Improving Machine Virtualization with 'Hotplug Memory'

Shlomit S. Pinter
IBM Research Laboratory in Haifa
Haifa 31905, Israel
shlomit@il.ibm.com

Yariv Aridor
IBM Research Laboratory in Haifa
Haifa 31905, Israel
yariv@il.ibm.com

Steven Shultz
IBM Linux/zVM Development
Endicott, USA
shultzss@us.ibm.com

Sergey Guenender
IBM Research Laboratory in Haifa
Haifa 31905, Israel
guenen@il.ibm.com

Abstract

Machine virtualization has emerged as a key technology for server consolidation and on-demand server provisioning. To support this trend, it is essential to improve the performance of virtualization software and hence enable the efficient running of many virtual machines.

We present a virtualization system that can dynamically extend the real memory of its guest virtual machines. We describe an implementation of dynamic memory extension for Linux guests running on the IBM zVM virtualization environment. Our implementation for the Linux extension is based on device drivers for accessing these dynamic memory extensions.

Moreover, we show that this new capability can improve utilization and performance of the Linux guests in our virtualization environment. Specifically, memory management is improved and more virtual machines can run at a given moment. We study the utilization of dynamic memory extension of a Linux guest for a JVM heap. Running Specjbb2000 benchmark on a small virtual machine extended with dynamic memory to host the heap, we measured an improvement in transaction throughput and a 23.23% reduction in paging activity compared to an initially large machine. We further studied the implication of our experiments on the number of virtual machines that can run efficiently.

1. Introduction

Most applications today do not fully utilize the physical resources of the machine on which they are running. Machine virtualization technology enables us to run several virtual machines on a single platform. Each virtual machine runs a separate operating system instance in a secure and isolated fashion. High utilization of the physical resources is achieved by sharing them between several virtual machines. Server consolidation based on virtual machines can reduce hardware requirements and management costs. Moreover, this type of simple server build-up (porting) answers the needs for today's on-demand market.

Virtual machines have long been used to concurrently run different operating systems on a single hardware platform [2, 3, 4, and 10]. The virtualization software, also known as hypervisor, host, or virtual machine monitor, exports an interface that reflects the hardware defined for the virtual machine; the virtual machine is also known as the guest, or domain. The main challenge of the virtualization software is the effective sharing of resources with the goal of efficiently running as many guests as possible.

In this work, we focus on the efficient utilization and management of physical memory in a virtualization environment. In most systems, a virtual machine is configured with memory resources that its operating system can handle effectively [e.g., 1, 3, and 10]. Once the system is initialized, these resources are fixed for the lifetime of the virtual machine. This leads to reduced flexibility when allocating memory to guest OSs. A few large virtual machines can be allocated to facilitate memory intensive workloads or many small virtual machines can be allocated to support many users. Because most operating systems do not support hotplug memory, allocating a small amount of physical memory up front makes dealing with some workloads impossible.

The situation is even more complex when Linux is the guest operating system. Each Linux guest will take over all of the memory it is allocated for its page and buffer cache. To avoid running out of physical memory, the host must allocate small amounts of memory to each Linux guest and restrict the total number of guests.

Our virtualization system can dynamically extend the real memory of a guest virtual machine, thus providing the needed flexibility. The guest OS can start with small memory allocations, which are extended as needed. This dynamic memory extension may not be a contiguous extension of the guest's real memory space, and it is 'plugged' and loaded (mapped) without the need for system shut down. In this respect, we can view it as hotplug memory that can be added and removed as needed.

Introducing dynamic memory extension to a guest operating system in a virtual environment presents new system challenges and new opportunities for improving virtualization performance. Since dynamic memory extension is external to the 'real memory' of the virtual machine it can be:

- Used on-demand as additional memory for a single virtual machine, or
- Shared by a set of virtual machines.

