Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers *

Manu Awasthi, David Nellans, Kshitij Sudan,
Rajeev Balasubramonian, Al Davis

School of Computing, University of Utah

{manua, dnellans, kshitij, rajeev, ald} @cs.utah.edu

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 piece of the physical memory. Data placement will therefore become increasingly critical in minimizing memory access latencies.

To date, no prior work has examined the effects of data placement among multiple MCs in such systems. Future chip-multiprocessors are likely to comprise multiple MCs and an even larger number of cores. This trend will increase the memory access latency variation in these systems. Proper allocation of workload data to the appropriate MC will be important in reducing the latency of memory service 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 the physical memory. We introduce adaptive first-touch page placement, and dynamic page-migration mechanisms to reduce DRAM access delays for multi-MC systems. These policies yield average performance improvements of 17% for adaptive first-touch page-placement, and 35% for a dynamic page-migration policy.

*This work was supported in parts by NSF grants CCF-0430063, CCF-0811249, CCF-0916436, NSF CAREER award CCF-0545959, SRC grant 1847.001, Intel, HP, and the University of Utah.

PACT'10, September 11–15, 2010, Vienna, Austria.

Categories and Subject Descriptors

B.3.1 [**Memory Structures**]: Semiconductor Memories–DRAM; B.3.2 [**Memory Structures**]: Design Styles–Shared Memory

General Terms

Design, Performance, Experimentation

Keywords

DRAM Management, Data Placement, Memory Controller Design

1. INTRODUCTION

Modern microprocessors increasingly integrate the *memory controller (MC)* on-chip in order to reduce main memory access latency. Memory pressure will increase with increasing core-counts per socket and a single MC will quickly become a bottleneck. In order to avoid this problem, modern multi-core processors (chip multiprocessors, CMPs) have begun to integrate multiple MCs per socket [46, 50, 55]. 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 MC's is likely to continue and exploiting MC locality will be critical to aggregate system performance.

Recent efforts [5, 34, 51, 55] have incorporated multiple MCs in their designs, but there is little evidence on how data placement should be managed and how a particular placement policy will affect main memory access latencies. In addressing this problem, we note 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, etc. 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 of performance.

To our knowledge, this is the first attempt at intelligent data placement in a multi-MC platform. We note however the similarities in previous efforts to optimize data place-

ment in last-level shared NUCA caches [6, 10–13, 19, 21, 32, 43, 49, 56]. However, the key difference is that caches tend to be capacity limited, while DRAM access delays are governed primarily by other issues such as long queuing delays and row buffer hit rates. To the best of our knowledge, there are only a handful of papers that explore challenges faced in the context of multiple on-chip MCs. Abts et al. [5] explore the physical layout of multiple on-chip MCs to reduce contention in the on-chip interconnect. The optimal layout makes the performance of memory-bound applications predictable, regardless of which core they are scheduled on. Kim et al. [26] propose a new scheduling policy in the context of multiple MCs which requires minimal coordination between MCs. However both of these proposals do not consider how data should be distributed in a NUMA setting while taking into account the complex interactions of row-buffer hit rates, queuing delays, on-chip network traffic, etc. Our work takes these DRAM-specific phenomena into account and explores both first-touch page placement and dynamic page-migration to reduce access delays. We show average performance improvements of 17% with an adaptive first-touch page-coloring policy, and 35% with a dynamic page-migration policy.

The paper is organized as follows. We provide background and motivational discussion in Section 2. Section 3 details the adaptive first-touch and dynamic migration policies and Section 4 provides quantitative comparison of the proposed policies. We discuss related work in Section 5 and conclude in Section 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 addresspath, and an 8-bit command/control-path [37]. 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.

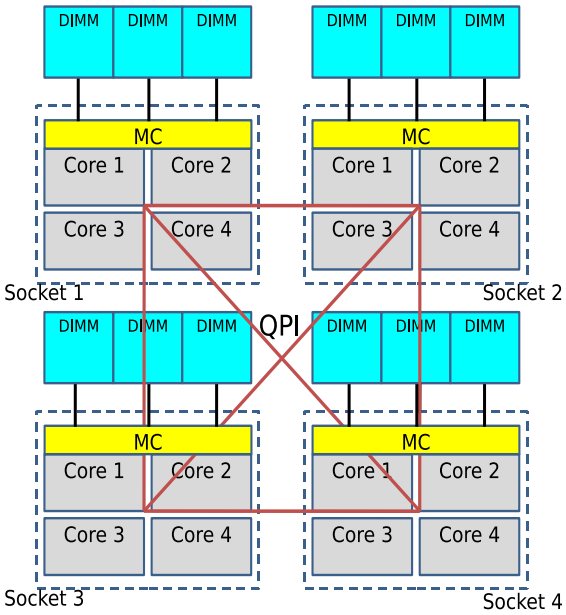
Commodity DRAMs are very cost sensitive and have been optimized to minimize the cost/bit. Therefore, an orthogonal 2-part addressing scheme is utilized where row and column addresses are multiplexed on the 17-bit address channel. The MC first generates the *row address* that causes an entire row of the 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 the most favored one. The 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.

