Implementing Page Coloring in the Xen Hypervisor

Andrea Braschi, Matr: 797136

February 14, 2014

1 Cloud Computing and Performance Predictability

Cloud computing is one of the emerging paradigm in computer science. The rationale behind cloud computing is sharing hardware resources among multiple tenants, thus increasing their utilization and decreasing energy consumption.

In order to improve performance mainly two family of techniques are adopted to avoid contention on shared resources: scheduling and isolation. The first family tries to optimize resources' utilization rescheduling application and reassigning them the resources, instead, isolation techniques assign to different applications their own partition of a certain resource in order to avoid to different application to be stealing each others their resource wasting time in continuous context switches. These two techniques guarantee also best predictability of performance since they avoid unpredictable contentions on resources.

In this project, I have been focusing my attention on a particular isolation technique: page coloring. This technique permits to partition last-level cache through processes and in particular through virtual machines.

In this report I will first provide a brief explanation on how caches work, then I will explain what page coloring is and why it would be effective in performance enhancement, then I will introduce the Xen hypervisor [1], one of the most utilized open source hypervisor (also Amazon's EC2 cloud [2] uses it) and I will describe how I have implemented page coloring algorithm in Xen; after that I will introduce some experiments I did to prove the correctness of my work and suggest some possible future developments of this project.

2 Background on Caches and Introduction of the Problem

Technological evolution lead to an exponential but diverging growth in CPU processing speed and in RAM speed. For that reason from the 80s we are at the point that CPU are faster than memories and need more bandwidth in respect from what memory can do.

From the principles of locality (temporal and spatial) we know that if a certain program accesses a certain memory address, in the near future it will look for the data near that address (spatial locality) and it also will ask again for this address (temporal locality). Caches are introduced exactly to exploit these phenomena in order to gave CPU more bandwidth, in fact caches are smaller and volatile memory attached to the CPU in order to make faster the access to data that are more probable to be asked. When CPU asks for a data that is not in cache, cache will import a certain amount of data that are near to it (block or line), this exploit also the known fact that in RAM a random access costs more time than a sequential access.

2.1 Cache Hierarchies

In modern multicore processors, typically we have three different levels of caches. The first level cache (L1) which essentially is the smallest one is divided into two parts one dedicated to instructions and one for data, every core has its own L1 cache. If we have a miss on L1 (CPU asks for data missing on L1) data are searched in second level cache (L2) which is also core dedicated but is bigger and slower in respect to L1 cache. After that, cores have a common third level cache (L3) which is the last step before the memory. All the arguments discussed in the following sessions, can also be applied to L2 cache in architectures with 2 level of caches in which the second level is shared among cores. For this reason, in order to be as general as possible, from now on, I will refer to Last-Level Cache (LLC).

2.2 N-ways Set Associative Caches

As I have already explained, the elementary units in caches are lines, a line is greater than a processor's word (typically a power of two), a cache loads (or discards) contiguous words in units of lines from main memory. In that way CPU exploits the advantage from spatial locality, storing in cache a certain quantity of words near the one that is required.

Most common caches are called n-ways set associative, and they work as follows: A cache is composed by 2^s sets which are groups of n lines. This subdivision make easier and faster search operation in cache, in fact, in

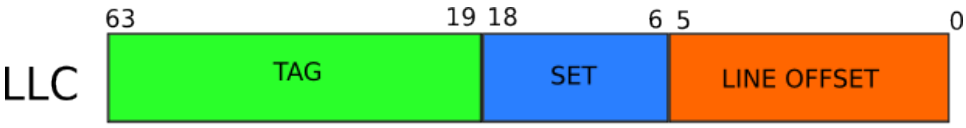

Figure 1: Address subdivision in LLC

words' addresses, the least l significant bits (where 2^l is line size in words) represent the line offset of the word, the following s bits represent the set number, remaining bits represent the tag. When a word is required by the CPU the tag of the n lines in the set corresponding to the set number of the word's address are compared to the word's tag and then if there is a match we have an hit and the line offset is used to retrieve the data needed, otherwise we have a miss.

