Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning

Sai Prashanth Muralidhara
Pennsylvania State University
smuralid@cse.psu.edu

Lavanya Subramanian
Carnegie Mellon University
lsubrama@ece.cmu.edu

Onur Mutlu
Carnegie Mellon University
onur@cmu.edu

Mahmut Kandemir
Pennsylvania State University
kandemir@cse.psu.edu

Thomas Moscibroda
Microsoft Research Asia
moscitho@microsoft.com

ABSTRACT

Main memory is a major shared resource among cores in a multicore system. If the interference between different applications' memory requests is not controlled effectively, system performance can degrade significantly. Previous work aimed to mitigate the problem of interference between applications by changing the scheduling policy in the memory controller, i.e., by prioritizing memory requests from applications in a way that benefits system performance.

In this paper, we first present an alternative approach to reducing inter-application interference in the memory system: *application-aware memory channel partitioning (MCP)*. The idea is to map the data of applications that are likely to severely interfere with each other to different memory channels. The key principles are to partition onto separate channels 1) the data of light (memory non-intensive) and heavy (memory-intensive) applications, 2) the data of applications with low and high row-buffer locality.

Second, we observe that interference can be further reduced with a combination of memory channel partitioning and scheduling, which we call *integrated memory partitioning and scheduling (IMPS)*. The key idea is to 1) always prioritize very light applications in the memory scheduler since such applications cause negligible interference to others, 2) use MCP to reduce interference among the remaining applications.

We evaluate MCP and IMPS on a variety of multiprogrammed workloads and system configurations and compare them to four previously proposed state-of-the-art memory scheduling policies. Averaged over 240 workloads on a 24-core system with 4 memory channels, MCP improves system throughput by 7.1% over an application-unaware memory scheduler and 1% over the previous best scheduler, while avoiding modifications to existing memory schedulers. IMPS improves system throughput by 11.1% over an application-unaware scheduler and 5% over the previous best scheduler, while incurring much lower hardware complexity than the latter.

MICRO'11 December 3-7 2011, Porto Alegre, Brazil

Categories and Subject Descriptors: C.1.0 [Processor Architectures]: General, C.1.2 [Multiple Data Stream Architectures (Multiprocessors)]

General Terms: Design, Performance

Keywords: Main Memory, Multicore, Interference, Data Allocation, Memory Controllers, Memory Scheduling

1. INTRODUCTION

Applications executing concurrently on a multicore chip contend with each other to access main memory, which has limited bandwidth. If the limited memory bandwidth is not managed well, different applications can harmfully interfere with each other, which can result in significant degradation in both system performance and individual application performance [11, 12, 17, 19, 20, 22]. Several techniques to improve system performance by reducing memory interference among applications have been proposed [11, 12, 17, 19, 20, 22, 24]. Fundamentally, these proposals viewed the problem as a memory access scheduling problem, and consequently focused on developing new memory request scheduling policies that prioritize the requests of different applications in a way that reduces inter-application interference. However, such application-aware scheduling algorithms require (non-negligible) changes to the existing memory controllers' scheduling logic [12, 30].

In this paper, we present and explore a fundamentally-different alternative approach to reducing inter-application interference in the memory system: controlling the mapping of applications' data to memory channels. Our approach is based on the observation that multicore systems have multiple main memory channels [2, 4], each of which controls a disjoint portion of physical memory and can be accessed independently without any interference. This reveals an interesting trade-off. On the one hand, interference between applications could (theoretically) be completely eliminated if each application's data were mapped to a different channel, assuming there were enough channels in the system. But, on the other hand, even if so many channels were available, mapping each application to its own channel would underutilize memory bandwidth and capacity (some applications may need less bandwidth/capacity than they are assigned, while others need more) and would reduce the opportunity for bank/channel-level parallelism within each application's memory access stream. Therefore, the main idea of our approach is to find a sweet spot in this trade-off by *mapping the* *data (i.e., memory pages) of applications that are likely to cause significant interference (i.e., slowdown) to each other, to different memory channels.*

