Efficient LRU-Based Working Set Size Tracking

by Weiming Zhao, Xinxin Jin, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, Xiaoming Li

Michigan Technological University
Computer Science Technical Report
CS-TR-11-01
March 28, 2011

Efficient LRU-Based Working Set Size Tracking

Weiming Zhao, Xinxin Jin, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, Xiaoming Li

March 28, 2011

Abstract

Efficient memory resource management requires the 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 previous techniques adopt some optimizations, they still incur large overheads. To attack this problem, this paper proposes a low cost working set size tracking method, which seeks to generate an accurate MRC of programs with negligible overhead. We first introduce two enhancements over current techniques: AVL-tree based LRU structure and dynamic hot set sizing. Nonetheless, the overhead is still as high as 16% on average. Based on a key insight that for most programs the working set sizes 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 execution time while the accuracy loss is no more than 4%. More importantly, this design is orthogonal to existing optimizing techniques. By combining the three enhancements, the mean overhead is lowered to only 2%. We show that when applied it to memory balancing for virtual machines, our scheme brings a speedup of 2.12.

1 Introduction

Modeling the relationship between physical memory allocation and performance is indispensable for optimizing memory resource management. As early as 1970s, working sets were considered as effective tools for measuring and modeling memory demands [1]. Further, in the studies of phase behavior in real programs, people gradually realized that disruptive transitions between phases play an important role in describing the programs' locality. Because working set provides a desirable metric for memory management, many solutions have been proposed to track them. One widely proposed method is to build page-level LRU-based miss ratio curves (MRCs). This approach tracks memory accesses and emulates the LRU replacement policy by constructing an LRU stack and a histogram that records the number of hits to each stack location. Based on the LRU histogram, a miss ratio curve can be derived to correlate memory allocation with page misses and system performance. Studies show that this approach can estimate not only the current working set size (WSS) but also the performance gain or loss when the system or application's memory allocation increases or decreases [2, 3, 4, 5, 6, 7]. 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 [5], the overhead is still considerably high.

This paper introduces a low cost working set size tracking approach. We first add two refinements on existing optimizations: AVL-Tree based LRU organization (ABL) and dynamic hot set sizing (DHS). Compared with the canonical linked-list-based LRU implementation, the AVL-tree based algorithm lowers the asymptotic cost of LRU operations from $O(N)$ to $O(\log N)$. In addition, we utilize hardware performance monitoring counters (PMCs) to improve the accuracy of WSS estimation when *dynamic hot set sizing* [5, 8] is used. Dynamic hot set sizing helps to lower overhead by controlling the number of memory access interceptions. It takes program locality into account and adjusts the size of hot set accordingly. However, in some cases, for some programs with very small working set size, it can substantially overestimate the working set size. Using hardware performance monitors, our design is able to detect such pathological case and make appropriate adjustments. Although ABL and DHS reduce the overhead significantly, our experiments show that the overhead is still as high as 16% on average.

To further lower the overhead without a significant loss of accuracy, we design a novel technique, *intermittent memory tracking* (IMT). 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 set size of the new phase.

The key challenge is the phase change prediction. Since we turn off memory tracking within a phase, we of course cannot rely on the predicted working set sizes to derive phase changes. 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. Inspired by this fact, we design a phase detector by monitoring those PMCs. A phase change on those counters suggests a likely phase change on memory demands. However, DTLB and cache level events show much higher fluctuations (noise) than memory demands, which challenge the accuracy of phase prediction. An over-sensitive phase detector may give false positive results, which leads to unnecessary activation of memory tracking. On the contrary, an over-tolerant design may miss the actual phase changes and thus result in loss of accuracy. And the noise level varies by programs or even different phases of the same program. In addition, unlike off-line phase analysis, which can adopt complicated signal processing techniques, on-line analysis requires that the algorithm be fast and simple.

To solve this problem, we design a quick self-adaptive phase detector, which adaptively selects a phase-detection threshold based on the past predicted memory demand trend. 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%.

Our main contribution is to present a low-overhead software-only miss ratio curve tracking method. We examine and refine the existing techniques of MRC implementation. Our three approaches, AVL-based LRU data structure, dynamic hot set sizing, and intermittent memory tracking, are cooperative and complementary and the combination of all the three leads to an efficient WSS estimator. Figure 1 illustrates the system organization.

To evaluate the performance of our tracking mechanism, we implement it in a virtual machine monitor (VMM), which delivers a great potential for the VMM to make a wise memory management decision. Because most memory accesses of virtual machines bypass the VMM, there is no straightforward information for a VMM to estimate the working set size of a guest OS. We show that our mechanism implemented in the VMM is transparent to the guest OS and can work together with the existing memory management techniques for virtual machines. We then apply the WSS tracking to memory balancing for multiple virtual machines by predicting the memory demands of each virtual machine. Our previous work [8] has implemented a hypervisor memory balancer, but it incurred considerable overhead, especially for applications with large WSSs. The optimized MRC tracking method helps the balancer achieve further performance improvement.

The remainder of this paper is organized as follows. We provide some background information and discuss related work in Section 2. Section 3 presents the details of the optimizations on WSS monitoring. Sections 4 and 5 discuss the system implementation and experimental results and give an application scenario. Section 6 concludes the paper.

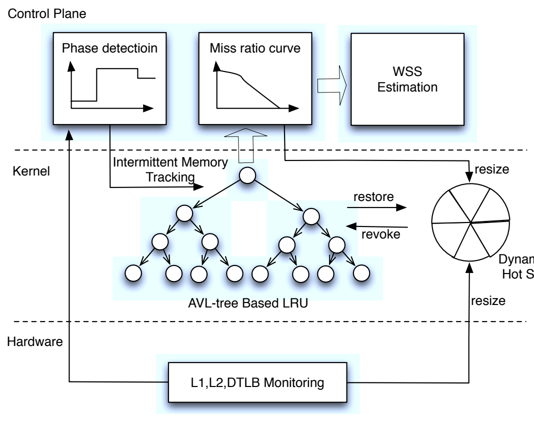


Figure 1: System Overview

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. The application's working set size is the size of memory footprint of its working set. Typically, if the WSS is larger than the available memory in the system, the memory manager usually swaps some pages to disk, which significantly slows down the program. As a result, the significance of working set size estimation is that it provides a necessary metric that ensures the system to reach the maximum performance as expected.

Many approaches have been proposed to estimate the active working set size. VMware ESX server adopts a sampling strategy [9]. 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 because the performance of applications is not usually proportional to the size of allocated memory. Geiger [6] 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.

Under many circumstances, such as multi-workload scheduling, resource allocation and power saving, knowing the current working set size alone is insufficient. For example, when two applications compete for memory resources, in order to achieve optimal overall performance, the arbitration scheme needs to know how the performance would be impacted by varying their allocation sizes. The miss ratio curve plots the page miss ratio against varying amounts of available memory and provides a better correlation between memory allocation and system performance. With an MRC, we can redefine WSS as the size of memory that results in less than a predefined tolerable page miss rate. Figure 2 shows an example of MRC with varying memory from 0 to 10 pages. We notice that even if we reduce the memory size by a half, the corresponding page miss ratio is only 10%, so we may consider the corresponding WSS as five pages.

