Int J Parallel Prog (2012) 40:57–83
DOI 10.1007/s10766-011-0178-1

Managing Data Placement in Memory Systems with Multiple Memory Controllers

M. Awasthi · D. Nellans · K. Sudan · R. Balasubramonian · A. Davis

Received: 4 March 2011 / Accepted: 28 June 2011 / Published online: 7 August 2011

Abstract Modern processors such as Tilera's Tile64, Intel's Nehalem, and AMD's Opteron are migrating memory controllers (MCs) on-chip, while maintaining a large, flat memory address space. This trend to utilize multiple MCs will likely continue and a core or socket will consequently need to route memory requests to the appropriate MC via an inter- or intra-socket interconnect fabric similar to AMD's HyperTransport™, or Intel's Quick-Path Interconnect™. Such systems are therefore subject to non-uniform memory access (NUMA) latencies because of the time spent traveling to remote MCs. Each MC will act as the gateway to a particular region of the physical memory. Data placement will therefore become increasingly critical in minimizing memory access latencies. Increased competition for memory resources will also increase the memory access latency variation in future systems. Proper allocation of workload data to the appropriate MC will be important in decreasing the variation and average latency when servicing memory requests. The allocation strategy will need to be aware of queuing delays, on-chip latencies, and row-buffer hit-rates for each MC. In this paper, we propose dynamic mechanisms that take these factors into account when placing data in appropriate slices of physical memory. We introduce adaptive first-touch page placement, and dynamic page-migration mechanisms to reduce DRAM access delays for multi-MC systems. We also introduce policies that can handle data placement in memory systems that have regions with heterogeneous properties. The proposed policies yield average performance improvements of 6.5% for adaptive first-touch page-placement, and 8.9% for a dynamic page-migration policy for a system with

M. Awasthi (✉) · D. Nellans · K. Sudan · R. Balasubramonian · A. Davis
School of Computing, University of Utah, Salt Lake City, UT, USA
e-mail: manua@cs.utah.edu

D. Nellans
FusionIO, Cottonwood Heights, UT, USA

homogeneous DRAM DIMMs. We also show improvements in systems that contain DIMMs with different performance characteristics.

Keywords DRAM · Phase change memory · Data locality · Heterogeneous memory hierarchies

1 Introduction

Modern microprocessors increasingly integrate the *memory controller (MC)* on-chip in order to reduce main memory access latency. Memory pressure will increase as core-counts per socket rise resulting in a single MC becoming a bottleneck. In order to avoid this problem, modern multi-core processors (chip multiprocessors, CMPs) have begun to integrate multiple MCs per socket [53,57,62]. Similarly, multi-socket motherboards provide connections to multiple MCs via off-chip interconnects such as AMD's HyperTransport™(HT) and Intel's Quick Path Interconnect™(QPI). In both architectures, a core may access any DRAM location by routing its request to the appropriate MC. Multi-core access to a large physical memory space partitioned over multiple MCs is likely to continue and exploiting MC locality will be critical to overall system throughput.

Recent efforts [1,36,58,62] have incorporated multiple MCs in their designs, but there is little analysis on how data placement should be managed and how a particular placement policy will affect main memory access latencies. In addressing this problem, we show that simply allocating an application's thread data to the closest MC may not be optimal since it does not take into account queuing delays, row-buffer conflicts, and on chip interconnect delays. In particular, we focus on placement strategies which incorporate: (i) the communication distance and latency between the core and the MC, (ii) queuing delay at the MC, and (iii) DRAM access latency, which is heavily influenced by row-buffer hit rates. We show that improper management of these factors can cause a significant degradation in performance.

Further, future memory hierarchies may be heterogeneous. Some DIMMs may be implemented with PCM devices while others may use DRAM devices. Alternatively, some DIMMs and channels may be optimized for power efficiency, while others may be optimized for latency. In such heterogeneous memory systems, we show that additional constraints must be included in the optimization functions to maximize performance.

To our knowledge, this is the first attempt at intelligent data placement in a multi-MC platform. This work builds upon ideas found in previous efforts to optimize data placement in last-level shared NUCA caches [2,7–10,17,20,34,50,56,63]. One key difference between DRAM and last level cache placement however, is that caches tend to be capacity constrained, while DRAM access delays are governed primarily by other issues such as long queuing delays and row buffer hit rates. There are only a handful of papers that explore challenges faced in the context of multiple on-chip MCs. Abts et al. [1] explore the physical layout of multiple on-chip MCs to reduce contention in the on-chip interconnect. An optimal layout makes the performance of memory-bound applications predictable, regardless of which core they are scheduled on. Kim et al. [27] propose a new scheduling policy in the context of multiple MCs which requires minimal coordination between MCs. However, neither of these proposals consider how data should be distributed in a NUMA setting while taking into account the interaction of row-buffer hit rates, queuing delays, and on-chip network traffic. Our work takes these DRAM-specific phenomena into account and explores both first-touch page placement and dynamic page-migration designed to reduce access delays. We show average performance improvements of 6.5% with an adaptive first-touch page-coloring policy, and 8.9% with a dynamic page-migration policy. The proposed policies are notably simple in their design and implementation.

The rest of this paper is organized as follows: We provide background and motivational discussion in Sect. 2. Section 3 details our proposed adaptive first-touch and dynamic migration policies and Sect. 4 provides quantitative comparison of the proposed policies. We discuss related work in Sect. 5 and conclusions in Sect. 6.

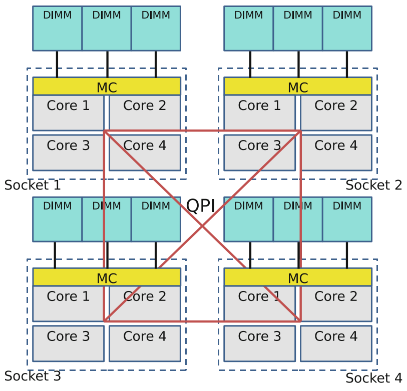
2 Background and Motivational Results

2.1 DRAM Basics

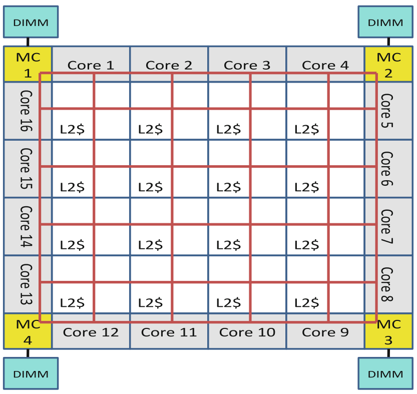
For JEDEC based DRAM, each MC controls one or more dual in-line memory modules (DIMMs) via a bus-based channel comprising a 64-bit datapath, a 17-bit row/column address path, and an 8-bit command/control-path [41]. The DIMM consists of 8 or 9 DRAM chips, depending on the error correction strategy, and data is typically N-bit interleaved across these chips; N is typically 1, 4, 8, or 16 indicating the portion of the 64-bit datapath that will be supplied by each DRAM chip. DRAM chips are logically organized into banks and DIMMs may support one or more ranks. The bank and rank organization supports increased access parallelism since DRAM device access latency is significantly longer than the rate at which DRAM channel commands can be issued.

Commodity DRAMs are very cost sensitive and have been optimized to minimize the cost/bit. Therefore, an orthogonal 2-part addressing scheme is utilized where row and column addresses are multiplexed on the 17-bit address channel. The MC first generates the *row address* that causes an entire row of the target bank to be read into a *row-buffer*. A subsequent *column address* selects the portion of the row buffer to be read or written. Each row-buffer access reads out 4 or 8 kB of data. DRAM sub-array reads are destructive but modern DRAMs restore the sub-array contents on a read by over-driving the sense amps. However if there is a write to the row-buffer, then the row-buffer must be written to the sub-arrays prior to an access to a different row in the same bank. Most MCs employ some variation of a row-buffer management policy, with the *open-page* policy being most favored. An open-page policy maintains the row-buffer contents until the MC schedules a request for a different row in that same bank. A request to a different row is called a "row-buffer conflict". If an application exhibits locality, subsequent requests will be serviced by a "row-buffer hit" to the currently active row-buffer. Row-buffer hits are much faster to service than row-buffer conflicts.

In addition to a row-buffer management policy, the MC typically has a queue of pending requests and must decide how to best schedule requests. Memory controllers



(a) Logical organization of a multi-socket Nehalem.



(b) Assumed 16-core 4-MC model

Fig. 1 Platforms with multiple memory controllers. **a** Logical organization of a multi-socket Nehalem. **b** Assumed 16-core 4-MC model

choose the next request to issue by balancing timing constraints, bank constraints, and priorities. One widely adopted scheduling policy is FR-FCFS (First Ready - First Come First Serve) [52] that prioritizes requests to open rows and breaks ties based on age. The requests issued by the memory controller are typically serviced across a dedicated channel that receives no interference from other memory controllers. As a result, each memory controller has global knowledge about what memory access patterns are occurring on their private slice of physical memory.

