When NUMA meets Virtualization

Paper #11

Abstract

1 Introduction

2 Background

Cette section présente le background nécessaire pour comprendre la motivation de notre travail. D'autres information sur le background viendront au fur et à mesure à chaque fois que cela sera nécessaire.

2.1 Heap memory managers

Description des heap memory manager en général. Voir: NUMA aware heap memory manager

HotSpot

LibC

2.2 Non Uniform Memory Access (NUMA)

Voir: Optimizing the Memory Management of a Virtual Machine Monitor on a NUMA System

L'étude du NUMA a fait l'objet de plusieurs travaux de recherche dans le domaine des environnements natifs [**?**, **?**, **?**, **?**, **?**, **?**]. Les premiers travaux [**?**, **?**, **?**, **?**] se sont concentrés sur la prise en compte du NUMA au niveau des systèmes d'exploitation. Par exemple, Linux implante quatre politiques NUMA à savoir first touch (la politique par default), interleave, preffered et xxx. D'autres travaux de recherche [**?**, **?**, **?**, **?**, **?**] ont montré que la prise en compte du NUMA uniquement au niveau du noyeau n'est pas suffisant pour certaines applications, notamment les applications gérant elle même un tas de mémoire présentées ci-dessus.

2.3 NUMA aware heap memory manager (noted NaHMM)

Gescription général général de comment les heap memory manager prennent en compte le NUMA. Voir: NUMA aware heap memory manager.

Figure 1: Current hypervisors deal with NUMA in three ways: (1) xx, (2) xx, and (3) xx.

HotSpot

LibC

2.4 Virtualization

Décrire ce qu'uest la virtualisation. Et montrer que entraine un empilement de couche. L'OS n'a plus la main mise sur le matériel, c'est l'hypervisuer. Donc l'efficacité des politiques NUMA à la fois dans l'OS mais surtout dans les heap memory manager dépend de la topologie que l'hyperviseur à bien voulu montré à l'OS. La section suivante montre comment les hyperviseur actuel prennent en compte le NUMA, ainsi que la limite de leur approche.

3 Motivation and Assessment

This section presents the motivations for our work. Although all the contributions in this paper are principles which can be implemented in any virtualisation system, we rely on Xen to illustrate all our descriptions. Details about our experimentation environment are provided in Section 5.1.1.

3.1 The need for vNUMA

When a virtualised environment is executed on a NUMA architecture, performance of VMs (including those of applications embedded in these VMs) heavily depends on how the NUMA architecture is taken into account. Currently, three approaches are followed, as illustrated on Figure 1.

(1) The hypervisor is not NUMA aware and presents a UMA architecture to the hosted VMs. This is the most disastrous approach for NaHMM. Figure 2 reports performance evaluations with well-known benchmarks in such

Figure 2: A Non NUMA aware hypervisor is very bad for NaHMM.

Figure 3: A NUMA aware hypervisor limits performance degradation but need to be improved.

an environment (based on a non NUMA aware Xen hypervisor) when all resources of the VM are located on a single socket (and therefore a single node) and when they are distributed on several nodes and sockets. We can observe up to xx% performance degradation for benchmark xx.

(2) The hypervisor is NUMA aware but the hosted VMs are not and still have the vision of a UMA architecture, even if their allocated resources are distributed on several nodes and sockets. In this approach, the hypervisor has to implement some of the NUMA policies that were implemented in NUMA aware OS. For instance, Xen implements the first touch policy from Linux. This approach requires a significant engineering effort for the integration of policies in the hypervisor. It performs much better compared to the previous approach as reported in Figure 3. We observe up to xx% performance improvement for benchmark xxx. However, its performance is still far from the optimum (when VM's resources are all located on a single socket).

(3) A VM is given a virtual NUMA topology (hardware architecture) which describes the mapping of its allocated resources on nodes and sockets. This approach (called vNUMA [**?**]) is the most recent one. It followed the same evolution as operating systems (see Section 2.2), i.e. NUMA should not be only managed in the underlying system layer (hypervisor or operation system) but can benefit from many optimizations if managed in the upper layer (guest OS or user level heap memory manager). As motivated in Section 2.2, NaHMM can implement such optimizations and requires knowledge of the NUMA topology. Figure 4 shows the results with the same benchmark as previously, when vNUMA is activated in Xen. We observe a performance improvement of nearly xx% with benchmark xxx. Our contribution falls in this category, which has rarely been studied (evaluated) in the litterature. We focus on applications which include a NaHMM, which represent a large majority of existing applications (e.g. all data processing applications include a NaHMM). The next section first presents the implementation principles of vNUMA which is adopted by all hypervisors. It then analyzes its limitations, which affect its adoption by cloud operators.

Figure 4: vNUMA is the best way to take NUMA into account.

Figure 5: Hypervisor's decision may lead to different virtual NUMA topologies.