In this paper, we describe a concrete implementation of dynamically extended memory in a virtualization environment and investigate its potential. Linux does not currently support hotplug memory; thus we present a new generic Linux interface to this dynamic memory via device drivers. This saves us from having to modify the Linux memory management software, in contrast to other works that suggest changes to the memory management system for integration of hotplug memory in Linux. We note that in the latter case, the sharing option is not considered.

We leverage the concept of Discontiguous Saved Segments (DCSS) as a dynamic memory extension for a Linux guest machine running in the zVM virtualization system of IBM S/390[1]. Our experiments exploit a usage scenario, which takes advantage of the additional memory that can be dynamically allocated by our Linux guests.

We experimented with memory extension for hosting a JVM heap in a small virtual machine. This extended memory may be even larger than the size of the virtual machine itself. Our experiments with the Specjbb2000 benchmark, which is used for evaluating the performance of server-side Java [12], show an improvement in the transaction throughput, favoring the use of dynamic memory extension as heap space for a JVM. The paged-out activity was 76.77% of the standard (larger) configuration, which initially included the space used for the heap. This enabled us to run more Linux guests on small virtual machines that grow when needed, as opposed to initially defining and running large guests. The extra space was used only when the JVM was running.

The remainder of this paper is organized as follows. Section 2 describes a realization of dynamically extended memory in virtualization software and its use within Linux. In Section 3, we describe and investigate the use of dynamic memory extension as extra private memory for a user application. We describe related work in Section 4 and provide a closing discussion and summary in Section 5.

2. Dynamic memory extension architecture

Dynamic memory extensions are allocated in units of segments. Each segment is a chunk of host memory which guests can load (attach). Segments are named (i.e., can be uniquely identified) and can be persistent so they retain their data even when no guest is attached to them. The hypervisor creates, manages, and destroys the segments. A guest can load (attach to) segments and unload (detach from) segments. To load a segment, the guest sends the hypervisor a request for segment load. Similarly, to unload a segment, the guest requests a segment unload.

Segments can be loaded and unloaded at any time, provided the requesting guest has sufficient privileges. Once a segment has been loaded, it remains accessible to the guest for the rest of the guest's lifetime or until it is unloaded. A segment can be attached as read-only or read-write and can be private for a single guest or shared between multiple guests. When a segment is shared between multiple guests, standard shared memory semantics are presented.

Once a guest has loaded a segment, the kernel device driver running in the guest OS is responsible for providing the guest with access to the segment. In Linux, the segment is usually exposed as a block device, which means the device driver appears as a disk to the rest of the kernel and translates disk read and write requests to memory operations on the segment. Alternatively, the driver can expose the

[1] Currently, only the operating system CMS [11] utilizes DCSS memory as a fully hotplug memory that is supported by the memory management (rather than via device drivers).

segment as a character device, which provides standard mmap() semantics. Guest programs can then access the segment via standard memory accesses. It is the device driver's responsibility to map the segment into the program's virtual address space.

If the guest OS supports full hotplug memory, the hotplug memory infrastructure can be used to make the segment appear as part of the guest's real memory. As long as hotplug memory support is not available, memory extensions cannot be used as part of real memory and they can be accessed only by special device drivers. Such devices restrict the use of the memory, yet they are fairly simple and provide convenient interfaces to guest programs.

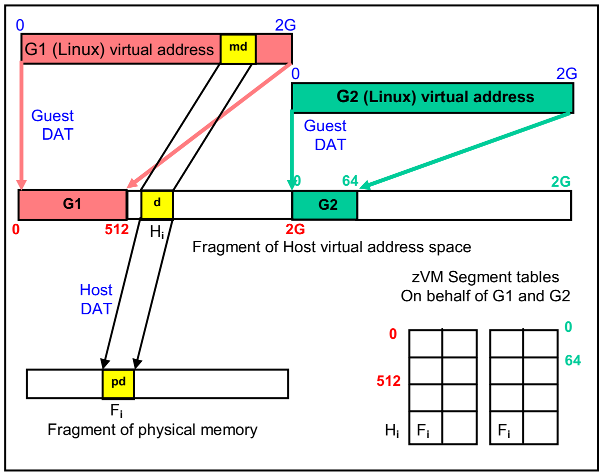
The replace segment operation saves the current contents of the attached segment, such that the next time the segment is attached, these contents will be the initial contents of the segment.