2.3 Contention Problem

Since LLC is shared among cores is an easy reason of contentions among processes running on different cores. Operative systems allocate memory to processes one page at time, pages are bigger than processors' words, admitting that our memory is divided into 2^n page, the most significant n bits in memory addresses indicate the page number. Since, LLC has less set than the number of memory pages, it happens that different pages are mapped on the same cache set, that lead to the fact that different pages are loaded in the same cache set, and this may be cause of contention on this cache set. Contention is also raised from the policy adopted by caches in choosing which cache line to evict (from a certain set) in order to load a new line. The most common policy used is LRU (Least Recently Used) that evict the line that has been unused for the longer time (to exploit temporal locality). This policy, since is unable to predict future requests from CPU, may, some time, evict line that have an high locality and will soon be required by the processor.

Cache contentions belong to two categories *thrashing* and *polluting*: In thrashing [3] contentions a line with high locality replace an other line with high locality, these facts lead to continuous misses on cache. These continuous misses causes the memory manager to became the bottleneck of the system. Differently, in polluting happens that lines with low locality are replaced by line with low locality, so that with former requests LRU policy will replace lines with an higher locality. These lines will be soon then reloaded from memory causing contention. Both these two phenomena are hard to predict since they depend on a huge number of factor.

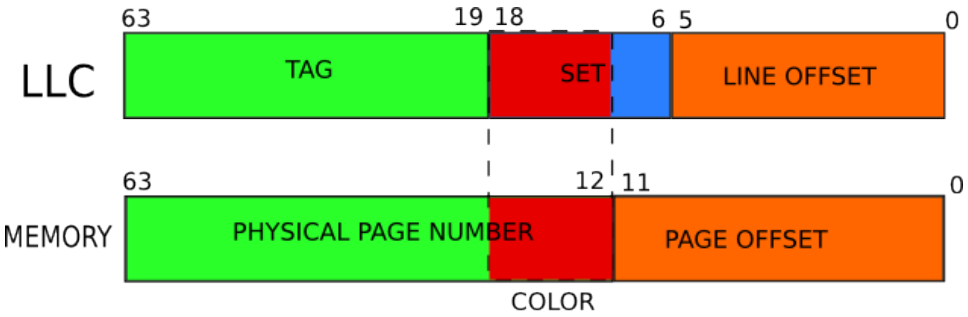

Figure 2: Graphical representation of color in word's address

2.4 The Solution: Cache Partitioning

The solution to this problem will be to partition cache through processes (in our specific case virtual machines). In other words, the solution will be to assign to distinct processes distinct areas in LLC.

The only known technique to do this without any hardware modification is called page coloring, it consists in assigning to each process one or more "color" and then allocate to it pages only if they are of one of the "colors" it owns. Colors are numbers, and exactly they are represented with the j less significant bits in page number which are the bits that are in common with the page number and the set number for LLC in this way a process will have all and only the pages that map into some given sets of cache, which means that, indirectly, we are partitioning LLC among processes. We are assigning, to distinct processes, pages in memory that will not collide in LLC. This technique has some drawback, in fact in doing this we are also limiting the maximum amount of memory we assign to a process, for that reason, it is preferable to use it for application in which processes have a limited and, preferably, known from the beginning memory footprint, that is why we decided to implement this solution in the context of virtualization in which processes are virtual machines and have a preassigned amount of memory.

3 Xen

Xen [1] is one of the most popular open-source hypervisor and is widely used in the context of cloud computing. For example, Amazon, which is probably the largest provider of cloud computing services, uses Xen to back its infrastructure.

3.1 Hypervisor, Dom0 and DomUs

Xen put it self as an underlying layer for the already present operative systems, right after the installation the previous OSs run as usual but if, from the grub menu, at boot time you decide to start your OS with Xen, now you have all the tool and services to start virtualization with Xen. In order to better understand how Xen works, it is necessary to make some distinctions and introduce some proper terminology: in Xen VMs are called Domains (dom), and essentially there are two kinds of domain dom0 and domU:

- dom0 is the guest domain from which Xen can be managed. Differently from domUs, dom0 has elevate privileges and direct control over hardware. Dom0 can run Xen control script, can create and manage new domains.
- domUs are all the other domains that you can run and create with Xen, in particular Xen also runs three specific domains (Cow, I/O and Idle) for the memory management, I/O operations, and idling.