A common method to calculate an MRC is the stack algorithm [2]. It uses an LRU stack to store the page numbers of accessed page frames. Each stack entry i is associated with a counter, denoted as *Hist(i)*. One can plot an *LRU histogram* relating each counter value to its corresponding LRU location, i.e., *stack distance*. When a reference hits a page, the algorithm first computes the stack distance, *dist*, increments *Hist(dist)* by one, and then moves the page to the top. If there is a stack with depth D and we reduce it to

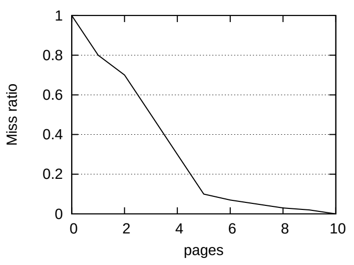


Figure 2: Page Miss Ratio Curve Example

depth d, then the expected miss ratio can be calculated as follows:

$$Miss\_ratio(d) = \frac{\sum_{i>d}^{D} Hist(i)}{\sum_{i=0}^{D} Hist(i)}.$$

Constructing the LRU histogram and MRC requires capturing or sampling a sufficient amount of page accesses. Previous research traced MRC through a permission protection mechanism in the OS or hypervisor [8, 4]. The OS or hypervisor can revoke access permission of pages, so the next accesses to those pages will cause minor page faults. The page fault handling mechanism is extended to analyze the faults and build the MRC. However, handling minor page faults and updating the LRU histogram are time consuming, the key challenge is to minimize the overhead which we attempt to address in this paper.

Zhou *et al.* [4] propose both hardware-based and software-based approaches to trace MRC. The hardware approach needs extra circuits. In the software solution, for each process, it uses a doubly-linked list to maintain the stack order. To reduce the cost of maintaining the LRU list, it further adopts a "scan list" to track those frequently accessed pages by scanning their page table entries. We later on extend the scan list approach and call it *hot set* [8].

CRAMM [5] is an MRC tracking based virtual memory manager, which builds a model that correlates the WSS of a Java application and its heap size. It improves the MRC implementation using dynamic AVL-based page lists. Desirable as the improvement is, there still exist limitations: (1) On the extreme condition, the searching operation still yields $O(N)$ time but not $O(logN)$; (2) The dynamic list size is adjusted according to system overhead, which may sacrifice the accuracy.

Hypervisor exclusive cache [10] uses an LRU-based MRC to estimate the WSS for each virtual machine. In this design, each VM gets a small amount of machine memory, called direct memory, and the rest of the memory is managed by the VMM in the form of exclusive cache. Because it can track all memory accesses above the minimum memory allocation, both WSS increases and decreases can be deduced, although the maximum increment is bounded by the cache size. The overhead of MRC construction is analyzed but not quantified in this work.

MEB [8] also uses the permission protection mechanism to build the MRC for each VM to support memory balancing. Although this method reaches a desirable accuracy, 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.* [11] propose to identify hot pages through scanning the page table of each process. Using "locality jumping", the scanning cost per application is greatly lowered. 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, 12].

Sherwood *et al.* [13] adopt a runtime hardware based phase detection and classification architecture to capture the periodic behavior of programs. Through the changes of program running characteristics such as IPC and energy, they divide instructions into a set of buckets based on branch PCs.

Shen *et al.* [12] predict locality phases by a combination of offline profiling and runtime prediction. Their scheme first identifies locality phases using reuse distance profiling. When the instrumented program runs, it uses the first few executions of a phase to predict all its later executions.

RapidMRC [14] estimates L2 cache MRC by collecting memory access traces using the PMU available in PowerPC. But for most of the modern processors, the PMU cannot provide complete samples of memory accesses. RapidMRC selects L2 cache miss rate as the parameter to detect a phase change.

3 Low Cost Working Set Size Tracking

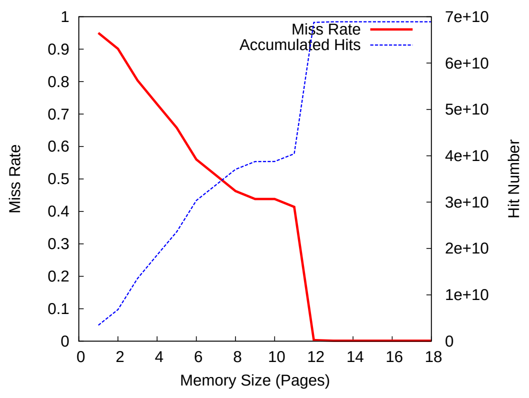
In this section, we first introduce two enhancements over existing optimizations. Then, we present our intermittent memory tracking technique.

3.1 Dynamic Hot Set Size (DHS)

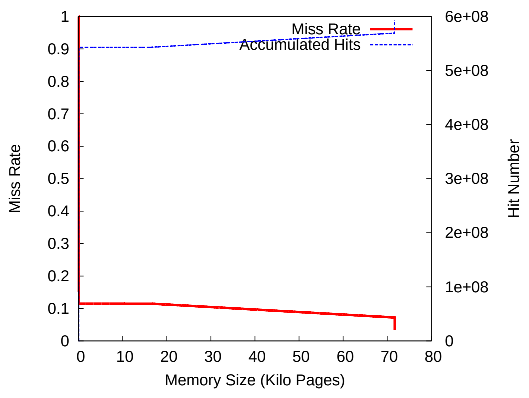
The LRU monitoring overhead is proportional to the number of intercepted memory accesses. It is prohibitive to intercept every memory page access. Since most programs show good locality, in our previous work [8], we use a small hot set to reduce the number of interceptions. A page access is intercepted only when it is not in the hot set, or in other words, in the cold set. The size of the hot set can significantly affect the number of memory access interceptions as well as estimation accuracy. Though the concept of hot/cold pages is similar to the active/inactive page lists in [5], our design decouples the page-level hot/cold control from the LRU list management. The LRU list and histogram can work on a coarser tracking granularity to save space and time while the fine-grained hot/cold page control provides more precision. Even so, a program with poor locality can exhibit large overhead due to a great number of interceptions. To control the overhead, we suggested a dynamic hot set sizing scheme in our previous work [8]. The idea is to dynamically increase the hot set size when the locality is poor. The locality is evaluated by examining the miss ratio curve. Let $m(x)$ be the function of the miss ratio curve, where x is miss ratio and $m(x)$ is the corresponding memory size x. We use $\alpha = m(50\%)/\frac{WSS}{2}$ to quantify the locality. The smaller the value of α is, the better the locality is. When α is greater than a preset threshold, we incrementally increase the size of hot set until α is lower than the threshold or the hot set reaches a preset upper bound.

A general knowledge is that the hot set size is far less than the WSS and thus we can rely on the LRU list itself to estimate the WSS. However, when the hot set size is too big and too many memory accesses fall into the hot set, the LRU histogram will become more flat, which can cause an overestimation of memory requirement.