We make two major contributions. First, we explore the potential for reducing inter-application memory interference purely with channel partitioning, without modifying the memory scheduler. To this end, we develop a new **application-aware Memory Channel Partitioning** (MCP) algorithm that assigns preferred memory channels to different applications. The goal is to assign to different channels the data of any two applications whose interference would cause significant slowdowns. Our algorithm operates using a set of heuristics that are guided by insight about how applications with different memory access characteristics interfere with each other. Specifically, we show in Sec 3 that applications with largely divergent memory-intensities or row-buffer-hit rates should preferably be separated to different channels because they cause significant slowdowns when mapped to the same channel.

Second, we show that MCP and application-aware memory request scheduling are orthogonal in the sense that both concepts can beneficially be applied together. Specifically, whereas our MCP algorithm is agnostic to the memory scheduler (i.e., we assume an unmodified, commonly used row-hit-first memory scheduler [25, 32]), we show that additional gains are possible when using MCP in combination with a minimal-complexity application-aware memory scheduling policy. We develop an **Integrated Memory Partitioning and Scheduling** (IMPS) algorithm that seamlessly divides the work of reducing inter-application interference between the memory scheduler and the system software's page mapper based on what each can do best.

The key insight underlying the design of IMPS is that interference suffered by very low memory-intensity applications is more easily mitigated by prioritizing them in the memory scheduler, than with channel partitioning. Since such applications seldom generate requests, prioritizing their requests does not cause significant interference to other applications, as previous work has also observed [11, 12, 20, 14]. Furthermore, explicitly allocating one or more channels for such applications can result in a waste of bandwidth. Therefore, IMPS prioritizes requests from such applications in the memory scheduler, without assigning them dedicated channels, while reducing interference between all other applications using channel partitioning.

Overview of Results: We evaluated MCP and IMPS on a wide variety of multi-programmed applications and systems and in comparison to a variety of pure memory scheduling algorithms. Our first finding is that, on a 24-core 4-memory controller system with an existing application-unaware memory scheduler, MCP provides slightly higher performance than the best previous memory scheduling algorithm, Thread Cluster Memory Scheduling (TCM) [12]: 7.1% performance improvement vs. 6.1% for TCM. This performance improvement is achieved with no modification to the underlying scheduling policy. Furthermore, we find that IMPS provides better system performance than current state-of-the-art memory scheduling policies, pure MCP, as well as combinations of MCP and state-of-the-art scheduling policies: 5% over TCM, while incurring smaller hardware complexity.

Our main conclusion is that *the task of reducing harmful inter-application memory interference should be divided between the memory scheduler and the system software page mapper*. Only the respective contributions of both entities yields the best system performance.

2. BACKGROUND

We present a brief background about the DRAM main memory system; more details can be found in [6, 19, 25]. A modern main memory system consists of several channels. Each channel has an address and a data bus and can be accessed independently, i.e., accesses to different channels can proceed in parallel. A channel consists of one or more ranks. A rank is a set of DRAM chips that collectively respond to a command. Each rank consists of multiple banks, each of which is a two-dimensional DRAM data array. Multiple banks within a channel can be accessed in parallel, but they share the address and data bus of the channel. As a result, data from only one bank can be sent through the channel in a given cycle.

A DRAM bank has a 2D structure consisting of rows and columns. A column is the smallest addressable unit of memory, and a large number of columns make up a row. When a unit of data has to be accessed from a bank, the entire row containing the data is brought into a small internal buffer called the *row-buffer*. If a subsequent memory access request is to the same row, it can be serviced faster (2-3 times) than if it were to a different row. This is called a row-hit.

Memory Request Scheduling Policy. FR-FCFS [25, 32] is a commonly used scheduling policy in current commodity systems. It prioritizes row-hits over row-misses, to maximize DRAM throughput and within each category, it prioritizes older requests. The analyses in this paper assume the FR-FCFS scheduling policy, but our insights are applicable to other scheduling policies as well. Sec 7 describes other memory scheduling policies and qualitatively compares our approach to them.