The MC typically has a queue of pending requests and schedules the next request while dealing with timing constraints, bank constraints, and priorities. A widely adopted scheduling policy is FR-FCFS (First Ready - First Come First Serve) [45] that prioritizes requests to open rows and breaks ties based on age. In modern architectures with multiple MCs, each MC has a dedicated channel to the DIMMs it controls.

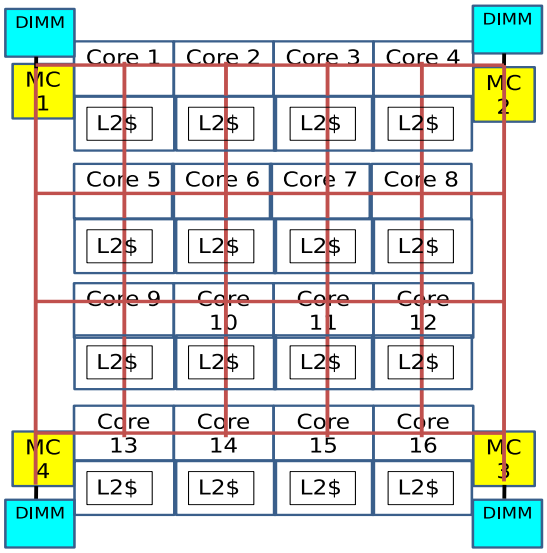
2.2 Current/Future Trends

Several commercial designs have not only moved the MC on chip, but have also integrated multiple MCs on a single multi-core die. For example, Intel's Nehalem processor [50] shown in Figure 1(a) 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 Northbridge. 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 using one MC [46]. The Tile64 processor [55] 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 placement details are not publicly available. The Corona architecture from HP [51] 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 dictates 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, as shown in Section 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, as shown in Figure 1(b). Sixteen cores share four MCs that are uniformly distributed at the edge of the chip. Given that on-chip wire delays are important constraints, this layout helps reduce MC to I/O pin or core distance.



(a) The logical organization of a multi-socket Nehalem.



(b) The 16-core 4-MC model assumed in this study.

Figure 1: Platforms with multiple memory controllers.

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 [10], page spilling [13], and their derivatives [6, 11, 12, 19, 21, 32, 43, 49, 56] being the most commonly known examples. There has been little prior work on OS-based page coloring to place pages in different DIMMs or banks to promote either DIMM- or bank-level parallelism (Zhang et al. [57] proposed to employ 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 this is not the case.

As mentioned before, the NUMA factor in a modern system can be as high as 1.5 [50]. 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 of delay to send requests across the length of the chip [17, 55] which further increases the NUMA disparity.

Pin count restrictions prevent the MC count from increasing linearly with the number of cores, while maintaining a constant channel count/width per MC. Hence, the number of cores serviced by each MC will continue to rise, leading to non-trivial contention and long queuing delays. Many recent studies [22, 31, 40, 41, 58] have identified MC queuing delays as a major bottleneck and have proposed novel mechanisms to improve scheduling policies. To verify these claims, we carried out experiments for the configuration parameters described in Section 4 for a 16 core system with 1

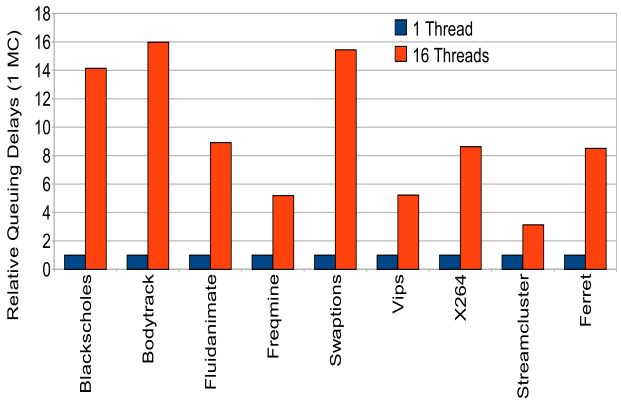


Figure 2: Relative Queuing Delays for 1 and 16 threads, single MC, 16 cores

on-chip MC. Each application was run with a single thread and then again with 16 threads, with one thread pinned on every core. Figure 2 shows the average queuing delays experienced by both the configurations; the 16 thread configuration is normalized against the single thread case of the workload. The average queuing delay across 16-threads as compared to a run with just one thread is higher by an order of magnitude, sometimes being as high as 16x (*Bodytrack*). For some applications, the average number of cycles spent by a request buffered in the MC queues can be as high as 280 CPU cycles for the 16-thread case. This constitutes a significant fraction of the average time to service a memory request (530 cycles), and hence makes a strong case for considering queuing delays in optimizing data placement.

