VOL. 12, NO. 9, MAY 2017 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2017 Asian Research Publishing Network (ARPN). All rights reserved.
www.arpnjournals.com

ENHANCING THE CONCERT OF LIVE MIGRATION OF VIRTUAL MACHINE'S WITH LAZY COUNTING–BASED SPLAY TREE ALGORITHM

K. Santhi[1],T. Chellatamilan[2] and G. Zayaraz[3]
[1]School of Computing Science and Engineering, VIT University, Vellore, India
[2]Department of Computer Science Engineering, Arunai Engineering College, Thiruvannamalai, India
[3]Department of Computer Science Engineering, Pondicherry Engineering College, Puducherry, India
E-Mail: santhikrishnan@vit.ac.in

ABSTRACT

Live migration an advancement with which the whole running virtual machine (VM) is relocated beginning with one physical machine then onto the next. Migration at the level of the whole VM suggests that in-memory state can be moved in a reliable and effective design. Migrating operating system instances across distinct physical hosts is a beneficial tool for clusters and administrators. Along these outlines, a significant issue in live migration is the complete migration time and the downtime. To enrich the concert of live migration, an optimized iterative pre-copy procedure is used to decrease the dirty rate of virtual machines. In pre-copy approach that is essentially utilized as a part of live migration, total migration time, which influence on the performance of VM, is delayed by iterative copy operations and the noteworthy measure of transferring data during the entire migration process. In this paper, we presented a system that incorporates pre-processing phase in traditional pre-copy based live migration for decreasing the amount of transferred data. In pre-processing stage, we recommend the prediction working set algorithm to show that complete migration time as well as downtime is controlled by specific memory utilization patterns. Applying the proposed lazy counting based splay tree algorithm, the system can diminish the amount transferred memory page. The lazy counting-based splay algorithm along with traditional pre-copy approach that contains a pre-processing phase for diminishing the amount of transferring memory page and total migration time.

Keywords: downtime, dirty page, live migration, total migration time, working set, virtual machine.

1. INTRODUCTION

Virtualization is gaining progressively attention within the high-performance computing world. Virtualization technology provide by better utilizing computing resources, agile capacity, reliability, workload migration, availability while bringing down the total cost of ownership and expanded disaster recovery options. Migrating operating system instances across distinct physical hosts is a most critical elements of virtualization technology. It permits a flawless separation between hardware and software, and enables fault management.

Live migration of VMs may be a treasured facility of virtualized clusters and information centers. It must be possible by carrying out while the operating system is as yet running. It permits more flexible management of available physical resources by making to load balance and do infrastructure maintenance without wholly compromising the application availability and responsiveness. VM migration is relied upon to be fast and VM service degradation is also expected to be minimal during migration.There are several methods for live migration that tradeoff two important constraints i) total migration time, the duration between the start of migration and the time when migrated VM starts to run in destination and informs that the source host can be disabled. ii) Downtime, the time interval from when migrated VM is deferred on the source node when it is restarted on the destination node in the previous migration phase.

The important challenge is to accomplish noteworthy performance with negligible service downtime and total migration time in live migration [1]. During the migration, resource in both machines must be reserved on migration application and the source machine may not be freed up for other purpose and so it is significant to minimize the total migration time. It inevitabilities to consider the moving of VM's memory content, storage, and network connections from the source node to the target node.

The best system for live migration of VMs is pre-copy [2]. It consolidates iterative push phases and a stop-and-copy phase which goes on for a brief length. By 'iterative', precopying happens in rounds in which the pages to be exchanged during round n are those that are modified during round n-1. The number of rounds in pre-copy migration is directly related to the working set which are being updated so frequently pages. The final phase stops the VM, copies the working set and CPU state to the destination host.

The concern of pre-copy based live migration is that total migration time is delayed. It is brought on by the huge measure of transferring data during the whole migration process and maximum number of iterations must be set because dirty pages, frequently updated page, are ensured to converge over multiple rounds.

This paper is meant to foresee the memory pages which are most frequently used. Recurrences of references are the essential parameter that determines the likelihood of a memory page to be accessed in the near future. The

©2006-2017 Asian Research Publishing Network (ARPN). All rights reserved.

optimal page replacement policy gives importance only to the page which cannot be used for the longest period of time. Constructing memory pages combination is very essential to forecast the memory pages and to measure how close a group of data is accessed together within an execution. They measure togetherness with a stack distance, which is defined as the amount of distinct data accessed between two memory references in an execution trace. We utilized lazy counting based splay tree algorithm for grouping the most frequently accessed memory pages.

