Micro-Pages: Increasing DRAM Efficiency with Locality-Aware Data Placement

Kshitij Sudan Niladrish Chatterjee David Nellans Manu Awasthi
Rajeev Balasubramonian Al Davis

School of Computing
University of Utah, Salt Lake City
{kshitij, nil, dnellans, manua, rajeev, ald}@cs.utah.edu

Abstract

Power consumption and DRAM latencies are serious concerns in modern chip-multiprocessor (CMP or multi-core) based compute systems. The management of the DRAM row buffer can significantly impact both power consumption and latency. Modern DRAM systems read data from cell arrays and populate a row buffer as large as 8 KB on a memory request. But only a small fraction of these bits are ever returned back to the CPU. This ends up wasting energy and time to read (and subsequently write back) bits which are used rarely. Traditionally, an open-page policy has been used for uni-processor systems and it has worked well because of spatial and temporal locality in the access stream. In future multi-core processors, the possibly independent access streams of each core are interleaved, thus destroying the available locality and significantly under-utilizing the contents of the row buffer. In this work, we attempt to improve row-buffer utilization for future multi-core systems.

The schemes presented here are motivated by our observations that a large number of accesses within heavily accessed OS pages are to small, contiguous "chunks" of cache blocks. Thus, the co-location of chunks (from different OS pages) in a row-buffer will improve the overall utilization of the row buffer contents, and consequently reduce memory energy consumption and access time. Such co-location can be achieved in many ways, notably involving a reduction in OS page size and software or hardware assisted migration of data within DRAM. We explore these mechanisms and discuss the trade-offs involved along with energy and performance improvements from each scheme. On average, for applications with room for improvement, our best performing scheme increases performance by 9% (max. 18%) and reduces memory energy consumption by 15% (max. 70%).

Categories and Subject Descriptors B.3.2 [*Memory Structures*]: Design Styles–Primary Memory

General Terms Design, Performance, Experimentation

Keywords DRAM Row-Buffer Management, Data Placement

1. Introduction

Main memory has always been a major performance and power bottleneck for compute systems. The problem is exacerbated by a recent combination of several factors - growing core counts for CMPs [5], slow increase in pin count and pin bandwidth of DRAM devices and microprocessors [28], and increasing clock frequencies of cores and DRAM devices. Power consumed by memory has increased substantially and datacenters now spend up to 30% of the total power consumption of a blade (motherboard) in DRAM memory alone [8]. Given the memory industry's focus on cost-per-bit and device density, power density in DRAM devices is also problematic. Further, modern and future DRAM systems will see a much smaller degree of locality in the access stream because requests from many cores will be interleaved at a few memory controllers. In systems that create memory pools shared by many processors [37, 38], locality in the access stream is all but destroyed.

In this work, our focus is to attack the dual problems of increasing power consumption and latency for DRAM devices. We propose schemes that operate within the parameters of existing DRAM device architectures and JEDEC signaling protocols. Our approach stems from the observation that accesses to heavily referenced OS pages are clustered around a few cache blocks. This presents an opportunity to co-locate these clusters from different OS pages in a single row-buffer. This leads to a dense packing of heavily referenced blocks in a few DRAM pages. The performance and power improvements due to our schemes come from improved row-buffer utilization that inevitably leads to reduced energy consumption and access latencies for DRAM memory systems.

We propose the co-location of heavily accessed clusters of cache blocks by controlling the address mapping of OS pages to DIMMs/DRAM devices. We advance two schemes which modify address mapping by employing software and hardware techniques. The software technique relies on reducing the OS page size, while the hardware method employs a new level of indirection for physical addresses allowing a highly flexible data mapping.

Our schemes can easily be implemented in the OS or the memory controller. Thus, the relative inflexibility of device architectures and signaling standards does not preclude these innovations. Furthermore, our mechanisms also fit nicely with prior work on memory controller scheduling policies, thus allowing additive improvements. Compared to past work on address mapping [39, 59], our schemes take a different approach to data mapping and are orthogonal to those advanced by prior work. This prior work on address mapping aimed at reducing row-buffer conflict as much as possible using cache-line or page interleaved mapping of data to DRAM devices. It did not however address inefficiency in data access from a row-buffer itself. If cache blocks within a page are ac-

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

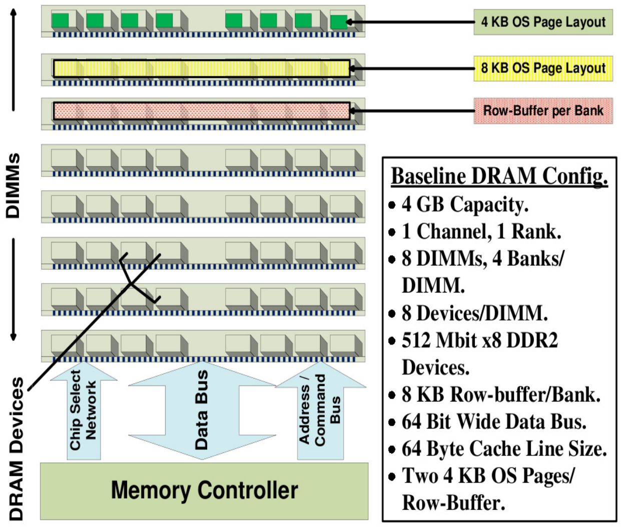

Figure 1. Baseline DRAM Setup and Data Layout.

cessed such that only a few blocks are ever touched, none of these prior schemes [39, 59], can improve row-buffer utilization. This fact, coupled with the increasing row-buffer sizes in newer DRAM devices, is the motivation for our proposals.

Of the signaling standards (JEDEC DDRx [30] and Rambus [16]) prominently used today, we focus on the more popular JEDEC-standard DDRx-based systems, and all our discussions for the rest of the paper pertain to the same. The paper is organized as follows. We briefly discuss the DRAM access mechanism in Section 2 along with a baseline architecture and motivational results. Our proposed schemes are described in Section 3 and evaluated in Section 4. A summary of related work appears in Section 5 and conclusions discussed in Section 6.

2. Background

In this section, we first describe the basic operation of DRAM based memories and mapping of data to DRAM devices. This is followed by results that motivate our approach.

2.1 DRAM Basics

Modern DDRx systems [29] are organized as modules (DIMMs), composed of multiple devices, each of which is an individually packaged integrated circuit. The DIMMs are connected to the memory controller via a data bus and other command and control networks (Figure 1). DRAM accesses are initiated by the CPU requesting a cache line worth of data in the event of last-level cache miss. The request shows up at the memory controller which converts the request into carefully orchestrated commands for DRAM access. Modern memory controllers are also responsible for scheduling memory requests according to policies designed to reduce access times for a request. Using the physical address of the request, the memory controller typically first selects the channel, then the DIMM, and then a rank within a DIMM. Within a rank, DRAM devices work in unison to return as many bits of data as the width of the channel connecting the memory controller and the DIMM.

Each DRAM device is arranged as multiple banks of mats - a grid-like array of cells. Each cell comprises of a transistor-capacitor pair to store a bit of data. These cells are logically addressable with a row and column address pair. Accesses within a device begin with first selecting a bank and then a row. This reads an entire row of bits (whose address is specified by the Row Access Strobe (RAS) command) to per-bank sense amplifiers and latches that serve as the row-buffer. Then a Column Access Strobe (CAS) command selects a column from this buffer. The selected bits from each device are then aggregated at the bank level and sent to the memory controller over the data bus. A critical fact to note here is that once a RAS command is issued, the row-buffer holds a large number of bits that have been read from the DRAM cells. For the DDR2 memory system that we model for our experiments, the number of bits in a per-bank row-buffer is 64 K bits, which is typical of modern memory configurations. To access a 64 byte cache line, a row-buffer is activated and 64 K bits are read from the DRAM cells. Therefore, less than 1% of data in a bank's row-buffer is actually used to service one access request.

On every access, a RAS activates wordlines in the relevant mats and the contents of the cells in these rows are sensed by bitlines. The values are saved in a set of latches (the row-buffer). These actions are collectively referred to as the activation of the row buffer. This read operation is a destructive process and data in the row-buffer must be written back after the access is complete. This write-back can be on the critical path of an access to a new row. The activation of bitlines across several DRAM devices is the biggest contributor to DRAM power. To reduce the delays and energy involved in row buffer activation, the memory controller adopts one of the following row buffer management policies:

- *Open-page policy:* Data in row-buffer is not written back after the access is complete.
- *Close-page policy:* Data is written back immediately after the access is complete.

The open-row policy is based on an optimistic assumption that some accesses in the near future will be to this open page - this amortizes the mat read energy and latency across multiple accesses. State-of-the-art DRAM address mapping policies [39, 59] try to map data such that there are as few row-buffer conflicts as possible. Page-interleaved schemes map contiguous physical addresses to the same row in the same bank. This allows the open-page policy to leverage spatial and temporal locality of accesses. Cache-line interleaved mapping is used with multi-channel DRAM systems with consecutive cache lines mapped to different rows/banks/channels to allow maximum overlap in servicing requests.