One example is `171.swim`. We use a precise instrumentation tool, PIN [15], to record every memory access of this program and then simulate the process of our LRU monitoring with different hot set sizes. Figure 3 illustrates the miss rates (in solid lines with respect to the left axes) and accumulated hits (in dashed lines with respect to the right axes) for two hot set sizes, respectively. As Figure 3(a) shows, when the hot set size is 0, 12 pages of memory cover more than 95% memory accesses. However, when the hot set size is 12, as shown in Figure 3(b), we can observe that the miss rates are above 5% until memory size reaches around 71000 pages, which suggests a memory requirement of 284 MB. And by comparing the accumulated hits, we can see that, if the hot set size is 12, the tracking system only intercepts about 0.1% memory accesses. Another case is `436.cactusADM`, as the simulation results show in Figure 4(a), its WSS is only 58 pages when hot set size is no more than 32 pages. When the hot set size exceeds 58, its WSS estimation soars to 408 MB. To avoid such pathological cases caused by an oversized hot set, we need to estimate how many accesses go to the hot set. If a majority of accesses are in the hot set, the WSS is the hot set size. To estimate hot set accesses, we resort to the number of data TLB misses that can be monitored using hardware performance counters. As the number of entries of a data TLB is relatively small, its miss number can be seen as an approximation of the number of memory page accesses. We observe that, normally,

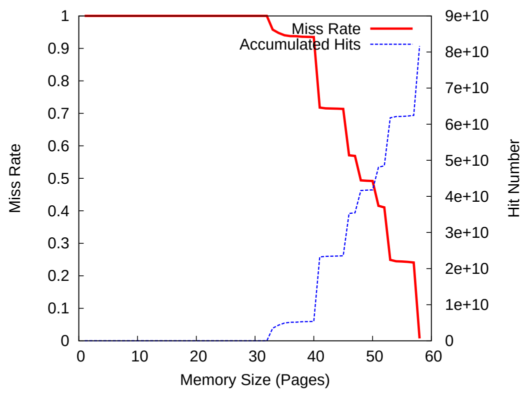


(a) Hot Set Size Is 0

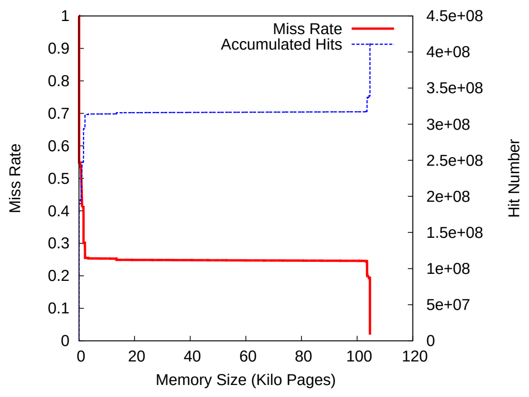


(b) Hot Set Size Is 12

Figure 3: Miss Ratio Curves of `171.swim`



(a) Hot Set Size Is 32



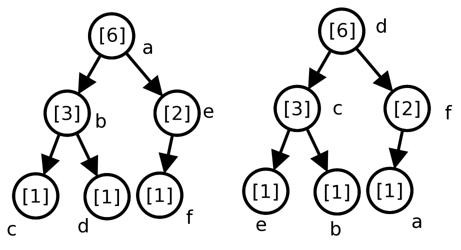
(b) Hot Set Size Is 64

Figure 4: Miss Ratio Curves of `436.cactusADM`

the number of the TLB misses, T_m, is no more than two orders of magnitude larger than the number of intercepted memory accesses, I. If $T_m >> I$, it implies that there are too many memory accesses that are not intercepted by the system. As long as such situation occurs, we can either gradually lower the hot set size or take the hot set size as its WSS.

3.2 AVL-Tree Based LRU Structure

In order to maintain an LRU ordering of page level memory accesses, a common design is to maintain a linked list of tags, each of which represents a (group of) page(s). Once an access to some page is trapped, its corresponding tag is moved from its current location to the head (the MRU location) of the linked list. The positional order of each tag, or its LRU distance, is required to update the LRU histogram counters. The linked list design enables $O(1)$ tag moving. However, the cost of locating a tag's position in the list requires linear search with a cost of $O(N)$ where N is the total number of tags. As a result, the overall time cost of LRU updating is in $O(N)$. For a program with good locality, a target tag can usually be found near the beginning of the linked list. Unfortunately, for a program with poor locality and large memory footprint,



(a) Before (b) After

Figure 5: An Example of an AVL-Based LRU List
Before page e is visited, the LRU sequence is c, b, d, a, f, e. Then, e is moved to the left-most position, and then the tree is re-balanced. In-order traversal gives e, c, b, d, a, f.

the linear search cost can be prohibitive. We observe an overhead up to 238% in one benchmark.

To attack this problem, we use an $O(\log(N))$-time AVL-tree based logical organization for the tags. For any tag, its left child has shorter LRU distance and its right child has longer LRU distance. That is, an in-order traversal of the tree gives the same sequence as given by a linked list traversal from the head. Figure 5 gives an example of a tree-based LRU list.

To facilitate locating the LRU position and the distance of a tag, each tag has a field, *size*, which counts the size of the sub-tree rooted from itself. In Figure 5, the numbers in the square brackets are the values of the *size* fields of the nodes. For any tag x, its LRU distance (LD) is calculated recursively by:

$$LD(x) = \begin{cases} 0 & x \text{ is } nil \\ LD(ANC(x)) + size(LC(x)) + 1 & x \text{ is not } nil \end{cases},$$

in which functions $size(x)$ and $LC(x)$ denote the size of the sub-tree rooted as x and x's left child, respectively. Function $ANC(x)$ returns either *nil* or the nearest ancestor y of x such that y's left child is neither x nor x's ancestor. That is, $LC(y) \notin \{x \text{ and } x\text{'s ancestors}\}$. Node y is indeed the immediate previous node of the leftmost child of x in in-order traversal. If such y cannot be found, $ANC(x)$ is *nil*. For example, in Figure 5(a), $ANC(c)$ is *nil* and $ANC(f)$ is a. Since function ANC walks up along the tree towards the root, and the tree is balanced, the time cost of LD is $O(\log(N))$. When an access to some physical page is intercepted, its corresponding tag's LRU distance is first computed and then the tag is removed and inserted as the leftmost leaf. During the insertion or removal, at most all the tag's ancestors' *size* fields need to be updated, which takes $O(\log N)$ time, the same cost upper bound for tree balancing. As a result, the overall time cost is lowered to $O(\log(N))$, while the space cost is still $O(N)$.

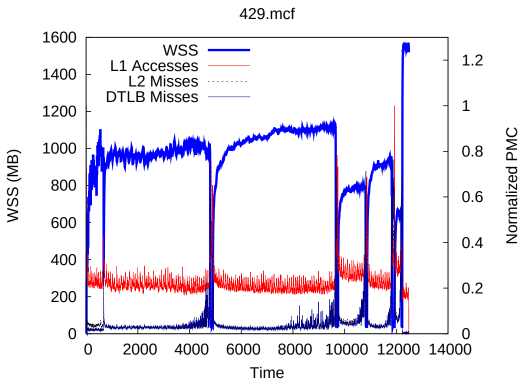
3.3 Intermittent Memory Tracking (IMT)

Most programs show typical phasing behavior in terms of memory demands. Within a phase, the working set size 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 phase detection method is required to wake up memory tracking when it predicts a phase change.

