Dynamic Tracking of Page Miss Ratio Curve for Memory Management *

Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou and Sanjeev Kumar†

Department of Computer Science,
University of Illinois at Urbana-Champaign
{pinzhou, pandey1, sundarsn, raghuram, yyzhou}@uiuc.edu

†Intel Labs
Santa Clara, CA
Sanjeev.Kumar@intel.com

ABSTRACT

Memory can be efficiently utilized if the dynamic memory demands of applications can be determined and analyzed at run-time. The page miss ratio curve(MRC), i.e. page miss rate vs. memory size curve, is a good performance-directed metric to serve this purpose. However, dynamically tracking MRC at run time is challenging in systems with virtual memory because not every memory reference passes through the operating system (OS).

This paper proposes two methods to dynamically track MRC of applications at run time. The first method is using a hardware MRC monitor that can track MRC at fine time granularity. Our simulation results show that this monitor has negligible performance and energy overheads. The second method is an OS-only implementation that can track MRC at coarse time granularity. Our implementation results on *Linux* show that it adds only 7-10% overhead.

We have also used the dynamic MRC to guide both memory allocation for multiprogramming systems and memory energy management. Our *real system* experiments on Linux with applications including Apache Web Server show that the MRC-directed memory allocation can speed up the applications' execution/response time by up to a factor of 5.86 and reduce the number of page faults by up to 63.1%. Our execution-driven simulation results with SPEC2000 benchmarks show that the MRC-directed memory energy management can improve the Energy * Delay metric by 27-58% over previously proposed static and dynamic schemes.

*This work is supported in part by an equipment donation from AMD, IBM SUR grant, a gift from Intel Corp., and the National Science Foundation under Grant No. CCR-0305854, and CCR-0313286.

ASPLOS'04, October 9–13, 2004, Boston, Massachusetts, USA.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*main memory*

General Terms

Algorithms, Management, Performance

Keywords

Memory management, Power management, Resource allocation

1. INTRODUCTION

1.1 Motivation

Main memory plays an increasingly important role in bridging the widening gap between processor and disk speeds. Even though memory sizes are increasing, memory is a scarce resource for computing environments with large data sets. This is especially the case for (i) host centers where multiple applications or virtual machines run on one physical machine [22], and (ii) mobile devices that have limited amount of memory due to thermal and packaging considerations. Moreover, the memory price also grows exponentially with size above certain standard configurations (1GB or 2GB). In addition, large physical memory can also lead to significant energy consumption in host centers [38, 10]. Therefore, memory management remains an important research problem.

This paper focuses on two aspects of efficient memory management. (1) *Memory allocation:* In a system with multiple applications, memory should be apportioned to applications in a way that maximizes overall system performance, while ensuring that every application receives a certain minimum amount of memory. Applications which are unlikely to show a significant performance gain with extra memory should not be given more memory. (2) *Memory energy management:* If by giving extra memory to a process, the improvement in performance is marginal, it is better to power down the extra memory to conserve energy. Optimizing energy consumption of computing components has become important not only for mobile, wireless and embedded devices due to short battery life, but also for high-end systems to reduce electricity bills, which contribute to more than 25%

of a host center's TCO (total-cost-of-ownership) [43]. Memory is a particularly important target for efforts to address the energy efficiency issues. It has been observed by previous studies [9, 32, 38, 56] that the memory subsystem is one of the dominant consumers of the overall system energy. Recent measurements from real server systems show that memory consumes as much as 50% more power than processors [39]. To address this problem, many modern memory modules support multiple low-power operating modes to conserve energy [49].

Even though the above two aspects of memory management focus on two different issues, both can benefit from information about the dynamic memory needs of the applications. A metric called *working set size* is often used to capture an application's dynamic memory behavior. The working set model was first proposed by Denning [19] and is defined as the number of referenced pages during a time period (called working set window). Various researchers have proposed techniques such as WSCLOCK [8] and VSWS [26] to dynamically estimate working sets and use them to guide page replacements.

However, the working set model is not enough to guide memory allocation or memory energy management because the working set does not relate directly to an application's performance. For example, if an application sequentially accesses 1000 different pages within the working set window, its working set would contain 1000 pages whereas the operating system needs to allocate only a small number of pages to this application for it to deliver the same performance (0% page hit ratio).

A better alternative is to use the page miss ratio curve (MRC), the miss ratio vs. memory size curve, to keep track of dynamic memory needs of applications. Previous studies [50, 59, 3, 34] have used MRC to configure the memory hierarchy for applications. These studies obtain MRC *statically* by running the applications multiple times, each time using a different memory size. Such a method is ineffective for many applications whose memory requirements vary for different computation phases or different user inputs. In addition, the multiprogramming environment makes the memory availability hard to predict statically. To deal with these issues, techniques that can dynamically track MRC at run time are desirable.

Even though dynamically tracking MRC is relatively easy for file systems by using ghost buffers [36, 44], it is quite challenging to track MRC for the virtual memory (VM) system. Unlike file systems where each file access goes through the file system, not every memory access is known to the OS. Only page misses are handled by the OS, while all the page hits directly access the physical memory without going through the OS. Therefore, the simple techniques used in file systems cannot be used to track MRC in systems with virtual memory.

1.2 Contributions

In this paper, we propose two methods to dynamically track applications' MRC and then use the MRC to guide memory allocation and memory energy management. To the best of our knowledge, our methods are the first in tracking MRC at run time in virtual memory. Our study is also the first in using MRC to direct memory allocation and memory energy management. Specifically, our paper has the following contributions:

- We have designed a hardware MRC monitor to dynamically track MRC for applications at run time. This MRC monitor is relatively simple and has low power requirement, as shown by our execution-driven simulation results.

- We have also proposed a software-only solution that can track MRC approximately at run time in the OS. Our implementation results on Linux with real applications show that this software only solution imposes only 7-10% overhead.

- We have applied MRC to direct memory allocation to maximize memory utilization in multiprogramming systems. Our *real* system experiments on Linux with real applications, including Apache Web Server, show that this scheme can speed up applications' execution/response time by up to a factor of 5.86 and reduce the number of page faults by up to 63.1%.

- We have also used MRC to direct memory energy management in order to reduce energy consumption while still providing acceptable performance. Our execution-driven simulation results with SPEC2000 benchmarks show that MRC-directed memory energy management can improve the Energy * Delay metric by 27-58% over previously proposed schemes.

1.3 Paper Organization

The rest of the paper is organized as follows. Section 2 briefly describes the background of this work. Section 3.1 and Section 3.2 respectively present the hardware and the software methods to dynamically track MRC at run time. Section 4 describes the MRC-directed memory allocation scheme and the experimental results observed on real systems. Section 5 presents the MRC-directed memory energy management scheme and the simulation results obtained with an execution driven simulator. Section 6 discusses the related work. Section 7 concludes the paper.

2. BACKGROUND

2.1 Miss Ratio Curve (MRC)

Memory can be efficiently utilized if an application's *dynamic* memory behavior can be analyzed at run time. One effective method to measure an application's memory needs is to use its dynamic miss-ratio curve (MRC) that plots the application's page miss-ratios against varying amounts of physical memory. Please note, in our paper, MRC is for accesses filtered by caches. Similar to all previous work on main memory management, only accesses to physical memory (i.e. cache misses) are considered. Figure 1 shows the MRC for the bzip benchmark from SPEC2000 during a particular time interval (MRC may dynamically change in different time intervals). We see that MRC decreases initially but eventually starts to flatten out after a certain point (say n pages). This indicates that even if the system provides more than n memory pages to this application during that interval, it does not help in reducing its miss ratio significantly. In other words, an efficient memory manager should allocate only n pages to handle most of accesses made by this application during that interval(assuming that all n pages can fit into main memory).