On the other hand, the rationale behind close-page policy is driven by the assumption that no subsequent accesses will be to the same row. This is focused towards hiding the latency of write-back when subsequent requests are to different rows and is best suited for memory access streams that show little locality - like those in systems with high processor counts.

Traditional uni-processor systems perform well with an open-page policy since the memory request stream being generated follows temporal and spatial locality. This locality makes subsequent requests more likely to be served by the open-page. However with multi-core systems, the randomness of requests has made open-page policy somewhat ineffective. This results in low row-buffer utilization. This drop in row-buffer utilization (hit-rate) is quantified in Figure 2 for a few applications. We present results for 4 multi-threaded applications from the PARSEC [9] suite and one multi-programmed workload mix of two applications each from SPEC CPU2006 [24] and BioBench [4] suites (*lbm*, *mcf*, and *fasta-dna*, *mummer*, respectively). Multi-threaded applications were first run with a single thread on a 1-core system and then with four threads on a 4-core CMP. Each application was run for 2 billion instructions, or the end of parallel-section, whichever occurred first. Other simulation parameters are described in detail in Section 4.

We observed that the average utilization of row-buffers dropped in 4-core CMP systems when compared to a 1-core system. For

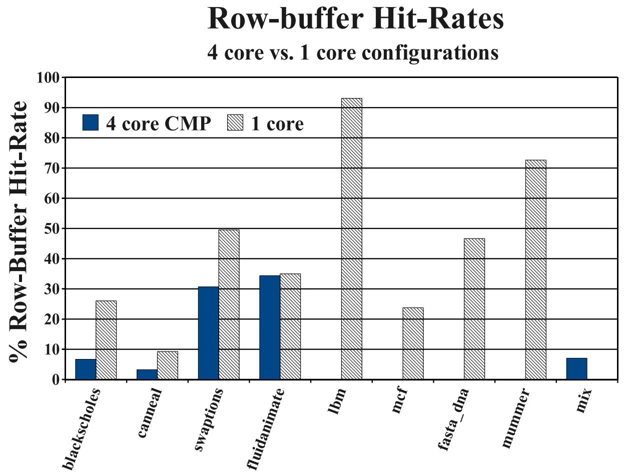

Figure 2. Row-buffer Hit-Rates for 1 and 4-Core Configurations

multi-threaded applications, the average row-buffer utilization dropped from nearly 30% in the 1-core setup to 18% for the 4-core setup. For the 4 single-threaded benchmarks in the experiment (*lbm*, *mcf*, *fasta-dna*, and *mummer*), the average utilization was 59% when each benchmark was run in isolation. It dropped to 7% for the multi-programmed workload mix created from these same applications. This points towards the urgent need to address this problem since application performance is sensitive to DRAM access latency, and DRAM power in modern server systems accounts for nearly 30% of total system power [8].

Past work has focused on increasing energy efficiency by implementing a power-aware memory allocation system in the OS [25], decreasing the row-buffer size [60], interleaving data to reduce row-buffer conflicts [39, 59], coalescing short idle periods between requests to leverage power-saving states in modern DRAM devices [26], and on scheduling requests at the memory controller to reduce access latency and power consumption [21, 44, 45, 60, 62, 63].

In this work, we aim to increase the row-buffer hit rates by influencing the placement of data blocks in DRAM - specifically, we co-locate frequently accessed data in the same DRAM rows to reduce access latency and power consumption. While application performance has been shown to be sensitive to DRAM system configuration [17] and mapping scheme [39, 59], our proposals improve row-buffer utilization and work in conjunction with any mapping or configuration. We show that the strength of our schemes lies in their ability to tolerate random memory request streams, like those generated by CMPs.

2.2 Baseline Memory Organization

For our simulation setup, we assumed a 32-bit system with 4 GB of total main memory. The 4 GB DRAM memory capacity is spread across 8 non-ECC, unbufferred DIMMs as depicted in Figure 1. We use Micron MT47H64M8 [41] part as our DRAM device - this is a 512 Mbit, x8 part. Further details about this DRAM device and system set-up are summarized in Table 1 in Section 4. Each row-buffer for this setup is 8 KB in size, and since there are 4 banks/device there are 4 row-buffers per DIMM. We model a DDR2-800 memory system, with DRAM devices operating at 200 MHz, and a 64-bit wide data bus operating at 400 MHz. For a 64 byte sized cache line, it takes 8 clock edges to transfer a cache line from the DIMMs to the memory controller.

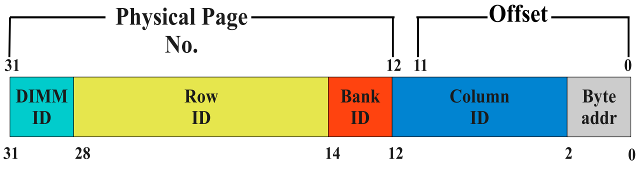

Figure 3. Baseline DRAM Address Mapping

2.2.1 Baseline DRAM addressing

Using the typical OS page size of 4 KB, the baseline page-interleaved data layout is as shown in Figure 3. Data from a page is spread across memory such that it resides in the same DIMM, same bank, and the same row. This mapping is similar to that adopted by Intel 845G Memory Controller Hub for this configuration [27, 29, 57], and affords simplicity in explaining our proposed schemes. We now describe how the bits of a physical address are interpreted by the DRAM system to map data across the storage cells. For a 32 bit physical address (Figure 3), the low order 3 bits are used as the byte address, bits 3 through 12 provide the column address, bits 13 and 14 denote the bank, bits 15 through 28 provide the row I.D, and the most significant 3 bits indicate the DIMM I.D. An OS page therefore spans across all the devices on a DIMM, and along the same row and bank in all the devices. For a 4 KB page size, a row-buffer in a DIMM holds two entire OS pages, and each device on the DIMM holds 512 bytes of that page's data.

2.3 Motivational Results

An interesting observation for DRAM memory accesses at OS page granularity is that for most applications, accesses in a given time interval are clustered around few contiguous cache line sized blocks in the most referenced OS pages. Figures 4 and 5 show this pattern visually for a couple of SPEC CPU2006 applications. The X-axis shows the sixty-four 64 byte cache blocks in a 4 KB OS page, and the Z-axis shows the percent of total accesses to each block within that page. The Y-axis plots the most frequently accessed OS pages in sorted order.

We present data for pages that account for approximately 25% of total DRAM requests served for a 2 billion instruction period. These experiments were run with 32 KB, 2-way split L1 I and D-cache, and 128 KB, 8-way L2 cache for a single core setup. To make sure that results were not unduly influenced by configuration parameters and not limited to certain applications, we varied all the parameters and simulated different benchmark applications from PARSEC, SPEC, NPB [7], and BioBench benchmark suites. We varied the execution interval over various phases of the application execution, and increased cache sizes and associativity to make sure we reduced conflict misses. Figures 4 and 5 show 3D graphs for only two representative experiments because the rest of the workloads exhibited similar patterns. The accesses were always clustered around a few blocks in a page, and very few pages accounted for most accesses in a given interval (less than 1% of OS pages account for almost 1/4th of the total accesses in the 2 billion instruction interval; the exact figures for the shown benchmarks are: *sphinx3* - 0.1%, *gemsFDTD* - 0.2%).

These observations lead to the central theme of this work - *co-locating clusters of contiguous blocks with similar access counts, from different OS pages, in a row-buffer to improve its utilization.*

In principle, we can imagine taking individual cache blocks and co-locating them in a row-buffer. However, such granularity would be too fine to manage a DRAM memory system and the overheads would be huge. For example, in a system with 64 B wide cache lines and 4 GB of DRAM memory, there would be nearly 67 million