2.2 Current/Future Trends in MC Design

Several commercial designs have not only moved the MC on chip, but have also integrated multiple MCs on a single multi-core die. Intel's Nehalem processor [57] shown in Fig. 1a integrates four cores and 1 MC with three channels to DDR3 memory. Multiple Nehalem processors in a multi-socket machine are connected via a QPI interconnect fabric. Any core is allowed to access any location of the physical memory, either via its own local MC or via the QPI to a remote processor's MC. The latency for remote memory access, which requires traversal over the QPI interconnect, is 1.5x the latency for a local memory access (NUMA factor). This change is a result of on-die MCs: in earlier multi-socket machines, memory access was centralized via off-chip MCs integrated on the north-bridge. This was then connected via a shared bus to the DIMMs. Similar to Intel's Nehalem architecture, AMD's quad-core Opteron integrates two 72-bit channels to a DDR2 main memory subsystem [53]. The Tile64 processor [62] incorporates four on-chip MCs that are shared among 64 cores/tiles. A specialized on-chip network allows all the tiles to access any of the MCs, although physical placement details are not publicly available. The Corona architecture from HP [58] is a futuristic view of a tightly coupled nanophotonic NUMA system comprising 64 4-core clusters, where each cluster is associated with a local MC.

It is evident that as we increase the number of cores on-chip, the number of MCs on-chip must also be increased to efficiently feed the cores. However, the ITRS roadmap [23] expects almost a negligible increase in the number of pins over the next 10 years, while Moore's Law implies at least a 16x increase in the number of cores. Clearly the number of MCs cannot scale linearly with the number of cores. If it did, the number of pins per MC would reduce dramatically, causing all transfers to be heavily pipelined leading to long latencies and heavy contention, which we will show in Sect. 4. The realistic expectation is that future many-core chips will accommodate a moderate number of memory controllers, with each MC servicing requests from a subset of cores. This is reflected in the layout that we assume for the rest of this paper, shown in Fig. 1b. Sixteen cores share four MCs that are uniformly distributed at the edge of the chip. On-chip wire delays are an important constraint in minimizing overall memory latency, this simple layout of memory controllers helps minimize the average memory controller to I/O pin and core distance.

While FB-DIMM designs [24] are not very popular today, the FB-DIMM philosophy may eventually cause a moderate increase in the number of channels and MCs. In order to increase capacity, FB-DIMM replaces a few wide channels with many skinny channels, where each channel can support multiple daisy-chained DIMMs. Skinny channels can cause a steep increase in data transfer times unless they are accompanied by increases in channel frequency. Unfortunately, an increase in channel frequency can, in turn, cause power budgets to be exceeded. Hence, such techniques may cause a small increase in the number of channels and MCs on a single processor to find the appropriate balance between energy efficiency and parallelism across multiple skinny channels.

Finally, there is a growing sentiment that future memory systems are likely to be heterogeneous. In order to alleviate the growing concerns over memory energy, some memory channels and DIMMs may be forced to operate at lower frequencies and voltages [15,48]. While FB-DIMM and its variants provide higher capacity, they cause a steep increase in average latency and power. Therefore, they may be employed in a limited extent for a few memory channels, yielding heterogeneous properties for data access per channel. New memory technologies, such as PCM [5], exhibit great advantages in terms of density, but have poor latency, energy, and endurance properties. We believe that future memory systems are likely to use a mix of FB-DIMM variants, DDRx, and PCM nodes within a single system. We therefore consider heterogeneous properties of DIMMs as a first class input into the cost functions used to efficiently place memory across various NUMA nodes.

2.3 Motivational Data

This paper focuses on the problem of efficient data placement at OS page granularity, across multiple physical memory slices. The related problem of data placement across multiple last-level cache banks has received much attention in recent years, with approaches such as cooperative caching [7], page spilling [10], and their derivatives [2,8,9,17,20,34,50,56,63] being the most well known techniques. There has been little prior work on OS-based page coloring to place pages in different DIMMs

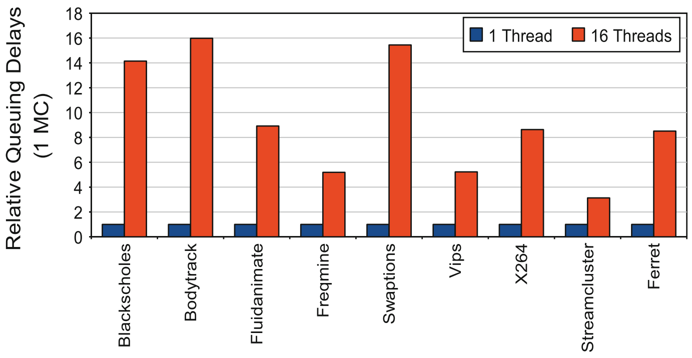


Fig. 2 Relative queuing delays for 1 and 16 threads, single MC, 16 cores

or banks to promote either DIMM or bank-level parallelism (Zhang et al. [64] propose a hardware mechanism within the memory controller to promote bank-level parallelism). The DRAM problem has not received as much attention because there is a common misconception that most design considerations for memory controller policy are dwarfed by the long latency for DRAM chip access. We argue that as contention at the memory controller grows, this is no longer the case.

As mentioned in Sect.2.1, the NUMA factor in a modern multi-socket systems can be as high as 1.5 [57]. This is because of the high cost of traversal on the off-chip QPI/HT network as well as the on-chip network. As core count scales up, wires emerge as bottlenecks. As complex on-chip routed networks are adopted, one can expect tens of cycles in delay when sending requests across the length of the chip [14,62], further increasing the NUMA disparity.

Pin count restrictions prevent the memory controller count from increasing linearly with the number of cores, while simultaneously maintaining a constant channel width per MC. Thus, the number of cores serviced by each MC will continue to rise, leading to contention and long queuing delays within the memory controller. Recent studies [22,33,44,45,65] have identified MC queuing delays as a major bottleneck and have proposed novel mechanisms to improve scheduling policies. To verify these claims, we evaluated the impact of increasing core counts on queuing delay when requests are serviced by just a single memory controller. Section 4 contains a detailed description of the experimental parameters. For the results shown in Fig. 2, each application was run with a single thread and then again with 16 threads, with one thread pinned on every core. The average queuing delay within the memory controller across 16-threads, as compared to just one thread, can be as high as 16x (*Bodytrack*). The average number of cycles spent waiting in the MC request queue was as high as 280 CPU cycles for the 16-thread case. This constitutes almost half the time to service an average (495 cycles) memory request, making a strong case for considering queuing delays for optimized data placement across multiple memory controllers.

When considering factors for optimizing memory placement, it is important to maximize row-buffer hit-rates. For DDR3-1333, there is a factor of 3 overhead, 25 to 75 DRAM cycles, when servicing row-buffer hits versus conflict misses. Figure 3 shows row-buffer hit-rates for a variety of applications when running with 1, 4, and

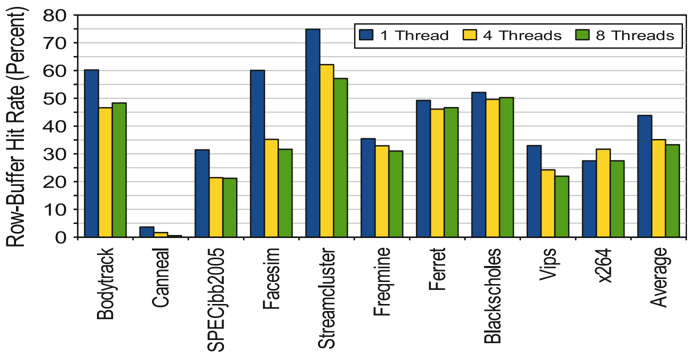


Fig. 3 Row-buffer hit rates, dual-socket, quad-core opteron

8-threads. These measurements were made using hardware performance counters [46] on a dual-socket, quad-core AMD Opteron 2344HE system with 16 2-GB DIMMs. While there is significant variation in the row-buffer hit-rate among applications, the key observation is that in all cases moving from a single to multiple threads decreases the average row-buffer hit-rate seen at the memory controllers due to more frequent row-buffer conflicts. This supports our hypothesis that there is contention within the memory controller that is reducing the effectiveness of open-page policy and it may be possible to alleviate some of this contention through intelligent placement of application data across memory controllers.

Three important observations that we make from the above discussion are: (i) NUMA factor is likely to increase in the future as the relative contribution of wire delay increases. (ii) Higher core and thread counts per memory controller lead to high MC contention, raising the importance of actively tracking and managing MC properties such as queuing delay. (iii) Increased interleaving of memory accesses from different threads leads to a reduction in row-buffer hit rates. (iv) Intelligent memory placement policy must balance all three of these first order effects when choosing where to allocate data to physical memory slice.

