PATH: Page Access Tracking to Improve Memory Management

Reza Azimi, Livio Soares, Michael Stumm
Department of Electrical and Computer Engineering
University of Toronto
{azimi,livio,stumm}@eecg.toronto.edu

Thomas Walsh, Angela Demke Brown
Department of Computer Science
University of Toronto
{tom,demke}@cs.toronto.edu

Abstract

Traditionally, operating systems use a coarse approximation of memory accesses to implement memory management algorithms by monitoring page faults or scanning page table entries. With finer-grained memory access information, however, the operating system can manage memory much more effectively. Previous work has proposed the use of a software mechanism based on virtual page protection and soft faults to track page accesses at finer granularity. In this paper, we show that while this approach is effective for some applications, for many others it results in an unacceptably high overhead.

We propose simple Page Access Tracking Hardware (PATH) to provide accurate page access information to the operating system. The suggested hardware support is generic and can be used by various memory management algorithms. In this paper, we show how the information generated by PATH can be used to implement (i) adaptive page replacement policies, (ii) smart process memory allocation to improve performance or to provide isolation and better process prioritization, and (iii) effectively prefetch virtual memory pages when applications have non-trivial memory access patterns. Our simulation results show that these algorithms can dramatically improve performance (up to 500%) with PATH-provided information, especially when the system is under memory pressure. We show that the software overhead of processing PATH information is less than 6% across the applications we examined (less than 3% in all but two applications), which is at least an order of magnitude less than the overhead of existing software approaches.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISMM'07, October 21–22, 2007, Montréal, Québec, Canada.
Copyright © 2007 ACM 978-1-59593-893-0/07/0010…$5.00

Categories and Subject Descriptors C.4 [*Computer System Organization*]: Performance of Systems—Design studies; D.4.2 [*Operating Systems*]: Storage Management—main memory, virtual memory

General Terms Algorithms, Management, Measurement, Performance, Design, Experimentation

Keywords page access tracking, translation lookaside buffer, page replacement policy, process memory allocation, prefetching

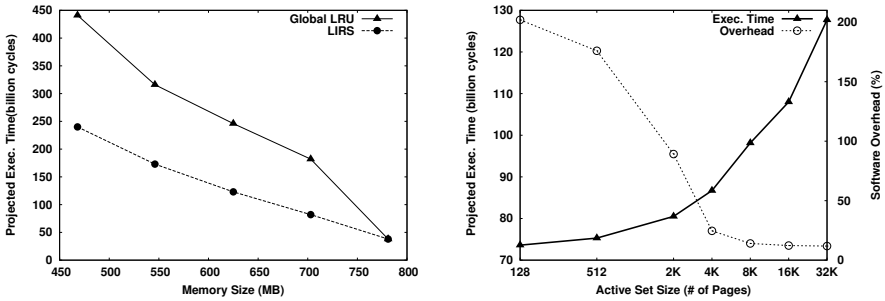
1. Introduction

Computer system physical memory sizes have increased consistently over the years, yet counter to popular conception, optimizing the allocation and management of memory continues to be important. Numerous scientific and engineering applications exist that can exhaust even large physical memory [1, 8, 31]. Moreover, while physical memory is generally considered to be inexpensive, it continues to be one of the dominant factors in the cost of today's medium to large scale computer systems, and also a major factor in energy consumption.

To use memory effectively, accurate information about the memory access pattern of applications is needed. Traditionally, operating systems track application memory accesses at a relatively coarse granularity, either by monitoring page faults or by periodically scanning page table entries for specific bits set by hardware. While these approaches provide a coarse approximation of the *recency* of page accesses, important information about the *sequence* of accesses, which is required by most sophisticated memory management algorithms, is absent.

In systems with software-managed TLBs, page accesses can be recorded and processed on each TLB miss. While this approach can provide significantly more fine-grained information on page accesses, it adds prohibitively large overhead to a software TLB miss handler, which is already a performance-critical component.

A software-only alternative in which virtual pages are divided into an *active set* and an *inactive set* has been sug-



(a) Performance of LIRS vs. LRU (b) The effect of active set size

Figure 1. Graph (a) shows how LIRS manages to outperform LRU for different memory sizes for FFT. Graph (b) shows, for a fixed memory size (703Mbytes), how LIRS' performance changes as the active set size increases, while the runtime overhead of maintaining the active set deceases (the projected execution time does not include the runtime overhead).

gested by recent research [30, 33]. Pages in the inactive set are protected by appropriately setting page-table bits, so that every access to them will generate an exception so that the operating system can record the access. Pages in the active set are not protected, and as a result, accesses to these are efficient and not directly tracked. Pages are moved from the inactive set to the active set on access, and a simple replacement algorithm such as CLOCK [5] is used to move stale pages out of the active set. The active set, although much smaller than the inactive set, is meant to absorb the majority of page accesses, thus greatly reducing the software overhead compared to raising an exception on every access.

Although this software approach is shown to be effective with certain types of applications, its overhead for many memory-intensive applications is unacceptably high. Adaptive resizing of the active set can be used to control the overhead [30]. However, the larger the active set, the more accesses it absorbs and, hence, the less *accurate* the sequence of recorded page accesses will be, making the memory management algorithms less effective. An example of such a case is shown in Figure 1. On the left, the performance of LIRS [13], a well-known memory management algorithm, is compared against LRU assuming no overhead for collecting page access information. The graph on the right shows how the performance of LIRS degrades as the active set size increases, while the overhead of recording page accesses naturally decreases. To achieve LIRS' potential in improving performance, a high runtime overhead must be paid, otherwise, much of the advantage of LIRS over LRU disappears.

To cope with this potentially large overhead, custom hardware is suggested by Zhou et al. [33]. While their approach effectively tracks physical memory *Miss Ratio Curves*, it does not provide raw page access information to the operating system, and thus cannot be used for memory management algorithms other than the one for which it is intended. Moreover, the hardware required by this approach is substantial and grows with the size of physical memory.

In this paper, we propose Page Access Tracking Hardware (PATH) to be added to the processor micro-architecture to monitor application memory access patterns at fine granularity and with low overhead. Similar to the software approach, PATH is designed based on two observations. First, a relatively small set of *hot* pages are responsible for a large fraction of the total page accesses. Second, the exact order of page accesses within the hot set is unimportant since these pages should always be in memory. By ignoring accesses to hot pages, we can vastly reduce the number of accesses that must be tracked, while focusing on the set of pages that are interesting candidates for memory management optimizations.

The key innovation with PATH lies in the tradeoff between functionality assigned to hardware and functionality assigned to software. The hardware we propose is (i) small and simple, (ii) scalable, in that it is independent of system memory size, and (iii) low overhead, imposing no delays on the common execution path of the micro-architecture. We delegate to software (specifically, an exception handler) the online maintenance of data structures to be used by the memory manager when making policy decisions.