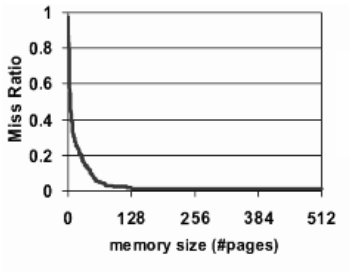


Figure 1: Miss Ratio Vs. Memory Size Curve (MRC) for bzip

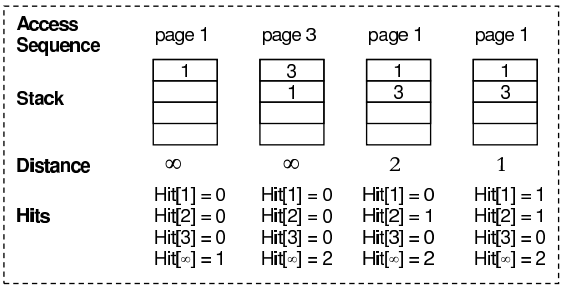


Figure 2: Illustration of Mattson's stack algorithm

MRC provides useful information about applications' dynamic memory needs. For example, we can use MRC to dynamically allocate memory to multiple applications to maximize the overall system performance. Similarly, MRC can also be used to direct memory energy management. If applications need only n pages to provide the best possible page miss ratio during a period, the system needs to keep only n pages active and can power down the rest of memory to conserve energy in that period.

In the past, a metric called "working set size" has been widely used to capture applications' memory needs [20, 18, 19]. For an application, the working set is defined as the set of pages that it has recently referenced. Formally speaking, the working set of an application is the collection of pages referenced during the recent working set window (time interval). The working set size is just the memory footprint size of the working set. For example, if an application sequentially accesses 10000 pages within a working set window, its working set size is 10000 (pages), even though its page miss rate is always 100% no matter how much physical memory is allocated to it.

Working set and MRC are two different but complementary models. While an application's working set can be used to determine *what* pages should be replaced or what pages can be powered down, the MRC is better suited to determine *how many* pages should be allocated to an application or how many pages can be powered down. In our memory allocation and memory energy management schemes, we have used both MRCs and working sets. Our methods for dynamic tracking of an application's MRC also track the application's working set.

Most previous works [50, 59, 3, 34] obtain an application's MRC statically by running the application multiple times, each time with a different memory size. However, most applications are dynamic and the multiprogramming environment makes the memory resource availability hard to predict statically. Therefore, static methods are not very useful for dynamic memory management. An algorithm that can dynamically determine the MRC is better suited for memory management.

2.2 Mattson's Stack Algorithm

Our methods for dynamically tracking MRC at run time are based on the Mattson's stack algorithm, which was initially proposed by Mattson et al. in 1970 [24] to reduce trace-driven processor cache simulation time. It can be used to determine the miss ratios for all processor cache sizes under certain replacement algorithms with a single pass through the trace file. Mattson's stack simulation method was later extended by Hill et al. [29, 28] and Wang et al. [57] to provide efficient trace simulations for set associative caches.

The main idea of Mattson's stack algorithm is to take advantage of the *inclusion property* in many cache/memory replacement algorithms [24] including the commonly used Least Recently Used (LRU) and the Least Frequently Used (LFU) algorithms. The inclusion property states that, during any sequence of memory accesses, the contents of a memory of size k (pages) should be a subset of the contents of a memory of size $k+1$ or larger. To explain Mattson's stack algorithm, we assume LRU as the replacement algorithm since virtual memory is typically managed using the CLOCK algorithm, a close approximation of the LRU algorithm [52]. We use a "stack" to store the page numbers of accessed pages. The entries in the stack are ordered based on the time of their last use—the most recently used page is on the top of the stack. Then for a given trace, the Mattson's stack algorithm consists of the following three steps for each reference:

Step1: Search the referenced page number in the stack. If the referenced page is the i^{th} element in the stack from top, its stack distance is i. If the referenced page is not in the stack, its distance is ∞.

Step2: Update the hit counter for the memory size corresponding to the stack distance. For a reference at distance i, the hit counter for the memory size i, i.e. $Hit[i]$ is increased by 1. This reflects the fact that if the memory size is between 1 and $i-1$ pages, then the reference will result in a page miss. For memory sizes greater than or equal to i, the reference will result in a page hit.

Step3: Update the stack to reflect the memory content based on the replacement algorithm. For example, under the LRU policy, the referenced page is moved to the top of the stack.

Finally, after processing the whole trace, miss ratios can be calculated for multiple memory sizes according to the hit counters. The miss ratio for memory size m pages, $MR(m)$ for a system with total memory of n pages is given by the following formula.

$$MR(m) = 1 - \frac{\sum_{i=1}^{m} Hit[i]}{\sum_{i=1}^{n} Hit[i] + Hit[\infty]} \qquad (1)$$

where $Hit[\infty]$ is the number of accesses which result in a miss even if the process is allocated all n pages. Figure 2 shows an example of how Mattson's Stack Algorithm works. In this example, the reference sequence is "1311". Using the above formula, we calculate the miss ratios for memory sizes of 1, 2, and 3 pages as 0.75, 0.5, and 0.5 respectively.

Even though Mattson's stack algorithm was originally used for cache simulation, it can also be used to track MRC for

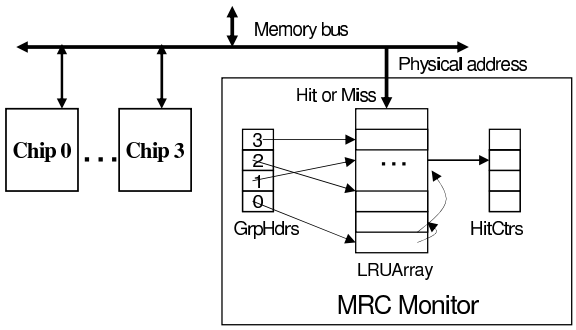


Figure 3: MRC monitor architecture

file system accesses [36, 44] and physical memory accesses as in this study.

3. RUN-TIME TRACKING OF MRC

While the Mattson's algorithm gives us the basic method to find the MRC at runtime, implementing it without incurring high space or time overhead is challenging. Unlike file systems that can monitor every file access, not every memory access goes through the OS. Therefore, it is infeasible for the OS to use a straightforward implementation of the Mattson's algorithm as done in some file system studies [36, 44].

To dynamically generate the MRC of an application, time is divided into fixed intervals called *epochs*. During each epoch, we use a variant of the Mattson's stack algorithm to record the observed accesses. At the end of each epoch, the MRC for that epoch is computed.

This section presents two implementations of the Mattson's stack algorithm[1] to dynamically track an application's MRC. The first implementation is a hardware MRC monitor that tracks the MRC of an application at a fine time granularity. The second method is an OS-only solution that tracks the MRC at a coarse time granularity.

3.1 Method 1 : Hardware Approach

We design a hardware module, the *MRC monitor*, to dynamically track MRC. The MRC monitor connects to the memory bus and can observe all memory accesses by snooping the bus. It can also be built into the memory controller. The advantage of the hardware approach is that it allows MRC to be computed at a fine granularity. The disadvantage is that it requires special-purpose hardware.

The MRC monitor consists of three buffer-like structures: an LRU stack *LRUArray*, a group header array *GroupHdrs*, and a hit counter array *HitCtrs*. Figure 3 shows the architecture of the MRC monitor. Table 1 gives the size of each component and the number of operations at a memory access for each component.

