MemRx: "What-If" Performance Prediction for Varying Memory Size

Stephen T. Jones
Andrea Arpaci-Dusseau
Remzi Arpaci-Dusseau

Technical Report #1573

August 2006

MemRx: “What-If” Performance Prediction for Varying Memory Size

Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
University of Wisconsin, Madison

Abstract

Understanding and managing complex computer systems is quickly becoming intractable for an unaided administrator. Questions about how to provision server and distributed systems or how workload changes will affect system performance are often hampered by the lack of a clear understanding of how a workload behaves under various system configurations. In this paper we describe and evaluate MemRx, an operating system extension designed to allow an administrator or other systems management agent to answer what-if questions about a workload's runtime when one important system parameter, main memory size, is increased. Our evaluation of a prototype implementation of MemRx in the Linux kernel shows that it can consistently predict the runtime of a suite of microbenchmark and application workloads to within 10% of their actual value as memory size increases. The runtime overhead imposed by MemRx is small enough (less that 6% in a worst case scenario) to allow the extension to run continuously.

1 Introduction

Computer systems are complex, interacting collections of hardware, system software, and applications. It is not uncommon for a single hardware platform to host many distinct services, each of which supports hundreds or thousands of local and remote clients. A typical enterprise consists of many such communicating and interdependent systems. In the future, the trend of increasing complexity can be expected to continue.

System complexity makes it difficult to intelligently provision new systems or adapt existing systems to changing workloads. It is difficult to predict how a new, differently configured system will perform on an existing workload because so many factors contribute to workload performance.

It is also difficult to make workload management decisions. For example, how should jobs or services be assigned to hosts in a distributed system? Once assigned, how should jobs or services be migrated for balanced load and good performance?

This paper focuses on developing and evaluating techniques to answer what-if [3, 19] questions about the performance effect of changing one important system parameter, namely the size of physical memory. The amount of system memory has a huge impact, both on workload performance and on system cost. Hence, sizing memory properly is crucial. Too little memory leads to unacceptable performance, while too much memory leaves expensive resources underutilized and wastes money.

Existing techniques [21, 23], can be used to determine when memory is being *underutilized*, that is when the sum of demand does not exceed the total physical memory size. Determining how much additional memory a system can profitably exploit when all currently installed memory is in active use, in other words, when memory is *oversubscribed*, is still an open question. This paper develops techniques to fill that void.

In addition to knowing how much memory a workload can profitably use, it is also useful to quantify the performance effect of additional memory, for example, to know the marginal reduction in workload runtime given each feasible memory increment. This additional information allows an administrator or resource manager to quantitatively judge the value of assigning additional memory to a workload based on the concrete, real-world metric of runtime. Existing working set measurement and miss-ratio curve research does not provide this ability for the important case when memory is scarce.

MemRx is an extension to the operating system's page cache that uses measurement and a model of runtime components to predict the marginal performance benefit that a workload would experience if memory were added to the system. MemRx enables an administrator or management software to answer what-if questions about how the runtime of a workload changes as memory size increases.

MemRx does this by carefully tracking I/O requests related to capacity misses in the page cache. MemRx has two key components: a performance prediction module (PPM) and a virtual extended cache (VEC). The PPM dynamically measures the performance impact of each capacity miss, a critical feature that allows MemRx to predict runtime under different memory configurations. The VEC uses a ghost-buffering technique [16] to produce miss-ratio curves, allowing it to associate the penalties produced by the PPM with a memory increment. The PPM and VEC allow MemRx to predict performance across a range of memory configurations.

MemRx can be applied to resource management problems in multiple domains. For example, a systems administrator can use the output of MemRx to quantitatively evaluate the trade-off between the cost of installing additional memory and the benefit of improved performance leading to more intelligent procurement and configuration decisions. A MemRx-like component could also be deployed in batch-computing environments [12], sometimes referred to as "computational grids" [7] or "utility computing centers" [22]. In these environments, non-interactive, computationally intensive jobs are executed on a large collection of clustered hosts. Such systems attempt to avoid workload thrashing using resource hints provided by users. Unfortunately, these hints are often wrong or omitted altogether. Using MemRx, a batch scheduler can independently detect a poorly performing job and estimate how much additional memory is needed by the job. If the benefit of additional memory predicted by MemRx outweighs the cost of migration, the scheduler can then intelligently *migrate* [6, 11, 15, 17] the job to another machine with adequate memory.

We have built a MemRx prototype within the Linux kernel that is consistently able to predict the performance of a suite of microbenchmark and application workloads to within 10% of their actual values as memory size increases. In the common case when the system is not thrashing, MemRx induces no perceivable overhead. When the system is thrashing, (and hence already performing poorly), MemRx incurs a small, but measurable overhead of roughly 6%.

In the process of designing and implementing MemRx we were able to make two observations that have important implications on the design of introspective systems. The first is that detailed categorical or "end-to-end" [19] information that describes how and why events have occurred in the system is critical for an introspective service like MemRx. The second observation is that in some cases simple changes can be made to a system to make it more predictable, which significantly increases the ability of a system to support accurate what-if queries. Our experience shows that these changes do not necessarily have to hurt performance, but can enhance it by reducing the effects of difficult to predict pathological behaviors.

The rest of this paper is structured as follows. We first place MemRx in context with related work in Section 2. Then we discuss the design of MemRx in Section 3, followed by its implementation details in Section 4. Section 5 evaluates the accuracy and overhead of MemRx for a variety of synthetic and application workloads. We discuss related work in Section 2 and conclude in Section 6.

2 Related Work

Working set estimation has been a part of computer systems for many years [4, 5]. As Denning said in his classic paper on the topic, "the operating system must determine on its own the behavior of the programs it runs" [4]. Operating systems researchers have long understood this, and have developed algorithms to quantify the memory behavior of programs that *fit within main memory*.

An excellent recent example of such a technique is found in Zhou *et al.*'s paper on miss ratio curves (MRC) [23]. In that work, the authors show how to estimate the working set size of a process so as to perform more effective memory allocation among competing processes, and also for the novel application of shutting off unused memory to conserve power. MemRx extends this work by focusing on a domain it does not address, namely systems for which the aggregate memory demand is greater than available physical memory. In addition, MemRx supplies performance predictions in terms of workload runtime rather than the more abstract miss-ratio curve, easing its interpretation and application.

The techniques used by one component of MemRx, the VEC, have been used in other research. For example, Patterson *et al.* use a similar technique to estimate the cache hit ratio for various filesystem buffer cache sizes [16]. This estimate is used as part of a larger framework to evaluate the trade-off between using memory for aggressive, hint-guided pre-fetching and LRU caching of recently used memory. The technique has not, however, been used to estimate the runtime benefits of adding physical memory. Further, the authors of that study do not dynamically measure actual miss costs as the PPM does, instead choosing a single value that is obtained by running various workloads and finding a suitable "average" value. MemRx shows that accurate miss cost estimates are the key to producing accurate performance predictions.

