Towards Efficient Working Set Estimations in Virtual Machines

Davidlohr Bueso, Elisa Heymann, Miquel Angel Senar[1]

Abstract— **During the last decade, virtualization technology has become increasingly important, having the potential to optimize the usability of both high performance and high throughput computing as well as server consolidation by reducing operational costs and facilitating green computing. Furthermore, this technology is one of the key factors that have allowed the growth of cloud computing and its adoption over the Internet, in what is nowadays, a software as a service oriented industry. Virtualization, by multiplexing the physical hardware resources, enables multiple, independent, operating systems and applications to run on a single machine. In order to make this efficient and acceptable to users, the physical resources of a computer must be well managed, being the system's main memory one crucial factor.**
This paper analyzes, under a unified criteria, current alternatives to estimating working set sizes in virtual machines, focusing on Least Recently Used (LRU) miss rate curves. We also present a generic framework that serves as a potential testbed for comparing different stack distance algorithms and page monitoring policies. Finally, we propose applying different stack distance algorithms to virtual machine Working Set Size (WSS) estimations.

Keywords— **virtual machines, virtualization, memory management, hypervisors, memory overcommitment, miss rate curves, stack distances.**

I. Introduction

Virtualization technology, introduced by IBM in the mid 1960s, is not a new concept, yet it is only in the past few years that it has become a hot topic in academia and industry research, after VMWare released the first commercial x86 virtualization product in 1999. Since then, this technology has become increasingly important and plays an important role in computer science. The term virtualization can be ambiguous, as it is widely used in many different contexts, referring to emulation, operating system (OS) containers and system virtualization. We adopt the later meaning, where, through the use of hypervisors, or Virtual Machine Monitors (VMM - used interchangeably in this document), physical hardware is multiplexed to allow multiple, independent, OS and userspace applications to run simultaneously. This has been beneficial to the IT industry in many ways, allowing easier systems development (by deploying and testing), administration and maintainability, while at the same time, greener and less expensive data centers. In order to make virtualization efficient and acceptable to users, the physical resources of a computer must be well managed. It is the responsibility of the VMM to present to its guest virtual machines (VM) the illusion of a non virtualized computer system, while satisfying Popek and Goldberg's requirements [1]. All these benefits come, nonetheless, with a performance cost when compared to a non virtualized environment. This cost is the overhead caused by the *virtual layer*. Current research efforts aim at minimizing this overhead, as user acceptance depends on it.

From a von Newmann perspective, while CPU, device IO and main memory can all be virtualized, it is memory that is least amenable to multiplexing [2]. The difficulties of memory management at a hypervisor level can be seen as (i) address translation overhead, and (ii) memory overcommitment, or overbooking. Although they are not directly related to each other, they both have a profound impact on the performance and usability of virtual machines. This paper deals with challenges that come when overcommiting memory. When running under a virtual machine, access to the physical memory must be limited to only the VMM, otherwise, if a guest kernel does so there is no guarantee against data corruption and guests crashing each other. A guest OS kernel has the same vision of virtual memory as it would in a traditional environment, yet in reality the VMM acts as an intermediate between it and the physical resources.

The rest of this paper is organized as follows: Section II explains basic concepts of memory overbooking and the challenges that arise when balancing memory among virtual machines. In Section III, we go through recent research and state of the art on estimating memory demands in virtual environments, focusing specifically on LRU based miss rate curves. Section IV introduces our KVM based framework for trapping guest memory accesses and analyzing them both online and offline. Then, in Section V we propose adding well known LRU miss rate algorithms into the framework and what we expect to gain from this when compared to current alternatives. Finally, Section VI concludes our work.

II. VMM Memory Overcommitment

Overcommiting a resource is creating the illusion that this resource is actually more available than it really is. In the case of virtualization, it is quite common for the sum of memory assigned to all virtual machines (VMs) to be larger than the physical RAM the host possesses, as illustrated in Figure 1. Today most, if not all, important hypervisors - Xen, KVM, VMWare - deal with overcommiting memory.