The *LRUArray* maintains an LRU stack. Each entry in the stack corresponds to a physical page. The *LRUArray* is indexed directly using the page number. For instance, if the page size is 4 KB, then the 20 most significant bits in the physical address form the index. The LRU order is

[1]To efficiently track the MRC, changes to basic Mattson's algorithm are made so that the MRC is computed approximately. However, the computed MRCs are accurate enough to be useful (See the results of Section 4 and 5).

Component	Size (in Bytes)		Operations/Access
	General	Example	
LRUArray	$8 * nps$	2 MB	$(7 + 2 * I)$ accesses
GrpHdrs	$4 * ng$	1 KB	$2 * I$ accesses
HitCtrs	$4 * (2 + ng)$	1 KB	2 counter increment

Table 1: MRC Monitor components. Size of the components and number of operations at an access to a page in group I. nps is the number of physical pages in DRAM and ng is the number of groups. In the example, the DRAM has capacity of 1 GB, with 256 groups each having 1024 pages of size 4 KB.

implemented by using a doubly linked list. Each entry in *LRUArray* has three fields *(GroupID, next, prev)*, a total of 8 bytes. *GroupID* uses 2 bytes to store the page group to which the corresponding page currently belongs(Significance of page groups is explained in the next paragraph). The pointers *next* and *prev*, requiring 3 bytes each, are used to maintain a doubly-linked list of the entries in the LRU order.

The MRC monitor uses page groups to avoid scanning the entire *LRUArray* to compute and update the stack distances on a memory access. The idea is similar to the fast stack implementation by Kim and Hill to reduce the trace simulation time for hardware caches [37]. Pages in the LRU stack are grouped into multiple page groups, following the LRU order. Each entry in *GrpHdrs* points to the header of the corresponding page group. For example, the first entry in *GrpHdrs* points to the top of the LRU stack, the most recently accessed page. Pages in the same group have the same stack distance. The stack distance for a page in the group I is $I \times GrpSize$, where $GrpSize$ is the number of pages in each group. Even though this page group scheme treats pages in the same group equally, the amount of loss in accuracy is small, which can be traded off for efficiency.

The array *HitCtrs* (4 bytes per entry) keeps track of the number of hits for different memory sizes. Since the stack distance is only accurate at the granularity of page group, *HitCtrs* only needs to record the numbers of hits for memory sizes that are integral multiples of *GrpSize*. In particular, counter j $(1 \leq j \leq NumGroups)$ in *HitCtrs* records the number of accesses that would result in a hit with a memory of size $j * GrpSize$ pages, but a miss for any smaller memory size. The last two counters in *HitCtrs* record the total number of memory accesses and the total number of misses respectively. For example, on an access to a page that currently belongs to page group i, counter i and the penultimate counter (total accesses) of *HitCtrs* need to be incremented by one.

In case of a page miss, the OS page fault handler selects a victim page for replacement and informs the MRC monitor regarding the selected physical page frame to hold the missed virtual page. The entry in the *LRUArray* corresponding to the physical page is marked as invalid. After the page fault is handled and the process accesses the virtual page again, the MRC monitor observes this access and notices that the corresponding entry is invalid. Therefore, the stack distance for this access is infinite and the last two entries in *HitCtrs* is incremented by one.

Not every access to memory needs to update the structures of the MRC monitor. The MRC monitor can record the pages of several recent accesses to exploit temporal locality. An access to a recorded page only needs to increment

the first and the last counter in *HitCtrs* since the accessed page would be in page group 1 and among the top *GrpSize* elements of the LRU stack. If the workload has good temporal locality, this optimization would significantly reduce the number of operations in the MRC monitor. However, this optimization is currently not implemented in our simulator.

At the end of an epoch, the MRC can be generated from the data in *HitCtrs* using equation 1 by noting that $Hit[i]$ corresponds to $HitCtrs[i]$ and i is now an index over page groups rather than individual pages. The denominator in equation 1 need not be explicitly computed, as we store the total number of accesses in the penultimate counter in *HitCtrs*. Since the hardware MRC monitor can observe all memory accesses, it can generate MRC at a fine granularity. This is essential because a short epoch length of around 10,000 CPU cycles is necessary for MRC-directed memory energy management (Section 5).

The MRC monitor described so far tracks the miss ratio-curve for the entire system including operating system and all applications. For applications like memory energy management (Section 5), this is appropriate because it can be used to conserve the energy in the system as a whole. However, for applications like memory allocation in a multiprogramming system (Section 4), it is better to compute MRCs for each process separately. To achieve this, the *GrpHdrs* and *HitCtrs* should be private to the running process. The *LRUArray* would be shared among all processes, but inside the *LRUArray*, pages that belong to the same process are linked together following the LRU order. At a context switch, the *GrpHdrs* and *HitCtrs* should also be swapped.

Since the MRC Monitor works in parallel with memory accesses, it imposes little runtime overhead. Typically, most memory instructions will access the processor caches and only a small fraction of memory instructions executed will result in memory accesses which are our only concern for memory management. For applications whose memory access rate is too high for the MRC monitor to keep up, a queue can be used to buffer some accesses. In the rare event that the queue overflows, a few accesses would be dropped. However, this is unlikely to significantly distort the MRC, and also we do not need very accurate MRC for both memory management and power management.

3.2 Method 2: OS Approach

Although the MRC monitor can track an application's MRC at fine time granularity, it requires extra hardware support. For some purposes, such as memory allocation for multiprogramming environments, tracking MRC at fine time granularity might not be required. To provide a simpler solution to serve these purposes, we have designed an OS-only method to dynamically track MRC at run time. We have implemented this method in the Linux kernel and have conducted experiments based on this implementation.

There are several challenges in implementing the Mattson's stack in the OS without extra hardware support. (1) *Accuracy*. Not every memory access passes through the OS. The OS is only notified upon page faults. Page hits are handled directly by the hardware. The hardware sets the "accessed" bit for the accessed page at every page reference. Consequently, the OS can get only a rough approximation of the set of pages referenced during an interval by resetting the accessed bits at the beginning of the interval and scanning the bits at the end of the interval. (2) *Time Overhead*. Scanning the "accessed" bit of all the pages would result in unacceptable time overheads. Moreover we need to scan these bits frequently enough so that we do not miss the accesses to the pages. (3) *Space Overhead*. Implementing a full stack for an application's entire virtual address space would require a lot of space, which may offset the benefits of MRC-directed memory management.

To address these challenges, our OS implementation makes three key improvements to the basic Mattson's stack algorithm: (1) It uses page protection to reduce the number of pages whose "accessed" bit needs to be scanned, so that the scanning can be done with small time overhead (2) Similar to the hardware MRC monitor, the OS implementation also uses page groups instead of pages as the basic memory unit size to reduce the time overhead. (3) To reduce the space overhead, the maximum number of entries in the stack are limited by the number of total physical pages. Due to space limitations, we give only a brief overview of our OS implementation with focus on these three ideas. More details can be found in [46].

For each monitored process, the OS maintains the following data structures: (1) A doubly linked list to maintain the stack order. Each entry contains a virtual page number and a page group ID. (2) A group header array similar to the one used in the hardware MRC monitor for tracking the headers of page groups. (3) A hit counter array to record the number of hits for different memory sizes. We only keep track of sizes that are multiples of page groups. (4) A pointer called "scan list pointer", which points to one of the elements in the doubly linked list. All the elements in the stack which are above this pointer form the *scan list*, the significance of which will be explained shortly.

The OS needs to determine the set of recently accessed pages so that it can update the hit counters. To reduce the overhead of scanning the accessed bits, the OS implementation partitions the pages in the physical memory into two groups. For the recently accessed pages (members of the scan list), the OS checks the accessed bits to determine if they were accessed. To catch references to the infrequently accessed pages, the OS uses the page protection mechanism.