3.2 Actual vNUMA implementation principles and limitations

The vNUMA implementation principle consists in (for the hypervisor) storing the virtual resources topology of a VM in its ACPI tables, so that the guest OS executed by the VM uses it (at boot time) as any OS does. This implementation principle has the advantage to be straightforward (simply initializing the VM's ACPI tables according to the resources allocated to the VM). However, its main limitation is that a change in the NUMA topology cannot be taken into account without a system reboot. This limitation is quite weak in a native OS as a dynamic modification of the NUMA topology is not frequent if at all possible in the lifecycle of a machine. However, such NUMA topology changes are frequent in a virtualised environment, since the hypervisor can dynamically adapt the mapping between physical and virtual resources. More precisely, the hypervisor can change the NUMA topology of a VM with the following operations (see Figure 5):

- Scheduling may trigger the migration of vCPUs between sockets (see Figure 5.a).
- Ballooning[1] may induce a modification of the mapping between guest physical pages and host physical pages (see Figure 5.b). For instance, pages that were reclaimed (on balloon inflate) can be given back (on balloon deflate) on a different NUMA node.
- Upon VM migration between servers, the mapping of both vCPUs and guest physical pages will be reassigned on the machine of arrival (see Figure 5.c).

Due to the inability of today's hypervisors (Xen [**?**], KVM [**?**], VMware [**?**], and HyperV [**?**]) to dynamically adapt vNUMA's topologies, VMs have to execute during their whole lifetime with their boot time topology. This limitation of vNUMA can lead to inconsistent situations where the guest OS and its applications exploit a virtual NUMA topology which is very different from the actual one, as illustrated in Figure 6.a. The actual topology may have evolved as a consequence of a set of ballooning operations. Figure 6.b reports the evaluation of

[1] Ballooning is widely used for optimising memory management [**?**] in datacenters.

Figure 6: After live migration of a VM, its actual NUMA topology can be completely different from the virtual NUMA topology assigned at boot time (left). A static vNUMA implementation will dramatically degrade the application performance (right).

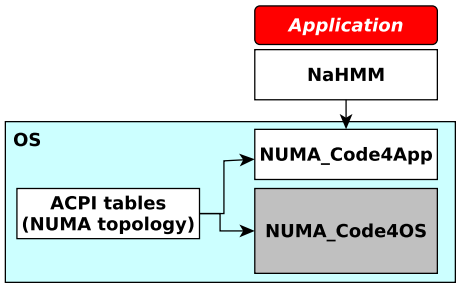

Figure 7: NUMA policy code organization

benchmark xxx executed in such an inconsistent situation. Compared with an execution on a non NUMA aware hypervisor (which is assumed to be the worse situation), we observe that a static vNUMA is disastrous with a performance loss of xx%. This limitation of vNUMA is recognised by all virtualisation providers which make the following recommendations: (1) either enable vNUMA and disable all hypervisor optimisations which would modify VM's topologies (which leads to a waste of resources), (2) either disable vNUMA and present a UMA architecture to VMs (which drastically degrade performance as we have seen above).

In this article, we introduce a new vNUMA implementation principle which allows VM given topologies to be dynamically adapted and exploited by guest OS and applications, without VM restart.

4 Adaptable vNUMA

Our contribution is a generic and straigtforward implementation of an adaptable vNUMA targetting NUMA aware applications, more specifically NaHMM. The next sections introduce the overall scheme and then detail its implementation in different contexts.

4.1 Design principles of an adaptable vNUMA

An analysis of NUMA management in the Linux operating system code shows that it is roughly composed

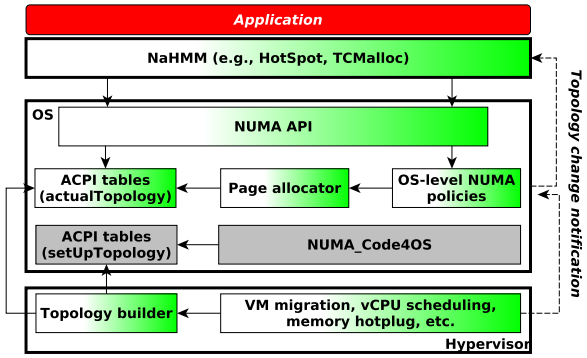

Figure 8: Adaptable vNUMA schema

of three types of elements (Figure 7): (1) ACPI tables, (2) the code which implements NUMA policies for the OS itself (noted OS_NUMA_Code4OS) and (3) the code which implements NUMA policies to be used by upper level applications (noted OS_NUMA_Code4App). ACPI tables are exploited by both OS_NUMA_Code4OS and OS_NUMA_Code4App.