Section 2 presents our hardware design for PATH and Section 3 shows how key low-level data structures can be constructed by software. We show in Section 4 that the operating system can use PATH-generated information to enhance memory management by (i) implementing more adaptive page replacement policies, (ii) allocating memory to processes or virtual machines so as to provide better isolation and to enforce process priorities more precisely, and (iii) prefetching pages from virtual memory swap space or memory-mapped files when applications have non-trivial memory access patterns. Section 5 describes our experimental methodology. Our simulation results, presented in Section 6, show that substantial performance improvements (up to 500% in some cases) can be achieved, especially when the system is under memory pressure. While the algorithms based on PATH have different time and space over-

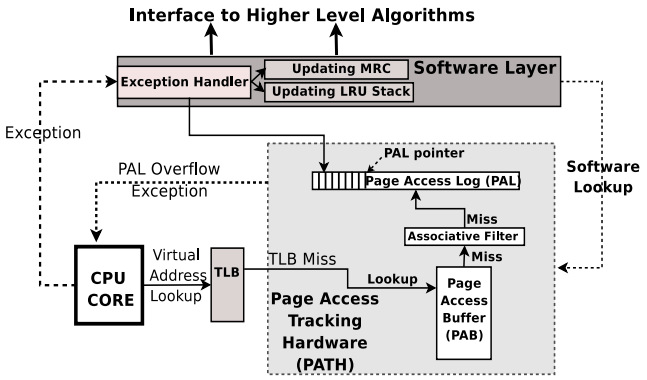


Figure 2. Page Access Tracking Hardware (PATH) Architecture.

head tradeoffs, the basic overhead of providing fine-grained page-access information to the operating system is less than 6% across all the applications we examined (less than 3% in all but two applications) – at least an order of magnitude less than that of existing software approaches.

2. Design of PATH Architecture

Memory management algorithms are often first described theoretically under the assumption that a complete page access sequence is available. Later, they are implemented using a coarse approximation of this sequence, collected by system software. For example, the well-known least-recently-used (LRU) page replacement algorithm requires the complete access sequence to implement exactly, but is commonly approximated by the CLOCK algorithm which coarsely groups pages into recently-used, somewhat recently used, and not recently used categories. Optimizations to the basic LRU algorithm, and other sophisticated memory management strategies, require more detailed page access information than systems currently provide. Tracking *all* accesses, however, is prohibitively expensive and generates too much information for online processing. The key question, then, is how to reduce the volume of information to a manageable level, while retaining sufficient detail on the order of page accesses.

Current memory management hardware already contains an effective filter to catch accesses to the hottest N pages, namely the Translation Lookaside Buffer (TLB). Thus, one way to track page accesses is to augment existing hardware or software TLB miss handlers to record a trace of all TLB misses. Aside from the overhead that this would add to the critical path of address translation, the primary problem with this strategy is that first-level TLBs are too small (with up to 128 entries) to capture the set of hot pages, leading to traces that are still too large for online use. Simply increasing the size of the first-level TLB is not a viable option, since the size is limited by fast access requirements.

Thus, we propose the addition of a new hardware structure that essentially functions as a significantly larger TLB for the purpose of filtering out accesses to hot pages, while recording a trace of accesses to all other pages. We call this structure *Page Access Tracking Hardware (PATH)*. Figure 2 depicts the three major components of PATH. The *Page Access Buffer* (PAB) and the *Associative Filter* work together to remove accesses to hot pages from the trace; all other accesses are recorded in the *Page Access Log* (PAL) which raises an exception to allow for software processing when it becomes full.

The Page Access Buffer (PAB) contains the set of recently accessed virtual pages, augmented with an address space identifier to distinguish between pages from different processes. The PAB is structurally similar to a TLB except that (i) it is updated only on a TLB miss, (ii) it need not contain the physical addresses of the pages it holds, and (iii) it is significantly larger than a typical TLB. As the PAB size increases, more pages are considered *hot* and more accesses are filtered out of the trace, thus reducing both processing overhead and accuracy. In Section 6.4, we examine in detail the tradeoff between overhead and usefulness of the traces with varying PAB sizes. Our experiments show that a PAB with 2048 entries is a good point in this tradeoff. Moreover, with a 2K-entry PAB, PATH will have a very small chip footprint. Finally, some existing architectures such as IBM POWER and AMD Opteron already have a fairly large (e.g. 512 to 1024 entry) second-level TLB (In IBM POWER processors, the first level address translation cache is usually 128 entries and is called the Effective-to-Real Address Table or ERAT). One can envision integrating PATH with a slightly larger version of such a second-level TLB. We show in Section 6.5 that using the same 2K size for the active set in the software approach will result in unacceptably high overhead.

A page access is considered for recording only if it misses in the PAB. However, because of the limited associativity of the PAB, it can be susceptible to repeated conflict misses from the same small set of (hot) pages. To deal with this problem, PATH includes an Associative Filter that functions somewhat like a victim cache. The associative filter is a small (e.g., 64 entries), fully-associative table with an LRU replacement policy that is updated on every PAB miss. Its purpose is to prevent the recording of accesses to hot pages caused by short term conflict misses in the PAB.

Misses in the associative filter are recorded in the Page Access Log (PAL) which is a small (e.g., 128 entries) buffer. When the log becomes full, an exception is raised, causing an operating system exception handler to read the contents of the PAL and mark it empty by resetting the PAL pointer.

Given this architecture, PATH provides a fine-grained approximation of the sequence of pages that are accessed. Hot pages will always reside in the PAB, while sequential or looping access patterns over an area larger than that covered by the PAB (e.g., 8MB) are very likely to be completely

recorded by PATH in their proper order. For "less hot" pages, the reuse distance can also be accurately captured by PATH due to the subsequent PAB misses it causes. In the following section, we show how system software can use the information recorded in PAL to construct a variety of data structures useful for memory management.

Finally, PATH must also include an interface for the operating system to control it and to perform lookup operations on it. The operating system can also dynamically turn off PATH when the system is not under memory pressure, thereby reducing both the processing overhead and power consumption.

3. Low-level Software Structures

The benefits of having LRU stacks and/or Miss Rate Curves (MRC) available are well recognized [33]. In this section we argue that these data structures can be constructed efficiently in software from the information obtained by PATH. Specifically, we show how both LRU stacks and Miss Rate Curves can be maintained on-line by the PAL overflow exception handler. Both structures can, in turn, be used by memory management software to make informed decisions. By delegating the maintenance of these structures to software, our design provides greater flexibility and customizability than previous proposed hardware support.

3.1 LRU Stack

The LRU stack maintains a recency order among the pages within an address range. The top of the stack is the most recently accessed page, while the bottom of the stack is the least recently accessed page. In our scheme, each page accessed (as recorded by the PAL) is moved from its current location in the stack to the top of the stack. The LRU stack is updated for every entry recorded in the PAL.