The OS periodically scans the accessed bits for pages that are in the scan list. If the accessed bit for such a page is set, the OS calculates the stack distance based on the group ID associated with the corresponding stack entry. It then increases the corresponding hit counter by 1. If this page is not in group 0, it is "moved" to the top of the stack by manipulating the doubly linked list and its page group ID is set to be 0. For all page groups in between, their last elements are pushed to the next page group.

For detecting references to pages that are not in the scan list, the OS switches off the read-write permission of these pages. An access to such a page causes a protection fault. This is similar to the Shared Virtual Memory (SVM) systems [40]. Then the OS calculates the stack distance, updates the hit counter array, grants the read-write permission to this page to avoid future faults, and moves the corresponding entry to the top of the stack, thus adding it to the scan list. The last page in the scan list is evicted from the list by switching off its read-write permission to detect its future accesses, and scan list pointer is adjusted to reflect that this page is no more in the scan list.

The page protection optimization incurs only small overhead because of the following reasons: (1) Since most pro-

grams have good temporal locality, the probability that non recently-used pages (whose read-write permission is set off) are accessed is relatively low. Therefore the number of such protection faults is small. (2) Only the first access to a protected page triggers a page fault (since then the protection bit is turned off), and thus the overhead of handling the protection faults is amortized over many subsequent accesses.

If the number of page protection faults are large (could be caused by non-LRU accesses, such as sequential accesses), it indicates that the temporal locality is poor. In this case, we can increase the scan list length and disable the page protection optimization. Moreover, previous work has proposed many techniques to capture loop references or sequential references [36]. We can incorporate those techniques to track MRC without scanning the whole list for these regular access patterns. However, in our current prototype we did not include this.

We have implemented the above method in Linux by modifying the virtual memory system and adding around 1000 lines of code. In addition to the background scanning procedure for the NRU (not recently used) page replacement algorithm implemented by the original Linux system, we have added another scanning procedure to scan the reference bits of the scan list. Due to optimizations described above, the time overhead of our procedure is very low.

4. APPLICATION I : MRC-DIRECTED MEMORY ALLOCATION (MRC-MM)

4.1 Main Idea

This section outlines our MRC-based memory allocation scheme for multiprogramming systems. In a multiprogramming environment, the amount of memory demanded by a process is not always equal to the amount allocated to it. When a process is allocated less memory than it needs, each memory reference outside the resident set of the process triggers a page miss (page fault), which involves blocking the process execution for tens of thousands of CPU cycles (for doing slow disk I/O). In addition, each page miss incurred can further decrease the performance of the process by allowing other processes to steal its resident set when it is not executing. This can cause the process to *thrash* and degrade the system performance. To avoid this, efficient memory management assumes importance, especially in host centers that run multiple applications on the same machine to improve system throughput.

The memory manager can use the methods described in Section 3 to determine the MRC, and use the MRC to guide memory allocation. The basic idea of the scheme described in this section is to allocate more memory to processes that exhibit a higher marginal performance gain.

4.1.1 Problem Formalization

To find a good memory management algorithm, we first formalize the problem as a resource utility problem. Consider a computer system running n processes P_i, $0 \leq i \leq n-1$. Let M_{tot} represent the total memory of the system. The problem is to assign memory to the n processes in order to maximize memory utilization. We need to associate with each process P_i, a utility function $U_i : \mathrm{R} \rightarrow \mathrm{R}$ where $U_i(m)$ gives the utility obtained by the process P_i when memory of size $m \geq 0$ is allocated to it. As this is a maximization problem, we represent the utility using hit ratios instead of miss ratios. It is also possible to use the end-performance as the utility function by converting miss ratio gain (loss) to performance loss (gain) based on the process's memory reference rate.

Let M_i represent the memory size allocated to process P_i. Process P_i requires a certain minimum memory size, say M_i^{min}, in order to provide acceptable performance. Each process is also weighted by w_i that describes how important that process is to the system. The total system utility U is the weighted sum of the utilities of individual processes. Then the problem of memory allocation is to find M_i for $0 \leq i \leq n-1$ that maximize the total system utility given by:

$$U = \sum_{i=0}^{n-1} w_i U_i(M_i) \tag{2}$$

An allocation $\mathbf{M} = (M_0, \ldots, M_{n-1})$ is said to be feasible if it satisfies the following two constraints. *(1) Finite Memory:* $\sum_{i=0}^{n-1} M_i \leq M_{tot}$. The sum of memory allocations to the different processes must not exceed the total system memory. *(2) Minimum Memory:* $M_i \geq M_i^{min}$, $0 \leq i \leq n-1$. The minimum memory requirement of every process must be satisfied. Note that we assume that the problem set has at least one feasible solution. $\Sigma_{i=0}^{n-1} M_i^{min} \leq M_{tot}$.

This is essentially a QoS-based resource allocation problem and has been addressed in great detail and higher complexity in previous work [47, 48] that deals with resources for real-time applications. As the optimal solution for this has shown to be NP-hard [47], we give a greedy algorithm to get a good solution for the problem.

4.1.2 A Greedy Algorithm

The main idea of the greedy algorithm is to divide the memory into small units δ and incrementally allocate δ amount of memory to the process which can benefit the most from this memory at each step. In other words, the process that receives this δ memory is the one with the highest gradient of the utility function at its current memory allocation.

Without loss of generality, we assume that the utility functions have already been weighted by their priorities w_i, thus eliminating the need to carry them on. Similarly, we can assume that each process has already been allocated the minimum memory which it needs for an acceptable performance. Thus we take $M_i^{min} = 0$.

The greedy algorithm does the following repeatedly until all memory has been allocated or all processes' memory demands have been satisfied. M_{tot} represents the total memory available.

1. Let $M_i = 0, 0 \leq i \leq n-1$
2. Assume for all i, process P_i has been allocated memory M_i so far. Compute the gradients of the utility curves viz., $U_i'(M_i)$, where $U_i'(M_i)$ is the derivative of U_i at $m = M_i$.
3. Select the process P_j with maximum gradient. Ties can be broken arbitrarily using priority.
4. Allocate δ memory to process P_j. Increase M_j by δ and the decrease free memory M_{free} by δ. Other M_i values remain unchanged.
5. If $M_{free} = 0$, then quit as no more allocations can be made. Otherwise go to step 2.

Benchmark	Execution/Response Time(s)			# Page Misses		
	Orig	MRC -MM	Speed -up	Orig	MRC -MM	Reduction(%)
Apache	0.07	0.03	2.29	474*	301*	36.5
gzip	739	706	1.05	46279	47341	-2.3
Apache	0.10	0.04	2.53	604*	305*	49.5
gcc	325	353	0.92	20175	21506	-6.6
Apache	0.07	0.017	4.04	336*	174*	48.2
LINEAR	381	279	1.37	39369	40108	-1.9
INTR	9.8	2.7	3.61	30317	21115	30.4
LINEAR	462	337	1.37	25114	25193	-0.3
INTR	8.5	1.4	5.86	58159	21454	63.1
gzip	883	797	1.10	50913	52959	-4.0
INTR	5.7	2.3	2.53	35471	15190	57.2
gcc	801	842	0.95	23387	25164	-7.6

Table 2: Overall results of MRC-MM. For Apache and INTR, the time reported is the average response time per request. For LINEAR, gcc, and gzip, the time reported is the execution time. The speedup is calculated by dividing the number in the original system by the corresponding MRC-MM number. Note (*): Number of page faults for Apache is per 100 seconds.