3.2 The Old Buddy Allocator

For the realization of the page coloring I have been focusing my attention on how memory allocation is implemented in Xen. As it is, Xen allocates memory and stores information about the allocation according to a data structure called buddy; in this section I will show how currently a buddy algorithm works, then in the following section I will illustrate the modifications I gave to Xen's buddy algorithm in order to implement page coloring.

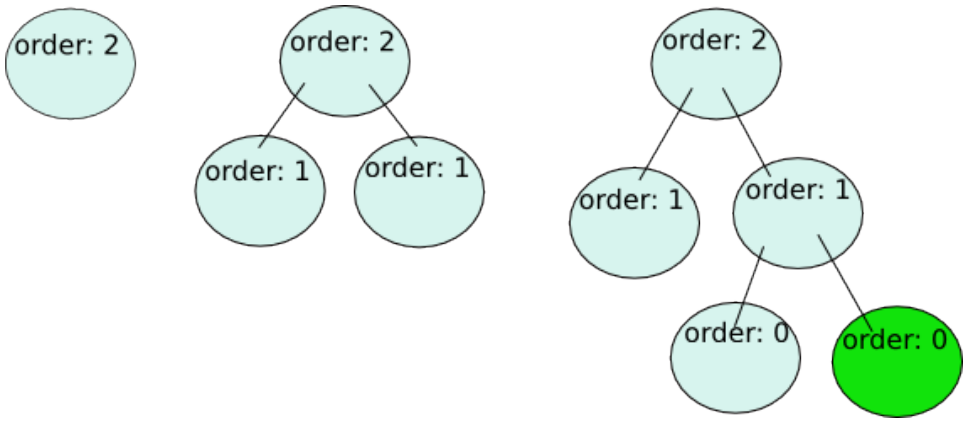

Figure 3: Example of allocation of a page in the buddy data structure

Initially no memory is allocated, so the data structure contains a single element (called buddy) representing the whole memory, this buddy belong to a list corresponding to the highest order, then when a memory request occurs the buddy is removed, split into two parts and then the first one is put in the list of buddy belonging to a lower order, instead the second part is recursively split again until the lowest order capable of containing the requested memory is reached. Orders indicate the size of the buddy according to the formula $size = 2^{order}[page]$, the lowest order is 0 and it contains buddy of one page. After that initial allocation if a memory allocation request occurs the algorithm look in the lowest order greater than the memory's area requested and if the list is not empty, it will allocate a buddy from this list to the process and remove it from the list, otherwise it will look recursively into higher order's buddy until it find a non empty list and then recursively split the first free buddies in order to allocate the one with the minimum size able to satisfy the request. When a page (or a wider area) in memory is freed by a process its buddy is added to its right order list, if its adjacent buddy is not in the list it means that is allocated to a process and nothing is done, otherwise if the adjacent buddy is in the list the two buddy are merged and then moved to the higher order, this operation is repeated recursively until there is no more possibility to merge buddies. In order to understand better how buddy algorithm works in Figure 3 you will find some graph simulating an allocation.

3.3 The New Buddy Allocator

To implement the page coloring algorithm, explained in the previous section, it is necessary to modify the buddy data structure and also the structure in which are saved information about the domains. To keep track of each color assigned to a domain I have added to the struct domain a new integer variable, representing a mask in which the bit of a certain position indicate whether or not the color corresponding to that position is assigned to this domain. Furthermore the OS store an array of length equal to the number of total color, in which it takes note about which domain has what color. If an element of the array as value equal to -1 it means that the corresponding color is free and can be assigned. When a domain is paused or destroyed the OS simply set to -1 all the element of the array having a value corresponding to the id of the domain.