[1]Computer Architecture and Operating Systems Department (CAOS), Universitat Autonoma de Barcelona; e-mail: `dbueso@caos.uab.es`, `elisa.heymann@uab.es`, `miquelangel.senar@uab.es`

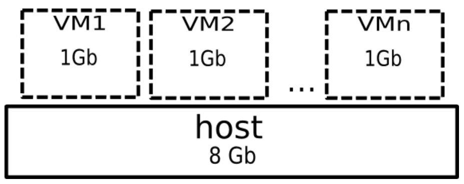

Fig. 1. Overcommiting memory for n virtual machines, $n > 8$

There are two common techniques [3] that allow memory overcommitment, and have been applied, in one way or another, to commonly used VMMs. The first method takes advantage of the fact that different virtual machines are possibly running the same tasks throughout part of their lifetime, for example: checking email, editing documents and browsing the Internet. By detecting similar pages, based on their content, virtual machines can share a single copy and therefore save frames for other purposes. When a page is modified, the copy is broken with standard copy-on-write techniques.

The second technique uses a special, *balloon driver*, that enables the VMM to give or take away physical memory from its virtual machine guests. It is commonly implemented as a kernel space driver located within the hypervisor and the guests. Further, it does not require any downtime, being done dynamically on the fly. The host can give the guest's balloon instructions to expand or shrink to a certain size; the guest cooperates but does not directly control the balloon. When the host wants to reclaim memory from the guest, it is said to inflate the balloon, by allocating physical pages within the guest's balloon. This balloon expansion makes the guest have less memory and forces its natural way of dealing with this situation: if enough free memory, then return a portion of this, otherwise it must swap from disk. The pages absorbed by the balloon can then be released back to the host system which, presumably, has a more pressing need for them elsewhere. Letting *air* out of the balloon makes memory available to the guest once again.

The primary reason for ballooning is to better use memory that would otherwise be wasted by a virtual machine. Operating systems are in charge of maintaining information about used and available memory, so when running on a virtual machine, the host system has no way of knowing or accessing this information. This implies that when an application running on a VM frees memory, only the guest OS will know about it, yet the host OS will never actually free the corresponding frame(s), thus keeping idle or stale data. For example, if a guest normally uses 300Mb of memory, but for some small amount of time it peaked at 900Mb, there will be 600Mb of underutilized memory. Ballooning by itself does not know about guest memory needs [4] and, therefore, cannot determine how it will impact the guest when it reclaims memory from it.

This leads to a *memory balancing* problem, where the idea is to reduce the maximum number of resources required for virtual machines, thus getting the most work done at the lowest cost in ever-changing environments. This allows physical frames to be distributed efficiently among VMs, and can address concerns such as: (i) If a VM has more memory assigned to it than it actually needs, and on the same machine, another VM has a high swapping rate, indicating it has a memory pressure, transfer some memory among the machines to balance the load and so both can provide a better quality of service. (ii) If two or more VMs have similar workloads and are all under memory pressure, one can have more priority over the others, and thus its more important for it to get more memory. Therefore the other machines can give up their memory rights, favoring the most needed VM.

This can be seen as a scheduling problem, where the physical memory must be distributed efficiently among guest virtual machines; much like what a CPU scheduler is to processes in an OS. However, certain problems come with this:

1. The future of memory demands of a virtual machine workload can only be estimated. If a VM is capable of giving some of its memory, it cannot demand it back immediately.
2. The benefits of performance of each additional frame that is moved to/from a virtual machine must be quantifiable in order to understand the performance and cost correlation.
3. The scheduling overhead must be maintained at a minimum; the cost of moving capacity efficiently must be measurable to overall VM impact on performance.

Memory balancing among virtual machines is traditionally divided into two phases: (i) an estimator of VM demands and (ii) a balancer that, through policies, will decide what actions to take based on the information provided by the estimator. Ballooning is a typical way of moving memory once the correct amount is known. We focus on the first phase, in which by knowing the working set size of each VM, the hypervisor can dynamically reallocate an appropriate amount of memory to each guest [5].

III. Related Work

The active working set of an application refers to the set of pages that it has referenced during the recent working set window. Knowing the WSS enables memory resources to be utilized more efficiently. During recent years there have been various studies on obtaining the memory demands of virtual machines.

A. OS Exported Information

By monitoring a system's swap and disk IO activity, memory pressures can be inferred [5] [6], with some accuracy, through black box monitoring techniques. Alternatively, gray box strategies [6] use information exported from the OS in the virtual machine to estimate its memory behavior. For example, the */proc* filesystem on Unix guests can be used [7] to easily obtain an accurate estimate. These mechanisms have the advantage of being very simple to implement, imply a low monitoring overhead and are noninvasive to the host, VMM or guests - a simple userspace daemon can monitor the needed file(s). Despite these attractive benefits the major problem is that there is no way of quantifying how a virtual machine will react when its reallocating memory, and therefore does not address problem 2 (previous section), thus making it only a reactive measurement. For example, when a guest's working set size is small enough to give some of its memory, there's no way of knowing the correct amount that would maintain its current level of service. Furthermore, due to security reasons, like the lack of trust between the hypervisor and a virtual machine, it is better to let the VMM collect information about VM memory usage and demands, instead of the guests [8].