For convex utility functions, the greedy algorithm has shown to be optimal in [46]. For non-convex utilities, the algorithm can be applied to the convex hull of the utility function curve. We note that MRCs are usually convex, and therefore the greedy solution we obtain is very close to the optimal solution.

4.1.3 Implementation of MRC-MM in Linux

We modified the memory manager in Linux to incorporate memory reclamation and memory allocation procedures based on the above greedy algorithm.

In the memory reclamation procedure, when the amount of free memory in the system falls below some threshold, the memory manager removes some memory from processes that have the minimum performance loss. For example, comparing two processes A and B, if taking δ memory from A would not increase A's miss ratio at all whereas taking the same amount of memory from B would cause more misses for B, then process A is chosen to give up δ memory. The memory manager iterates this until the system has enough free memory.

In the memory allocation procedure, when a process commits a page fault, the memory manager checks if it should replace one of that process's own page or get one from the free page pool. The decision is made based on the performance gain the process is likely to have when given more memory. If the performance gain is significant, it is given δ more memory. Otherwise, one of that process's own page is replaced.

4.2 Real System Experiments

We have implemented our OS method for tracking MRC dynamically, as discussed in 3.2, in Linux operating system. We have also modified the memory management scheme to implement the MRC-directed memory allocation scheme discussed in 4.1.3. We did not use the hardware MRC monitor because our simulator does not support an OS. Moreover, the coarse granularity at which the OS implementation tracks the MRC is sufficient for memory allocation, as the following sections will show.

4.2.1 Methodology

We conduct our experiments on a system with a single 1.8 GHz CPU, Pentium 4 machine running Linux-2.4.20, with 128MB memory and 512MB swap space. The page size is 4KB. We use 128MB of memory because our applications' data set sizes are relatively small. We expect our results to be similar if we scale up both the memory size and data set sizes. We use 64 pages as the group size and 10ms as the epoch length for tracking MRC in OS.

We evaluate the modified system with three real applications (Apache Web Server [1], gcc, and gzip) and two synthetic benchmarks. For the Apache Web server, we use the flood [2] to generate requests, and Apache's logging facilities to find out the response time to serve each request. The inputs for gcc and gzip are 166.i and input.graphic respectively, which have been taken from the SPEC2000 benchmark suite. The two synthetic benchmarks are very similar to the ones used by Brown and Mowry in their study of using compiler-inserted releases to manage physical memory [6]. The two synthetic benchmarks are: (1) LINEAR, which emulates an application having a loop access pattern. The program consists of a loop that touches pages in a sequential fashion. (2) INTR, which emulates an interactive application consisting of periods of activity followed by inactivity (we use sleep() to emulate this). The program consists of a loop, each iteration of which touches a set of pages based on zipf distribution [4] to simulate temporal locality. We measure the response time as the time taken to touch the pages in a given loop iteration.

We compare our MRC-MM with the original Linux memory management. The original Linux MM uses the global page replacement (the LRU-like CLOCK algorithm) to manage the entire physical memory space without considering which process a page belongs to. It is simple but suffers from thrashing.

4.2.2 Overall Results

Table 2 shows the execution/response time and number of page faults for each application in MRC-MM and the original system for six multiprogramming settings: (1) Apache with gzip, (2) Apache with gcc, (3) Apache with LINEAR, (4) INTR with LINEAR, (5) INTR with gzip, and (6) INTR with gcc. Since interactive applications are more sensitive to response time, the performance of Apache/INTR is a more important consideration when compared to the execution time of the other three applications. In all the times presented, the overhead imposed by tracking MRC is reflected in the results.

For interactive applications such as Apache and INTR, MRC-MM can reduce response time by a factor of 2.29 to 5.86. For example, with Apache and LINEAR running, the average response time of Apache is reduced by a factor of 4.04 when run in MRC-MM. This is because MRC-MM effectively redistributes memory by giving more memory to applications that would benefit most. As a result, MRC-MM reduces the number of page faults for interactive applications by 36.5-63.1%. Due to the intervention of process scheduling and request queuing delay, the percentage reduction in the average response time in some settings is larger than the percentage reduction in the number of page misses.

Even though MRC-MM favors interactive applications, non-interactive applications (gcc, gzip and LINEAR) suffer only minor (5-8%) performance slowdown with MRC-MM in

the worst case and in some cases even their performance improves. The main reason is that MRC-MM takes away extra memory from these applications only when it does not result in a significant performance degradation. In some cases, MRC-MM can even improve the performance. This is because fewer page faults result in reduced page-ins and hence reduced swap-outs. As a result the swap disk in MRC-MM has less contention and therefore page misses can be served faster.

A detailed analysis of Apache+gcc case is presented in Figure 4. In the experiment, Apache begins first and gcc begins after 140s. Figure 4(a) shows the response time of Apache in the Apache+gcc setting on the original and the MRC-MM system. When gcc begins after 140s, in the original system, the response time of Apache increases dramatically to a maximum of 250ms and remains high till around 300 seconds, when gcc releases the memory after its first pass. When gcc starts its second pass, the Apache response time increases to a high value again. With MRC-MM, when gcc begins, the response time of Apache rises initially. However, MRC-MM finds that giving Apache more memory at the cost of gcc is beneficial for the overall system performance, and the performance of gcc is not greatly degraded by giving it less memory. Thus, the response time of Apache resumes much earlier and more quickly in MRC-MM than in the original system. Similar explanation holds for the second phase of the gcc. Figures 4(b) and (c) show the resident set sizes (RSS) of Apache in the original and MRC-MM systems. In the original system, when gcc starts, the RSS of Apache drops from around 20000 pages to around 8000 pages, while gcc is allotted more than 20000 pages. At around 300 seconds, gcc releases its memory after its first pass. As its second pass starts, it is again allocated around 20000 pages. In contrast, with MRC-MM, the RSS of Apache drops initially, but is soon restored to around 15000 pages. The RSS of gcc is initially around 20000 pages, but soon decreases to around 8000 pages. These results show that MRC-MM is able to reduce the response times of Apache in presence of an out-of-core application. It achieves this by reducing RSS of the out-of-core application to the size necessary for an acceptable performance.

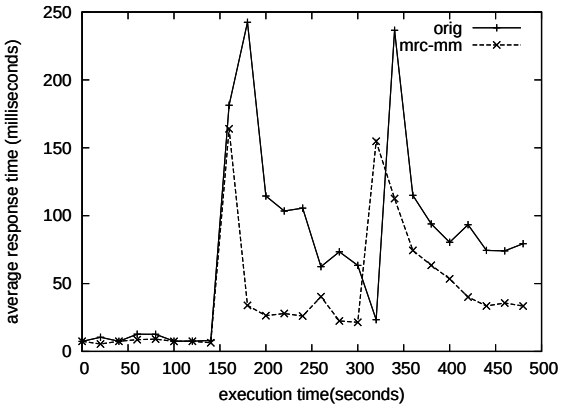
In summary, our results show that MRC-MM is very effective in improving memory utilization for multiprogramming environments with applications of diverse types. As the host center-based computing paradigm of running multiple applications or virtual machines on the same machine becomes predominant, we believe that MRC-MM will be very useful.

4.2.3 Overheads of tracking MRC

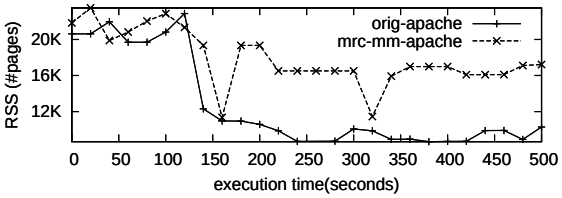
To measure the overhead of our tracking MRC implementation, we compare the application's total execution time in the original Linux system and in the modified Linux system that computes the MRC periodically. While measuring this execution time, we do not perform any memory management, because we want to calculate only the pure tracking overhead, and not the performance gains that will arise from the MRC-MM allocation scheme. As we have seen in results presented in Section 4.2.2 which already include the MRC tracking overhead, the performance gains of MRC-MM outweigh the overhead of MRC computation. Table 3 shows the time overhead for three applications – LINEAR, gcc, and gzip.

