the virtual machine case, the resource control is at a higher level of granularity, namely, at the virtual machine level. This higher granularity can simplify the job of allocating and accounting for resources both for the provider of the resources as well as for the user of resources. Moreover, this model fits better with the prevailing model in mainframe commercial computing, where charges are based on the specified performance of the system where an application runs.
- **Portability:** Portability of applications is a key requirement for a computing grid. A process-based grid and a system VM—based grid both emphasize portability, but they differ in philosophy over how it is implemented. In a process-based grid, the responsibility to ensure portability is left to the application programmer. This may mean, for example, writing the application for an environment supported on a large number of grid nodes. Alternatively, it may mean writing the application in a high-level language, such as Java, which is then supported on multiple nodes, perhaps with different ISAs. The system VM—oriented grid approach, on the other hand, allows an application to run on any platform that has a virtual machine monitor supporting the virtual machine specified by the application. This may even extend to applications on environments with different ISAs, the best performance for an application being given, of course, on a node that matches the ISA desired by the application.

Viewed from the Java computing model, the difference between the

Viewed from the Java computing model, the difference between the process-based grid and the system VM—based grid may be attributed to the nature of virtual machines supported. A node in a process-based grid implements a specific virtual machine environment, such as Java, and expects applications to be written for it. In the system VM—based approach, each node supports a range of virtual machines, especially those expected by the more common applications. Thus the former may have some advantages for new applications currently being written, while the latter is more likely the choice for the deployment of legacy applications on the grid.

In fact, the system VM—based grid could also be the solution to a thorny problem faced when trying to migrate an HLL VM, such as a Java VM, from one platform to another. In a Java VM, for example, it is probably not difficult to encapsulate the pure Java state of the applications and the libraries. But the typical JVM also has its own implementation state in native code and may be running native primitives or even user native code, all of which are difficult to encapsulate at the JVM level. Rather than migrate the JVM, it may be easier to raise the granularity of migration and to migrate the entire system in which the JVM resides. For example, the JVM could be running on a virtual machine with Linux as the host operating system. The encapsulated state of the virtual machine will then include not only the Java state but the rest of the state of the JVM implementation along with the state of the Linux environment. Various techniques outlined in **Section 10.2**, such as ballooning, could be used to reduce the size of the encapsulated module.

Figueiredo, Dinda, and Fortes (2003) identified three important aspects of a task in an system VM—based grid. The first is the ability to capture the state of a virtual machine, the second is the ability to instantiate a virtual machine either from scratch or from its state description, and the third is the ability to save user data. They point out that these three tasks do not necessarily have to be on the same machine. For example, the state of a virtual machine can be cap-

tured at the location where it is currently executing, but it may be resumed at a machine at another location, where the image is transported. This is similar in concept to the ideas embodied in the Collective (**Sapuntzakis et al. 2002**) and Internet Suspend/Resume (**Kozuch and Satyanarayanan 2002**), discussed in **Section 10.2**; many of the optimizations mentioned in connection with those schemes are applicable in this case too. Saving user data in a distributed file system allows the state encapsulation to be small, as already discussed in connection with Internet Suspend/Resume.

One can identify three different types of servers performing these three different functions for a system VM—based grid. The application is run on a virtual machine in a *computation server* (or VM host) whose virtual machine monitor has the capability to encapsulate the state of the machine at some checkpoint and make it accessible from other locations. When the virtual machine needs to be resumed, the image of the state encapsulated earlier is provided by an *image server*. Uniform access to data from any of the computation servers on the grid is facilitated by retaining the data on a *data server*.

The tools needed to manage the grid in such a heterogeneous server system are not much different from those required for the process-based grid, and one could use essentially all the basic infrastructure tools provided for this purpose, for example, by using the Globus toolkit mentioned in **Section 10.3.2**. Such tools are critical for the success of the system VM—based grid. They facilitate the functions associated with grid management, e.g., the scheduling of virtual machines, the migration of virtual machines, and the enforcement of service-level agreements. More importantly, they also allow users to specify the requirements of their jobs and allow providers to advertise the capabilities of the virtual machines they are willing to support, to bid on user tasks, and to monitor the usage of resources on their systems.

We end this section by reproducing a simple scenario from **Figueiredo, Dinda, and Fortes (2003)**, which takes a fictitious grid (**Figure 10.13**) and illustrates ways in which many of the functions needed to instantiate a virtual machine on the grid can be performed with tools that already exist today.

1. User *X* accesses the grid and publishes his/her requirements using grid middleware on frontend server *F.* An information service is consulted that checks against the list of available physical machines matching *X*'s requirements and that can support a dynamic VM. Alternatively, the user may simply advertise his/her requirements and solicit bids from potential providers for the task.
2. *X* also consults the information service to determine an image server that could provide an image with the base operating system installation that meets the needs of the application. Alternatively, *X* may itself provide a VM image containing a customized version of some operating system.
3. A new virtual machine is instantiated at physical server *P,* using an image from image server *I.* The image can be transferred from *I* to *P* using an explicit command for transferring data, such as GridFTP (**Allcock et al. 2001**), or *P* can access the image through a distributed grid virtual file system.

