Low Cost Working Set Size Tracking *

Weiming Zhao[1], Xinxin Jin[2], Zhenlin Wang[1], Xiaolin Wang[2], Yingwei Luo[2], and Xiaoming Li[2]

[1]Dept. of Computer Science, Michigan Technological University
[2]Dept. of Computer Science and Technology, Peking University

Abstract

Efficient memory resource management requires knowledge of the memory demands of applications or systems at runtime. A widely proposed approach is to construct an LRU-based miss ratio curve (MRC), which provides not only the current working set size (WSS) but also the relationship between performance and target memory allocation size. Unfortunately, the cost of LRU MRC monitoring is nontrivial. Although optimized with AVL-tree based LRU structure and dynamic hot set sizing, the overhead is still as high as 16% on average. Based on a key insight that for most programs the WSSs are stable most of the time, we design an intermittent tracking scheme, which can temporarily turn off memory tracking when memory demands are predicted to be stable. With the assistance of hardware performance counters, memory tracking can be turned on again if a significant change in memory demands is expected. Experimental results show that, by using this intermittent tracking design, memory tracking can be turned off for 82% of the execution time while the accuracy loss is no more than 4%. More importantly, this design is orthogonal to existing optimizing techniques, such as AVL-tree based LRU structure and dynamic hot set sizing. By combining the three approaches, the mean overhead is lowered to only 2%. We show that when applied to memory balancing for virtual machines, our scheme brings a speedup of 1.85.

1 Introduction

Modeling the relationship between physical memory allocation and performance is indispensable for optimizing memory resource management. As early as the 1970s, working sets were considered as effective tools for modeling memory demands [1]. Because working sets provide a desirable metric for memory management, many solutions have been proposed to track them. One widely proposed approach is to build page-level LRU-based miss ratio curves (MRCs). This approach tracks memory accesses, and constructs a miss ratio curve to correlate memory allocation with page misses. Studies show that this approach can estimate not only the current working set size (WSS) but also the performance impact when the system or application's memory allocation is varied [2, 3, 4, 5].

However, the runtime overhead of maintaining such miss ratio curves is nontrivial. Especially because the complexity and data size of modern applications increase dramatically, the overhead of MRC tracking may overshadow its potential benefits. For example, for `SPEC CPU2006`, using a simple linked-list-based implementation, the overall execution time is increased by a factor of 1.73. Although some previous research optimizes MRC monitoring in terms of data structures [4], the overhead is still considerably high.

This paper introduces a low cost working set size tracking approach. We have implemented an AVL-based LRU structure and dynamic hot set sizing (DHS), which are detailed in our technical report [6], to lower the tracking overhead. However, our experiments show that it is still as high as 16% on average.

By taking advantage of the phase behavior of programs, we further design a novel technique, *intermittent memory tracking* (IMT), to lower the overhead without a significant loss of accuracy. This idea is based on the fact that the execution of a program can be divided into *phases*, within each of which, the memory demands are relatively stable [1]. Thus, when the monitored system or process is predicted to stay in a phase, the memory tracking can be temporarily disabled to avoid tracking cost. Later on, when a phase change is predicted to occur, the memory tracking is resumed to track the working

*Supported by NSF Career CCF0643664, the 973 Program of China No. 2007CB310900, NSFC No. 90718028 and No. 60873052, the 863 Program No.2008AA01Z112, and MOE-Intel Information Technology Foundation under No. MOE-INTEL-10-06. Many thanks to Carl Waldspurger for shepherding this paper.

set size of the new phase.

The key challenge is to predict phase changes when memory tracking is off. Fortunately, we observe that the stability of memory demands is closely correlated with that of some hardware events such as data TLB misses, and L1 and L2 cache misses. This inspired us to utilize these hardware events to predict phase changes when memory tracking is off. However, DTLB and cache-level events show much higher fluctuations (noise) than memory demands, which challenge the accuracy of phase prediction. Worse yet is that the noise level varies by application or even different phases of the same program.