2.1. Host support for dynamic memory extension

In a virtual environment, the host manages the guest's real memory via segment tables. Dynamic memory is an extension, unknown to the guest OS at boot/ipl time; therefore, the segments tables, which are managed by the host, must grow and shrink dynamically. This raises new challenges, especially since the extended memory is not necessarily contiguous to the real memory space of a guest and may be shared by multiple guests.

During its life cycle, a dynamic memory extension must be defined in the host memory and then loaded by a privileged guest. At this point, the relevant tables are extended to accommodate the new extension and the memory is mapped to the guest's virtual address space. When the memory is used, the accessed pages are fetched, assigned to physical frames, and used through the hotplug device driver. When guests want to share a page of the memory, the respective tables must point to the same physical frame. Detaching an extension involves shrinking back the tables. To increase time performance, some future implementations may prefer not to shrink the tables. The use of dynamic memory extension by a client is presented for hosting a JVM heap (see Section 3).

2.1.1. Concrete Implementation. This section describes a concrete realization built on the DCSS infrastructure implemented for the hypervisor zVM. A DCSS memory segment is a dynamically loaded extension to a guest's main memory[2]. The segment is defined with a memory range (page range) that refers to an address space with origin zero. Because the real memory of each guest starts at zero, this constitutes an address relative to any guest's real memory. For a Linux guest, the DCSS memory segment must not overlap with the real memory space. The support provided is schematically described in Figure 1.



0
2G
G1 (Linux) virtual address
md
0
2G
G2 (Linux) virtual address
Guest DAT
Guest DAT
G1
d
0
64
2G
G2
0
512
H_i
2G
Fragment of Host virtual address space
zVM Segment tables On behalf of G1 and G2
Host DAT
pd
F_i
Fragment of physical memory
0
512
H_i
F_i
0
64
F_i

Figure 1: Address translation and DCSS placement in z/VM

In Figure 1, we show what happens when a DCSS is mapped and accessed by a Linux guest, followed by access of a second Linux guest. G1 and G2 denote two guest virtual machines. Guest G1 is defined with 512M physical memory (real), while guest G2 is defined with 64M. In principle, the translation of a memory address from a guest virtual machine into physical memory involves two stages [5]. In the first stage, a guest Direct Access Translation (DAT) is used to translate memory addresses between a guest virtual address space and the guest main memory. All guests' main memories are in the domain of the hypervisor, thus, we view them as located in the *hypervisor (host) virtual address space*.

In the second stage, a hypervisor DAT translates memory addresses between the hypervisor virtual address space and the real physical memory. For this, the hypervisor maintains a separate segment table for each guest. Each table includes a mapping of memory addresses onto the physical memory of the machine. To improve performance, the two-stage translation process is shortened into a single stage by a special component (an assist - SIE, see [5])[3]. Other virtualization software also provides direct access to hardware resources to shorten translation time [3].

[2] Denoted by *guest main storage* in IBM mainframe terminology.

[3] A shadow translation table is used in the translation process.

Consider the following example: the DCSS "**d**" in Figure 1 is defined in a page range at address H_i which is above 512M (i.e., above the real memory of G1 and G2). When "**d**" is loaded (attached) by G1, the space range at address H_i is mapped by the DCSS character device driver onto the virtual memory of G1, denoted by "**md**". During the load, the hypervisor generates new entries in the table (entry H_i in the G1 table). When "**d**" is accessed by G1, it is actually fetched into the physical memory, denoted "**pd**" in the physical frame F_i, and entry H_i in G1's table is populated by a pointer to F_i. From this point on, the DCSS can be used as in any character device driver.