In this paper, we advise the algorithm to foresee the working set; the collection of most recurrently used memory pages, in pre-copy based migration for VMs and then state working set. We also present the proposed framework for pre-copy based live migration to diminish the total migration time. Agreeing to the experimental results, we can diminish the total migration time of live migration of VMs.

The rest of this paper is structured as follows. Section 2 describes the associated work. In Section 3, we discourse the live migration of VMs, optimal page replacement algorithm and lazy counting based Splay Tree algorithm. In Section 4, the framework of pre-copy based live migration and proposed lazy counting based Splay Tree algorithm are described. In Section 5 experimental results are described to predict the total migration and downtime. Finally, Section 6 completions the paper.

2. RELATED WORK

In live VM migration, Pre-Copy [2] is the default migration algorithm for Xen. Because of programs' local principles, VM's downtime is expected to be minimal, and in the same time, the source node maintains the newest memory image until migration is finished. The whole process is reliable, because if destination node crashes, Pre-Copy can abort migration and continue to run VM on the source node. However, when applications' loads are intensive, Pre-Copy has to transfer too much memory image data, and consequently has great time overhead. Post-Copy [4] is proposed to solve this problem, and it works in two phases. In the first phase, VM is suspended on the source node, and its VCPU context and minimal memory working set are copied to destination. In the second phase, VM is started on the destination node, and all the memory write operations are executed locally.

When VM needs to read some pages that the source node has the newest version, these pages are fetched through network. In the meantime, the source node keeps pushing remaining memory image to the destination node until done. Post-copy thus ensures that each memory page is transferred at most once. However, both source and destination have part of the newest memory status during migration. If the destination node crashes, VM cannot restart on the source node, so Post-Copy does not have the same level of reliability as Pre-Copy.

In pre-copy based live migration [5] of VM such as VMware [6], XEN [7] and KVM [8] and post-copy based live migration, sorting the page-cache in LRU order performs better than non-LRU cases by improving the locality of reference of neighboring memory pages in the pseudo-paging device. Recency and Frequency of references are the two important parameters that determine the likelihood of a memory page to be accessed in the near future. The LRU policy gives importance only to the regency of references. FBR is a frequency-based policy that is similar to LRU but uses the concepts of correlated references [9]. In LRU-K policy, replacements are based on the time of the K th to last non-correlated reference to each block [10].

The proposed algorithm used the optimal page replacement policy with lazy counting based splay tree for grouping the frequently accessed memory pages according to their process ID to predict the working set in Live VM migration.

3. BACKGROUND THEORY

A. Live migration of VMs

VM migration takes a running VM and moves it from one physical machine to another. This migration must be transparent to the guest operating system, applications running on the operating system, and remote clients of the VM. Live Migration migrate OS instances including the applications that they are running to alternative VMs freeing the original VM for maintenance. It rearranges OS instances across VMs in a cluster to relieve load on congested hosts without any interruption in the availability of the VM as shown in Figure-1.

©2006-2017 Asian Research Publishing Network (ARPN). All rights reserved.

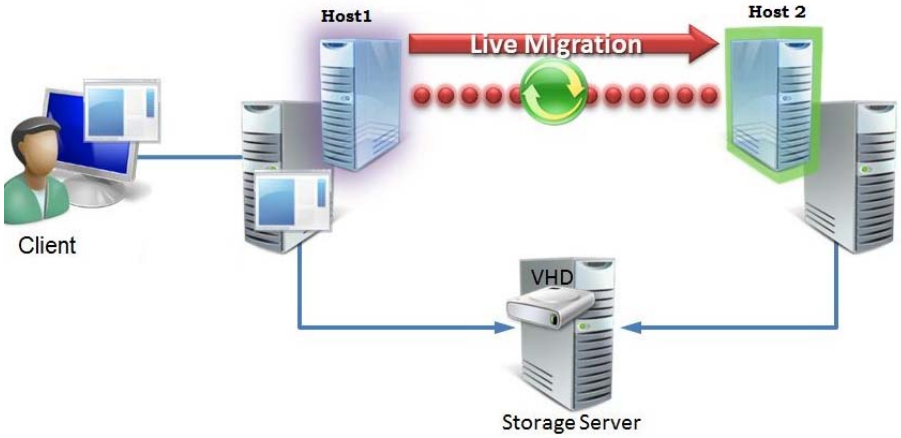

Figure-1. Live migration of VMs.