To solve this problem, we design a quick self-adaptive mechanism which can adaptively select a phase-detection threshold. Experimental results show that, during an average of 82% of the time of program execution, memory tracking can be turned off and mean relative error is merely 3.9%.

2 Background and Related Work

2.1 Working Set and Miss Ratio Curve

The active working set of an application refers to the set of pages that it has referenced during the recent working set window. Knowing the working set size (WSS) enables memory resources to be utilized more efficiently. Many approaches have been proposed to estimate the WSS. VMware ESX server adopts a sampling strategy [7]. During a sampling interval, accesses to a set of random pages are monitored. By the end of sampling period, the page utilization of the set is used as an approximation of global memory utilization. This technique can tell how much memory is inactive but it cannot predict the performance impact when memory resources are reclaimed. Geiger [5] detects memory pressure and calculates the amount of extra memory needed by monitoring disk I/O and inferring major page faults. However, when the memory is over-allocated, it is unable to tell how to shrink the memory allocation.

In addition to the current WSS of a system, when memory resource competition occurs, in order to achieve optimal overall performance, we also need to know how performance would be affected by varying the memory allocation size. The miss ratio curve (MRC) that plots the page miss ratio against various amounts of available memory allocation solves the problem. Given an MRC, we can redefine WSS as the size of memory that results in less than a predefined tolerable page miss rate.

A common method to calculate an MRC is the stack algorithm [2]. The stack orders the page numbers based on their recency of accesses. Each stack entry *i* is associated with a counter, denoted as *Hist(i)*. When a reference hits a page, its stack distance, *dist*, is computed, then *Hist(dist)* is incremented by one, and finally this page is moved to the top of the stack. From *Hist*, the page miss ratio with respect to various memory allocation sizes can be computed. Constructing an MRC requires capturing or sampling a sufficient amount of page accesses. Previous research traced MRCs through a permission protection mechanism in the OS or hypervisor [8, 3, 4]. The OS or hypervisor can revoke access permission of pages, so the next accesses to those pages will cause page faults and be captured to build the MRC. For each page interception, the overhead mainly comes from page fault handling and the operation to find the stack distance which is bounded by the WSS. Zhou *et al.* [3] also proposed a hardware-based approach, but it needs extra circuits.

Hypervisor exclusive cache [9] uses an LRU-based MRC to estimate the WSS for each virtual machine. The overhead of MRC construction is analyzed but not quantified in this work. MEB [8] also uses the permission protection mechanism to build the WSS for each VM to support memory balancing. However, the overhead from MRC monitoring is significantly high, especially for applications with poor locality and very large WSSs. For example, `Gems.FDTD` in `SPEC CPU2006` exhibits a 238% overhead. To optimize cache utilization, Zhang *et al.* [10] propose to identify hot pages through scanning the page table of each process using "locality jumping" as an optimization. However, the cost of monitoring a virtualized OS is not evaluated.

2.2 Phase Prediction

Most programs show a typical phasing behavior where the program behavior in terms of IPC, branch prediction, memory access patterns, etc. is stable within a phase while there exists disruptive transition between phases [1, 11].

Shen *et al.* [11] predict locality phases by a combination of offline reuse distance profiling and runtime signal processing. Sherwood *et al.* [12] identify different phases by profiling basic block frequency and using Fourier analysis to filter out noise. However, for online phase detection, the methods that require profiling and sophisticated signal processing techniques are inappropriate. RapidMRC [13] estimates the L2 cache MRC by utilizing the PowerPC-specific Sampled Data Address Register. It selects L2 cache miss rate as the parameter to detect a phase change.

3 Intermittent Memory Tracking

Most programs show typical phasing behavior in terms of memory demands. Within a phase, the WSS remains nearly constant. This inspired us to temporarily disable memory tracking when the monitored program enters a

stable phase and re-enable it when a new phase is encountered. Through this approach, the overhead can be substantially lowered. However, when memory tracking is off, the memory tracking mechanism itself is unable to detect phase transitions anymore. Hence, an alternative method is required to wake up memory tracking when it predicts a phase change.