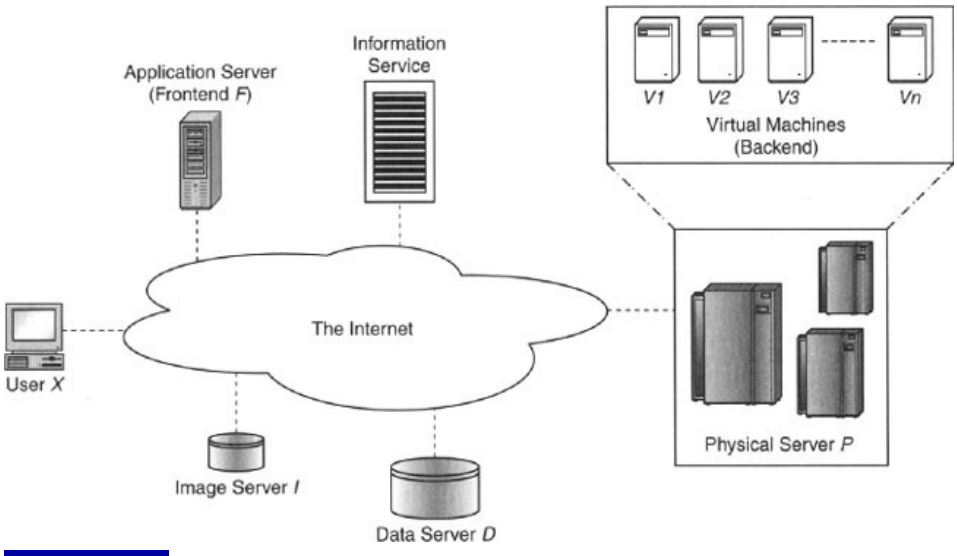


Figure 10.13 Example to Demonstrate Some of the Elements of a VM-Based Grid Service (Adapted from **Figueiredo, Dinda, and Fortes 2003**). *User X avails the services of frontend F and gets allocated a virtual machine Vi on physical server P. The virtual machine gets its image from image server I. The application gets its data from data server D.*

4. The location of physical server *P* is made known to the user, who then utilizes a secure shell, e.g., OpenSSH (OpenSSH) or Globus GRAM (**Czajkowski et al. 1998**), to negotiate the start-up of virtual machine *Vi.* This virtual machine may

1998), to negotiate the start-up of virtual machine *Vi*. This virtual machine may be a new one that is booted up, or it may be an existing virtual machine, which is just being resumed from a saved image. The virtual machine instance *Vi* may also be assigned a dynamic IP address at this time, for example, using DHCP (DHCP). An interesting technique has recently been developed (**Sunderaraj and Dinda 2004**) that allows a virtual machine to keep its IP address even when it migrates from host to host.

5. Once VM instance *Vi* is running and connected to the network, additional data sessions may be established to connect the guest operating system within *Vi* to application server *A* and to user data server *D*. As before, these sessions can be realized with transfers, such as GridFTP, or implicit transfers through a distributed file system.
6. The application now begins executing under the guest operating system in virtual machine *Vi*. If it is a stand-alone batch application, the application runs unattended and may inform the user only at completion. If it is an interactive application, the user interacts with the application either through a login session window or through a virtual display, such as VNC (**Richardson et al. 1998**).

In the foregoing, it was assumed that the user needs to interact only with the application. A more sophisticated user may be presented with a console for the operating system or even for the virtual machine itself. The user, rather than the grid scheduler, can then make decisions about when to shut down, hibernate, restore, or migrate the virtual machine. The mechanisms needed to perform these tasks are similar to those mentioned earlier, namely, file transfer mechanisms for efficient transfer between machines, distributed virtual file systems, and virtual networks.

The virtual machine, like a real machine, may persist even after completion of the application that initiated it. If it remains inactive for a sufficiently long period of time, the grid management system can place it in hibernation, saving its image on the shared file system, or in some other globally accessible space. When its services are required, the virtual machine is reawakened at any one of several possible computation servers, and appropriate changes are made to the handles used to communicate with the image servers and user display sessions that may be open. The virtual machine session ends only when there is no image of the virtual machine

present on any of the image servers or in permanent storage anywhere on the system.

The importance of executing legacy applications should not be underestimated. The world is full of such applications, critical for businesses, either because it is difficult to migrate the applications to newer platforms or because the environments on which they run provide a capability not matched by the newer platforms. The high reliability and availability of the original IBM mainframes has been maintained through its evolution into the zSeries, which remains the workhorse of large commercial organizations. The VM-based grid computing system aims to serve a similar role for critical applications being designed today on a variety of platforms. Moreover, the system VM—based model has inherently attractive features, even for the deployment of new applications, as observed earlier. The techniques described in earlier chapters to ensure the correctness and efficiency of virtual machines may eventually find use in such grid implementations.

10.3.5 Concluding Remarks

Many of the scenarios mentioned in this section are also relevant for businesses, especially large corporations. Business-specific research and development often involve the use of massive amounts of data, large groups of internal and external collaborators distributed across a geographically wide region, and massive computational resources to perform analysis and mining spread across the globe. There is a feeling in some quarters that just as the Internet developed from a medium for collaboration among scientific researchers to a powerful medium for everyone, particularly for businesses, the grid concept is about to take off and become a vehicle that transcends the scientific community and embraces the business community and the world at large.