3 Proposed Mechanisms

We are interested in developing a general approach to minimize memory access latencies for a system that has many cores, multiple MCs, with varying interconnect latencies between cores and MCs. For this study, we assume a 16-core processor with four MCs, as shown in Fig. 1b where each MC handles a distinct subset of the aggregate physical address space and the memory requests (L2 misses) are routed to the appropriate MC based on the physical memory address. The L2 is shared by all cores, and physically distributed among the 16 tiles in a tiled S-NUCA layout [26,63]. Since the assignment of data pages to an MC's physical memory slice is affected by the mapping of virtual addresses to physical DRAM frames by the OS, we propose two different schemes that manage/modify this mapping to be aware of the DIMMs directly connected to an MC.

When a new virtual OS page is brought into physical memory, it must be assigned to a DIMM associated with a single MC and a DRAM channel associated with that MC. Proper assignment of pages attempts to minimize access latency to the newly assigned page without significantly degrading accesses to other pages assigned to the same DIMM. Ultimately, DRAM access latency is strongly governed by the following factors: (i) the distance between the requesting core and the MC, (ii) the interconnection network load on that path, (iii) the average queuing delay at the MC, (iv) the amount of bank and rank contention at the targeted DIMM, and (v) the row-buffer hit-rate for the application. To make intelligent decisions based on these factors, we must be able to both monitor and predict the impact that assigning or moving a new memory page will have on each of these parameters. Statically generating profiles offline can help in page assignment, but this is almost never practical. For this work, we focus on policies that rely on run-time estimation of application behavior.

To reduce memory access delays we propose: *adaptive first-touch placement* of memory pages, and *dynamic migration of pages* among DIMMs at the OS page granularity. The first scheme is based on DRAM frame allocation by the OS which is aware of MC load (queuing delays, row-buffer hit-rates and bank contention) and the on-chip distance between the core the thread is executing on and the MC that will service requests to this frame. We propose modifications to the OS' memory allocator algorithm so that it is aware of these factors in order to create improved virtual-to-physical mappings only when natural page-faults occur. The second scheme aims to dynamically migrate data at run-time to reduce access delays. This migration of pages occurs when there is excess memory bandwidth to support remapping operations. We also propose mechanisms that allow dynamic migration to occur without stalling CPUs that are accessing the pages being migrated.

3.1 Adaptive First-Touch Page Placement Policy

In the common case, threads/tasks[1] will be assigned to cores rather arbitrarily based on program completion times and task queues maintained by the OS for each core. The OS' task scheduling algorithm could be modified to be aware of multiple-MCs and leverage profile based aggregated MC metrics to intelligently schedule tasks to cores. Clever task scheduling must rely on pre-computed profiles that can be inaccurate and are closely tied to the behavior of co-scheduled applications; this makes achieving a general purpose approach challenging. Instead, we believe that intelligently placing data, such that the overall throughput of the system is improved, is likely to out-perform coarse grained task scheduling optimization because of the fine granularity at which changes to the memory mappings can be made and updated at run-time.

In our adaptive first-touch (AFT) approach for page allocation, when a thread starts executing on some core, each new page it touches will generate a page fault. At this time, the virtual page is assigned to a DRAM frame (physical page) such that it is serviced by an MC that minimizes an objective cost function. The intuition behind the objective function is that most pages associated with a thread will be mapped to the

[1] We use threads and tasks interchangeably in the following discussion, unless otherwise specified.

nearest MC, with a small number of pages being spilled to other nearby MCs, only when beneficial to overall performance. The following cost function is computed for each new page and for each MC j:

$$cost_j = \alpha \times load_j + \beta \times rowhits_j + \lambda \times distance_j$$

where $load_j$ is the average queuing delay at MC j, $rowhits_j$ is the average row-buffer hit-rate seen by MC j, and $distance_j$ is the distance between the core requesting the memory page and the MC, in terms of number of interconnect hops that need to be traversed. The role of $load$ and $distance$ is straightforward; the row buffer hit rate is considered based on the assumption that the new page will be less disruptive to other accesses if it resides in a DIMM with an already low row buffer hit-rate. The relative importance of each factor is determined by the weights α, β, and λ. After estimating the cost function for each MC, the new page is assigned to the MC that minimizes the cost function. In essence, this is done by mapping the virtual page to a physical page in the slice of memory address space being controlled by the chosen MC j. Since allocation of new DRAM frames on a page-fault is on the critical path, we maintain a small history of the past few (5) runs of this cost function for each thread. If two consecutive page faults for a thread happen within 5000 CPU cycles of each other, the maximally recurring MC from the history is automatically chosen for the new page as well. Once the appropriate MC is selected, a DRAM frame managed by this MC is allocated by the OS to service the page-fault.

3.2 Dynamic Page Migration Policy

While adaptive first-touch can allocate new pages efficiently, for long-running programs that aren't actively allocating new pages, we need a facility to react to changing program phases or changes in the environment. We propose a dynamic data migration scheme that tries to adapt to this scenario. Our dynamic migration policy starts out with the AFT policy described above. Then during the course of the program execution, if an imbalance is detected in DRAM access latency between memory controllers, we choose to migrate *N* pages from the highest loaded MC to another one. Decisions are made every *epoch*, where an epoch is a fixed time interval.

The above problem comprises of two parts - (i) finding which MC is loaded and needs to shed load (the *donor* MC), and (ii) deciding the MC that will receive the pages shed by the donor (*recipient* MC). For our experiments, we assume if an MC experiences a drop of 10% or more in row-buffer hit rates from the last epoch, it is categorized as a donor MC[2]. Other reasonable metrics can also be used, such as detecting a large imbalance in queuing delays. When finding a recipient MC, care has to be taken that the incoming pages do not disrupt the locality being experienced at the recipient. As a first approximation, we choose the MC which (i) is physically proximal

[2] This value can be made programmable to suit a particular workload's needs. After extensive exploration, we found that 10% works well across all workloads that we considered.

to the donor MC, and (ii) has the lowest number of row-buffer hits in the last epoch. Hence for each MC k in the *recipient* pool, we calculate

$$cost_k = \Lambda \times distance_k + \Gamma \times row\_hits_k$$

The MC with *least* value for the above cost is selected as the recipient MC. Once this is done, N least recently used pages at the *donor* MC are selected for migration.

It is possible to be more selective regarding the choice of pages and the choice of new MC, but we resort to this simple policy because it is effective and requires very few resources to implement. We note that even when the dynamic migration policy is in use, freshly allocated pages are steered towards the appropriate MCs based on the AFT cost function. Pages that have been migrated are not considered for re-migration for the next two epochs to prevent thrashing of memory across memory controllers.

When migrating pages, the virtual address of the page does not change, but the physical location does. Thus, to maintain correctness two steps need to be taken for pages that are undergoing migration:

1. *Cache Invalidate* The cache lines belonging to the migrated pages have to be invalidated across all cores. With our S-NUCA cache, only one location must be looked up for each cache line. When invalidating the lines, copies in L1 must also be invalidated through the directory tracking these entries forcing any dirty data to be written back to memory prior to migration occurring.
2. *TLB Update* TLBs in all cores have to be informed of the change in the page's physical address. Therefore any core with an active TLB mapping must be updated after the page is physically migrated.

Both of these steps are costly in terms of both power and performance. Thus, premature page migration is likely to result in decreased system performance. Instead, migration should only occur when the anticipated benefit outweighs the overhead of page migration.

To avoid forcing an immediate write back of dirty data when migrating pages, we propose a mechanism that delays the write-back and forwards any dirty data that is flushed to the new physical page rather than the old. To do this, we defer invalidating the TLB entry until an entire page has been copied to its new physical location. Any incoming requests for the page can still be serviced from the old physical location. Only after the page has been copied is the TLB shootdown triggered, forcing a cache write back. The memory controller servicing requests from the old page is notified that it should redirect writes intended for the old physical location N to the new physical location M on an alternate memory controller. With this redirection in place, a TLB shootdown is issued triggering the write back if there is dirty data, and finally the old physical page can be deallocated and returned to the free page list. Only then can the memory controller be instructed to stop forwarding requests to the alternate location, and normal execution resumes. This method of delaying TLB shootdowns is referred to as *lazy-copying* in later sections.

3.3 Heterogeneous Memory Hierarchy

Previously in Sections 3.1 and 3.2, we assumed a homogeneous memory system with DRAM DIMMs attached to all the MCs. However, future memory systems will likely be comprised of different memory technologies. These memory technologies will differ in a number of facets, with access latencies and bit-density being the two important factors considered in this work. There may also be different channel and wire protocols for accessing a particular memory type, but for this study we assume a unified standard which allows us to focus on the memory controller issues, not memory technology properties.