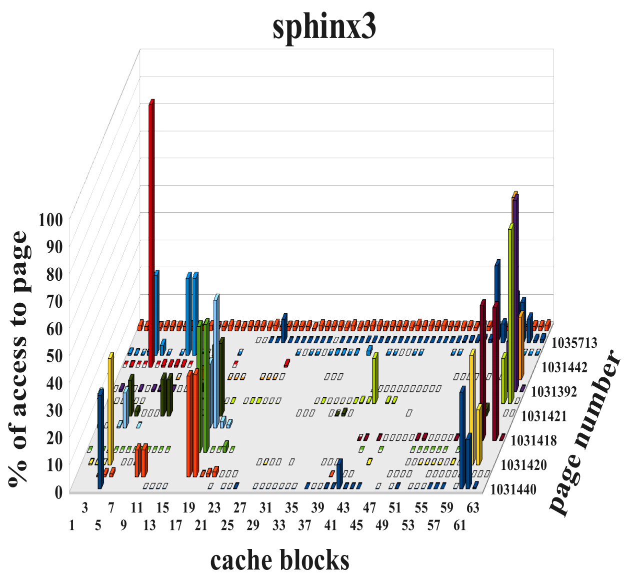

Figure 4. sphinx3

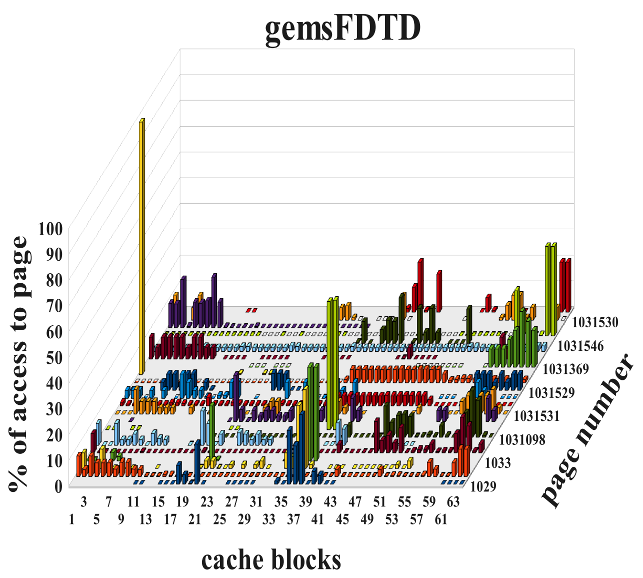

Figure 5. gemsFDTD

blocks to keep track of! To overcome these overheads, we instead focus on dealing with clusters of contiguous cache blocks. We call these clusters *"micro-pages"*.

3. Proposed Mechanisms

The common aspect of all our proposed innovations is the identification and subsequent co-location of frequently accessed micro-pages in row-buffers to increase row-buffer hit rates. We discuss two mechanisms to this effect - decreasing OS page size and performing page migration for co-location, and hardware assisted migration of segments of a conventional OS page (hereafter referred to as a micro-page). As explained in Sections 1 and 2, row-buffer hit rates can be increased by populating a row-buffer with those chunks of a page which are frequently accessed in the same window of execution. We call these chunks from different OS pages "hot" micro-pages if they are frequently accessed during the *same* execution epoch.

For all our schemes, we propose a common mechanism to identify hot micro-pages at run-time. We introduce counters at the memory controller that keep track of accesses to micro-pages within different OS pages. For a 4 KB OS page size, 1 KB micro-page size, and assuming application memory footprint in a 50 million cycle epoch to be 512 KB (in Section 4 we show that footprint of hot micro-pages is actually lower than 512 KB for most applications), the total number of such counters required is 512. This is a small overhead in hardware and since the update of these counters is not on the critical path while scheduling requests at the memory controller, it does not introduce any latency overhead. However, an associative look-up of these counters (for matching micro-page number) can be energy inefficient. To mitigate this, techniques like hash-based updating of counters can be adopted. Since very few micro-pages are touched in an epoch, we expect the probability of hash-collision to be small. If the memory footprint of the application exceeds 512 KB, then some errors in the estimation of hot micro-pages can be expected.

At the end of an epoch, an OS daemon inspects the counters and the history from the preceding epoch to rank micro-pages as "hot" based on their access counts. The daemon then selects the micro-pages that are suitable candidates for migration. Subsequently, the specifics of the proposed schemes take over the migrations and associated actions required to co-locate the hot micro-pages. By using the epoch based statistics collection, and evaluating hotness of a page based on the preceding epoch's history, this scheme implicitly uses both temporal and spatial locality of requests to identify hot micro-pages. The counter values and the preceding epoch's history are preserved for each process across a context switch. This is a trivial modification to the OS' context switching module that saves and restores application context. It is critical to have such a dynamic scheme so it can easily adapt to varying memory access patterns. Huang et al. [25] describe how memory access patterns are not only continuously changing within an application, but also across context switches, thus necessitating a dynamically adaptive scheme. By implementing this in software, we gain flexibility not afforded by hardware implementations.

3.1 Proposal 1 - Reducing OS Page Size (ROPS)

In this section we first describe the basic idea of co-locating hot micro-pages by reducing the OS page size. We then discuss the need to create superpages from micro-pages to reduce bookeeping overhead and mitigate the reduced TLB reach due to smaller page size.

Basic Idea: In this scheme, our objective is to reduce the OS page size such that frequently accessed contiguous blocks are clustered together in the new reduced size page (a micro-page). We then migrate hot micro-pages using DRAM copy to co-locate frequently accessed micro-pages in the same row-buffer. The innovation here is to make the OS' Virtual Address (VA) to Physical Address (PA) mapping cognizant of row-buffer utilization. This is achieved by modifying the original mapping assigned by the OS' underlying physical memory allocation algorithm such that hot pages are mapped to physical addresses that are co-located in the same row-buffer. Every micro-page migration is accompanied by an associated TLB shoot-down and change in its page table entry.

Baseline Operation: To understand the scheme in detail, let's follow the sequence of events from the first time some data is accessed by an application. The application initially makes a request to the OS to allocate some amount of space in the physical memory. This request can be explicit via calls to *malloc( )*, or implicit via compiler indicated reservation in the stack. The OS in turn assigns a virtual to physical page mapping for this request by creating a new page table entry. Subsequently, when the data is first accessed by the application, a TLB miss fetches the appropriate entry from the page table. The data is then either fetched from the disk into the appropriate physical memory location via DMA copy, or the OS could copy another page (copy-on-write), or allocate a new empty page. An important point to note from the perspective of our schemes is that in either of these three cases (data fetch from the disk, copy-on-write, allocation of an empty frame), the page table is accessed. We will discuss the relevance of this point shortly.

Overheads: In our scheme, we suggest a minor change to the above sequence of events. We propose reducing the page size across the entire system to 1 KB, and instead of allocating one page table entry the first time the request is made to allocate physical memory, the OS creates four page table entries (each equal to page size of 1 KB). We leverage the "reservation-based" [46, 53] allocation approach for 1 KB base-pages, i.e., on first-touch, contiguous 1 KB virtual pages are allocated to contiguous physical pages. This ensures that the overhead required to move from a 1 KB base-page to a 4 KB superpage is only within the OS and does not require DRAM page copy. These page table entries will therefore have contiguous virtual and physical addresses and will ensure easy creation of superpages later on. The negative effect of this change is that the page table size will increase substantially, at most by $4X$ compared to a page table for 4 KB page size. However, since we will be creating superpages for most of the 1 KB micro-pages later (this will be discussed in more detail shortly), the page table size will shrink back to a size similar to that for a baseline 4 KB OS page size. Note that in our scheme, 4 KB pages would only result from superpage creation and the modifications required to the page table have been explored in the literature for superpage creation. The smaller page size also impacts TLB coverage and TLB miss rate, but similar to the discussion above, this impact is minimal if most 1 KB pages are eventually coalesced into 4 KB pages.

The reason to choose 1 KB micro-page size was dictated by a trade-off analysis between TLB coverage and a manageable micro-page granularity. We performed experiments to determine the benefit obtained by having a smaller page size and performance degradation due to drop in TLB coverage. For almost all applications, 1 KB micro-page size offered a good trade-off.

Reserved Space: Besides the above change to the OS' memory allocator, we also reserve frames in the main memory that are never assigned to any application (even the kernel is not mapped to these frames). These reserved frames belong to the first 16 rows in each bank of each DIMM and are used to co-locate hot micro-pages. The total capacity reserved for our setup is 4 MB and is less than 0.5% of the total DRAM capacity. We later present results that show most of the hot micro-pages can be accommodated in these reserved frames.

Actions per Epoch: After the above described one-time actions of this scheme, we identify "hot" micro-pages every epoch using the OS daemon as described earlier. This daemon examines counters to determine the hot micro-pages that must be migrated. If deemed necessary, hot micro-pages are subsequently migrated by forcing a DRAM copy for each migrated micro-page. The OS' page table entries are also changed to reflect this. To mitigate the impact of reduced TLB reach due to smaller page size, we create superpages. Every contiguous group of four 1 KB pages that do not contain a migrated micro-page are promoted to a 4 KB superpage.

Thus the sequence of steps taken for co-locating the micro-pages for this scheme are as follows:

- Look up the hardware counters in the memory controller and designate micro-pages as "hot" by combining this information with statistics for the previous epoch - performed by an OS daemon described earlier.
- To co-locate hot micro-pages, force DRAM copies causing migration of micro-pages. Then update the page table entries to appropriate physical addresses for the migrated micro-pages.
- Finally, create as many superpages as possible.

Superpage Creation: The advantages that might be gained from enabling the OS to allocate physical addresses at a finer granularity may be offset by the penalty incurred due to reduced TLB reach. To mitigate these effects of higher TLB misses we incorporate the creation of superpages [46, 49]. Superpage creation [22, 46, 49, 52] has been extensively studied in the past and we omit the details of those mechanisms here. Note that in our scheme, superpages can be created only with "cold" micro-pages since migrated hot micro-pages leave a "hole" in the 4 KB contiguous virtual and physical address spaces. Other restrictions for superpage creation are easily dealt with as discussed next.