Regarding OS_NUMA_Code4OS, its code is widely spread in the kernel code and its modification to take into account topology changes is impracticle. However, this is feasible for OS_NUMA_Code4App as the code is much smaller and concentrated in a reduced set of locations. Moreover, NUMA aware applications (here NaHMM) are sensitive to OS_NUMA_Code4App only. Based on these observations, we designed an adaptable vNUMA implementation scheme which is illustrated in Figure 8.

At VM startup, the VM receives two topologies from the hypervisor: a flat/UMA topology (noted *flatTopology*) and a topology which describes the current mapping of the VM resources on physical resources (noted *actualTopology*). Then, our design relies on three main features:

- The *flatTopology* is stored in the VM's ACPI tables and used at boot time by OS_NUMA_Code4OS in the guest OS. This is the default behavior of the guest OS as OS_NUMA_Code4OS is not modified.
- The *actualTopology* is actually implemented as fake ACPI tables and OS_NUMA_Code4App is modified to use the *actualTopology* (so that it is used by NUMA aware applications).
- Any modification (by the hypervisor) of the topology of a VM is reflected in its *actualTopology*.

One may wonder why booting with a *flatTopology* rather than with the *actualTopology* which

can be exploited by the guest OS (more precisely by OS_NUMA_Code4OS). However, since OS_NUMA_Code4OS is not modified to take into account changes in the *actualTopology*, OS_NUMA_Code4OS would start with the initial state of the *actualTopology* and use it during its entire lifetime, which would be worse (regarding performance) than using a *flatTopology*.

The code modifications which have to be made in OS_NUMA_Code4App include: NUMA APIs which are presented to user space, OS-level NUMA policies which manages resources allocated to applications, and the memory page allocator. These modifications are detailed in the next sections, showing that they are fairly simple and generic, i.e. applicable to any Unix like OS.

Regarding adaptations at runtime, the hypervisor is instrumented so that any modification of the mapping of virtual resources is reflected in the *actualTopology* of the involved VM. Such modifications occur when: a memory page from one VM is migrated from one NUMA node to another, a vCPU is migrated from one socket to another, or a VM is migrated to a another machine where its *actualTopology* cannot be restored as is. Modifications are notified to the VM's guest OS and handled by a specific handler (noted *topologyChangeHdlr*). This latter updates the *actualTopology*, updates the page allocator data structures and notifies the application layer. Since we target applications which rely on a NaHMM, taking into account this notification at the application level consists in re-engineering the NaHMM so that it becomes adaptable (handling notifications). By implementing adaptable NaHMM in Spot (Java) and TCMalloc (C), we cover a large majority of applications.

The next sections describe the implementation of the above principles in different hypervisors (Xen and KVM) and different guest OS (Linux, FreeBSD), demonstrating its genericity and its relative simplicity (in terms of LoC).

4.2 Hypervisor modifications

4.2.1 vNUMA construction

Purpose
Implementation in Xen
Implementation in KVM

4.2.2 Mapping modificartion tracking

Purpose
Implementation in Xen
Implementation in KVM

4.3 Hypervisor-Guest OS communication

4.3.1 vNUMA sending

Purpose
Implementation in Xen
Implementation in KVM

4.3.2 Advice

Purpose
Implementation in Xen
Implementation in KVM

4.4 vNUMA management in the guest OS

4.4.1 Setup

Purpose
Implementation in Linux
Implementation in FreeBSD

4.4.2 Update

Purpose
Implementation in Linux
Implementation in FreeBSD

4.5 vNUMA utilization within the duest OS

4.5.1 Memory page allocation

Purpose
Implementation in Linux
Implementation in FreeBSD

4.5.2 NUMA policy

Purpose
Implementation in Linux
Implementation in FreeBSD

4.5.3 NUMA user space API

Purpose
Implementation in Linux
Implementation in FreeBSD

4.5.4 On vNUMA update

Purpose
Implementation in Linux
Implementation in FreeBSD

4.6 Application aware NUMA

4.6.1 Setup

Purpose
Implementation in the JVM
Implementation in LibC

4.6.2 On vNUMA update

Purpose
Implementation in the JVM
Implementation in libC

5 Evaluations

5.1 Experimental setup

5.1.1 Hardware

6 Related work

A notre connaissance, un seul travail de recherche [**?**] étudié cette limitation. Pour cela [**?**], propose de revenir à la situation (2) que nous avons mentionné dans la section 3.1 avec une émélioration permettant à la VM de choisir sa politique. En effet, [**?**] implante au niveau hyperviseur toutes les politiques NUMA d'un OS et fournit un hypercall à la VM permettant de choisir sa politique. Cette solution a plusieurs limites: elle ne satisfait pas les applicatrions qui apportent leur propre politiques NUMA (JVM), elle suppose que la VM n'utilise qu'une politique à un moment donné (alors que la VM peut exécuter plusieurs applicartions, chacune ayant sa solution), ce qui représente une large majorité d'applications.

7 Conclusion

References