We assume a scenario where memory controllers in the system can only access one of two possible types of technologies in the system. For example, in the 4-MC model, one MC controls a gang of DDR3 DIMMs, while the rest control FB-DIMMs. Alternatively, the NUMA architecture might comprise two MCs controlling DDR3 DRAM while the other two MCs are controlling PCM based devices. For heterogeneous memory hierarchies, there is an inherent difference between the capacity and latency of different devices (listed in Table 1). As a result, a uniform cost function for device access across the entire memory space cannot be assumed; care has to be taken to assign/move heavily used pages to faster memory (e.g. DRAM), while pages that are infrequently used can be moved to slower, but denser regions of memory (e.g. PCM). To account for the heterogeneity in such systems, we modify the cost function for the adaptive first touch policy as follows. On each new page allocation, for each MC_j, we evaluate the following cost function:

$$\begin{aligned} cost_j &= \alpha \times load_j + \beta \times rowhits_j \\ &\quad + \lambda \times distance_j + \tau \times LatencyDimmCluster_j + \mu \times Usage_j \end{aligned} \tag{1}$$

The new term in the cost function, $LatencyDimmCluster_j$, is indicative of the latency of a particular memory technology. This term can be made programmable (with the average or worst case access latency), or can be based on runtime information collected by the OS daemon. $Usage_j$ represents the percentage of DIMM capacity that is currently allocated; it is intended to account for the fact that different memory technologies have different densities and pages must be allocated to DIMMs in approximately that proportion. As before, the MC with the least value of the cost function is assigned the new incoming page.

Long running applications tend to touch a large number of pages, with some of them becoming dormant after a period of initial use as the application moves through distinct phases of execution. To optimize our memory system for these pages we propose a variation of our initial dynamic page migration policy. In this variation we target two objectives: (i) For pages that are currently dormant or *sparingly* (Least Recently Used, LRU) used in the faster memory nodes, these pages can be migrated onto a slower memory node, further reducing the pressure on faster node. (ii) Place infrequently used pages in higher density memory (PCM) allowing more space for frequently used pages in the faster and lower capacity memory (DRAM). A dynamic migration policy for heterogeneous memory can be of two distinct flavors: (i) Pages from any MC can

Table 1 Timing parameters [39,32]

Parameter	DRAM (DDR3)	DRAM (Fast)	PCM	Description
tRCD	12.5ns	9ns	55ns	Interval between row access command and data ready at the sense amps
tCAS	12.5ns	12.5ns	12.5ns	Interval between column access command and the start of data burst
tRP	12.5ns	12.5ns	12.5ns	Time to precharge a row
tWR	12.5ns	9ns	125ns	Time between the end of a write data burst and a subsequent precharge command to the same bank
tRAS	45ns	45ns	45ns	Minimum interval between row access and precharge to same bank
tRRD	7.5ns	7.5ns	7.5ns	Minimum gap between row access commands to the same device
tRTRS	2 Bus Cycles	2 Bus Cycles	2 Bus Cycles	Rank-to-rank switching delay
tFAW	45 ns	45 ns	45 ns	Rolling time window within which maximum of 4 bank activations can be made to a device
tWTR	7.5ns	7.5ns	7.5ns	Delay between a write data burst and a column read command to the same rank
tCWD	6.5ns	6.5ns	6.5ns	Minimum delay between column access command and the write data burst

be migrated to any other MC without considering the memory technology attached to it. (ii) The policy is cognizant of the memory technology. When the memory technologies considered are extremely different in terms of latency *and* density, only policy (ii) is considered. In the former case, the pool of recipient MCs is all MCs except the donor. In the latter, the pool is restricted to MCs with only slower devices attached to them. The recipient cost function remains the same in both cases.

3.4 Overheads of Estimation and Migration

Employing any of the proposed policies incurs some system-level (hardware and OS) overheads. The hardware (MC) needs to maintain counters to keep track of per-workload delay and access counts which most of the modern processors already implement for measuring memory system events (Row-Hits/Misses/Conflicts) [46]. In order to calculate the value of the cost function, a periodic system-level daemon has to read the values from these hardware counters. Currently, the only parameter that cannot be directly measured is $load$ or queuing delay. However, performance monitoring tools can measure average memory latency easily. The difference between the measured total memory latency and the time spent accessing devices (which is known by the memory controller when negotiating channel setup) can provide an accurate estimate of $load$. We expect that future systems will include hardware counters to directly measure $load$.

Migrating pages across memory nodes requires trapping into the OS to update page-table entries. Because we only perform dynamic migration when there is excess available memory bandwidth, none of these operations are typically on the application's critical path. However, invalidating the TLB on page migration results in a requisite miss and ensuing page-table walk. We include the cost of this TLB shootdown and page table walk in all experiments in this study. We also model the additional load of copying memory between memory controllers via the on-chip network. We chose not to model data transfers via DMA because of the synchronization complexity it would introduce into our relatively simple migration mechanism. Also, our simulation framework does not let us quantify the associated costs of DMA only page migrations.

4 Results

The full system simulations are built upon the Simics [37] platform. Out-of-order and cache timings are simulated using Simics' *ooo-micro-arch* and *g-cache* modules respectively. 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 [39]. 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 closed-page policies. The details pertaining to the simulated system are shown in Table 2. Other major components of Gries' model that we adopted for our platform are: the bus model, DIMM and device models, and overlapped processing of commands by the memory controller. Overlapped processing allows simultaneous processing of access requests on the memory bus, while receiving further requests from the CPU. This allows hiding activation and pre-charge latency using the pipelined interface of DRAM devices. We model the CPU to allow non-blocking load/store execution to support overlapped processing. Our MC scheduler implements an FR-FCFS scheduling policy and an open-page row-buffer management policy. PCM devices are assumed to be built along the same lines as DRAM. We adopted the PCM architecture and timing parameters from [32]. A detailed list of DRAM and PCM timing parameters is listed in Table 1.

DRAM address mapping parameters for our platform were adopted from the DRAMSim framework [61], and was assumed to be the same for PCM devices. We implemented basic SDRAM mapping, as found in user-upgradeable memory systems, (similar to Intel 845G chipsets' DDR SDRAM mapping [21]). Some platform specific implementation suggestions were taken from the VASA framework [60]. 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 pre-charge, activation, CAS, write-back to DRAM cells etc. are recorded and total energy consumed reported using energy parameters derived from a modified version of CACTI [43]. Since pin-bandwidth is limited (and will be in the future), we assume a constant bandwidth from the chip to the DRAM sub-system. In case of multiple MCs, bandwidth is equally divided among all controllers by reducing the burst-size. We study a diverse set of workloads including PARSEC [3] (with sim-large working set), SPECjbb2005 (with number of warehouses

Table 2 Simulator parameters

ISA	UltraSPARC III ISA
L1 I-cache	32KB/2-way, 1-cycle
L2 Cache (shared)	2 MB/8-way, 3-cycle/bank access
Hop access time	2 cycles
(Vertical and horizontal)	
Processor frequency	3 GHz
On-chip network width	64 bits
CMP size and Core Freq.	16-core, 3 GHz
L1 D-cache	32KB/2-way, 1-cycle
L1/L2 Cache line size	64 Bytes
Router overhead	3 cycles
Page Size	4 KB
On-chip network frequency	3 GHz
Coherence protocol	MESI
DRAM parameters	
DRAM device parameters	Micron MT41J256M8 DDR3-800
	Timing parameters [40],
	2 ranks, 8 banks/device,
	32768 rows/bank, x8 part
DIMM configuration	8 Non-ECC un-buffered DIMMs,
	64 bit channel, 8 devices/DIMM
DIMM-level row-buffer size	8KB/DIMM
Active row-buffers per DIMM	8 (each bank in a device maintains a row-buffer)
Total DRAM capacity	4 GB
DRAM bus frequency	1,600 MHz
Values of cost function constants	
$\alpha, \beta, \lambda, \Lambda, \Gamma, \tau, \mu$	10, 20, 100, 200, 100, 20, 500

equal to number of cores) and Stream benchmark (number of threads equal to number of cores).

For all experiments involving dynamic page migration with homogeneous memory subsystem, we migrate 10 pages ($N = 10$, Sect. 3) from each MC[3], per epoch. An Epoch is 5 million cycles long. For the heterogeneous memory subsystem, all pages that have not been accessed in the last two consecutive epochs are migrated to appropriate PCM MCs.

We (pessimistically) assume the cost of *each* TLB entry invalidation to be 5000 cycles. We warm-up caches for 25 million instructions and then collect statistics for the next 500 million instructions. The weights of the cost function were determined

[3] Empirical evidence suggested that moving more than 10 pages at a time significantly increased the associated overheads, hence decreasing the effectiveness of page migrations.

after an extensive design space exploration[4]. The L2 cache size was scaled down to resemble an MPKI (misses per thousand instructions) of 10.6, which was measured on the real system described in Sect. 2 for PARSEC and commercial workloads.

4.1 Metrics for Comparison

For comparing the effectiveness of the proposed schemes, we use the total system throughput defined as $\sum_i (IPC^i_{shared}/IPC^i_{alone})$ where IPC^i_{shared} is the IPC of program i in a multi-core setting with one or more shared MCs. IPC^i_{alone} is the IPC of program i on a stand-alone single-core system with one memory controller.