TLB Status Bits: We allocate contiguous physical addresses for four 1 KB micro-pages that were contiguous in virtual address space. This allows us to create superpages from a set of four micro-pages which do not contain a hot micro-page that has been migrated. When a superpage is created, a few factors need to be taken into account, specifically the various status bits associated with each entry in the TLB.

- The included base-pages must have identical protection and access bits since the superpage entry in the TLB has only one field for these bits. This is not a problem because large regions in the virtual memory space typically have the same state of protection and access bits. This happens because programs usually have large segments of densely populated virtual address spaces with similar access and protection bits.
- Base-pages also share the dirty-bit when mapped to a superpage. The dirty-bit for a superpage can be the logical OR of the dirty-bits of the individual base-pages. A minor inefficiency is introduced because an entire 4 KB page must be written back even when only one micro-page is dirty.
- The processor must support a wide range of superpage sizes (already common in modern processors), including having a larger field for the physical page number.

Schemes proposed by Swanson et al. [52] and Fang et al. [22] have also shown that it is possible to create superpages that are non-contiguous in physical address space and unaligned. We mention these schemes here but leave their incorporation as future work.

The major overheads experienced by the ROPS scheme arise from two sources:

- DRAM copy of migrated pages and associated TLB shootdown, and page table modifications.
- Reduction in TLB coverage.

In Section 4, we show that DRAM migration overhead is not large because on an average only a few new micro-pages are identified as hot every epoch and moved. As a result, the major overhead of DRAM copy is relatively small. However, the next scheme proposed below eliminates the above mentioned overheads and is shown to perform better than ROPS. The performance difference is not large however.

3.2 Proposal 2 - Hardware Assisted Migration (HAM)

This scheme introduces a new layer of translation between physical addresses assigned by the OS (and stored in the page table while allocating a new frame in main memory) - and those used by the memory controller to access the DRAM devices. This translation keeps track of the new physical addresses of the hot micro-pages that are being migrated and allows migration without changing the OS' page size, or any page table entries. Since the OS page size is not changed, there is no drop in TLB coverage.

Indirection: In this scheme we propose look-up of a table at the memory controller to determine the new address if the data has been migrated. We organize the table so it consumes acceptable energy and area and these design choices are described subsequently. The address translation required by this scheme is usually not on the critical path of accesses. Memory requests usually wait in the memory controller queues for a long time before being serviced. The above translation can begin when the request is queued and the delay for translation can be easily hidden behind the long wait time. The notion of introducing a new level of indirection has been widely used in the past, for example, within memory controllers to aggregate distributed locations in the memory [11]. More recently, it has been used to control data placement in large last-level caches [6, 13, 23].

Similar to the ROPS schemes, the OS daemon is responsible for selecting hot micro-pages fit for co-location. Thereafter, this scheme performs a DRAM copy of "hot" micro-pages. However, instead of modifying the page table entries as in ROPS, this scheme involves populating a "mapping table" (MT) in the memory controller with mappings from the old to the new physical addresses for each migrated page. Features of this scheme are now described in more detail below.

Request Handling: When a request (with physical address assigned by the OS) arrives at the memory controller, it searches for the address in the MT (using certain bits of the address as described later). On a hit, the new address of the micro-page is used by the memory controller to issue the appropriate commands to retrieve the data from its new location - otherwise the original address is used. The MT look-up happens the moment a request is added to the memory controller queue and does not extend the critical path in the common case because queuing delays at the memory controller are substantial.

Micro-page Migration: Every epoch, micro-pages are rated and selected for migration. Also, the first 16 rows in each bank are reserved to hold the hot micro-pages. The OS's VA to PA mapping scheme is modified to make sure that no page gets mapped to these reserved rows. The capacity lost due to this reservation (4 MB) is less than 0.5% of the total DRAM capacity and in Section 4 we show that this capacity is sufficient to hold almost all hot micro-pages in a given epoch.

At the end of each epoch, hot micro-pages are moved to one of the slots in the reserved rows. If a given micro-page is deemed hot, and is already placed in the reserved row (from the previous epoch), we do not move it. Otherwise, a slot in the reserved row is made empty by first moving the now "cold" micro-page to its original address. The new hot micro-page is then brought to this empty slot. The original address of the cold page being replaced is derived from the contents of the corresponding entry of the MT. After the migration is complete, the MT is updated accordingly. The design choice to not swap the cold and hot micro-pages and allow "holes" was taken to reduce book-keeping overheads and easily locate evicted cold blocks.

This migration also implies that the portion of the original row from which a hot micro-page is migrated now has a "hole" in it. This does not pose any correctness issue in terms of data look-up for a migrated micro-page at its former location since an access to a migrated address will be redirected to its new location by the memory controller.

Mapping Table (MT): The mapping table contains the mapping from the original OS assigned physical address to the new address for each migrated micro-page. The size and organization of the mapping table can significantly affect the energy spent in the address translation process. We discuss the possible organizations for the table in this subsection. As mentioned earlier, we reserve 16 rows of each bank in a DIMM to hold the hot micro-pages. The reservation of these rows implies that the total number of slots where a hot micro-page might be relocated is 4096 (8 DIMMS $*$ 4 banks/DIMM $*$ 16 rows/bank $*$ 8 micro-page per row).

We design the MT as a banked fully-associative structure. The MT has 4096 entries, one for each slot reserved for a hot micro-page. Each entry stores the original physical address for the hot micro-page resident in that slot. The entry is populated when the micro-page is migrated. Since it takes 22 bits to identify each 1 KB micro-page in the 32-bit architecture, the MT requires a total storage capacity of 11 KB.

On every memory request, the MT must be looked up to determine if the request must be re-directed to a reserved slot. The original physical address must be compared against the addresses stored in all 4096 entries. The entry number that flags a hit is then used to construct the new address for the migrated micro-page. A couple of optimizations can be attempted to reduce the energy overhead of the associative search. First, we can restrict a hot micro-page to only be migrated to a reserved row in the same DIMM. Thus, only $1/8^{th}$ of the MT must be searched for each look-up. Such a banked organization is assumed in our quantitative results. A second optimization can set a bit in the migrated page's TLB entry. The MT is looked up only if this bit is set in the TLB entry corresponding to that request. This optimization is left as future work.

When a micro-page must be copied back from a reserved row to its original address, the MT is looked up with the ID (0 to 4095) of the reserved location, and the micro-page is copied into the original location saved in the corresponding MT entry.

4. Results

4.1 Methodology

Our detailed memory system simulator is built upon the Virtutech Simics [2, 40] platform and important parameters of the simulated system are shown in Table 1. Out-of-order timing is simulated using Simics' *sample-micro-arch* module and the DRAM memory sub-system is modeled in detail using a modified version of Simics' *trans-staller* module. It closely follows the model described by Gries in [41]. The memory controller (modeled in *trans-staller*) keeps track of each DIMM and open rows in each bank. It schedules the requests based on open-page and close-page policies. To keep our memory controller model simple, we do not model optimizations that do not directly affect our schemes, like finite queue length, critical-word-first optimization, and support for prioritizing reads. Other major components of Gries' model that we adopt for our platform are: the bus model, DIMM and device models, and most importantly, simultaneous pipelined processing of multiple requests. The last component allows hiding activation and pre-charge latency using pipelined interface of DRAM devices. We model the CPU to allow non-blocking load/store execution to support overlapped processing.

DRAM address mapping parameters for our platform were adopted from the DRAMSim framework [57]. We implemented single-channel basic SDRAM mapping, as found in user-upgradeable memory systems, and it is similar to Intel 845G chipsets' DDR SDRAM mapping [27]. Some platform specific implementation suggestions were taken from the VASA framework [56]. Our DRAM energy consumption model is built as a set of counters that keep track of each of the commands issued to the DRAM. Each

CMP Parameters

ISA	UltraSPARC III ISA
CMP Size and Core Frequency	4-core, 2 GHz
Re-Order-Buffer	64 entry
Fetch, Dispatch, Execute, and Retire	Maximum 4 per cycle
L1 I-cache	32 KB/2-way, private, 1-cycle
L1 D-cache	32KB/2-way, private, 1-cycle
L2 Cache	128 KB/8-way, shared, 10-cycle
L1 and L2 Cache Line Size	64 Bytes
Coherence Protocol	Snooping MESI

DRAM Parameters

DRAM Device Parameters	Micron MT47H64M8 DDR2-800 Timing parameters [41], t$_{CL}$=t$_{RCD}$=t$_{RP}$=20ns(4-4-4 @ 200 MHz) 4 banks/device, 16384 rows/bank, 512 columns/row, 8-bit output/device
DIMM Configuration	8 Non-ECC un-buffered DIMMs, 1 rank/DIMM, 64 bit channel, 8 devices/DIMM
DIMM-Level Row-Buffer Size	8 KB
Active Row-Buffers per DIMM	4
Total DRAM Capacity	512 MBit/device × 8 devices/DIMM × 8 DIMMs = 4 GB

Table 1. Simulator Parameters.

pre-charge, activation, CAS, write-back to DRAM cells, etc. are recorded and total energy consumed reported using energy parameters derived from Micron MT47H64M8 DDR2-800 datasheet [41]. We do not model the energy consumption of the data and command buses as our schemes do not affect them. The energy consumption of the mapping table (MT) is derived from CACTI [43, 54] and is accounted for in the simulations.