We find that a phase change of WSS tends be accompanied by sudden changes of the occurrences of memory-related hardware events like TLB misses, L2 misses, etc. And when the WSS remains stable, the activeness of those events is relatively stable too. These events can be monitored by special registers (Performance Monitor Counters, PMCs) built into most modern processors and accessed with negligible overhead. The key challenge is to differentiate phase changes from random fluctuations.

We propose a simple yet effective algorithm to detect behavior changes for both memory demands and performance counters. First, a moving average filter is applied for signal de-noising. Let v_i denote the sampled value (WSS or the number of occurrences of some hardware event) during ith time interval. We pick $f(i) = (v_i + v_{i-1} + \ldots + v_{i-k+1})/k$ as the filtering function to smooth the sampled values, in which k is the filtering parameter, an empirical value. When the moving average filter has not been filled up with k data, memory tracking is always enabled. Once enough data have been sampled, let v_j be the current sampled value and let $f_{mean} = mean(\{f(x)|x \in (j-k, j]\})$, $err_r = f(j)/f_{mean}$ and $err_a = |f(j) - f_{mean}|$. err_r is the relative difference between the current sampled value (smoothed) and the average of history data in the window and err_a is the absolute difference between the two. If $err_r \in [1-\mathbf{T}, 1+\mathbf{T}]$, where $\mathbf{T}$ a small threshold of choice discussed later, we assume the input signal is in a stable phase. Otherwise, we assume that a new phase is encountered. In this case, all the data in the moving average filter is cleared so the data that belong to the previous phase will not be used.

Fixed-Threshold Phase Detection $\mathbf{T}$ is the key parameter in phase detection. We first propose a scheme that uses a fixed $\mathbf{T}$. One phase detector, based on past WSS, checks if the memory demands reach a stable state so the WSS tracking can be turned off. The other detector uses PMC values to check if a new phase is seen so the WSS tracking should be woken up.

For the stability test of WSS, $\mathbf{T}$ can be set to a small value (0.05 in our evaluation) to avoid accuracy loss. In addition, err_a can also be used to guide memory tracking. For example, if memory tracking is at a MB granularity, then as long as $err_a <$ 1MB, WSS can still be assumed in a stable state even when $err_r > \mathbf{T}$.

For phase detection of hardware performance events, an over-strict threshold may cause memory tracking to be enabled unnecessarily and thus undermine performance. On the other hand, if the threshold were too large, WSS changes would not be detected, causing inaccurate tracking results. Our experiments show that, for a given hardware event, the appropriate $\mathbf{T}$ may vary between programs or even vary between phases for the same program. In practice, an empirical value of $\mathbf{T}$ can be used though it may not be the optimal one.

Adaptive-Threshold Phase Detection To improve upon fixed-threshold phase detection, we propose a self-adaptive scheme which adjusts $\mathbf{T}$ dynamically to achieve better performance. The key is to feed the current stability of WSS back to the hardware performance phase detector to construct a closed-loop control system, as illustrated in Figure 1. Initially, the PMC-based phase detector can use the same threshold as used in fixed-threshold phase detection. When memory tracking is on, its current stability is computed and compared with the PMC-based phase detector's decision.

If both of the results are consistent, nothing will be changed. If the current memory demands are stable, while the PMC-based detector makes the opposite decision ($err_r > \mathbf{T}$), it implies that the current threshold is too tight. As a result, its $\mathbf{T}$ is relaxed to its current err_r. Next time, with increased $\mathbf{T}$, the PMC-based detector will most likely find that the system enters a "stable" state and thus turn off memory tracking. On the contrary, if the current memory demands are unstable, while the PMC-based phase detector assumes stable PMC values, i.e. $err_r < \mathbf{T}$, it implies an over-relaxed threshold. Thus, its current $\mathbf{T}$ is lowered to err_r. In short, when the WSS is stable and memory tracking is on, it is only because the PMC-based phase detector is overly sensitive. As a result, $\mathbf{T}$ will be increased until PMC values are considered to be stable too. Then, memory tracking will be turned off.