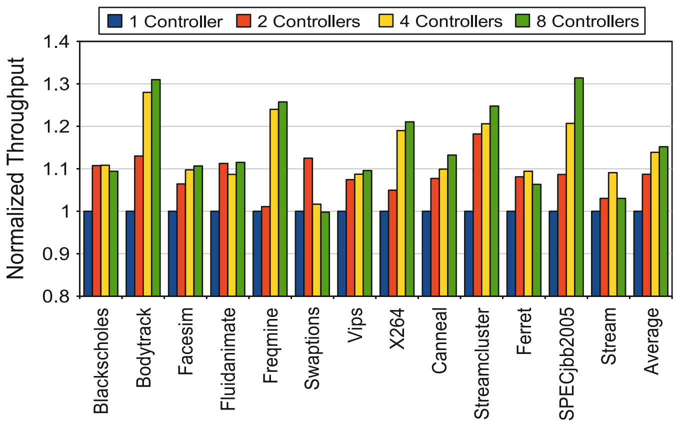
We also report queuing delays which refer to the time spent by a memory request at the memory controller waiting to get scheduled plus the cycles spent waiting to get control of DRAM channel(s). This metric also includes additional stall cycles accrued traversing the on-chip network.

4.2 Multiple Memory Controllers: Homogeneous DRAM Hierarchy

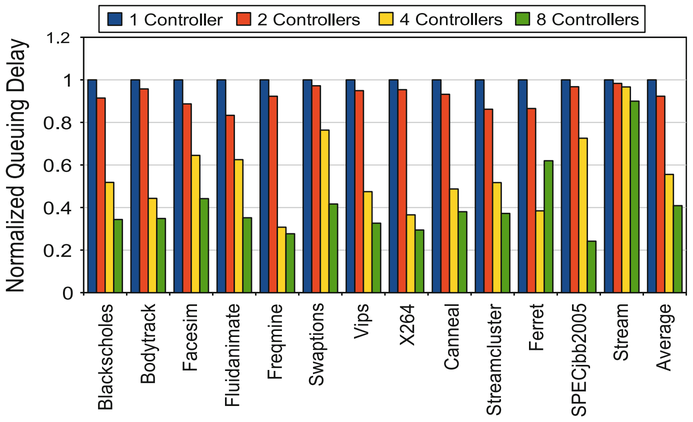
First we study the effect of multiple MCs on the overall system performance for the homogeneous DRAM hierarchy (Fig. 4). We divide the total physical address space equally among all MCs, with each MC servicing an equal slice of the total memory address space. All MCs for these experiments are assumed to be located along chip periphery (Fig. 1b). The baseline is assumed to be the case where OS' page allocation routine tries to allocate the new page at the nearest (physically proximal) MC. If no free pages are available at that MC, the next nearest one is chosen.

For a fixed number of cores, additional memory controllers improve performance up to a given point (4 controllers for 16 cores), after which the law of diminishing returns starts to kick in. On an average across all workloads, as compared to a single MC, 4 MCs help reduce the overall queuing delay by 47% and improve row buffer hits by 28%, resulting in an overall throughput gain of 15%. Adding more than 4 MCs to the system still helps overall system throughput for most workloads, but for others, the benefits are minimal because (i) naive assignment of threads to MCs increases interference and conflicts, and (ii) more MCs lead to decreased memory channel widths per MC, increasing the time taken to transfer data per request and adding to overall queuing delay. Combined, both these factors eventually end up hurting performance. For example, for an eight MC configuration, as compared to a 4 - MC case, *ferret* experiences increased conflicts at MC numbers 3,5 and 7, with the row buffer hit rates going down by 13%, increasing the average queuing delay by 24%. As a result, the overall throughput for this workload (*ferret*) goes down by 4%.This further strengthens our initial assumption that naively adding more MCs doesn't solve the problem and makes a strong case for intelligently managing data across a small number of MCs. Hence, for all the experiments in the following sections, we use a 4 MC configuration.

[4] We report results for the best performing case.



(a) Number of Controllers vs. Throughput

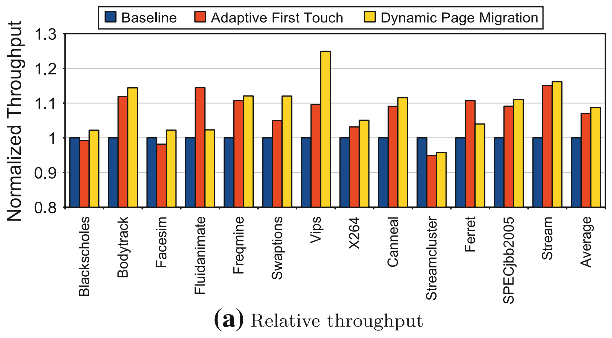


(b) Number of Controllers vs Avg. Queuing Delays

Fig. 4 Impact of multiple memory controllers, homogeneous hierarchy. **a** Number of controllers versus throughput, **b** number of controllers versus avg. queuing delays

4.2.1 Adaptive First-Touch and Dynamic Migration Policies: Homogeneous Hierarchy

Figure 5 compares the average throughput improvement of adaptive first-touch and dynamic-migration policies for the homogeneous DRAM hierarchy over the baseline. On an average, over all the workloads, adaptive first-touch and dynamic page-migration perform 6.5% and 8.9% better than the baseline, respectively. Part of this improvement comes from the intelligent mapping of pages to improve row-buffer hit rates, which are improved by 15.1% and 18.2% respectively for first-touch and dynamic-migration policies. The last cluster in Fig. 5b (STDDEV) shows the standard deviation of individual MC row-buffer hits for the three policies. In essence, a higher value of this statistic implies that one (or more) MC(s) in the system is (are) experiencing more



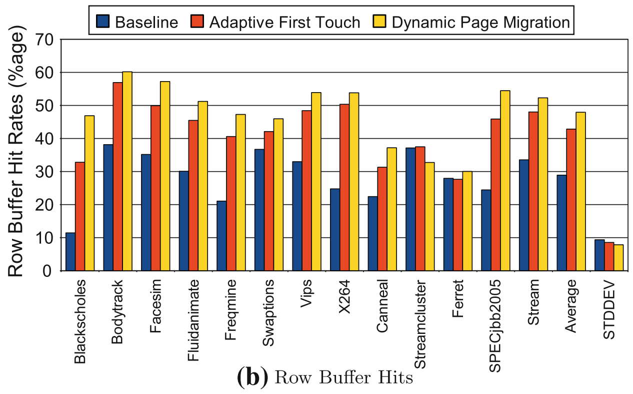


Fig. 5 Adaptive first-touch and dynamic-migration policies versus baseline—homogeneous hierarchy. **a** Relative throughput, **b** Row buffer hits

conflicts than others, hence providing a measure of load across MCs. As compared to the baseline, Adaptive first-touch and dynamic-migration schemes reduce the standard deviation by 8.3% and 21.6% respectively, hence fairly distributing the system DRAM access load across MCs. Increase in row-buffer hit-rates has a direct impact on queuing delays, since a row-buffer hit costs less than a row-buffer miss or conflict, allowing the memory system to be freed sooner to service other pending requests.

For the homogeneous memory hierarchy, Fig. 6 shows the breakdown of total memory latency as a combination of four factors (i) queuing delay (ii) network delay - the extra delay incurred for traveling to a "remote" MC, (iii) device access time, which includes the latency reading(writing) data from(to) the DRAM devices and (iv) data transfer delay. For the baseline, a majority of the total DRAM access stall time (54.2%) is spent waiting in the queue and accessing DRAM devices (25.7%). Since the baseline configuration tries to map a group of physically proximal cores onto an MC, the network delay contribution to the total DRAM access time is comparatively smaller (13.4%). The adaptive policies change the dynamics of this distribution. Since some pages are now mapped to "remote" MCs, the total network delay contribution to the average memory latency goes up (to 18% and 28% for adaptive first-touch and dynamic page migration schemes respectively). Because of increased row-buffer hit rates, the device access time contribution to the overall access latency goes down for

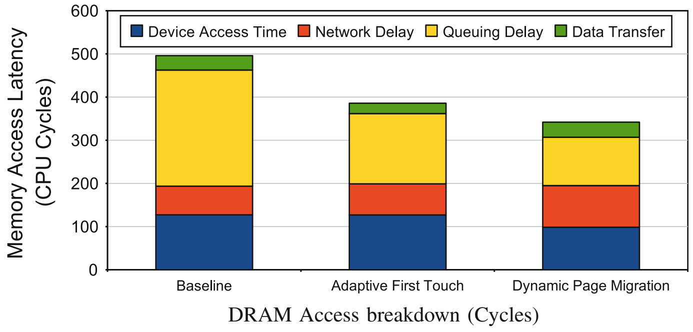


Fig. 6 DRAM access latency breakdown—homogeneous hierarchy

Table 3 Dynamic page migration overhead characteristics

Benchmark	Total number of pages copied (shared/un-shared)	Total cacheline invalidations + Writebacks	Page copying overhead (percent increase in network traffic)
Blackscholes	210 (53/157)	134	5.8%
Bodytrack	489 (108/381)	365	3.2%
Facesim	310 (89/221)	211	4.1%
Fluidanimate	912 (601/311)	2687	12.6%
Freqmine	589 (100/489)	856	5.2%
Swaptions	726 (58/668)	118	2.4%
Vips	998 (127/871)	232	5.6%
X264	1007 (112/895)	298	8.1%
Canneal	223 (28/195)	89	2.1%
Streamcluster	1284 (967/317)	3018	18.4%
Ferret	1688 (1098/590)	3453	15.9%
SPECjbb2005	1028 (104/924)	499	4.1%
Stream	833 (102/731)	311	3.5%

the proposed policies, (down by 1.5% and 11.1% for adaptive first-touch and dynamic migration respectively), as compared to baseline. As a result, the overall average latency for a DRAM access goes down from 495 cycles to 385 and 342 CPU cycles for adaptive first-touch and dynamic migration policies, respectively.