Our schemes are evaluated with full system simulation of a wide array of benchmarks. We use multi-threaded workloads from the PARSEC [9], OpenMP version of NAS Parallel Benchmark [7], and SPECJBB [3] suites. We also use the STREAM [1] benchmark as one of our multi-threaded workloads. We use single threaded applications from SPEC CPU 2006 [24] and BioBench [4] suites for our multi-programmed workload mix. While selecting individual applications from these suites, we first characterized applications for their total DRAM accesses, and then selected two applications from each suite that had the highest DRAM accesses. For both multi-threaded and single-threaded benchmarks, we simulate the application for 250 million cycles of execution. For multi-threaded applications, we start simulations at the beginning of the parallel-region/region-of-interest of the application, and for single-threaded we start from 2 billion instructions after the start of the application. Total system throughput for single-threaded benchmarks is reported as weighted speedup [51], calculated as $\sum_{i=1}^{n}(IPC^{i}_{shared}/IPC^{i}_{alone})$, where IPC^{i}_{shared} is the IPC of program i in an "n" core CMP.

The applications from PARSEC suite are configured to run with *simlarge* input set, applications from NPB suite are configured with *Class A* input set and STREAM is configured with an array size of 120 million entries. Applications from SPEC CPU2006 suite were exectued with the *ref* inputs and BioBench applications with the default input set. Due to simulation speed constraints, we only simulated each application for 250 million cycles. This resulted in extremely small working set sizes for all these applications. With 128 KB L1 size and 2 MB L2 size, we observed very high cache hit-rates for all these applications. We therefore had to scale down the L1 and L2 cache sizes to see any significant number of DRAM accesses. While deciding upon the scaled down cache sizes, we chose 32 KB L1 size and 128 KB L2 size since these sizes gave approximately the same L1 and L2 hit-rate as when the applications were run to completion with larger cache sizes. With the scaled down cache sizes, the observed cache hit-rates for all the applications are show in Table 2.

Benchmark	L2 Hit Rate	Input Set
blackscholes	89.3%	simlarge
bodytrack	59.0%	simlarge
canneal	23.8%	simlarge
facesim	73.9%	simlarge
ferret	79.1%	simlarge
freqmine	83.6%	simlarge
streamcluster	65.4%	simlarge
swaptions	93.9%	simlarge
vips	58.45%	simlarge
IS	85.84%	class A
MG	52.4%	class A
mix	36.3%	ref (SPEC), default (BioBench)
STREAM	48.6%	120 million entry array
SPECJBB	69.9%	default

Table 2. L2 Cache Hit-Rates and Benchmark Input Sets.

For all our experiments, we record the row-buffer hit rates, measure energy consumed by the DRAM memory system, and compute the throughput for our benchmarks. All experiments were performed with both First-Come First-Serve (FCFS) and First-Ready First-Come First-Serve (FR-FCFS) memory controller scheduling policies. Only results for the FR-FCFS policy are shown since it is the most commonly adopted scheduling policy. We show results for three types of platforms:

- **Baseline** - This is the case where OS pages are laid out as shown in Figure 1. The OS is responsible for mapping pages to memory frames and since we simulate Linux OS, it relies on the buddy system [33] to handle physical page allocations. (For some applications, we use Solaris OS.)
- **Epoch Based Schemes** - These experiments are designed to model the proposed schemes. Every epoch an OS sub-routine is triggered that reads the access counters at the memory controller and decides if migrating a micro-page is necessary. The mechanism to decide if a micro-page needs to be migrated, and the details on how migrations are affected have already been described in detail in Section 3 for each proposal.
- **Profiled Placement** - This is a two pass experiment designed to quantify an approximate upper-bound on the performance of our proposals. It does so by "looking into the future" to determine the best layout. During the first pass, we create a trace of OS page accesses. For the second pass of the simulation, we pre-process these traces and determine the best possible placement of micro-pages for every epoch. The best placement is

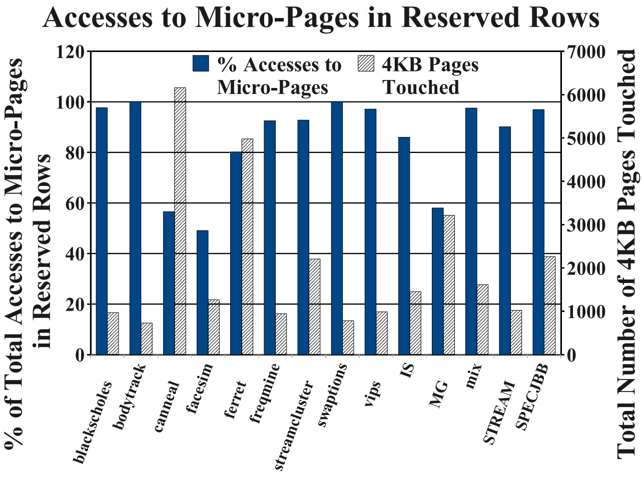

Figure 6. Accesses to Micro-Pages in Reserved DRAM Capacity.

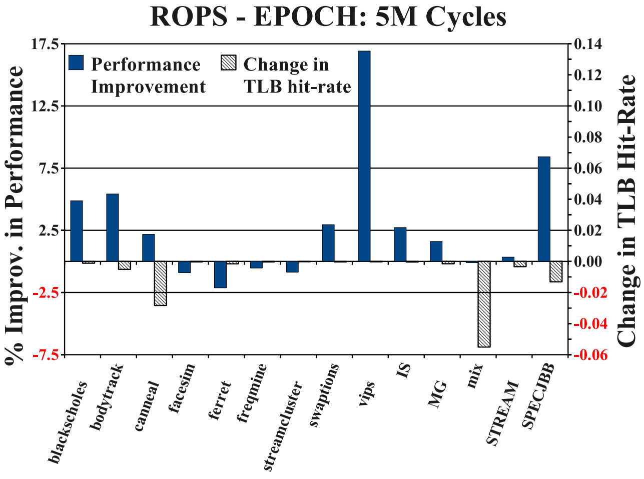

Figure 7. Performance Improvement and Change in TLB Hit-Rates (w.r.t. Baseline) for ROPS with 5 Million Cycle Epoch Length

decided by looking at the associated cost of migrating a page for the respective scheme, and the benefit it would entail. An important fact to note here is that since multi-threaded simulations are non-deterministic, these experiments can be slightly unreliable indicators of best performance. In fact for some of our experiments, this scheme shows performance degradation. However it's still an important metric to determine the approximate upper bound on performance.

For all our experimental evaluations, we assume a constant overhead of 70,000 cycles per epoch to execute the OS daemon and for DRAM data migrations. For a 5 million cycle epoch, this amounts to 1.4% overhead per epoch in terms of cycles. From our initial simulations, we noticed that approximately 900 micro-pages were being moved every epoch for all our simulated applications. From the parameters of our DRAM configuration, this evaluates to nearly 60,000 DRAM cycles for migrations. For all the above mentioned schemes, we present results for different epoch lengths to show the sensitivity of our proposals to this parameter. We assume 4 KB OS pages size with 1 KB micro-page size for all our experiments.

For the ROPS scheme, we assume an additional, constant 10,000 cycle penalty for all 1 KB → 4 KB superpage creations in an epoch. This overhead models the identification of candidate pages and the update of OS book-keeping structures. TLB shoot-down and the ensuing page walk overheads are added on top of this overhead. We do not model superpage creation beyond 4 KB as this behavior is expected to be the same for the baseline and proposed models.

The expected increase in page table size because of the use of smaller 1 KB pages is not modeled in detail in our simulator. After application warm-up and superpage promotion, the maximum number of additional page table entries required is 12,288 on a 4 GB main memory system, a 1.1% overhead in page table size. This small increase in page table size is only expected to impact cache behavior when TLB misses are unusually frequent – which is not the case, thus it does not affect our results unduly.

We chose to model 4 KB page sizes for this study as 4 KB pages are most common in hardware platforms like x86 and x86_64. An increase in baseline page size (to 8 KB or larger) would increase the improvement seen by our proposals, as the variation in sub-page block accesses increase. UltraSPARC and other enterprise hardware often employ a minimum page size of 8 KB to reduce the page table size when addressing large amounts of memory. Thus, the performance improvements reported in this study are likely to be a conservative estimate for some architectures.