B. Statistical Sampling

VMWare implements a statistical sampling approach [3] at the hypervisor level to obtain VM working set estimates without any guest involvement. During a configurable window of time, or sampling period, a random set of n pages, accessed by the virtual machine, are monitored. Each time a page in the monitored set is accessed, a counter t is incremented. When the sampling period is over, the memory that was actively used can be computed from $f = t/n$, and therefore the page usage by the VM will always be a fraction of the total memory utilization. As with black-box and gray-box techniques, mentioned above, statistical sampling cannot induce growth or shrinkage of WSS [4] [5] [9] when it is below its full allocation capacities.

C. LRU Stack Distances

Perhaps the most studied alternative for obtaining memory demands of a VM is LRU stack distances. Being a very common way of calculating Miss Rate Curves (MRC), it has been applied in many fields of computer science, including file systems, compilers, caching systems and memory management in virtual machines, among others. We have conducted experiments to verify MRCs for virtual machines and generated data that support the referenced literature. The miss ratio curve plots the page miss (page fault) ratio against varying amounts of available memory and provides a better correlation between memory allocation and system performance [10]. This addresses problem 2 (from the previous section) as we can quantify performance vs cost. Figure 2 shows the resulting MRC for an experiment running a Linux 3.0 kernel build on a Linux guest VM. Apart from traditional system daemons, no other program was being executed at the time, so *gcc* was the main running process on the system. In this case it can be seen that as more pages are allocated, the miss rate decreases, which is expected. Considering a standard 4 Kib page size, when 50000 pages are used, there's an acceptable miss rate below 10% with around 195.3 Mib of memory. From this point on, it is worth considering not to allocate any more memory for the guest, as most of its working set is already resident in memory.

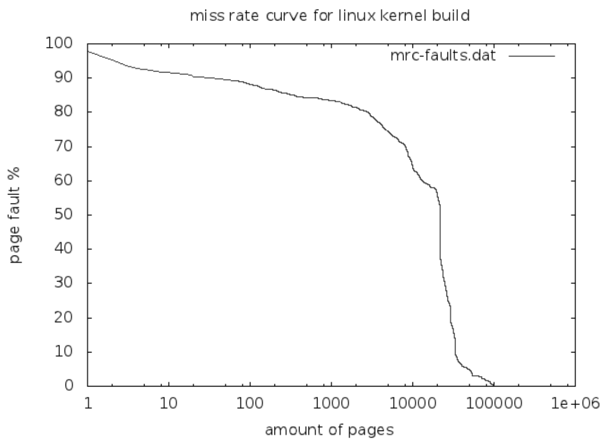

Fig. 2. MRC for VM running a Linux 3.0 kernel compilation

In order to create an accurate MRC, enough pages must be sampled to make it effective and capable of describing the general behavior of the system, and thus memory IO must be intercepted rather frequently. When capturing pages at an OS level, its important to notice that because of architecture semantics, it is only possible to capture the ones triggered by cache misses, as these are the ones the OS has control over when a page walk begins. The VMM, however, has more control over its guests pages and memory accesses, specially with software virtualized MMUs, and can revoke access permission of pages, so the next accesses to those pages will cause hidden page faults. This way, the page fault handling mechanism can be extended to analyze the faults and build the MRC, much like the statistical sampling method. By modifying the hypervisor, the guest OS kernels need not be altered and the whole process is totally transparent. It is also crucially important to keep the interception of memory access with the lowest possible overhead (problem 3 in the previous section), otherwise the trade off between it and performance optimization might not be worth it. Minimizing overhead sacrifices accuracy, as less pages can be monitored, and vice versa. Additionally, Zhou et al. [11] mention a concern about the monitoring space overhead, since tracking entire address spaces of processes can demand large amounts of memory. In the case of virtual machines, process address spaces within them are irrelevant, as a VM is considered a single entity - it could, however, incur in larger monitoring space requirements if there are a large number of VMs running.