MemRx can be seen as part of a larger on-going effort in the community towards creating introspective and self-managing or "autonomic" [10] computing systems. Such research seeks to reduce the total cost of ownership through automated management. The Self-* project [8, 19], for instance, is working towards an architecture for large scale, self-managing storage infrastructures. Their work endorses pervasive, end-to-end instrumentation and a multi-layer management architecture that allows their system to answer what-if performance and availability questions based on well-defined system policies. MemRx contributes an implemented and evaluated example of a system introspection technique that could be incorporated into such a self-managing architecture.

3 The Design of MemRx

The goal of MemRx is to enable a systems management agent like an administrator or management software platform like a grid-computing job scheduler, to pose and answer what-if questions about the runtime of a workload under various hypothetical, larger memory sizes. Given the workload's current runtime, this task can be accomplished by calculating the runtime *benefit* of additional memory.

3.1 Benefits of Additional Memory

Additional memory can act as a larger cache for previously accessed data and can therefore reduce the number of *capacity misses* experienced by a workload. A capacity miss occurs when data is accessed that was in physical memory at some time in the past, but was evicted due to memory pressure. A capacity miss causes the referenced data to be reloaded from disk back into physical memory. In this paper we use the term *reload* interchangeably with capacity miss. While reloaded data is being retrieved, the workload often stalls, leading to longer runtimes. Hence, the primary benefit of avoiding capacity misses is eliminating unnecessary wait time.

A workload can benefit from additional memory in other ways as well, for example by allowing more aggressive prefetching to avoid compulsory misses, but MemRx does not track these benefits. MemRx focuses only on benefits gained by avoiding capacity misses. The reason is that all workloads can benefit from reducing capacity misses without modification while running on existing systems. Other benefits of additional memory require application modifications, such as application hinting to enable aggressive prefetching [16], or require an application to monitor available free memory and adapt, a feature most applications lack. Limiting MemRx in this way simplifies its design and reduces its overhead without unduly limiting its potential application.

Our treatment of the design of MemRx is broken into a discussion of its two major components. We first describe how to measure the runtime penalty of capacity misses. This task is accomplished by the PPM. Then we explore how to associate individual capacity misses with a particular memory size increment, which is the task of the VEC. The information produced by the PPM and the VEC, when combined, allows MemRx to achieve it's goal.

3.2 The Performance Prediction Model

The PPM measures the runtime impact of capacity misses experienced by a workload. This information is then used by MemRx to predict runtime when some or all of these misses are eliminated due to additional available memory. The $Runtime$ of a workload can be decomposed into several components.

- C = The semantically required computation
- W = The semantically required I/O wait time (including compulsory misses)
- C_{miss} = The unnecessary computation imposed by capacity misses in the page cache (e.g., additional page faults and management of the page cache)
- W_{miss} = The unnecessary I/O wait time imposed by capacity misses

MemRx models workload runtime using the simple additive model $Runtime = C + W + C_{miss} + W_{miss}$.

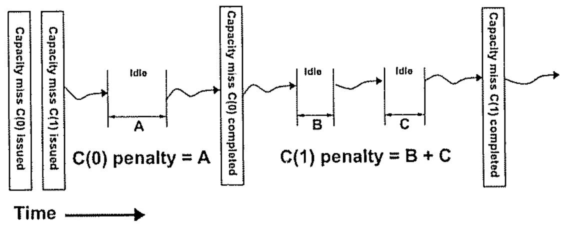
MemRx calculates what the performance of a workload would be if some or all capacity misses were avoided (due to a larger memory configuration) by subtracting W_{miss} from the total $Runtime$ to yield a predicted runtime experienced under a more liberal memory configuration. C_{miss} is ignored since the unnecessary I/O wait time, W_{miss}, typically dominates the computational overhead of capacity misses.

To isolate W_{miss}, MemRx must do two things; 1) identify each reload, and 2) measure each reload's impact on the progress of the workload.

Identifying reloads is straightforward. The operating system is aware of each page evicted from the page cache due to memory pressure and where the evicted data resides on disk. MemRx keeps a registry of all evicted pages along with their on-disk location. By referring to the registry on each disk block read, the PPM can identify reads of previously evicted data.

The PPM characterizes the performance impact of each capacity miss experienced by a system. Other research [16], has typically accomplished this by employing an average miss cost based on storage system specifications or calibration measurements. Early versions of MemRx used this approach as well, but we found that it is not suitable for performance prediction for two important reasons.

The first reason is that the distribution of service times for data access is very broad, ranging from microseconds to seconds depending on the cache state of the system and device load. Because MemRx associates individual penalties with memory increments using the VEC, it is important to characterize the penalty induced by each miss as accurately as possible. Even breaking service times into rough categories like "sequential" and "random" with appropriate measured average service times results in poor performance predictions. For this reason, MemRx measures actual, individual miss penalties experienced by a workload.



Capacity miss C(0) issued
Capacity miss C(1) issued
Idle
A
C(0) penalty = A
Capacity miss C(0) completed
Idle
B
Idle
C
C(1) penalty = B + C
Capacity miss C(1) completed
Time

Figure 1: **Penalty Assignment.** *The diagram shows how idle times are accumulated and assigned as runtime penalties to capacity misses.*

The second complicating factor is the ability of multi-threaded workloads to hide some or all of the latency of I/O requests by overlapping those requests with useful computation. Using an average or even measuring the time between I/O request submission and completion for each reload overestimates the true penalty by not accounting for I/O overlapped by computation. The PPM must instead determine the amount of time a workload is prevented from making progress because it is waiting for a reload to complete.

The PPM does this by keeping a counter, `reload_count`, of outstanding reload I/O requests. When a system is idle and $reload\_count > 0$, the system could make progress if it were not for the blocked reload I/O requests. The PPM records the length of each idle period for which $reload\_count > 0$. The sum of all such idle times represents the total time spent unproductively waiting for reloads to complete by the workload. The PPM computes the runtime penalty for an individual capacity miss by accumulating idle time between reload I/O completions. A reload is assigned a penalty consisting of all capacity-miss induced idle time experienced by the system between the last reload completion and its own completion. Figure 1 depicts the assignment of idle times to capacity misses. At the left of the figure, capacity misses $C(0)$ and $C(1)$ are outstanding. $C(0)$ then completes, resetting the cumulative idle time to zero. Between the completion of $C(0)$ and the completion of $C(1)$ the system experiences two idle times. The sum of these two idle periods is the miss penalty assigned to miss $C(1)$.

In addition to handling multi-threaded I/O, the techniques used by the PPM also naturally take filesystem prefetching into account. The PPM only increments the reload counter for an explicit capacity miss that will cause a thread to block. Prefetch I/O is asynchronous with respect to the foreground thread and does not cause the foreground thread to block. If, however, a prefetch I/O is submitted and is not complete by the time an explicit synchronous request for that block is made, the PPM does count this towards the reload penalty. The PPM model implicitly assumes that prefetch behavior does not depend on the amount of available memory.