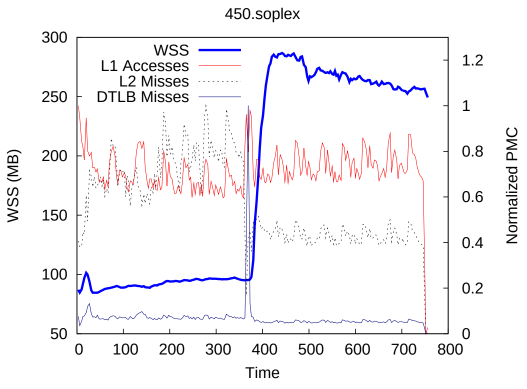
We find that a sudden change of working set size tends be accompanied by sudden changes of the occurrences of memory-related hardware events like TLB misses, L2 misses, etc. And when the working set size remains stable, the number of occurrences of those events is relatively stable too. These events can be monitored by special registers built into most modern processors and accessed with negligible overhead. The key challenge is to differentiate phase changes from random fluctuations. The remainder of this section will focus on our solutions to this problem.

3.3.1 Selection of Events

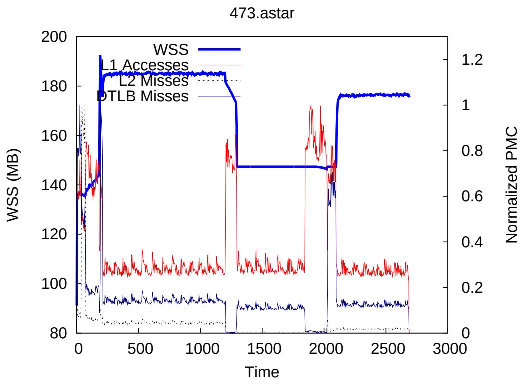
There exist numerous memory-related hardware events in modern processors, such as L1/L2 accesses/misses, TLB accesses/misses and so on. As a change of working set size implies changes in memory access patterns at the page level, it suggests that data TLB (DTLB) misses would be a good candidate. L1 accesses and L2 misses may also be candidates for detecting phase changes of memory demands. For example, in Figure 6, it shows the correlation between WSS and the occurrences of three events, L1 accesses, L2 misses and data TLB misses, for a set of `SPEC CPU2006` benchmarks. The three events are all closely correlated with WSS.



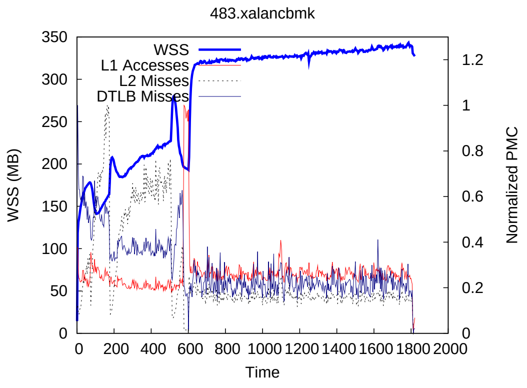
(a) 429.mcf



(b) 450.soplex



(c) 473.astar



(d) 483.xalancbmk

Figure 6: Examples of WSS and Performance Events
The left Y-axis shows the working set size (MB) and the right Y-axis shows normalized occurrences of the three events. Sample interval is 3 seconds.
In Fig 6(a), the lines for L2 and DTLB misses overlap most of the time and the spikes of L1 accesses overlap with sudden drops of WSS as well.

In addition, since some processors support counting multiple events simultaneously, a phase change could be determined by different policies. An *aggressive policy* would determine a phase change only if all the events encounter phase changes. This policy minimizes the time of memory tracking. On the contrary, a *conservative policy* is sensitive to any possible phase changes. That is, if any of the events show phase changes, it will turn on memory tracking. Beside, a moderate *voting policy* can decide a phase change only when the majority of the events show phase changes. Without extensive experiments, it is difficult to conclude which events and policy are the most appropriate ones. In Section 5.2.2, we compare the results of using different events and policies.

3.3.2 Phase Detection

Previous studies rely on sophisticated signal processing techniques such as Fourier transformation or wavelet analysis [12, 16] to detect cache-level phases. Though these techniques are able to effectively filter out noises and identify phase changes in off-line analysis, their prohibitive costs make them inappropriate for on-line phase detection.

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 (memory demand 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. If the moving average filter has not been filled up with k data, it means there is not enough information to make any decisions. So memory tracking is always turned on during this period. When 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 $\mathbf{T}$. Figure 7 illustrates its organization. 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 stability test of memory demands, it 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 used for memory resource balancing 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, which leads to 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. One solution to find the appropriate value of $\mathbf{T}$ is by means of experiments. By trying different $\mathbf{T}$ on an extensive set of programs, a $\mathbf{T}$ can be found such that the average overhead can be lowered with a tolerable accuracy loss.

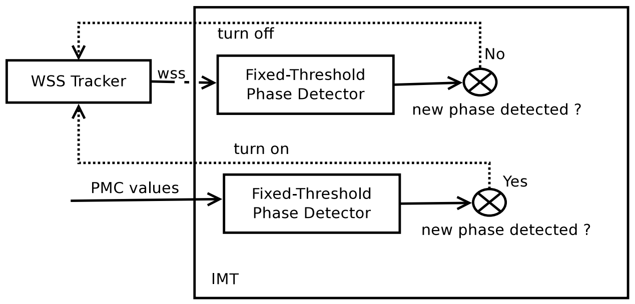


Figure 7: Fixed-Threshold IMT

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 8. 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.

As long as WSS tracking is enabled, $\mathbf{T}$ is calibrated to make decisions that are consistent with the stability of current WSS. 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 turned off 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. In the ideal case, memory tracking will be deactivated except for checkpointing. 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 its default value, $ckpt_{init}$. In the ideal case, the ratio of the time that memory tracking is on to the whole execution time, called *up ratio*, is nearly $1/ckpt_{max}$.

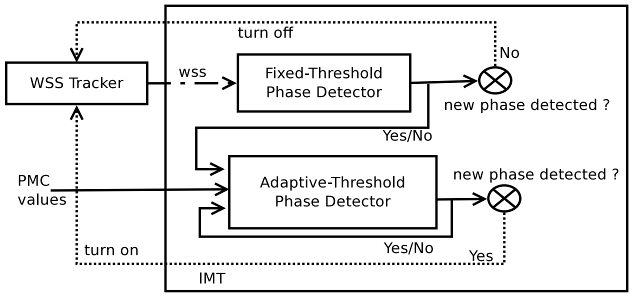


Figure 8: Adaptive-Threshold IMT

4 Implementation

To verify the effectiveness of our WSS tracking and evaluate its application in virtualized environments, we use the Xen 3.4 [17] 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 an access from the guest OS to that page is made, 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. Meanwhile, the the page is added to the hot set. If the hot set is full, the oldest page is evicted and set to be cold again. So if the evicted page is accessed again in future, it will be trapped.

