Tracking Working Set Sizes of Virtual Machines Using Miss Ratio Curves

by

Sarandeth Reth

Submitted to the
Department of Electrical Engineering and Computer Science
In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 2017

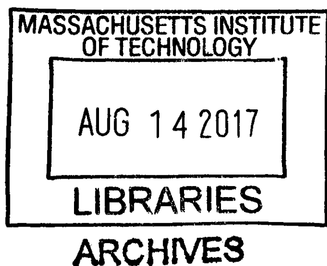
© 2017 Massachusetts Institute of Technology. All rights reserved.

Author: Signature redacted
Department of Electrical Engineering and Computer Science
May 26, 2017

Certified by: Signature redacted
Saman Amarasinghe
Professor
Thesis Supervisor
May 26, 2017

Certified by: Signature redacted
Yuri Baskakov
Thesis Co-Supervisor
May 26, 2017

Accepted by: Signature redacted
Christopher Terman
Chairman, Masters of Engineering Thesis Committee


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
AUG 14 2017
LIBRARIES
ARCHIVES

Tracking Working Set Sizes of Virtual Machines Using Miss Ratio Curves

by

Sarandeth Reth

Submitted to the
Department of Electrical Engineering and Computer Science
May 26, 2017
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Working sets are sets of pages that have been most recently accessed by virtual machines (VMs). They are often used within the memory scheduler of a hypervisor to estimate the memory demands of VMs running on the hypervisor. In order to manage the memory resources of the hypervisor efficiently, it is essential that these working set sizes be estimated accurately at any given point in time.

Currently, a statistical sampling strategy is used within VMware ESX hypervisors to estimate the working set sizes of VMs. Using this technique, a small number of random pages is selected to form a sample set. Access to these sampled pages is then tracked and the percentage of sampled pages that are accessed is used to estimate the working set size of a VM. This technique, though simple, does not provide a very accurate estimation of the working set size.

A more promising tool that can be used to accurately estimate the working set size of a VM is a miss ratio curve (MRC). An MRC is a curve that plots the predicted miss ratio of a VM against the total available memory given to the VM. Even though MRCs can estimate working set sizes of VMs with much better accuracy, they are still not widely used in practice because building these curves incurs too much overhead, thus affecting the overall system performance. However, a recent study has found a way to reduce the cost of building these curves, making them a promising tool that can be used to estimate working set sizes. In this thesis, I propose that MRCs be used as an alternative to the statistical sampling strategy currently employed within VMware ESX. I will demonstrate how to apply the state of the art technique found in the recent study to construct accurate MRCs without incurring too much overhead, and use these curves to track working set sizes of VMs. I will also show that these curves can estimate working set sizes of VMs with much better accuracy than the statistical sampling strategy.

Thesis Supervisor: Saman Amarasinghe
Title: Professor

Thesis Co-Supervisor: Yuri Baskakov

Acknowledgements

Professor Saman Amarasinghe for serving as my supervisor and for providing me with his advice and discussions about this project.

Yuri Baskakov for being an awesome mentor while I was at VMware. I am indebted to him for providing me with so much help and many suggestions about the implementation of this work. This project would not be possible without his support and guidance.

My mother, Sokcheun Kov, my father, San Lim, and my younger brother, Darasy Reth, for their unwavering support throughout my life.

Contents

1. Introduction 7
1.1 Related Work 11

2. MRC Construction 13
2.1 Stack Implementation 15
2.2 Randomized Spatial Sampling 17
2.2.1 Sampling Condition 17
2.2.2 Fixed-Size Implementation 18

3. Memory Tracing 23
3.1 Sampled Pages Collection 23
3.2 Sampled Pages Tracking 24

4. Overhead Measurement 25
4.1 Time Overhead 25
4.1.1 Experimental Setup 25
4.1.2 Experimental Results 26
4.2 Space Overhead 27

5. Sample Set Size 29
5.1 Small Virtual Machine 30
5.1.1 Experimental Setup 30
5.1.2 Experimental Results 30
5.2 Large Virtual Machine 34
5.2.1 Experimental Setup 34
5.2.2 Experimental Results 34

6. Bucket Size 39
6.1 Small Virtual Machine 39
6.1.1 Experimental Setup 39
6.1.2 Experimental Results 39
6.2 Large Virtual Machine 40
6.2.1 Experimental Setup 40
6.2.2 Experimental Results 41

7. Evolution of Sampling Rate 43
7.1 Intensive Workloads 43

7.1.1 Small Virtual Machine 43
7.1.1.1 Experimental Setup 43
7.1.1.2 Experimental Results 44
7.1.2 Large Virtual Machine 46
7.1.2.1 Experimental Setup 46
7.1.2.2 Experimental Result 47
7.2 Light Workloads 48
7.2.1 Experimental Setup 48
7.2.2 Experimental Results 48
7.3 Implication 50
8. Working Set Size Tracking **57**
8.1 Small Virtual Machine 57
8.1.1 Experimental Setup 57
8.1.2 Performance Graph 58
8.1.3 Working Set Estimated by MRC 59
8.2 Large Virtual Machine 61
8.2.1 Experimental Setup 61
8.2.2 Performance Graph 62
8.2.3 Working Set Estimated by MRC 63
8.2.4 Importance of Stable Sampling Rate 65
8.3 Mixed Workloads 67
8.3.1 Experimental Setup 67
8.3.2 Experimental Results 68
9. Modified Version of SHARDS **71**
9.1 Experimental Setup 72
9.2 Experimental Results 72
9.2.1 First Workload (xml.validation) 72
9.2.2 Second Workload (serial) 73

1. Introduction

Large companies like Microsoft, Google, and Facebook deploy hundreds of thousands of servers to serve their users around the world. Each server needs to be kept constantly running in order to provide non-disruptive services to the users. As a result, it costs these companies millions of dollars each year just to operate their servers. It is, therefore, important that each server is utilized efficiently in order to reduce their operating costs.

Virtualization has emerged as one of the most promising solutions to this cost cutting problem. Virtualization is a process that creates a virtual machine (VM) by simulating physical hardware resources in a server. With virtualization, multiple VMs can run on the same server at the same time. This is very beneficial because workloads that need to be distributed among multiple servers can now be run on multiple VMs running on just one server. This ability to run multiple VMs on the same server leads to a more efficient usage of resources by reducing the number of servers that need to be kept running at any point in time.

Running multiple VMs on the same server, though desirable, is a challenge. These VMs need to share and compete for the same computing resources, such as memory and disk space, available on the server that they run on. Without proper management, some greedy VMs might take all the available resources, leaving others to starve. The performance of some VMs might, therefore, be compromised if they are not given enough computing resources and the benefits gained from running multiple VMs on the same server no longer hold. Therefore, in order for virtualization to work, efficient allocation techniques and policies are required to distribute these limited resources among VMs.

Memory is one of the most crucial computing resources inside a server. To achieve an efficient usage of memory resources, a VMware ESX server overcommits by assigning more total available memory to VMs than is physically available on the host machine [1]. This is beneficial in practice because VMs do not use all their assigned total available memory at all times. In such an ESX host with multiple VMs, the goal of a memory scheduler is to provision memory resources among the VMs efficiently in order to achieve the optimal performance of the

overall system. At the same time, the memory scheduler also has to respect the memory allocation priorities of those VMs. Ideally, under memory pressure, memory can be claimed from a VM to improve the overall system performance as long as the performance of that VM is not compromised. The claimed memory should then be distributed among other VMs where extra memory matters the most.

Working sets are often used by the memory schedulers of hypervisors to estimate the memory demands of VMs in order to provision memory resources among them [2]. The working set of a VM is defined as the set of pages that are most recently accessed by the VM [3]. In order to manage memory resources efficiently, it is important that the size of each VM's working set is accurately approximated at any given point in time [4].

Currently, a statistical sampling strategy is used within the memory scheduler of a VMware ESX hypervisor to estimate the working set size of each VM running on the hypervisor. Using this technique, a small number of random pages is selected to form a sample set for each sampling period. Access to these sampled pages is then tracked and the percentage of sampled pages that are accessed is used to estimate the working set size of the VM [2]. An example of how this statistical sampling strategy is used to estimate the working set size of a VM is illustrated in Figure 1. This technique, though simple, does not provide a very accurate estimation of the working set size.

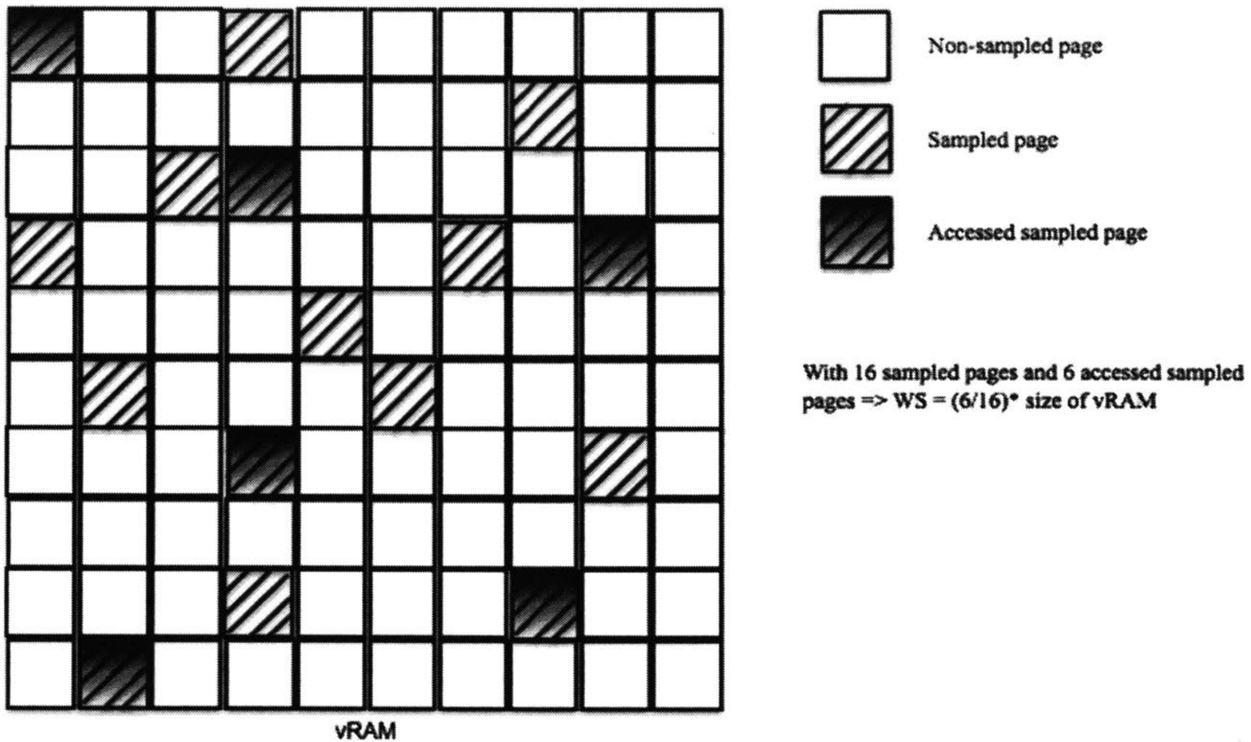


Figure 1: Statistical sampling strategy used to estimate the working set size of a VM. In this example, 16 pages out of all pages in the entire vRAM of the VM are selected to form a sample set. Access to these sampled pages is tracked. Since 6 out of 16 sampled pages are accessed, the working set size of the VM is estimated to be 6/16 of the size of the entire vRAM.

A more promising tool that can be used to accurately estimate the working set size of a VM is a miss ratio curve (MRC). An MRC is a graph that plots the predicted miss ratio of a VM against the total available memory given to that VM [5], as shown in Figure 2.

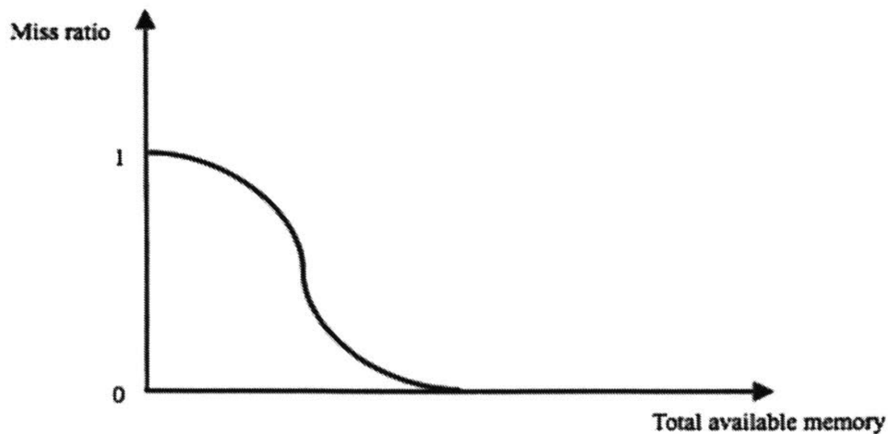


Figure 2: An example of a miss ratio curve (MRC).

A miss ratio is defined as the fraction of page accesses of a VM that result in page faults. For each page fault, a page is often retrieved from a swap file that is usually stored in secondary storage, such as a disk. This significantly slows down the execution of the VM. The number of page faults is therefore one of the main factors that determine the execution time of the workload running in the VM, and the MRC can thus be used to explain how performance of the VM changes with its total available memory [4].

The change in the miss ratio around the current total available memory given to a VM can be used to determine when memory can be claimed from the VM without affecting its performance, as well as when memory should be given to the VM to improve its performance. For example, in Figure 3, if the current total available memory given to the VM is less than *X*, then giving this VM more memory would result in better performance as it leads to a reduction in the miss ratio. However, if the current total available memory is at *X*, then giving the VM more memory would not result in performance gain since the number of page faults is zero either way. The tail of the curve, point *X*, indicates the minimum total available memory that would result in the optimal performance of the VM. This point *X* therefore corresponds to the working set size of the VM. In this case, the working set size of the VM can be tracked simply by looking at how the tail of the curve evolves.

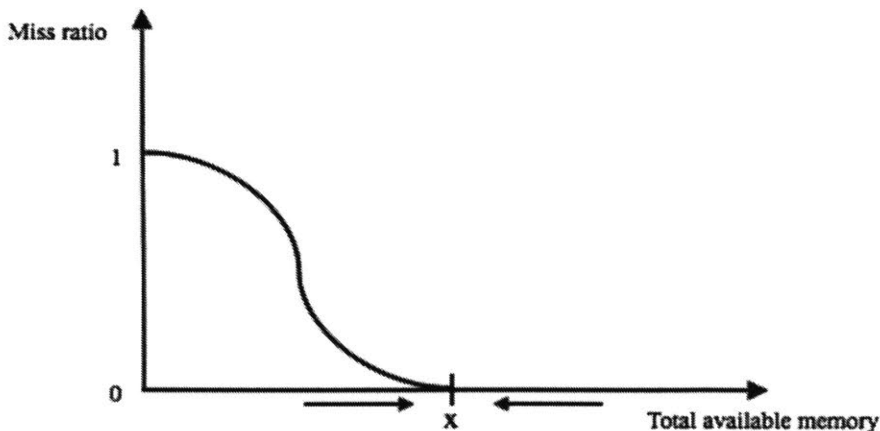


Figure 3: An increase in the current total available memory leads to better performance of the VM when it is less than *X*, while further increase beyond *X* does not affect the VM's performance. Point *X* is the VM's working set size.

Although MRCs can give accurate estimations, they are still not widely used in practice because building these curves incurs too much overhead, thus affecting the overall system performance. However, a recent study has found a way to reduce the cost of building these curves, making them a promising tool that can be used to support virtualization. In this thesis, I will demonstrate how to apply the state of the art technique found in the recent study to construct accurate MRCs without incurring too much overhead, and use these curves to track working set sizes of VMs running on a hypervisor. I will also show that MRCs can estimate working set sizes of VMs with much better accuracy, thus proving that they can be used as an alternative to the statistical sampling strategy currently employed within VMware ESX hypervisors.

1.1 Related Work

With regard to the recent memory management techniques used inside VMware ESX, [2] details how the statistical sampling strategy is used to estimate a VM's working set. A recent study shown in [5] attempts to use MRCs in memory management. However, this study was done in the context of operating systems, while my research project is about memory management inside VMware ESX hypervisors. Another study, shown in [4], details the use of an MRC to estimate the size of a working set. It also shows useful examples of how MRCs are used to balance memory resources among VMs. However, the implementation of that MRC construction has more overhead than the one presented in my research project. The SHARDS technique presented in [6] is the most efficient implementation of MRC construction. It models cache utility curves by scanning through block I/O traces collected from data centers. Many of the implementation details presented in my research project are inspired by the ones used in [6]. However, the SHARDS technique concerns itself mainly with cache analysis. An MRC produced by this technique is used to estimate the cache size that would result in a desired I/O latency. In contrast, my research project is mainly about using MRCs to provision memory among VMs.

2. MRC Construction

An MRC can be constructed using Mattson's stack algorithm [7]. This algorithm computes the fraction of page accesses that are misses for each total available memory size given to a VM by building a histogram of reuse distances out of the memory trace [6]. The histogram of reuse distances records the number of page references for each reuse distance as shown in Figure 4.

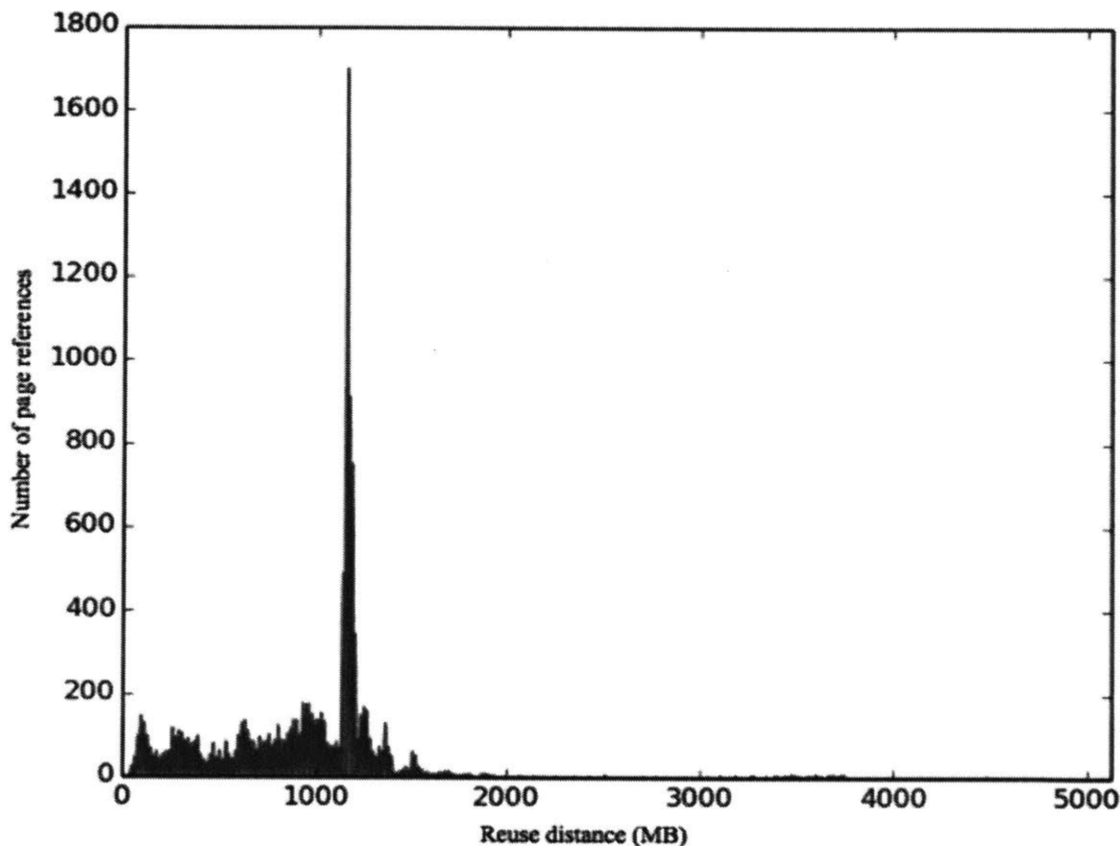


Figure 4: Histogram of reuse distances records the total number of page references for each reuse distance.