A key challenge in managing the live migration of OS instances is how to manage the resources which include networking, storage devices and memory [10].Networking: In order for a migration to be transparent all network connections that were open before a migration must remain open after the migration completes. To address these requirements, the network interfaces of the source and destination machines typically exist on a single switched LAN. Storage Devices: We rely on storage area networks (SAN) or NAS to allow us to migrate connections to storage devices. This allows us to migrate a disk by reconnecting to the disk on the destination machine. Memory: Memory migration is one of the most important aspects of VM migration. Moving the memory instance of the VM from one physical state to another can be approached in any number of ways.

In pre-copy based live migration of VM, it involves a bounded iterative push phase and then typically a very short stop-and-copy phase. It first transfers the memory pages iteratively, in which the pages modified in a certain round will be transferred later in the next round. The iterative push phase continues until stop conditions. After that, the source VM stops and transfers its own state and modified pages from the last iteration.Many hypervisor-based approaches such as VMware [6], XEN [7] and KVM [8] is using the pre-copy approach for live migration of VMs.

B. Optimal page replacement algorithm

It associates with each page the time of that page's last use. When a page must be replaced, the optimal page replacement chooses the page that has not been used for the longest period of time [10]. The problem is to determine and order for the frames defined by the time of last use. Two implementations are feasible: Counters and Stack [12]. We apply Stack implementation that keeps a stack of page numbers and most frequently accessed memory page move to the top. So, the memory pages used in the recent past are always on the top of the stack. By dynamically monitoring memory accesses and constructing the optimal page replacement list, we can predict the Working Set list of a VM.

C. Lazy counting based splay tree

A splay tree [3] [12] is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O (log n) amortized time, n means number of nodes. For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.

They use a heuristic restructuring called splaying to move a specified node to the root of the tree via a sequence of rotations along the path from that node to the root. Thus, future calls to this node will be accessed faster. Splay trees even enjoy a constant operation time. The key to a splay tree is, of course, the restructuring splay heuristic [3] [13].Specifically, when a node is searched, it repeats the following splay step until is the root of the tree.

A counting-based self-adjusting search tree that is similar to splay trees moves more frequently injected nodes closer to the root. After M injections on N items, Q of which access some item V1, an operation on V1 passes through a path length of O (log M/Q) while performing fewer if any rotations [14]. In lazy splaying, in addition to the item's value, each node w has three counters: selfCnt, which is an estimate of the total number of operations performed on the item in w (number of find (w: v) and insert (w: v) operations) and rightCnt and leftfCnt, which are estimates of the total number of operations that have been performed on items in the right and left subtrees, respectively. Each find (i) and inject (i) operation increments selfCnt of the node containing i. When node i is found in the tree, all the nodes along the path from the root to i's parent increase their rightCnt/leftCnt counter depending on whether i is in their right or left subtree, respectively[3] [13] [14] [15].

©2006-2017 Asian Research Publishing Network (ARPN). All rights reserved.