In the architecture used for these experiments there are three levels of caching and the L3 cache's cache-line size is of 64 Bytes, so 6 bits are used for line offset. In x86 architecture page size is of 4 KBytes, so 12 bit are used for page offset. Since in this architecture L3 cache has 8192 sets, 7 bits are needed to represent a color (Figure 2), which means that there are 128 physical colors. In order to utilize the whole L2 cache (which is private

for cores) are necessary at least 8 physical colors so we decide to utilize the definition of virtual color (Vcolor) proposed by Scolari et al. [4].

In the buddy algorithm the change made are, simply, to add to every order a number of list equal to the number of Vcolors we are using, in particular in the specific machine I have been using for tests there are 16 Vcolors. For what concerns the highest orders of the structure (from 32 to 7, 32 is the maximum order since I am working on a machine with $12GB$ of RAM and an architecture of x86 family which typically has pages of $4KB$) nothing really changes: there are 16 lists but they are not used because all the buddies here contains pages belonging to all colors since from the 7th order up, each buddy contains at least 128 contiguous physical pages, which is equal to the number of physical colors. Instead the four lower orders use all the lists and so at the time at which a domain asks for a page in memory the buddy checks if the lists of the lowest order of the structure belonging to the colors owned by the domain are empty, if not it allocates the first free page, otherwise it proceeds as shown in the previous algorithm, having care, when splitting buddies, to put the new buddy in the right list depending on the color. In orders between 6 and 3, things are a little bit more complicated, because here buddies are greater and contains more colors (but not all) so, in order to classify buddies into the right list, colors are grouped in macro-colors, in particular: order 4 contains 8 macro-colors each ones containing two Vcolors, order 5 contains 4 macro-colors made by 4 Vcolors and order 6 contains 2 macro-colors of 8 Vcolors.

For example (Figure 4) if a domain owning color number 10 ($10_{10} = 1010_2$) asks for a page and the first free buddy is of order 7 (i.e. the maximum order), what happen is that order 7 buddy is split in 2 order 6 buddies, one belonging to macro color 0 and the other from 1, since domain has color 10 the second buddy is split again, instead the first 1 is kept in its list, now we have two new buddies of order 5 belonging one from macro color 2 and the other from macro color 3, since domain has color 10 we go on splitting the first buddy, now in order 4 we have two buddies from macro color 5 and 6, splitting the one from 5 we finally obtain two buddies of order 3, one from color 10 and one from 11, we can keep going on splitting until we find one buddy of order 0 and color 10.