To enable fast page lookup and efficient update in the LRU stack, we suggest using a structure similar to those used to maintain page tables. Each element in this structure represents a virtual page and contains two references: one to the previous page in the LRU stack and one to the next page in the LRU stack. Conceptually, the LRU stack is a doubly-linked list, and elements are repositioned within the stack by adjusting references to neighboring elements. Thus, a virtual page can be looked up with a few (usually 2 or 3) linear indexing operations, and moving a page to the top of the LRU stack involves updating at most 6 reference fields in the stack: 2 references associated with the page being moved, 2 of its previous neighbors, 1 at the previous head of the list, and the head of the list itself.

The LRU stack has an element for each page that was ever accessed (not just the pages currently in memory). Assuming 4 KB virtual pages, 32-bit page references can be used for address ranges up to 16 TB, resulting in a space overhead of 8 bytes per virtual page used. The working set size of the LRU stack is roughly proportional to the working set size of the address range. Hence, a working set size of several GB implies that several MB will be consumed by the LRU stack.

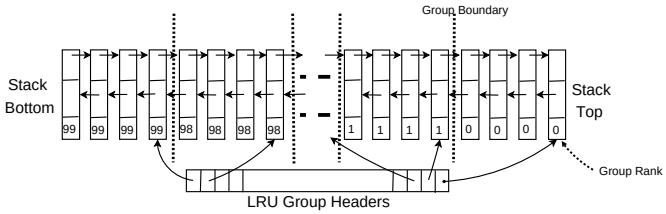


Figure 3. The LRU stack with group headers.

3.2 Miss Rate Curve (MRC)

An MRC depicts the page miss rate for different memory sizes, given a page replacement strategy. More formally, MRC is a function $\lambda_{r,p}(M)$, defined for address range r and page replacement policy p, identifying the number of page misses the process will incur on r over a certain time period if M physical pages are available. Often, the slope of λ at a given memory size is of more interest than its actual value. If the slope is flat then making additional pages available will not significantly reduce the miss rate, but if the slope is steep then even a few additional pages can significantly reduce the page miss rate.

Our method of maintaining λ on-line is based on Mattson's stack algorithm [19] and Kim *et al.*'s algorithm [17]. We augment the elements of the LRU stack described in Section 3.1 with a *rank* field used to record the distance of the element from the top of the stack (i.e., the reuse distance). Each λ is maintained as a histogram. Conceptually, whenever a page is accessed, the histogram values corresponding to memory sizes smaller than the rank of the accessed page are incremented by one. In addition, the page is moved to the top of the stack, while setting its rank field to zero and decrementing the rank field of every element between the original position of the page and the previous top of stack by one.

Time is divided into a series of *epochs* (e.g., a few seconds). At the end of each epoch, the value of λ is saved and reset. Each process may store a history of values of λ for several epochs to be able to make more accurate decisions about the memory consumption of that process.

To reduce overhead, page groups of size g can be defined and the rank field can be redefined to record the distance to the top of the stack in terms of the number of page groups. By keeping an array of references to the head of each page group, the cost of updating the rank fields can be reduced by a factor of g. Figure 3 shows how the group header array is used to find the group boundaries, since only the elements at these boundaries need to be updated. Algorithm 1 shows the basic steps that must be taken for every page recorded in the PAL to maintain λ histograms for the LRU replacement policy. Note that the group size g is defined by software and can change according to the desired level of precision for λ.

Algorithm 1. Update λ_{LRU} and the LRU stack on each recorded page V_{addr}.

1: $lruRank \Leftarrow Stack[Vaddr].rank$
2: move $Vaddr$ element to the top of the LRU stack
3: $Stack[Vaddr].rank = 0$
{*update group headers and page ranks for groups lower than* $lruRank$}
4: **for** $i = 0$ to $lruRank$ **do**
5: $GroupHeaders[i] \Leftarrow Stack[GroupHeaders[i]].prev$
6: $Stack[GroupHeaders[i]].rank++$
7: **end for**
{*update MRC for LRU*}
8: **for** $j = 0$ to $lruRank$ **do**
9: $\lambda_{LRU}[j]++$
10: **end for**

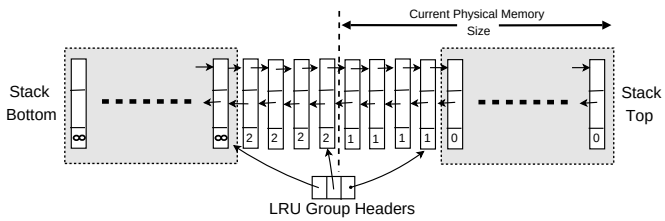


Figure 4. The optimized structure for LRU group headers.

A further optimization is possible based on the observation that at any instance in time, we are only interested in λ at the point corresponding to the amount of physical memory allocated to the virtual address range under study and the slope of λ around that point. Hence, the LRU stack can be divided into 4 groups as shown in Figure 4: the top $M - g$ pages, where M is the current physical memory allocated to the address range, two groups of g pages on both sides of M, and all the remaining pages at the bottom of the LRU stack. With this optimization, only four entries need to be updated on each page access to maintain λ.

4. Example Use Cases

In this section we describe several ways that the information provided by PATH, and the LRU stacks and MRC curves that are constructed by software, can be used to implement sophisticated memory management strategies, including adaptive page replacement, improved process memory allocation, and virtual memory prefetching. In Section 6 we evaluate their effectiveness.

4.1 Adaptive Replacement Policies

There is a large body of research on page replacement policies [2, 7, 9, 10, 12, 13, 14, 16, 20, 23, 26, 32]. Many of the algorithms proposed are approximations of LRU with extensions to deal with sequential and looping patterns for which LRU performs poorly. The effectiveness of most of these algorithms has only been shown in the context of file system caching, where precise information on the timing and order of accesses is available.

Using information from PATH, we have implemented two adaptive page replacement algorithms. The first one, *Region-Specific Replacement*, attempts to automatically apply the appropriate replacement policy on a per-region basis for different memory regions defined in the application's virtual address space. The second one is the recently proposed adaptive policy called *Low Inter–Reference Set (LIRS)* [13]. We chose to implement LIRS because it is fairly simple and, for file system caching, has proven to be competitive with the best algorithms.

We should note that the algorithms or models that can exploit information provided by PATH are not limited to the examples presented in this section. For instance Vilayannur et al. [28] present a model to proactively predict when a page is not actively used and hence is ready to be replaced. The model is based on accurately measuring the distance between consecutive accesses to a page, which can easily be provided by PATH.

4.1.1 Region-Specific Replacement

The rationale behind region-specific page replacement is the desire to be able to react individually to the specific access patterns of each large data structure within a single application. Studies in the context of file system caching [7] have shown that by analyzing the accesses to individual files separately, one can model the access pattern of the applications more accurately. We argue that memory-consuming data structures (e.g., multidimensional arrays, hash-tables, graphs) usually have stable access patterns, and by detecting these patterns, one can optimize the caching scheme for each of these data structures individually.

