The Cost Virtualiza

ULRICH DREPPER, RED HAT

Software developers need to be aware of the compromises they face when using virtualization technology.

of
tion

The Cost of Virtualization

Virtualization can be implemented in many different ways. It can be done with and without hardware support. The virtualized operating system can be expected to be changed in preparation for virtualization, or it can be expected to work unchanged. Regardless, software developers must strive to meet the three goals of virtualization spelled out by Gerald Popek and Robert Goldberg: fidelity, performance, and safety.[1]

We may make compromises for each of the goals. For example, people in some situations are OK with sacrificing some performance. In fact, this is almost always mandatory for performance: compared with execution of an operating system on naked hardware, execution of a virtualized operating system takes more effort and somehow must be paid for.

This article is about the compromises developers have to be aware of when dealing with various types of virtualization. Ignorance of these issues can mean that a program's execution speed is unnecessarily reduced. It can also lead to malfunction, even if the fidelity of the virtualization is 100 percent. The implications of virtualization for block and network I/O are discussed elsewhere in this issue of *Queue* and therefore are not specifically addressed here. Only the related topic of DMA (direct memory access) is discussed.

First, however, we should look at a few details about commonly used virtualization technologies.

SOME VIRTUALIZATION TECHNOLOGY DETAILS

Virtualization implementations on commodity hardware were developed before processors and chipsets gained hardware support for virtualization. Existing features of the hardware allowed the implementation of virtualized operating systems, but at a price:

- The virtualized operating systems had to cooperate (i.e., had to be modified to be used).
- Performance suffered in some situations to achieve 100 percent fidelity.

This type of virtualization—*paravirtualization*—is popular, but developers need to know about its pitfalls to avoid major performance penalties. Despite these penalties, virtualization enables certain technologies, such as migration and check-pointing. Since there is a piece of software underneath the executed operating systems—the VMM (virtual machine monitor), or hypervisor—that controls the execution of the operating system domains, it is possible just to stop executing them. The VMM also knows about all the resources allotted to the executed domain. This means it can store all that information and later resume execution as if nothing happened.

With almost the same technology, it is also possible to resume the execution on a different machine. This domain migration adds another powerful weapon to the arsenal of the system architect, as it can be used to increase system utilization and availability.

Migration is a problem, however, if the network of machines is heterogeneous. If the operating system and/or applications running on it depend on CPU functionality, the migrated domain must depend on and use only the functionality that is available on all machines.

SAFETY REQUIREMENTS

To implement virtualization safely, the individual domains running on the virtual machines must be isolated. The extent of the isolation depends on the use case.

At the bare minimum, the domains must not be able to crash each other (i.e., a mistake or willful attack in one virtual machine must not have repercussions for the other domains). It is therefore paramount that the VMM centrally controls hardware resources such as memory and that the individual domains can modify only the memory allotted to them. In the end this means that the domains cannot be granted administration rights to the actual physical memory.

This is true for other hardware devices as well. Direct access to a hard drive or NIC (network interface card) used by multiple domains is a security problem. Therefore, in many situations domains do not get to access that hardware. Instead, these domains get access to virtual devices implemented by the VMM. The data is transported indirectly from the real underlying hardware device to the domains.

This indirection is not acceptable in some situations. For servers, hard drive and network bandwidth and latency are critical. Both are hampered by the virtual device implementation. Support for 3D graphics cards is at least equally as complicated. As a result, domains get dedicated hardware devices that they then can control on their own. For efficiency, a single hardware device can represent itself as multiple devices to the VMM and the domains. In this case the safety aspect of the virtualization is pushed down to the hardware (or firmware running directly on the hardware).

For memory, though, such separation at the hardware level has not been proposed yet, and it is unlikely it ever will be since shared memory optimizations are still useful and widely applicable.

COSTS OF USING VMMS

The VMM is a piece of software that is independent of the operating system kernel running in the domains. Using functionality in the VMM from a guest operating system kernel therefore requires a kind of execution transition that did not exist before. These VMM enter-and-exit operations can be implemented in various ways. In paravirtualized kernels it might be a specialized jump, and in hardware-assisted virtualization environments it is a set of new instructions. Regardless of the approach, there is one common factor: the entry and exit do not come cheap.