Table 3 presents the overheads associated with the dynamic-migration policy. Applications which experience a higher percentage of shared-page migration (fluidanimate, streamcluster and ferret) tend to have higher overheads. Compared to baseline, the three aforementioned applications see an average of 13.5% increase in network traffic as compared an average 4.2% increase between the rest. Because of higher costs of shared-page migration, these applications also have a higher number of cacheline invalidations and writebacks.

4.3 Sensitivity Analysis

Figure 7 compares the effects of proposed policies for a different physical layout of MCs for the homogeneous DRAM hierarchy. As opposed to earlier, these configurations assume MCs being located at the *center* of the chip than periphery (similar to layouts assumed in [1]). We compare the baseline, adaptive first-touch (AFT) and dynamic migration (DM) policies for both the layouts : *periphery* and *center*. For almost all workloads, we find that baseline and AFT policies are largely agnostic to choice of MC layout. Being a data-centric scheme, dynamic migration benefits a little from the new layout. Although, due to the reduction in the number of hops, DM-Center performs marginally better than DM-periphery.

4.3.1 Effects of TLB Shootdowns

To study the performance impact of TLB shootdowns in Dynamic Migration scheme, we increased the cost of *each* TLB shootdown from 5000 cycles (as assumed previously) to 7500, 10,000 and 20,000 cycles. Since shootdowns are fairly uncommon, and happen only at epoch boundaries, the average degradation in performance in going from 5000 to 20,000 cycles across all applications is 5.8%. For the three applications that have significant sharing among threads (ferret, streamcluster, fluidanimate), the average performance degradation for the same jump is a little higher, at 6.8%.

4.3.2 Results for Multi-Socket Configurations

To test the efficacy of our proposals in the context of multi-socket configurations, we carried out experiments with a configuration similar to one assumed in Fig. 1a. In these experiments, we assume a 4-socket system; each socket housing a quad-core chip, with similar configuration as assumed in Table 2. Each quad-core incorporates one on-chip MC which is responsible for a quarter of the total physical address space.

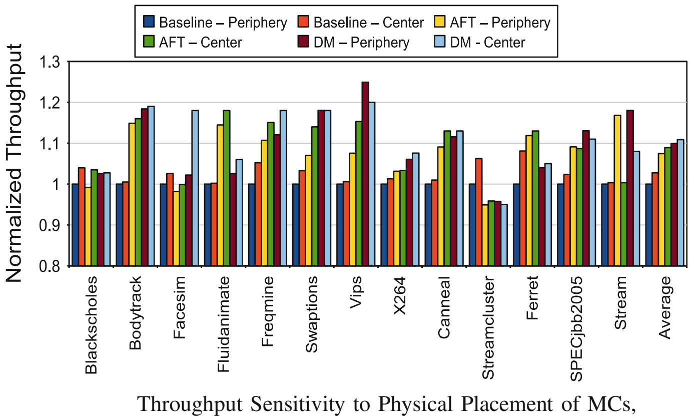


Fig. 7 Sensitivity Analysis, Dynamic page migration policy, Homogeneous hierarchy

Each quad-core has similar L1s as listed in Table 2, but the 2 MB L2 is equally divided among all sockets, with each quad-core receiving 512 KB L2. The inter-socket latencies are based on the observations in [47] (48 ns). The baseline, as before, is assumed to be where the OS is responsible for making page placement decisions. The weights of the cost function are also adjusted to place more weight to $distance_j$, when picking *donor* MCs.

We find that adaptive first-touch is not as effective as the earlier, with performance benefits of 1% over baseline. For the dynamic migration policy, to reduce the overheads of data copying over higher latency inter-socket links, we chose to migrate 5 pages at a time. Even with these optimizations, the overall improvement in system throughput was 1.3%. We attribute this to the increased latency of cacheline invalidations and copying data over inter-socket links.

4.4 Multiple Memory Controllers: Heterogeneous Hierarchy

In this study, a heterogeneous memory hierarchy is assumed to comprise different types of memory devices. These devices could be a mix of different flavors of DRAM (Table 1), or a mixture of different memory technologies, e.g. DRAM and PCM. For the baseline, we assume the default page allocation scheme, i.e. pages are allocated based on one unified free-page list, to the most physically proximal MC, without any consideration for the type of memory technology.

As a first experiment, we divide the total physical address space equally between DDR3 [5] devices and a faster DRAM variant. Such a hierarchy comprising two different kinds of DRAM devices considered in this study (DDR3 and *faster* DRAM) is referred to as *N DRAM - P Fast* hierarchy. For example, 1 DRAM - 3 Fast refers to a hierarchy with 3 MCs controlling faster DRAM DIMMs, while one with DDR3 DIMMs. Likewise, a hierarchy with *N* MCs controlling DRAM devices and *P* MCs controlling PCM devices is referred to as a *N* DRAM - *P* PCM hierarchy.

4.5 Adaptive First-Touch and Dynamic Migration Policies: Heterogeneous Hierarchy

In this section, we try to explore the potential of adaptive first touch and dynamic page migration policies in a heterogeneous memory hierarchy.

First, we consider the case of *N* DRAM - *P* Fast hierarchies. For these experiments, we assume similar storage density of both devices. For Adaptive First Touch policy, the only additional consideration for deciding the relative merit of assigning a page to an MC comes from $LatencyDimmCluster_j$ factor in the cost function.

Figure 8 presents the results of these experiments, normalized to the 2 DRAM - 2 Fast baseline. Despite faster device access times, the ratio of average performance improvement of the proposed polices still remains the same as that for homogeneous hierarchy. For example, for the 1 DRAM - 3 Fast configuration, adaptive first touch and

[5] Unless other specified, all references to *DRAM* refer to DDR3 devices.

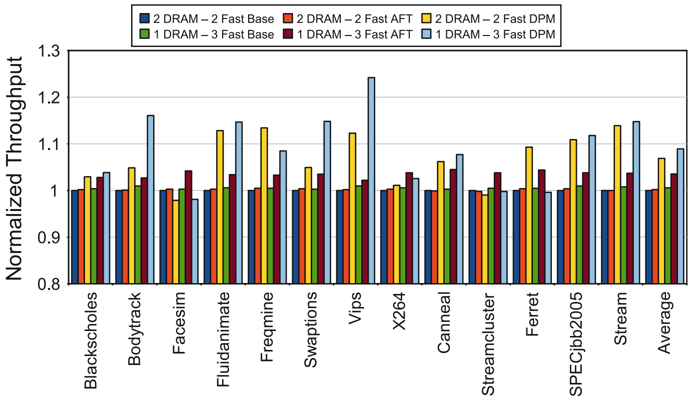


Fig. 8 Impact of proposed policies in heterogeneous memory hierarchy (*N* DRAM - *P* Fast)

and dynamic page migration policies perform 6.2 and 8.3% better than the baseline for the same configuration.

The other heterogeneous memory hierarchy considered in this study is of the *N* DRAM - *P* PCM variety. For these experiments, we assume PCM to be 8 times as dense as DRAM. Also, we statically program the $LatencyDimmCluster_j$ factor to be the worst case (closed-page) access latency of both DRAM and PCM devices (37.5 ns and 80 ns respectively).

Figure 9 presents the throughput results for the different combinations of DRAM and PCM devices. In a 3 DRAM - 1 PCM hierarchy, adaptive first touch and dynamic page migration policies outperform the baseline configuration by 1.6% and 4.4% respectively. Overall, we observe that for a given heterogeneous hierarchy, dynamic page migration tends to perform slightly better than adaptive first touch (2.09% and 2.41% for 3 DRAM - 1 PCM and 2 DRAM - 2 PCM combinations respectively), because adaptive first touch policy places some frequently used pages into PCM devices, increasing the overall access latency. For example, in a 3 DRAM - 1 PCM configuration, 11.2% of the total pages are allocated to PCM address space. This value increases to 16.8% in 2 DRAM - 2 PCM configuration.

4.6 Sensitivity Analysis and Discussion: Heterogeneous Hierarchy

Sensitivity to Physical Placement of MCs For the best performing heterogeneous hierarchy (3 DRAM - 1 PCM), for the baseline, performance is completely agnostic to physical position of MCs. AFT for the same configuration with MCs at the periphery(AFT-periphery), performs 0.52% better with MCs at the center(AFT-center), while DM-periphery performs 0.48% better than DM-center.