10.4 Summary

10.7 Summary

In the earlier chapters of this book we described various types of virtual machines that have been implemented over the years. VLSI process technology progressed so rapidly in the last couple of decades that processing power became relatively inexpensive and this may have caused some of the classic types of virtual machines to temporarily fall out of favor. This chapter suggests that new problems and new organizations for information processing are likely to cause resurgence in interest in such virtual machines. We focused our attention in this chapter on three applications. Security concerns in modern computing systems of all kinds are making headlines almost every day. The potential of virtual machines in protecting systems, both large mainframes and small portable devices, cannot be ignored. We have demonstrated examples of use of virtual machines in combating various types of security problems.

The use of virtual machine technology in the migration of whole computing environments from one hardware platform to another is proving to be of great importance in enterprise computing, where applications often run continuously for large periods of time, and where such migration may be necessitated due to hardware failures, power considerations, load balancing, or system upgrades.

We have also seen how virtualization techniques extend to the emerging concept of grid computing. Indeed, virtualization and virtual machine technology are poised for pervasive deployment in future computing systems of all types.

Previous chapter
< Chapter 9: Multiprocessor

Next chapter
Appendix A: Real >

69 of 70 12/23/2024, 2:50 PM
727

Virtualization

Machines

Appendix A

Real Machines

Many virtual machines present an interface essentially identical to that of some desired real machine. Furthermore, all virtual machines are implemented on top of some real machine; virtual resources are ultimately realized with real resources. In order to fully understand virtual machines, therefore, it is important to understand the major components of a typical computer system, including their interfaces and the corresponding resources that are managed through the interfaces. This appendix outlines the major computer system components, with emphasis on the ones most relevant to virtual machine implementations.

The discussion will begin with an overview of the three primary computer system hardware components: processors, memory, and I/O. Then we will discuss instruction set architecture (ISA) features and the ways in which the ISA can be used both to perform computation and to manage the hardware resources. Next, the organization of an operating system is described, with emphasis on managing system resources. We follow this with a discussion of the important aspects of multiprocessor systems. Finally, two specific ISAs, the PowerPC ISA and the Intel IA-32 ISA, are summarized; these two ISAs are used in examples throughout the book.

Over the years, many different computer architectures have been defined. As they have evolved, they have tended to take on similar characteristics, although there are still some significant variations among the architectures prevalent today. It is not practical to discuss all these variations here (or to cover them all in the book). We instead consider only typical architecture features that are relevant to our discussion of VMs. It is assumed that the reader has some fa-

to our discussion of VMs. It is assumed that the reader has some familiarity with operating systems and the general concept of instruction set architectures (ISAs).

A.1 Computer System Hardware

Figure A.1 illustrates a computer system typical of what one would find in a desktop system. Multiprocessor server organizations are overviewed in Section A.7. Both types of systems are composed of processor(s), memory, and I/O subsystems containing both high- and low-speed buses. The following subsections describe each of the three major components.

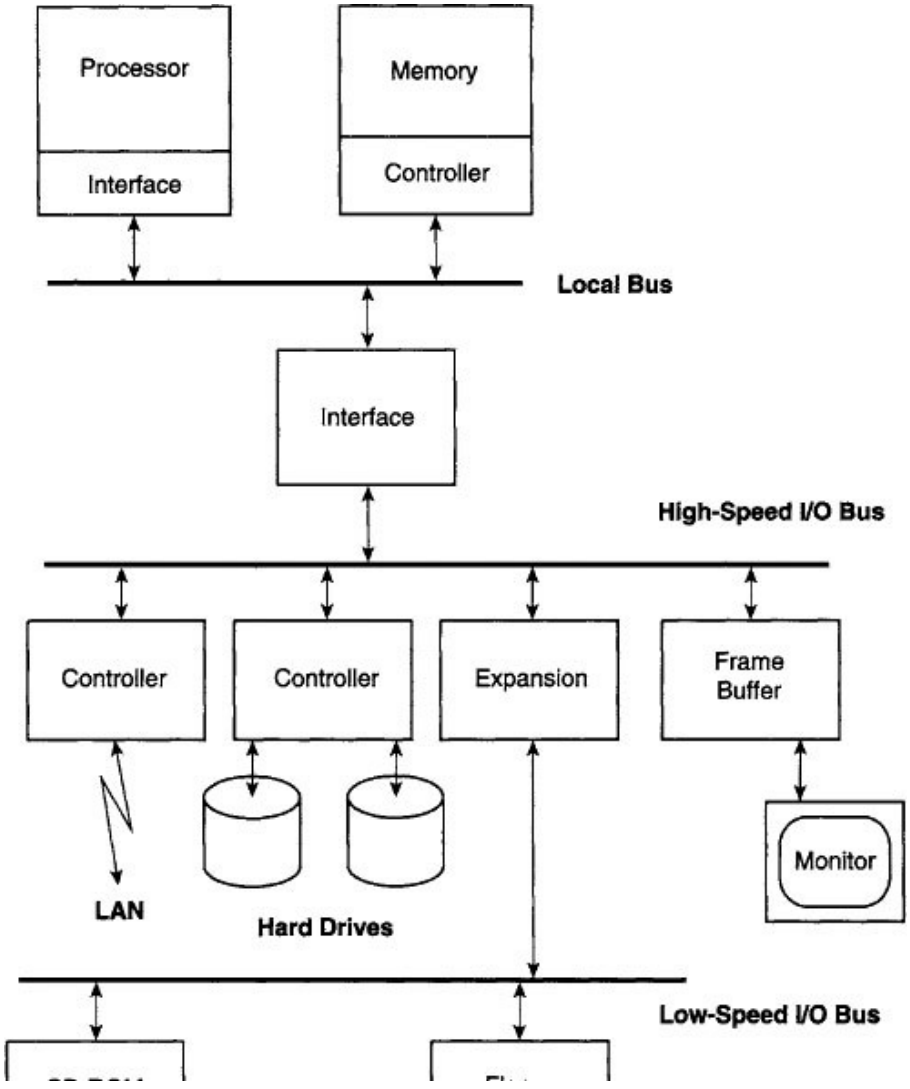




Figure A.1 System Organization.