It is also important to maximize row-buffer hit-rates. Not

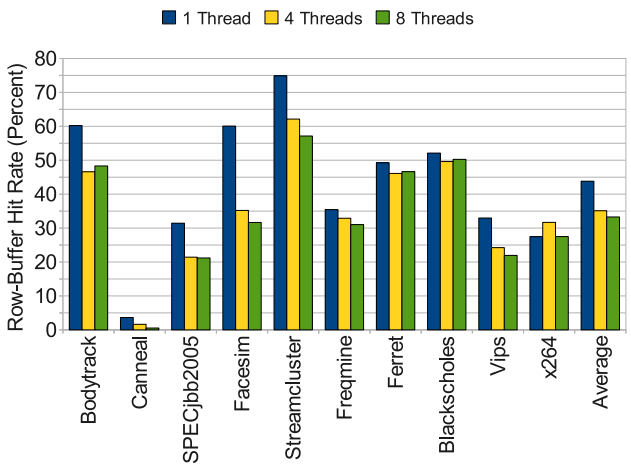


Figure 3: Row-Buffer Hit Rates, Dual-Socket, Quad-Core Opteron

accounting for system overheads, for our choice of simulation parameters, row-buffer hits can return data in 25 DRAM cycles, while row-buffer conflicts require at least 75 DRAM cycles for a DDR3-1333 memory and a 3.0 GHz CPU. Figure 3 shows row-buffer hit-rates for a variety of applications when running in 1-, 4-, and 8-thread modes. These measurements were made using hardware performance counters [1] on a dual-socket, quad-core AMD Opteron 2344HE system with 16 2-GB DIMMs. These results exhibit the variation in row-buffer hit-rates among applications. More importantly, with increased thread counts, we observe that hit-rates decrease substantially due to row-buffer conflicts. We conclude that proper allocation of data in multi-MC systems can have a non-trivial impact on row-buffer hit rates and hence performance.

The three most important observations that we make from the above discussion are: (i) the NUMA factor is likely to increase; (ii) more threads per MC leads to high contention, raising the importance of system overheads such as queuing delays; (iii) increased interleaving of memory accesses from different threads leads to reduced row-buffer hit rates. It is imperative to be cognizant of all three factors when allocating data to an MC's 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 Figure 1(b). Each MC handles a distinct subset of the aggregate physical address space. 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 [25, 56]. 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 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 some MC and a particular DRAM channel. Proper assignment will minimize access latency to that page and not significantly degrade accesses to other pages assigned in 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. We therefore need estimates of how each of these factors will be impacted by assigning a page to an MC. Profiles generated beforehand can help in page assignment, but this is almost never practical. We therefore focus on policies that rely on run-time estimation of application behavior.

We describe two schemes to reduce memory access delays: *adaptive first-touch placement* of pages, and *dynamic migration of data* among DIMMs at OS page granularity. The first scheme is based on DRAM frame allocation by the OS while being aware of factors like MC load (queuing delays, row-buffer hit-rates and bank contention), and 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 when a page-fault occurs. The second scheme aims at dynamically migrating data at run-time to reduce access delays. This migration of data occurs during spare cycles in the background, and we propose mechanisms to prevent stalls at the CPU while data migration takes place.

3.1 Adaptive First-Touch (AFT) 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 schedule tasks to cores. Such an approach is likely more complex than simply managing how a thread's working set gets organized across MCs. Besides, clever task scheduling must rely on pre-computed profiles that are sometimes inaccurate and closely tied to the behavior of co-scheduled applications. For these reasons, we believe that our adaptive first-touch approach, that is based on run-time statistics, can out-perform a heuristic-based schedule using application profiles.

In our adaptive first-touch approach for page allocation, when a thread starts executing on some core, each new page it touches causes 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 hope is that most pages of a thread will be mapped to the nearest MC, with a few pages being spilled to other nearby MCs, if necessary. The following cost function is

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

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 rate of row-buffer hits 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 hops that need to be traversed. The role of *load* and *distance* is straightforward; the row buffer hit rate is considered because the assumption is that the new page will be less disruptive to other accesses if it resides in a DIMM with an already high row buffer miss 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 stop touching 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 correct this. The dynamic migration policy starts out as the AFT policy described above. During the course of the program execution, if an imbalance is detected in how MCs are being accessed, it may be necessary to migrate pages from a heavily loaded MC to a lightly loaded one. If we see a substantial difference between the highest-loaded and other MCs, 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]. 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 to the donor MC, and (ii) has the lowest number of page-conflicts in the last epoch. Hence for each MC k in the *recipient* pool, we calculate

$$cost_k = \Lambda \times distance_k + \Gamma \times page\_conflicts_k$$

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.

[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.

It is perhaps possible to be more selective regarding the choice of pages and the choice of new MC, but we resort to this simple low-overhead policy. Even when the dynamic migration policy kicks in, new incoming pages' MCs are decided by the AFT cost function. Pages that have been migrated once are not considered for re-migration for the next two epochs.