Upon entry to the VMM, the CPU has to be put into a more privileged state. A lot of internal state has to be saved and restored. This can be quite costly, and as is the case with system calls, the content of some of the CPU caches is lost. It is possible to play some tricks and avoid explicit flushes of caches (even those using virtual addresses), but the VMM code needs cache lines for its code and data, which means some of the old entries will be evicted.

These cache effects must not be underestimated. VMM invocations can be frequent, at which point the costs are measurable. The invocations often happen implicitly, triggered by memory handling.

For this reason some processors today (and likely all in the future) have means to avoid cache flushes for some caches. The most performance-critical cache, when it comes to VMM entry and exit, is the TLB (translation look-aside buffer). This cache stores the physical addresses resulting from the translation of a virtual address. These translations can be very costly, and the cache is indexed using virtual addresses that are shared across different guest domains.

Without employing some trickery, flushing the TLB caches is necessary whenever the VMM is entered or exited. This can be avoided by extending the index of the cache: in addition to the virtual address, use a token that is unique for each guest domain and the VMM (the TLB is tagged). Then, flushing the TLB caches is not necessary, and entering/exiting the VMM does not automatically lead to TLB misses.

MEMORY HANDLING OF THE VMM

As explained before, access to the physical memory must be restricted to the VMM. There are no signs that direct hardware support for compartmentalization of the memory is forthcoming.

This means the guest domain operating system kernels get access to virtualized physical memory only, and the functionality of the processor used to implement virtual memory is reused to implement this second level of virtualization. For paravirtualized kernels, this means using the ring structure of access permissions (as implemented at least in the x86 and x86-64 processors). Only the VMM has access at ring-level zero (which means full access to physical memory). The guest domain operating system

The reality is that domains should have memory allocated according to their current and immediate-future needs.

kernels have access only at ring-level one and only to the memory pages that the VMM explicitly allocates to them. The guest domain operating system kernel in turn can subdivide the memory to make it available at even higher ring levels (usually three) to the individual processes.

For hardware-assisted virtualization, the processor implements one or more additional rings that the VMM can use to virtualize memory. The guest domain operating system kernels run as usual, using the rings as they always do, but the memory they use for the ring-zero access is in fact limited to what the VMM allots to them. As such, it is not true ring-zero memory access.

The complications do not stop with ring use. Memory use changes over the uptime of a system. Things would be simple if each domain were assigned a specific amount of physical memory and then started the processes that

The Cost of Virtualization

use it. Once all the page tables were set up, the memory handling would be static. This is far removed from the reality of most systems, however.

The reality is that domains should have memory allocated according to their current and immediate-future needs. This allows virtualized environments to run more efficiently, since fewer resources remain unused. This is akin to the overcommit situation in virtual memory handling. Additionally, the guest domain operating system kernels create new processes that have to get their own page-table trees. These trees have to be known to the VMM.

For paravirtualized operating system kernels, the memory operations requiring VMM cooperation are performed as explicit requests to the VMM. In hardware-assisted virtualization the situation is more complicated: the VMM has to second-guess the operations performed by the guest domain operating system kernel and mimic the operation. The VMM has to maintain a so-called shadow page table, which is the page table that is actually used by the CPU, instead of the CPU's own page table.

Both of these approaches incur significant costs. Every operation of the guest domain operating system kernel that deals with memory allocation becomes measurably slower. The very latest versions of the x86 and x86-64 processors include more hardware support for memory, which reduces the overhead of the administration by implementing virtual physical memory in hardware. This approach still involves two sets of page tables, but they complement each other. The guest virtual address is first translated into a host virtual address, which in turn is translated into the real physical address. The schematics in figure 1 show how the two page-table trees work together. (For a detailed explanation of the diagram, see my article *What Every Programmer Should Know About Memory*.[2]) This means the address translation becomes more complicated and slower.