A.1.1 Processors

Processors fetch and execute instructions held in memory. The various types of instructions and their operation are described in Section A.2. In this section, we briefly outline the more important types of processor microarchitectures.

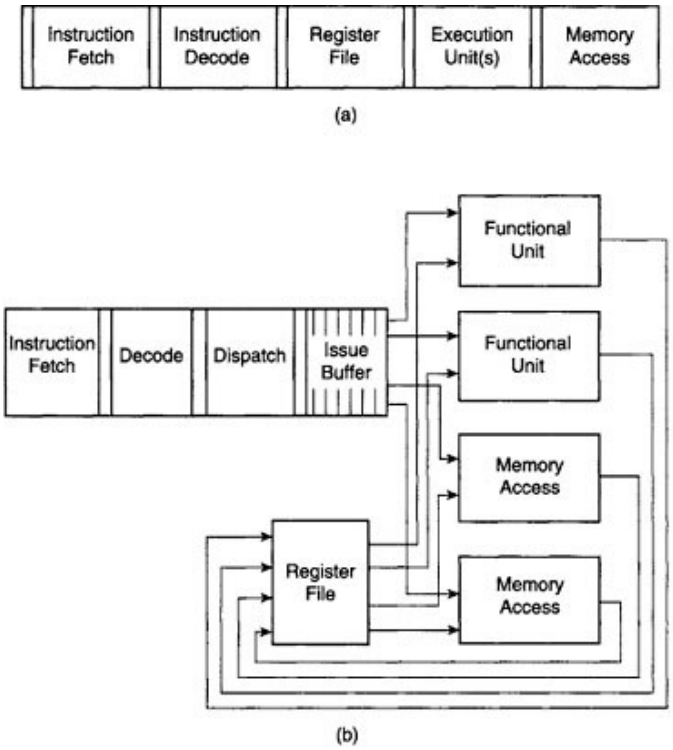
Figure A.2a illustrates a simple in-order pipeline, where instructions pass through a series of stages as they are executed. The pipeline essentially performs its operations in assembly-line fashion. This allows multiple instructions to be in the pipeline at the same time, though at most one instruction is in each pipeline stage at any given time. In Figure A.2a instructions pass through the pipeline in their natural, architected program sequence (i.e., “in order”). They are first fetched and decoded. Then, when the input operands of instructions are ready, they are read either from registers or from memory, and the instructions are executed. Finally, instruction results are written to registers or stored in memory.

Some instructions depend on the results of preceding instructions; if the results are not ready as inputs when an instruction needs them, the pipeline must “stall” the instruction until the preceding instruction produces the result. If an instruction takes several cycles to produce its results, e.g., if it loads data from memory, then a following dependent instruction may suffer a number of stall cycles. It should be apparent that the sequence in which instructions appear and their data dependences influence the number of stalls and, therefore, overall pipeline performance.

Many high-performance processors use a superscalar microarchitecture, as shown in Figure A.2b. In a superscalar processor, more than one instruction can be fetched and decoded in the same clock

cycle. Thus, the peak instruction throughput is increased when compared with the simple pipeline. After decoding, instructions are dispatched into an instruction issue buffer. From the issue buffer, instructions can issue and begin execution when their input operands are ready, without regard to the original program sequence; i.e. they can issue "out of order." This avoids many of the aforementioned pipeline stalls when instructions depend on preceding instructions.

A third class of processors, illustrated in Figure A.2c, can execute multiple instructions per cycle but only in the original compiled sequence. It is up to the compiler to arrange and combine multiple parallel (independent) instructions into a very long instruction word (VLIW). All the instructions in one VLIW issue before the next VLIW is considered for issuing. The in-order-VLIW approach simplifies the instruction issue logic, compared to an out-of-order superscalar processor, but it puts the burden of finding and reordering groups of independent instructions on the compiler rather than the hardware as in superscalar out-of-order machines.



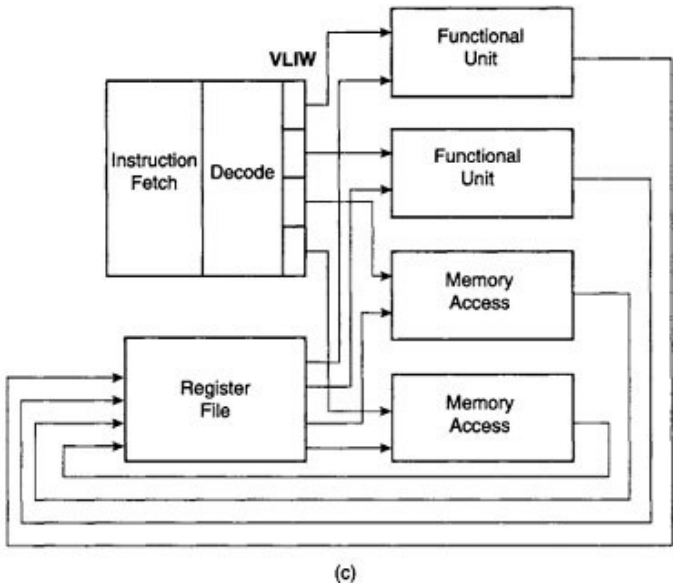


Figure A.2 Three Processor Microarchitectures. *(a) Simple pipeline; (b) superscalar processor; (c) VLIW processor.*

A.1.2 Memory Systems

The memory system of a modern computer consists of a combination of main memory and cache memories. The main memory is explicitly managed by the operating system, while cache memories are generally managed by hardware and hidden from software.