To verify the effects of our WSS tracking in memory balancing, we use the VM memory balancer that was implemented in [8]. When multiple VMs compete for memory resource, given each VM's miss ratio curve, the balancer tries to find an allocation plan to achieve optimal overall performance. Each VM's memory allocation is resized through the ballooning mechanism [9].

Both IMT and the memory balancer run as services 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 "pause/resume" commands 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. Figure 9 illustrates the organization of our system implementation.

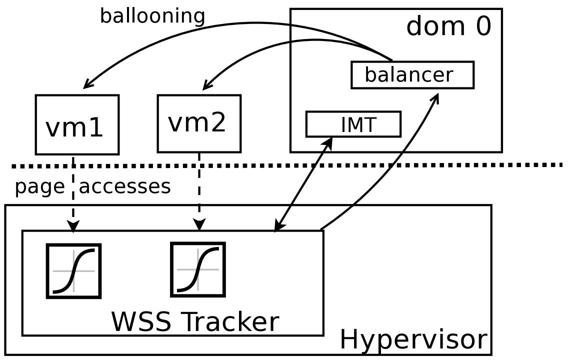


Figure 9: Implementation Overview

5 Evaluation

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 the memory balancing test). We select a variety of benchmark suites, including the `SPEC CPU2006` [18] benchmark suite, the `186.crafty` in `SPEC CPU2000` and `DaCapo` [19], a Java benchmark suite, to represent real world workload and evaluate the effectiveness of our work.

In this section, we first evaluate the effectiveness of DHS and AVL-tree based LRU organization. Then we measure the performance of IMT with various configurations. And we examine the overhead of WSS tracking again when all optimizations are put together. Finally, we design a scenario to demonstrate the application of WSS tracking.

5.1 Effectiveness of DHS and ABL

To measure the effects of various techniques on lowering overhead, we first measure the running time of SPEC 2006 and DaCapo benchmark suite without memory tracking as the baseline performance. For SPEC 2006, each program is measured individually. While for DaCapo, each of them is run in alphabetic order and the total execution time is measured because some programs finish in several seconds, which is too short for IMT to start working. Then we measure the running time of with memory tracking enabled, using plain linked list LRU implementation, dynamic hot set sizing, AVL-tree based LRU implementation and the combination of the latter two respectively. Columns two to five of Table 1 list the normalized overhead against the baseline setting. For the whole SPEC 2006 benchmark suite, the plain linked-list design incurs a mean overhead of 173%. Using DHS and ABL separately, the mean overheads are lowered to 39% and 43%, respectively. Applying DHS and ABL together, the mean overhead is further reduced to 16%. When the memory working set size is small or the locality is good, the advantage of ABL and DHS over the regular linked list implementation is not obvious. However, for benchmarks with large WSS, the performance gain of them is prominent. For example, in SPEC CPU2006 suite, the top three programs with the largest WSSs are `459.GemsFDTD`, `429.mcf` and `410.bwaves`, whose WSSs are 800 MB, 680 MB and 474 MB, respectively [20]. Using DHS and ABL together, the overhead against the linked-list setting is reduced by 69.8%, 98.7%, and 85.7%, respectively. For `483.xalancbmk`, although its WSS is merely 28 MB, its poor locality leads to a 681% overhead under the linked-list design. Replacing the linked-list LRU with the AVL-tree based LRU and applying DHS, its overhead is cut to only 5%. However, even both of ABL and DHS are enabled, for the whole SPEC 2006 benchmark suit, the mean overhead of 16% is still non-trivial.

5.2 Performance of Intermittent Memory Tracking

The performance of intermittent memory tracking 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 of the SPEC 2006 benchmarks and sample the memory demands and hardware performance counters every 3 seconds without intermittent memory tracking. Then, we feed the trace results to the intermittent memory tracking algorithm to simulate its operations. That is, given inputs $\{M_0, \ldots, M_i\}$ and $\{P_0, \ldots, P_i\}$, the intermittent memory tracking 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 when memory tracking is on. Given a trace with n samples, its mean relative error is computed as

$$MRE = (\sum_{i=1}^{n} \frac{|M_i - m_i|}{M_i})/n,$$

in which n is the number of samples.

5.2.1 Fixed-Threshold vs. Adaptive-Threshold

To evaluate the performance of fixed and adaptive thresholds for IMT, we use a DTLB miss as the hardware performance event for phase detection. For fixed thresholds, $\mathbf{T}$ varies from 0.05 to 0.3, two extreme ends of the spectrum. Table 2 shows the details.

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 $\mathbf{T}$ 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 $\mathbf{T} = 0.3$, while its mean relative error is even smaller than that of $\mathbf{T} = 0.05$. Clearly, the adaptive-threshold algorithm outperforms the fixed-threshold algorithm.

Figures 10, 11 and 12 show 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. Figures 10(a) and 10(b) show two simple cases. Both WSS and DTLB misses are stable during the whole execution except for a spike. Hence, most of the time, memory tracking is turned off except for checkpointing. Figures 11 shows the typical cases where there are multiple phases in terms of WSS and DTLB misses. `416.gamess` (see Figure 12(a)) is an exception. When examined from an overall scope, its

Program	Normalized Execution Time						Up Ratios	
	L	D	A	D+A	D+A+I(f)	D+A+I(a)	I (f)	I (a)
400.perlbench	1.54	1.28	1.07	1.05	1.00	1.02	0.22	0.11
401.bzip2	1.03	1.04	1.01	1.02	1.01	1.01	0.76	0.14
403.gcc	1.96	1.17	1.37	1.08	1.02	1.03	0.87	0.11
410.bwaves	3.30	2.81	1.30	1.33	1.19	1.02	0.56	0.15
416.gamess	1.01	1.01	1.01	1.01	1.00	1.00	0.18	0.09
429.mcf	59.16	9.07	3.21	1.75	1.41	1.04	0.72	0.37
433.milc	13.08	8.45	3.35	2.74	2.46	1.05	0.52	0.11
434.zeusmp	2.49	1.33	1.22	1.14	1.06	1.06	0.13	0.21
435.gromacs	1.01	1.01	1.00	1.00	1.00	0.99	0.13	0.10
436.cactusADM	1.20	1.12	1.08	1.09	1.00	1.02	0.14	0.15
437.leslie3d	2.68	1.01	1.37	1.00	1.00	1.00	0.13	0.10
444.namd	1.02	1.01	1.01	1.01	1.00	1.00	0.15	0.11
445.gobmk	1.07	1.02	1.02	1.02	1.01	1.01	0.38	0.10
447.dealII	1.34	1.09	1.17	1.03	1.02	1.01	0.87	0.11
450.soplex	3.16	1.27	1.43	1.13	1.01	1.10	0.49	0.21
453.povray	1.01	1.01	1.01	1.01	1.00	1.00	0.26	0.09
454.calculix	1.03	1.01	1.01	1.01	1.00	1.00	0.10	0.12
456.hmmer	1.01	1.01	1.01	1.01	1.00	1.01	0.25	0.09
458.sjeng	9.74	1.13	1.79	1.06	1.01	1.01	0.31	0.10
459.GemsFDTD	6.79	4.17	3.28	2.75	0.99	0.99	0.15	0.10
462.libquantum	7.89	1.02	1.29	1.02	1.00	1.01	0.15	0.10
464.h264ref	1.04	1.02	1.02	1.01	1.01	1.00	0.15	0.14
465.tonto	1.01	1.00	1.01	1.00	1.00	1.01	0.13	0.09
470.lbm	4.31	2.34	1.71	1.65	1.01	1.00	0.17	0.10
471.omnetpp	41.13	1.07	4.60	1.05	1.00	1.04	0.26	0.23
473.astar	15.77	1.02	2.92	1.01	1.01	1.00	0.51	0.13
481.wrf	1.18	1.12	1.12	1.13	1.00	1.00	0.13	0.09
482.sphinx3	1.03	1.01	1.02	1.02	1.00	1.00	0.14	0.09
483.xalancbmk	7.81	1.33	2.28	1.05	1.02	1.00	0.57	0.09
Mean (SPEC 2006)	2.73	1.39	1.43	1.16	1.06	1.02	0.26	0.12
DaCapo	1.28	1.07	1.14	1.07	1.04	1.01	0.82	0.20