To maintain correctness, certain steps need to be taken for all pages that are being migrated:

1. **TLB Update:** TLBs in all cores have to be informed of the change in the page's physical address. Before a page is physically migrated, the corresponding TLB entries have to be invalidated.

2. **Cache Invalidations :** The cache lines belonging to the migrated pages have to be invalidated across all cores. Since the cache is S-NUCA, only one location has to be looked up for each cache line. When invalidating the lines, copies in L1 must also be invalidated. The L2 copy typically maintains a directory to keep track of these L1 lines. Any dirty lines being invalidated have to be written back to memory prior to migration.

Both of these steps are potentially costly in terms of both power and performance. The key is to reduce this cost as much as possible and to understand when the benefits of migration will save more than the cost of migration.

The writeback for dirty cache lines is required when a migration moves pages in DRAM. Instead of immediately flushing the dirty lines, we propose a delayed write-back scheme. Instead of immediately invalidating TLB entries, this is deferred until the entire page has been copied. Any read requests from the CPU for the old address are still serviced from the old location. Once the actual DRAM copy associated with migration is complete, the TLB entries are invalidated, dirty lines in the cache are flushed and written back at the new memory address, and the page is added to the free-page list. This method of delaying TLB shootdowns is referred to as *lazy-copying* in later sections.

3.2.1 Discussion

Employing the proposed policies incurs some system-level (hardware and OS) overheads, which we enumerate in this section. The hardware (MC) needs to maintain counters to keep track of per-workload delay and access counts. Most of the modern processors already come with counters for measuring memory system events (Row-Hits/Misses/Conflicts) [1]. 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 trivially. Based on DRAM device specifications and measured memory system hits/ misses/ conflicts, we can measure average time spent accessing the devices. The difference between the measured memory latency and the time spent accessing devices can provide an accurate estimate of *load*. We expect that future systems can easily include hardware counters to directly measure *load*.

Also, copying pages in memory requires a trap into the OS to update page-tables. Since none of these operations are on the critical path, they do not lead to performance

degradation. However, to demonstrate the impact of simple mechanisms where the cost of invalidating the current TLB entry is incurred with every page migration (resulting in a TLB miss at the next access and an ensuing page-walk), we include these costs in our experiments. We also include the cost of packetizing the page data to be copied and sent over the on-chip network in our experiments. Thus, we account for all performance degrading features of the most simplified scheme that implements the above policy.

4. RESULTS

The full system simulations are built upon the Virtutech Simics [35] 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 [4]. 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 1. 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.

DRAM address mapping parameters for our platform were adopted from the DRAMSim framework [54]. We implemented basic SDRAM mapping, as found in user-upgradeable memory systems, (similar to Intel 845G chipsets' DDR SDRAM mapping [3]). Some platform specific implementation suggestions were taken from the VASA framework [53]. 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 [39]. 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 [7] (with sim-large working set), SPECjbb2005 (with number of warehouses equal to number of cores) and Stream benchmark (number of threads equal to number of cores). For all experiments involving dynamic page migration, we migrate 10 pages (N, section 3) from each MC[3], per epoch, which is assumed to be 5 million cycles. We (pessimitically) 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 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 desribed in Section 2.3 for PARSEC and commercial workloads.

[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.

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 queueing 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

First we study the effect of multiple MCs on the overall system performance (Figure 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 (Figure 1(b)). 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 65% and improve row buffer hits by 55%, resulting in an overall throughput gain of 41.2%. 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, *ferret* experiences increased conflicts at MC numbers 2 and 6, with the row buffer hit rates going down by 15.8%, increasing the average queuing delay by 20.2%. 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.

4.3 Adaptive First-Touch and Dynamic Migration Policies

Figure 5 compares the average throughput improvement of adaptive first-touch and dynamic-migration policies over the baseline. On an average, over all the workloads, adaptive first-touch and dynamic page-migration perform 17.1% and 34.8% better than the baseline, respectively. Part of

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

ISA	UltraSPARC III ISA	CMP size and Core Freq.	16-core, 3 GHz
L1 I-cache	32KB/2-way, 1-cycle	L1 D-cache	32KB/2-way, 1-cycle
L2 Cache (shared)	2 MB/8-way, 3-cycle/bank access	L1/L2 Cache line size	64 Bytes
Hop Access time (Vertical and Horizontal)	2 cycles	Router Overhead Page Size	3 cycles 4 KB
Processor frequency	3 GHz	On-Chip Network frequency	3 GHz
On-chip network width	64 bits	Coherence Protocol	MESI