Main Memory

Memory systems are built primarily from random-access memory (RAM) chips, organized so that multiple bits can be accessed in parallel. Usually, at least a word (four bytes) is accessed, but often it is a larger quantity, i.e. a *line* of 32 to 128 bytes. In most modern processors, the smallest quantity of main memory that can be uniquely addressed is the byte, and if there are n real memory address bits, then the 2^n bytes form the *real address space.* Processor load and store instructions generate real addresses in a manner to be described in Section A.2.2. However, as seen in Figure A.3, all locations in the real address space do not necessarily correspond to RAM—some of the locations may correspond to addresses in a read-only memory

(ROM), and some others may not correspond to memory addresses at all. Of the addresses that do not correspond to memory locations, some may be reserved for I/O devices, as we will see later, and some may be unused, with no memory chips populating that region of the address space.

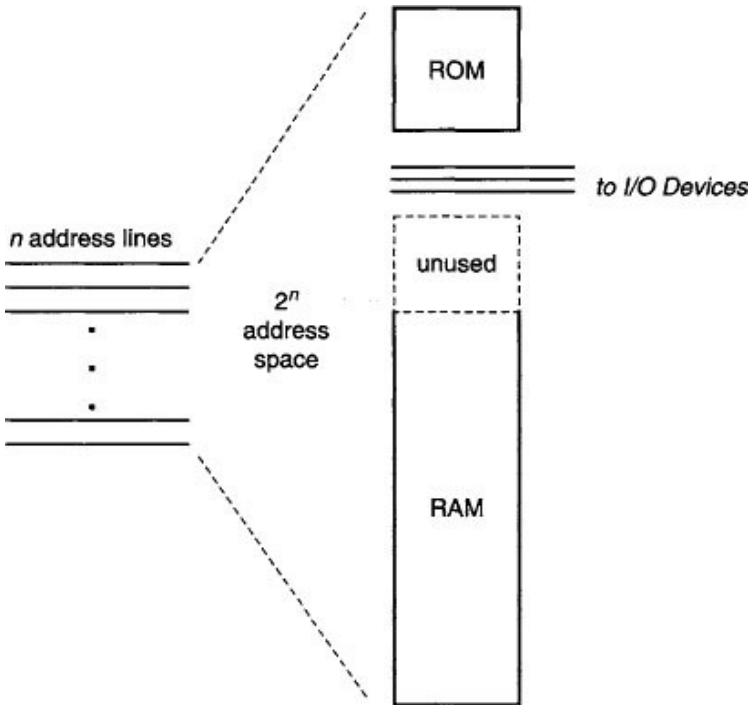


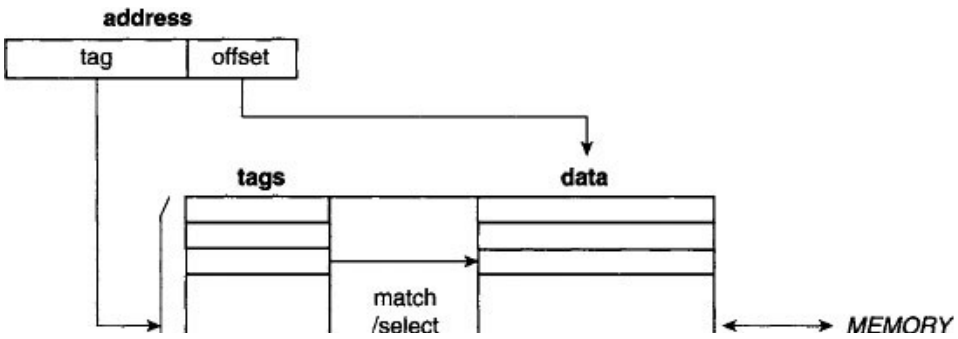
Figure A.3 Real Address Space. *A Real address space is often used for addressing RAM or ROM. Parts of the real address space may also be used for addressing I/O devices, or they may be unused.*

Cache Memories

The amount of main memory needed in modern processors is quite large, and, including the delay in the memory controller and buses, accessing a location in such large memories takes many tens or even a few hundreds of processor cycles. In order to reduce the observed latency of accessing main memory, smaller, faster memories, called *cache memories*, are employed for holding both instructions and data likely to be accessed in the near future. Cache memories are designed around the principle of *locality*. This principle states that data or instructions that have been used recently are likely to be used again in the near future (*temporal locality*) or that data or in-

structions in locations close to locations currently being accessed are likely to be accessed themselves in the near future (*spatial locality*).