4.2 Evaluation

We first show that reserving 0.5% of our DRAM capacity (4 MB, or 4096 1 KB slots) for co-locating hot micro-pages is sufficient in Figure 6. We simulated applications with a 5 million cycle epoch length. For each application, we selected an epoch that touched the highest number of 4 KB pages and plotted the percent of total accesses to micro-pages in the reserved DRAM capacity. The total number of 4 KB pages touched is also plotted in the same figure (right hand Y-axis). For all but 3 applications (*canneal*, *facesim*, and *MG*), on an average 94% of total access to DRAM in that epoch are to micro-pages in the reserved 4 MB capacity. This figure also shows that application footprints per epoch are relatively small and our decision to use only 512 counters at the DRAM is also valid. We obtained similar results with simulation intervals several billions of cycles long.

Next we present results for the reduced OS page size (ROPS) scheme, described in Section 3.1, in Figure 7. The results are shown for 14 applications - eight from PARSEC suite, two from NPB suite, one multi-programmed mix from SPEC CPU2006 and BioBench suites, STREAM, and SPECjbb2005.

Figure 7 shows the performance of the proposed scheme compared to baseline. The graph also shows the change in TLB hit-rates for 1 KB page size compared to 4 KB page size, for a 128-entry TLB (secondary Y-axis, right hand-side). Note that for most of the applications, the change in TLB hit-rates is very small compared to baseline 4 KB pages. This demonstrates the efficacy of superpage creation in keeping TLB misses low. Only for three applications (*canneal*, multi-programmed workload mix, and *SPECjbb*) does the change in TLB hit-rates go over 0.01% compared to baseline TLB hit-rate. Despite sensitivity of application performance to TLB hit-rate, with our proposed scheme both *canneal* and *SPECjbb* show notable improvement. This is because application performance tends to be even more sensitive to DRAM latency. Therefore, for the ROPS proposal, a balance must be struck between reduced DRAM latency and reduced TLB hit-rate. Out of these 14 applications, 5 show performance degradation. This is attributed to the overheads involved in DRAM copies, increased TLB miss-rate, and the daemon overhead, while little performance improvement

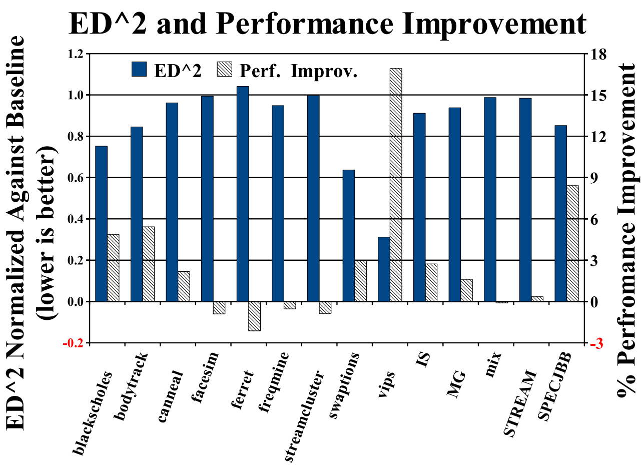

Figure 8. Energy Savings ($E * D^2$) for ROPS with 5 Million Cycle Epoch Length

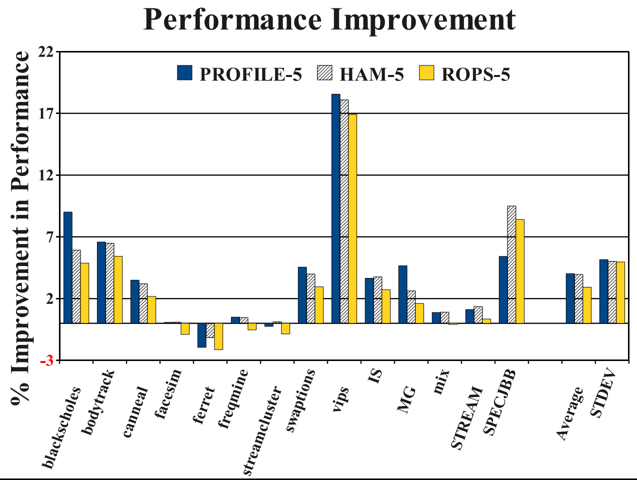

Figure 9. Profile, HAM and ROPS - Performance Improvement for 5 Million Cycle Epoch Length

is gained from co-locating micro-pages. The reason for low performance improvements from co-location is the nature of these applications. If applications execute tight loops with regular access pattern within OS pages, then it is hard to improve row-buffer hit-rates due to fewer, if any, hot micro-pages. We talk more about this issue when we discuss results for the HAM scheme below.

In Figure 8, we present energy-delay-squared ($E * D^2$) for the ROPS proposal with epoch length of 5 million cycles, normalized to the baseline. E refers to the DRAM energy (excluding memory channel energy) per CPU load/store. D refers to the inverse of throughput. The second bar (secondary Y-axis) plots the performance for convenient comparison. All applications, except *ferret* have lower (or as much) $E * D^2$ compared to baseline with the proposed scheme. A higher number of row-buffer hits leads to higher overall efficiency: lower access time *and* lower energy. The average savings in energy for all the applications is nearly 12% for ROPS.

Figure 9 shows the results for Hardware Assisted Migration (HAM) scheme, described in Section 3.2, along with ROPS and the profiled scheme, for a 5 million cycle epoch length. Only two benchmarks suffer performance degradation. This performance penalty occurs because a quickly changing DRAM access pattern does not allow HAM or ROPS to effectively capture the micro-pages that are worth migrating. This leads to high migration overheads without any performance improvement, leading to overall performance degradation. For the Profile scheme, the occasional minor degradation is caused due to non-determinism of multi-threaded workloads as mentioned earlier.

Compute applications, like those from NPB suite, some from PARSEC suite, and the multi-programmed workload (SPEC-CPU and BioBench suites), usually show low or negative performance change with our proposals. The reason for these poor improvements are the tight loops in these applications that exhibit very regular data access pattern. This implies there are fewer hot micro-pages within an OS page. Thus compared to applications like *vips* - that do not stride through OS pages - performance improvement is not as high for these compute applications. The prime example of this phenomenon is the *STREAM* application. This benchmark is designed to measure the main memory bandwidth of a machine, and it does so by loop based reads and writes of large arrays. As can be seen from the graph, it exhibits a modest performance gain. Applications like *blackscholes*, *bodytrack*, *canneal*, *swaptions*, *vips* and *SPECjbb* which show substantial performance improvement on the other hand, all work on large data sets while they access some OS pages (some micro-pages to be precise) heavily. These are possibly code pages that are accessed as the execution proceeds. The average performance improvement for these applications alone is approximately 9%.

Figure 10 plots the $E * D^2$ metric for all these 3 schemes. As before, lower is better. Note that while accounting for energy consumption under the HAM scheme, we take into account the energy consumption in DRAM, and energy lost due to DRAM data migrations and MT look-ups. All but one application perform better than the baseline and save energy for HAM. As expected, Profile saves the maximum amount of energy while HAM and ROPS closely track it. ROPS almost always saves less energy than HAM since TLB misses are costly, both in terms of DRAM access energy and application performance. On an average, HAM saves about 15% on $E * D^2$ compared to baseline with very little standard deviation between the results (last bar in the graph).

Epoch length is an important parameter in our proposed schemes. To evaluate the sensitivity of our results to epoch length, we experimented with many different epoch durations (1 M, 5 M, 10 M, 50 M and 100 M cycles). The results from experiments with epoch length of 10 M cycles are summarized in Figures 11 and 12. A comparison of Figures 9, 10, 11, and 12 shows that some applications (such as *SpecJBB*) benefit more from the 5 M epoch length, while others (*blackscholes*) benefit more from the 10 M epoch length. We leave the dynamic estimation of optimal epoch lengths for future work. We envision the epoch length to be a tunable parameter in software. We do observe consistent improvements with our proposed schemes, thus validating their robustness. Since performance and energy consumption are also sensitive to the memory controller's request scheduling policy, we experimented with First-Come First-Serve (FCFS) access policy. As with variable epoch length experiments, the results show consistent performance improvements (which were predictably lower than FR-FCFS policy).

Finally, to show that our simulation interval of 250 million cycles was representative of longer execution windows, we also evaluated our schemes for 1 billion cycle simulation window. In Figure 13 we present results for this sensitivity experiment. We only show the performance improvement over baseline as $E * D^2$ results are similar. As can be observed, the performance improvement is almost identical to 250 million cycle simulations.