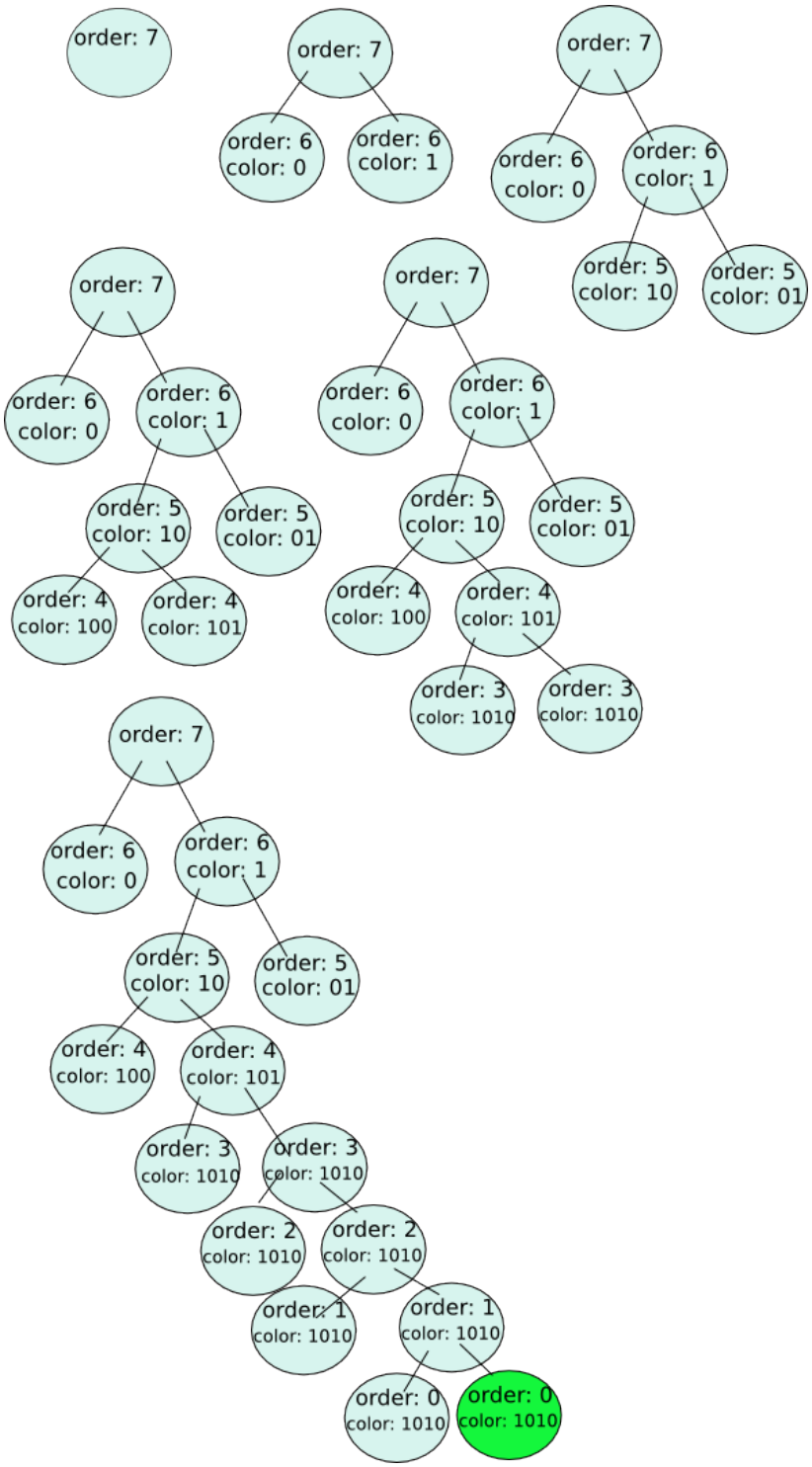

Figure 4: Example of an allocation in “buddy with color” data structure

The releasing algorithm works quite in the same way, again taking care of putting buddies in the right color list.

For reason to not put constraints upon the memory used by the hypervisor, in order to not slow down or make crash the whole system I have decided to not color page assigned to Xen and its private domains (dom0,domI/O, domCow, domIdle).

4 Experiments

After the implementation I have conduced two experiments to confirm that this implementation was working correctly

4.1 Experimental Setup

For these experiments I have been using a computer with an Intel® Core™ i5-750 processors with 4 cores and 64 bits' words, with L2 caches of $256KB$ with 8 ways per 512 sets, L3 cache of $8192KB$ with 16 ways per 8192 sets, which means that there are 128 possible physical colors and 16 virtual colors, this machine has also $12GB$ of RAM. On this machine there is installed Debian 7 with the new version of Xen compiled from the modified source of Xen-4.2.3. Upon that, for these experiments, I run two identical virtual machine with Ubuntu server 12.10 and the spec CPU2006 benchmark suite [5] installed, the first VM (called "colorato") has assigned $1GB$ of memory (the minimum requirements for running bzip2) instead to the second VM, "colorato2", I have assigned $4GB$ of memory in order to run libquantum benchmark.

4.2 Experiments

bizip2 in isolation In the first experiment I run the first VM alone on the hypervisor and running bzip2 benchmark, initially I have set Xen to assign to the VM 2 Vcolors, then I have repeated the experiments increasing the number of colors. Since previous studies [6] have demonstrated that bzip2 benchmark is the most sensitive one with respect to the quantity of cache it uses, the results expected were that there will be an exponential slow-down of the application with respect to the time occupied running using the whole L3 cache. The graph in Figure 5 shows exactly this kind of results, so here we have demonstrated that the modification adduced to Xen lead it to partition cache through VMs.