The reuse distance of a page access can be computed by counting the number of distinct pages that have been accessed since its last access [6]. For example, the reuse distance for an access to page X in Figure 5 is 2, since there are only two other distinct pages, Y and Z, that have been accessed between the two consecutive accesses to page X.

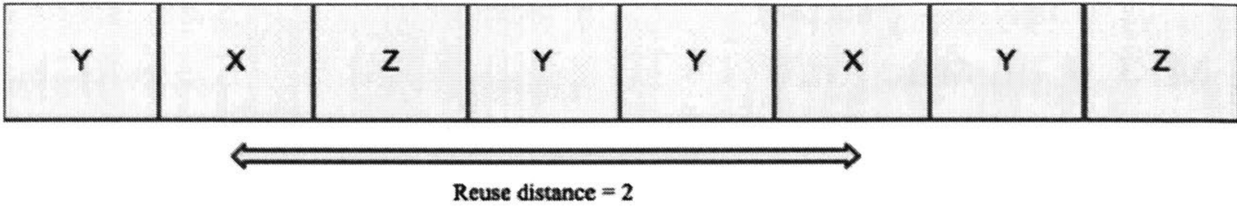


Figure 5: This figure shows the trace of memory access where each block represents a page. The reuse distance for an access to page *X* is 2 since there are only 2 other distinct pages, *Y* and *Z*, that have been accessed between the two consecutive accesses to page *X*.

An LRU stack is used to compute these reuse distances. Accessed pages are arranged in this stack such that the most recently accessed page is at the top of the stack and the least recent one is at the bottom [6]. When a page is accessed, the stack distance of that page is computed by counting the number of other pages inside the stack that are above this page, as shown in Figure 6. This stack distance is then assigned to be the reuse distance of that page [6].

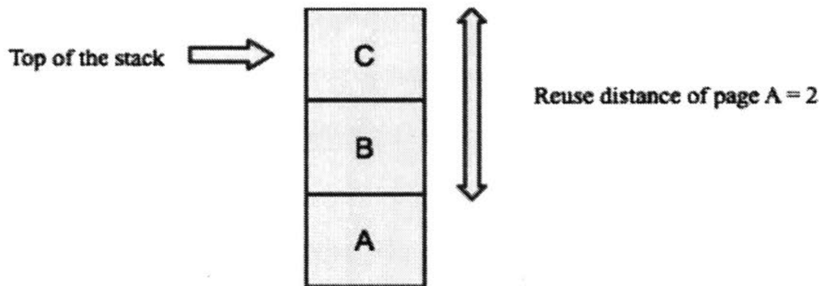


Figure 6: Each box in this figure represents a sampled page. The reuse distance of page *A* is 2 since there are 2 pages that stack on top of it.

The counter of the histogram bucket that corresponds to the computed reuse distance is then incremented. The accessed page becomes the most recent one and is moved from its initial position inside the stack to the top of the stack [6]. Subsequently accessed pages in the memory trace are then processed similarly to collect their reuse distances. After some interval of time the histogram of reuse distances is converted to an MRC, as described in Figure 7.

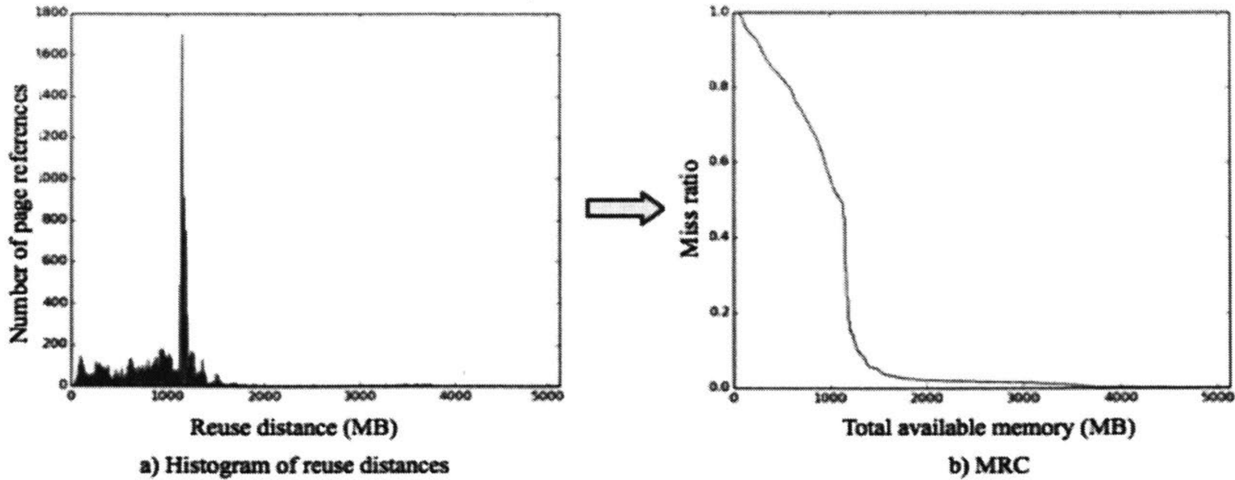


Figure 7: The histogram of reuse distances can be converted to an MRC by computing the miss ratio for each reuse distance. The miss ratio of a reuse distance *d* can be computed as follows:

$$miss\ ratio\ (d) = \frac{\sum_{i>d}^{\infty} histogram\ count\ (i)}{\sum_{i=0}^{\infty} histogram\ count\ (i)}$$

The numerator represents the total number of page misses if the size of total available memory of the VM is set to *d*, while the denominator represents the total number of page misses if the VM is given no memory at all.

MRCs, though very useful, are not widely used inside memory schedulers because tracking these curves dynamically incurs too much overhead. To construct a perfect MRC, an interception of every page access is required. As a result, the latency of each page access increases and the performance of the VM is compromised. Much space is also required to record these page accesses [6]. This overhead becomes even more problematic for large VMs with terabytes of memory. To reduce this overhead, efficient implementation of MRCs is required.

2.1 Stack Implementation

Locality of page references is a situation where recently accessed pages are the most likely to be accessed again [8]. Due to most programs' locality of page references, efficient implementation of the LRU stack can be achieved using a splay tree [6] since such a tree permits recently accessed nodes to be easily accessed again. Using a splay tree, most stack operations can be done efficiently in most practical cases. (*O(log N)* is the amortized time where N is the number of nodes inside the splay tree.) Each node inside the splay tree can be represented by the *Node* struct shown below:

```
unsigned counter = UINT_MAX;        // used for ordering nodes

struct Node {
     Node *parent;                  // pointer to parent node
     Node *left;                    // pointer to left child node
     Node *right;                   // pointer to right child node
     unsigned key;                  // key value used for sorting
     unsigned left_tree_size;       // the number of nodes in
                                    // left subtree
};
```

In this struct, the *parent*, *left* and *right* fields are used for tree traversal. The *key* field is used for ordering nodes where the node with the smallest key corresponds to the most recently accessed page, the one at the top of the stack. The global variable *counter* is used to assign a unique key to each node. This variable is initially set to some maximum value and is decremented after every time it is assigned as the key of a newly inserted node. Later nodes inserted into the tree would have smaller keys, thus corresponding to more recently accessed pages. Each node also stores the total number of nodes in its left subtree, the *left_tree_size* field, in order to support stack distance computation. The stack distance of a page is the number of nodes that have smaller keys than the key of the node of that page. For a given page, its corresponding node can be efficiently searched for by using a hash table to map each page to its node, as shown in Figure 8 [6]. Each page can be hashed using its physical page number (PPN) and doubly linked lists are used to avoid collisions inside the hash table.

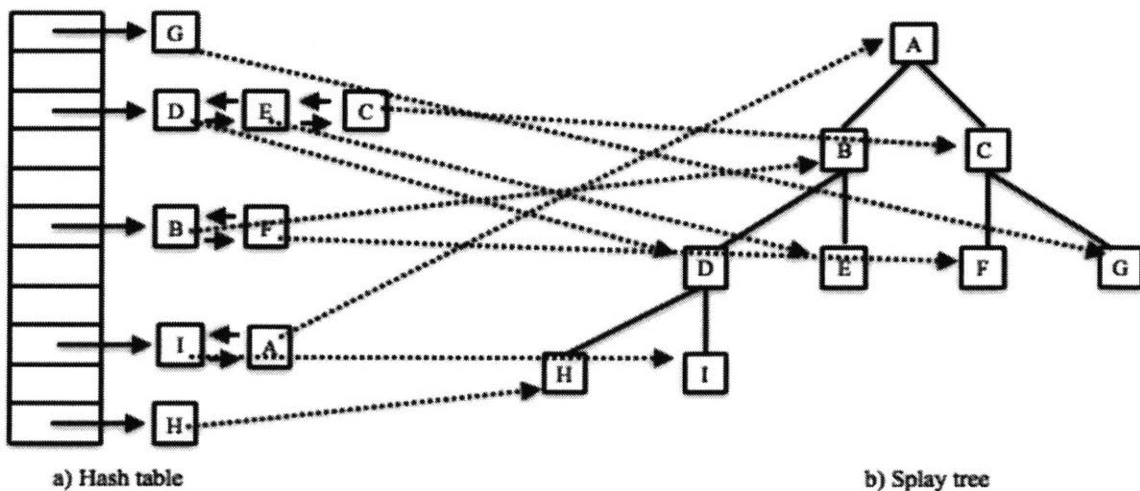


Figure 8: The hash table maps each page to its corresponding node. Each page is hashed using its physical page number (PPN) and doubly linked lists are used to avoid collisions inside the hash table.

With these efficient data structures, the algorithm performed to collect a histogram of reuse distances has the time complexity of $O(M \log(N))$ and the space complexity of $O(N)$ where M is the size of the memory trace in pages and N is the number of unique pages [6]. For online MRCs that are built every T time interval, M would be the maximum number of non-distinct pages that can be accessed during this T period and N would be the total number of pages that can fit into memory. As a result, for large VMs with terabytes of memory, the time and space overhead is still great. Extra measures therefore need to be taken to reduce this overhead.

2.2 Randomized Spatial Sampling

The randomized spatial sampling technique, to be discussed here, is inspired by the SHARDS technique described in [6]. It can be added to the existing implementation to reduce the overhead of MRC construction. It incurs minimal overhead and still produces very accurate MRCs. Using this technique, only the pages included in a sample set are monitored. The hash value of a page's PPN is used to determine whether that page should be sampled. The idea is to keep the number of monitored pages small in order to reduce the overhead, but these sampled pages still accurately represent all accessed pages, so that an accurate MRC can still be constructed using a small sample set [6].

2.2.1 Sampling Condition

When a page is accessed, it is put into the filter to determine whether it should be included inside the sample set. An overview of the sampled page selection process is shown in Figure 9. The sampling condition is that the hash value of the PPN of a sampled page must be less than the threshold T. As shown in Figure 9, since T out of P possible hash values are accepted, the size of the sample set is approximately T/P percent of the entire memory space. This value, T/P, is also the value of the sampling rate R and every $1/R = P/T$ accessed pages are therefore represented by a sampled page. This means that the stack distance is no longer equal to the reuse distance. Each reuse distance is $1/R$ of the computed stack distance [6], as shown in Figure 10.

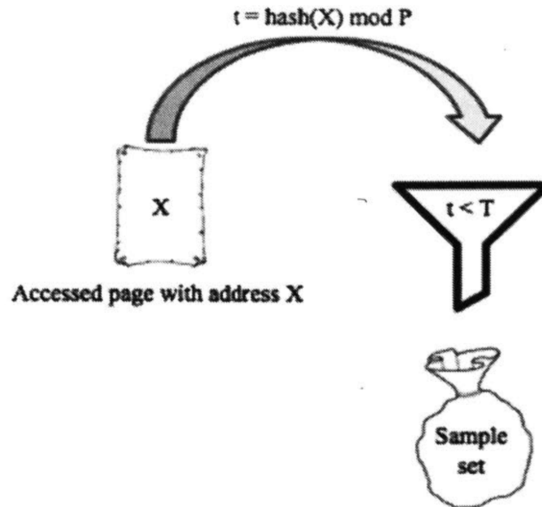


Figure 9: An overview of the sampled page selection process. The PPN of the accessed page, X, is hashed to a value t that falls with equal probability between 0 and an upper limit P. If the hash value t satisfies the sampling condition, if it is less than the threshold T, then the accessed page is part of the sample set, otherwise it is ignored [9].

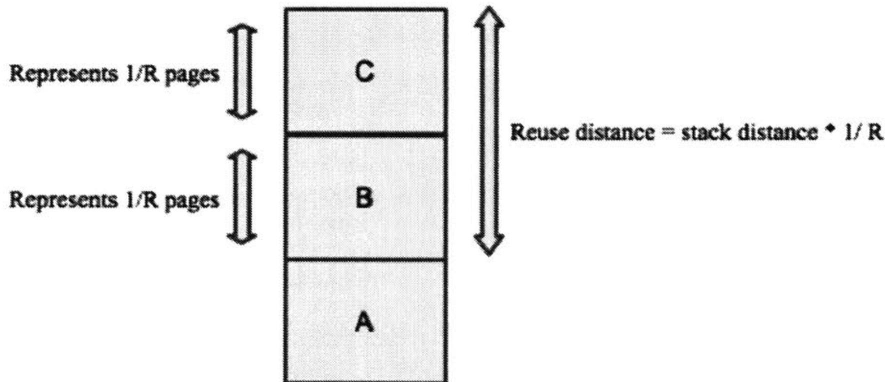


Figure 10: Each box in this figure represents a sampled page. Each reuse distance is approximately $1/R$ of the computed stack distance since each sampled page represents $1/R$ pages.

2.2.2 Fixed-Size Implementation

With the filter described in Figure 9, accessed pages that do not satisfy the sampling condition are not processed. Therefore, the size of the memory trace that needs to be processed is only about RM pages and the number of unique pages that are processed is only about RN pages. With a small sampling rate R, the overhead can be reduced significantly. However, there is still

no definite upper limit on this overhead. The sample set size can still increase indefinitely. This leads to a need for a fixed-size implementation that limits the sample set size. This fixed-size implementation would reduce the space overhead further to *O(1)* and time overhead to *O(M)*. However, when an upper limit is placed on the sample set size, it is necessary that there be a sample replacement policy to update the sample set once it is full. The policy used is again inspired by the SHARDS technique presented in [6]. It requires a dynamic adjustment of the sampling rate in order to fix the sample set size. This policy is made possible by exploiting the subset-inclusion property of the sampling condition described in Figure 11 [6].

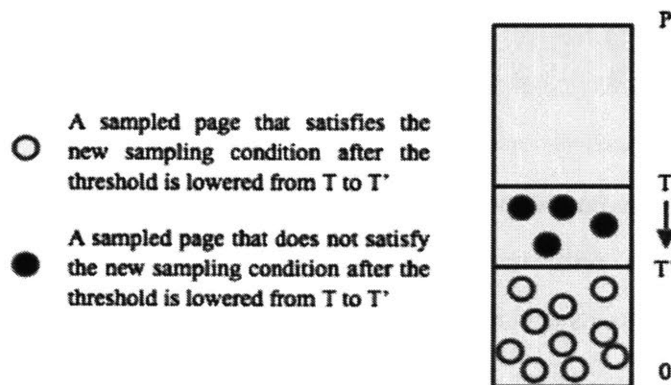


Figure 11: Since the sampling rate is proportional to the threshold, $R = T/P$, when the value of the threshold is lowered from T to T', the sampling rate is also lowered from R to R' by the same factor. Since $T > T'$, the subset-inclusion property is preserved: sampled pages that satisfy the new sampling condition $hash(X) \bmod P < T'$ after the threshold is lowered (all white balls shown above) also satisfied the old sampling condition $hash(X) \bmod P < T$ before the threshold was lowered.

Before the memory trace is processed, the sampling rate is set to some initial value *R* by assigning the threshold to some value *T*. While the memory trace is being scanned, the sampling rate does not change if the sample set is not yet full. The threshold is still fixed at *T* and the accessed pages are added to the sample set as long as they satisfy the sampling condition. However, if the size of the sample set exceeds its upper limit when a new sampled page is added, a sampled page needs to be removed from the sample set to limit the sample set size, and both the threshold and the sampling rate need to be adjusted, as described in Figure 12. The hash value of the PPN of a sampled page can be considered as the threshold of that sampled page [6].

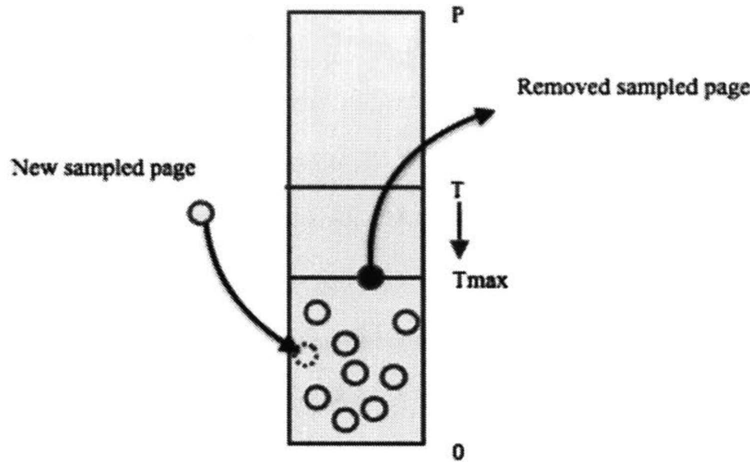


Figure 12: If the size of the sample set exceeds its upper limit when a new sampled page is added, the sampled page with the PPN that has the maximum hash value, *Tmax*, is removed from the sample set to bring the size of the sample set back to its upper limit. The hash value *Tmax* becomes the new threshold and the new sampling rate becomes *Rnew* = *Tmax/P* [6].

With the sample replacement policy shown in Figure 12, a priority queue is required to determine the sampled page with the maximum threshold [6]. This priority queue can be implemented using a max heap, as described in Figure 13.

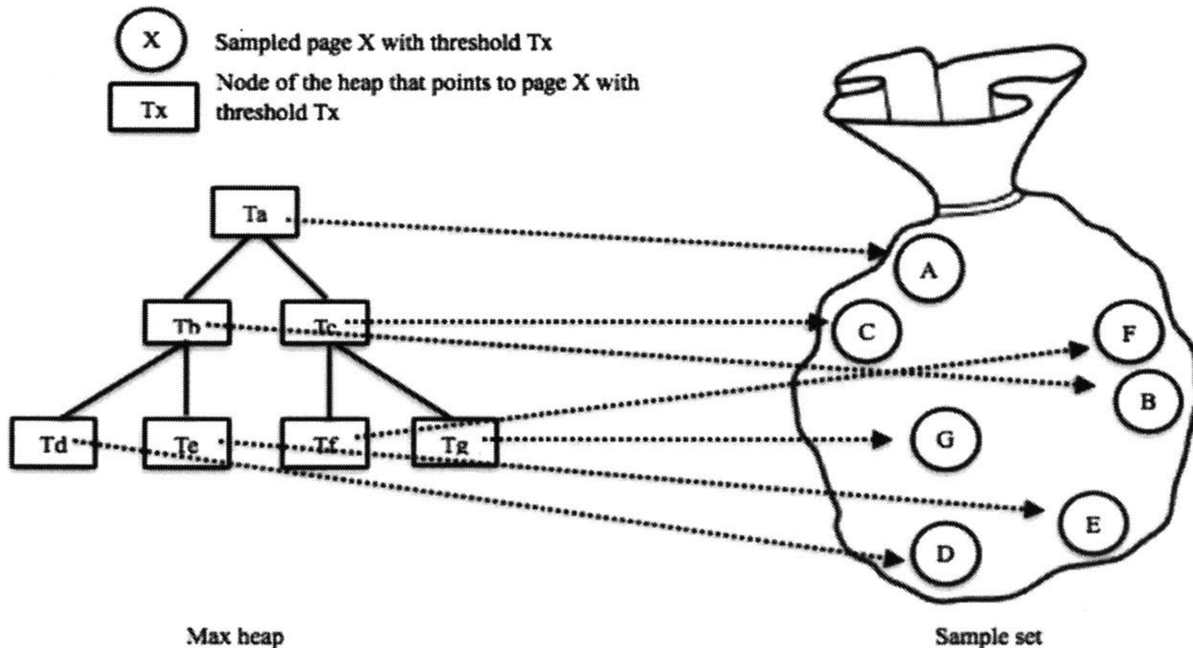


Figure 13: A priority queue can be implemented efficiently using a max heap, where each sampled page is pointed to by an element in the heap. All elements in the heap are compared using the thresholds of the sampled pages that they point to [6]. The top element of the heap therefore points to the sampled page with the maximum threshold. In this example, sampled page A has the maximum threshold since it is pointed to by the top element of the heap.