However, when memory tracking is off, this self-calibration is paused as well, which might miss the chance to tighten the threshold as it should had memory tracking been on. To solve this problem, we introduce a checkpoint design. When memory tracking has been disabled for $ckpt$ consecutive sampling intervals, it is woken up to check if $\mathbf{T}$ should be adjusted or not. If no adjustment is needed, it will be turned off again until it reaches the next checkpoint or meets a new phase. The value of $ckpt$ is adaptive. Initially, it is set to some pre-defined value $ckpt_{init}$. Afterward, if no adjustment is made in the previous checkpoint, it can be increased by some amount ($ckpt_{step}$) until it reaches a maximum value $ckpt_{max}$. Whenever an adjustment is made, $ckpt$ is restored to $ckpt_{init}$. In the ideal case, the ratio of the time that memory tracking is on to the whole execution time, called *up ratio*, is nearly $1/ckpt_{max}$.

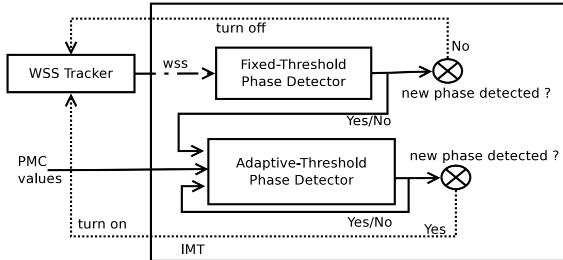


Figure 1: Adaptive-Threshold IMT

4 Implementation and Evaluation

To verify the effectiveness of our WSS tracking and evaluate its application in virtualized environments, we use the Xen 3.4 [14] hypervisor, an open source virtual machine monitor, as the base of our implementation. When a para-virtualized guest OS that runs in user mode attempts to modify its page tables, it has to make a hypercall to explicitly notify the hypervisor to do the actual update. In our modified hypervisor, once such requests are received, our code will first perform the requested update as usual and then revoke the access permission by setting the corresponding bit on the page table entry. For hardware-assisted virtualized machines (HVM), this permission revoking mechanism can be done during the emulation of page table writing or propagation of guest page tables to shadow page tables. Later on, if the guest OS attempts to access that page, it will trigger a minor page fault, which will trap into the hypervisor first. In the modified page fault handling routine, the miss ratio curve is updated for that access and permission is restored.

To verify the effects of our WSS tracking in memory balancing, we use the VM memory balancer that was implemented in [8]. Both IMT and the memory balancer run on Dom-0, a privileged virtual machine. IMT is written in about 300 lines of Python code, with a small C program to initiate hypercalls. Via customized hypercalls, IMT communicates with the WSS tracker to receive current WSS estimation and PMC counter values and send its decisions to the WSS tracker. Based on the assumption that the memory access pattern is nearly unchanged in a stable phase, when the WSS tracker is woken up, it uses the same LRU list and histogram as in the last tracking interval. In our experiments, WSS and PMCs are sampled every 3 seconds. For checkpointing, its initial value ($ckpt_{init}$), the increment, and $ckpt_{max}$ are set to 10, 5, and 20 sampling intervals, respectively, which means the minimum up ratio is nearly 0.05.

All experiments are performed on a server equipped with one 2.8 GHz Intel Core i5 processor (4 cores with HT enabled) and 8 GB of 800 MHz DDR2 memory. Each virtual machine runs 64-bit Linux 2.6.18, configured with 1 virtual CPU and 3 GB of memory (except in

$\mathbf{T} = 0.05$		$\mathbf{T} = 0.2$		$\mathbf{T} = 0.3$		Adaptive	
UR	MRE	UR	MRE	UR	MRE	UR	MRE
.27	.057	.13	.100	.11	.126	.11	.039

Table 1: Mean Up Ratios and MREs of SEPC 2006

the memory balancing test). We select a variety of benchmark suites, including the `SPEC CPU2006` benchmark suite and `DaCapo` [15], a Java benchmark suite, to represent real world workload and evaluate the effectiveness of our work.