OS Page Mapping Policy. The Operating System (OS) controls the mapping of a virtual address to a physical address. The address interleaving policy implemented in the memory controller in turn maps the physical address to a specific channel/bank in main memory. Row interleaving and cache line interleaving are two commonly used interleaving policies. In the row interleaving policy, consecutive rows of memory are mapped to consecutive memory channels. We assume equal sizes for OS pages and DRAM rows in this work and use the terms page and row interchangeably, without loss of generality.[1] Pure cache line interleaving maps consecutive cache lines in physical address space to consecutive memory channels. MCP cannot be applied on top of this, as a page has to stay within a channel for MCP. However, we can potentially apply MCP on top of a restricted version of cache line interleaving that maps consecutive cache lines of a page to banks within a channel.

Commonly used OS page mapping and address interleaving policies are unaware of inter-application interference and map applications' pages across different channels. The OS does not consider application characteristics like cache miss rate and row-buffer locality that affect inter-application interference while mapping a virtual page to a physical page. We build our discussions, insights and mechanisms assuming such an interference-unaware OS page mapping policy

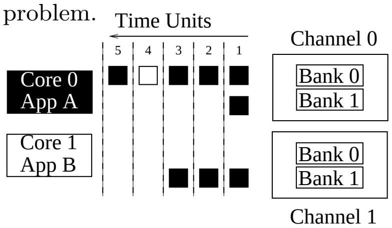
[1] Our mechanism works as long as the row size is greater than the OS page size, as is the case in typical systems.

and a row interleaved address mapping policy. However, we also evaluate MCP on top of cache line interleaving across banks in Sec 9.4.

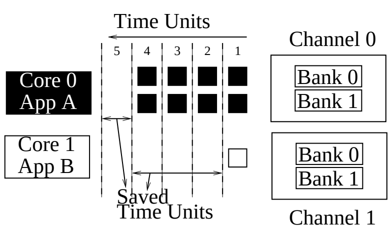
Memory Related Application Characteristics. We characterize memory access behavior of applications using two attributes. *Memory Access Intensity* is defined as the rate at which an application misses in the last level on-chip cache and accesses memory – calculated as *Misses per Kilo Instructions* (MPKI). *Row-Buffer Locality* is defined as the fraction of an application's accesses that hit in the row-buffer. This is calculated as the average *Row-Buffer Hit Rate* (RBH) across all banks.

3. MOTIVATION

In this section, we motivate our partitioning approach by showing how applications with certain characteristics cause more interference to other applications, and how careful mapping of application pages to memory channels can ameliorate this problem.



(a) Conventional Page Mapping



(b) Channel Partitioning

Figure 1: Conceptual example showing benefits of mapping data of low and high memory-intensity applications to separate channels

In Figure 1, we present a conceptual example showing the performance benefits of mapping the pages of applications with largely different memory-intensities to separate channels. Application A on Core 0 has high memory-intensity, generating memory requests at a high rate; Application B on Core 1 has low memory-intensity, generating requests at a much lower rate. Figures 1(a) and 1(b) show characteristic examples of what can happen with conventional page mapping (where the pages of applications A and B are mapped to the same channels) and with application-aware channel partitioning (where the pages of applications A and B are mapped to separate channels), respectively. In the first case, B's single request is queued behind three of A's requests in Bank 0 of Channel 0 (see Fig 1(a)). As a result, Application B stalls for a long period of time (four time units) until the three previously scheduled requests from A to the same bank get serviced. In contrast, if the two applications' data are mapped to separate channels as shown in Figure 1(b), B's request can be serviced immediately, enabling its faster progress (from four time units to one time unit). Application A's access latency improves (from five to four time units) because the interference caused to it by B's single request is eliminated.