Because the PPM measures the actual wait time imposed by capacity misses, device utilization is also naturally accounted for without complicating the simple additive model. If the device servicing capacity-miss I/O has a high utilization this will increase the service times for all I/O and will be included in the measured penalty.

3.3 The Virtual Extended Cache

We now turn our attention to the second component of MemRx, the VEC. Its job is to associate a capacity miss to a particular memory size. This allows MemRx to associate a runtime penalty (calculated by the PPM) with a particular memory increment. The memory size with which a capacity miss is associated is the smallest amount of memory that would have prevented it from occurring, (*i.e.*, the memory increment that would have prevented the relevant data from being evicted from memory in the first place).

The VEC does this by modeling the page cache behavior of the operating system as if more memory were available. To do this the VEC monitors page cache evictions and promotions. When a page is evicted, a reference to the page's location on disk is inserted at the head of a queue maintained in LRU order by the VEC. Subsequent evictions push previous references deeper in the queue.

When a previously evicted page is read from disk, *i.e.*, promoted into the page cache, the reference to that page is removed from the queue and its distance (D) from the head of the queue is computed. The distance is approximately equal to the number of evictions that have taken place between that page's eviction and its subsequent reload. MemRx then uses (D) to compute the amount of memory that would have been required to prevent the original evictions from taking place as $size_{page} \times (D+1)$.

For example, if a page is evicted and immediately reloaded before any other pages are evicted, MemRx would record that the eviction could have been prevented by one additional page of physical memory. If a page's eviction is followed by 1024 evictions of 4 KB pages, MemRx would report that $(1024 + 1) \times 4 \ KB$ (roughly 4 MB) of additional memory would be required to prevent the original eviction.

Our general strategy, which is similar to Patterson *et al.*'s ghost buffering scheme [16], relies upon certain properties of the operating system cache replacement policy to function correctly. Specifically, the algorithm used must (roughly) preserve the *inclusion* or *stack* property [14].

The key aspect of the stack property is that a cache of a size $N + 1$ has the same contents as a cache of size N, plus the one additional buffer which has some other block within it. LRU and LFU obey this property; FIFO does not [2]. By assuming the stack property holds, the VEC can efficiently *simulate* the contents of larger caches, safe in the knowledge that the buffers of the main page cache would be comprised of the same contents even if more memory were available.

Neither Linux, nor most other operating systems, employ a page replacement strategy that perfectly maintains the stack property. Our evaluation in Section 5, however, demonstrates that MemRx is quite robust to these deviations under Linux for many useful cases.

3.4 Combining the PPM and VEC

The information from the PPM and the VEC are combined to allow MemRx to predict the runtime of a workload under various amounts of additional memory.

For each reload L_i, the PPM calculates the miss penalty. The VEC associates this penalty with a memory increment by consulting its page cache model for I/O request L_i. The penalties associated with each simulated memory increment are tallied resulting in an array $Penalty$ indexed by memory increment size. The runtime of a workload executed under a configuration with X MB of additional memory is calculated using the following equation.

$$Runtime_{+X} = Runtime_{current} - \sum_{i=0\ MB}^{X\ MB} Penalty[i]$$

That is, the runtime of the system, when X MB of memory is added is the total measured runtime minus the cumulative penalty that would be prevented by the additional X MB of memory.

3.5 Limitations

One potential limitation of MemRx is that it does not provide information about workloads that have the maximum amount of memory they could ever use. Such information is useful and other researchers [21, 23] have developed techniques to measure working set size and behavior when memory is not scarce. MemRx is specifically designed to address the other important performance regime of workloads forced to execute with less memory than they can profitably use.

A second concern is that our approach assumes that an LRU-like replacement algorithm is used by the operating system to manage the page cache. While most modern operating system replacement algorithms are based on LRU, virtually none are pure LRU. Therefore, one question we implicitly address in Section 5 by evaluating the performance of MemRx on one such system, Linux, is whether the lack of a strict LRU-managed page cache affects results. The evaluation bolsters our confidence that the assumption that the page cache is managed in an LRU-like manner is a reasonable one; indeed, most replacement algorithms strive to approximate LRU in most cases, diverging only to avoid worst-case LRU behavior under looping sequential access patterns [9, 18].

4 The Implementation of MemRx

We now describe our prototype MemRx implementation within the Linux 2.4 kernel. Our first and most important goal in implementing MemRx was to ensure low overhead. Overhead for MemRx has two axes: space and time. Excessive time or space overhead will make MemRx impractical to deploy in real systems, hence both types of overhead should be low enough to enable MemRx to run at all times in even modestly-configured systems.

4.1 PPM Implementation

The PPM requires notification of both evictions (*i.e.*, blocks that are thrown out of the page cache due to memory pressure), and reloads (*i.e.*, disk read operations that retrieve previously evicted blocks back into the page cache). A few key locations in the Linux memory management system and generic block device handling code are instrumented to collect this information.

4.1.1 Tracking Eviction

Linux version 2.4 uses a unified page cache for cached file blocks as well as virtual memory pages. This implies that all evictions, whether of cached disk pages or anonymous virtual memory pages, are eventually handled by a single code path. The task of detecting evictions in Linux is therefore straightforward. All page cache evictions are handled by the routine `shrink_cache` in `mm/vmscan.c`. At eviction time the disk location of each evicted data page is known. For pages residing in a filesystem file, the location is represented as the triple `(device_identifier, inode_number, page_offset)`. For virtual memory pages destined for a swap area, the disk location is encoded into a 32-bit swap location that identifies one of the potential swap areas and the offset of the swap page in that area. The PPM uses this disk location information as a lookup key to uniquely identify evicted data and match it against later attempts to reload the same data block from disk.

4.1.2 Tracking Reloads

Initiation of a reload is detected by instrumenting the code paths by which data may enter memory from disk. For swap area blocks, the instrumentation point occurs in the routine `do_swap_page` in the file mm/memory.c. For file system pages, the routines do_generic_file_read, and filemap_nopage in mm/filemap.c, suffice to capture all explicit read operations required by the PPM. Implicit reads that occur as a result of readahead are ignored. These references do not cause the system to block. The specified routines use the same disk location information as that stored by the PPM on eviction and so can be used to search for matches to previously evicted data.

The PPM additionally instruments the I/O completion routines that the Linux block device subsystem invokes whenever a submitted I/O completes. These routines include end_buffer_io_sync, end_buffer_io_async, and end_buffer_io_kiobuf in fs/buffer.c, journal_end_buffer_io_sync in fs/jbd/commit.c, and finally bounce_end_io in mm/highmem.c. By monitoring these routines, the PPM is aware when each reload I/O completes.

The PPM must also measure the time a system spends idle because it is blocked by the I/O from at least one reload. To do this it instruments the function schedule in the file kernel/sched.c to detect when the system becomes idle or wakes from idle.