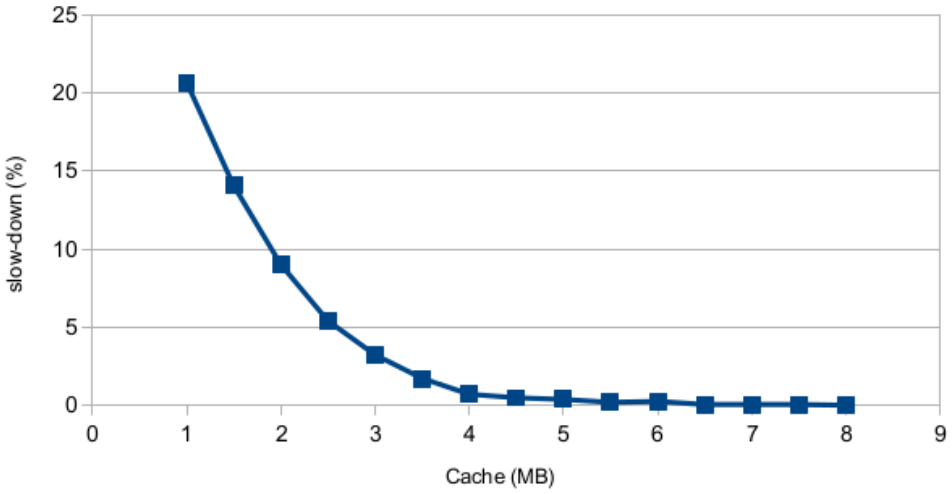

Figure 5: bizip2 in isolation

bzip2 and libquantum in colocation After having shown that this new version of Xen works as supposed, in the next experiment I have introduced a new VM running libquantum benchmark in an infinite loop, libquantum is not sensitive to the quantity of cache it uses and so there is nothing to notice on it, but it will decrease bzip2 performance. Results are shown in the graph in Figure 6, the straight yellow line indicates the slow-down obtained when both colorato and colorato2 uses the cache without any partition, here bzip2 has a 20% of slow-down due to presence of libquantum. The dashed line represents results obtained in the previous experiment, now our curve (the red one) has the same shape as before, but there is a little delta between the two curves, as shown in the second graph this delta decreases until he reaches a minimum and then became constant. This additional slow-down is due to the presence of libquantum and the contention on the bandwidth of lines connecting RAM to LLC and LLC to cores, in fact the more cache bzip2 has the less it access memory the less is the delta, until the cache it has got equals the dimension of bizip2's working set, now access through memory are independent form cache dimensions and so delta is constant, because the memory is accessed by bizip2 only in the initial warm-up of LLC and then the only reason of conflict with libquantum are lines connecting L3 cache with CPU.