Cost of TLB Shootdowns For the best performing heterogeneous (3 DRAM - 1 PCM) hierarchy, there is a greater effect of increased cost of TLB shootdowns in Dynamic Migration scheme. The average degradation in performance in increasing the cost from 5000 cycles to 10,000 cycles is 7.1%. When increased further to 20,000 cycles,

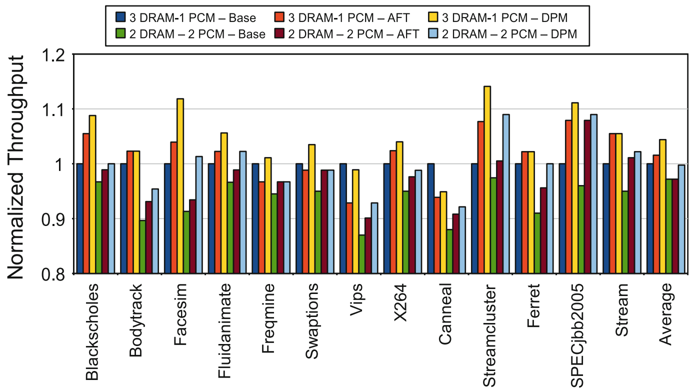


Fig. 9 Impact of proposed policies in heterogeneous memory hierarchy (N DRAM - P PCM)

the workloads exhibit an average degradation of 12.8%, with SpecJBB2005 exhibiting the greatest drop of 17.4%.

5 Related Work

Memory Controllers: Some recent papers [1,36,58,62] examine multiple MCs in a multi-core setting. Blue Gene/P [19] is an example of a production system that employs multiple on-chip MCs. Loh [36] takes advantage of plentiful inter-die bandwidth in a 3D chip that stacks multiple DRAM dies and implements multiple MCs on-chip that can quickly access several fine-grain banks. Vantrease et al. [58] discuss the interaction of MCs with the on-chip network traffic and propose physical layouts for on-chip MCs to reduce network traffic and minimize channel load. The Tile64 processor [62] employs multiple MCs on a single chip, accessible to every core via a specialized on-chip network. The Tile64 microprocessor [62] was also one of the first processors to use multiple (four) on-chip MCs. More recently, Abts et al. [1] explore multiple MC placement on a single chip-multiprocessor so as to minimize on-chip traffic and channel load. None of the above works considers intelligently allocating data and load across multiple MCs. Kim et al. propose ATLAS [27], a memory scheduling algorithm that improves system throughput without requiring significant coordination between the on-chip memory controllers.

Recent papers [44,45] have begun to consider MC scheduler policies for multi-core processors, but only consider a single MC. Since the memory controller is a shared resource, all threads experience a slowdown when running concurrently 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 queuing delays for threads that tend to not access open rows. To deal with such unfairness, they introduce a Stall-Time Fair Memory (STFM) scheduler that estimates the disparity and overrules the prioritization of open row access if necessary. 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. The same authors also introduce a Parallelism-Aware Batch Scheduler (PAR-BS) [45]. The PAR-BS policy first breaks up the request queue into batches based on age and then services a batch entirely before moving to the next batch (this provides a level of fairness). Within a batch, the scheduler attempts to schedule all the requests of a thread simultaneously (to different banks) so that their access latencies can be overlapped. In other words, the scheduler tries to exploit memory-level parallelism (MLP) by looking for bank-level parallelism within a thread. The above described bodies of work are related in that they attempt to alleviate some of the same constraints as us, but not with page placement.

Other MC related work focusing on a single MC include the following. Lee et al. [33] design an MC scheduler that allocates priorities between demand and prefetch requests from the DRAM. Ipek et al. [22] build a reinforcement learning framework to optimize MC scheduler decision-making. Lin et al. [35] design prefetch mechanisms that take advantage of idle banks/channels and spatial locality within open rows. Zhu and Zhang [67] examine MC interference for SMT workloads. They also propose scheduler policies to handle multiple threads and consider different partitions of the memory channel. Cuppu et al. [12,13] study the vast design space of DRAM and memory controller features for a single core processor.

Memory Controllers and Page Allocation Lebeck et al. [31] studied the interaction of page coloring and DRAM power characteristics. They examine how DRAM page allocation can allow the OS to better exploit the DRAM system's power-saving modes. In a related paper [18], they also examine policies to transition DRAM chips to low-power modes based on the nature of access streams seen at the MC. Zhang et al. [64] investigate a page-interleaving mechanism that attempts to spread OS pages in DRAM such that row-buffers are re-used and bank parallelism is encouraged within a single MC.

Page Allocation Page coloring 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 [4,16,25,42,54]. Page coloring and migration have been employed to improve proximity of computation and data in a NUMA multi-processor [6,11,28–30,59] and in NUCA caches [2,10,51]. 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 [12,13,52] that shed insight on the bottlenecks arising from memory controller constraints. Here, we not only apply the well-known concept of page coloring to a different domain, we extend our policies to be cognizant of the several new constraints imposed by DRAM memory schedulers (row-buffer re-use, bank parallelism, queuing delays, etc.). More recently, McCurdy et al. [38] observe that NUMA-aware code could make all the difference in most multi-threaded scientific applications scaling perfectly across multiple sockets, or not at all. They then propose a data-centric tool-set based on performance counters which helps to pin-point problematic memory access, and utilize this information to improve performance.

Task Scheduling The problem of task scheduling onto a myriad of resources has been well studied, although not in the context of multiple on-chip MCs. While the problem formulations are similar to our work, the constraints of memory controller scheduling are different. Snavely et al. [55] schedule tasks from a pending task queue on to a number of available thread contexts in an SMT processor. Zhou et al. [66] schedule tasks on a 3D processor in an attempt to minimize thermal emergencies. Similarly, Powell et al. [49] attempt to minimize temperature by mapping a set of tasks to a CMP comprised of SMT cores.

6 Conclusions

This paper proposes a substantial shift in DRAM data placement policies which must become cognizant of both the performance characteristics and load on individual NUMA nodes in a system. We are headed for an era where a large number of programs will have to share limited off-chip bandwidth through a moderate number of on-chip memory controllers. While recent studies have examined the problem of fairness and throughput improvements for a workload mix sharing a single memory controller, this is the first body of work to examine data-placement issues for a many-core processor with a moderate number of memory controllers. We define a methodology to compute an optimized assignment of a thread's data to memory controllers based on current system state. We achieve efficient data placement by modifying the OS' frame allocation algorithm. We then improve on this first touch policy by dynamically migrating data within the DRAM sub-system to achieve lower memory access latencies across multiple program phases of an application's execution.

These dynamic schemes adapt with current system state and allow spreading a single program's working set across multiple memory controllers to achieve better aggregate throughput via effective load balancing. Our proposals yield improvements of 6.5% (when assigning pages on first touch), and 8.9% (when allowing pages to be migrated across memory controllers).

As part of our future work we intend to investigate further improvements to our original design, for example, considering additional memory scheduler constraints (intra-thread parallelism, handling of prefetch requests, etc.). Shared pages in multi-threaded applications may benefit from a placement algorithm that takes the sharing pattern into account. Page placement to promote bank parallelism in this context also remains an open problem.

References

1. Abts, D., Jerger, N., Kim, J., Gibson, D., Lipasti, M.: Achieving predictable performance through better memory controller in many-core CMPs. In: Proceedings of ISCA (2009)
2. Awasthi, M., Sudan, K., Balasubramonian, R., Carter, J.: Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In: Proceedings of HPCA (2009)
3. Benia, C., et al.: The PARSEC benchmark suite: characterization and architectural implications. Technical report, Department of Computer Science, Princeton University (2008)
4. Bershad, B., Chen, B., Lee, D., Romer, T.: Avoiding conflict misses dynamically in large direct-mapped caches. In: Proceedings of ASPLOS (1994)
5. Burr, G.W., Breitwisch, M.J., Franceschini, M., Garetto, D., Gopalakrishnan, K., Jackson, B., Kurdi, B., Lam, C., Lastras, L.A., Padilla, A., Rajendran, B., Raoux, S., Shenoy, R.S.: Phase Change Memory Technology. (2010). http://arxiv.org/abs/1001.1164v1