Since most operating systems use an LRU based page eviction policy, Mattson stack algorithm [12] comes as a natural approach. A stack data type is used to store accesses of pages. Since stacks are a type of LIFO, the entries will be sorted with more recently used pages on the top. Each time a page is accessed, it is searched for through the list to obtain its distance; so if the page is found in the ith element of the stack, distance i is incremented, otherwise if it is a first time reference, the distance ∞ is increased instead. The next step adds the newly referenced page to the top of the stack to represent it is the newest access. The most computationally expensive operation of the entire algorithm is searching for pages in the stack, and a naive implementation will sequentially iterate over the list with $O(n)$ worst case complexity - n being the amount of pages in the stack. Although most programs have a good degree of locality and access memory within a small range of addresses, new references and access to old data when dealing with large programs can de gradate performance significantly [13]. For example, from the experiment described previously, monitoring frame accesses from a Linux kernel build on a virtual machine, there were around 1.4 million references, of those, 83% were only accessed once. Based on these factors, it makes sense not to monitor all accessed and only care about those pages that are under demand by the workload.

Zhao et al. [4] implement MEB (MEmory Balancer), which instead of trapping all memory accesses, divide all pages into hot and cold sets, and only monitor accesses to pages belonging to the cold set. Initially all pages are marked cold and as they are accessed they turn hot, since subsequent accesses to hot pages aren't monitored, they will not incur in any overhead. In order to assure accuracy, the hot page set must be limited to a certain size, and therefore when the amount of hot pages reach the limit, the oldest reference is popped with FIFO semantics and reintroduced into the cold set. Since stack data types are used, this approach incurs in a linear overhead as the amount of pages increase.
Pin et al. [8] introduce hypervisor exclusive caches that also use LRU miss rate curves to obtain VM working set sizes, in which guests are given a small amount of memory, while the rest is managed in the form of a cache. It traps all page evictions in a virtual machine and maintains a stack to obtain the MRC, and therefore it suffers from the same overhead issues mentioned above. Furthermore, it requires modifications to the guest OS and therefore sacrifices flexibility, and at the same time, by adding a new layer of memory management, it comes as a complex alternative for memory balancing.
By turning off presence bits in Intel hardware MMU page tables (EPT), Min et al. [9] transparently monitor guest memory accesses at the VMM level. Similarly to MEB, instead of monitoring all pages, they are divided into hot, warm and cold lists, where only references to pages in the last two sets are trapped. To calculate the stack distances efficiently, a weighted red-black tree is used and logarithmic complexity is much more attractive than iterating linearly. This approach is very promising, reaching similar cases to unmonitored cases, yet the stack distance overhead still composes nearly 40% of the entire overhead - including monitoring policies and guest memory reallocation for all workloads. Another drawback is that it depends entirely on Intel specific hardware extensions, and for workloads with bad locality or that require many context switches, is not the best choice when compared to shadow pages. This problem is known as two dimensional page walks that hardware memory management require for address translations.

IV. MRC Generation Framework for KVM

We have developed a testing framework that extends KVM, the Linux Kernel Virtual Machine, to transparently trap guest accesses to host physical frames. Although any algorithm can be applied, by default, it uses the naive implementation to calculate stack distances and build miss rate curves. Additionally, through tracing, entire VM memory accesses can be obtained for offline analysis. Although offline analysis is beyond the scope of this paper, it's worth mentioning some useful applications that can be potentially exploited in the future. The first is studying the impact of page replacement policies done by the host - in this case, with KVM, it will always be Linux. Since the policy will have a significant impact on the miss rate (page fault) and therefore overall performance, it is interesting to compare it to optimal algorithms [14], which requires the entire reference trace. Another application is to verify the correctness of online generated miss rate curves. Since the same algorithm(s) can be applied for both online and offline mechanisms, we can verify that they match in each case. Errors in estimating working set sizes can produce catastrophic decisions in VM memory balancing. Alternatively, virtual machine behavior characterization can inferred by knowing its memory access patterns, such as predicting locality [15], swap IO distribution and memory latency. Similar studies have been conducted on VMs analyzing disk IO for characterizing workloads [16] [17]. Obviously, when doing offline analysis, we do not care about performance or overhead, and can use any algorithm to monitor all memory accesses from the virtual machine.

A. Framework Architecture

As shown in Figure 3, the framework traps memory accesses from guest virtual machines for both shadow page tables and hardware nested pages by turning off the presence bits in each. Since we intercept them at both levels, no special configurations are required depending on the virtual MMU mechanism used by the hypervisor. Once the physical frame number is trapped, it is fed to the LRU algorithm(s) that in turn calculate the stack distances and can build MRCs at any point in time, through sampling windows. Several LRU algorithms can coexist and be used simultaneously to generate similar curves. This can be dynamically configured at runtime through standard Linux kernel module *sysfs* interfaces.