A micro-benchmark shows the penalties incurred better than a thousand words. The following program is a simple pointer-chasing program that uses different total amounts of memory:

```
struct list {
struct list *n;
long pad[NPAD];
};
void chase(long n) {
l = start_of_list;
while (n->0)l = l->n;
}
```

This program measures the time it takes to follow the closed list along the pointers. (More on this program, the data structure, the data that can be gathered with it, and other memory-related information can be found in my previously mentioned article). The result, shown in figure 2, is a graph showing the slowdown the same program incurs when running in a virtual guest domain as compared with running the code without virtualization.

Each of the graphs has two distinct areas: where the data caches of the processor are sufficient to hold the data the program needs, the slowdown is close to zero; but

Guest Address Translation with Extended Page Table Trees

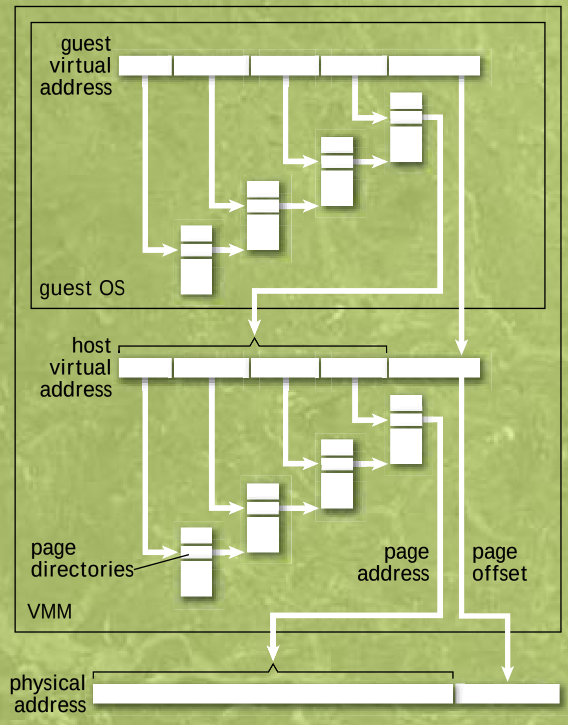

FIG 1

once the data does not fit in the data caches, the slowdowns grow rapidly. The different processors handle the situation differently as well. On Intel processors virtualization costs about 17 percent in performance, and on AMD processors about 38 percent. (The higher cost for AMD processors in this specific case are likely a result of the specific version of the processor used. In most cases the difference is likely not to be that big.) These are both significant numbers, which can be measured in real life for applications with large memory requirements. This should shake developers awake.

VIRTUAL MEMORY IN GUEST DOMAINS

At this point it's worth going into a few more details about the memory access in guest domains. We distinguish here between two cases: the use of shadow page tables and the use of virtual physical memory. The important events to look at are TLB misses and page faults.

With shadow page tables, the page-table tree that the processor is actually using is the one maintained by the VMM. Whenever a TLB miss occurs, this page-table tree is used to resolve the cache miss and compute the physical address. This requires three to five memory accesses, depending on the platform. When a page fault occurs, the guest domain operating system kernel gets invoked. It determines the action to be performed. In the case of a paravirtualized kernel it will call into the VMM to set up the mapping for the new page. In the case of hardware-assisted virtualization the kernel modifies its own page-table tree. This change is not visible to the processor, though. When the program is continued it will fault again. This time the VMM notices that its page tables are not in sync with the page tables of the guest domain operating system kernel, and it makes appropriate changes to reflect the change in that kernel.

If the processor supports virtual physical memory, a TLB miss becomes more complex. Now we have two page-table trees to handle. First, the virtual address in the program is translated into a virtual physical address using the page-table tree of the guest domain operating system kernel. Then this address must be translated into a real virtual address using the page-table tree of the VMM. That means the process might now require up to twice as many memory accesses as before. This complexity is made up for by simplified memory handling in the VMM. No longer are shadow page tables needed. The guest domain operating system kernel, when being invoked in response to a page fault, modifies its page-table tree and continues the program. Since its processor uses the tree that has just been modified, no further steps are necessary.