Most large data structures either reside in contiguous regions in the virtual address space (e.g., arrays), or could reasonably be made to do so. For example, one can use custom allocators that allocate correlated data from a pre-allocated pool of virtual memory. Lattner and Adve [18] show how to cluster individually allocated, but correlated, memory items automatically. As a result, large data structures (e.g., a graph of millions of nodes) have a high probability of being located in a large contiguous region of address space. The contiguity of data structure memory is not an essential factor, but it simplifies the implementation of region-specific replacement. For our simulation study, we have assigned a separate region for each large static data structure as well as any large `mmap`ped areas.

We choose the replacement policy by separately, but simultaneously, computing λ for each region for both LRU and MRU policies; and picking the policy that would result in a lower miss rate. To compute λ_{MRU} we use the same scheme shown in Figure 4 and Algorithm 1, but with pages ranked in reverse order. Hence, for each page, we maintain two ranks, one for LRU and the other for MRU. Given that

the rank value is at most 4, the rank can be represented by two bits, so the space overhead is negligible.

We switch to a new policy only if it is consistently better than the current policy. The default policy is LRU. If a region is being accessed in a looping pattern, it will have lower values for λ_{MRU}, but if the region is being accessed in temporal clusters, λ_{LRU} will have lower value.

With region-specific page replacement, it is necessary to decide how many physical pages to allocate to each region. At the end of each epoch, we use the computed λ values for the epoch to calculate how much memory each region actually needs. We define $benefit$ and $penalty$ functions for each region as follows:

$$benefit_r(g) = \lambda_{r,p}(M - g) - \lambda_{r,p}(M)$$
$$penalty_r(g) = \lambda_{r,p}(M) - \lambda_{r,p}(M + g)$$

and balance memory among regions within a process address space by taking pages away from regions with low penalty and awarding them to the regions with higher benefit. The number of regions in an application is typically small (e.g., usually less than 10). Thus, balancing memory within a single application at the end of each epoch is not a costly operation.

4.2 Process Memory Allocation

In most general-purpose operating systems today, memory is allocated to a process on-demand, in response to a page fault, from a global pool of pages. All pages are equal candidates for replacement, irrespective of the process to which they belong. The actual amount of memory allocated to each process is a direct function of its page fault rate and the page replacement policy in use. Processes that access more pages than others over a period of time will be allocated a larger number of pages, since they fault on more pages and keep their own pages recent. Global page replacement has two major advantages. First, it is simple and easy to implement with little overhead. Second, for workloads in which applications have similar access patterns, global page replacement naturally tends to minimize the total number of page-faults. Despite its wide adoption, global page replacement has two significant shortcomings:

Sub–optimal System Throughput: Global page replacement assumes each application receives the same benefit when given an extra page. In reality, however, one application's throughput may rise sharply as it is given more pages, whereas others may see no performance gains. If the goal is to maximize overall system throughput, pages should be taken away from processes that derive little benefit from them and given to processes that benefit the most.

Lack of Isolation and Unfair Prioritization: Global page replacement does not guarantee any level of service for applications. So-called "memory hogs" can starve applications with even a small working set size [4]. Similarly, in a system under memory pressure, process prioritization done only through CPU scheduling can become ineffective.

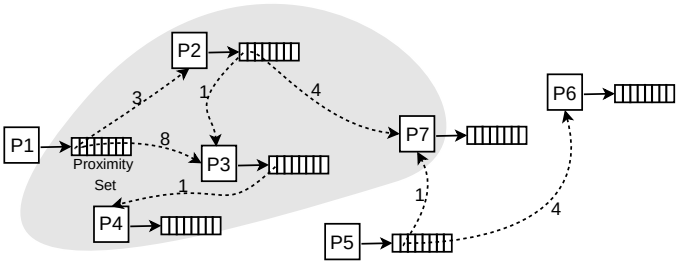


Figure 5. Page Proximity Graph. The shaded area shows the prefetch set for page P1 when traversing to a depth of 2.

Chapin identified the prioritization problem due to lack of memory isolation in operating systems, and motivated the concept of *memory prioritization* [6].

Our approach to optimizing throughput is similar to the greedy algorithm used by Zhou *et al.* [33] with a different level of hardware integration. In this approach, each process is initially allocated an equal amount of physical memory. At each memory allocation step, λ is calculated for all processes, and $penalty_P$ and $benefit_P$ for process P are calculated as follows:

$$benefit_P(g) = \lambda_p(M) - \lambda_p(M + g)$$
$$penalty_P(g) = \lambda_p(M - g) - \lambda_p(M)$$

The greedy algorithm takes g pages away from the process with the least value for $penalty_P(g)$, and assigns them to the process with the highest value for $benefit_P(g)$.

To address unfair prioritization, different policies can be implemented using λ. For example, physical memory may be partitioned to balance the miss rates of concurrently running applications. We are continuing to explore different schemes for fairness and process isolation using fine-grained memory access information provided by PATH.

4.3 Virtual Memory Prefetching

Increases in I/O bandwidth over the years now allow for aggressive and speculative prefetching of memory pages. An aggressive prefetching scheme, however, risks replacing pages that are more valuable (to the same or other applications) than those prefetched.

A simple operating system-level prefetching approach is based on *spatial locality*: pages adjacent to the faulted page in the virtual address space are candidates for prefetching on the assumption that they will be accessed soon. More precisely, whenever a page-fault happens, the next w pages in the address space are prefetched from the swap space, where w could be either fixed or dynamically adjusted based on how accurately the prefetching policy has been performing. This scheme is effective in many cases, since large, memory-consuming applications often access pages in contiguous chunks that are much larger than a virtual page. However, there are important classes of applications that have stable access patterns, but with little or no spatial locality.

As an alternative, we have implemented a prediction model similar to a Markov predictor [15] that incorporates the temporal proximity of accesses to pages as the key factor. We use the LRU stack to find temporal proximity among pages, similar to recency-based prediction models, such as the one proposed by Saulsbury et al. [25]. Note that the LRU stack must be precise to provide accurate information on the proximity of page accesses. As we showed in Section 3.1, the LRU stack is accurately maintained by using the PATH-generated information.

Our model uses a weighted graph, called the *Page Proximity Graph* (PPG), which identifies how often two virtual pages are accessed shortly after each other. For each page p, we maintain a *Proximity Set*, X_p, where $|X_p|$ is at most D pages. Figure 5 shows a simple example of a PPG where D is equal to 8.

The PPG is updated on each page fault as follows. A window of W_{scan} pages in the LRU stack is considered, starting from the current location of the faulted page, p, towards the top of the stack. If any page, q, in the scan window is already in X_p, the weight on (p, q) is incremented by one. Otherwise, q is considered as a candidate to be added to X_p. The weight to all other nodes in X_p that do not appear in the scan window is decremented to decay obsolete proximity information. If the weight on any edge (p, s) reaches zero, s is removed from X_p.