Our results show that the pure tracking overheads for these three applications are in the range 7%-10%. This overhead comes from manipulating the extra data structures, scanning the scan list, and handling page protection faults.

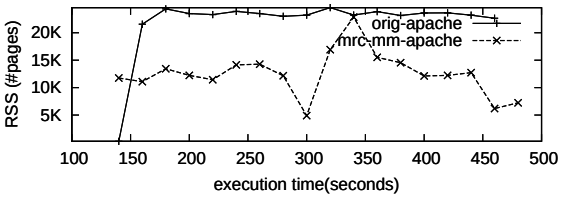
In our experiments, the MRC tracking process is enabled all the time. When used in a real system, MRC tracking code would run only when the system has high memory contention, and it will never be triggered when there is enough memory. Therefore, in a real system deployment, the overhead will be even smaller.



(a) Response time of Apache in Apache+gcc



(b) Resident Set Size of Apache in Apache+gcc



(c) Resident Set Size of gcc in Apache+gcc

Figure 4: Detailed analysis of Apache+gcc.

Application	LINEAR	gcc	gzip
Overhead	7 %	10 %	8 %

Table 3: Tracking overhead as a percentage of the execution time of the original program

5. APPLICATION II : MRC-DIRECTED MEMORY ENERGY MANAGEMENT (MRC-EM)

5.1 Main Idea

As explained in the Introduction, memory is a particularly

Power State/Transition	Power	Time
Active	300mW	60ns
Standby	180mW	-
Nap	30mW	-
Powerdown	3mW	-
Standby → Active	240mW	+6ns
Nap → Active	160mW	+60ns
Powerdown → Active	150mW	+6000ns

Table 4: Power consumption and transition time of different power modes.

important target for efforts to address the energy efficiency issues in both mobile devices and high-end server systems. Several recent studies [38, 15, 16] have explored the power-mode control capabilities of modern RDRAM modules [49] to transition memory modules into a low power operating mode after being idle for a certain threshold value of time. Accessing data stored in a memory module in a low power mode requires the module to be transitioned to the active mode, which can incur non-negligible resynchronization delay [30]. This gives us a way to trade off performance for energy savings. Table 4 shows the energy consumption and resynchronization time for RDRAM, which is the same as the one used in a previous study [38]. Our key idea is to identify and power down the chips that are not being accessed by any application, so that we can save energy without incurring any performance degradation.

Our MRC-directed memory energy management (MRC-EM) assumes hardware MRC monitor since it needs to track MRC at fine time granularity. It enhances the previous schemes by using the application's dynamic MRC to determine the minimum number of active memory chips necessary to deliver acceptable performance. For example, if a process can achieve almost the maximum possible page hit ratio with N_{mrc} memory pages, the memory power control algorithm only needs to adapt the power modes for N_{mrc} pages and power down the rest. Unfortunately, current memory technology only supports power adaptation at chip granularity rather than page granularity. So we need a way to map selected pages to chips, which we henceforth refer to as "working chips". Once working chips have been identified, they can be managed by the underlying memory energy management scheme such as the ones studied in previous works [38, 15, 16], and all other chips can be powered down.

Since MRC only determines the number of working chips, we also need to determine (1) what chips should be selected as working chips, and (2) how to handle an access to a non-working chip. To address the first issue, we take advantage of the working set model. Since our dynamic MRC-tracking method maintains the LRU stack, it is easy to find the pages in the top N_{mrc} elements of the Mattson's stack. Any chip with such a page is selected as a working chip. All the other chips are powered down. The power states of working chips are controlled by the basic energy management scheme (in our experiments, we use both the static and dynamic schemes studied in [38]).

Accessing a page in a non-working chip results in transitioning it to active state, and this chip is then included in the set of working chips. Even though accessing a powered down chip incurs high latency and energy cost, the penalty is amortized over multiple accesses due to the temporal and spatial locality exhibited in most of the programs. In our experiments, we use the sequential page allocation mechanism proposed in [38] to increase the spatial locality at physical address level.

At any time, if a chip does not contain any of the top N_{mrc} pages of the Mattson's stack, MRC-EM deletes it from the set of working chips and powers it down. To keep track of whether a chip contains any of the top N_{mrc} pages in the stack, MRC-EM maintains a counter for each memory chip. Whenever a new page is moved to the top of the stack, the counter of the chip in which this new page resides is incremented by one. The counter for the chip that contains the $(N_{mrc}+1)^{th}$ page in the stack is decremented by one. Whenever a chip's counter reaches zero, it is removed from the set of working chips and is transitioned into the power down mode.

MRC-EM does not increase the number of page faults since we do not change the page replacement policy of the system. MRC-EM can be combined with most previous memory energy management schemes. In our experiments, we combined it with the static and dynamic schemes proposed by Lebeck et al. [38]. Similar to previous works in memory energy management [9, 32, 38, 56, 15, 16], we assume an intelligent memory controller to implement the energy management schemes.

5.2 Experimental Results

We use execution-driven simulation to evaluate MRC-EM. We enhance the widely used SimpleScalar [7] simulator with the MRC monitor and RDRAM power model. We use Wattch [5] to model the energy consumption of the MRC monitor. The *LRUArray*, *GrpHdrs* and *HitCtrs* are modeled as array structures. We use the chip size as the page group size for the MRC monitor, since the power adaption is at chip granularity. The power model of the array structure is parametrized based on the number of rows and columns, and the number of ports. These parameters determine the number of decoders, wordlines, bitlines, and their sizes. The power dissipation is calculated based on these configuration parameters and the number of operations are calculated in a similar way as shown on Table 1 but with smaller memory capacity and smaller page group sizes. For the MRC-EM schemes, the energy consumption of the MRC monitor is also included in the total energy consumption. We do not simulate the MRC Monitor time overhead, because it works in parallel with memory accesses and imposes little overhead.

The default chip size is 64 pages unless stated otherwise in all our simulations. The memory page size is 4KB. We simulated a 600 MHz 4-issue processor with a 4-way set-associative 16K L1 data cache and a direct-mapped 16K L1 instruction cache (both having 32B blocks). The unified L2 cache is 256K with 4-way set associativity and 64B blocks. The processor executes PISA binaries. We report the results with memory sizes similar to the corresponding application's data set size. The default epoch length is 10,000 cycles, except in the experiments that studied the effects of epoch length.