In this section, we first evaluate the performance of IMT with various configurations. However, even for the same program, its WSS may not be identical among all runs, which undermines the fairness of comparison. Besides, if IMT is actually used and when it turns off memory tracking, the accuracy loss caused by IMT cannot be precisely measured. As a result, we run IMT against simulated inputs. Then we examine the overhead of WSS tracking with actual runs. Finally, we design a scenario to demonstrate the application of WSS tracking.

4.1 Performance of IMT

The performance of IMT is evaluated by two metrics: (1) the time it saves by turning off memory tracking, reflected by *up ratio*, and (2) the accuracy loss due to temporary inactivation of memory tracking, indicated by *mean relative error*. We first run each benchmarks and sample the WSS and PMC values every 3 seconds without IMT. Then, we feed the trace results to the IMT algorithm to simulate its operations. That is, given inputs $\{M_0, \ldots, M_i\}$ and $\{P_0, \ldots, P_i\}$, the IMT algorithm outputs m_i, in which M_i and P_i are the i-th memory demand and i-th PMC value sampled in the trace results, respectively, and m_i is the estimated memory demand. When the IMT algorithm indicates the activation of memory tracking, $m_i = M_i$, otherwise, $m_i = M_j$ where j is the last time that memory tracking is on. Given a trace with n samples, its mean relative error is computed as $MRE = (\sum_{i=1}^{n} \frac{|M_i - m_i|}{M_i})/n$.

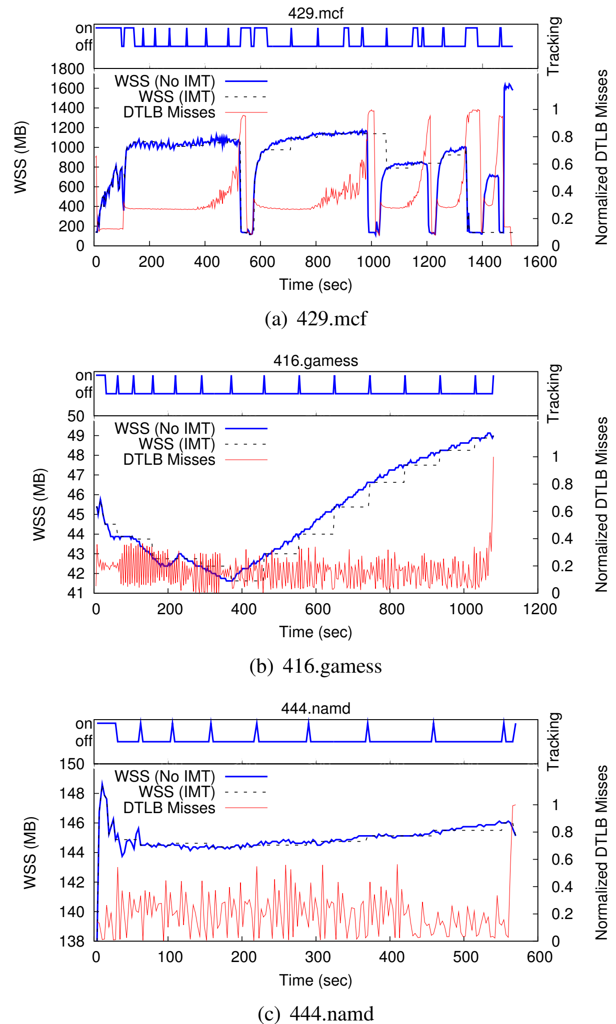
To evaluate the performance of fixed and adaptive thresholds for IMT, we use a DTLB miss as the hardware performance event for phase detection. We have indeed examined three memory related hardware events, DTLB misses, L1 references, and L2 misses as well as their combinations, for phase detection. Interestingly, there is no obvious difference both in accuracy and up ratio [6]. For fixed thresholds, $\mathbf{T}$ varies from 0.05 to 0.3, two extreme ends of the spectrum. Table 1 shows mean up ratios and MREs of `SPEC CPU2006`. The results of individual programs are presented in [6].