Since the threshold is lowered as additional sampled pages are added to the full sample set, the sampling rate decreases and each sampled page represents more pages. As a result, the histogram counts collected when the threshold was at T are considered to have lower values than when the threshold is lowered to T' [9]. Each of these histogram counts needs to be rescaled, as described in Figure 14, to reflect the new threshold. Ideally, every time the threshold value is lowered, rescaling should be done on all buckets. To avoid this costly process, further optimization can be done by performing rescaling on only the bucket that needs to be updated, leaving other buckets unaffected. In this case, each bucket needs to associate itself with its own threshold T_{bucket}, which is the threshold used in the sampling condition the last time the histogram count of that bucket was rescaled. Therefore, the scale factor of each bucket is different and equal to T/T_{bucket}, where T is the most recent threshold used in the sampling condition and T_{bucket} is the threshold of the bucket. When the MRC is constructed, the histogram count of each bucket needs to be scaled by the bucket's scale factor T/T_{bucket} before it can be added to compute miss ratios [6].

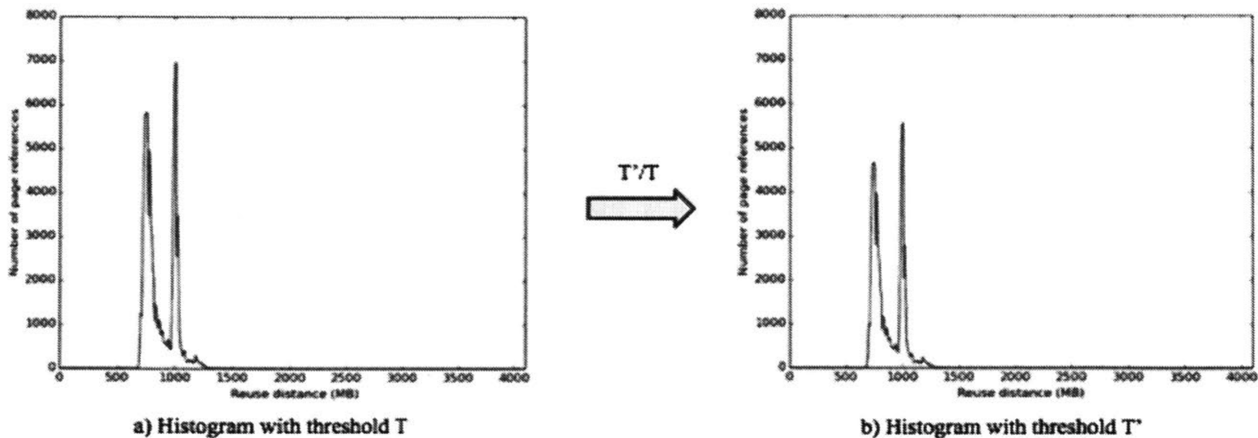


Figure 14: When the threshold is lowered from T to T', each histogram count needs to be rescaled by the scale factor = R'/R = T'/T.

3. Memory Tracing

3.1 Sampled Pages Collection

Sampled pages can be collected during the page fault handling process. When a page is accessed for the very first time by a VM, it only exists inside the virtual address space of the VM and is not yet loaded into the vRAM of the VM. There is no mapping, inside either page tables or TLBs, between the virtual address and the PPN. As a result, every accessed page must go through the page fault handler at least once and all sampled pages can therefore be collected by the page fault handler.

During the page fault handling process, after the VMkernel successfully establishes the mapping between the virtual address and the PPN of an accessed page, the mapped PPN is used to determine whether that page should be sampled, as described in Figure 9. If the page is sampled, then there are two cases to consider:

1. The page is not yet inside the sample set. This can be quickly checked using the hash table, as shown in Figure 8. In this case, the page is added to the sample set and both the threshold and the sampling rate must be adjusted as described in Figure 12. Rescaling of histogram counts must also be performed accordingly [6], as described in Figure 14.
2. The page is already part of the sample set. This can happen when the page was evicted to the disk, causing the old mapping between its virtual address and its PPN to be lost. In this case:
    - The splay tree is used to compute the stack distance of the page
    - The page is moved from its initial position to the top of the stack
    - The stack distance is scaled by 1/R to get the reuse distance
    - The histogram count of the bucket that corresponds to the computed reuse distance is scaled by the bucket's scale factor before it is incremented [6].

3.2 Sampled Pages Tracking

A simple approach to tracking sampled pages accessed by a VM is to intercept access to every sampled page [6]. This can be done by removing the mappings between the virtual addresses and the PPNs of all sampled pages in both page tables and TLBs, so that there are always page faults when sampled pages are accessed. When a page fault occurs, the control of execution is transferred to the kernel of VMware ESX, the VMkernel. Inside the page fault handler of the VMkernel, the accessed page can then be inspected and recorded. However, tracking memory access this way would slow down the execution of the VM significantly, especially when the VM runs memory-intensive workloads. The cost associated with tracking accessed sampled pages would outweigh the performance gain obtained from using MRCs.

An alternative approach is to periodically scan all sampled pages, to determine whether they have been accessed, by walking a page table to check the accessed bits of those pages. For processors that support accessed bits, the accessed bit of a page is automatically set when that page is accessed. When a sampled page is accessed:

- Its stack distance is computed using the splay tree
- It is then moved from its initial position to the top of the stack
- The computed stack distance is then scaled by $1/R$ to get the reuse distance
- The histogram count of the bucket that corresponds to the reuse distance is scaled by the bucket's scale factor before it is incremented [6]
- The accessed bit of the page is then clear for the next access

After all sampled pages are scanned, TLBs are then flushed so that the next time a sampled page is accessed, there is a TLB miss. This causes the page fault handler to walk the page table and the accessed bit of the sampled page is then set. The accessed sampled page can then be recorded during the next scanning process. This approach can construct approximate MRCs with very little overhead since it does not cause a page fault for every access to a sampled page, and it only periodically interferes with the execution of the VM.

4. Overhead Measurement

4.1 Time Overhead

As a result of the accessed bits scanning process, the execution of a VM is interrupted at regular intervals. The more frequently the scanning process is invoked, the greater the time overhead imposed on the system. Each scanning process involves many page walks to check the accessed bit of each sampled page. The larger the sample set size, the more page walks are required. Therefore, the time overhead imposed by the scanning process depends on both the sample set size and the scanning frequency. Their impacts can be determined by looking at the change in the percentage of the total execution time the VM spends on the scanning process when different scanning frequencies and sample set sizes are used.

Each scanning process also imposes an adverse side effect on the overall time overhead. Since TLBs are flushed after each scanning process, all mappings inside TLBs are lost each time the scanning process is done. As a result, there is an increase in the number of TLB misses when the scanning process is done frequently. Handling a TLB miss requires a page walk to fetch the lost address translation from a page table, thus incurring some time overhead. The cost associated with these TLB misses can be determined by looking at the change in the execution time of the workload when the scanning frequency is high.

The following experiments were conducted to determine how the time overhead changes with the sample set size and the frequency of the scanning process.

4.1.1 Experimental Setup

The host machine ran VMware ESXi 6.1.0 and was configured with 512GB RAM and 8 six-core 2.0 GHz AMD Opteron(tm) processors. The VM that ran on this host machine ran the Ubuntu 14.04 operating system and was configured with 128GB vRAM and 48 vCPUs. The fixed size Compiler workload of the SPECjvm2008 benchmark was used to analyze the time overhead. The number of benchmark threads used was 4 and the number of iterations of the

workload was 30. The maximum Java heap size, the minimum Java heap size, and the initial Java heap size were set to 128GB, 116GB, and 128GB, respectively. Three different scanning frequencies were tested: 10 Hz, 100 Hz, and 1000 Hz. Four different sample set sizes were tested for each frequency: 1024, 2048, 4096, and 8192. The average execution time spent on the scanning process was measured at the end of each experiment.

4.1.2 Experimental Results

Table 1 shows results of experiments conducted using different combinations of scanning frequencies and sample set sizes. Without incorporating the MRC implementation into VMware ESX, the total execution time of the Compiler workload, the base execution time, was 1959s. As Table 1 shows, based on the percentage of the total execution time the VM spent on the scanning process (*%Total*), the time overhead imposed by the scanning process is nearly negligible, close to 0%, for all chosen frequencies and sample set sizes. This minimal overhead suggests that all the implemented algorithms and data structures used to construct MRCs are very efficient.

Table 1: Time overhead imposed by the accessed bits scanning process for different sample set sizes and scanning frequencies. *%Total* is the percentage of the total execution time (not the workload execution time) the VM spent on the scanning process.

Frequency	Sample set size	Workload execution time (s)	%Total
10 Hz	1024	1916	0.0
	2048	1933	0.0
	4096	1999	0.0
	8192	1943	0.0
100 Hz	1024	2080	0.0
	2048	2261	0.0
	4096	2008	0.0
	8192	2301	0.0
1000 Hz	1024	2217	0.0
	2048	2263	0.0
	4096	2411	0.0
	8192	2451	0.0

For the 10 Hz frequency, the total execution times of the Compiler workload for all sample set sizes are very close to the base execution time. This indicates that TLB misses have very little impact on the execution of the workload at this frequency. However, for higher frequencies, 100 Hz and 1000 Hz, the total execution times of the workload for all sample set sizes are higher than the base execution time. This suggests that even though the total execution time spent on the scanning process might be negligible, the cost associated with the increased number of TLB misses due to a high frequency of TLB flushes is no longer trivial. Therefore, the scanning process should not be done at high frequencies since the increased number of TLB misses can interfere with the execution of the VM. To construct all subsequent MRCs, the 10 Hz scanning frequency was chosen because, regardless of the sample set size, this frequency results in trivial overall time overhead.

4.2 Space Overhead

Data structures such as splay trees, hash tables, max heaps, and arrays used to represent histograms all take up some memory space. The sizes of some of these data structures are dependent on the sample set size since they are used to store information related to sampled pages. With an upper limit on the sample set size, these data structures can all be statically allocated since their maximum sizes are known at compile-time. Therefore, since no data structures are created dynamically, the space overhead introduced by the MRC implementation can be determined by looking at the change in the size of the compiled code. Table 2 shows the space overhead imposed by the MRC implementation for different sample set sizes (number of samples) and histogram sizes (number of buckets). As this table shows, the total space overhead is less than 1 MB for all chosen sample set sizes and histogram sizes. This small space overhead suggests that the MRC implementation is efficient and can be incorporated into the memory scheduler of a hypervisor.

Table 2: Space overhead imposed by the MRC implementation. *text* is the code segment, *bss* is the uninitialized data segment, *rodata* is the read-only data segment and *data* is the initialized data segment that can be modified.

Sample set size	Histogram size	Change in size (bytes)				
		text	bss	rodata	data	total
512	512	4868	0	248	56k	61k
	1024	4868	0	248	68k	73k
	2048	4868	0	248	92k	97k
	4096	4868	0	248	140k	145k
	8192	4868	0	248	236k	241k
1024	512	4868	0	248	100k	105k
	1024	4868	0	248	112k	117k
	2048	4868	0	248	136k	141k
	4096	4868	0	248	184k	189k
	8192	4868	0	248	280k	285k
2048	512	4868	0	248	188k	193k
	1024	4868	0	248	200k	205k
	2048	4868	0	248	224k	229k
	4096	4868	0	248	272k	277k
	8192	4868	0	248	368k	373k
4096	512	4868	0	248	364k	369k
	1024	4868	0	248	376k	381k
	2048	4868	0	248	400k	405k
	4096	4868	0	248	448k	453k
	8192	4868	0	248	544k	549k
8192	512	4868	0	248	716k	721k
	1024	4868	0	248	728k	733k
	2048	4868	0	248	752k	757k
	4096	4868	0	248	800k	805k
	8192	4868	0	248	896k	901k

5. Sample Set Size

The sample set size is one of the main factors that determines the accuracy of MRC construction. With a large sample set, more pages are required to be tracked and each sampled page therefore represents only a small number of accessed pages, thus leading to more accurate MRC construction. For example, an exact MRC can be constructed if the sample set size is equal to the total number of unique accessed pages, assuming that the cost of exact MRC construction can be ignored. However, the study presented in [6] showed that only a small sample set is enough to produce a good approximate MRC. The following experiments were conducted to study how the shape of an MRC changes when different sample set sizes were used in MRC construction, and to determine the minimum sample set sizes that would result in good approximate MRCs of small and large VMs. In these experiments, for each VM, MRCs were constructed using sample sets of different sizes. These MRCs were then combined to show how the shape of the MRC changes with the sample set size.

Since constructing an exact MRC would drastically slow the execution of the VM, it would be difficult to determine the shape of the exact MRC and hence to assess whether the constructed MRCs are close to the exact MRC. However, since a larger sample set size leads to more accurate MRC construction, as claimed in [6], the tail of the constructed MRC is expected to converge to the actual working set size as the sample set size increases. Furthermore, even though the shape of the exact MRC is not known, the tail of the exact MRC can be determined using a performance graph, a graph that plots the execution time of a workload running in the VM vs total available memory given to the VM. The tail of the exact MRC corresponds to the minimum total available memory size, found in the performance graph, that leads to the optimal performance of the VM. This is the actual working set size of the VM. In this case, as the sample set size increases, the tail of the constructed MRC is expected to approach the actual working set size obtained from the performance graph.

5.1 Small Virtual Machine

For a small VM with 4GB vRAM and sampled pages of size 2MB each, a sample set of size 4GB/2MB = 2048 is enough to represent every page in the VM's vRAM.

5.1.1 Experimental Setup

The host machine ran VMware ESXi 6.1.0 and was configured with 24GB RAM and 2 quad-core 2.3 GHz AMD Opteron(tm) processors. The VM that ran on this host machine ran the Ubuntu 14.04 operating system and was configured with 4GB vRAM and 4 vCPUs. The fixed size xml.transform workload of the SPECjvm2008 benchmark was chosen to run on this VM. The number of benchmark threads used was 4 and the number of iterations of the workload was 50. The maximum Java heap size, the minimum Java heap size, and the initial Java heap size were set to 4GB, 2GB, and 4GB, respectively. The frequency of accessed bits scanning process was 10 Hz. The sample set sizes used for MRC construction were 64, 128, 256, 512, 1024, and 2048.

5.1.2 Experimental Results

Without incorporating the MRC implementation into VMware ESX, the execution time of the xml.transform workload, the base execution time, was 520 seconds. The execution time of the workload for each sample set size used in MRC construction is shown in Table 3. This table suggests that with a 10 Hz scanning frequency, regardless of the sample set size, the scanning process does not interfere with the execution of the small VM since the execution time of the workload for each sample set size is very similar to the base execution time. This table also suggests there is little overhead imposed by the MRC implementation, regardless of the sample set size.

Table 3: Different sample set sizes used in MRC construction and the corresponding execution times of the xml.transform workload.

Sample set size	Execution time (s)
64	509
128	512
256	496
512	511
1024	511
2048	520

Table 4 shows the execution time of the xml.transform workload for each total available memory size given to the small VM. The performance graph of this VM is plotted in Figure 15. This performance graph shows that the minimum total available memory size that led to the optimal performance of the VM, the actual working set size, was approximately 3GB.

Table 4: Different total available memory sizes given to the small VM, with 4GB vRAM, and the corresponding execution times of the xml.transform workload running in the VM.

Total available Memory (GB)	Execution time (s)
4.0	498
3.5	498
3.0	502
2.9	557
2.8	636

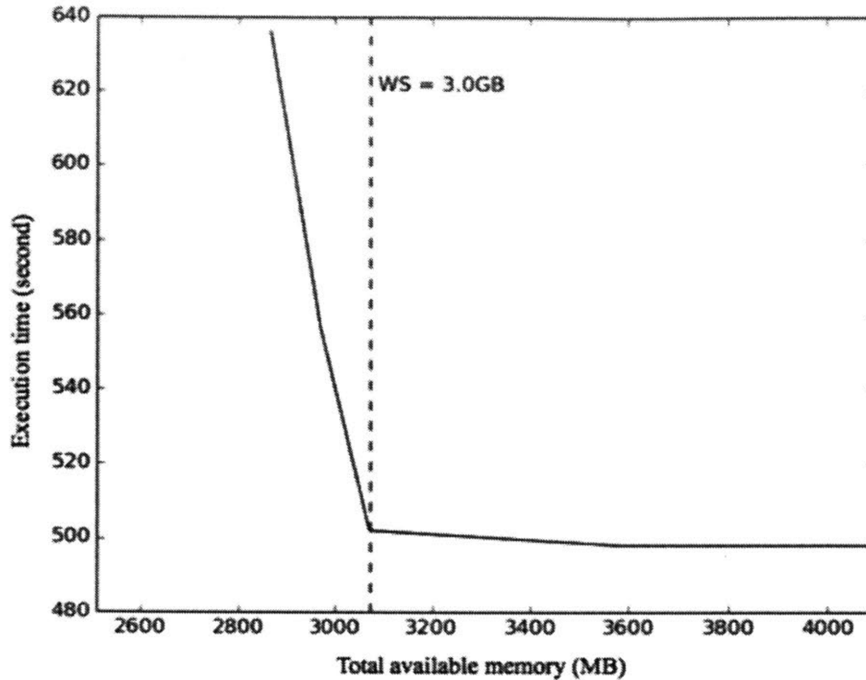


Figure 15: Performance graph of the small VM running the fixed size xml.transform workload of the SPECjvm2008 benchmark. This graph shows that the actual working set size of the VM is approximately 3GB.

Snapshots of MRCs, built using sample sets of different sizes, were taken 5 minutes after the workload started. A combination of these snapshots is shown in Figure 16. The shape of the MRC started to converge once the sample set size exceeded 256. Furthermore, the tail of the MRC also started to approach the actual working set size, 3GB, once the sample set size exceeded 256. Snapshots of MRCs, built using only sample sets of sizes greater than 256, are shown in Figure 17. Both Figure 16 and Figure 17 suggest that even though a large sample set with 2048 sampled pages is required to construct an exact MRC for this small VM, a small sample set with only 512 sampled pages is more than enough to produce an approximate MRC that is very close to the exact MRC.

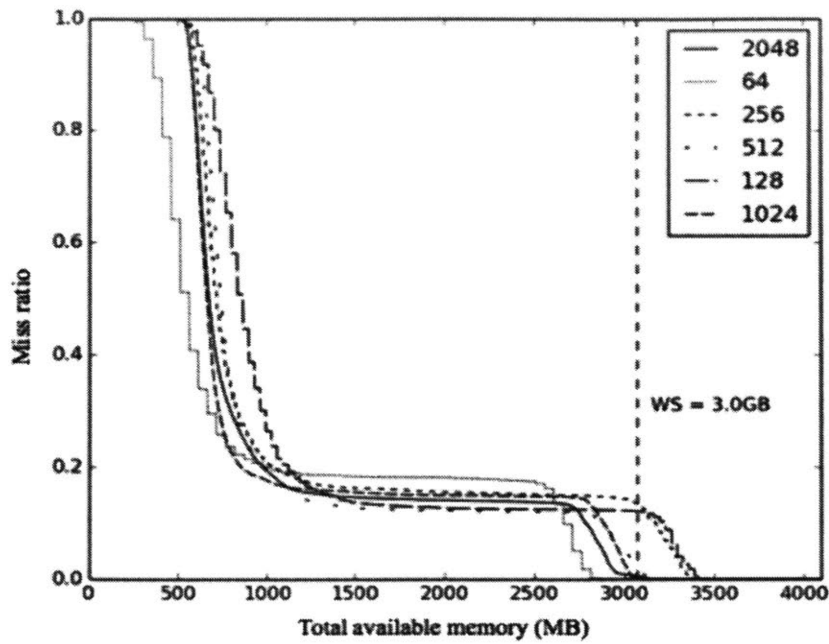


Figure 16: Snapshots of MRCs constructed using sample sets of different sizes. Each snapshot was taken 5 minutes after the workload started.