Prefetching is initiated whenever a page fault occurs on a page, such as p. To generate the set of pages to prefetch, the PPG is traversed, starting from p, in a breadth-first fashion, and all pages encountered are added to the prefetch set. If a page in the prefetch set is already resident in memory, it will be artificially touched to prevent the page replacement algorithm from evicting it, under the assumption that it will likely be accessed soon. In Figure 5, the gray region shows the prefetch set when starting from P_1 and traversing to a depth of 2. The deeper the breadth-first traversal, the more speculative prefetching will be. One can dynamically adjust the depth of the traversal according to the current prefetching effectiveness and available I/O bandwidth.

5. Experimental Framework

We used Bochs [3], a widely used full-system functional simulator for the IA-32 architecture, to run the applications and record their memory accesses. This memory trace was then fed to a simulator that simulates the memory-management algorithms in a multi-programmed environment to obtain the page fault rate. To estimate execution time, we first timed the execution of all workloads on an AMD Athlon 1.5GHz system with enough memory to ensure that no page faults occurred. Then, we calculated a *projected execution time* given the page fault rate determined by simulation.

The projected execution time is calculated as follows:

$$Projected_Exec_Time = Exec_Time_0 + Wait_{PF}$$
$$Wait_{PF} = Average_Latency_{Page_Fault} * Total_Page_Faults$$

where $Exec_Time_0$ is the execution time measured when no page fault occurs. We assume that once a process faults on a page, it will be blocked for $Average_Latency_{Page_Fault}$ cycles; we use a fixed value of one million CPU cycles for $Average_Latency_{Page_Fault}$. This value conservatively underestimates the cost of page faults as the average disk access latency of even fast disks is on the order of a few milliseconds.

5.1 Applications

We evaluated the effectiveness of PATH on a set of memory-consuming applications that we chose from various benchmark suites: six applications from Splash-2 [29], four from the NAS Parallel Benchmark (NPB) suite [22], SPECJBB 2000 [27], MMCubing from the Illimine data mining suite [11], and MrBayes, a Bayesian inference engine for phylogeny [21]. We did not include SPEC CPU benchmarks, since they have fairly small memory footprints.

We ran the applications with large problem sizes within the practical limits of the simulation environment (e.g., on the order of a few hundred megabytes). However, all of these applications will consume up to tens of gigabytes of memory for large but still realistic problem sizes. For our experiments, we collected memory traces that cover the execution of a few hundred billion instructions for each application. A *warm up* time is considered at the beginning of the simulation in which no measurement is done.

6. Experimental Results

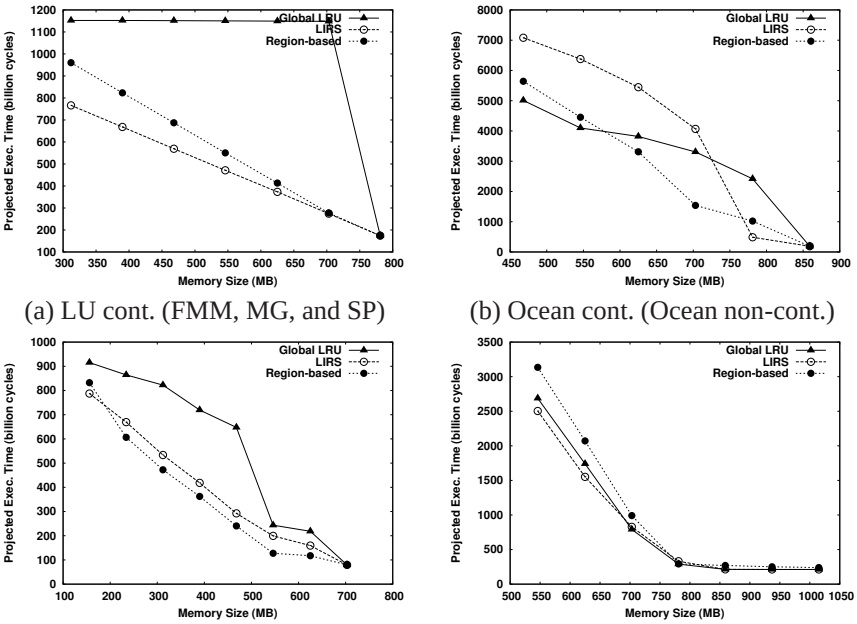
6.1 Adaptive Replacement Policies

Figure 6 shows the effect of using different replacement policies on execution time as memory size is varied. Due to space limitations, we show the results only for a set of four applications with representative behavior.

For the great majority of applications, using one of the adaptive policies resulted in a significant improvement in the projected execution time (e.g., around 500% for `LU cont.`). Comparing region-specific and LIRS policies, in some cases one performs slightly better than the other and vice versa, but generally their difference is not significant. There are also rare cases in which one of the adaptive policies performs slightly worse than the basic LRU algorithm (e.g., `Ocean` for LIRS and `SPECJbb` for region-specific). Note that most of the benefit of both LIRS and region-specific policies are the result of having accurate page access information from PATH.

6.2 Process Memory Allocation

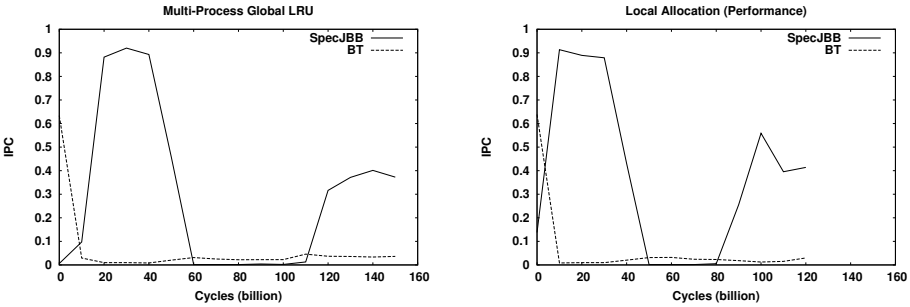
To demonstrate the benefits of fine-grained memory access pattern information for local (per-process) page replacement schemes, we show that total system throughput (in terms of Instructions Per Cycle) can be improved over a traditional global replacement strategy. In this experiment, we simulate two applications running simultaneously: `SPECJbb` and `BT`.



(a) LU cont. (FMM, MG, and SP) (b) Ocean cont. (Ocean non-cont.)

(c) BT (FFT and MrBayes) (d) SPECJbb (LU non-cont. and CG)

Figure 6. Projected execution time of selected applications with different replacement policies. The applications in parenthesis are those with similar behavior.



(a) Global LRU (IPC) (b) Local (Maximizing Throughput) (IPC)

Figure 7. Comparing the traditional global replacement versus a local replacement strategy that aims to maximize system throughput in multi-programmed scenario: SpecJBB and BT.