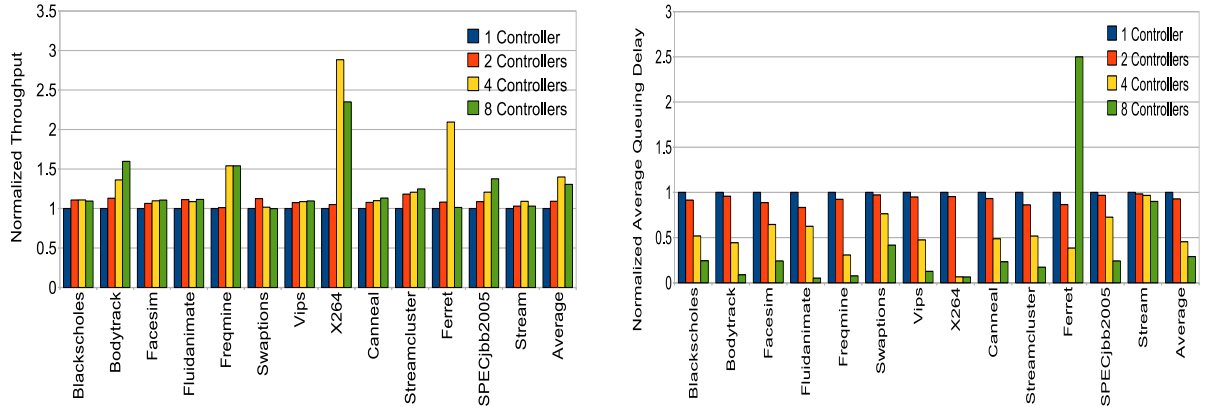
DRAM Parameters

DRAM Device Parameters	Micron MT47H64M8 DDR3-1333 Timing parameters [4], t_{CL}=t_{RCD}=t_{RP}=(10-10-10 @ 800 MHz)
	4 banks/device, 16384 rows/bank, 512 columns/row, 32 bits/column, 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	32 bits/column × 512 columns/row × 8 devices/DIMM = 8 KB/DIMM
Active row-buffers per DIMM	4 (each bank in a device maintains a row-buffer)
Total DRAM Capacity	512 MBit/device × 8 devices/DIMM × 8 DIMMs = 4 GB
Burst length	8 (for 1 MC, reduces proportionally for each new MC)

Values of Cost Function Constants

$\alpha, \beta, \lambda, \Lambda, \Gamma$	10, 20, 100, 100, 100

Table 1: Simulator parameters.



a. Number of Controllers vs. Throughput

b. Number of Controllers vs Avg. Queue Lengths

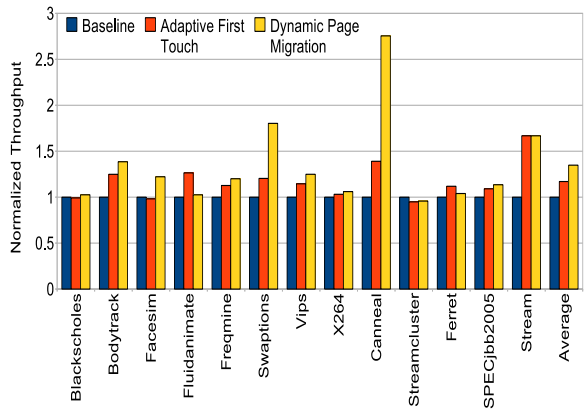
Figure 4: Impact of Multiple Memory Controllers

this improvement comes from the intelligent mapping of pages to improve row-buffer hit rates, which are improved by 16.6% and 22.7% respectively for first-touch and dynamic-migration policies. The last cluster in Figure 5(b) (STD-DEV) 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 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 48.2% and 60.9% 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.

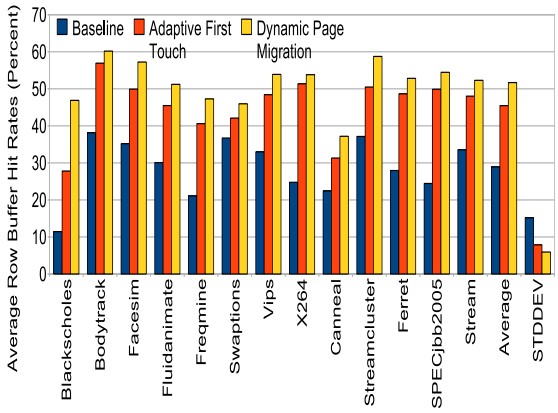
Figure 6 shows the breakdown of total memory latency as a combination of four factors (i) queuing delay (ii) network delay - the extra delay incurred for travelling to a "remote" MC, (iii) device access time, which includes the latency reading(writing) data from(to) the DRAM devices and (iv) data transfer delay. For the baseline, a majority of the total DRAM access stall time (52.6%) is spent waiting in the queue and accessing DRAM devices (28%). 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 (13.1%). 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 22.1% and 33.5% for adaptive first-touch and dynamic page migration schemes respectively). Because of increased row-buffer hit rates, the device access time contribution goes down for the proposed policies, (down by 1.2 and 7.1% respectively 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 530 cycles to 325 and 258 cpu cycles for adaptive first-touch and dynamic migration policies, respectively.

4.4 Adaptive Policy Overheads

Table 2 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 15.6% increase in network traffic as compared an average 4.4% increase between the rest. Because of higher costs of shared-page migration, these applications also have a higher number of cacheline invalidations and writebacks.