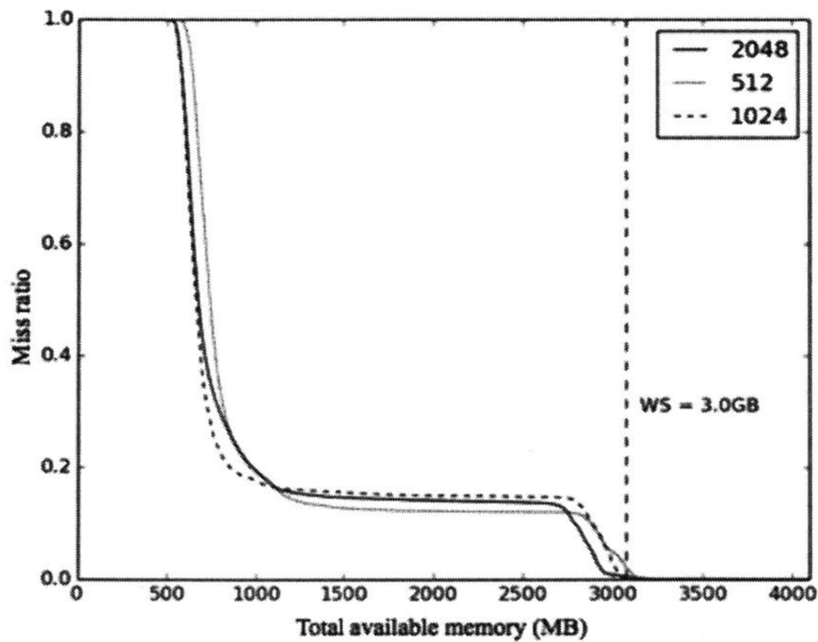


Figure 17: Snapshots of MRCs constructed using only sample sets of sizes greater than 256. This figure suggests that a sample set of size 512 was enough to produce good approximate MRCs for this small VM.

5.2 Large Virtual Machine

For a large VM with 128GB vRAM and sampled pages of size 2MB each, a sample set of size up to 128GB/2MB = 65536 is required to represent every page in the VM's vRAM.

5.2.1 Experimental Setup

The host machine ran VMware ESXi 6.1.0 and was configured with 512GB RAM and 8 six-core 2.0 GHz AMD Opteron(tm) processors. The VM that ran on this host machine ran the Ubuntu 14.04 operating system and was configured with 128GB vRAM and 48 vCPUs. The fixed size compiler.sunflow workload of the SPECjvm2008 benchmark was chosen to run on this VM. The number of benchmark threads used was 4 and the number of iterations of the workload was 3. The maximum Java heap size, the minimum Java heap size, and the initial Java heap size were set to 128GB, 116GB, and 128GB, respectively. The frequency of the accessed bits scanning frequency was 10 Hz. The chosen sample set sizes were 256, 512, 1024, 2048, 4096, and 8192.

5.2.2 Experimental Results

Without MRC implementation incorporated into VMware ESX, the execution time of the compiler.sunflow workload, the base execution time, was 207 seconds. Table 5 shows the execution time of the compiler.sunflow workload for each sample set size used in MRC construction. This table shows that with a 10 Hz scanning frequency, regardless of the sample set size, the scanning process does not interfere with the execution of the large VM and the time overhead imposed by the MRC implementation is minimal since the execution time of the compiler.sunflow workload for each sample set size is similar to the base execution time.

Table 5: Different sample set sizes used in MRC construction and the corresponding execution times of the compiler.sunflow workload.

Sample set size	Execution time (s)
256	206
512	208
1024	222
2048	217
4096	209
8192	226

Table 6 shows the execution times of the compiler.sunflow workload for different total available memory sizes given to the large VM. The performance graph of the large VM is plotted in Figure 18. This performance graph shows that the execution time of the compiler.sunflow workload started to increase once the total available memory size fell below 87.5GB. As a result, the minimum total available memory size that led to the optimal performance of the VM, the actual working set size of the VM, was approximately 87.5GB.

Table 6: Different total available memory sizes given to the large VM, with 128GB vRAM, and the corresponding execution times of the compiler.sunflow workload running in the VM.

Total available Memory (GB)	Execution time (s)
128	220
120	220
110	221
100	218
88	223
87.5	217
87.4	281
87.3	303
87.2	315

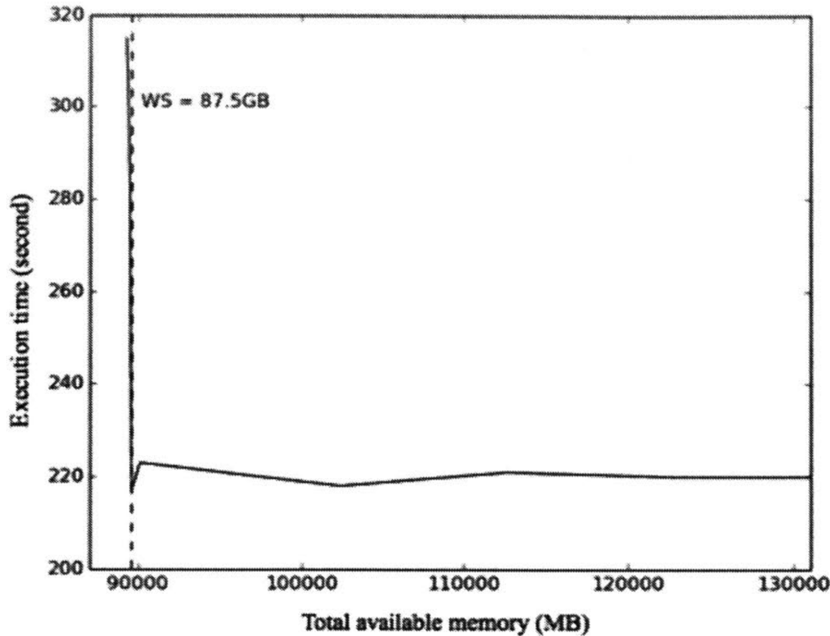


Figure 18: Performance graph of the large VM running the compiler.sunflow workload. This graph shows that the actual working set size of the VM is approximately 87.5GB.

Snapshots of MRCs, built using sample sets of different sizes, were taken 5 minutes after the workload started. The combination of these snapshots is shown in Figure 19. As this figure shows, the shape of the MRC started to converge once the sample set size exceeded 512. Furthermore, the tail of the MRC also started to approach the actual working set size, 87.5GB, once the sample set size exceeded 512. In this case, a sample set of size 1024 was more than enough to produce good approximate MRCs for this large VM. The combination of snapshots of MRCs, built using only sample sets of sizes greater than 512, is shown in Figure 20.

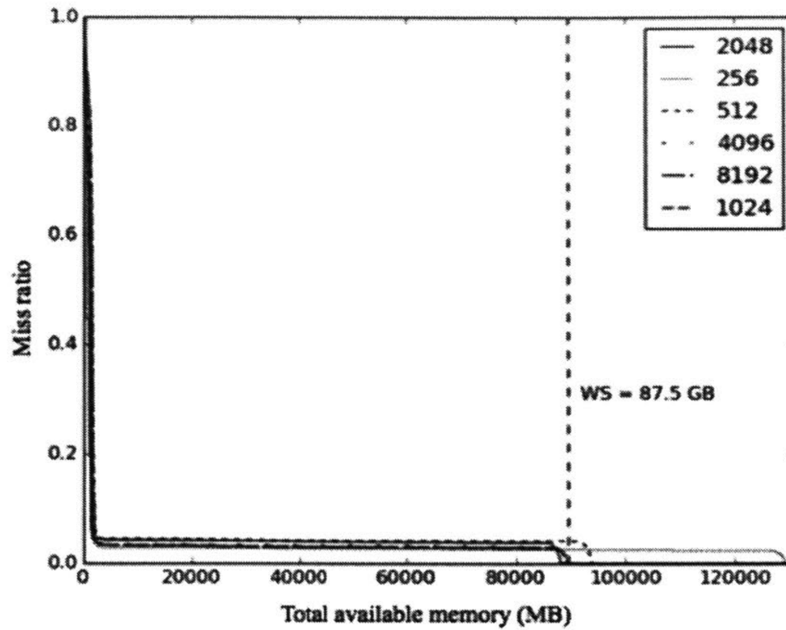


Figure 19: Snapshots of MRCs constructed using sample sets of different sizes. Each snapshot was taken 5 minutes after the compiler.sunflow workload started.

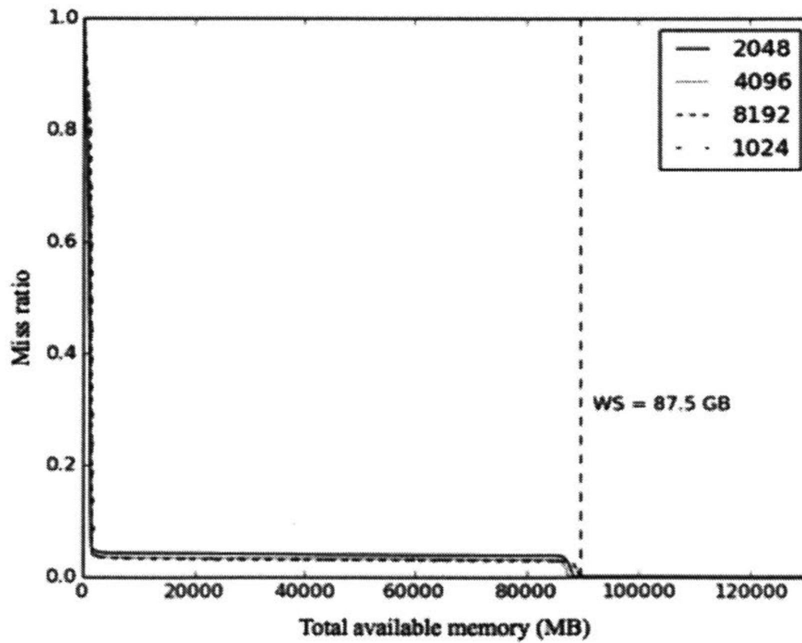


Figure 20: Snapshots of MRCs constructed using only sample sets of sizes greater than 512. The relative shapes of these MRCs were very similar to each other. The tails of these MRCs were also very close to the actual working set size of the VM. This suggests that a sample set of size 1024 was enough to produce good approximate MRCs for this large VM.

6. Bucket Size

For very large VMs, the memory space allocated for all histograms used in MRC construction can be huge, especially when each histogram bucket accommodates only one reuse distance. It is therefore necessary to have each bucket accommodate multiple reuse distances when the size of the VM's vRAM is large, in order to reduce the space overhead imposed by the MRC implementation. The following experiments were conducted to investigate how bucket size might affect MRC construction.

6.1 Small Virtual Machine

6.1.1 Experimental Setup

The host machine ran VMware ESXi 6.1.0 and was configured with 24GB RAM and 2 quad-core 2.3 GHz AMD Opteron(tm) processors. The small VM that ran on this host machine ran the Ubuntu 14.04 operating system and was configured with 4GB vRAM and 4 vCPUs. The workload chosen to run on this VM was the fixed size xml.validation workload of the SPECjvm2008 benchmark. The maximum Java heap size, the minimum Java heap, and the initial Java heap size were set to 4GB, 2GB, and 4GB, respectively. The number of benchmark threads used was 4 and the number of iterations was 10. The sample set size was 2048. Many experiments were conducted to construct MRCs with different bucket sizes, ranging from 2 to 256. For each bucket size, a snapshot of the MRC was taken 5 minutes after the VM started. All snapshots were then combined and compared to determine whether a large bucket size has any impact on MRC construction.

6.1.2 Experimental Results

Figure 21 shows a combination of all snapshots of MRCs. This figure shows that most MRCs were almost identical to each other. Regardless of the bucket size, the tail of the MRC always ended at about the same total available memory size. This suggests that the bucket size

has little influence on the MRC construction and, if the tail of the curve is used to estimate the working set size of the VM, a large bucket size would not affect the estimated working set size. The next experiments were conducted to confirm whether these observations still hold true when MRCs are constructed for a larger VM.

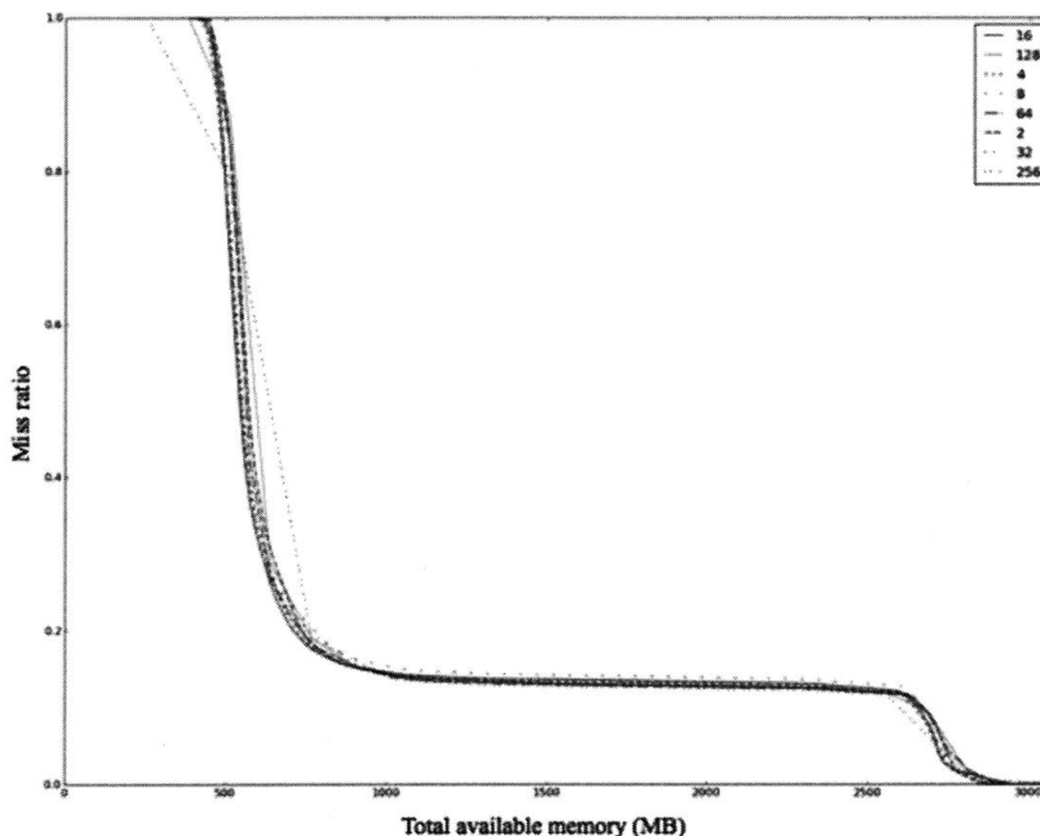


Figure 21: Snapshots of MRCs, constructed using different bucket sizes, were recorded 5 minutes after the xml.validation workload started. Most of these MRCs looked almost identical to each other. Notice that regardless of the bucket size, the tail of the curve always ended at about the same total available memory size. This suggests that the bucket size has little influence on the MRC construction and the estimated working set size.

6.2 Large Virtual Machine

6.2.1 Experimental Setup

The host machine ran VMware ESXi 6.1.0 and was configured with 512GB RAM and 8 six-core 2.0 GHz AMD Opteron(tm) processors. The large VM that ran on this host machine ran the Ubuntu 14.04 operating system and was configured with 128GB vRAM and 48 vCPUs. The same xml.validation workload of the SPECjvm2008 benchmark was used to run on this VM

except that the maximum Java heap size, the minimum Java heap size, and the initial Java heap size were increased to 128GB, 116GB, and 128GB, respectively.

6.2.2 Experimental Results

Figure 22 shows a combination of snapshots of MRCs, each constructed using a different bucket size ranging from 16 to 512, and taken 5 minutes after the VM started. As in Figure 21, all MRCs shown in Figure 22 looked very similar to each other and, regardless of the bucket size, the tail of the curve always ended at about the same total available memory size. This confirms that the bucket size has little influence on the MRC construction and, if the tail of the MRC is used to estimate the working set size of a VM, a large bucket size would not affect the estimated working set size.

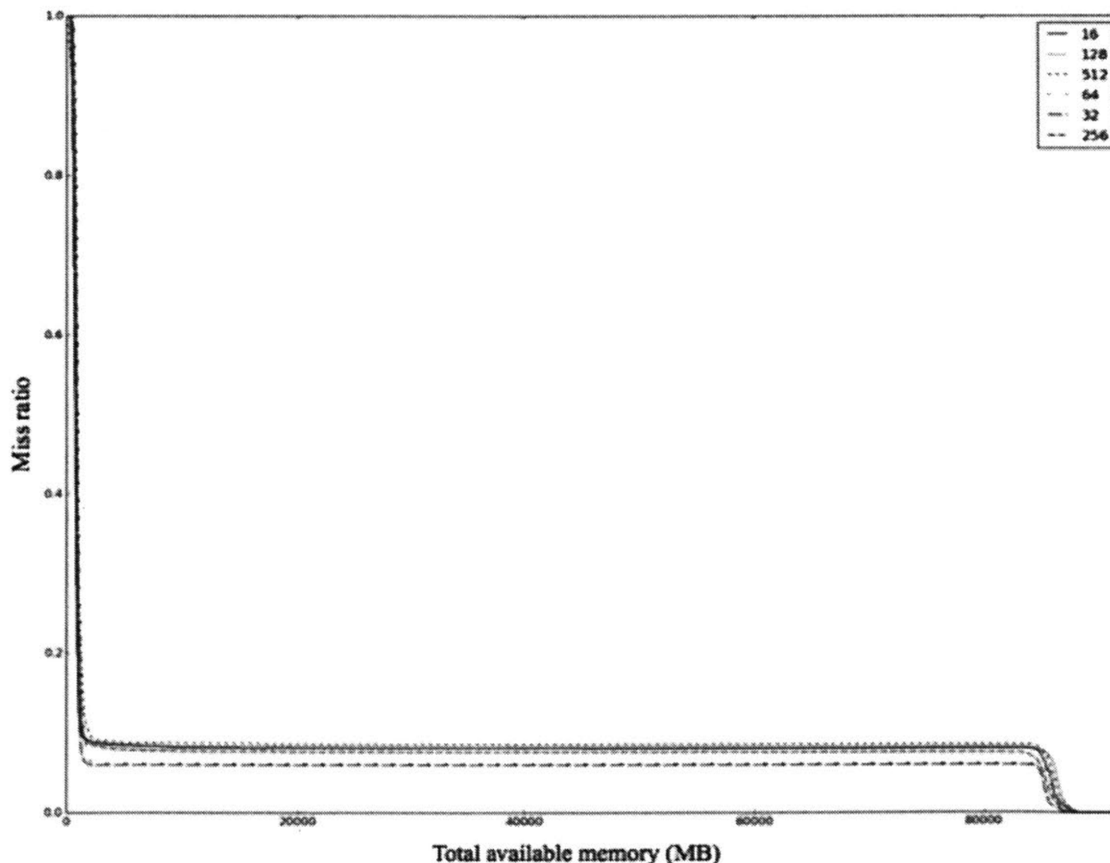


Figure 22: Snapshots of MRCs, constructed using different bucket sizes, were recorded 5 minutes after the xml.validation workload started. Regardless of the bucket size, the tail of the MRC always ended at about the same total available memory size. Together with Figure 21, this confirms that the bucket size has little influence on the MRC construction and the estimated working set size.