Results Summary. For both ROPS and HAM, we see consistent improvement in energy and performance. The higher benefits with HAM are because of its ability to move data without the overhead

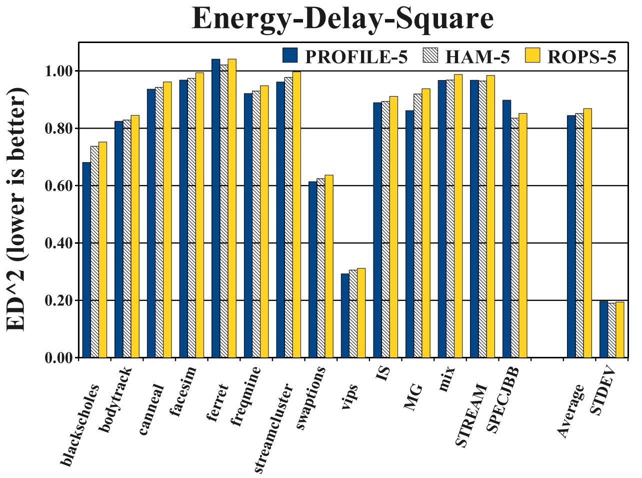

Figure 10. Profile, HAM and ROPS - Energy Savings ($E * D^2$) for 5 Million Cycle Epoch Length

of TLB shoot-down and TLB misses. For HAM, since updating the MT is not expensive, the primary overhead is the cost of performing the actual DRAM copy. The energy overhead of MT look-up is reduced due to design choices and is marginal compared to page table updates and TLB shoot-downs and misses associated with ROPS. Due to these lower overheads, we observe HAM performing slightly better than ROPS (1.5% in terms of performance and 2% in energy for best performing benchmarks). The two schemes introduce different implementation overheads. While HAM requires hardware additions, it exhibits slightly better behavior. ROPS, on the other hand, is easier to implement in commodity systems today and offers flexibility because of its reliance on software.

5. Related Work

A large body of work exists on DRAM memory systems. Cuppu et al. [18] first showed that performance is sensitive to DRAM data mapping policy and Zhang et al. [59] proposed schemes to reduce row-buffer conflicts using a permutation based mapping scheme. Delaluz et al. [19] leveraged both software and hardware techniques to reduce energy while Huang et al. [25] studied it from OS' virtual memory sub-system perspective. Recent work [60, 61] focuses on building higher performance and lower energy DRAM memory systems with commodity DRAM devices. Schemes to control data placement in large caches by modifying physical addresses have also been studied recently [6, 13, 23, 58]. We build on this large body of work to leverage our observations that DRAM accesses to most heavily accessed OS pages are clustered around few cache-line sized blocks.

Page allocation and migration have been employed in a variety of contexts. Several bodies of work have evaluated page coloring and its impact on cache conflict misses [10, 20, 32, 42, 50]. Page coloring and migration have been employed to improve proximity of computation and data in a NUMA multi-processor [12, 15, 34–36, 55] and in a NUCA caches [6, 14, 47]. These bodies of work have typically attempted to manage capacity constraints (especially in caches) and communication distances in large NUCA caches. Most of the NUMA work pre-dates the papers [17, 18, 48] that shed insight on the bottlenecks arising from memory controller constraints. Here, we not only apply the well-known concept of page allocation to a different domain, we extend our policies to be cognizant of the several new constraints imposed by DRAM memory systems, particularly row-buffer re-use.

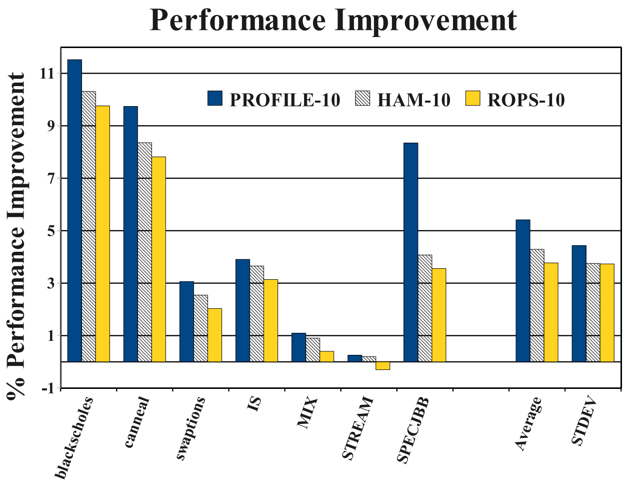

Figure 11. Profile, HAM and ROPS - Performance Improvement for 10 Million Cycle Epoch Length

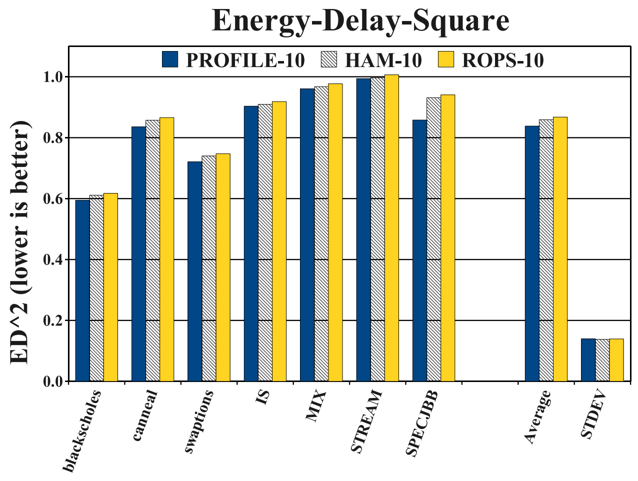

Figure 12. Profile, HAM and ROPS - Energy Savings ($E * D^2$) Compared to Baseline for 10 Million Cycle Epoch.

A significant body of work has been dedicated to studying the effects of DRAM memory on overall system performance [17, 39] and memory controller policies [21, 48]. Recent work on memory controller policies studied effects of scheduling policies on power and performance characteristics [44, 45, 60, 62] for CMPs and SMT processors. Since the memory controller is a shared resource, all threads experience a slowdown when running in tandem with other threads, relative to the case where the threads execute in isolation. Mutlu and Moscibroda [44] observe that the prioritization of requests to open rows can lead to long average queueing delays for threads that tend to not access open rows. This leads to unfairness with some threads experiencing memory stall times that are ten times greater than that of the higher priority threads. That work introduces a Stall-Time Fair Memory (STFM) scheduler that estimates this disparity and over-rules the prioritization of open row access if the disparity exceeds a threshold. While this policy explicitly targets fairness (measured as the ratio of slowdowns for the most and least affected threads), minor throughput improvements are also observed as a side-effect. We believe that such advances

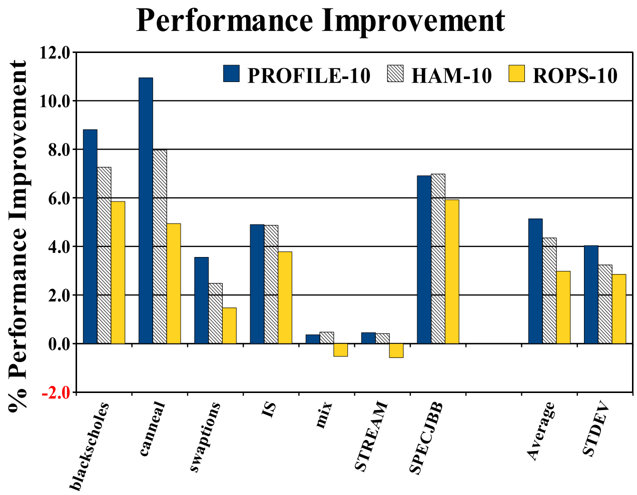

Figure 13. Profile, HAM and ROPS - Performance Improvements for 10M Cycle Epoch Length, and 1 Billion Execution Cycles.

in scheduling policies can easily be integrated with our policies to give additive improvements.

Our proposals capture locality at the DRAM row-buffer, however we believe our approach is analogous to proposals like victim caches [31]. Victim caches are populated by recent evictions, while our construction of an efficient "DRAM region" is based on the detection of hot-spots in the access stream that escapes whatever preceding cache level. Our proposal takes advantage of the fact that co-location of hot-spots leads to better row-buffer utilization, while the corresponding artifact does not exist in traditional victim caches. The optimization is facilitated by introducing another level of indirection. Victim caches, on the other hand, provide increased associativity for a few sets, based on application needs. Therefore, we don't believe that micro-pages and victim caches are comparable in terms of their design or utility.

6. Conclusions

In this paper we attempt to address the issues of increasing energy consumption and access latency being faced by modern DRAM memory systems. We propose two schemes that control data placement for improved energy and performance characteristics. These schemes are agnostic to device and signaling standards and therefore their implementation is not constrained by standards. Both schemes rely on DRAM data migration to maximize hits within a row-buffer. The hardware based proposal incurs less run-time overhead, compared to the software-only scheme. On the other hand, the software-only scheme can be easily implemented without major architectural changes and can be more flexible. Both schemes provide overall performance improvements of 7-9% and energy improvements of 13-15% for our best performing benchmarks.