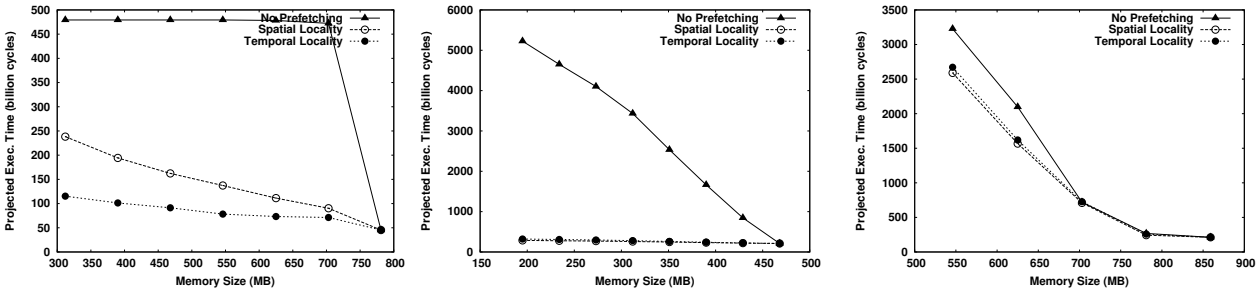
Without loss of generality, to make the experiment more clear, we assumed that the IPC of both applications is 1 when running in isolation. As noted in Section 5, each page fault is assumed to have a fixed latency of one million cycles. We used a warm-up time of 30 billion instructions and a running time of 60 billion instructions combined.

Figure 7 (a) shows the average IPC for both applications when run with global LRU replacement; Figure 7 (b) shows the average IPC when the applications run with local LRU replacement and memory allocation set to maximize throughput. The trend in IPC is similar for both setups; however, our local allocation policy achieves higher overall IPC, needing roughly 18% fewer cycles to execute the same number of instructions (145 billion cycles vs. 178 billion cycles for the global strategy). This is mainly because `SPECJbb` derives a higher benefit from extra pages than `BT`, while a global scheme considers the utility of each page to be the same for both applications.

6.3 Virtual Memory Prefetching

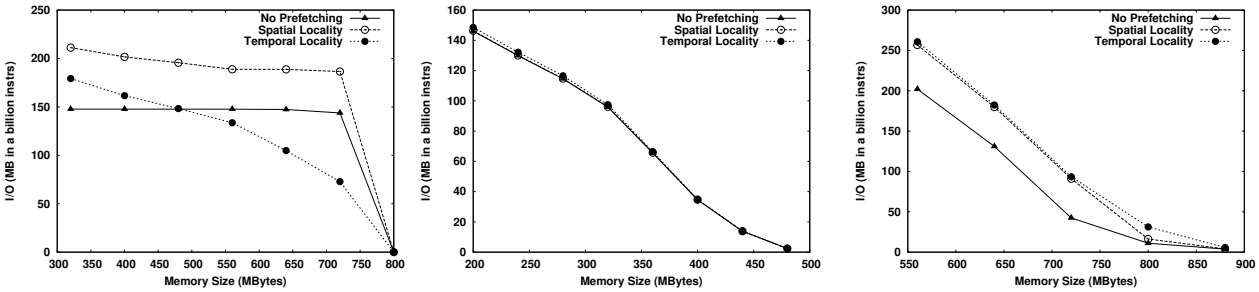
For a selected set of applications, we show the effects of prefetching on projected execution time and on required I/O bandwidth for both page-in and page-out operations in Figures 8 and 9, respectively. The rest of the applications we examined perform similarly to the ones of shown here, and are, again, classified based on similarity and listed in parenthesis in the figures.

For the spatial locality-based policy, we set the initial prefetching window, w, to 64, which can dynamically grow depending on achieved precision. For the temporal locality-



(a) LU Non-cont. (MMCubing, and MrBayes) (b) MG (SP, LU cont., BT, FFT, and Ocean) (c) SPECJbb (FMM)

Figure 8. The effect of prefetching on the projected execution time. In parenthesis are applications which present similar behavior.



(a) LU Non-cont. (MMCubing, and MrBayes) (b) MG (SP, LU cont., BT, FFT, and Ocean) (c) SPECJbb (FMM)

Figure 9. The effect of prefetching on the required I/O bandwidth. In parenthesis are applications which present similar behavior.

based policy, we set the size of the proximity set for each page to 10 and the scan window size W_{scan} to 64 pages. The depth of the breadth–first traversal in the PPG graph was limited to 3. Finally, for both algorithms we set the size of the pool of the pages that are prefetched, but not accessed yet, to be at most 10% of physical memory.

For many applications, such as `MG` and `FFT`, the spatial locality-based policy is quite effective, both in terms of recall and precision. The temporal locality-based algorithm (that monitors the sequence of the accessed pages) is also able to detect regularity in the access pattern with similar effectiveness. There are applications, such as `LU non-cont.` and `MMCubing`, however, for which the temporal locality-based algorithm significantly outperforms the spatial locality-based one, both in terms of improving performance and being precise. Note that the temporal-locality based approach needs fine-grained information on the sequence of page accesses, which in our setup is produced by PATH. Remarkably, temporal locality-based prefetching *reduces* the I/O bandwidth requirements for `LU non-cont.` because artificially touching pages in the prefetched set prevents them from being replaced. Finally, for some applications such as `SPECJbb`, neither prefetching algorithm is effective.

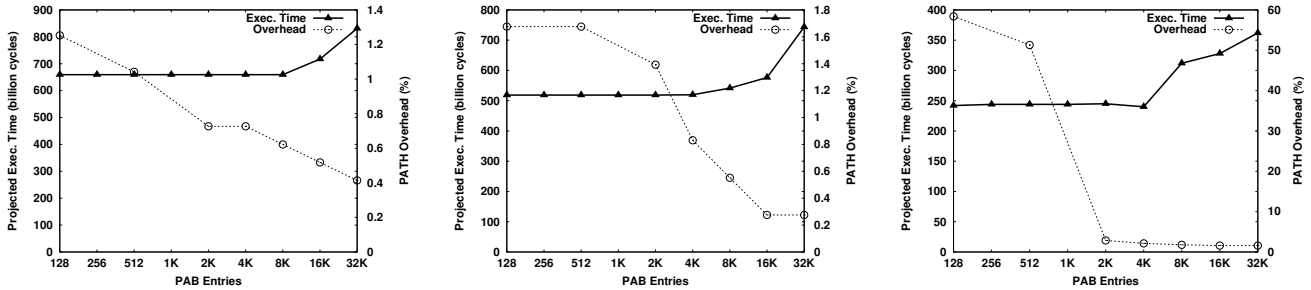
6.4 Effect of PAB Size

For some of the applications that benefit from fine-grained page access information, we evaluate the effect of different PAB sizes on the projected execution time and the runtime overhead. Figures 10 and 11 show this effect for page replacement and prefetching algorithms, respectively. In these experiments, we vary the size of the PAB from 128 to 32K entries. As the PAB size increases, we expect that an increased number of page accesses are filtered by PATH and thus the page access information generated becomes less accurate. At the same time, we expect processing overhead to decrease as fewer page accesses are recorded.