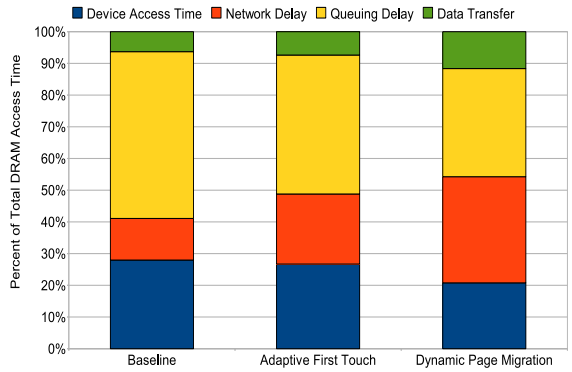


a. Relative throughput

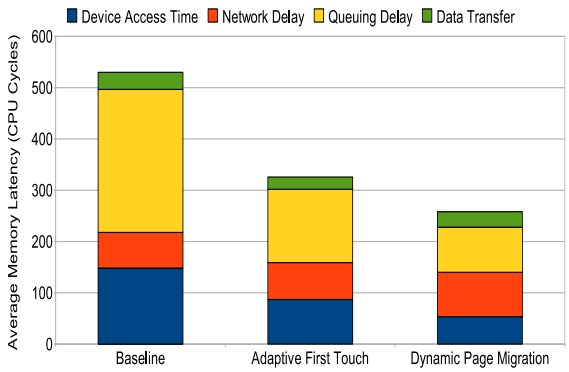


b. Row Buffer Hits

Figure 5: Adaptive First-Touch and Dynamic-Migration Policies vs Baseline



a. DRAM Access breakdown (Percentage)



b. DRAM Access breakdown (Cycles)

Figure 6: DRAM Access Breakdown

Applications with a high number of total migrated pages, but a small percentage of shared ones benefit a lot from the lazy-copying scheme described in section 3.1. As compared to a dynamic page migration policy *without* lazy page-copying, a dynamic page migration policy *with* lazy page-copying performs 8.1% better on average, across all benchmarks.

4.5 Sensitivity Analysis

We perform experiments to characterize performance sensitivity of workloads to individual terms in the cost function. To do so, in our dynamic migration policy, we change the cost function to use only one factor at a time and compare the performance with the configuration using the actual cost functions described in Section 3. Figure 7(a) presents the results of this experiment. We notice that all workloads are very sensitive to average MC load and row-buffer hit rates. With just row-buffer hit rates as the deciding factor for selecting MCs in the cost function, a performance drop of 29% is observed. For dynamic migration policy, we also characterize the sensitivity to the *cost of migration*(*Migration* bar in Figure 7(a)). Cost of migration is defined as the amount of data to be copied times the distance it has to be copied over, translating as the increased on-chip network traffic for data (page) migration. In Figure 7(a), for *Migration* experiments, decisions are made to migrate pages to the MC with the lowest cost of migration as the *only* deciding factor. Most workloads are less sensitive to distance and migration; notable exceptions being the ones with higher number of shared pages (ferret, streamcluster, fluidanimate). These workloads show a 13.1, 15.7 and 14.9% drop in performance as compared to the configuration with original cost function as the basis of decision making.

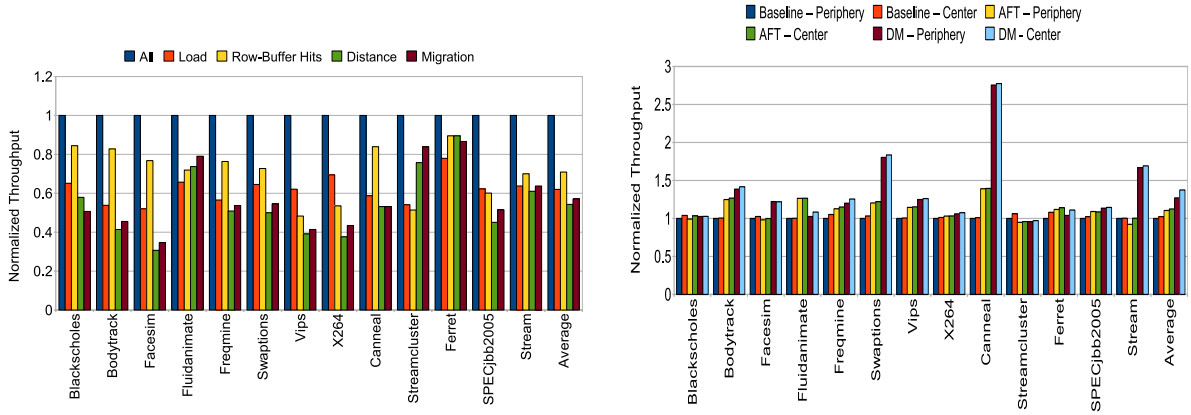
Figure 7(b) compares the effects of proposed policies for a different physical layout of MCs. As opposed to earlier, these configurations assume MCs being located at the *center* of the chip than periphery (similar to layouts assumed in [5]). We compare the baseline, adaptive first-touch (AFT) and dynamic 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 the most from the new layout. Due to the reduction in number of hops that have to traversed while copying data, DM-Center performs 10% better then DM-Periphery.