7. Evolution of Sampling Rate

The fixed size implementation of MRC construction requires a dynamic adjustment of the sampling rate in order to fix the sample set size. For the sampling condition $hash(X) \bmod P < T$, the initial value of the threshold T is assigned to the value of the modulus P. In this case, the initial value of the sampling rate $R = T/P$ is 1. After a new page is added to the already full sample set, the page with the highest threshold is removed from the sample set to restore the sample set size to its maximum value. The value of the threshold T is then lowered to the new value of the highest threshold. As a result, the threshold T is gradually lowered as new pages are added to the sample set and the sampling rate R gradually decreases as more pages are accessed. The following experiments were conducted to study the evolution of the sampling rate R and how it affects the MRC when different workloads run in the VM.

7.1 Intensive Workloads

An intensive workload is a workload that has a high memory bandwidth requirement. When a VM runs this type of workload, most of the pages in the VM's vRAM are touched. As a result, the sampling rate would converge to some stable value when the entire memory space is explored. The experiments below were conducted to study how sampling rates evolve and to determine their stable values when VMs run this type of workload.

7.1.1 Small Virtual Machine

7.1.1.1 Experimental Setup

The host machine ran VMware ESXi 6.1.0 and was configured with 24GB RAM and 2 quad-core 2.3 GHz AMD Opteron(tm) processors. The VM that ran on this host machine ran the Ubuntu 14.04 operating system and was initially configured with 4GB vRAM and 4 vCPUs. The chosen workload was the fixed size sunflow workload of the SPECjvm2008 benchmark. The number of benchmark threads was 4 and the number of iterations of the workload was 10. The maximum Java heap size, the minimum Java heap size, and the initial Java heap size were

initially set to 4GB, 2GB, and 4GB, respectively. These heap sizes were chosen to ensure that nearly every page in the the VM's vRAM was touched when the workload ran. The sample set size used was 64.

7.1.1.2 Experimental Results

Figure 23 shows the evolution of the sampling rate R, recorded over an hour period after the VM started. The workload took 1949 seconds (or 32 minutes) to complete all the 10 iterations. The VM was then left to run, without running the sunflow workload, for another half an hour. Figure 23 suggests that the sampling rate *R* tends to converge to a stable value as the execution of the VM progresses. With a sampling rate *R*, each sampled page represents about *1/R* pages and the approximate number of represented pages is therefore *(sample set size) * (1/R)*. Since each represented page is a 2MB page, the size of the represented memory space is therefore approximately *2 * (sample set size) * (1/R)* MB. As shown in Figure 23, the sampling rate *R* approached a stable value equal to 0.03125. The approximate size of the represented memory space was therefore equal to 2 * 64 * (1/0.03125) = 4096MB = 4GB. This was also the size of the VM's vRAM. In this case, Figure 1 suggests that the sampling rate *R* converges to a stable value R_{stable} that allows the represented memory space to cover the entire memory space of the VM. This stable sampling rate R_{stable} is equal to *2 * (sample set size) * 1/(size of vRAM in MB)* which is equivalent to the ratio of the sample set size to the total number of large pages inside the VM's vRAM.

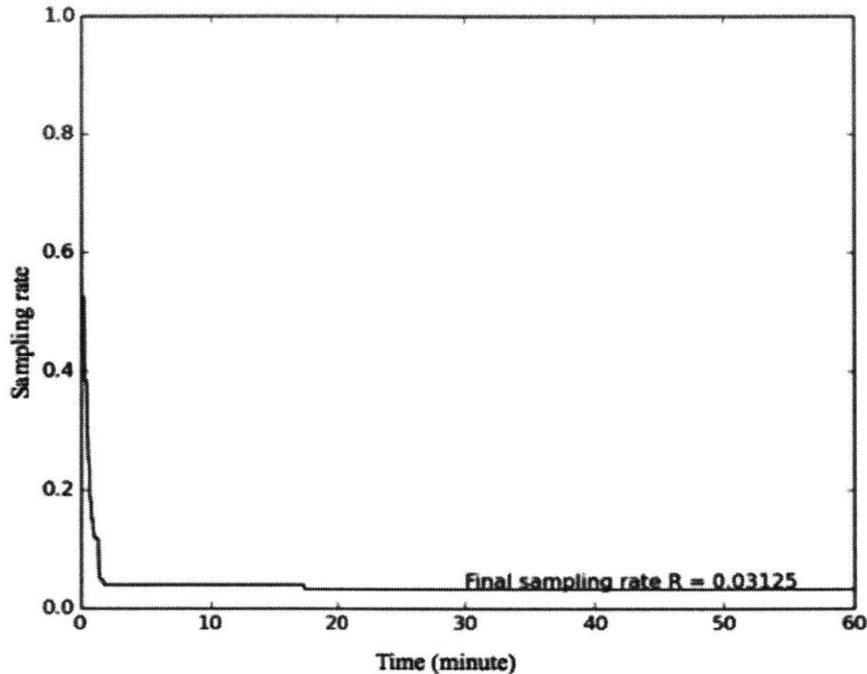


Figure 23: Evolution of the sampling rate R when a small VM with 4GB vRAM ran the sunflow workload of the SPECjvm2008 benchmark.

Two more experiments were conducted to confirm that the derived stable sampling rate is correct. In these experiments, the same host machine and VM were used. However, the size of the VM's vRAM was doubled to 8GB for the first experiment and quadrupled to 16GB for the second experiment. The same workload was also used. However, the maximum Java heap size, the minimum Java heap size, and the initial Java heap size were doubled to 8GB, 4GB, and 8GB, respectively, for the first experiment and quadrupled to 16GB, 8GB, and 16GB, respectively, for the second experiment. These heap sizes were chosen to ensure that every page in the VM's vRAM was touched. Figure 24 plots the evolution of the sampling rates for the first and the second experiments. For the first experiment, the workload took 1849 seconds (or 31 minutes) to run to completion. For the second experiment, the workload took 1792 seconds (or 30 minutes) to run to completion. As Figure 24 shows, when the VM's vRAM was doubled from 4GB to 8GB, the final sampling rate was halved from 0.03125 to 0.015625 and when the VM's vRAM was doubled again from 8GB to 16GB, the final sampling rate was also halved again from 0.015625 to 0.0078125. This consistency suggests that the derived stable sampling rate R_{stable} is correct and that the sampling rate R converges to this value as the entire memory space of the VM is explored.

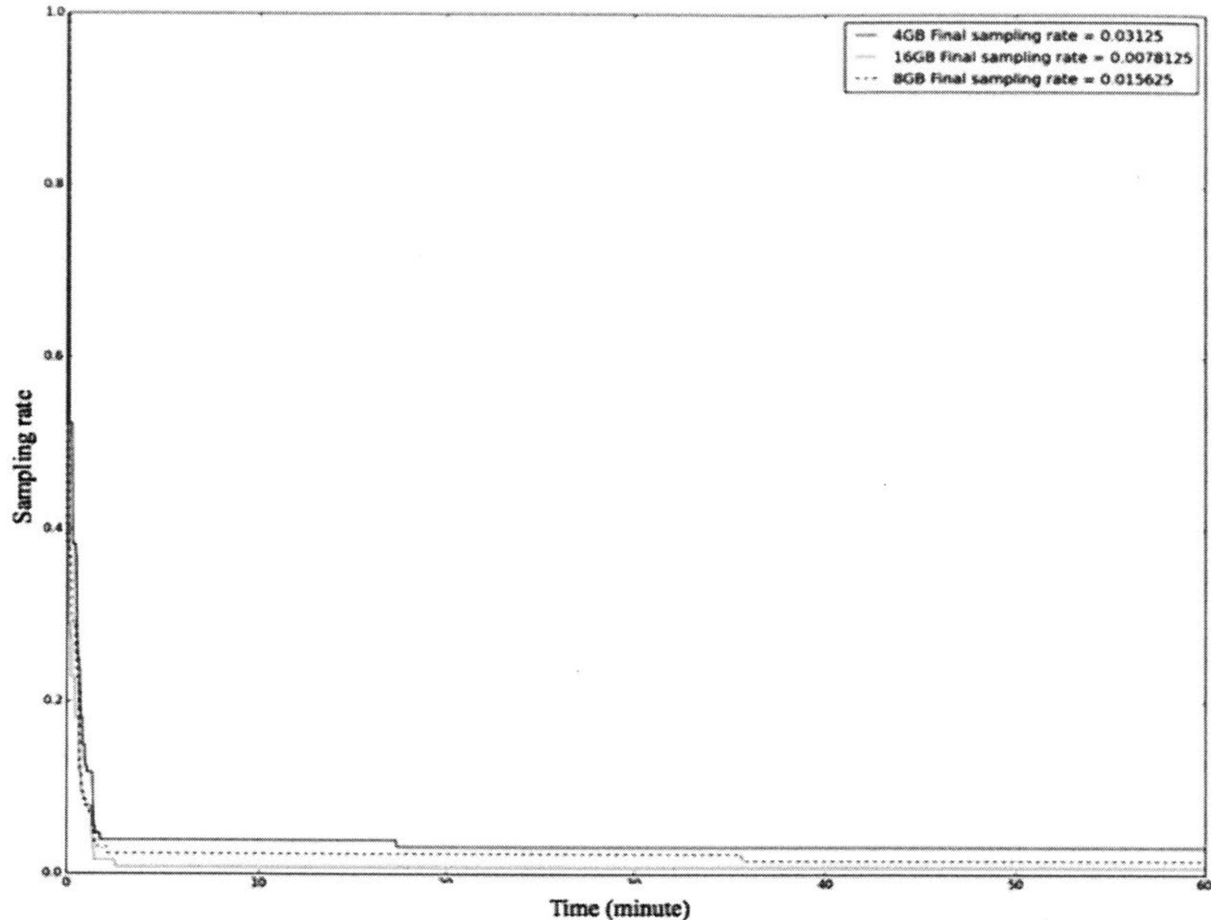


Figure 24: Evolution of sampling rates when a small VM ran the sunflow workload and was configured with different sample set sizes: 4GB, 8GB, and 16GB.

The next experiment was conducted to illustrate that the derived stable sampling rate R_{stable} is still correct even though the VM ran a different memory-intensive workload and was configured with a larger vRAM. A much larger sample set size was also used to confirm the correctness of the derived stable sampling rate.

7.1.2 Large Virtual Machine

7.1.2.1 Experimental Setup

The host machine ran VMware ESXi 6.1.0 and was configured with 512GB RAM and 8 six-core 2.0 GHz AMD Opteron(tm) processors. The VM that ran on this host machine ran the Ubuntu 14.04 operating system and was configured with 128GB vRAM and 48 vCPUs. The

chosen workload was the workload of the SPECjbb2013-Composite benchmark. This workload was allowed to run to completion once and the sampling rate was recorded while the workload was running. The maximum Java heap size, the minimum Java heap size, and the initial Java heap size were assigned to 128GB, 116GB, and 128GB, respectively. The sample set size was 8192.

7.1.2.2 Experimental Result

The workload of the SPECjbb2013-Composite benchmark took a total of 8504 seconds (or 142 minutes) to run to completion. Figure 25 plots the evolution of the sampling rate R of the VM. Since the size of the VM's vRAM was 128GB, the total number of large pages was 65536. The stable sampling rate R_{stable} was therefore equal to 8192/65536 = 0.125. As Figure 25 shows, the recorded sampling rate R was 0.126037597656. This sampling rate was very close to the predicted stable sampling rate R_{stable}. This suggests that the derived stable sampling rate is correct and that the sampling rate of the VM converges to this stable value as more pages inside the VM's vRAM are accessed.

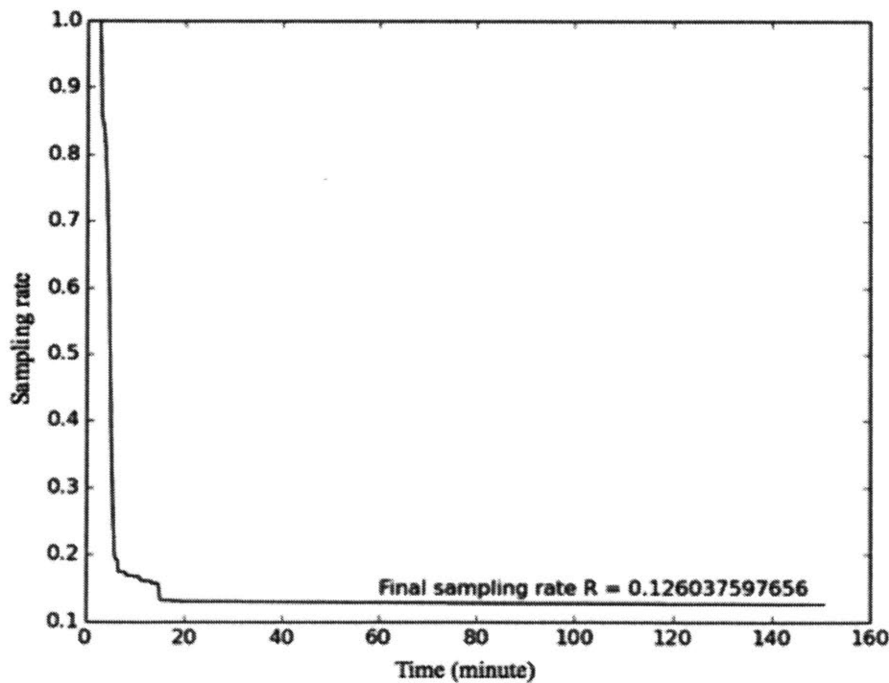


Figure 25: Evolution of the sampling rate when the large VM ran the workload of the SPECjbb2013-Composite benchmark.

7.2 Light Workloads

The above experiments were conducted using memory-intensive workloads, where most pages inside the VM's vRAM are accessed. When the workload is light, however, some of the pages might not be explored. As a result, the sampling rate R might not converge to its stable value R_{stable}. The following experiments were conducted to illustrate this problem.

7.2.1 Experimental Setup

The host machine ran VMware ESXi 6.1.0 and was configured with 24GB RAM and 2 quad-core 2.3 GHz AMD Opteron(tm) processors. The VM that ran on this host machine ran the Ubuntu 14.04 operating system and was configured with 8GB vRAM and 4 vCPUs. The sample set size used was 64. A light workload, the bzip2 workload of the SPEC CPU2006 benchmark, was chosen to run for 1 iteration.

7.2.2 Experimental Results

The bzip2 workload took 853 seconds (or 14 minutes) to run to completion. Figure 26 shows the evolution of the sampling rate R recorded over a 200-minute period. Since the VM's vRAM was 8GB and each page was 2MB, the total number of pages was 4096 and the stable sampling rate R_{stable} was equal to 64/4096 = 0.015625. However, the recorded final sampling rate R (= 0.0234375) was not very close to its stable value R_{stable}. This is because the workload was light and, therefore, there were not many accessed pages. The threshold T no longer decreased after those few accessed pages were put into the filter and the sampling rate could not converge to its stable value. To illustrate that 0.0234375 was not the stable sampling rate, the bzip2 workload was then allowed to run for 10 iterations instead of 1. The workload took 8504 seconds (or 142 minutes) to finish all the 10 iterations. The final sampling rate recorded was 0.015625, as shown in Figure 27. This suggests that even if the workload is light, the sampling rate still converges to its stable value if the workload is allowed to run for multiple iterations. This is because each iteration of the workload might not involve the same part of the memory space. Eventually, most pages inside the VM's vRAM are touched and the sampling rate then

converges to its stable value. The convergence time was, however, quite long, as shown in Figure 27. It took two hours for the sampling rate to converge to its stable value. This suggests that the convergence of the sampling rate depends largely on the workload running in the VM. The sampling rate should take a long time to converge if the VM runs a light workload and a short time if the VM runs a memory-intensive workload.

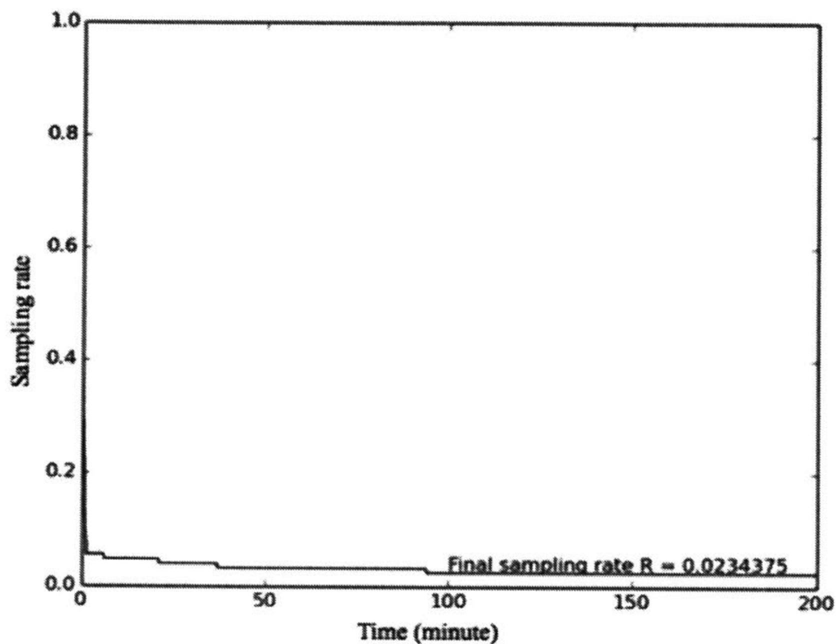


Figure 26: Evolution of the sampling rate when the VM ran the bzip2 workload for 1 iteration. The sample rate did not converge to its stable value (0.015625).

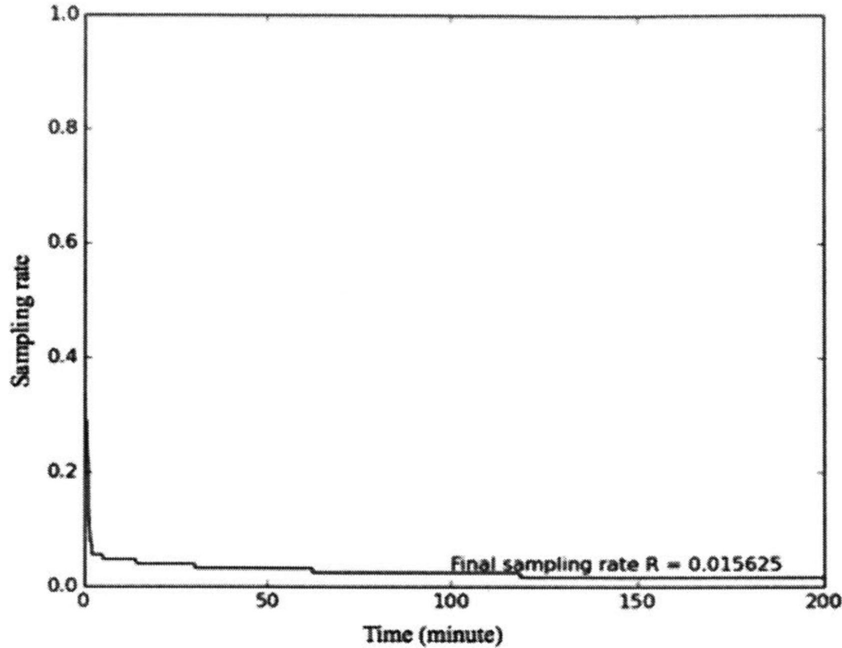


Figure 27: Evolution of the sampling rate when the VM ran the bzip2 workload for 10 iterations. The sampling rate converged to its stable value.