Table 1: Normalized Execution Time and Up Ratios When IMT Is Enabled
L: linked list, D: dynamic hot set, A: AVL-Tree based LRU, I(f): IMT with fixed-threshold of 0.2, I(a): IMT with adaptive-threshold

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 12(b) 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.

Overall, adaptive-threshold based IMT achieves an up ratio of 11.5% with MRE of 3.9%. With adaptive-threshold IMT, the mean MRE of all other programs except for `429.mcf` is merely 2%. For `429.mcf`, as Figure 11(d) 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%.

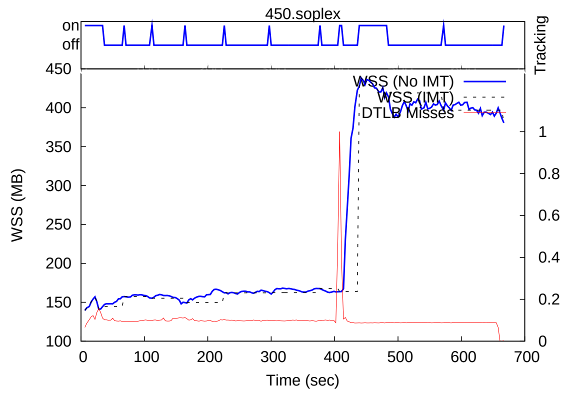
Program	**T** = .05		**T** = .10		**T** = .15		**T** = .20		**T** = .25		**T** = .30		Adaptive	
	UR	MRE	UR	MRE	UR	MRE	UR	MRE	UR	MRE	UR	MRE	UR	MRE
40.perlbench	.47	.075	.33	.119	.21	.162	.20	.167	.20	.167	.20	.171	.29	.031
401.bzip2	.83	.003	.79	.007	.68	.013	.62	.016	.58	.023	.50	.038	.23	.035
403.gcc	.84	.002	.80	.008	.68	.017	.64	.017	.61	.017	.51	.020	.23	.036
41.bwaves	.47	.020	.45	.022	.45	.022	.43	.012	.40	.014	.35	.019	.13	.045
416.gamess	.26	.017	.03	.045	.03	.045	.03	.045	.03	.045	.03	.045	.07	.009
429.mcf	.53	.025	.40	.042	.35	.103	.33	.146	.29	.352	.24	.621	.23	.387
433.milc	.21	.054	.07	.108	.06	.132	.06	.132	.02	.167	.02	.167	.08	.077
434.zeusmp	.80	.004	.54	.009	.04	.018	.04	.018	.04	.018	.04	.018	.13	.017
435.gromacs	.13	.005	.05	.011	.04	.011	.04	.011	.04	.011	.04	.011	.09	.002
436.cactusADM	.08	.114	.06	.095	.05	.086	.04	.083	.04	.083	.04	.083	.08	.015
437.leslie3d	.06	.009	.03	.016	.03	.016	.03	.016	.03	.016	.03	.016	.07	.003
444.namd	.10	.002	.05	.003	.05	.003	.05	.003	.05	.003	.05	.003	.10	.001
445.gobmk	.60	.004	.15	.012	.05	.021	.05	.021	.05	.021	.05	.021	.10	.008
447.dealII	.98	.000	.98	.000	.97	.000	.99	.000	.96	.001	.88	.010	.50	.039
45.soplex	.18	.041	.18	.041	.14	.083	.13	.084	.13	.084	.13	.084	.16	.037
453.povray	.17	.062	.17	.062	.17	.062	.16	.064	.16	.064	.15	.065	.17	.011
454.calculix	.14	.015	.02	.063	.02	.063	.02	.063	.02	.063	.02	.063	.06	.005
456.hmmer	.23	.054	.03	.740	.03	.740	.03	.740	.03	.740	.03	.740	.13	.035
458.sjeng	.09	.029	.07	.031	.05	.059	.02	.061	.02	.061	.02	.061	.10	.026
459.GemsFDTD	.03	.004	.02	.005	.02	.005	.02	.005	.02	.005	.01	.005	.05	.006
462.libquantum	.04	.000	.04	.002	.04	.002	.04	.002	.04	.002	.04	.003	.07	.002
464.h264ref	.60	.002	.56	.006	.14	.014	.07	.022	.07	.022	.07	.022	.08	.005
465.tonto	.79	.001	.29	.017	.14	.109	.14	.109	.14	.109	.14	.109	.30	.035
47.lbm	.05	.025	.05	.025	.05	.025	.05	.025	.05	.025	.04	.025	.09	.006
471.omnetpp	.02	.426	.02	.426	.02	.426	.02	.426	.02	.426	.02	.426	.07	.036
473.astar	.16	.004	.13	.004	.13	.006	.12	.009	.10	.017	.10	.017	.08	.010
481.wrf	.25	.017	.17	.019	.05	.061	.05	.028	.05	.028	.06	.011	.08	.005
482.sphinx3	.30	.004	.04	.051	.04	.051	.04	.051	.04	.052	.04	.052	.08	.005
483.xalancbmk	.19	.070	.14	.106	.10	.112	.09	.106	.09	.106	.09	.107	.14	.032
Mean	.27	.057	.19	.089	.14	.099	.13	.100	.12	.113	.11	.126	.11	.039

Table 2: Up Ratios and MREs With Respect To Various Fixed-Thresholds and Adaptive-Threshold