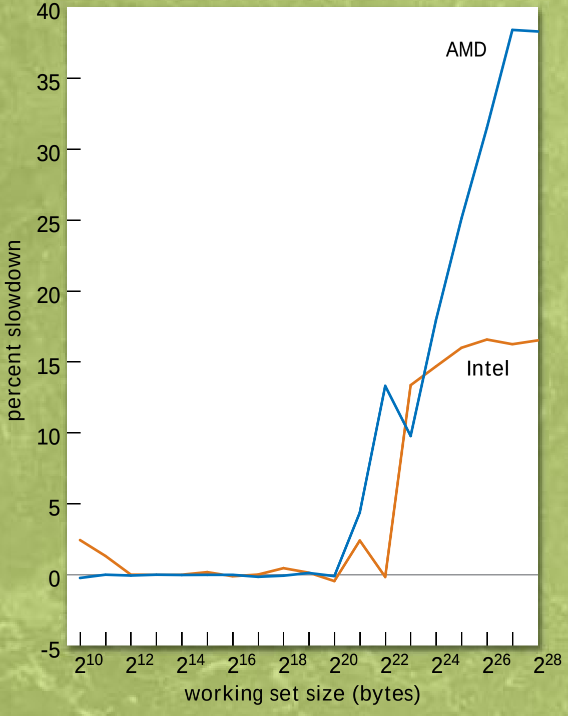

FIG 2

DMA AND MEMORY PINNING

The implementation of virtual physical memory brings with it a few more problems. The first one has mainly to do with the use of I/O devices but is also felt by user-level applications. For efficient I/O most modern devices sidestep the CPU when transmitting data and directly read or write data from/to memory. To keep the devices and the operating system implementation simple, the I/O devices in most cases have no idea about virtual memory. This means that for the duration of the I/O request the used memory region must be fixed (*pinned* is the operating system-developer jargon) and cannot be reused for any other purpose at that time.

These devices also do not know anything about virtualization. This means that the pinning of the memory must extend beyond the guest physical memory implementation. Implementing this correctly means additional work and costs.

The problem of pinning memory is, as hinted at earlier, not exclusively a problem of the operating system kernel. User-level applications can request memory to be *locked* (yet another term for the same concept, this time stemming from POSIX) so that no virtual memory implementation artifacts can affect the use of that memory region. This locking is a privileged operation since it consumes precious system resources. Locking memory in user-level code therefore requires the appropriate support in the operating system kernel and the VMM, to have the privileges to perform the operation, and for the system to have sufficient resources to comply in both the guest domain operating system kernel and the VMM. Real-time, low-latency, and high-security programming—the three situations where memory locking is predominantly used—is much harder in virtualized environments.

MEMORY HANDLING OPTIMIZATIONS

A closely related problem is that the guest domain operating system kernel needs control over physical memory to provide support for large memory pages. Large memory pages are an optimization some operating systems provide to reduce the cost of memory accesses when large amounts of memory are used. We won't go into the details (see my previously mentioned article), but it is sufficient to know that the speed-ups can be significant and that the optimization is based on reducing the TLB miss rate by treating a large number of the normal, small memory pages as one big page.

For the reasons explained previously, this optimization is even more important in virtual environments where TLB misses are more expensive. It requires the pages making up the large page to be consecutive in physical memory. Achieving this by itself is tricky because of fragmentation. If a VMM controls the physical memory, this is even more problematic since both the VMM and the guest-domain operating-system kernel need to coordinate. Even with guest physical memory this coordination is needed: the guest kernel can use large pages, but unless they also map to large pages in the page-table tree of the VMM, there will be no big wins.

Another memory-related problem we should discuss is one that will grow in importance and that needs optimizations at the application level. Almost all multiprocessor commodity systems in the future will have a NUMA (nonuniform memory architecture). This is the result of attaching memory banks to each processor in the system instead of just to (a set of) memory controllers. The result is that access costs to different physical memory addresses can differ.