A cache memory holds *blocks* (or *lines*) of data and instructions that correspond to recently accessed main memory locations. At any given time, the cache memory holds a small subset of all main memory lines. In order to determine which main memory lines are located in a cache, the cache is designed to be *associatively* accessed—each cache line has an associated tag that indicates the main memory locations (addresses) from which the cache line came. Figure A.4 illustrates a *fully associative* cache, which is one of the easier cache implementations to understand (although not necessarily the easiest to build). In the fully associative cache, a memory address is first compared with all the cache tags to see if the contents of the addressed memory location are present in the cache. If there is a *cache hit,* the data or instructions are read from the cache and are immediately available to the processor. If there is a cache miss, then the contents of the line must be fetched from memory and placed in the cache. To make room for the new line, some other line must be replaced, according to a *replacement algorithm.* Two commonly used replacement policies are least recently used (LRU) and first-in-first-out (FIFO). Temporal locality is exploited by using such a replacement algorithm, while spatial locality is exploited by the fact that a cache line typically holds more data items (e.g., words or bytes) than the one requested; that is, it contains data items that are in a block of contiguous locations adjoining or surrounding the accessed item.



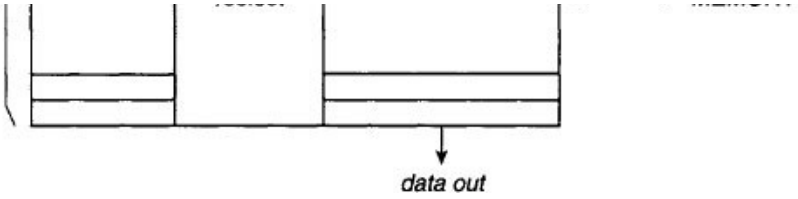


Figure A.4 A Fully Associative Cache Memory.

A.1.3 Input/Output Systems

In Figure A.1, we saw a typical system with its complement of I/O devices along with a variety of interfaces to these devices. This is typical of desktop systems. In comparison, server systems typically have fewer types of devices, although there may be more devices of certain types (e.g., disks), and the devices in such systems are connected using interconnections of much higher bandwidth.

An I/O system consists of a number of buses that connect the processor and memory to the I/O devices. These I/O buses are often standardized, so third-party vendors can manufacture new devices that immediately attach to the system. An example is the PCI bus in desktop systems. Most devices interface to the bus through a *controller,* and typical devices include disks, tapes (which are becoming less common), monitors, keyboards, etc. The bus serves as the conduit through which devices (or their controllers) are addressed and given commands and through which data is transferred between the processor or memory and the I/O devices.

There are four ways in which I/O is organized on a typical system. These are illustrated in Figure A.5. Each organization requires the OS to invoke an I/O operation in a distinct way.

- In *programmed I/O,* shown in Figure A.5a, the OS issues an I/O request over the I/O bus and then polls the device controller until the request is satisfied. In general-purpose systems, this method is not commonly used because it ties up the processor while the I/O operation is being performed.
- The second type of I/O is *interrupt-driven I/O,* shown in Figure A.5b, in which the processor continues with some other task after issuing an I/O request. An

interrupt from the I/O controller informs the OS about the status of the request, e.g., when it is finished. The granularity of control is still quite small: Every unit of data transferred between the controller and memory is controlled by the OS.

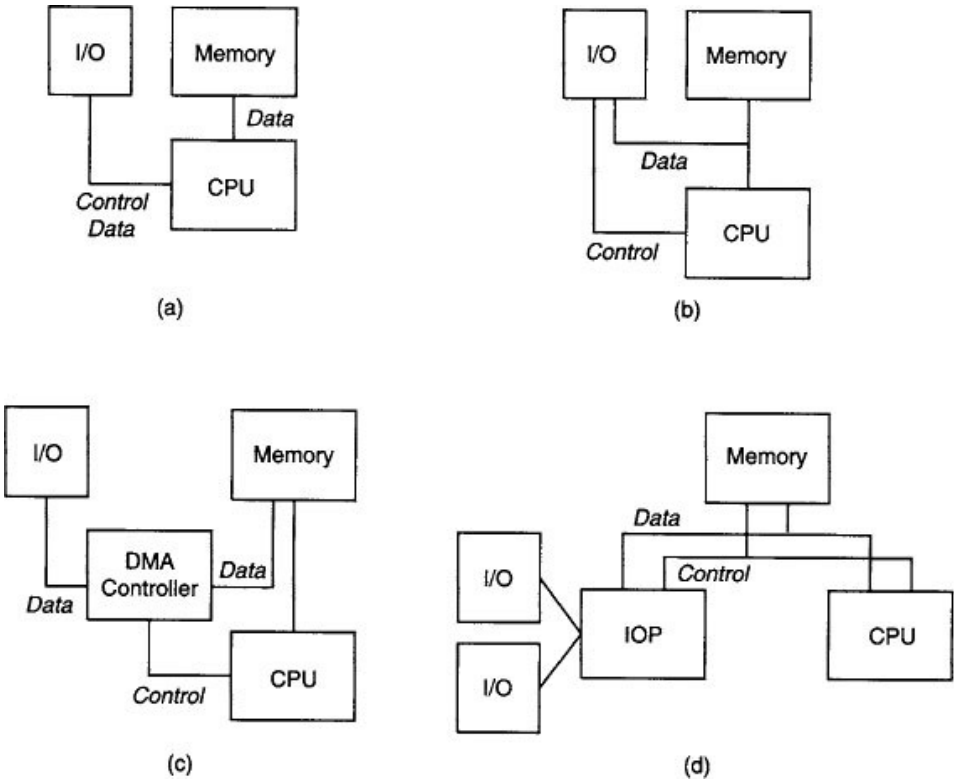


Figure A.5 Different Types of Input/Output. *(a) Programmed I/O; (b) interrupt-driven I/O; (c) DMA-managed I/O; (d) IOP-based I/O.*