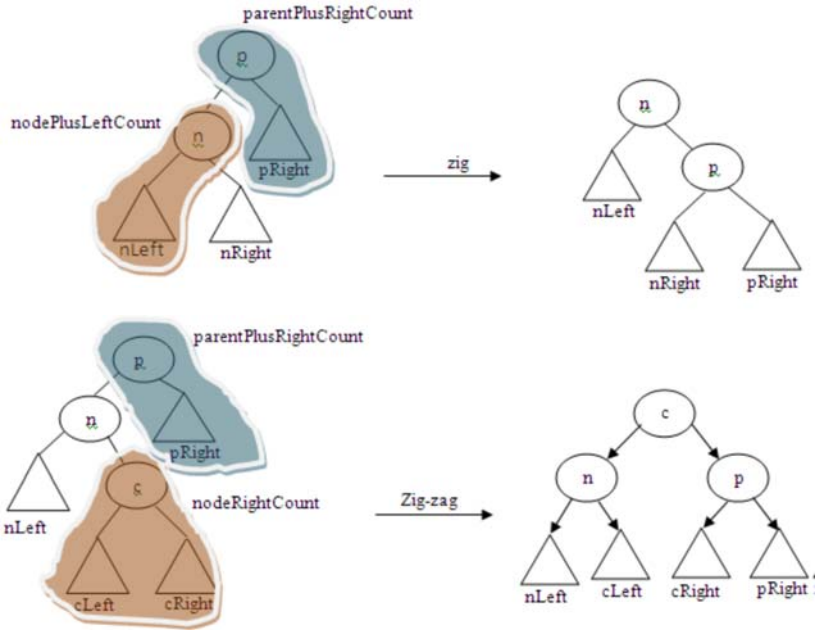

Figure-2. Splaying steps: zig and zig-zag[3].

Figure-2, zig-zag is carried out if the total number of accesses to the node's right subtree is greater than the total number of accesses to the node-parent and its right subtree. If zig-zag is not performed, then zig is performed if the total number of accesses to the node and its left subtree is greater than the total number of accesses to the node-parent and its right subtree [3].

After the rotation the rightCnt and leftCnt counters are updated to represent the number of accesses in the new right/left subtrees, respectively. Note that zig and zig-zag have symmetric mirror operations when the subtree at node p leans to the right. To avoid a chain of nodes where all nodes are left (or right) parents of their children in the case of a descending/ascending insertion order, when a new node is inserted into the tree a re-balancing operation, as specified in Table 1, which in turn calls Table-2, is performed from this new node up to the root.

Good performance for a splay tree depends on the fact that it is self-optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more quickly. The worst-case height - though unlikely - is O (n), with the average being O (log n). Having frequently-used nodes near the root is an advantage for nearly all practical applications.

The advantages of splay tree algorithm are i) Simple implementation, ii) Comparable performance, iii) Small memory footprint, and iv) Working well with nodes containing identical keys. The disadvantage of splay tree algorithm is the height of a splay tree can be linear. To avoid this in Lazy counting-based splay tree , if the depth is greater than 2logN, splaying is performed up to the root by using either double rotation, in which the total number of accesses to x's right subtree is greater than or equal to the total number of accesses to x's parent and its right subtree, or single rotation, in which the total number of accesses to x and its left subtree is greater than the total number of accesses to x's parent and its right subtree, as shown in Figure-3.

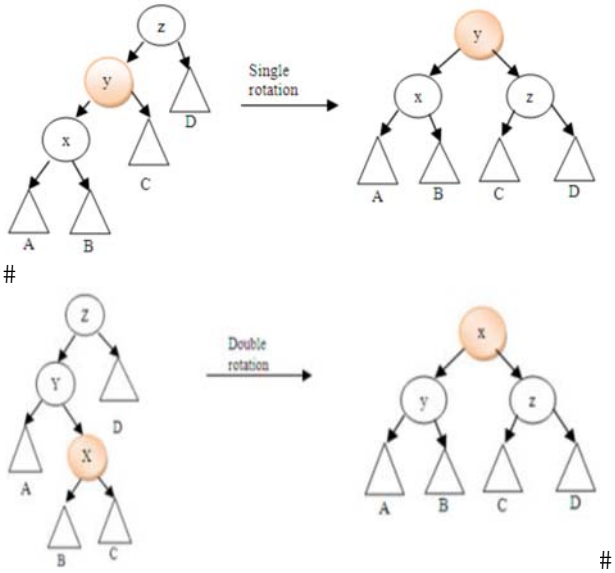

Figure-3. Semi-splaying: the colored node is the current node for splaying [3].

©2006-2017 Asian Research Publishing Network (ARPN). All rights reserved.