As we see in these graphs, runtime overhead drops significantly as PAB size increases. At the same time, the projected execution time does not seem to be very sensitive as the PAB size is increased from 128 to 2K entries. One exception is `FFT` with LIRS (shown in Figure 1). Overall, a 2K-entry PAB seems to be a good tradeoff between overhead and accuracy.

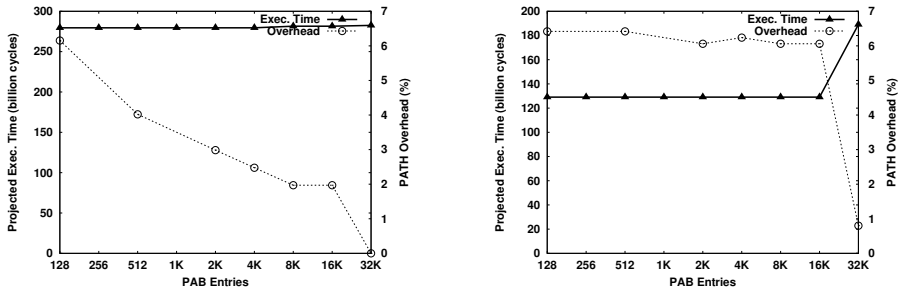
6.5 Analysis of Overhead

In this section, we compare PATH's runtime overhead to the software-only approach. To measure PATH's basic overhead, we *emulated* exceptions generated by PATH in a real envi-



(a) LU Contiguous LIRS (575 MB) (b) Ocean Contiguous LIRS (780 MB) (c) FFT Region-Specific Replacement (576 MB)

Figure 10. The effect of PAB size on the projected execution time and runtime overhead for page replacement algorithms.



(a) MMCubing Prefetching (234 MB) (b) LU Non Contiguous Prefetching (312 MB)

Figure 11. The effect of PAB size on the projected execution time and runtime overhead for prefetching algorithms.

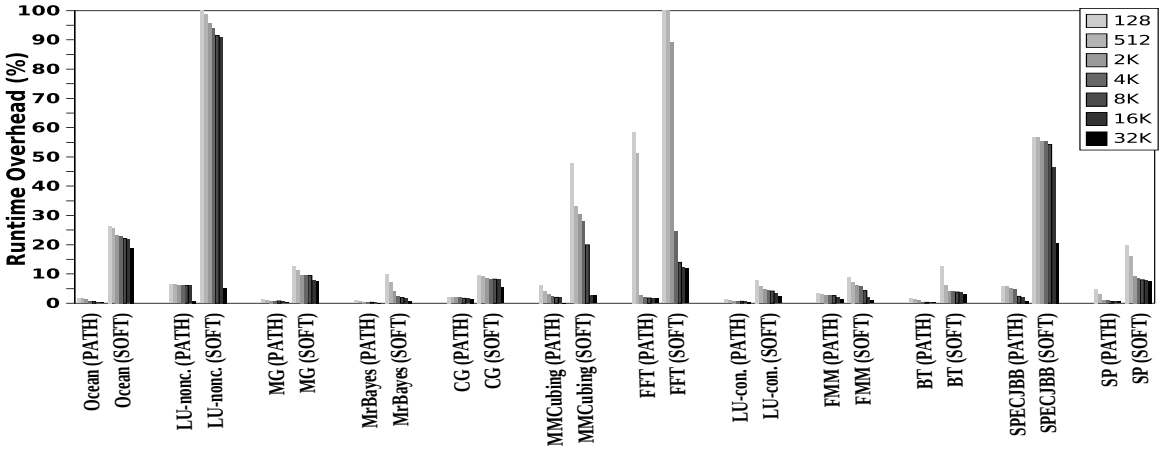


Figure 12. Runtime overhead of PATH-generated information compared to the software-only approach (SOFT). To help visualize the comparison, all runtime overhead numbers larger than 100% are truncated.

ronment using a 1.5GHz AMD Athlon processor. For each application, we collected a trace of PAL overflow exceptions along with the content of the PAL at the time of exception. Each overflow event is time-stamped using the number of instructions retired since the start of the application. We then *replayed* these traces by artificially generating exceptions at the same rate as in the trace by using hardware performance counter overflow exceptions. At each exception, we read the contents of the PAL from the trace and updated the LRU stack and MRC data structures. To calculate the overhead, we measure the total number of CPU cycles needed to execute a certain number of application instructions (e.g. a few tens of billions), with and without PATH exceptions.

The software-only approach was implemented in Linux-2.6.15. We measure only the cost of maintaining the active set which includes the cost of extra page protection faults, page table walks to set the protection bits, flushing the corre-

sponding TLB entries, and occasionally trimming the active set using CLOCK.

Figure 12 shows the runtime overhead of both PATH and the software-only approach across the selected set of applications, as a function of active set size (PAB size in PATH). There are a number of important observations. First, the overhead of the software-only approach is high for a number of applications (e.g., `FFT`, `LU-nonc.`, `MMCubing` and `SPECJbb`) even with a fairly large active set size. Second, the runtime overhead of PATH is very small in all applications if a large PAB (e.g., 32K) is used. For the designed 2K size, the overhead of the PATH remains less than 3% in all but two applications (`LU-nonc.`, and `SPECJbb` for both of which the overhead is less than 6% with a 2K-entry PAB). Such a small overhead is easily paid off by the substantial performance improvement achieved by the PATH-generated information when the system is under memory pressure. Note that the OS can turn off PATH when the system is not under memory pressure, and as a result there will not be any unwanted runtime overhead.

7. Related Work

Zhou *et al.* [33] suggest the use of a custom-designed hardware monitor on memory bus to efficiently calculate MRC online. In their approach much of the overhead of computing MRC can be avoided by offloading to hardware almost completely. In contrast, we argue in favor of having a simpler hardware that provides lower-level but more generic information about page accesses that can be used to solve many problems including the memory allocation problem. We have shown that with the use of fine-grained page access information the operating system can make better decisions on at least three different problems. In terms of hardware resources required, the data structures in PATH are simpler and smaller, and unlike the MRC monitor in Zhou *et al.*'s approach, do not grow proportionally with the size of system physical memory.

Also, *Cooperative Robust Automatic Memory Management* (CRAMM) collects detailed memory reference information to be used to adjust the heap size of a Java virtual machine dynamically in order to prevent a severe performance drop during garbage collection due to paging [30]. The authors have used the software-only approach to track MRC in order to predict memory usage and adjust the JVM heap size accordingly. To reduce overhead, CRAMM dynamically adjusts the size of the active set by monitoring runtime overhead. Such an approach is presumably effective in tracking MRC for JVM's heap size. However, our results show that for many memory intensive applications, increasing the size of the active set will result in significant performance degradation of memory management algorithms.