- The third type, *DMA-managed I/O,* shown in Figure A.5c, improves on the performance of interrupt-driven I/O by allowing the I/O controller to access memory directly. The controller can use a series of bus transactions to move large blocks of data to or from an I/O device and to interrupt the OS only when it completes the entire task. This I/O method is very commonly used for devices that transfer blocks of data, such as disks.
- The final type of I/O is a more sophisticated DMA-managed I/O method that uses *I/O processors* (IOPs), often referred to as *channels* in mainframe computers (see Figure A.5d). An IOP is a special processor that can execute software on its own and can therefore manage complex I/O transactions. The IOP gets its instructions from a program in memory set up by the OS. The OS and the IOP communicate through main memory. The IOP can buffer transactions for various devices and bundle them to make the best utilization of the available I/O resources.



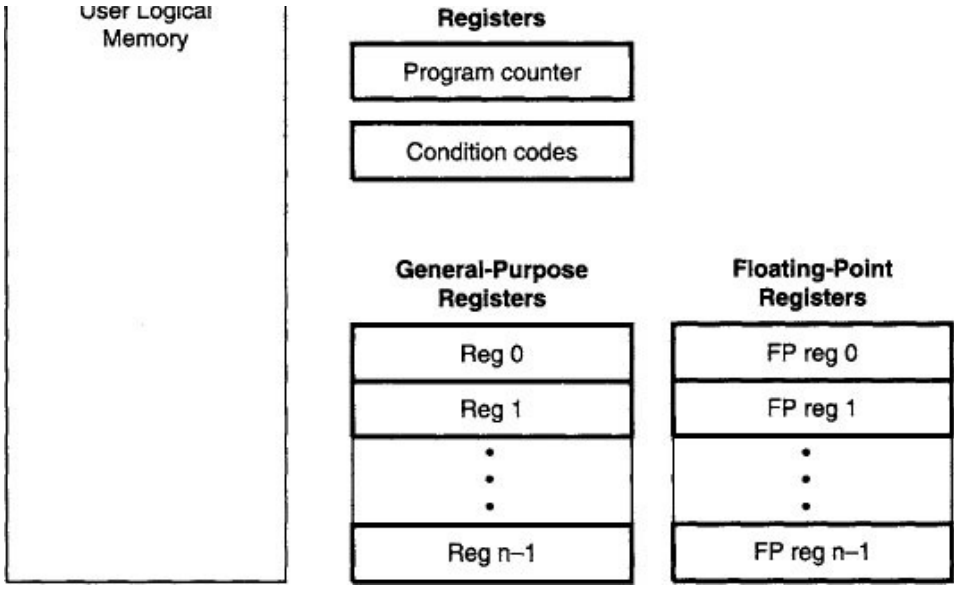


Figure A.6 Key Architected User State of a Typical ISA. *This includes a logical (virtual) memory space, a special-purpose register, general-purpose registers, and floating-point registers.*

A.2 The User ISA: Computation

The architecture of a processor, in its basic form, typically defines a set of storage resources, e.g., registers and memory, and a set of instructions that transform data held in the registers and memory. The definition of the storage resources and the instructions that manipulate data are documented in the instruction set architecture (ISA) of the processor. In order to ensure software compatibility between various implementations of the ISA, it is necessary that the description of an instruction be detailed enough to specify the exact operations that must occur as a result of executing the instruction. The key elements of the architected storage visible to a user program are illustrated in Figure A.6. The user memory is the bulk of storage, yet it is the registers that are important for performing computation. The register state is more varied, both in format and function, than the memory.

A.2.1 Register Architecture

In typical applications the registers that are primarily used are the general-purpose registers and a few of the special-purpose registers. The use of other special-purpose registers or specially typed registers, such as the floating-point registers, varies with the nature of the application.

General-Purpose Registers

These registers are often referred to as “working” registers. They are commonly used to hold operands for instructions. They are also used as temporary storage for intermediate values of complex operations and for frequently used constants or addresses. General-purpose registers often hold different types of values. For example, the contents of a general-purpose register may be a Boolean variable, a character variable, a string of characters, a halfword, a 32-bit integer, a 64-bit integer, or an address in memory. Occasionally a general-purpose register may have a special function, for example, as a stack pointer.

Typed Registers

It is often convenient to provide separate registers for special types of operations. For example, the PowerPC provides separate registers used as operands only for floating-point operations and are hence called floating-point registers. The IA-32 ISA provides a register type for holding pointers to segments of memory. Using multiple register types allows the hardware to be implemented more cleanly, with operands being located close to the functional units that perform the operation. However, in some cases, it can make the compiler writer’s job more difficult because it requires keeping track of the register types that hold certain data items, and it may occasionally require movement of a data item to get it into the right register type.

Special-Purpose Registers

The most important of these is the program counter (PC). Other

special-purpose registers include condition code registers, stack pointers, link registers, and loop count registers. Special-purpose registers often tend to be used implicitly by certain instructions; they may not be explicitly specified as operands in the instructions that manipulate them. This is often the case with link registers used by procedure branches or jumps. As another example, the Intel IA-32 ISA has a single register, EFLAGS, that holds implicitly set condition code bits along with a number of other status bits.