The base-line memory energy management schemes are the static nap and dynamic schemes proposed in [38]. These two are the best static and dynamic schemes reported in that study. In the MRC-EM static scheme, the working chips always stay in nap mode. In the MRC-EM dynamic scheme (MRC-EM-Dynamic), the working chips adapt their power

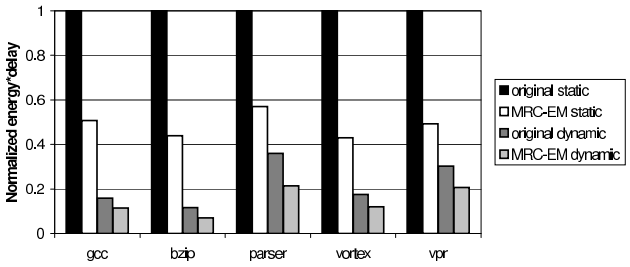
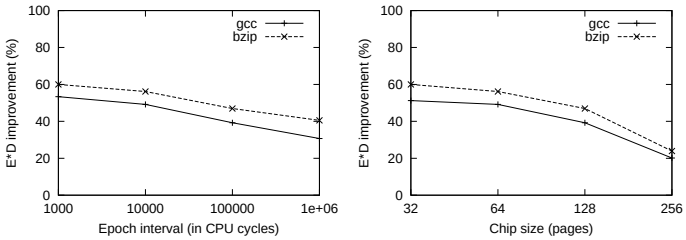


Figure 5: ***The value of the Energy*Delay metric for different schemes.***



(a) Effect of epoch length (b) Effect of memory chip size

Figure 6: ***Effects of parameters (epoch length, chip size)***

mode dynamically based on the original dynamic scheme [38]. For both original and MRC-EM dynamic schemes, we use the *same* threshold setting, namely the found to be the best in [38].

5.2.1 Overall Results

The Energy*Delay (ED) metric is commonly used in most previous studies in memory or processor energy management [9, 32, 38, 56, 15, 16]. Figure 5 compares the normalized ED results among original static, MRC-EM static, original dynamic, and MRC-EM dynamic schemes. Due to space limitation, we do not show the absolute energy consumption and performance delay. These detailed results can be found in [46].

The results show that MRC-EM can significantly reduce the ED over the corresponding original scheme. For example, MRC-EM static has 42-58% smaller ED than original static, and MRC-EM dynamic has 27-42% smaller ED than original dynamic scheme. This indicates that MRC-EM is effective in directing memory energy management. The main reason is that it provides a guideline to the underlying memory energy management scheme on how much memory is needed to deliver acceptable performance.

The relative energy consumption shown in the figure illustrates that the dynamic schemes are better than the static schemes [38], which is in agreement with the findings in [38]. We note that the ED savings by MRC-EM is more prominent in the case of static scheme when compared to the dynamic scheme. This is because the dynamic scheme performs better than the static scheme and therefore the extra gain we obtain by using the MRC-EM with dynamic scheme is less when compared to using it on a relatively naive scheme like the static scheme. However, the dynamic scheme relies on thresholds, and therefore requires cumbersome threshold tuning. The MRC approach is application-independent, does not require threshold tuning, and can be coupled with any suitable fine grained power saving mechanism in order to conserve memory energy.

5.2.2 Effects of Epoch Length

Figure 6(a) shows the variation of improvement in ED of MRC-EM static over original static with epoch lengths for bzip and gcc. As the epoch length increases, the improvement in ED decreases. The reason is that when each epoch is long, the system cannot detect MRC changes in a timely manner. As a result, it may either miss some opportunities to save energy by transitioning unneeded memory chips to low-power modes, or degrade performance by using less than the required number of working chips. In both the cases, we end up getting a higher ED value. In the OS implementation, the epoch length is 10 milliseconds which translates into 6,000,000 CPU cycles, too coarse-grained to be effective for MRC-directed memory energy management. We have also studied the effects of epoch length on MRC-EM-Dynamic and the results are similar.

It should be noted that the epoch length cannot be too small. Otherwise, the overhead of determining inactive/active memory chips and transitioning them at the end of each epoch would dominate and cancel out the benefits of energy saving. We use 10,000 cycles as the default epoch length.

5.2.3 Effect of Memory Chip Size

Figure 6(b) shows the effect of memory chip size on ED improvement of MRC-EM static over original static for bzip and gcc benchmarks. Since we simulated applications that have medium data set sizes (around 2000 pages), we simulate chip sizes that result in number of chips ranging from 8 to 64. We expect the results to be similar for large data sizes and larger chip sizes. As shown in Figure 6(b), the ED reduction by MRC-EM schemes is more significant with smaller chip sizes than with large chip sizes. The reason is that, with smaller chip sizes we gain more resolution in choosing the chips containing the N_{mrc} pages, thereby reducing inaccuracy due to false sharing.

6. RELATED WORK

This section briefly summarizes closely related work in the areas of memory allocation for multiprogramming systems and memory energy management. Due to space limitation, we do not repeat those described in earlier sections.

6.1 Memory Allocation

Research on memory allocation in a multiprogramming system can be grouped into three categories: *global replacements*, *local replacements*, and *demand-based replacements*.

As mentioned in Section 4.2.1, global page replacements manage the entire physical memory space without considering which process a page belongs to. They are simple to implement and thereby commonly used in many modern OSs including FreeBSD and Linux [21, 55]. However they suffer from thrashing [17, 53]. Existing operating systems address thrashing using load controls [17]. Recently, Jiang and Zhang have proposed a Trashing Protection Facility to protect interactive applications from thrashing [31].

Local page replacements are performed inside memory page pool of individual process [23]. Since this scheme can-

not efficiently utilize memory, it is not commonly used except in VMS [35].

Demand-based replacements manage memory based on applications dynamic memory needs. Our MRC-directed method falls in this category. Most previous works use the working set model [20, 18, 19] to guide memory management. Several schemes have been proposed in the past, including the page fault frequency (PFF) algorithm [11, 27, 53] and the variable-interval sampled working set (VSWS) policy [26]. Windows NT/XP/2K uses a demand-based replacement policy by allowing programs to specify working set sizes explicitly [12, 45]. As discussed in the Introduction, an application's working set gives *what* pages should not be replaced, whereas MRC shows *how many* pages should be resident to provide acceptable performance. Therefore, these two models complement each other.

Several studies have also used compiler to direct memory management by estimating the memory requirements of a program at compile time [41, 42] or insert "release" in out-of-core programs [6]. This approach is static and limited by the aliasing problem.

Our work is also related to some virtual memory studies [51, 33, 58] and disk cache study [54]. Our work differs from and also complements these studies in that we focus on *virtual memory systems* where not every memory reference goes through the OS, and provide two *methods (implementations)*: hardware-only and OS-only, to dynamically track MRC for virtual memory systems. In addition, we also use MRC for memory power management.

6.2 Memory Energy Management

A lot of research has been done in memory energy management. Lebeck et al. have explored the interaction of page placement with static and dynamic hardware policies [38]. Their simulation results show that power aware page allocation by an informed operating system coupled with dynamic hardware policies can dramatically improve energy efficiency of main memory. Fan et al. have investigated DRAM memory controller policies for determining chip power states based on the estimated idle time between access [25]. Delaluz et al. have studied compiler-directed techniques [14, 15] as well as operating-system-based approaches [13, 16] to reduce the energy consumed by the memory subsystem. Recently, Huang et al. [30] have proposed power-aware virtual memory implementation in OS to reduce memory energy consumption. Our work complements previous works by determining how much memory should be used to provide similar level of performance.

7. CONCLUSIONS

An application's miss ratio curve (MRC) contains valuable information that can be used for memory management. However, tracking MRC dynamically at run time is very challenging because not every memory access passes through the OS. This paper addresses this challenge by proposing two methods, a hardware method and an OS-only method, to track an applications' miss ratio curve (MRC) dynamically at run time. The hardware method can track MRC at a fine time granularity, whereas the OS-only method tracks MRC at a coarse time granularity.

This study also applies MRC to two applications: (1) MRC-directed memory allocation (MRC-MM) for multiprogramming systems and (2) MRC-directed memory energy management (MRC-EM). Our *real system* experiments on Linux with applications including Apache Web Server show that the MRC-MM can speed up applications by up to a factor of 5.86 and reduce the number of page faults by up to 63.1%. Our execution-driven simulation results with SPEC2000 benchmarks show that the MRC-EM can improve the Energy*Delay metric by 27-58% over previously proposed static and dynamic schemes.