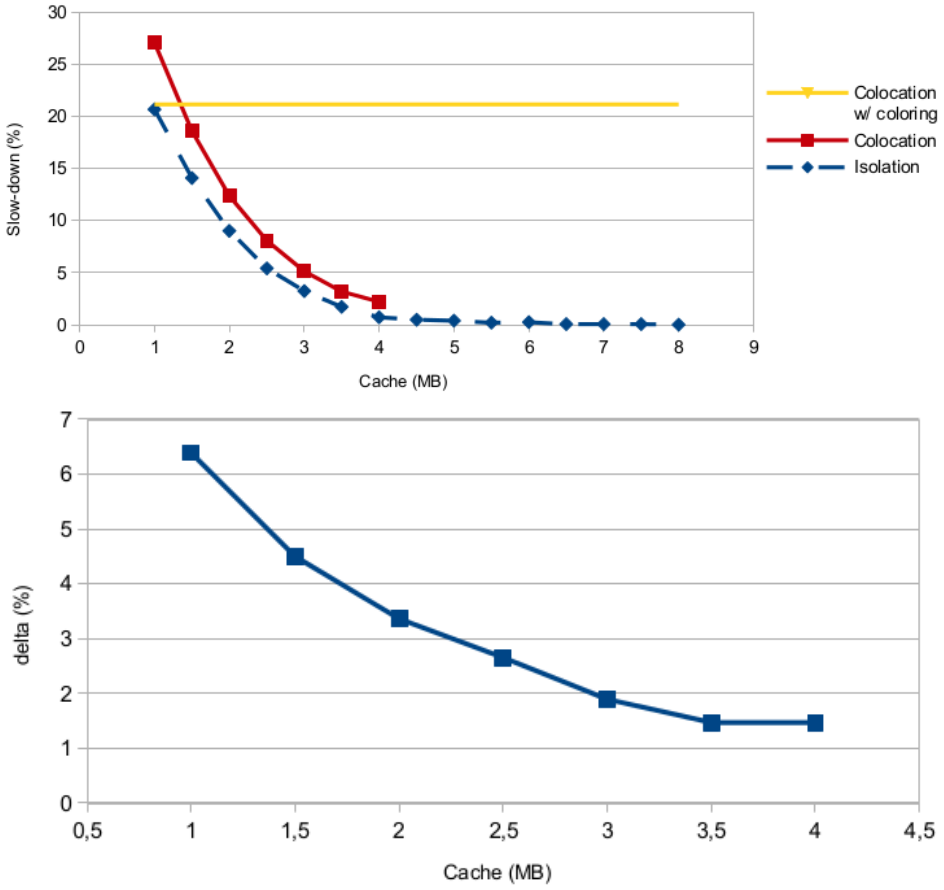

Figure 6: bizip2 in colocation with libquantum

5 Related works

In page coloring technique and more in general in the field of software techniques devoted to cache partitioning a lot of researches and projects have been conduced, since these techniques involve in realization only changes and modification to the OS without touching the physical asset of the computer and so can be deployed easily and cheaply.

In particular, a study conduced by Tam et al. [7] have demonstrated the effectiveness of static partitioning, they profile applications looking at their Miss Rate Curve and Stall Rate Curve in order to find the cache slice to obtain the best performance. Xinxin et al. [6] have already shown in a more elaborated article, that VMs running in Xen environments could benefit from cache partitioning. In particular is remarkable the fact that results obtained for bzip2 benchmark running in isolation are very similar to results obtained in this relation.

Since static page coloring strongly constraints memory in applications, it could not, really, be applied to application which have more variable memory footprint, so a lot of work has been done on dynamic page coloring [8] and techniques able to guarantee some more elasticity. In particular two approaches have shown nice results: "hot page coloring" [9] and "ROCS" [10]. The first one, "hot page coloring", relies on the hardware support that modern CMPs provide, which allows the operating system to periodically look at the list of pages in memory and see what are the pages that generate more miss and replacement and so more performance degradation, then these pages will be recolored in order to resolve these conflicts, instead "cold" pages are left free to be allocated by every applications without constraints. A, somehow, opposite approach has been exploited in "ROCS" (Run-time Operating system Cache-filtering Service) to enhance performance of a single application running in isolation, in fact ROCS track the pages that show a bad cache behavior (continuous miss and replacements without any benefits) and confine them in an area of the cache called pollution buffer in order to not make them interfere with other page that will behave well.

6 Possible future works

The goal of this project was to provide a functional implementation of page coloring within the Xen hypervisor. However, the proposed implementation requires a few refinements to improve its usability.

For example, the assignment of virtual colors to virtual machine requires re-compiling the Xen hypervisor; a first refinement consists in extending the

xen-tools [11] to support the configuration of page coloring.

Furthermore, some work could be done to try to implement some of the already seen techniques for dynamic recoloring in order to make this framework more elastic and easy to use in real scenarios. Also, recently has been shown a new way that will permit to partition, by means of software, the use of bandwidth to access memory [12]; I think that adding this new feature at this project will add more predictability to performance, in fact as shown in the second experiment bandwidth is one of the most influential causes of contention left among applications running concurrently.

7 Appendix

In this appendix I will go deep into details of the configurations of the machine used in the previous experiments; since I got some trouble in setting up all and make it work, and since there is not a lot of documentation about it, I think that this appendix would be very useful for anyone willing to repeat or improve these experiments.

First of all Xen as to be compiled and installed from source following the guide from the site [13] which is well documented (remembering that 32bits architecture are no longer supported).