Finally, the PPM maintains a set of accumulators, one per VEC sub-queue, to track the runtime penalty of reloads associated with each of the memory segments simulated by the VEC (discussed in Section 4.2). Each time a miss occurs in the current memory configuration, the PPM adds the measured I/O wait-penalty to the accumulator associated with the memory increment that would have turned the miss into a hit in a larger cache.

4.2 VEC Implementation

The LRU queue is the heart of the VEC. It is functionally equivalent to that described in Section 3, but is structured to make common operations fast. For example, since the queue is searched on every disk operation, each entry is entered into a hash table when it is added to the queue.

The depth of an entry in the queue is efficiently approximated by organizing the queue proper into a set of sub-queues as shown in Figure 2. Each sub-queue represents an increment of memory that equals the size of a page multiplied by the number of entries in the sub-queue. For our prototype implementation, all sub-queues represent 32 MB memory increments. Each queue entry keeps track of the sub-queue in which it currently appears, making its depth in the queue (up to the granularity provided by the queue length) immediately available. Hence, in our prototype, a penalty can be associated with each 32 MB memory increment and performance prediction occurs on these configuration boundaries. This sub-queue approach avoids a costly walk of the queue on reload to determine

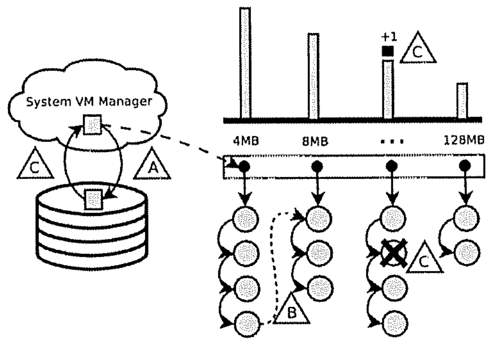


Figure 2: **The VEC in Operation.** *A) When a page is evicted by the system, an entry is added to the head of a series of VEC queues. B) If necessary, queue entries ripple from the tail of one queue to the head of the next. C) Upon reload, the associated queue entry is removed and the array entry associated with that queue is incremented by the I/O wait-penalty calculated by the PPM.*

the depth of an entry.

When a sub-queue exceeds its maximum length, the tail entry is moved to the head of the next deeper sub-queue. This action can ripple the entire length of the queue. When the last sub-queue overflows, that queue entry is discarded and subsequent reloads of that page will not be detected. Hence, the size of the VEC determines what the largest emulated memory configuration is. The implementation of the VEC used to evaluate MemRx in Section 5 models 1 GB of additional memory.

4.3 Categorical Information

While designing and implementing MemRx it became clear that, in addition to counts, interval measurements, and event notifications, an introspective OS service requires categorical information about the activities it observes. This point agrees with very recent qualitative work by Thereska *et al.* [19] which advocates "end-to-end" OS instrumentation.

For example, MemRx must be able to distinguish reloads from normal I/O requests. It is also important to be able to differentiate explicit, synchronous I/O requests from prefetch requests. This kind of categorical information is available at the higher layers of the operating system, but is typically discarded as a request is passed downward through the software stack because it is not strictly necessary to fulfill the request. At the level where interval measurements are made by MemRx, (*e.g.*, in the generic block submission and completion routines), there is no remaining indication of the reason for or the type of the I/O request being issued. MemRx works around this limitation by changing the system to annotate I/O requests with additional information, by keeping additional data structures, or by inferring the type of request from its context. One practical outcome of the MemRx implementation effort is evidence that designers of future, introspective systems should consider annotating requests that cross system layer boundaries with categorical or end-to-end information to enhance the ability of services like MemRx to understand and properly process the event streams they observe.

5 Evaluation

In this section we empirically evaluate our MemRx prototype. All experiments reported here were run using a PC configured with a 2.4 GHz Intel Pentium IV processor, 512 MB of physical memory, and a Western Digital WD1200BB 120 GB ATA disk drive. The base operating system kernel is Linux version 2.4.30 with small modifications to support MemRx as described in Section 4.

In cases where we wish to artificially limit the amount of memory available to the OS, (*e.g.*, to induce thrashing with reasonably sized workloads), we utilize a simple in-kernel balloon [21] that allocates the appropriate amount of memory to reduce the available memory to the desired level. This technique was compared to the behavior produced by limiting available memory using the Linux kernel command line option `mem=<size>` and the results are comparable.

In the following sections, we first evaluate the accuracy with which MemRx can predict runtime for several microbenchmarks and application workloads. We then examine the time and space overheads incurred by MemRx.

5.1 Accuracy

To evaluate the prediction accuracy of MemRx for a given workload, we execute the workload under the MemRx-enhanced kernel. MemRx can then predict the runtime of the workload under larger memory sizes in increments of the VEC sub-queue size, (in this case 32 MB). To judge the accuracy of the predictions, we re-run the workload at each of the memory increments for which performance was predicted by MemRx and compare predicted to actual runtime at each memory size configuration.

5.1.1 Microbenchmark Workloads

We use a variety of microbenchmarks and application workloads to evaluate MemRx. The purpose of the microbenchmarks is to evaluate the performance of MemRx in an environment where the behavior of the program under test is somewhat predictable. Table 1 lists each of the microbenchmarks and the actions they perform.

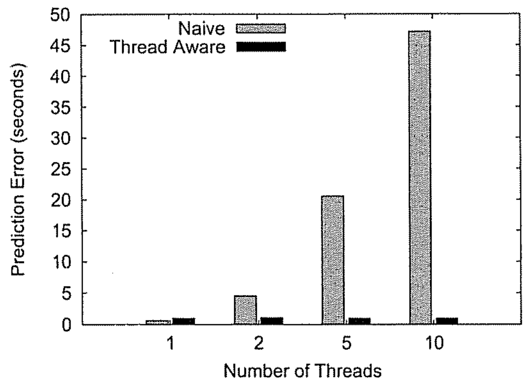


Figure 4: **Thread Awareness in the PPM.** *The figure compares prediction error incurred by two versions of the PPM. The first, labeled "naive" does not take multi-threading into account, the other version is thread aware. The graph shows how runtime prediction error increases with the number of worker threads for the naive PPM, where the error remains constant for the thread-aware version.*

Each benchmark is initially executed with 128 MB of available memory. The workloads are configured to require a working set of approximately 256 MB. For example, FS Sequential sequentially reads 256 MB of a 1 GB file 10 times. FS Random randomly reads 512 MB from the same 256 MB section of a 1 GB file. The workload size for the VM workloads is the same as for the FS workloads. The VM workloads write each page they touch to force page allocation.

MemRx predicts the runtime of a workload for each 32 MB memory increment from 128 MB up to 512 MB. Each workload is subsequently executed with the same amount of available memory as the increments predicted by MemRx. The predicted vs. actual miss-ratio curve and runtime of the microbenchmarks is shown in Figure 3. A miss-ratio curve depicts the fraction of the total misses observed that would still be misses given the corresponding memory configuration. Miss-ratio curves are included to isolate the accuracy of the VEC from the PPM. This aides in understanding and evaluating each component as a source of error.