6. Chandra, R., Devine, S., Verghese, B., Gupta, A., Rosenblum, M.: Scheduling and page migration for multiprocessor compute servers. In: Proceedings of ASPLOS (1994)
7. Chang, J., Sohi, G.: Co-operative caching for chip multiprocessors. In: Proceedings of ISCA (2006)
8. Chaudhuri, M.: PageNUCA: selected policies for page-grain locality management in large shared chip-multiprocessor caches. In: Proceedings of HPCA (2009)
9. Chishti, Z., Powell, M., Vijaykumar, T.: Optimizing replication, communication, and capacity allocation in CMPs. In: Proceedings of ISCA-32 (June 2005)
10. Cho, S., Jin, L.: Managing distributed, shared L2 caches through OS-level page allocation. In: Proceedings of MICRO (2006)
11. Corbalan, J., Martorell X., Labarta J.: Page Migration with dynamic space-sharing scheduling policies: the case of SGI 02000. Int. J. Parallel Prog. **32**(4) (2004)
12. Cuppu, V., Jacob, B.: Concurrency, latency, or system overhead: which has the largest impact on uniprocessor DRAM-System performance. In: Proceedings of ISCA (2001)
13. Cuppu, V., Jacob, B., Davis, B., Mudge, T.: A performance comparison of contemporary DRAM architectures. In: Proceedings of ISCA (1999)
14. Dally, W.: Report from Workshop on On- and Off-Chip Interconnection Networks for Multicore Systems (OCIN). (2006). http://www.ece.ucdavis.edu/~ocin06/
15. Deng, Q., Meisner, D., Ramos, L., Wenisch, T., Bianchini, R.: MemScale: active low-power modes for main memory. In: Proceedings of ASPLOS (2011)
16. Ding, X., Nikopoulosi, D.S., Jiang, S., Zhang, X.: MESA: Reducing cache conflicts by integrating static and run-time methods. In: Proceedings of ISPASS (2006)
17. Dybdahl, H., Stenstrom, P.: An adaptive shared/private NUCA cache partitioning scheme for chip multiprocessors. In: Proceedings of HPCA (2007)
18. Fan, X., Zeng, H., Ellis, C.: Memory controller policies for DRAM power management. In: Proceedings of ISLPED (2001)
19. Gara, A., Blumrich, M.A., Chen, D., Chiu, G.L.-T., Coteus, P., Giampapa, M.E., Haring, R.A., Heidelberger, P., Hoenicke, D., Kopcsay, G.V., Liebsch, T.A., Ohmacht, M., Steinmacher-Burow, B.D., Takken, T., Vranas, P.: Overview of the blue gene/l system architecture. IBM J. Res. Dev. **49** (2005)
20. Hardavellas, N., Ferdman, M., Falsafi, B., Ailamaki, A.: Reactive NUCA: near-optimal block placement and replication in distributed caches. In: Proceedings of ISCA (2009)
21. Intel 845G/845GL/845GV Chipset Datasheet: Intel 82845G/82845GL/82845GV Graphics and Memory Controller Hub (GMCH) (2002)
22. Ipek, E., Mutlu, O., Martinez, J., Caruana, R.: Self optimizing memory controllers: a reinforcement learning approach. In: Proceedings of ISCA (2008)
23. ITRS. International Technology Roadmap for Semiconductors, 2007 Edition
24. Jacob, B., Ng, S.W., Wang, D.T.: Memory systems—cache, DRAM disk. Elsevier, New York (2008)
25. Kessler, R.E., Hill, M.D.: Page placement algorithms for large real-indexed caches. ACM Trans. Comput. Syst. **10**(4) (1992)
26. Kim, C., Burger, D., Keckler, S.: An Adaptive, non-uniform cache structure for wire-dominated on-chip caches. In: Proceedings of ASPLOS (2002)
27. Kim, Y., Han, D., Mutlu, O., Harchol-Balter, M.: ATLAS: a scalable and high-performance scheduling algorithm for multiple memory controllers. In: Proceedings of HPCA (2010)
28. LaRowe, R., Ellis, C.: Experimental comparison of memory management policies for NUMA multiprocessors. Technical report (1990)
29. LaRowe, R., Ellis, C.: Page placement policies for NUMA multiprocessors. J. Parallel Distrib. Comput. **11**(2) (1991)
30. LaRowe, R., Wilkes, J., Ellis, C.: Exploiting operating system support for dynamic page placement on a NUMA shared memory multiprocessor. In: Proceedings of PPOPP (1991)
31. Lebeck, A., Fan, X., Zeng, H., Ellis, C.: Power aware page allocation. In: Proceedings of ASPLOS (2000)
32. Lee, B., Ipek, E., Mutlu, O., Burger, D.: Architecting phase change memory as a scalable DRAM alternative. In: Proceedings of ISCA (2009)
33. Lee, C., Mutlu, O., Narasiman, V., Patt, Y.: Prefetch-aware DRAM controllers. In: Proceedings of MICRO (2008)
34. Lin, J., Lu, Q., Ding, X., Zhang, Z., Zhang, X., Sadayappan, P.: Gaining insights into multicore cache partitioning: bridging the gap between simulation and real systems. In: Proceedings of HPCA (2008)

35. Lin, W., Reinhardt, S., Burger, D.: Designing a Modern memory hierarchy with hardware prefetching. In: Proceedings of IEEE transactions on computers (2001)
36. Loh, G.: 3D-stacked memory architectures for multi-core processors. In: Proceedings of ISCA (2008)
37. Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: a full system simulation platform. IEEE Comput. **35**(2), 50–58 (2002)
38. McCurdy, C., Vetter, J.: Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In: Proceedings of ISPASS (2010)
39. Micron DDR3 SDRAM Part MT41J512M4.(2006) http://download.micron.com/pdf/datasheets/dram/ddr3/2Gb_DDR3_SDRAM.pdf,
40. Micron Technology Inc. Micron DDR2 SDRAM Part MT47H64M8. (2004)
41. Micron Technology Inc. Micron DDR2 SDRAM Part MT47H128M8HQ-25. (2007)
42. Min, R., Hu, Y.: Improving performance of large physically indexed caches by decoupling memory addresses from cache addresses. IEEE Trans. Comput. **50**(11) (2001)
43. Muralimanohar, N., Balasubramonian, R., Jouppi, N.: Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In: Proceedings of MICRO (2007)
44. Mutlu, O., Moscibroda, T.: Stall-time fair memory access scheduling for chip multiprocessors. In: Proceedings of MICRO (2007)
45. Mutlu, O., Moscibroda, T.: Parallelism-aware batch scheduling: enhancing both performance and fairness of shared DRAM systems. In: Proceedings of ISCA (2008)
46. Perfmon2 Project Homepage. http://perfmon2.sourceforge.net/
47. Performance of the AMD Opteron LS21 for IBM BladeCenter. ftp://ftp.software.ibm.com/eserver/benchmarks/wp_ls21_081506.pdf
48. Phadke, S., Narayanasamy, S.: MLP-aware Heterogeneous Main Memory. In: Proceedings of DATE (2011)
49. Powell, M., Gomaa, M., Vijaykumar, T.: Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In: Proceedings of ASPLOS (2004)
50. Qureshi, M.K.: Adaptive spill-receive for robust high-performance caching in CMPs. In: Proceedings of HPCA (2009)
51. Rafique, N., Lim, W., Thottethodi, M.: Architectural support for operating system driven CMP cache management. In: Proceedings of PACT (2006)
52. Rixner, S., Dally, W., Kapasi, U., Mattson, P., Owens, J.: Memory access scheduling. In: Proceedings of ISCA (2000)
53. Romanchenko, V.: Quad-Core Opteron: Architecture and Roadmaps. http://www.digital-daily.com/cpu/quad_core_opteron
54. Sherwood, T., Calder, B., Emer, J.: Reducing cache misses using hardware and software page placement. In: Proceedings of SC (1999)
55. Snavely, A., Tullsen, D., Voelker, G.: Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In: Proceedings of SIGMETRICS (2002)
56. Speight, E., Shafi, H., Zhang, L., Rajamony, R.: Adaptive mechanisms and policies for managing cache hierarchies in chip multiprocessors. In: Proceedings of ISCA (2005)
57. Swinburne, R.: Intel Core i7—Nehalem Architecture Dive. http://www.bit-tech.net/hardware/2008/11/03/intel-core-i7-nehalem-architecture-dive/
58. Vantrease, D., et al.: Corona: system implications of emerging nanophotonic technology. In: Proceedings of ISCA (2008)
59. Verghese, B., Devine, S., Gupta, A., Rosenblum, M.: Operating system support for improving data locality on CC-NUMA compute servers. SIGPLAN Not. **31**(9) (1996)
60. Wallin, D., Zeffer, H., Karlsson, M., Hagersten, E.: VASA: a simulator infrastructure with adjustable fidelity. In: Proceedings of IASTED International Conference on Parallel and Distributed Computing and Systems (2005)
61. Wang, D., et al.: DRAMsim: A memory-system simulator. In: SIGARCH Computer Architecture News (September 2005)
62. Wentzlaff, D., et al.: On-Chip Interconnection Architecture of the Tile Processor. In: IEEE Micro **22**, (2007)
63. Zhang, M., Asanovic, K.: Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors. In: Proceedings of ISCA (2005)
64. Zhang, Z., Zhu, Z., Zhand, X.: A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In: Proceedings of MICRO (2000)

65. Zheng, H., et al.: Mini-Rank: Adaptive DRAM architecture for improving memory power efficiency. In: Proceedings of MICRO (2008)
66. Zhou, X., Xu, Y., Du, Y., Zhang, Y., Yang, J.: Thermal management for 3D processor via task scheduling. In: Proceedings of ICPP (2008)
67. Zhu, Z., Zhang, Z.: A Performance comparison of DRAM memory system optimizations for SMT processors. In: Proceedings of HPCA (2005)