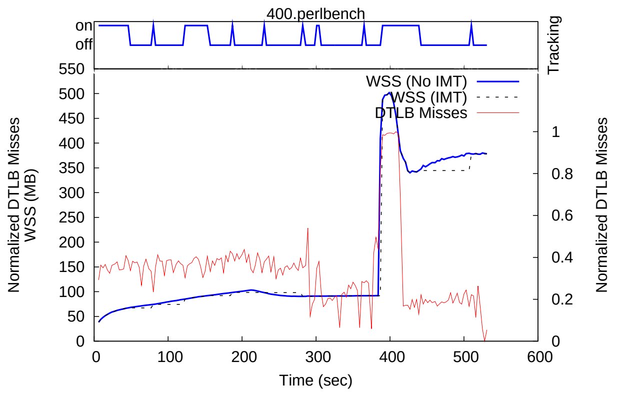
5.2.2 Selection of Hardware Performance Events

In addition to the fixed/adaptive threshold algorithms, the other dimension of the design space of IMT is the selection of hardware performance events. For comparison purpose, three memory related events are traced simultaneously: DTLB misses, L1 references and L2 misses. First, each of the events is used separately. Second, all combinations of these events are tested with different phase identification policies. Table 3 lists the test results. The first column lists the name of the monitored events and the letter in parentheses denotes the adopted policy: "a", "c" and "v" stand for the aggressive policy, the conservative policy and the voting-based policy, respectively. (The details of each policy are discussed in Section 3.3.1).

Interestingly, each of the single events and any combinations of them with the aggressive policy achieve similar performance. The conservative policy gives the best accuracy. However, the up ratios are the highest too. When all three events are used, the voting policy shows a moderate result. Since using more than one event does not boost performance significantly, in the following experiments, we use only DTLB misses.



(a) 450.soplex



(b) 400.perlbench

Figure 10: Examples of Using Adaptive-Thresholds IMT (Simple Cases)

Events and Policies	UR	MRE
DTLB	0.115	0.039
L2 Miss	0.103	0.046
L1 Ref	0.118	0.041
DTLB+L1 (a)	0.114	0.031
DTLB+L1 (c)	0.185	0.016
DTLB+L2 (a)	0.102	0.044
DTLB+L2 (c)	0.179	0.017
L1+L2 (a)	0.106	0.041
L1+L2 (c)	0.177	0.020
DTLB+L1+L2 (a)	0.105	0.041
DTLB+L1+L2 (v)	0.137	0.025
DTLB+L1+L2 (c)	0.190	0.018

Table 3: Effects of Different Hardware Performance Events and Policies

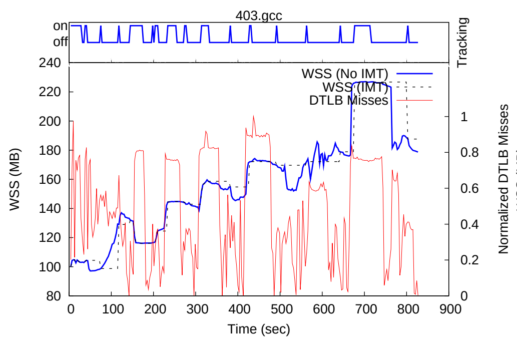
5.2.3 Overhead Revisited

To evaluate the actual effects of using IMT, we measure the WSS tracking overhead on actual runs. As Table 1 shows, for `SPEC CPU2006`, even optimized with AVL-based LRU and dynamic hot set sizing, the mean overhead 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.

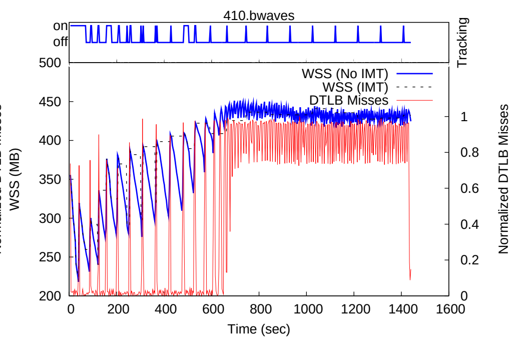
Enhanced with fixed-threshold IMT ($\mathbf{T} = 0.2$), the mean overhead is lowered to 6%. Using adaptive-threshold IMT, the mean overhead is further reduced to 2% by cutting off half of the up time of memory tracking.

5.3 Application to VM Memory Balancing

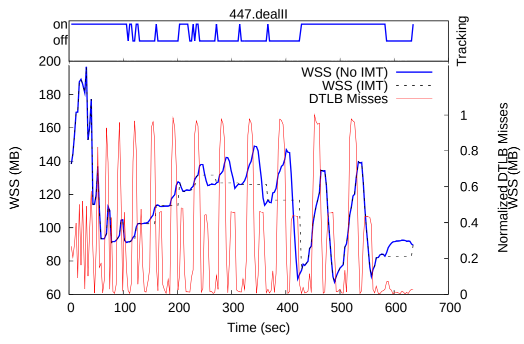
One typical scenario of the working set tracking is memory resource balancing. Two virtual machines are monitored on a Xen-based host. We design two experiments with different benchmarks and initial memory size. In the baseline setting of both experiments, no memory balancing or WSS tracking is used. When memory balancing is used, three variations are compared: memory tracking without IMT, using IMT with a fixed threshold of 0.2 and using IMT with adaptive threshold.