Using fixed thresholds, when $\mathbf{T} = 0.05$, memory

tracking is off nearly three fourths of the time with an MRE of about 6%. When **T** is increased to 0.3, memory tracking is activated for only about one tenth of the time, while the MRE increases to 13%. With adaptive thresholds, its up ratio is nearly the same as that of **T** = 0.3, while its MRE is even smaller than that of **T** = 0.05. Clearly, adaptive thresholding outperforms the fixed-threshold algorithm.

Figure 2 shows the results of several cases using adaptive-threshold IMT. The upper parts of each figure show the status of memory tracking: a high level means it is enabled and a low level means it is disabled. In the bottom parts, thick lines and thin lines plot the WSS and normalized data TLB misses from the traces (sampled without IMT), respectively. Dotted lines plot the WSS assuming IMT is enabled. Figure 2(a) shows the common case where there are multiple phases in terms of WSS and DTLB misses. In Figure 2(b) and Figure 2(c), two representative cases, where checkpointing and adaptive thresholding take effect, are presented. For Figure 2(b), when examined from an overall scope, the WSS varies gradually. However, the WSS looks more stable when examined from each small time window. This makes the program assume that the WSS is in the stable mode and thus turns off memory tracking. Nonetheless, with the checkpointing mechanism, the WSS variances are still captured. Figure 2(c) shows that, though the WSS is stable most of the time, the DTLB miss fluctuates randomly. With the adaptive algorithm, the noise is filtered by increased thresholds.

With adaptive-threshold IMT, `429.mcf` shows an MRE of 38.7% while all others are less than 8% with a mean of 2%. For `429.mcf`, as Figure 2(a) shows, most of the time, the WSS estimation using IMT follows the one without using IMT. The high relative error is because its WSS changes dramatically up to 9 times at the borders of phase transitions. Though after a short delay, IMT detects the phase change and wakes up memory tracking, those exceptionally high relative errors lead to a large MRE. More specifically, during 67% of its execution time, the relative errors are below 4%, and during 84% of the time, the relative errors remain within 10%.



(a) 429.mcf

(b) 416.gamess

(c) 444.namd

Figure 2: Examples of Using Adaptive-Threshold IMT

4.2 Overhead Evaluation

To evaluate the actual effects of using IMT, we measure the WSS tracking overhead on actual runs. As Table 2 shows[1], even optimized with AVL-based LRU and dynamic hot set sizing, the mean overhead of `SPEC CPU2006` is 16% due to large WSSs and/or bad locality of some programs. For example, for high-overhead programs, such as `429.mcf` and `433.milc`, the average WSSs are 859 MB and 334 MB, respectively, while the average WSSs of `401.bzip2` and `416.gamess` are only 24 MB and 45 MB, respectively.

Enhanced with fixed-threshold IMT (**T** = 0.2), the mean overhead is lowered to 6%. Using adaptive-threshold IMT, the mean overhead is further reduced to 2% by cutting off half of the up time of memory tracking.

4.3 Applications to VM Memory Balancing

One typical scenario for WSS tracking is memory balancing. Two VMs are monitored on a Xen-based host. One VM runs `470.lbm`, meanwhile, the other VM runs `433.milc`. Initially, each VM is allocated 700 MB of memory. In the baseline setting, no memory balancing or WSS tracking is used. With memory balancing, three variations are compared: memory tracking without IMT, using IMT with a fixed threshold of 0.2 and using IMT with an adaptive threshold. Figure 3 shows the normalized speedups with memory balancing against the baseline setting. Note that, the balancer is designed to reclaim

[1]The complete list is in [6].