The graphs show that MemRx can track the important performance features of these workloads accurately. For example, in all cases, the memory configuration after which additional memory produces diminishing returns, the so-called "optimal operating point", is clearly visible. The optimal operating point answers the important procurement question of how to get the most "bang for the buck". In all cases, the total performance change between 128 MB and 512 MB is predicted by MemRx to within 10%, a margin that is useful in making workload migration decisions.

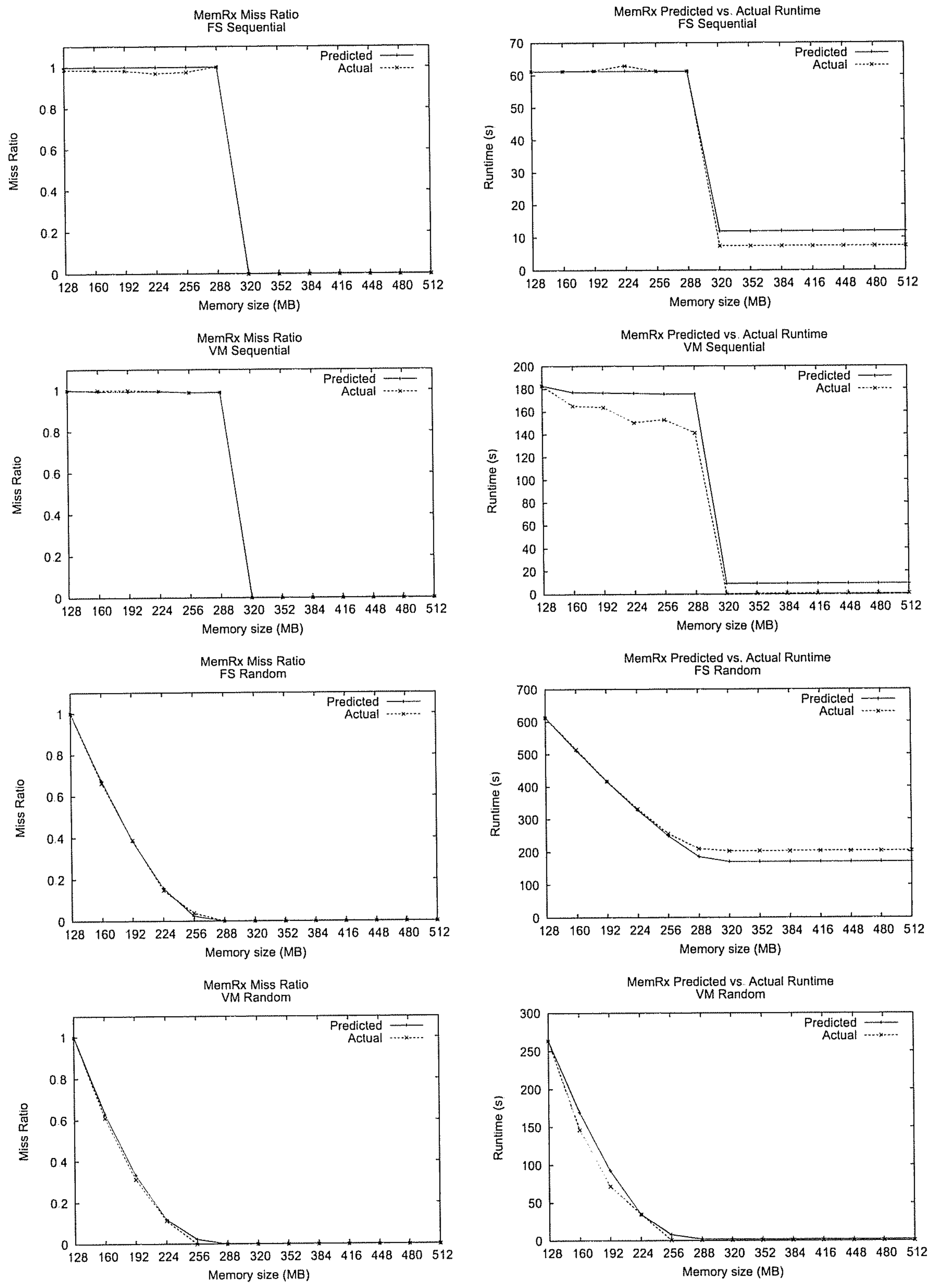


Figure 3: **MemRx Microbenchmark Results.** *The figures show the predicted vs. actual miss-ratio (left) and runtime (right) for the four microbenchmarks described in Table 1.*

Benchmark	Activity
FS Sequential	Sequentially scan a fixed size section of a filesystem file repeatedly
VM Sequential	Sequentially scan a fixed size section of allocated virtual memory repeatedly
FS Random	Randomly read page-sized blocks from a fixed-size filesystem file repeatedly
VM Random	Randomly touch virtual memory pages from a fixed-size virtual memory allocation repeatedly
FS Sequential-MT	A multi-threaded version of FS Sequential

Table 1: **Microbenchmarks** *The table describes each of the microbenchmarks used to evaluate MemRx.*

Figure 4 shows the importance of accounting for multi-threading when predicting runtime. It depicts the runtime prediction error incurred by two different versions of the PPM. The first, labeled "Naive" simply measures the total time required to service each capacity miss, (*i.e.*, it does not account for multi-threading). The second is the multi-thread aware PPM used in the remainder of the evaluations. In the experiment, a synthetic workload, (FS Sequential-MT), that repeatedly does a fixed-size linear scan using a specified number of worker threads is executed. The scan size exceeds available memory resulting in the workload thrashing. The naive PPM incurs increasingly large prediction errors for larger numbers of threads, whereas the thread-aware PPM does not.

The predictions made by MemRx are good, but not perfect. In the following sections we analyze the errors observed while evaluating MemRx.

5.1.2 PPM Error

The PPM calculates the miss penalty for each reload I/O. It does this using a simple model of program runtime in which only capacity miss I/O wait time is accounted for. The model is usually reasonable because most avoidable runtime is due to waiting for synchronous paging I/O to complete, but sometimes this simple model fails to capture behavior that affects performance. In some cases such behavior is complex and counter-intuitive even for extremely simple workloads. In this section we describe several instances of PPM error we observed while evaluating MemRx and their behavioral causes.