Table-1. Attempt injection algorithm.

```
AttemptInjection (key, parent, node, height)
{    if (node = = null)
        {
node = newnode (key, node);
node.selfcount++;
node.rightcount=0;
node.leftcount=0; node. height++;
 break
}
else
if (key = = node. key)
          {
 if (height >= ((2 * log-size)))
splay(node);
else
Rebalance (parent, node);
node.selfcount++;
 return node. value;
 }
  While (true)
 {
 //child in the direction of key
    Child = node.child (key);
if (child = = null)          //Not found
{
//generate a new node and link to node
child = newchild(key, node);
if ( height >= ((2 *log-size)))
splay(child);
return null;
 }
else
result= attemptInjection(key,node,child,height+1);
if (direction to child ==left)
node.leftcount++;
 else
 node.rightcount++;
if (result = = null)
       {//for new node check if re-balancing needed
       // Find the current child of the parent
      //Since it may have changed owing to rotation
//recursive call
curr-node= parent.child(key);
       //Child in the direction of key
Rebalance(parent,curr-node);
   }
```

Table-2. Rebalance algorithm.

```
Rebalance (Node parent, Node node)
{
nodeplusleftcount  =node.selfcount+node.leftcount;
parentplusrightcount  =parent.selfcount+parent.rightcount;
noderightcount  =node.rightcount;
//decide whether to perform zig-zag step
if (noderightcount>= parentplusrightcount)
{
Node grand = parent.parent;
ZigZag (grand, parent, node, rightchild);
parent.leftcount = rightchild.rightcount;
node.rightcount =rightchild.leftcount;
rightchild.rightcount  + = parentplusrightcount;
rightchild.leftcount  + = nodeplusleftcount;
}
else
//decide whether to perform zig step
if (nodeplusleftcount>parentplusrightcount)
{
Node grand = parent. parent;
Zig (grand, parent, node, node.right);
parent.leftcount = node.rightcount;
node.rightcount += parentplusrightcount;
}
}
```

4. A FRAMEWORK FOR LIVE VM MIGRATION

In this Section, we introduce a structure for pre-copy based live migration to diminish the total migration time. The proposed system comprises of the pre-processing phase, push phase and stop and copy phase as shown in Figure-4.

a) Pre-processing phase

The framework applies the proposed working set prediction algorithm as the pre-processing phase is based on optimal page replacement algorithm and lazy counting based splay tree algorithm to define the working set list that collects the most frequently used memory pages (operation 1).

b) Push phase

The framework exchange memory pages except working set list in the first iteration and then memory pages modified during the previous iteration are transferred to the destination (operation 2).

c) Stop and copy phase

This stage comprises of three stages. Firstly, the systems suspend the source VM for a final transfer round (operation 3). Secondly, the system discards the source VM and then transfer last modified pages and CPU state (operation 4). Lastly the system activates the Target VM (operation 5).

The design involves iteration though multiple rounds of replicating in which the VM memory pages that have been modified since the previous copy are resent to the destination.

©2006-2017 Asian Research Publishing Network (ARPN). All rights reserved.

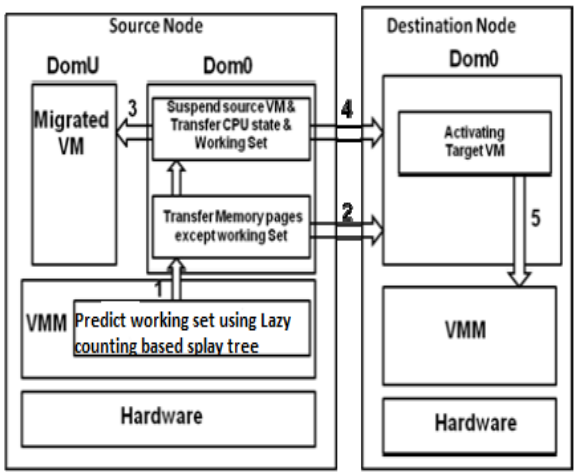

Figure-4. A Framework for pre-copy based live migration.

5. WORKING SET PREDICTION USING LAZY COUNTING-BASED SPALY TREE

In pre-copy based live migration, the system first exchanges all memory pages and afterward copies pages just modified duringthe last round iteratively. VM service downtime is expected tobe minimal by iterative copy operations but total migrationtime is prolonged that caused by the significant amount oftransferred data during the whole migration process. Theproposed Lazy counting based splay tree algorithm predicts the most recently used memorypages that directly impact the total migration time.
In the proposed algorithm as shown in Figure-5, memory ofVM utilizes with Lazy counting based splay tree. The page whichis at the top of the splay tree which collectmemory pages used in the same process are defined asWorking Set.