This is a well-known problem that can be handled in today's operating system kernel and programs on top of it. With virtual physical memory, however, the virtualized operating system kernel might not know the exact details

To avoid significant performance losses, **certain optimizations,** which have always been beneficial, become **more urgent.**

of the placement of the physical memory. The result is that unless the complete NUMA state can be communicated from the VMM to the guest domain operating system kernel, virtualization will prevent some NUMA optimizations.

PROBLEMS OF FEATURISM

The product lines of CPUs show more and more features, all of which hopefully benefit the performance of the operating system kernel and the applications. Software components, therefore, are well advised to use the new features.

In virtual machines, however, this can be a problem. The VMM needs to support the features, too. Sometimes this can be as simple as passing the information about the existence of the feature on to the guest domain operating system kernel. Sometimes it means more work has to be done in the administration of the guests. These are problems that can be handled in a single place, the VMM.

There is a closely related problem, though, that is not so easy to solve. When virtualization is used for migration and the network of machines is not completely homogeneous, announcing the existence of a feature to the guest domains can lead to trouble. If a guest domain is first executed on a machine with the appropriate feature available and then is migrated to a machine with the feature missing, the operating system kernel or application might fail to continue running.

This could be regarded as a management issue in that such problems should simply be avoided. In practice this is usually an unattainable goal, since all organizations have machines of different ages in use. Using the least common denominator of the features is one way to handle the situation, but this might sacrifice a significant portion of the performance. A better solution would be to handle the change of the available features dynamically (i.e., when features are not available anymore, the program stops using the features and vice versa). This is not at all supported in today's operating systems and will require extensive changes. Developers might nevertheless want to keep this in mind when designing their code. Failure to handle changing CPU features could lead to crashes and incorrect results.

WHAT PROGRAMMERS SHOULD CARE ABOUT

The previous sections highlighted the changes a program experiences when being executed in a virtual machine. Here is a summary of the points that developers must be aware of.

Accessing devices, such as hard drives, NICs, and graphics cards, can be significantly more expensive in a virtual machine. Changes to alleviate the costs in some situations have been developed, but developers should try even harder to use caches and avoid unnecessary accesses.

TLB misses in virtual environments are also significantly more expensive. Increased efficiency of the TLB cache is needed so as not to lose performance. The operating system developers must use TLB tagging, and everybody must reduce the number of TLB entries in use at any one time by allocating memory as compactly as possible in the virtual address space. TLB tagging will only increase the cache pressure.

Developers must look into reducing the code size and ordering the code and data of their programs. This minimizes the footprint at any one time.

Page faults are also significantly more expensive. Reducing the code and data size helps here, too. It is also possible to prefault memory pages or at least let the kernel know about the usage patterns so that it might page in more than one page at once.

The use of processor features should be more tightly controlled. Ideally, each use implies a check for the availability of the CPU feature. This can come in many forms, not necessarily explicit tests. A program should be prepared to see the feature set change over the runtime of the process and provide the operating system with a means to signal the change. Alternatively, the operating system could provide emulation of the newer features on older processors.

THE IMPORTANCE OF OPTIMIZATION

Today's virtualization technologies largely fulfill virtualization's three goals of fidelity, performance, and safety. As such, programmers generally do not have to care whether their code runs in a virtual environment or not.

To avoid significant performance losses, certain optimizations, which always have been beneficial, become more urgent. Developers have to shift their emphasis.

Especially important is optimizing memory handling, which is little understood today. For low-level languages, programmers must take matters in their own hands and optimize their code. For high-level, interpreted, and/or scripted languages the work must be done mainly by the implementers of those systems. It is the responsibility of the developer to select an appropriate implementation or, failing that, choose a programming language that is better suited for high-performance programming. Q

REFERENCES

1. Popek, G. J., Goldberg, R. P. 1974. Formal requirements for virtualizable third-generation architectures. *Communications of the ACM* 17(7): 412.
2. Drepper, U. 2007. What every programmer should know about memory; http://people.redhat.com/drepper/cpumemory.pdf.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

ULRICH DREPPER is a consulting engineer at Red Hat, where he has worked for the past 12 years. He is interested in all kinds of low-level programming and has been involved with Linux for almost 15 years.