7.3 Implication

Since the reuse distance is computed by scaling the stack distance by *1/(sampling rate)*, the sampling rate is one of the main factors that determine the shapes of constructed MRCs. If the sampling rate has not yet reached its stable value, the scaling factor, *1/(sampling rate)*, would be small and the shape of the constructed MRCs would be incorrect. The following experiments were conducted to investigate this problem. In these experiments, the host machine ran VMware ESXi 6.1.0 and was configured with 512GB RAM and 8 six-core 2.0 GHz AMD Opteron(tm) processors. The VM that ran on this host machine ran the Ubuntu 14.04 operating system and was configured with 128GB vRAM and 48 vCPUs. The workload chosen to run on this VM was the xml.validation workload of the SPECjvm2008 benchmark. The number of benchmark threads was 4 and the number of iterations of the workload was 5. The maximum Java heap size, the minimum Java heap size, and the initial Java heap size were assigned to 64GB, 52GB, and 64GB, respectively. The sample set size used in the MRC construction was 1024. In this case, the stable sampling rate of the VM was approximately 2*1024*1/(128*1024) = 0.015625. Table 7 shows the execution time of the xml.validation workload for each total available memory size

given to the VM, and the performance graph of the VM is plotted in Figure 28. This graph shows that the minimum total available memory size that leads to the optimal performance of the VM, the actual working set size of the VM, was approximately 44.2GB.

Table 7: Total available memory sizes given to the virtual machine and the corresponding execution times of the xml.validation workload.

Total available Memory (GB)	Execution time (s)
128	188
100	191
45	185
44.5	192
44.2	188
44	207
43.5	251
43	428
42	978

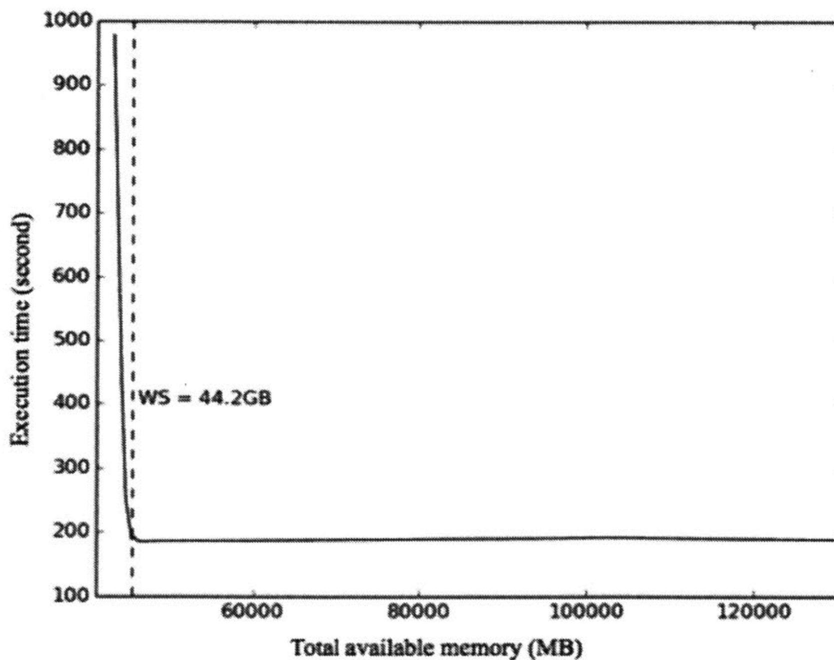


Figure 28: Performance graph of the VM running the fixed size xml.validation workload of the SPECjvm2008 benchmark. This graph shows that the actual working set size of this VM was approximately 44.2GB.

The total available memory size that corresponds to the tail of an accurate MRC is expected to be close to the actual working set size. Figure 29 plots the evolution of the sampling rate when the VM ran the xml.validation workload and Figure 30 shows a snapshot of the VM's MRC. The sampling rate of the VM, as shown in Figure 29, was nowhere near its stable value. The sampling rate converged to 0.0439453125, which was nearly 3 times its stable value. This was because the workload touched less than half the VM's memory space. As a result, many pages were not accessed and the sampling rate could not converge to its stable value. The tail of the MRC, as shown in Figure 30, was also nowhere near the actual working set size. The memory size that corresponds to the tail of the MRC was about 39GB, which was about 5.2GB less than the actual working set size. The mismatch between the tail of the MRC and the actual working set size might be due to the high sampling rate of the VM that caused the scale factor, *1/(sampling rate)*, to be low.

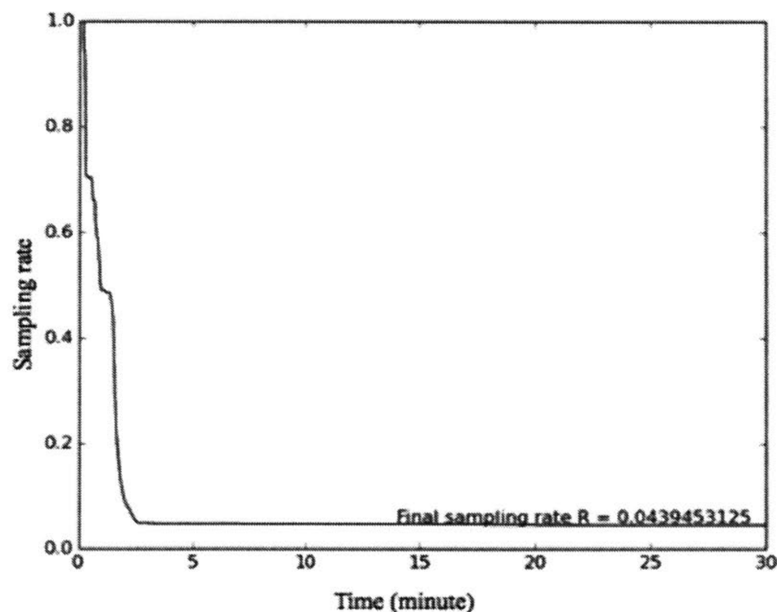


Figure 29: Evolution of the sampling rate when the VM ran the fixed size xml.validation workload of the SPECjvm2008 benchmark. The sampling rate did not converge to its stable value, 0.015625, since the majority of the VM's memory space was not yet explored.

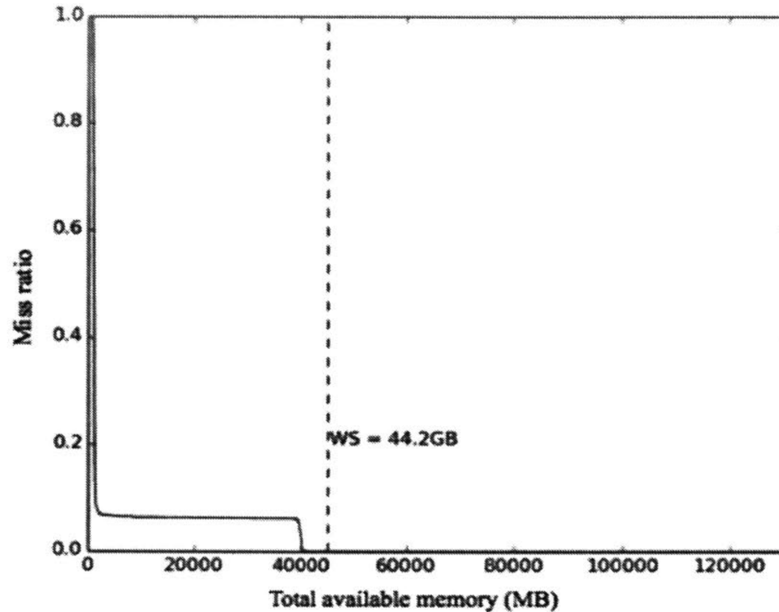


Figure 30: A snapshot of the MRC when the VM ran the fixed size xml.validation workload of the SPECjvm2008 benchmark and when the sampling rate was not close to its stable value. Notice that the tail of the MRC was nowhere near the actual working set size of the VM.

The experiment was repeated but the sampling rate was forced to converge to its stable value before the VM ran the xml.validation workload. This was done by running a very memory-intensive workload in the VM for many iterations until most pages inside the VM's memory space were touched. The chosen workload was the fixed size sunflow workload of the SPECjvm2008 benchmark. The number of benchmark threads was 4 and the number of iterations of the workload was 10. The maximum Java heap size, the minimum Java heap size, and the initial Java heap size were assigned to 150GB, 128GB, and 150GB, respectively. The heap sizes were chosen to be larger than or equal to the VM's vRAM to ensure that most pages were accessed by the workload. Once the sunflow workload ran to completion, the VM waited for 5 minutes before running the xml.validation workload. Figure 31 plots the evolution of the new sampling rate and Figure 32 shows a snapshot of the new MRC.

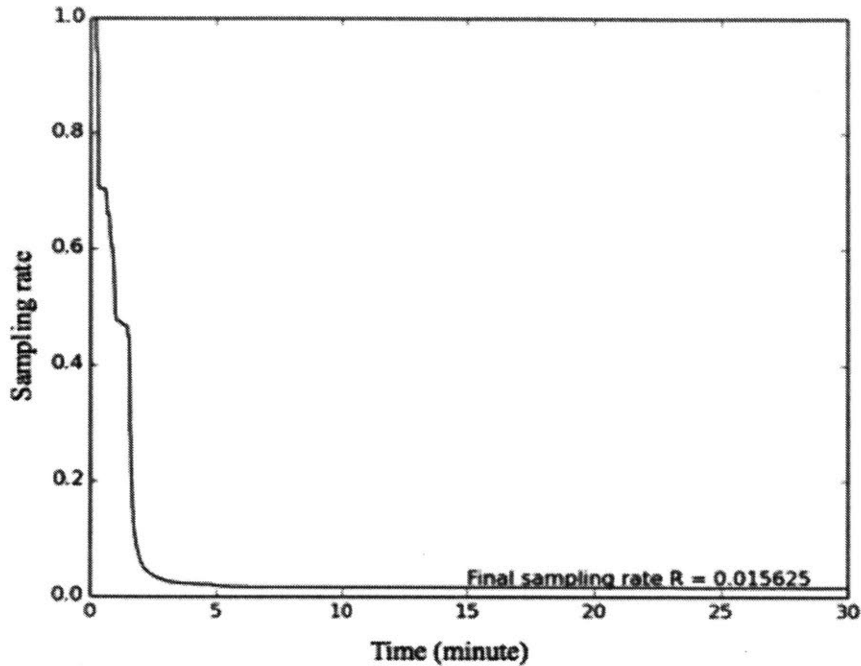


Figure 31: Evolution of the sampling rate when the VM ran the sunflow workload followed by the xml.validation workload. The sunflow workload was run for multiple iterations to ensure that the sampling rate reached its stable value before the xml.validation workload was run.

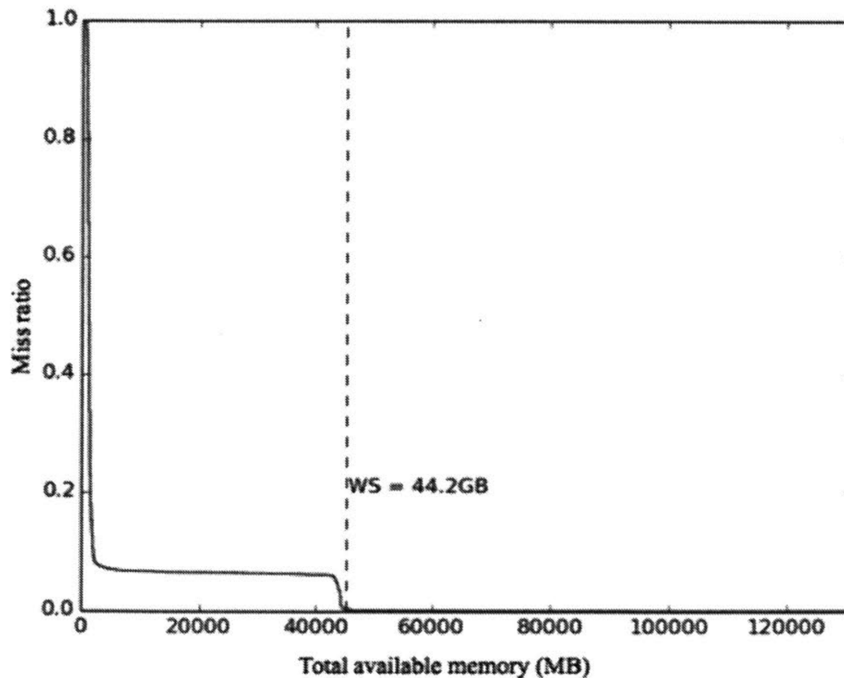


Figure 32: A snapshot of the MRC when the VM ran the sunflow workload followed by the xml.validation workload and when the sampling rate reached its stable value. Notice that the total available memory size that corresponds to the tail of the MRC is the same as the actual working set size of the VM.

Notice that the sampling rate converged to its stable value, shown in Figure 31, and the memory size that corresponded to the tail of the MRC also matched the actual working set size of the VM, shown in Figure 32. This suggests that it is very important that the sampling rate reaches its stable value in order for the MRC construction to be accurate. Furthermore, it can be inferred that the MRC construction is more accurate when VMs run memory-intensive workloads since these workloads force the sampling rates of these VMs to converge to their stable values much faster than light workloads.

8. Working Set Size Tracking

The size of a VM's working set is the minimum total available memory size given to the VM in order for the VM to achieve its optimal performance. The tail of an MRC is the minimum total available memory size given to the VM in order for the VM to achieve a zero miss rate. Once the VM's total available memory size falls below this tail, the VM starts to experience page faults and its performance starts to deteriorate. The tail of the MRC therefore corresponds to the minimum total available memory size that leads to the VM's optimal performance and can potentially be used to track the VM's working set size. The following experiments were conducted to determine whether the tails of MRCs are effective tools for estimating accurate working set sizes of both small and large VMs. In these experiments, estimated working set sizes were assessed using the actual working set sizes of the VMs obtained from their performance graphs. Each actual working set size corresponded to the total available memory size at which the performance of the VM started to deteriorate. The MRC of a VM running mixed workloads was also investigated to determine whether the working set size estimated using the tail of the MRC stays consistent when multiple workloads run in the VM.

8.1 Small Virtual Machine

8.1.1 Experimental Setup

The host machine ran VMware ESXi 6.1.0 and was configured with 24GB RAM and 2 quad-core 2.3 GHz AMD Opteron(tm) processors. The VM that ran on this host machine ran the Ubuntu 14.04 operating system and was configured with 4GB vRAM and 4 vCPUs. The sample set size used in MRC construction was 1024. In this case, the stable sampling rate was approximately $2 * 1024 * 1/(4 * 1024) = 0.5$. Two workloads were used to run on this VM. The first workload was the sunflow workload of the SPECjvm2008 benchmark. This workload was used as a memory-intensive workload to ensure that the sampling rate of the VM reached its stable value before the second workload started running. The number of benchmark threads used was 4 and the number of iterations of the workload was 3. The maximum Java heap size, the

minimum Java heap, and the initial Java heap size were set to 4GB, 3GB, and 4GB, respectively. These heap sizes were chosen to ensure that most pages inside the VM's memory space were touched. The second workload was the bwaves workload of the SPEC CPU2006 benchmark. This was the main workload for which the corresponding working set size of the VM was analyzed.

8.1.2 Performance Graph

Many experiments were conducted to plot the performance graph of the bwaves workload. For each experiment, the workload was allowed to run for 1 iteration and its execution time was measured at the end of the run. Table 8 shows the execution time of the bwaves workload for each total available memory size given to the VM. The performance graph of the VM running this workload is plotted in Figure 33. As this performance graph shows, the execution time of the bwaves workload started to increase once the total available memory fell below 1.7GB. This suggests that the actual working set size of this VM was approximately 1.7GB, since this was the minimum total available memory size at which the performance of the VM was at its optimal value. In this case, it was expected that the working set size estimated using the tail of the MRC was at around this value.

Table 8: Different total available memory sizes given to the VM and the corresponding execution times of the bwaves workload.

Total available memory (GB)	Execution time (s)
4.0	1572
2.0	1589
1.8	1599
1.7	1604
1.6	1649
1.5	1824
1.4	2214
1.3	3394

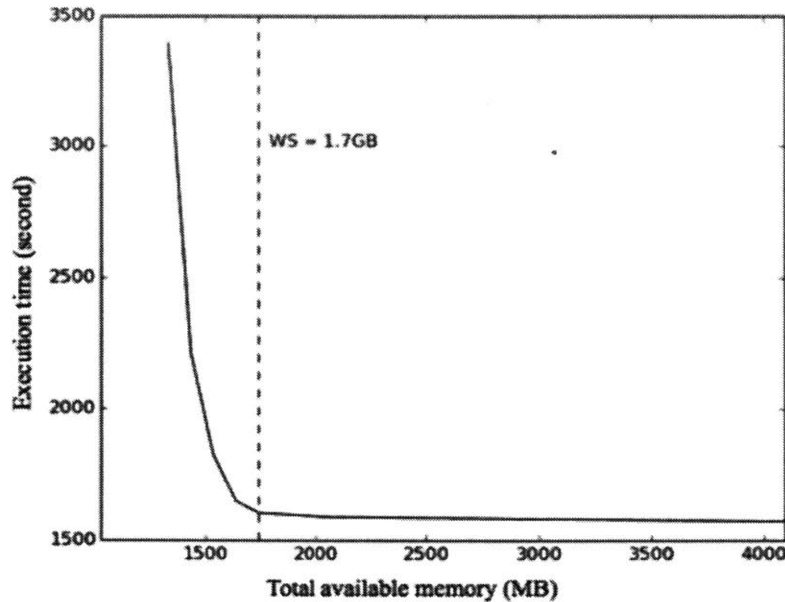


Figure 33: Performance graph of the VM running the bwaves workload. This performance graph shows that the actual working set size of the VM was approximately 1.7GB, since the performance of the VM started to deteriorate once the total available memory size fell below this value.

8.1.3 Working Set Estimated by MRC

Figure 34 shows the evolution of the sampling rate when the VM ran the sunflow workload followed by the bwaves workload. This figure shows that the sampling rate converged to its stable value, since the final sampling rate recorded was 0.484375, only 3% less than its stable value, 0.5. As a result, the requirement that the sampling rate be at its stable value in order for MRC construction to be accurate was fulfilled. A snapshot of the MRC of the VM is shown in Figure 35. This figure shows that the tail of the MRC was approximately 1.7GB. The evolution of the working set size of the small VM estimated using the tail of the MRC when the VM ran the bwaves workload is shown in Figure 36. Notice that the estimated working set size matched the actual working set size obtained from the performance graph quite well. This suggests that MRCs are highly effective tools for estimating the working set sizes of small VMs very accurately.

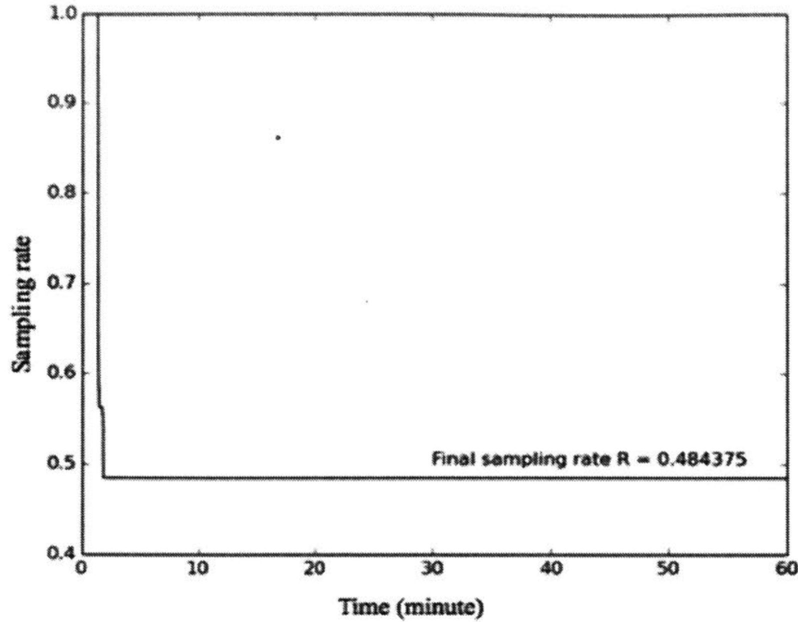


Figure 34: Evolution of the sampling rate when the VM ran the sunflow workload of the SPECjvm2008 benchmark followed by the bwaves workload of the SPEC CPU2006 benchmark. The final sampling rate was close to its stable value 0.5.