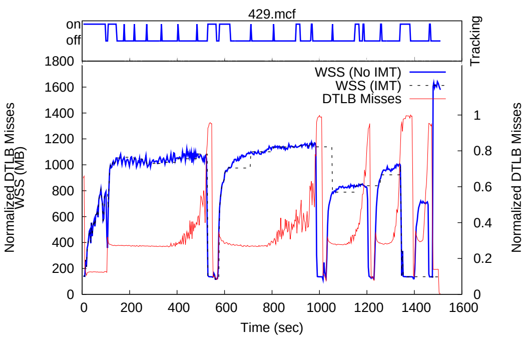
(a) 403.gcc



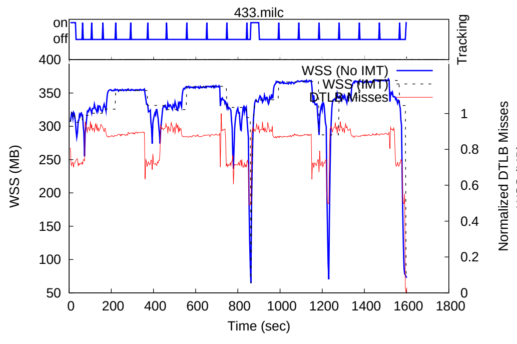
(b) 410.bwaves



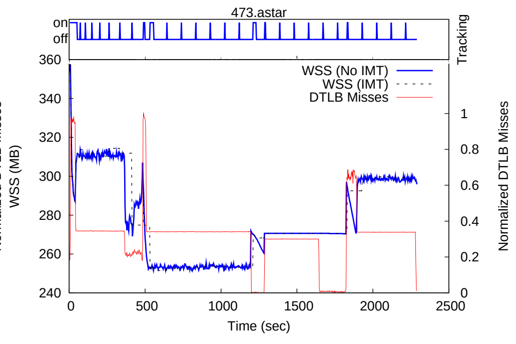
(c) 447.dealII



(d) 429.mcf

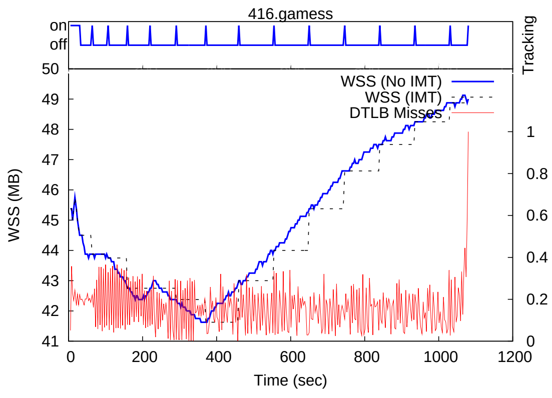


(e) 433.milc

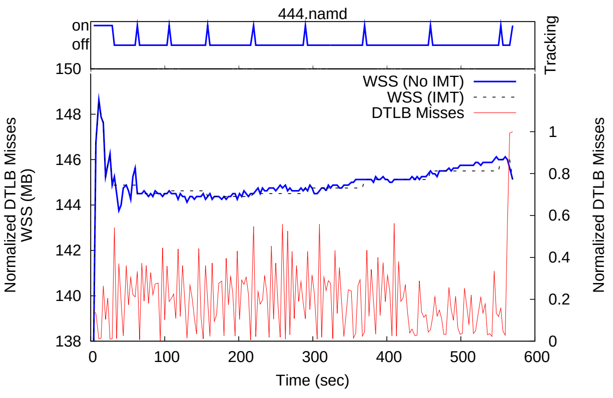


(f) 473.astar

Figure 11: Examples of Using Adaptive-Thresholds IMT (Common Cases)



(a) 416.gamess



(b) 444.namd

Figure 12: Examples of Using Adaptive-Thresholds IMT (Special Cases)

Experiment One Initially, each VM is allocated 250 MB of memory. One VM runs the *DaCapo* benchmark, which consists of ten Java programs with various working set sizes. The other VM runs *186.crafty*, a SPEC 2000 program that uses few memory resources. Figure 5.3 shows the normalized speed-ups with memory balancing against the baseline setting.

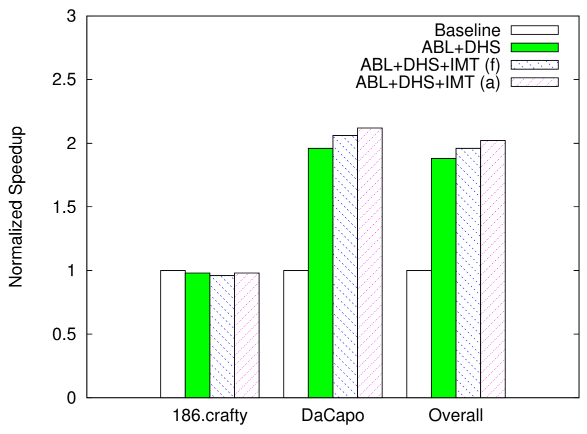


Figure 13: Memory Balancing of DaCapo and 186.crafty

For *186.crafty*, its performance degrades by 2% − 4% due to the overhead of memory tracking. For *DaCapo*, even with the overhead of memory tracking, its performance is boosted by at least 96% due to the extra memory it gets from the other VM. As a result of 3% and 6% overhead reduction provided by fixed-threshold IMT and adaptive-threshold IMT respectively, the speed-up of DaCapo is increased from 1.96 to 2.06 and 2.12 correspondingly. Compared with no IMT and fixed-threshold IMT, adaptive-threshold IMT delivers the best performance due to its low overhead and high accuracy.

Experiment Two Initially, each VM is allocated 700 MB of memory. One VM runs `470.lbm`, meanwhile, the other VM runs `433.milc`. Figure 14 shows the normalized speedups with memory balancing against the baseline setting. Note that, the balancer is designed to reclaim unused memory, so the total allocated memory to the two VMs may be less than 1400 MB.

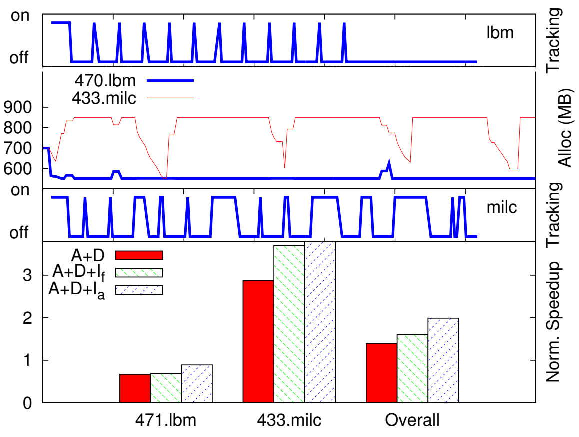


Figure 14: Memory Balancing of 470.lbm and 433.milc

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.

6 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 improve previous achievements on overhead control and present a novel intermittent memory tracking scheme. Experimental evaluation of shows that our solution is capable of reducing the overhead with enough 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. Moreover, we expect to develop a system to derive a cache-level miss ratio curve with the assistance of a page miss ratio curve, or vice versa.

References

[1] P. J. Denning, “Working sets past and present,” *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, 1980.

[2] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger, “Evaluation techniques for storage hierarchies,” *IBM System Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[3] D. Chandra, F. Guo, S. Kim, and Y. Solihin, “Predicting inter-thread cache contention on a chip multi-processor architecture,” in *HPCA ’05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 340–351.

[4] 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-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2004, pp. 177–188.

[5] T. Yang, E. D. Berger, S. F. Kaplan, J. Eliot, and B. Moss, "CRAMM: virtual memory support for garbage-collected applications," in *OSDI '06: Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 103–116.

[6] 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.*, vol. 40, no. 5, pp. 14–24, 2006.

[7] R. A. Sugumar and S. G. Abraham, "Efficient simulation of caches under optimal replacement with applications to miss characterization," in *SIGMETRICS '93: Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, May 1993, pp. 24–35.

[8] W. Zhao and Z. Wang, "Dynamic memory balancing for virtual machines," in *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. New York, NY, USA: ACM, 2009, pp. 21–30.

[9] C. A. Waldspurger, "Memory resource management in VMware ESX server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, 2002.

[10] P. Lu and K. Shen, "Virtual machine memory access tracing with hypervisor exclusive cache," in *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–15.

[11] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multi-core cache management," in *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer systems*. New York, NY, USA: ACM, 2009.

[12] X. Shen, Y. Zhong, and C. Ding, "Locality phase prediction," in *ASPLOS '04: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2004.

[13] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *ISCA '03: Proceedings of the 30th International Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2003.

[14] 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: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2009, pp. 121–132.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 2005, pp. 190–200.

[16] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," La Jolla, CA, USA, Tech. Rep., 2001.

[17] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, 2003.

[18] SPEC CPU2006. http://www.spec.org/cpu2006. [Online]. Available: http://www.spec.org/cpu2006

[19] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[20] D. Gove, "CPU2006 working set size," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 90–96, 2007.