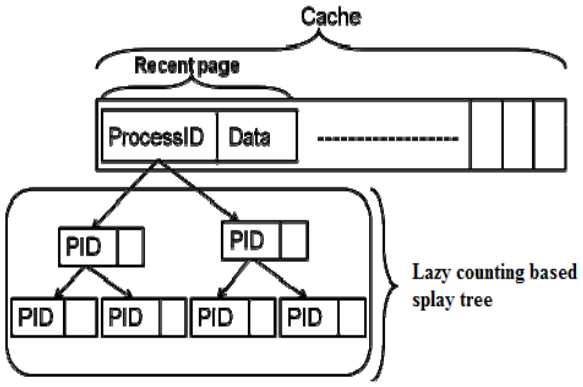

Figure-5. Block diagram for cache with Lazy counting-based Splay tree.

Algorithm: Working set prediction
Input: Request page j with processID
{
LCBSplayTree(processID,Requestpage);
if (!LRU cache is full)
{ Place the page j with Lazycounting based splay tree at the bottom of the LRU cache;}
else
{ if(! Request page j is in LRU cache){
Remove page i whose is located at the bottom of the LRU cache;
Insert request page j with Lazycountingbased splay tree at the bottom of the LRU cache}
else
{Increment the selfcounter of page j;
Increment left and right counters of pages from root to parent of page j, if necessary;}
}
Define page j with Lazy counting based splay tree as the working set;
}
Function: Lazy counting based splay Tree (processID, Requestpage)
{
if(!processID){
Construct new Lazycounting based splay Tree with page j;}
Else(!page j in corresponding LCBSplay tree;}
}

6. METRIC ESTIMATION AND DISCUSSIONS

The results are encouraging and designate great potential for lazy counting-based splays, which is fit for examining the overall performance of splay trees versus lazy counting-based splay trees. Fascinating results are acquired from this examination. Both splay trees and counting-based splay trees use implicit caching by conveying the page of the root element and taking benefit of the locality in incoming lookup requests for the pages. Locality in this perspective refers to looking for the same page a few times. A flood of requests exhibits no locality when every page is equally alluded at every point. For

©2006-2017 Asian Research Publishing Network (ARPN). All rights reserved.

these applications, locality does exist since pages have a tendency to be alluded over and again.

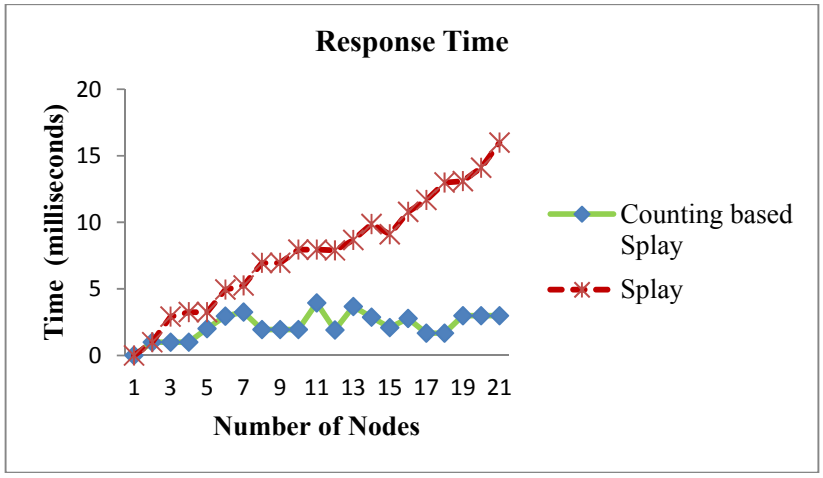

Figure-6. Response time vs number of nodes.

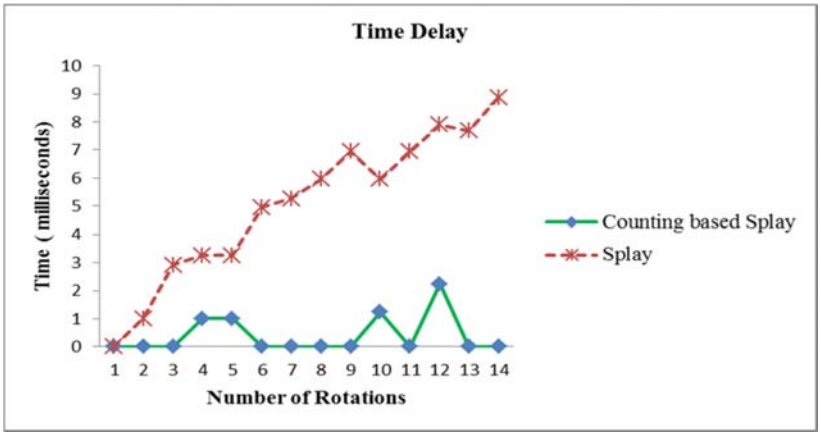

Figure-7. Delay vs number of nodes.