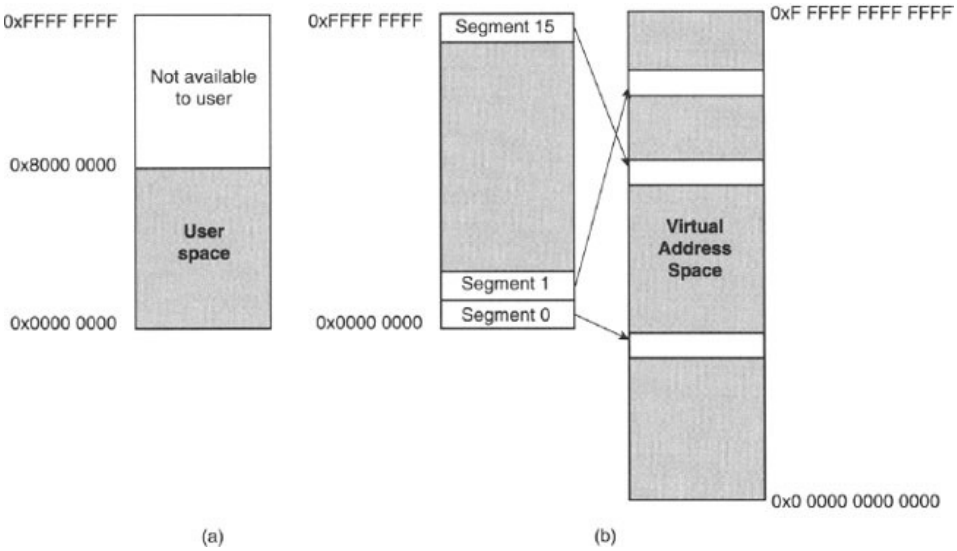


Figure A.7 Memory Architecture as Seen by the User. *(a) Linear addressing in MIPS R-series; (b) Segmented addressing in PowerPC.*

A.2.2 Memory Architecture

From the perspective of application programs, the logical memory architecture is fairly straightforward. In some ISAs, main memory appears as a single linear address space. In other ISAs, main memory appears as a set of segments pointed to by typed segment registers containing their base addresses. The Intel IA-32, for example, is architected as a set of segments, though an OS is able to make it appear as a single linear space by placing identical values in the segment registers (as is done in both Windows and UNIX). The IA-32 memory architecture is summarized in Section A.8.2.

An example of a linear address space is the user-mode address space (*kuseg*), as defined in the MIPS 32-bit ISA and shown in Figure A.7a. The 2GB (2^{31}bytes) user space starts at address zero, 0x0000 0000, and extends to address 0x7FFF FFFF. An address will typically be contained in a 32-bit register. The contents of such a register will be a valid address only if the high-order bit is 0. Any attempt to access memory with any other address causes an Address Error exception.

The PowerPC, on the other hand, divides the user address space into segments that each contain 256MB (2^{28} bytes), as shown in Figure A.7b. The architecture allows an application to address a large number of segments, even though at any one time only a small number are directly accessible. In the 32-bit addressing mode, only 16 segments are visible to the program. These 16 segments are pointed to by segment registers SR0—SR15. The segments that are currently visible form a contiguous 32-bit *effective* address space, with the high-order 4 bits of an address indicating the segment register associated with the address. Each segment register points to a 256MB segment contained in a very large 2^{52}-byte address space called the *virtual* address space.

For management and protection, memory can be divided into regions. Typically, the granularity of a management/protection region is a page, containing a power-of-2 number of bytes; 4KB, for example. Within a user's address space, different pages may be marked as having different access privileges, e.g., some combination of read, write, and execute privileges. The application program is usually not concerned about the exact page size, however; it becomes aware of the page size only, for example, when a sequence of accesses crosses from a page where accesses are valid to one that disallows such an access.

A.2.3 User Instructions

Instructions are the means for transforming data held in registers and memory. Table A.1 gives examples of various types of instructions. Instructions are usually classified according to the resources, specifically the storage elements and functional units, involved in their operation. Thus integer and logical operations generally involve the manipulation of general-purpose registers using integer arithmetic and logical units (ALUs); memory instructions involve main memory; and floating-point instructions involve floating-point registers and floating-point units. Besides these, there is a class of instructions that affect the flow of control of the program rather than the contents of data resources: the branch and jump instructions.

TABLE A.1 Example User Instructions

Memory Instructions	Integer Instructions	Floating-Point Instructions	Branch Instructions
load byte	add	add single	branch
load word	compare logical	multiply double	branch if negative
store byte	exclusive-OR	multiply-add double	branch and link
store multiple	count leading zeros	convert to integer	jump to subroutine
load double	rotate left with carry	compare double	return
.....	.....	.....	.....
.....	.....	.....	.....

Memory Load and Memory Store Instructions

These instructions cause the movement of data from a register to memory (a store operation) or from memory to a register (a load operation). The memory location involved is specified in the instruction by means of its address. The address itself is a value derived from adding the contents of one or more registers specified in the instruction. Sometimes an immediate field (constant) in the instruction may be used in addition to register operands. The value thus computed is the address as seen by the user, often referred to as the *virtual address,* the *logical address,* or the *effective address.* An address in a segmented memory may be split into a portion that specifies the desired segment and a portion that specifies the desired offset within that segment.