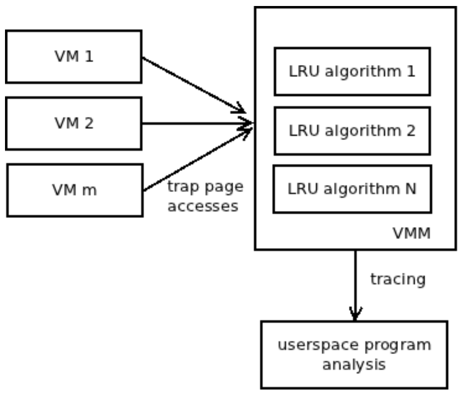

Fig. 3. KVM MRC Generation Framework Architecture

Additionally, at any point, tracing can be enabled to report memory access traces for offline analysis. We are currently working on a generic way of adding policies to determine which pages to monitor, applying similar concepts of separating hot and cold sets. This does not limit, however, comparing different algorithms between each other, but we do expect the overhead to gradually decrease since we currently monitor all references.

B. Spacial Requirements

As described in section III, although the memory footprint required to monitor VM page accesses is not of a crucial factor, we do attempt to make it as small as possible. Since the data structures will depend on the algorithm we cannot provide a single measurement for all cases, but the size S_i for virtual machine i currently monitoring n pages will naturally depend on:

$$S_i = n * sizeof(structure) \tag{1}$$

Furthermore, the entire memory T used by the framework, monitoring m virtual machines, will be given by:

$$T = \sum_{i=o}^{m} S_i \tag{2}$$

For example, we have implemented the naive method for storing page accesses as a linked list, like showed in Figure 4, which data structure uses 24 bytes per monitored frame. Running a single virtual machine, with a kernel build, having accessed 130051 different frames, will cost a little over 3048Kb of overhead on a x86-64 host architecture.

```
/* LRU—stack */
struct stack_list {
        pfn_t pfn; /* unsigned long long */
        struct list_head list;
};
```

Fig. 4. Stack data structure - uses 24 bytes on x86

Linked lists are simple structures that, when applied as a stack, automatically have the *notion* of access times. Other data types, such as trees and hash tables, do not have this advantage, and therefore require extra fields in their structures to incorporate this notion. This will naturally increase the monitoring space requirements.

C. Other Considerations

Since KVM, and therefore our framework, runs in kernel space, access to global data will need synchronization mechanisms. Locks can easily become a performance bottleneck as the number of threads increases, so we cannot take this matter lightly. While simple counters can use atomic operations to increment and decrement, accessing data on the stack, like pushing new references to the top and calculating stack distances, must be guarded more carefully. We currently use spinlocks for this purpose, however, we are considering using Read-Copy Update (RCU) instead. This mechanism allows multiple readers and writers to concurrently access data structures and can therefore provide better performance for the implemented algorithms, specially when incurring in expensive stack distance calculations.

V. New Approaches to WSS Estimations in VMs

It has been demonstrated that, until now, the best way for estimating a virtual machine's memory demands are through LRU miss rate curves, mainly due to the following reasons:

1. It provides information about a VM's memory demands with both growing and shrinking working set sizes.
2. It can quantify cost vs performance.
3. If applied to the hypervisor as a monitor it is transparent to guest operating systems, and therefore does not sacrifice flexibility.
4. It has proven [4] to be more accurate than statistical sampling.

There are, nonetheless, important drawbacks to this approach. The first is determining a good policy for estimating which pages to monitor. As we have demonstrated, when tracking all memory accesses it incurs in negligible accuracy, and therefore the extra monitoring overhead can be spared. Secondly, the data structures will play a paramount role when computing the stack distance as linear complexity does not scale and can significantly degrade overall performance. Although optimizing stack distance computation as been broadly studied [13] [18] [19] in the past, very few alternatives have been applied to virtual machine WSS estimation.

We plan on implementing popular algorithms such as binomial trees, Bennett & Kruskal and hole based trees as in our framework and directly compare them with the results of the weighted red-black tree, described in section III. We expect to see improvements in performance and lower overhead in some algorithms than what is currently available.