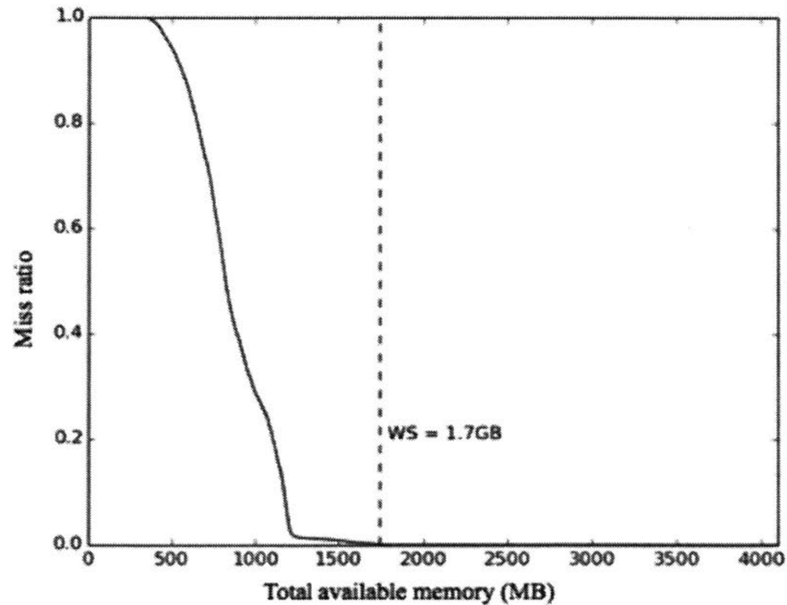


Figure 35: A snapshot of the MRC when the VM ran the bwaves workload of the SPEC CPU2006 benchmark. Using the tail of the MRC, the working set size was estimated to be around 1.70GB. This matched the actual working set size obtained from the performance graph.

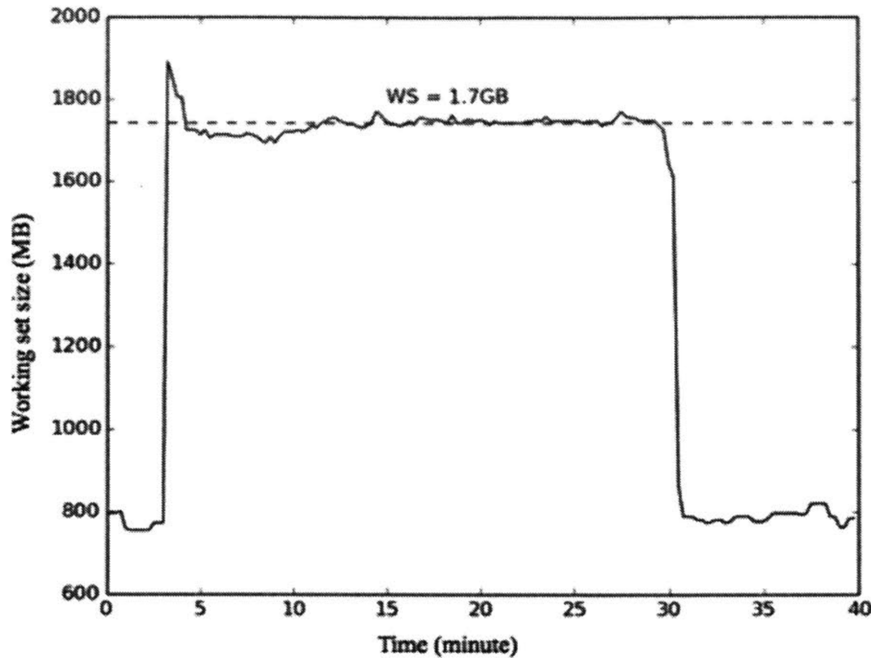


Figure 36: Evolution of the working set size of the small VM estimated using the tail of the MRC when the VM ran the bwaves workload of the SPEC CPU2006 benchmark. This figure shows that the estimated working set size matched the actual working set size obtained from the performance graph very well.

8.2 Large Virtual Machine

8.2.1 Experimental Setup

The host machine ran VMware ESXi 6.1.0 and was configured with 512GB RAM and 8 six-core 2.0 GHz AMD Opteron(tm) processors. The VM that ran on this host machine ran the Ubuntu 14.04 operating system and was configured with 128GB vRAM and 48 vCPUs. The sample set size used in MRC construction was 1024. In this case, the stable sampling rate was approximately 2*1024*1/(128*1024) = 0.015625. Two workloads were chosen to run on these VMs. The first workload was the fixed size sunflow workload of the SPECjvm2008 benchmark. This workload was used as a memory-intensive workload that forced the sampling rate of the VM to converge to its stable value. For this workload, the number of benchmark threads used was 4 and the number of iterations was 10. The maximum Java heap size, the minimum Java heap size, and the initial Java heap size were set to 150GB, 128GB, and 150GB, respectively.

These heap sizes were chosen to be greater than the VM's vRAM to ensure that most pages inside the VM's vRAM were explored by the workload. The second workload was the fixed size serial workload of the SPECjvm2008 benchmark. This was the main workload for which the corresponding working set size of the VM was analyzed. For this workload, the number of benchmark threads used was 4 and the number of iterations was 3. The maximum Java heap size, the minimum Java heap size, and the initial Java heap size were set to 128GB, 116GB, and 128GB, respectively.

8.2.2 Performance Graph

Many experiments were conducted to plot the performance graph of the serial workload. For each experiment, the VM was given a different total available memory size to run the serial workload, and the execution time of the workload was measured at the end of the run. Table 9 shows the execution time of the serial workload for each total available memory size given to the VM. The performance graph of the serial workload is plotted in Figure 37. As this performance graph shows, the execution time of the serial workload started to increase once the total available memory size fell below 92.8GB. Therefore, the actual working set size of the VM was approximately 92.8GB, since this was the minimum total available memory size that led to the optimal performance of the VM.

Table 9: Different total available memory sizes given to the VM and the corresponding execution times of the serial workload.

Total available memory (GB)	Execution time (s)
128	275
95	279
94	275
93	277
92.8	273
92.7	286
92.5	310
92	367
91	569

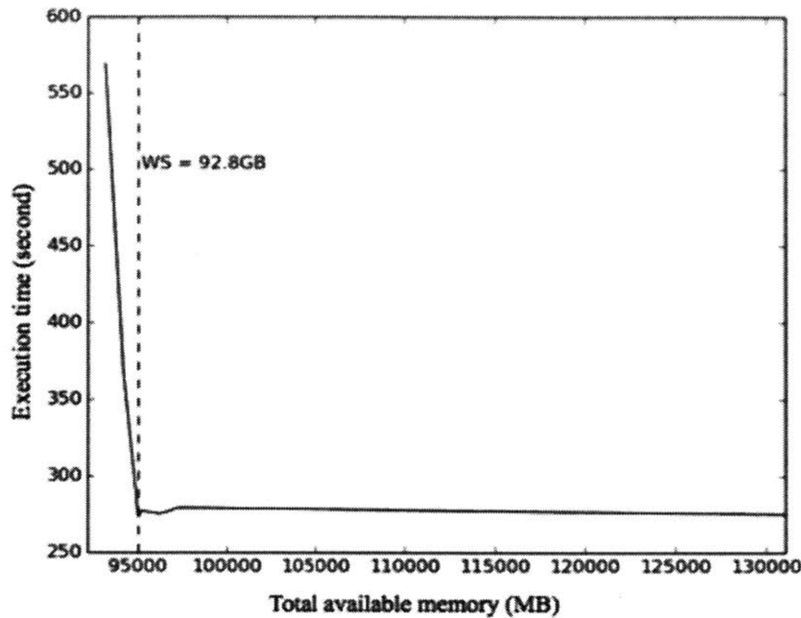


Figure 37: Performance graph of the VM running the serial workload. This graph shows that the actual working set size of the VM was approximately 92.8GB, since the performance of the VM started to deteriorate once the total available memory fell below this value.

8.2.3 Working Set Estimated by MRC

Figure 38 shows the evolution of the sampling rate when the VM ran the sunflow workload followed by the serial workload. This figure shows that the sampling rate converged to its stable value, since the final sampling rate recorded was 0.015625. As a result, the requirement that the sampling rate must be at its stable value in order for MRC construction to be accurate was fulfilled. A snapshot of the MRC of the serial workload is shown in Figure 39. This figure shows that the tail of the MRC was approximately 92.8GB. The evolution of the working set size of the large VM estimated using the tail of the MRC when the VM ran the serial workload is shown in Figure 40. Notice that the working set size estimated using the tail of the MRC matched the actual working set size obtained from the performance graph very well. This suggests that MRCs are effective tools for accurately estimating working set sizes of large VMs.

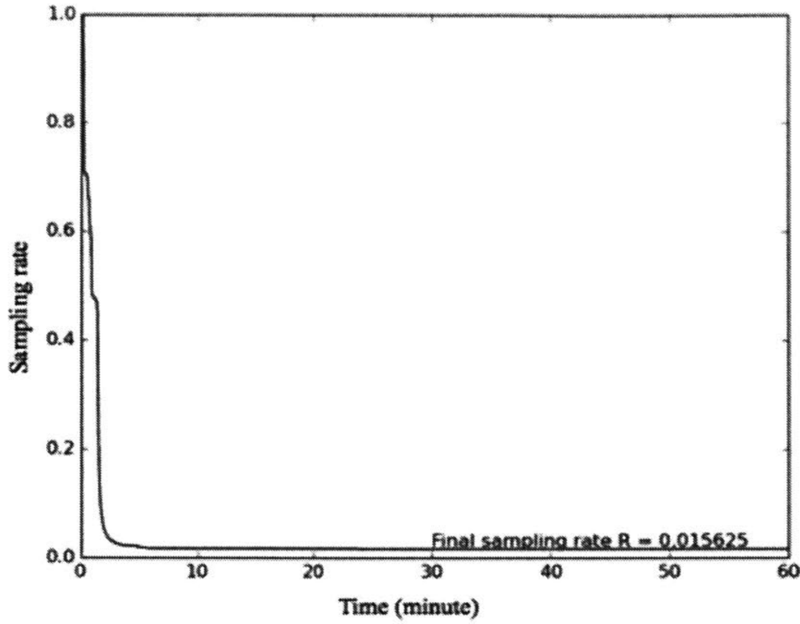


Figure 38: Evolution of the sampling rate when the VM ran the sunflow workload of the SPECjvm2008 benchmark followed by the serial workload of the SPECjvm2008 benchmark. The sampling rate converged to its stable value, 0.015625.

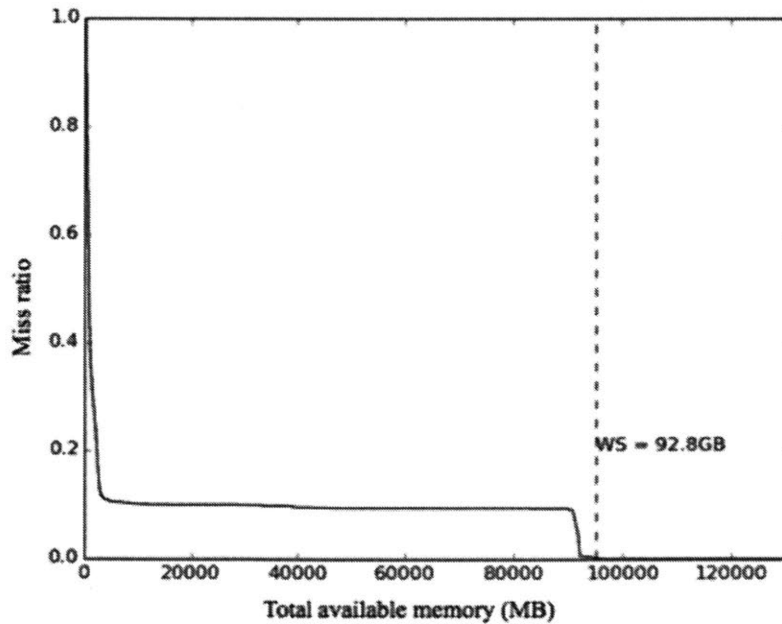


Figure 39: A snapshot of the MRC when the VM ran the serial workload of the SPECjvm2008 benchmark. Using the tail of the MRC, the working set size was estimated to be around 92.8GB. This was also the actual working set size obtained from the performance graph.

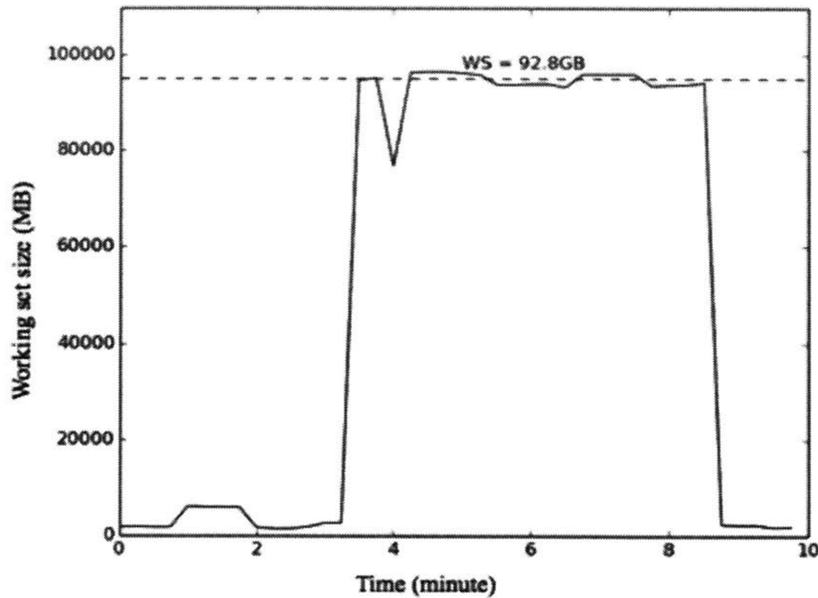


Figure 40: Evolution of the working set size of the large VM estimated using the tail of the MRC when the VM ran the serial workload of the SPECjvm2008 benchmark. Before this workload started running, the sunflow workload of the SPECjvm2008 benchmark was used to force the sampling rate to converge to its stable value. Notice that the estimated working set size matched the actual working set size obtained from the performance graph very well.

8.2.4 Importance of Stable Sampling Rate

To illustrate the importance of having the sampling rate be close to its stable value, the experiment was repeated without running the sunflow workload. Without having a memory-intensive workload running first, the sampling rate is no longer guaranteed to converge to its stable value. Figure 41 plots the evolution of the sampling rate when the large VM ran only the serial workload and Figure 42 plots the evolution of the working set size estimated using the tail of the MRC. As Figure 41 shows, the sampling no longer converged to its stable value. The final sampling rate recorded was 0.021484375, 37.5% greater than the value of the stable sampling rate. Notice that the working set size predicted using the tail of the MRC, as shown in Figure 42, was below the actual working set size. This was because the scale factor, 1/(sampling rate), used to convert each stack distance to a reuse distance was low, since the sampling rate was above its stable value. This suggests that in order to accurately track the working set size of

the VM using the tail of the MRC, it is essential that the sampling rate first be very close to its stable value.

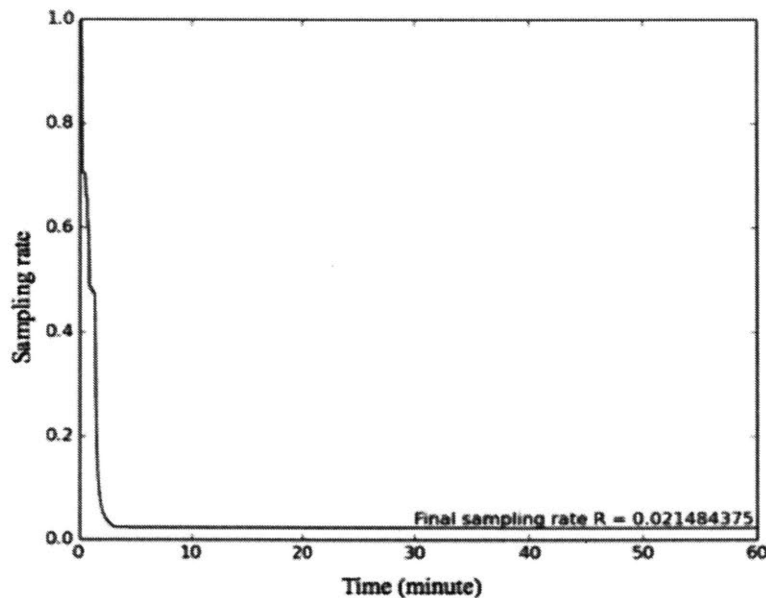


Figure 41: Evolution of the sampling rate when the VM ran only the serial workload of the SPECjvm2008 benchmark. The final sampling rate was 37.5% greater than its stable value, 0.015625.

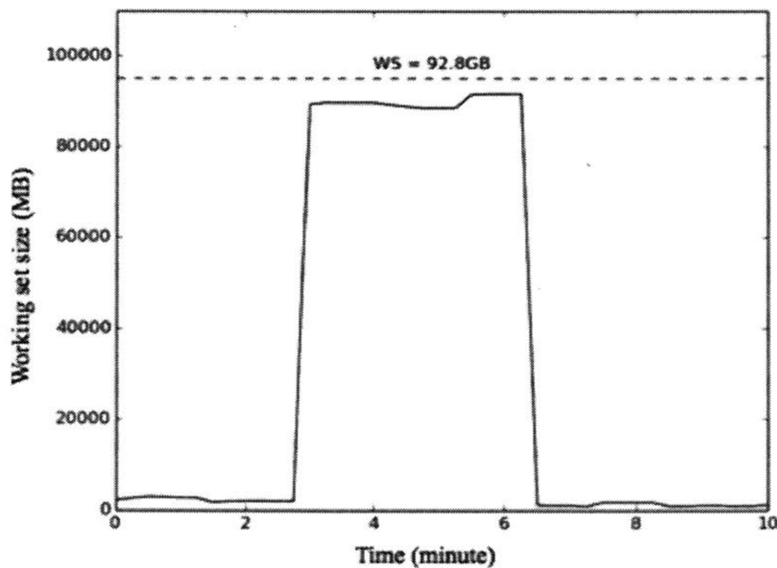


Figure 42: Evolution of the working set size of the large VM estimated using the tail of the MRC when the VM ran the serial workload of the SPEC CPU2006 benchmark only. The sampling rate was not close to its stable value. Notice that the estimated working set size was nowhere near the actual working set size obtained from the performance graph.

8.3 Mixed Workloads

The following experiment was conducted to assess whether the working set size estimated using the tail of the MRC stays consistent when multiple different workloads run in the VM.

8.3.1 Experimental Setup

The host machine ran VMware ESXi 6.1.0 and was configured with 512GB RAM and 8 six-core 2.0 GHz AMD Opteron(tm) processors. The VM that ran on this host machine ran the Ubuntu 14.04 operating system and was configured with 128GB vRAM and 48 vCPUs. The sample set size used in MRC construction was 1024. In this case, the stable sampling rate was approximately 2 * 1024 * 1/(128 * 1024) = 0.015625. Three fixed-size workloads of the SPECjvm2008 benchmark were used to run in the VM. The number of benchmark threads used for all these workloads was 4. The first workload, the sunflow workload, was used as a memory-intensive workload to force the sampling rate to converge to its stable value. For each run of the sunflow workload, the number of iterations was 10. The maximum Java heap size, the minimum Java heap size and the initial Java heap size were set to 150GB, 128GB, and 150GB, respectively. These heap sizes were chosen to be greater than the VM's vRAM to ensure that most pages inside the VM's vRAM were touched. The other two workloads were the previous serial workload and the previous xml.validation workload. These two workloads were chosen because they were configured with different heap sizes, thus ensuring that the working set size of the VM changed when the VM switched from running one workload to the other. For each run of the serial workload, the number of iterations was 3. The maximum Java heap size, the minimum Java heap size and the initial Java heap size were set to 128GB, 116GB, and 128GB, respectively. The actual working set size of the VM that corresponded to this workload was 92.8GB, as shown in Figure 37. For each run of the xml.validation workload, the number of iterations was 5. The maximum Java heap size, the minimum Java heap size and the initial Java heap size were set to 64GB, 52GB, and 64GB, respectively. The actual working set size that corresponded to this workload was 44.2GB, as shown in Figure 28.