Our study has several limitations that we plan to address in the future. The MRC-directed memory allocation scheme only considers memory utilization as the maximization objective. It can be easily extended to consider process priority and other attributes to ensure fairness. We are in the process of extending the MRC-directed memory manager to consider end-performance rather than page hit ratio for utility gain. For a given process, the end-performance gain/loss can be estimated from page hit ratio gain/loss based on the process's memory reference rate. We expect the results to be similar to those presented in this paper for most applications. Finally, we have performed experiments only in a few multiprogramming settings. We are in the process of evaluating more applications, especially for MRC-directed energy management. We are also exploring other applications for MRC, such as phase tracking and phase prediction of programs.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for useful feedback, the UIUC Opera groups for useful discussions, Paul Sack and Radu Teodorescu for proofreading the paper, and Professor Alvin R. Lebeck's research group for providing the RDRAM simulator.

9. REFERENCES

[1] Apache web server. http://httpd.apache.org/test/flood.
[2] Flood: A profile driven http load tester. http://httpd.apache.org/test/flood.
[3] G. A. Abandah and E. S. Davidson. Configuration independent analysis for characterizing shared-memory applications. In *IPPS-12*, Mar 1998.
[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM (1)*, pages 126–134, 1999.
[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA-27*, June 2000.
[6] A. D. Brown and T. C. Mowry. Taming the memory hogs: Using compiler-inserted releases to manage physical memory intelligently. In *OSDI*, Oct 2000.
[7] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, Univ. of Wisconsin-Madison, 1996.
[8] R. Carr and J. Hennessy. Wsclock — a simple and efficient algorithm for virtual memory management. In *SOSP*, Dec. 1981.
[9] F. Catthoor, S.Wuytack, E. Greef, F.Balasa, L.Nachtergaele, and A. Vandecappelle. Custom memory management methodology exploration of memory organization for embedded multimedia systems design. In *Kluwer Academic Publishers*, 1998.
[10] J. S. Chase, D. C. Anderson, P. N. Thakar, A. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centres. In *SOSP*, pages 103–116, 2001.
[11] W. Chu and H. Opderbeck. The page fault frequency replacement algorithm. In *AFIPS Conf. Proc.*, 1972.

[12] H. Custer. *Inside Windows NT*. Microsoft Press, Redmond, Washington, 2000.

[13] V. Delaluz, M. Kandemir, and I. Kolcu. Automatic data migration for reducing energy consumption in multi-bank memory systems. In *the 39th Design Automation Conference*, June 2002.

[14] V. Delaluz, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Energy-oriented compiler optimizations for partitioned memory architectures. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Nov 2000.

[15] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramniam, and M. J. Irwin. Hardware and software techniques for controlling dram power modes. *IEEE Transactions on Computers*, 2001.

[16] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Scheduler-based dram energy management. In *Proceedings of the 39th conference on Design automation*, pages 697–702. ACM Press, 2002.

[17] P. Denning. Thrashing: Its causes and prevention. In *AFIPS Fall Joint Computer Conference*, 1968.

[18] P. J. Denning. Memory allocation in multiprogrammed computers. In *Project MAC Computation Structures Group Memo*, Mar 1966.

[19] P. J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, May 1968.

[20] J. B. Dennis. Program structure in a multi-access computer. Project MAC Tech Rep. MAC-TR-11, M.I.T, 1967.

[21] M. Dillon. Design elements of the freebsd vm system. Daemon News, Jan 2001.

[22] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[23] J. E. G. Coffman and J. Thomas A. Ryan. A study of storage partitioning using a mathematical model of locality. In *Proceedings of the third symposium on Operating systems principles*, page 122, 1971.

[24] R. L. M. et al. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[25] X. Fan, C. S. Ellis, and A. R. Lebeck. Memory controller policies for dram power management. In *ISLPED*, Apr 2001.

[26] D. Ferrari and Y.-Y. Yih. VSWS: The variable- interval sampled working set policy. *IEEE Trans. on Software Engineering*, SE-9, 1993.

[27] R. K. Gupta and M. A. Franklin. Working set and page fault frequency replacement algorithms: A performance comparison. *IEEE Transactions on Computers*, C-27, 1978.

[28] M. D. Hill. Aspects of cache memory and instruction buffer performance. Technical Report CSD-87-381, University of California, Berkeley, Nov. 1987.

[29] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12), 1989.

[30] H. Huang, P.Padmanabhan, and K. Shin. Design and implementation of power-aware virtual memory. In *USENIX*, 2003.

[31] S. Jiang and X. Zhang. Tpf: A system thrashing protection facility. *Software: Practice and Experience*, 32:295–318, 2002.

[32] M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Design Automation Conference*, 2000.

[33] S. F. Kaplan, L. A. McGeoch, and M. F. Cole. Adaptive caching for demand prepaging. In *Proceedings of the third international symposium on Memory management*, 2002.

[34] M. Karlsson and P. Stenstrom. An analytical model of the working-set sizes in decision-support systems. In *SIGMETRICS*, pages 275–285, 2000.

[35] L. J. Kenah and S. F. Bate. *Vax/VMS Internals and Data Structures*. Digital Press, Bedford, 1984.

[36] J. Kim, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. OSDI, 2000.

[37] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. In *SIGMETRICS*, 1991.

[38] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *ASPLOS*, pages 105–116, 2000.

[39] C. Lefurgy, K. Rajamani, F. Rawson, W. F. elter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *IEEE Computer*, 36(12):39–48, December 2003.

[40] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume II Software, pages 94–101, Aug. 1988.

[41] M. Malkawi and J. H. Patel. Compiler directed memory management policy for numerical programs. In *SOSP*, 1985.

[42] M. Malkawi and J. H. Patel. Performance measurement of paging behavior in multiprogramming systems. In *ISCA*, 1986.

[43] B. Moore. Taking the data center power and cooling challenge. Energy User News, August 27th, 2002.

[44] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *the 15th SOSP*, 1995.

[45] S. P. Prasad Dabak, Milind Borate. *Undocumented Windows NT*. M&T Books, 1999.

[46] A. Raghuraman. Miss-ratio curve directed memory management for high performance and low energy. UIUC, Master Thesis, 2003.

[47] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for qos management. In *IEEE Real-Time Systems Symposium*, 1997.

[48] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. Practical solutions for qos-based resource allocation problems. In *IEEE Real-Time Systems Symposium*, 1998.

[49] Rambus. Rdram. http://www.rambus.com, 1999.

[50] E. Rothberg, J. P. Singh, and A. Gupta. Working sets, cache sizes and node granularity issues for large-scale multiprocessors. In *ISCA*, 1993.

[51] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: simple and effective adaptive page replacement. In *SIGMETRICS*, 1999.

[52] B. J. Smith. A pipelined, shared resource MIMD computer. In *Proceedings of International Conference on Parallel Proc essing*, pages 6–8, 1978.

[53] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, New Jersey, 1992.

[54] D. Thiebaut, H. S. Stone, and J. L. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Trans. Comput.*, 41(6):665–676, 1992.

[55] R. van Riel. Page replacement in linux 2.4 memory management. USENIX Annual Technical Conference - FREENIX Track, 2001.

[56] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. In *ISCA-27*, pages 95–106. ACM Press, 2000.

[57] W.-H. Wang and J.-L. Baer. Efficient trace-driven simulation method for cache performance analysis. In *SIGMETRICS*, 1990.

[58] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *USENIX*, 1999.

[59] S. Woo, M. Ohara, E. Torrie, J.P.Singh, and A. Gupta. Methodological considerations and characterization of the splash-2 parallel application suite. In *ISCA-23*, May 1996.