The architecture described above makes it easy to share a DCSS between different guests. It is simply a matter of updating the segment tables of all the corresponding guests. For example, consider guest G2 in Figure 1. When G2 loads and accesses DCSS "**d**", then "**d**" is mapped onto G2's virtual memory and the entry for address H_i in the segment table of G2 is updated with F_i. Note that all Linux guests sharing a DCSS need to have real address space below the range of this DCSS.

2.2. Linux implementation of dynamic memory extension

Two device drivers for DCSS are implemented and run in kernel mode: a block device driver ([8]) and a character device driver. Each driver has a kernel interface to DCSS that encapsulates all interactions with the hypervisor and provides the functionality necessary to exploit shared memory (since DCSS can be used in shared mode). In addition, the character device driver supports a shared mmap() interface with MAP_SHARED semantics.

The kernel shared memory interface consists of three functions: segment load, segment unload, and segment replace. Segment load attaches a previously defined DCSS to the virtual machine. Segment unload detaches a previously attached DCSS from a virtual machine. Segment replace saves the current contents of the attached DCSS such that the next time the DCSS is attached, these contents will be the initial contents of the DCSS[4]. These three functions require hypervisor functionality to accomplish their respective goals. The details of these functions, however, are beyond the scope of this work. When a driver is invoked to attach a DCSS, it uses the input DCSS name to segment load it. The memory attached via the DCSS is then accessed in a way similar to ramdisks for the block driver and similar to shared memory access for the character driver. Unlike a ramdisk, the segment replace function can be used to direct the DCSS block device driver to make the current contents of the block device permanent.

[4] A DCSS can be saved in zVM's spool space (be persistent).

Figure 2 illustrates the system and components used to access DCSS, as compared to the use of an I/O device driver. The character device driver (dcssshm) provides direct access to memory, whereas the dasd device driver works its way via several layers of software while accessing a disk.

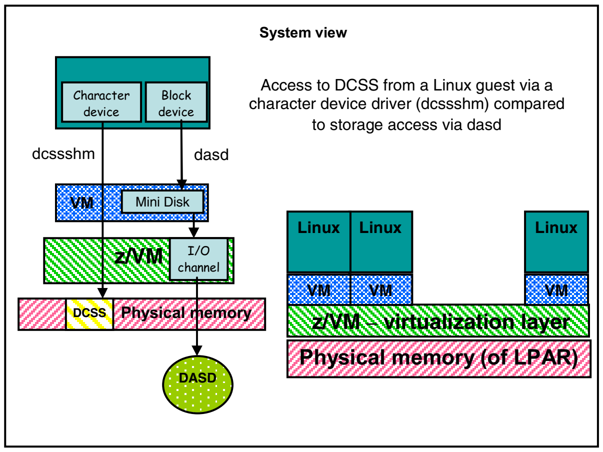

System view
Access to DCSS from a Linux guest via a character device driver (dcssshm) compared to storage access via dasd
Character device
Block device
dcssshm
dasd
VM
Mini Disk
z/VM
I/O channel
DCSS
Physical memory
DASD
Linux
Linux
Linux
VM
VM
VM
z/VM – virtualization layer
Physical memory (of LPAR)

Figure 2: System View

3. Private dynamic memory extension

Using hotplug as private memory to dynamically extend the effective physical memory defined for a guest virtual machine is advantageous because the guest OS and/or the application provide the hypervisor with the exact memory usage information it needs. As a result, the total virtual memory supported by the host can be tailored more closely to the 'on-demand' memory requirements of all the Linux guests. This serves to maximize the number of Linux guests that can be efficiently supported at a given instant in time. Running small virtual machines that grow when needed, as opposed to initially defining and running large guests, indirectly reduces swapping and thus also the double paging effect [9]. In double paging, also known as redundant paging, paging is done both by the guest to a virtual paging device and by the host. In particular, the DCSS space is paged out only by the host.

Any virtualization system limits the number of guests running at a given time. This number depends on the available resources and the resources requested by the guests. More guest machines may be accommodated when they are small. Thus, it is important to define small guests that can grow and shrink dynamically.