4.6 Effects of TLB Shootdowns

To study the performance impact of TLB shootdowns in

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)	121	5.8%
Bodytrack	489 (108/381)	327	3.2%
Facesim	310 (89/221)	221	4.1%
Fluidanimate	912 (601/311)	2687	12.6%
Freqmine	589 (100/489)	831	5.2%
Swaptions	726 (58/668)	107	2.4%
Vips	998 (127/871)	218	5.6%
X264	1007 (112/895)	286	8.1%
Canneal	223 (28/195)	89	2.1%
Streamcluster	1284 (967/317)	2895	18.4%
Ferret	1688 (1098/590)	3441	15.9%
SPECjbb2005	1028 (104/924)	487	4.1%
Stream	833 (102/731)	302	3.5%

Table 2: Dynamic page migration with lazy page migration, Overhead Characteristics



a. Throughput Sensitivity to Individual Terms in Cost function

b. Throughput Sensitivity to Physical Placement of MCs,

Figure 7: Sensitivity Analysis, Dynamic page migration policy

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 6.7%. 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 9.1%.

4.7 Energy Analysis

We use a modified version of CACTI [39] to calculate the energy costs of individual DRAM operations, which are listed in Table 3. Because of the increased row-buffer hit rates, the average DRAM energy consumption of AFT decreases by 14.1% as compared to baseline. For the data-migration scheme, the additional cost of copying pages adds to the total DRAM energy consumption. Even though row-buffer hit rates go up, the cost of copying pages makes the total energy consumption go up by 5.2%.

Operation	Energy Consumption in nJ
RCD (ACT+RAS)	20.7
CAS	1.1
PRECHARGE	19.2
Open-Page Hit	1.1
Empty-Page Access	21.8
Page-Conflict	41.1

Table 3: Energy Consumption of DRAM Operations for a CPU running at 3.0GHz

4.8 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 Figure 1(a). In these experiments, we assume a 4-socket system; each socket housing a quad-core chip, with similar configuration as assumed in Table 1. Each quad-core incorporates one on-chip MC which is responsible for a quarter of the total physical address space. Each quad-core has similar L1s as listed in Table 1, 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 [2] (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 4.8% 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 throughtput was 5.3%. We attribute this to the increased latency of cacheline invalidations and copying data over inter-socket links.

5. RELATED WORK

Memory Controllers: We discuss MC optimizations here that are related to our work. Some recent papers [5, 34, 51, 55] examine multiple MCs in a multi-core setting. Loh [34] 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. [51] 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 [55] employs multiple MCs on a single chip, accessible to every core via a specialized on-chip network. The Tile microprocessor [55] was one of the first processors to use multiple (four) on-chip MCs. More recently, Abts et al. [5] 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 [26], a memory scheduling algorithm that improves system throughput without requiring significant coordination between N on-chip memory controllers.

Recent papers [40, 41] 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 in tandem with other threads, relative to the case where the threads execute in isolation. Mutlu and Moscibroda [40] 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) [41]. The PAR-BS policy first breaks up the request queue into batches based on age and 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. [31] 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. [33] design prefetch mechanisms that take advantage of idle banks/channels and spatial locality within open rows. Zhu and Zhang [60] 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. [15, 16] study the vast design space of DRAM and memory controller features for a single core processor.

Memory Controllers and Page Allocation: Lebeck et al. [30] 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 [20], 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. [57] 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 [8, 18, 24, 38, 47]. Page coloring and migration have been employed to improve proximity of computation and data in a NUMA multi-processor [9, 14, 27–29, 52] and in NUCA caches [6, 13, 44]. 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 [15, 16, 45] 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. [36] 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 toolset 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 never in the context of multiple on-chip MCs. The related bodies of work that we borrow insight from are as follows: while the problem formulations are similar, the constraints of memory controller scheduling are different. Snavely et al. [48] schedule tasks from a pending task queue on to a number of available thread contexts in an SMT processor. Zhou et al. [59] schedule tasks on a 3D processor in an attempt to minimize thermal emergencies. Similarly, Powell et al. [42] attempt to minimize temperature by mapping a set of tasks to a CMP comprised of SMT cores.

6. CONCLUSIONS

The paper presents a substantial shift in memory controller design and data placement. We are headed for an era where a large number of programs will have to share limited off-chip bandwidth resources via a moderate number of memory controllers scattered on chip. 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 first 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 and this scheme works on first accesses to a given page. We then advance a scheme which dynamically migrates data within the DRAM sub-system to achieve lower memory access latencies. These dynamic schemes adapt with current system state and allow spreading a single program's working set across multiple memory controllers. As a result, these schemes yield improvements of 17% (when assigning pages on first touch), and 35% (when allowing pages to be copied across memory controllers).