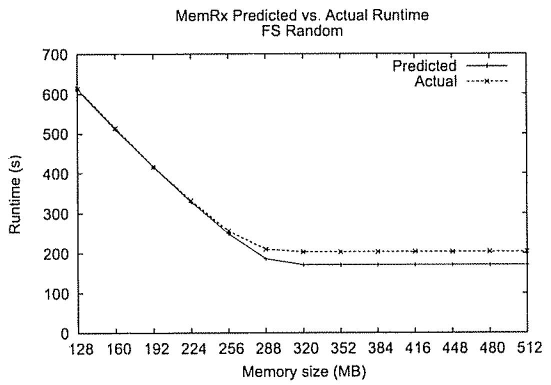
When most of the avoidable I/O tracked by MemRx is sequential, the workload runtime components ignored by the PPM model become more noticeable. The reason is that sequential I/Os complete far more quickly than random I/Os and therefore make up a smaller fraction of total execution time than random I/Os do. An example of this type of error is visible in the FS Sequential workload. Nearly all of the avoidable I/O experienced by this workload is sequential. MemRx therefore noticeably underestimates the penalty experienced by FS Sequential, so the predicted runtimes are slightly too high.

In most cases, MemRx underestimates the benefits of additional memory resulting in runtime predictions that are slightly too high. This is intuitive since the MemRx model excludes the purely computational components of thrashing that would serve to reduce the predicted runtime further. In one of the microbenchmark experiments, however, the opposite is true. For FS Random, a random filesystem workload, MemRx predicts a runtime that is slightly too low, violating our intuition. The reason is that under this workload, the effects of filesystem readahead change with the amount of memory available. This difference is not accounted for by MemRx, leading to a slight discrepancy. More specifically, when this workload is thrashing, readahead is *more* effective than when it is not thrashing. When the workload no longer thrashes, the beneficial readahead effect diminishes, but MemRx does not take variable readahead effects into account, which leads to error.

To support this claim we perform an additional experiment that shows the results of eliminating readahead as a complicating factor. Linux was modified to disable readahead for all I/O. The FS Random experiment from Figure 3 was then repeated. A comparison of predicted vs. actual runtimes is shown for the original and the no-readahead version of the experiment in Figure 5. When readahead is eliminated, the predicted and actual runtimes for FS Random match very closely indicating that a variable readahead effect is the primary source of this type of error.

Finally, we consider the error incurred by VM Sequential. This is an extremely simple workload. It allocates a fixed amount of anonymous memory, then sequentially writes to each page. In the instance of the experiment shown in Figure 3, 256 MB were allocated and accessed 10 times in a loop. The actual performance of this workload gradually improves under larger memory conditions until the allocation completely fits in available memory at which point runtime drops to nearly zero because the disk is no longer being accessed. The predictions made by MemRx miss the gradual change in behavior between the 128 MB and 288 MB configurations.

Initially, we thought this was an instance of VEC error. The nearly exact correspondence between predicted and actual miss-ratio curves contradicts this theory however.



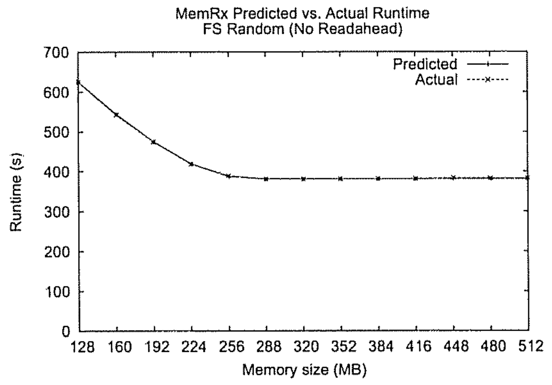


Figure 5: **Effect of readahead on FS Random.** *The graph compares the predicted vs. actual performance for the FS Random microbenchmark when readahead is enabled (left) and disabled (right).*

Available memory	Read/write distance (pages)
128 MB	29843
192 MB	31846
256 MB	18079

Table 2: **VM Sequential seek distances** *The table shows the average seek distance between overlapping clusters of reads and writes for an instance of VM Sequential that accesses 256 MB for three iterations.*

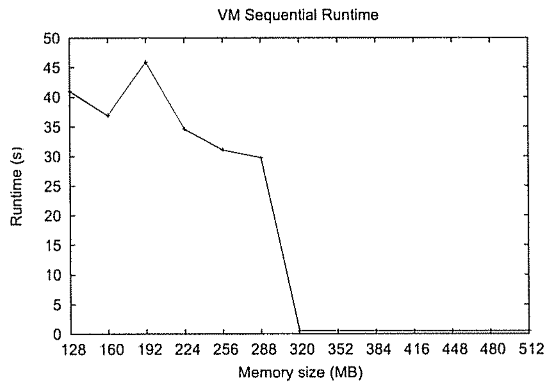


Figure 6: **A pathological instance of VM Sequential.** *The graph shows the non-intuitive runtime of an instance of VM Sequential and confirms the prediction of Table 2.*

The true reason is more subtle and serves to illustrate the kind of complex system behaviors a truly accurate performance predictor would need to model.

When the VM Sequential workload thrashes, it causes overlapping reads and writes in the swap file. Pages are written to the swap file because the workload dirties each page as it accesses it. Depending on how much memory is initially available, the seek distance between overlapping reads and writes changes. The PPM does not take this effect into account leading to the error observed in the 128-288 MB range.

To support this claim, we gathered detailed swap device traces for several additional instances of VM Sequential. Each instance of the workload was run under the same series of memory configurations between 128 and 512 MB as used throughout the evaluation and the average seek distance between overlapping read and write clusters was computed. In each of these instances, the performance experienced by the workload follows the trend predicted by the seek distance. We highlight one case where VM Sequential was configured to allocate 264 MB of memory and then accesses it three times sequentially. Table 2 shows the computed seek distance for three representative memory configurations. The values imply that runtime should be worse when 192 MB is available than when 128 MB is available and then substantially improve under a 256 MB configuration. The runtime graph for this workload shown in Figure 6 confirms that this is true.

As this case shows, real systems exhibit complex, non-intuitive performance behaviors for even simple workloads. It is unrealistic for an online predictive model to capture this type of pathological behavior. An attractive solution is to design systems to avoid these behaviors and thereby enhance their own predictability. In the next section we show one case of how this is possible by improving the predictability of VM Sequential under Linux.

5.1.3 Designing for Predictability

The root cause of the unintuitive behavior exhibited by VM Sequential under Linux is the variable effect of initial conditions on the profile of seeks within the paging file. The effect of variable seeks can be reduced

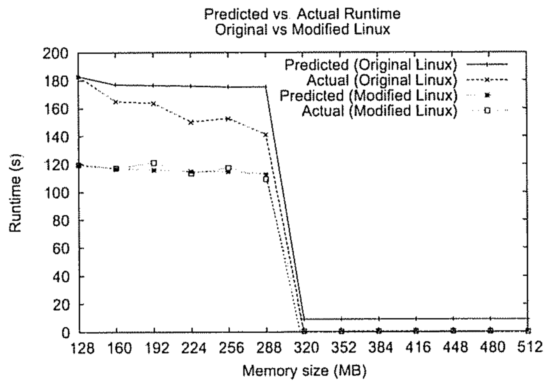


Figure 7: **A more predictable VM Sequential.** *The graph shows how the predictability and performance of VM Sequential increase with a simple configuration change.*

by amortizing seek costs across larger runs of sequential reads. In a memory-starved workload like VM Sequential, this means having adequate memory available to receive paged-in data. One accomplishes this by aggressively writing dirty pages to disk so that the pool of free or clean pages is always plentiful. Linux can be modified to exhibit this behavior by increasing the value of the compile-time parameter `SWAP_CLUSTER_MAX`.