Acknowledgments

We would like to thank our shepherd Michael Swift for his inputs to improve this paper. We also thank the anonymous reviewers for their valuable comments and suggestions. This work was supported in parts by NSF grants CCF-0430063, CCF-0811249, CCF-0702799, CCF-0916436, NSF CAREER award CCF-0545959, Intel, SRC grant 1847.001 and the University of Utah.

References

[1] STREAM - Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/.

[2] Virtutech Simics Full System Simulator. http://www.virtutech.com.

[3] Java Server Benchmark, 2005.
Available at `http://www.spec.org/jbb2005/`.

[4] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung. BioBench: A Benchmark Suite of Bioinformatics Applications. In *Proceedings of ISPASS*, 2005.

[5] K. Asanovic and et. al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.

[6] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. Dynamic Hardware-Assisted Software-Controlled Page Placement to Manage Capacity Allocation and Sharing within Large Caches. In *Proceedings of HPCA*, 2009.

[7] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, D. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5 (3):63–73, Fall 1991.

[8] L. Barroso and U. Holzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009.

[9] C. Benia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical report, Department of Computer Science, Princeton University, 2008.

[10] B. Bershad, B. Chen, D. Lee, and T. Romer. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *Proceedings of ASPLOS*, 1994.

[11] J. Carter, W. Hsieh, L. Stroller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a Smarter Memory Controller. In *Proceedings of HPCA*, 1999.

[12] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum. Scheduling and Page Migration for Multiprocessor Compute Servers. In *Proceedings of ASPLOS*, 1994.

[13] M. Chaudhuri. PageNUCA: Selected Policies For Page-Grain Locality Management In Large Shared Chip-Multiprocessor Caches. In *Proceedings of HPCA*, 2009.

[14] S. Cho and L. Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *Proceedings of MICRO*, 2006.

[15] J. Corbalan, X. Martorell, and J. Labarta. Page Migration with Dynamic Space-Sharing Scheduling Policies: The case of SGI 02000. *International Journal of Parallel Programming*, 32(4), 2004.

[16] R. Crisp. Direct Rambus Technology: The New Main Memory Standard. In *Proceedings of MICRO*, 1997.

[17] V. Cuppu and B. Jacob. Concurrency, Latency, or System Overhead: Which Has the Largest Impact on Uniprocessor DRAM-System Performance. In *Proceedings of ISCA*, 2001.

[18] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A Performance Comparison of Contemporary DRAM Architectures. In *Proceedings of ISCA*, 1999.

[19] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. Irwin. DRAM Energy Management Using Software and Hardware Directed Power Mode Control. In *Proceedings of HPCA*, 2001.

[20] X. Ding, D. S. Nikopoulosi, S. Jiang, and X. Zhang. MESA: Reducing Cache Conflicts by Integrating Static and Run-Time Methods. In *Proceedings of ISPASS*, 2006.

[21] X. Fan, H. Zeng, and C. Ellis. Memory Controller Policies for DRAM Power Management. In *Proceedings of ISLPED*, 2001.

[22] Z. Fang, L. Zhang, J. Carter, S. McKee, and W. Hsieh. Online Superpage Promotion Revisited (Poster Session). *SIGMETRICS Perform. Eval. Rev.*, 2000.

[23] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-Optimal Block Placement And Replication In Distributed Caches. In *Proceedings of ISCA*, 2009.

[24] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. In *Proceedings of ACM SIGARCH Computer Architecture News*, 2005.

[25] H. Huang, P. Pillai, and K. G. Shin. Design And Implementation Of Power-Aware Virtual Memory. In *Proceedings Of The Annual Conference On Usenix Annual Technical Conference*, 2003.

[26] H. Huang, K. Shin, C. Lefurgy, and T. Keller. Improving Energy Efficiency by Making DRAM Less Randomly Accessed. In *Proceedings of ISLPED*, 2005.

[27] *Intel 845G/845GL/845GV Chipset Datasheet: Intel 82845G/82845GL/82845GV Graphics and Memory Controller Hub (GMCH)*. Intel Corporation, 2002. http://download.intel.com/design/chipsets/datashts/29074602.pdf.

[28] ITRS. International Technology Roadmap for Semiconductors, 2007 Edition. http://www.itrs.net/Links/2007ITRS/Home2007.htm.

[29] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems - Cache, DRAM, Disk*. Elsevier, 2008.

[30] JEDEC. *JESD79: Double Data Rate (DDR) SDRAM Specification*. JEDEC Solid State Technology Association, Virginia, USA, 2003.

[31] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proceedings of ISCA-17*, pages 364–373, May 1990.

[32] R. E. Kessler and M. D. Hill. Page Placement Algorithms for Large Real-Indexed Caches. *ACM Trans. Comput. Syst.*, 10(4), 1992.

[33] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, third edition, 1997.

[34] R. LaRowe and C. Ellis. Experimental Comparison of Memory Management Policies for NUMA Multiprocessors. Technical report, 1990.

[35] R. LaRowe and C. Ellis. Page Placement policies for NUMA multiprocessors. *J. Parallel Distrib. Comput.*, 11(2), 1991.

[36] R. LaRowe, J. Wilkes, and C. Ellis. Exploiting Operating System Support for Dynamic Page Placement on a NUMA Shared Memory Multiprocessor. In *Proceedings of PPOPP*, 1991.

[37] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. Reinhardt, and T. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *Proceedings of ISCA*, 2009.

[38] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. In *Proceedings of ISCA*, 2008.

[39] W. Lin, S. Reinhardt, and D. Burger. Designing a Modern Memory Hierarchy with Hardware Prefetching. In *Proceedings of IEEE Transactions on Computers*, 2001.

[40] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, February 2002.

[41] *Micron DDR2 SDRAM Part MT47H64M8*. Micron Technology Inc., 2004.

[42] R. Min and Y. Hu. Improving Performance of Large Physically Indexed Caches by Decoupling Memory Addresses from Cache Addresses. *IEEE Trans. Comput.*, 50(11), 2001.

[43] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of MICRO*, 2007.

[44] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of MICRO*, 2007.

[45] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *Proceedings of ISCA*, 2008.

[46] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, Transparent Operating System Support For Superpages. *SIGOPS Oper. Syst. Rev.*, 2002.

[47] N. Rafique, W. Lim, and M. Thottethodi. Architectural Support for Operating System Driven CMP Cache Management. In *Proceedings of PACT*, 2006.

[48] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens. Memory Access Scheduling. In *Proceedings of ISCA*, 2000.

[49] T. Romer, W. Ohlrich, A. Karlin, and B. Bershad. Reducing TLB and Memory Overhead Using Online Superpage Promotion. In *Proceedings of ISCA-22*, 1995.

[50] T. Sherwood, B. Calder, and J. Emer. Reducing Cache Misses Using Hardware and Software Page Placement. In *Proceedings of SC*, 1999.

[51] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor. In *Proceedings of SIGMETRICS*, 2002.

[52] M. Swanson, L. Stoller, and J. Carter. Increasing TLB Reach using Superpages Backed by Shadow Memory. In *Proceedings of ISCA*, 1998.

[53] M. Talluri and M. D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of ASPLOS-VI*, 1994.

[54] S. Thoziyoor, N. Muralimanohar, and N. Jouppi. CACTI 5.0. Technical report, HP Laboratories, 2007.

[55] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. *SIGPLAN Not.*, 31(9), 1996.

[56] D. Wallin, H. Zeffer, M. Karlsson, and E. Hagersten. VASA: A Simulator Infrastructure with Adjustable Fidelity. In *Proceedings of IASTED International Conference on Parallel and Distributed Computing and Systems*, 2005.

[57] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: A Memory-System Simulator. In *SIGARCH Computer Architecture News*, volume 33, September 2005.

[58] X. Zhang, S. Dwarkadas, and K. Shen. Hardware Execution Throttling for Multi-core Resource Management. In *Proceedings of USENIX*, 2009.

[59] Z. Zhang, Z. Zhu, and X. Zhand. A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality. In *Proceedings of MICRO*, 2000.

[60] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu. Mini-Rank: Adaptive DRAM Architecture For Improving Memory Power Efficiency. In *Proceedings of MICRO*, 2008.

[61] H. Zheng, J. Lin, Z. Zhang, and Z. Zhu. Decoupled DIMM: Building High-Bandwidth Memory System from Low-Speed DRAM Devices. In *Proceedings of ISCA*, 2009.

[62] Z. Zhu and Z. Zhang. A Performance Comparison of DRAM Memory System Optimizations for SMT Processors. In *Proceedings of HPCA*, 2005.

[63] Z. Zhu, Z. Zhang, and X. Zhang. Fine-grain Priority Scheduling on Multi-channel Memory Systems. In *Proceedings of HPCA*, 2002.