8.3.2 Experimental Results

Figure 43 shows the evolution of the sampling rate when the VM ran the sunflow workload, the serial workload and the xml.validation workload.

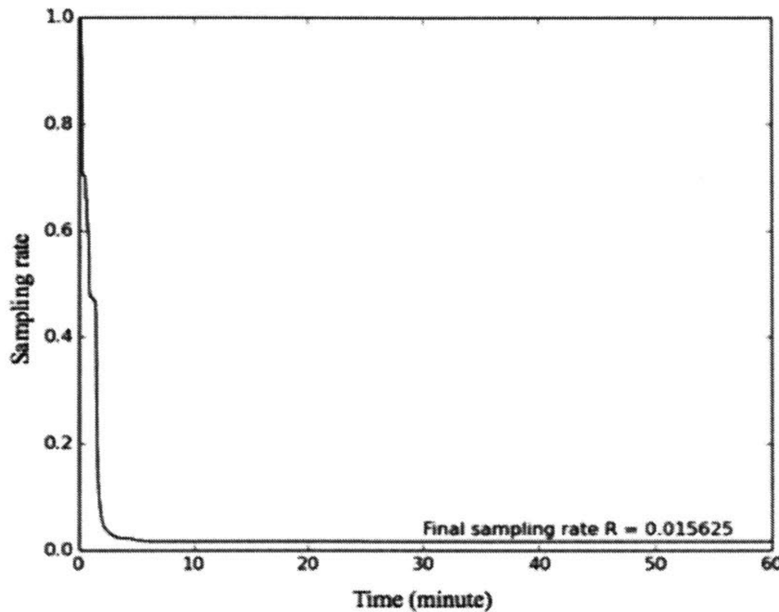


Figure 43: Evolution of the sampling rate when the VM ran the sunflow workload followed by the serial workload and the xml.validation workload of the SPECjvm2008 benchmark. Notice that the sampling rate converged to its stable value, 0.015625.

The sunflow workload was run thrice to force the sampling rate to converge to its stable value before the serial workload and the xml.validation workload started running. As Figure 43 shows, the sampling rate indeed converged to its stable value, 0.015625. This convergence ensures that the scale factor, *1/(sampling rate)*, used to convert each stack distance to a reuse distance, was at its right value, leading to more accurate MRC construction. Once the sunflow workload finished running, the serial workload was run thrice, followed by another 3 runs of the xml.validation workload before the serial workload was run 3 times again. This sequence of workloads was chosen so that one workload was followed by the other, and vice versa. The idea was to determine whether the estimated working set size is compromised if one workload is run after the other. After each run of these workloads, the VM was given 5 minutes to rest before

moving on to run the next workload. The evolution of the working set size estimated using the tail of the MRC is shown in Figure 44. As this figure shows, when the VM ran the serial workload, the estimated working set size was very close to 92.8GB and when the VM switched to running the xml.validation workload, the estimated working set size was very close to 44.2GB. This suggests that the estimated working set size was accurate and consistent and that the MRC is an effective tool for accurately tracking the VM's working set size.

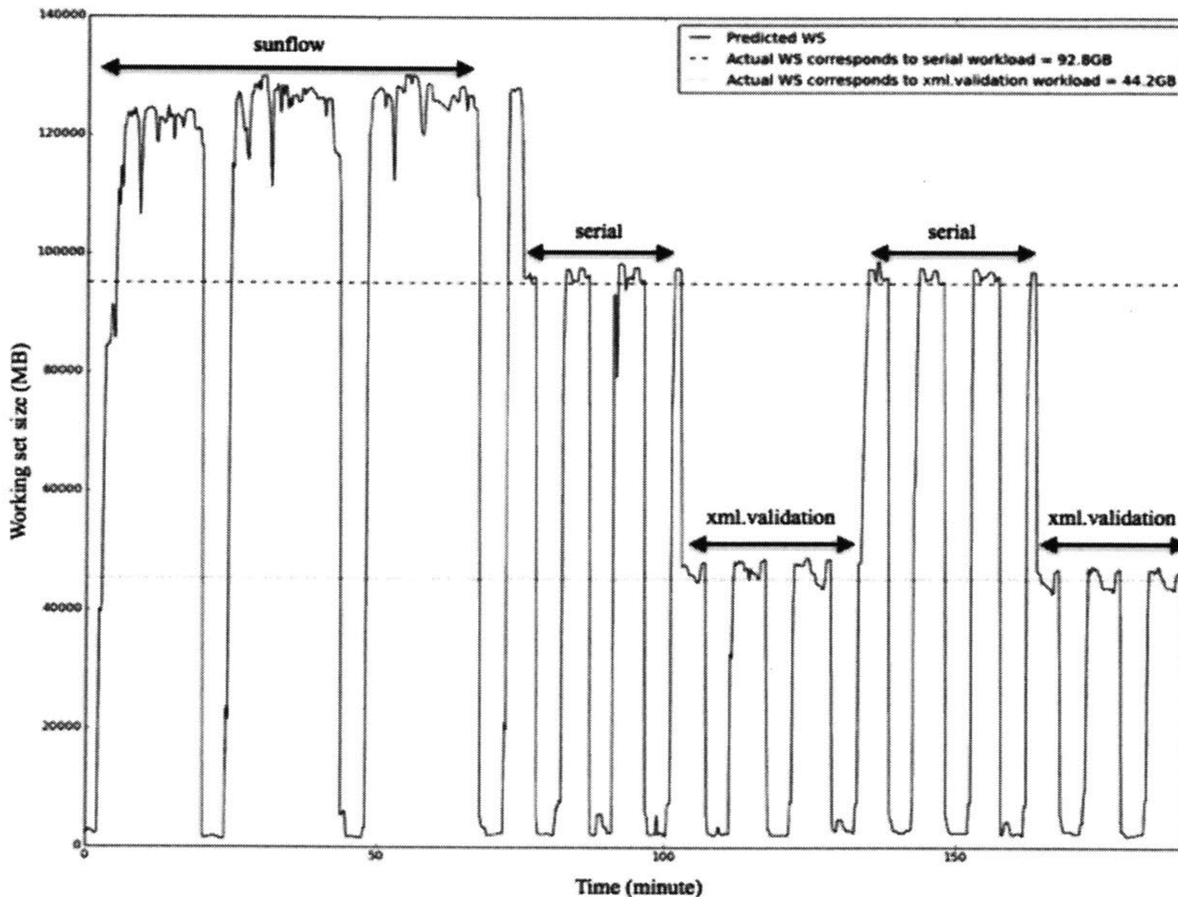


Figure 44: Evolution of the working set size of a large VM estimated using the tail of the MRC when the VM ran both the serial workload and the xml.validation workload of the SPECjvm2008 benchmark. Before these two workloads started running, the sunflow workload of the SPECjvm2008 benchmark was used to force the sampling rate to converge to its stable value. Notice that the estimated working set size matched the actual working set sizes that correspond to the serial workload and the xml.validation workload very well.

9. Modified Version of SHARDS

The original version of the SHARDS technique was designed to construct cache utility curves by scanning through very long block I/O traces collected from data centers over extremely protracted periods, more than 100 weeks [6]. In this case, more than enough time was available for the sampling rate to converge to its stable value, making the original SHARDS technique suitable for constructing accurate cache utility curves. However, this technique is not suitable for constructing an MRC to track a VM's working set size because, in the case of the VM, the time taken for the sampling rate to converge depends on the workload running in the VM and the size of the VM's vRAM. Since the accuracy of MRC construction depends greatly on the sampling rate being near its stable value, it is desirable that the convergence time of the sampling rate be short, so that the MRC can be used to estimate the VM's working set size as soon as the VM starts running. As a result, a modified version of the SHARDS technique is introduced in this thesis to shorten the convergence time of the sampling rate, making this technique more suitable for online tracking of the VM's working set size. In this modified version, all sampled pages are preselected, before the VM boots up, by iterating over addresses of all pages in the entire VM's memory space and selecting the ones that satisfy the sampling condition. For a large VM with large vRAM and many vCPUs, the work done to select sampled pages can be distributed among all vCPUs by having each vCPU be responsible for a portion of the VM's memory space. Preselecting all sampled pages ensures that all pages are explored, thus forcing the sampling rate to converge to its stable value before the VM even starts. As a result, accurate MRC construction can be achieved early without having to rely on running memory-intensive workloads. The following experiments were conducted to illustrate the effectiveness of using the modified version to track a VM's working set size online. The working set sizes estimated using tails of MRCs were compared with the working set sizes estimated by the statistical sampling strategy of VMware ESX to determine which of the two techniques is more effective in tracking the VM's working set size.

9.1 Experimental Setup

The host machine ran VMware ESXi 6.1.0 and was configured with 512GB RAM and 8 six-core 2.0 GHz AMD Opteron(tm) processors. The VM that ran on this host machine ran the Ubuntu 14.04 operating system and was configured with 128GB vRAM and 48 vCPUs. The sample set size used in MRC construction was 2048. In this case, the stable sampling rate was approximately 2*2048*1/(128*1024) = 0.03125. Two previous workloads with different memory demands were chosen to run on this VM. These two workloads were the serial workload and the xml.validation workload of the SPECjvm2008, where the corresponding actual working set sizes of the VM were known. The actual working set size that corresponded to the serial workload was 92.8GB, as shown in Figure 37, and the actual working set size that corresponded to the xml.validation workload was 44.2GB, as shown in Figure 28. Each workload was run multiple times and after each run the VM was allowed to rest for 5 minutes.

9.2 Experimental Results

9.2.1 First Workload (xml.validation)

Figure 45 shows the evolution of the two working set sizes, one estimated using the tail of the MRC (*MRC predicted WS*) and the other one estimated by the statistical sampling strategy of VMware ESX (*ESX instantaneous WS*). Notice that when the VM ran the xml.validation workload, the working set size estimated using the tail of the MRC was much closer to the actual working set size, 44.2GB, than the working set size estimated by the statistical sampling strategy of VMware ESX. Furthermore, the working set size estimated using the tail of the MRC was also more consistent and adapted more quickly to the changing workload.

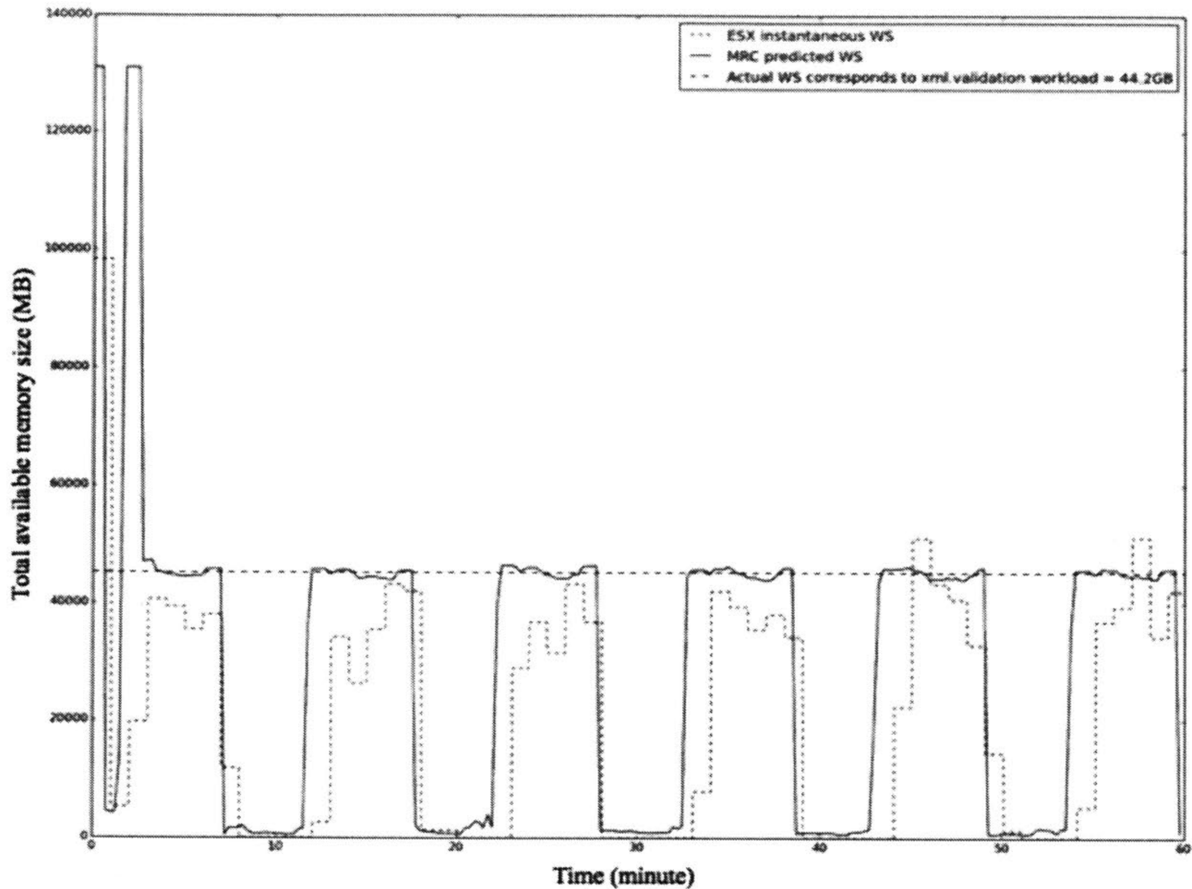


Figure 45: Evolution of the working set size estimated using the tail of the MRC (*MRC predicted WS*) and the working set size estimated by the statistical sampling strategy of ESX (*ESX instantaneous WS*) when the VM ran the xml.validation workload of the SPECjvm2008 benchmark.

9.2.2 Second Workload (serial)

Like Figure 45, Figure 46 shows the evolution of two working set sizes, one estimated using the tail of the MRC (*MRC predicted WS*) and the other one estimated by the statistical sampling strategy of VMware ESX (*ESX instantaneous WS*). Notice that when the VM ran the serial workload, the working set size estimated by the statistical sampling strategy was nowhere near the actual working set size, 92.8GB, obtained from the performance graph. The difference between the two working set sizes was much greater for this workload than for the previous workload, the xml.validation workload. On the other hand, the working set size estimated using the tail of the MRC was much closer to the actual working set size. The estimated working set size was also more consistent and adapted more quickly to the changing workload.

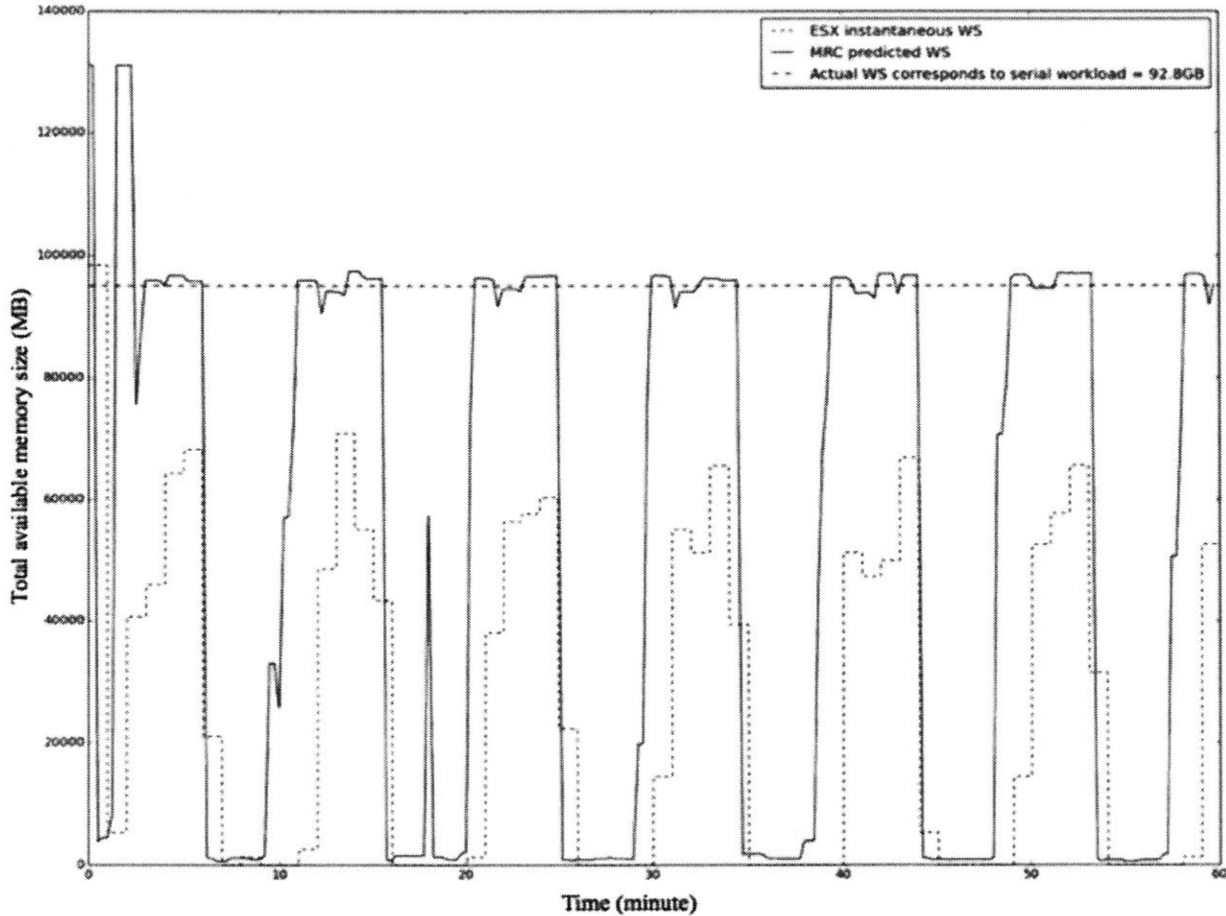


Figure 46: Evolution of the working set size estimated using the tail of the MRC (*MRC predicted WS*) and the working set size estimated by the statistical sampling strategy of VMware ESX (*ESX instantaneous WS*) when the VM ran the serial workload of the SPECjvm2008 benchmark.

The experimental results collected for both workloads, the xml.validation workload and the serial workload, suggest that the modified version of the SHARDS technique can estimate a VM's working set size with much better accuracy than the statistical sampling strategy currently implemented inside VMware ESX. Therefore, MRCs are an effective tool that should be used inside the memory scheduler of VMware ESX to track the working set sizes of VMs.

Bibliography

[1] Banerjee, Ishan et al. "Memory overcommitment in the ESX server." *VMware Technical Journal* 2 (2013).

[2] Waldspurger, Carl A. "Memory resource management in VMware ESX server." *ACM SIGOPS Operating Systems Review* 36.SI (2002): 181-194.

[3] Denning, Peter J. "The working set model for program behavior." *Communications of the ACM* 11.5 (1968): 323-333.

[4] Zhao, Weiming et al. "Efficient LRU-based working set size tracking." *Michigan Technological University Computer Science Technical Report* (2011).

[5] Zhou, Pin et al. "Dynamic tracking of page miss ratio curve for memory management." *ACM SIGOPS Operating Systems Review* 7 Oct. 2004: 177-188.

[6] Waldspurger, Carl A et al. "Efficient MRC construction with SHARDS." *13th USENIX Conference on File and Storage Technologies (FAST 15)* 16 Feb. 2015: 95-110.

[7] Mattson, Richard L. et al. "Evaluation techniques for storage hierarchies." *IBM Systems journal* 9.2 (1970): 78-117.

[8] Denning, Peter J. "The locality principle." *Communications of the ACM* 48.7 (2005): 19-24.

[9] Waldspurger, Carl A. "Efficient MRC Construction with SHARDS." *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA, USA. 17 Feb. 2015. Conference Presentation.