In counter-based splay trees, over a period of time as the number of most frequently accessed nodes increase, the response time is reduced. This is due to the movement of the frequently accessed pages closer to the root is shown in Figure-6.Over a period of time the most frequently accessed nodes are moved closer to the root which in turn uses rotations and hence the depth of the tree is reduced drastically. This leads to the stability in the rotations. The counting-based splay tree performs rotations infrequently and mostly at the bottom of the tree. Therefore, it scales with the level of concurrency is shown in Figure-7.

REFERENCES

[1] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan. 2009. Live VM migration with adaptive memory compression. In Proceedings of the 2009 IEEE International Conference on Cluster Computing (Cluster 2009).

[2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. July, C. Limpach, I. Pratt and A. Warfield. 2005. Live Migration of VMs. Proceedings of the 2nd USENIX Symposium on NetworkedSystems Design and Implementation.

[3] K. Santhi, G. Zayaraz and T. Chellatamila. 2015. Trade-off Analysis of Crosscutting Functionalities using Lazy Counting-based Splay Tree in Aspect Oriented Programming. Research Journal of Applied Sciences, Engineering and Technology. 9(6): 396-408.

[4] M. R. Hines and K. Gopalan. 2009. Post-copy based live VMmigration using adaptive pre-paging and dynamic selfballooning. In: Proceedings of the ACM/Usenix internationalconference on Virtual execution environments (VEE'09). pp. 51-60.

©2006-2017 Asian Research Publishing Network (ARPN). All rights reserved.

[5] P. Lu and K. Shen. 2007. VM memory access tracing with hypervisor exclusive cache. In: ATC'07: 2007 USENIXAnnual Technical Conference on Proceedings of the USENIX Annual Technical Conference, pp.1-15, Berkeley, CA, USA. USENIX Association. ISBN 999-8888-77-6.

[6] M. Nelson, B. Lim, and G. Hutchines. 2005. Fast transparent migration for VMs. In: Proceedings of the USENIX Annual Technical Conference (USENIX'05). pp. 391-394.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. 2003. XEN and the Art of Virtualization. Proceedings of the Nineteenth ACM Symposium Operating System Principles (SOSP19), pp 164-177. ACM Press.

[8] A. Kivity, Y. Kamay, and D. Laor. 2007. kvm: the linux VM monitor. In: Proc. of Ottawa Linux Symposium.

[9] J. T. Robinson and N. V. Devarakonda. 1990. Data Cache Management Using Frequency based Replacement. In: the Proceedings of the 1990 ACM SIGMETRICS Conference.pp. 134-142.

[10] A.Subramanian. 1996. An Explanation of Splaying. Academic Press. Inc.

[11] E. J.O'Neil, P. E. O'Neil, G. Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In:Proceedings of the 1993 ACM SIGMOD Conference. pp. 297- 306.

[12] K. Cheng and Y. Kambayashi. LRU-SP: A Size-Adjusted and Popularity-Aware LRU Replacement Algorithm for Web Caching.

[13] D. Lee, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho and C. Kim. 2001. LRFU: A Spectrum of Policies that Subsumes the LRU and LFU Policies. IEEE Transactions on Computers. 50(12): 1352-1361.

[14] D. Dominic Sleator, R. Endre Tarjan. 1985. Self-Adjusting Binary Search Trees. In Journal of the Association for Computing Machinery. 32(3): 652-686.

[15] Afek, Y., H. Kaplan, B. Korenfeld, A. Morrison and R.E. Tarjan. 2012. CBTree: A practical concurrent self-adjusting search tree. Proceeding of the 26th International Conference on Distributed Computing (DISC, 2012).pp. 1-15.

[16] Bronson N.G., J. Casper, H. Chafi and K. Olukotun. 2010. A practical concurrent binary search tree. Proceeding of the 15th ACM SIGPLAN Symposium on Principals of Parallel Programming.