Figure 7 compares the predicted vs. actual runtime curve for VM Sequential under a default Linux configuration to Linux with a large value of `SWAP_CLUSTER_MAX`. The performance of the modified Linux is both more predictable and has improved substantially over the default configuration. It is unlikely that this change improves performance for all workloads, but MemRx-like introspection can also be used to inform dynamic adaptation of system parameters to improve both performance and predictability for varying workloads.

5.1.4 VEC Error

The function of the VEC is to associate capacity misses with the memory increment that would have prevented the relevant data from being evicted. The VEC may assign an I/O to the wrong memory increment leading to incorrect performance predictions in the range of memory configurations where the workload can still use additional memory. VEC errors result from a mismatch between the LRU replacement policy used by the VEC and the true replacement policy employed by the OS. None of the microbenchmarks exhibit this type of error as evidenced by the close correspondence of predicted and actual miss-ratio curves in Figure 3. The behavior of the Linux page replacement policy is close to LRU for these workloads. An example of VEC error will be seen when we examine application workloads in the next section.

Thrashing	Runtime MemRx	Runtime Linux	% Diff
NO	18.93	18.93	0.0%
YES	20.18	21.33	5.7%

Table 4: **MemRx Run-time Overhead.** *For the test system configured with 128 MB of memory, the table lists run-times for a thrashing and a non-thrashing instance of VM Sequential.*

5.1.5 Application Workloads

We have also evaluated MemRx using application workloads that have complex behaviors. The names and a brief description of the applications used in the evaluation are listed in Table 3. Mogrify was used to scale a large bitmap image. Dbench was invoked using its standard file access patterns and configured to simulate 20 remote, concurrent clients. Simplescalar was used to simulate 10 million instructions from the gzip portion of the SPEC microprocessor benchmarks. Each of these workloads was initially executed on our test system configured with 128 MB of free memory and observed by MemRx. The working set of each application exceeds 128 MB. As with the microbenchmark experiments, the applications were then executed at each larger 32 MB interval up to 512 MB and their true runtime was recorded. The predicted vs. actual runtime graphs are shown in Figure 8.

In the case of Mogrify, MemRx clearly identifies the workload's optimal operating point at 288 MB, but underestimates the capacity miss penalty by about 2 seconds. Nevertheless, it accounts for approximately 87% of the total runtime difference between a 128 MB configuration and the 512 MB configuration. For Dbench, MemRx falls prey to VEC error in the range 160 MB through 224 MB. Under this workload, entries in the VEC are pushed too deep in the queue prior to reload. As a result, MemRx reports an optimal operating point of 224 MB rather than the true 160 MB. The fact that this error is due to the VEC is clearly demonstrated by similar deviations in the corresponding miss-ratio curve. MemRx does, however, predict the minimum runtime of Dbench very accurately, accounting for 99% of the total runtime difference. Simplescalar is an interesting case because, although its working set exceeds 128 MB, the application is CPU-bound and its runtime benefits very little from additional memory. MemRx captures this fact nicely.

5.2 Time Overhead

There are two performance regimes for which we wish to evaluate the runtime overhead incurred by MemRx. One is when system memory is severely underprovisioned, the other is when a system's working set fits in main memory.

Application	Activity
Mogrify [13]	Image processing application used to scale a large bitmap image
DBench [20]	Network file server benchmark, simulates network file I/O from many clients
Simplescalar [1]	Architecture simulator for the Alpha microprocessor

Table 3: **Evaluation Applications** *The table describes each of the applications used to evaluate MemRx.*