Tracking memory accesses at the hardware level has been suggested by other researchers, although to address different problems. For instance, Qureshi et al. [24] suggested the use of hardware *utility monitors* to monitor memory accesses solely to compute MRC at the granularity of individual CPU cache lines.

8. Concluding Remarks

Traditionally, operating systems track application memory accesses either by monitoring page faults or by periodically scanning page table entries. With this approach, important information on the reuse distance and temporal proximity of virtual page accesses that can be used for improving memory management algorithms is unavailable. Previous work has suggested the use of a purely software-based approach that uses virtual page protection to track page accesses more accurately. While this software-based approach is effective for some applications, for many applications it incurs unacceptably high overhead.

In this paper, we proposed novel Page Access Tracking Hardware (PATH) that records page access sequences in a relatively accurate yet efficient way. We showed how the operating system can exploit the information provided by PATH to improve memory management in three different ways: adaptive page replacement, process memory allocation, and virtual memory prefetching. Our experimental analysis showed that with this hardware support, significant performance improvements, as high as 500%, can be achieved for applications under memory pressure. Unlike software-only approaches, the runtime overhead of PATH remains small (under 3%-6%) across a wide range of applications.

We believe that additional uses of information provided by PATH will become apparent over time, as we experiment with a wider variety of memory intensive applications. Two possible ideas are super page management and page placement in a NUMA architecture.

An important extension is to explore the use of PATH in a multiprocessor setup. There are important open issues, such as how to collectively use PATH traces of parallel applications that are generated on multiple processors. Similarly, work needs to be done in perfecting PATH support for multithreaded applications. Currently, the PATH trace generated for an application running on a CPU is processed into a single LRU stack or the Page Proximity Graph. If the application is multithreaded, this approach results in intermingling traces of several threads into a single aggregate data structure. As a result, important information about both reuse distance and temporal proximity of page accesses on a per thread basis is lost. To solve this problem, simple extensions can be made to the software layer to keep track of multiple LRU stacks on a per thread basis.

References

[1] D. A. Bader, U. Roshan, and A. Stamatakis. Computational grand challenges in assembling the tree of life: Problems and solutions. *Proc. of ACM/IEEE conference on Supercomputing (SC), tutorial session*, 2005.

[2] S. Bansal and D. S. Modha. CAR: Clock with adaptive replacement. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, 2004.

[3] Bochs. An open source IA-32 emulator. *http:// bochs.sourceforge.net*.

[4] A. D. Brown and T. C. Mowry. Taming the memory hogs: Using compiler-inserted releases to manage physical memory intelligently. In *Proc. of the Symposium on Operating System Design and Implementation (OSDI)*, San Diego, CA, 2000.

[5] R. W. Carr and J. L. Hennessy. WSCLOCK: a simple and effective algorithm for virtual memory management. In *Proc. of the 8th ACM symposium on Operating systems principles, (SOSP)*, Pacific Grove, CA, 1981.

[6] J. Chapin. A fresh look at memory hierarchy management. In *Proc. of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, page 130, 1997.

[7] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *Proc. of the 2000 ACM SIGMETRICS*, Santa Clara, CA, 2000.

[8] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proc. of the 8th conference on Visualization '97 (VIS)*, 1997.

[9] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. In *Proc. of ACM SIGMETRICS*, Seattle, WA, 1997.

[10] G. Gniady, A. R. Butt, and Y. C. Hu. Program-counter-base pattern classification in buffer caching. In *Proc. of the 6th Symp. on Operating System Design and Implementation(OSDI)*, San Francisco, CA, 2004.

[11] Illimine. An open–source data mining toolset. *http:// illimine.cs.uiuc.edu*.

[12] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: an effective improvement of the clock replacement. In *Proc. of the Usenix Technical Conference (USENIX'05)*, Anaheim, CA, 2005.

[13] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Performance Evaluation Review*, 30(1), 2002.

[14] T. Johnson and D. Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *Proc. of the 20th International Conference on Very Large Databases (VLDB)*, Santiago, Chile, 1994.

[15] D. Joseph and D. Grunwald. Prefetching using markov predictors. *IEEE Trans. on Computers*, 48(2):121–133, 1999.

[16] S. F. Kaplan, L. A. McGeoch, and M. F. Cole. Adaptive caching for demand prepaging. In *Proc. of the 3rd International Symposium on Memory Management (ISMM)*, Berlin, Germany, 2002.

[17] Y. H. Kim, M. D. Hill, and D. A. Wood. Implementing stack simulation for highly-associative memories. In *Proc. of the 1991 ACM SIGMETRICS*, San Diego, CA, 1991.

[18] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *Proc. of the 2005 Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, 2005.

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

[20] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, 2003.

[21] MrBayes. Bayesian inference of phylogeny. *http:// mrbayes.csit.fsu.edu*.

[22] NASA Advanced Supercomputing. NAS Parallel Benchmarks. *http://www.nas.nasa.gov/Software/NPB/*.

[23] V. Phalke and B. Gopinath. An inter-reference gap model for temporal locality in program behavior. In *Proc. of the ACM SIGMETRICS*, Ottawa, Canada, 1995. ACM Press.

[24] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. of the 39th MICRO Symposium*, pages 423–432, Washington, DC, USA, 2006.

[25] A. Saulsbury, F. Dahlgren, and P. Stenstrom. Recency-based TLB preloading. In *Proc. of the 27th Intl. Symposium on Computer Architecture (ISCA)*, Vancouver, Canada, 2000.

[26] Y. Smaragdakis, S. Kaplan, and P. Wilson. The EELRU adaptive replacement algorithm. *Performance Evaluation*, 53(2):93–123, 2003.

[27] Standard Performance Evaluation Corporation (SPEC). SPECjbb2000. *http://www.spec.org/jbb2000*.

[28] M. Vilayannur, A. Sivasubramaniam, and M. Kandemir. Proactive page replacement algorithm for scientific applications: A characterization. In *Proc. IEEE Intl. Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, TX, 2005.

[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. *SIGARCH Computer Architecture News*, 23(2):24–36, 1995.

[30] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Proc. of the Symposium on Operating System Design and Implementation (OSDI)*, Seattle, WA, 2006.

[31] Y. Zhang, F. N. Abu-Khzam, N. E. Baldwin, E. J. Chesler, M. A. Langston, and N. F. Samatova. Genome-scale computational approaches to memory-intensive applications in systems biology. In *Proc. of the ACM/IEEE conference on Supercomputing (SC)*, Seattle, WA, 2005.

[32] F. Zhou, R. von Behren, and E. Brewer. AMP: Program context specific buffer caching. In *Proc. of the USENIX Technical Conference (USENIX'05)*, Anaheim, CA, 2005.

[33] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proc. of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, 2004.