VI. Conclusions and Future Work

Today, virtualization technology is an important player in computer science, providing multiple benefits to the IT industry. In order to continue and improve its acceptance, physical hardware resources must be used be as efficiently as possible. One of these areas is overcommiting memory so that more VMs can be hosted on the same machine and make better use of idle memory in the guests. Each virtual machine should have only the amount of memory it needs to provide a good QoS, this way memory can be assigned or reclaimed from it as the hypervisor sees fit. LRU miss rate curves are a good method for computing virtual machine memory demands as they provide information about how the VM's performance will impact when reallocating some of its memory.

This paper analyzes, under a unified criteria, current alternatives to estimating working set sizes in virtual machines, focusing on LRU miss rate curves. We also present a generic framework that serves as a potential testbed for comparing different algorithms and monitoring policies. Based on this framework, our next step will focus on implementing more LRU miss rate curve algorithms and formally benchmark them with current implementations. With the information that the WSS provides, informed decisions can be taken regarding guest memory reallocation, for example, by giving or reclaiming physical frames through ballooning techniques. This will make better use of the machine's physical memory and use it where it is needed, instead of underutilizing it.

Acknowledgments

This research has been supported by the MICINN Spain, under contract TIN2011-28689-C02-01.

References

[1] Gerald J. Popek and Robert P. Goldberg. *Formal requirements for virtualizable third generation architectures*, SIGOPS Oper. Syst. Rev. 7, 4, July 1974.

[2] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. *Difference engine: harnessing memory redundancy in virtual machines*, Commun. ACM 53, 85-93, October 2010.

[3] Waldspurger, Carl A., *Memory resource management in VMware ESX server*, SIGOPS Oper. Syst. Rev. 36, 181-194, December 2002.

[4] Zhao, Weiming and Wang, Zhenlin, *Dynamic memory balancing for virtual machines*, Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, 21-30, 2009.

[5] Jones, Stephen T. and Arpaci-Dusseau, Andrea C. and Arpaci-Dusseau, Remzi H. *Geiger: monitoring the buffer cache in a virtual machine environment*, SIGARCH Comput. Archit. October 2006.

[6] Wood, Timothy and Shenoy, Prashant and Venkataramani, Arun and Yousif, Mazin *Black-box and gray-box strategies for virtual machine migration*, Proceedings of the 4th USENIX conference on Networked systems design and implementation. 2007.

[7] Magenheimer, Dan. *Memory Overcommit...without the commitment*, Xen Summit, June 2008.

[8] Lu, Pin and Shen, Kai, *Virtual machine memory access tracing with hypervisor exclusive cache*, 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical.

[9] Changwoo Min and Inhyuk Kim and Taehyoung Kim and Young Ik Eom, *Hardware assisted dynamic memory balancing in virtual machines*, IEICE Electronics Express, April 2011.

[10] Weiming Zhao and Xinxin Jin and Zhenlin Wang and Xiaolin Wang and Yingwei Luo and Xiaoming Li, *Efficient LRU-Based Working Set Size Tracking*, Michigan Technological University Computer Science Technical Report. March 2011.

[11] Zhou, Pin and Pandey, Vivek and Sundaresan, Jagadeesan and Raghuraman, Anand and Zhou, Yuanyuan and Kumar, Sanjeev, *Dynamic tracking of page miss ratio curve for memory management*, SIGOPS Operating. Systems. Rev. October 2004.

[12] Mattson, R. L. and Gecsei, J. and Slutz, D. R. and Traiger, I. L., *Evaluation techniques for storage hierarchies*, IBM Systems Journal, June 1970.

[13] Almási, George and Caşcaval, Călin and Padua, David, *Calculating stack distances efficiently*, Proceedings of the 2002 workshop on Memory system performance.

[14] Belady, L. A. *A study of replacement algorithms for a virtual-storage computer*, IBM Syst. J. June 1966.

[15] Chan Ding and Yutao Zhong *Predicting whole-program locality through reuse distance analysis*, Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation. May 2003.

[16] Irfan Ahmad, *Easy and Efficient Disk I/O Workload Characterization in VMware ESX Server*, Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization.

[17] Azmandian, F., *Workload Characterization at the Virtualization Layer*, Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on. July 2011.

[18] Bennett, B. T. and Kruskal, V. J., *LRU stack processing*, IBM J. Res. Dev. July 1975.

[19] Hill, M. D. and Smith, A. J. *Evaluating Associativity in CPU Caches*, IEEE Trans. Comput. December 1989.