We believe that several other innovations are possible, such as the consideration of additional memory scheduler constraints (intra-thread parallelism, handling of prefetch requests, etc.). Pages shared by multiple threads of a parallel application will also require optimal placement within the set of memory controllers. Page placement to promote bank parallelism in this context remains an open problem as well.

7. REFERENCES

[1] Perfmon2 Project Homepage. http://perfmon2.sourceforge.net/.

[2] Performance of the AMD Opteron LS21 for IBM BladeCenter. `ftp://ftp.software.ibm.com/eserver/benchmarks/wp_ls21_081506.pdf`.

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

[4] Micron DDR3 SDRAM Part MT41J512M4. `http://download.micron.com/pdf/datasheets/dram/ddr3/2Gb_DDR3_SDRAM.pdf`, 2006.

[5] D. Abts, N. Jerger, J. Kim, D. Gibson, and M. Lipasti. Achieving Predictable Performance through Better Memory Controller in Many-Core CMPs. In *Proceedings of ISCA*, 2009.

[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] C. Benia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical report, Department of Computer Science, Princeton University, 2008.

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

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

[10] J. Chang and G. Sohi. Co-Operative Caching for Chip Multiprocessors. In *Proceedings of ISCA*, 2006.

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

[12] Z. Chishti, M. Powell, and T. Vijaykumar. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proceedings of ISCA-32*, June 2005.

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

[14] 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.

[15] 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.

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

[17] W. Dally. Report from Workshop on On- and Off-Chip Interconnection Networks for Multicore Systems (OCIN), 2006. `http://www.ece.ucdavis.edu/~ocin06/`.

[18] 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.

[19] H. Dybdahl and P. Stenstrom. An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors. In *Proceedings of HPCA*, 2007.

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

[21] 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.

[22] E. Ipek, O. Mutlu, J. Martinez, and R. Caruana. Self Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Proceedings of ISCA*, 2008.

[23] ITRS. International Technology Roadmap for Semiconductors, 2007 Edition.

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

[25] C. Kim, D. Burger, and S. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches. In *Proceedings of ASPLOS*, 2002.

[26] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *Proceedings of HPCA*, 2010.

[27] R. LaRowe and C. Ellis. Experimental Comparison of

Memory Management Policies for NUMA Multiprocessors. Technical report, 1990.

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

[29] 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.

[30] A. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power Aware Page Allocation. In *Proceedings of ASPLOS*, 2000.

[31] C. Lee, O. Mutlu, V. Narasiman, and Y. Patt. Prefetch-Aware DRAM Controllers. In *Proceedings of MICRO*, 2008.

[32] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *Proceedings of HPCA*, 2008.

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

[34] G. Loh. 3D-Stacked Memory Architectures for Multi-Core Processors. In *Proceedings of ISCA*, 2008.

[35] 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.

[36] C. McCurdy and J. Vetter. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *Proceedings of ISPASS*, 2010.

[37] Micron Technology Inc. *Micron DDR2 SDRAM Part MT47H128M8HQ-25*, 2007.

[38] 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.

[39] 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.

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

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

[42] M. Powell, M. Gomaa, and T. Vijaykumar. Heat-and-Run: Leveraging SMT and CMP to Manage Power Density Through the Operating System. In *Proceedings of ASPLOS*, 2004.

[43] M. K. Qureshi. Adaptive Spill-Receive for Robust High-Performance Caching in CMPs. In *Proceedings of HPCA*, 2009.

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

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

[46] V. Romanchenko. Quad-Core Opteron: Architecture and Roadmaps. http://www.digital-daily.com/cpu/quad_core_opteron.

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

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

[49] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. Adaptive Mechanisms and Policies for Managing Cache Hierarchies in Chip Multiprocessors. In *Proceedings of ISCA*, 2005.

[50] R. Swinburne. Intel Core i7 - Nehalem Architecture Dive. http://www.bit-tech.net/hardware/2008/11/03/intel-core-i7-nehalem-architecture-dive/.

[51] D. Vantrease et al. Corona: System Implications of Emerging Nanophotonic Technology. In *Proceedings of ISCA*, 2008.

[52] 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.

[53] 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.

[54] D. Wang et al. DRAMsim: A Memory-System Simulator. In *SIGARCH Computer Architecture News*, September 2005.

[55] D. Wentzlaff et al. On-Chip Interconnection Architecture of the Tile Processor. In *IEEE Micro*, volume 22, 2007.

[56] M. Zhang and K. Asanovic. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of ISCA*, 2005.

[57] 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.

[58] H. Zheng et al. Mini-Rank: Adaptive DRAM Architecture For Improving Memory Power Efficiency. In *Proceedings of MICRO*, 2008.

[59] X. Zhou, Y. Xu, Y. Du, Y. Zhang, and J. Yang. Thermal Management for 3D Processor via Task Scheduling. In *Proceedings of ICPP*, 2008.

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