In order to achieve better results I suggest to limit the amount of memory assigned to dom0, automatically Xen assign to it all the memory in order to release it gradually to all the new domains, this procedure will create some trouble using colors. So you will need to deactivate autoballooning [14] from $/etc/xen/xl.conf$ and then set Xen grub2 configurations to alloc a fixed amount of memory to dom0, this has to be done adding the line "$GRUB\_CMDLINE\_XEN\_DEFAULT =$ "$dom0\_mem = min : 512M, max : 512M$"" to $/etc/default/grub$, then update the grub and reboot the machine.

Now create a virtual machine is very easy using xen-tools [11], but since we are using Xen compiled from source we could not install it from the standard repository, otherwise we will install also the standard hypervisor overwriting the one we made, so you need to install it from source following the guide on the site. Once you have created the first VM, in order to clone it, is simply necessary to copy the two files representing disk and swap and copy the config file making it point to the new disks and changing the mac address in order to avoid conflicts.

Since, as I have already said, the number of colors we assign to a VM is hard-coded, every time I wanted to change this configuration I needed to change the source code, in particular the file $./xen/common/domain.c$ which contains all the functions used to instantiate and launch a new domain (here all paths and command are launched from the project local directory) then recompile Xen ($make -j8\, dist - xen$), re-install it ($make -j8\, install\, dist - xen$) and reboot to see modification effectively applied. For this project I have also been modifying the file $./xen/common/page_alloc.c$ which contains the buddy data structure.

Xen, from dom0, has also a dmesg utility ($xldmesg$) so with some $printk()$ placed after every page allocation and with some *grep* on dmesg output I was able to tell if the buddy was working right assigning colors. I have also observed some peculiar behaviour of Xen, in fact, when a new domain is created, Xen first allocates to it its assigned memory to see if it will have the possibility to work well, if not it fails in the creation of the new domain; the second peculiarity is that Xen hypervisor allocates to its self a large amount of memory at boot time taking also 6th orders buddies, which means that if in a near future someone likes to color also the memory allocated to Xen its self, this will be an hard task.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 164–177, Oct. 2003.

[2] Amazon Inc., "Amazon ec2." `http://aws.amazon.com/ec2/`.

[3] P. J. Denning, "Thrashing: Its causes and prevention," in *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), (New York, NY, USA), pp. 915–922, ACM, 1968.

[4] A. Scolari, F. Sironi, D. B. Bartolini, D. Sciuto, and M. D. Santambrogio, "Coloring the cloud for predictable performance," SoCC'13, 2013.

[5] Spec cpu2006, "Spec cpu2006." `http://www.spec.org/cpu2006/`.

[6] X. Jin, H. Chen, X. Wang, Z. Wang, X. Wen, Y. Luo, and X. Li, "A simple cache partitioning approach in a virtualized environment.," in *ISPA*, pp. 519–524, IEEE, 2009.

[7] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing shared l2 caches on multicore systems in software," in *In Proc. of the Workshop on*

the Interaction between Operating Systems and Computer Architecture (WIOSCA), 2007.

[8] X. Wang, X. Wen, Y. Li, Y. Luo, X. Li, and Z. Wang, "A dynamic cache partitioning mechanism under virtualization environment.," in *TrustCom* (G. Min, Y. Wu, L. C. Liu, X. Jin, S. A. Jarvis, and A. Y. Al-Dubai, eds.), pp. 1907–1911, IEEE Computer Society, 2012.

[9] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multi-core cache management," 2009.

[10] L. Soares, D. Tam, and M. Stumm, "Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, (Washington, DC, USA), pp. 258–269, IEEE Computer Society, 2008.

[11] xen-tools, "xen-tools." `http://xen-tools.org/software/xen-tools/`.

[12] M. Caccamo, R. Pellizzoni, L. Sha, G. Yao, and H. Yun, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Proceedings of the 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, RTAS '13, (Washington, DC, USA), pp. 55–64, IEEE Computer Society, 2013.

[13] Xen developers comunity, "Compiling xen from source." `http://wiki.xen.org/wiki/Compiling_Xen_From_Source`.

[14] Xen developers comunity, "Xen best practices." `http://wiki.xen.org/wiki/Xen_Best_Practices`.