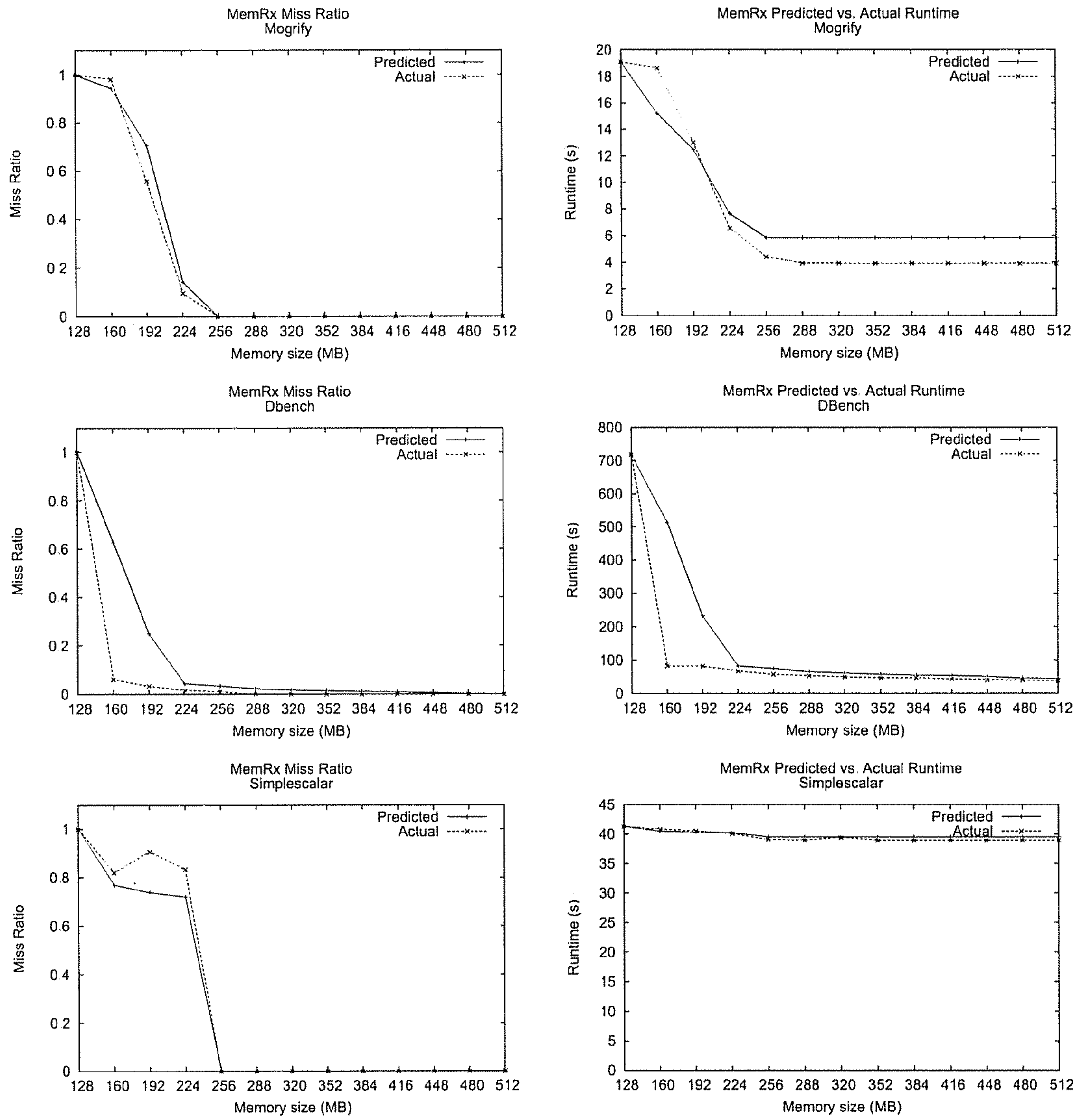


Figure 8: **MemRx Application Results.** *The figures show the predicted vs. actual miss-ratio (left) and runtime (right) for the three applications described in Table 3.*

Real Memory (MB)	Simulated Memory (MB)	MemRx Allocation (MB)	Percent Overhead (%)
128	512	4	3.1%
128	1024	8	6.3%
128	4096	32	25%
512	4096	32	6.2%

Table 5: **MemRx Memory Overhead.** *The memory allocated by MemRx for selected configurations is presented. The first column shows the size of physical memory, the second column the size of the total simulated memory, and the third column the amount of memory used by MemRx. Finally, the last column shows the percent of real memory used by the MemRx data structures.*

To answer these questions we employ VM Sequential, which allocates and sequentially accesses anonymous virtual memory. To test overhead while thrashing we configure the test system with 128 MB of memory and configure VM Sequential to access 256 MB of memory for two iterations. To test a non-thrashing configuration we use the same machine configuration, but configure VM Sequential to access 64 MB of memory 10,000 times. We perform and time these experiments five times with MemRx enabled, and five times using a stock Linux kernel. The resulting times were averaged and appear in Table 4. The variance of the timing samples was negligible.

In the case of a thrashing system, MemRx imposes an approximately 6% runtime overhead. Since runtime is already poor when a system is thrashing, such a small overhead is certainly acceptable. When a system is not thrashing, MemRx imposes no noticeable overhead. This is not surprising because when a system is not thrashing, MemRx is nearly dormant.

5.3 Space Overhead

MemRx requires a variety of data structures to track the disposition of each evicted memory page. In our current prototype implementation, each simulated memory page requires 32 bytes of storage. For our test configuration, which simulates the presence of an additional 1024 MB of memory, this results in an allocation of approximately 8 MB, a small and quite reasonable overhead. Table 5 shows the memory overhead for various other configurations. On low-memory systems where one would like to simulate a large amount of additional memory (*e.g.*, the third configuration in the table), this overhead could quickly become a burden.

In its current form, MemRx is thus likely limited in the size of memories it can simulate by space overhead. In many installations, the small overhead is likely worth the benefit. However, on machines with little memory, the cost of simulating a large amount of additional memory may be unacceptable. Therefore, we believe it would be useful to investigate space-saving, engineering solutions. For example, a sampling-based approach could track evictions and reloads probabilistically, or the memory required by MemRx could be allocated just-in-time, instead of the fixed, ahead-of-time allocation strategy used by our prototype.

6 Conclusions

"Is it live ... or is it Memorex?"
Memorex commercial

Memory configuration is one of the most important aspects of running workloads effectively, and yet few tools assist users or management software to quantitatively determine how a workload could benefit from additional physical memory. In this paper, we have designed, implemented, and evaluated MemRx, an operating system extension that provides concrete performance prediction estimates for a workload under larger memory configurations.

MemRx is comprised of two key components: a performance prediction module, which measures the runtime penalty for each capacity miss, and a virtual extended cache, which associates penalties with memory size increments by tracking hits and misses in larger, simulated memory configurations. This combination allows MemRx to accurately predict workload runtime for larger memory configurations using a simple model of workload runtime. Through microbenchmark and application workloads, we have evaluated MemRx and found that it is accurate and has moderate space overhead and small time overheads making it practical to run continuously within a system.

MemRx observes and measures system events like page table updates and disk I/O requests. Often, useful, categorical information describing the source and type of those events is discarded as they cross system layer boundaries. This hampers the implementation of services like MemRx. Designers of future introspective systems should consider how this kind of categorical information about requests and events can be efficiently maintained and distributed across system layers.

As systems and workloads become more complex, there is a need for more intelligent components to monitor activity and provide assistance to higher level systems or administrators in their management decision making. MemRx is one such tool. When supplied with information about the benefits of additional physical memory, an administrator or higher-level system can make more informed and reliable procurement and configuration decisions. MemRx helps transform memory configuration and workload scheduling from art to engineering.

References

[1] T. Austin. Simplescalar architecture simulator. http://www.simplescalar.com.

[2] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353, June 1969.

[3] S. Chaudhuri and V. Narasayya. Autoadmin "what-if" index analysis utility. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 367–378, New York, NY, USA, 1998. ACM Press.

[4] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[5] P. J. Denning. Working Sets: Past and Present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[6] F. Douglis and J. K. Ousterhout. Process Migration in the Sprite Operating System. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 18–25, Berlin, West Germany, September 1987. IEEE.

[7] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[8] G. R. Ganger, J. D. Strunk, and A. J. Klosterman. Self-* storage: Brick-based storage with automated administration. Technical Report CMU-CS-03-178, Department of Computer Science, Carnegie Mellon University, August 2003.

[9] T. Johnson and D. Shasha. 2Q: A Low-Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB 20)*, pages 439–450, Santiago, Chile, September 1994.

[10] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, January 2003.

[11] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report Technical Report 1346, University of Wisconsin-Madison Computer Sciences, April 1997.

[12] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of ACM Computer Network Performance Symposium*, pages 104–111, June 1988.

[13] I. S. LLC. Imagemagick image processing software. http://www.imagemagick.org.

[14] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9, 1970.

[15] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[16] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 79–95, Copper Mountain Resort, Colorado, December 1995.

[17] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[18] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson. EELRU: Simple and Effective Adaptive Page Replacement. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99)*, pages 122–133, Atlanta, Georgia, May 1999.

[19] E. Thereska, D. Narayanan, and G. R. Ganger. Towards self-predicting systems: What if you could ask "what-if"? In *3rd International Workshop on Self-adaptive and Autonomic Computing Systems*, Copenhagen, Denmark, Aug 2005.

[20] A. Tridgell. Dbench filesystem benchmark. http://samba.org/ftp/tridge/dbench.

[21] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[22] J. Wilkes, J. Mogul, and J. Suermondt. Utilification. In *11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[23] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamically Tracking Miss-Ratio-Curve for Memory Management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, Massachusetts, October 2004.