Program	Norm. Exec. Time				Up Ratio	
	L	A+D	A+D + I_f	A+D + I_a	I_f	I_a
401.bzip2	1.03	1.02	1.01	1.01	0.76	0.14
416.gamess	1.01	1.01	1.00	1.00	0.18	0.09
429.mcf	59.16	1.75	1.41	1.04	0.72	0.37
433.milc	13.08	3.83	2.46	1.05	0.52	0.11
470.lbm	4.31	1.77	1.01	1.00	0.17	0.10
...		...			...	
Mean	2.73	1.16	1.06	1.02	0.26	0.12

Table 2: Normalized Execution Time and Up Ratios
L: linked list, A+D: ABL and dynamic hot set, I_f: fixed-threshold IMT ($\mathbf{T} = 0.2$), I_a: adaptive-threshold IMT

unused memory, so the total allocated memory to the two VMs may be less than 1400 MB.

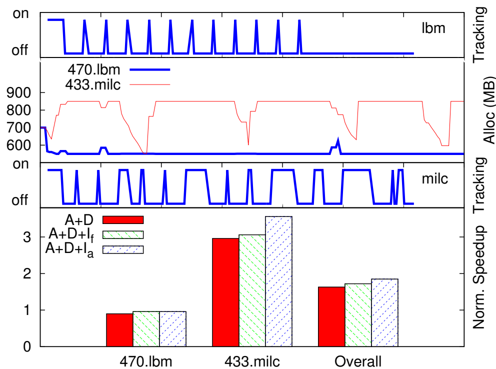


Figure 3: Speed-Ups With Memory Balancing

When balanced without IMT, the performance of `470.lbm` degrades by 10% due to the overhead of memory tracking, while the performance of `433.milc` is boosted by 2 times due to the extra memory it gets from the other VM. Using IMT, the performance impact of memory tracking on `470.lbm` is lowered to 4%. For `433.milc`, with fixed-threshold or adaptive-threshold IMT, its speedup is increased from 2.96 to 3.06 and 3.56 respectively. The overall speedups of balancing without IMT, with fixed-threshold IMT and adaptive-threshold IMT are 1.63, 1.72 and 1.85. Hence, using adaptive-threshold IMT, an additional 22% speedup is gained.

5 Conclusion and Future Work

LRU-based working set size estimation is an effective technique to support memory resource management. This paper makes this technique more applicable by significantly reducing its overhead. We present a novel intermittent memory tracking scheme. Experimental evaluation shows that our solution is capable of reducing the overhead with sufficient precision to improve memory allocation decisions. In an application scenario of balancing memory resources for virtual machines, our solution boosts the overall performance. In the future, we plan to develop theoretical models that verify the correlations among various memory events.

References

[1] P. J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1), 1980.

[2] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[3] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS'04*, pages 177–188, 2004.

[4] T. Yang, E. D. Berger, S. F. Kaplan, and J. Eliot B. Moss. CRAMM: virtual memory support for garbage-collected applications. In *OSDI'06*, pages 103–116, 2006.

[5] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. *SIGOPS Oper. Syst. Rev.*, 40(5):14–24, 2006.

[6] W. Zhao, X. Jin, Z. Wang, X. Wang, Y. Luo, and X. Li. Efficient LRU-based working set size tracking. Technical Report CS-TR-11-01, Department of Computer Science, Michigan Tech University, 2011.

[7] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[8] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. In *VEE'09*, 2009.

[9] P. Lu and K. Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *USENIX ATC'07*, pages 1–15, 2007.

[10] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multi-core cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, 2009.

[11] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS'04*, 2004.

[12] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT'01*, pages 3 –14, 2001.

[13] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating l2 miss rate curves on commodity systems for online optimizations. In *ASPLOS'09*, pages 121–132, 2009.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, 2003.

[15] S. M. Blackburn, R. Garner, and C. Hoffman et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06*, pages 169–190, 2006.