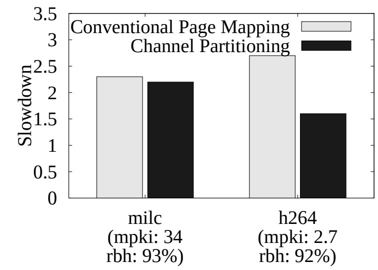


Figure 2: Application slowdowns due to interference between high and low memory-intensity applications

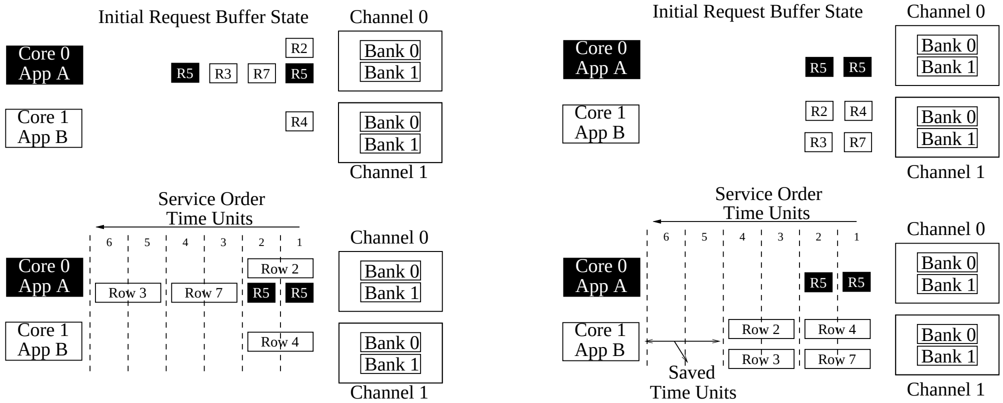
To determine to what extent such effects occur in practice, we run a large number of simulation experiments[2] with applications of vastly different memory-intensities and present a representative result: We run four copies each of *milc* and *h264* (from the SPEC CPU2006 suite [1]) on an eight-core, two-channel system. Figure 2 shows the effects of conventional channel sharing: *h264*, the application with lower memory-intensity, is slowed down by 2.7x when sharing memory channels with *milc* compared to when it is run alone. On the other hand, if *milc*'s and *h264*'s data are partitioned and mapped to Channels 0 and 1, respectively, *h264*'s slowdown is reduced to 1.5x. Furthermore, *milc*'s slowdown also drops from 2.3x to 2.1x, as its queueing delays reduce due to eliminating the interference from *h264*. This substantiates our intuition from the example: *separating the data of low memory-intensity applications from that of the high memory-intensity applications can largely improve the performance of the low memory-intensity application without significantly degrading the performance of the high memory-intensity application, thereby improving overall system performance.*

Memory-intensity is not the only characteristic that determines the relative harmfulness of an application (i.e., propensity to slowdown others). In Figure 3, we show potential benefits of mapping memory-intensive applications with significantly different row-buffer localities onto separate channels. In the example, Application A accesses the same row, Row 5, repeatedly and hence has much higher row-buffer locality than Application B, whose accesses are to different rows, incurring many row misses.

Figure 3(a) shows a conventional page mapping approach, while Figure 3(b) shows a channel partitioning approach. With conventional page mapping, the commonly used row-hit-first memory scheduling policy prioritizes A's requests over B's to Rows 7 and 3, even though B's requests had arrived earlier than one of A's (Figure 3(a)). This leads to increased queueing delays of B's requests, causing B to slow down. On the other hand, if the pages of A and B are mapped to separate channels (Figure 3(b)), the interference received by B is reduced and consequently the queueing delay experienced by B's requests is also reduced (by two time units). This improves Application B's performance without affecting Application A's.

A representative case study from among our experiments is shown in Figure 4. We ran four copies each of *mcf* and *libquantum*, two memory-intensive applications on an eight-core two-channel system. *Mcf* has a low row-buffer hit rate of 42% and suffers a slow down of 20.7x when sharing memory channels with *libquantum*, which is a streaming application with 99% row-buffer hit rate. The row-hit-first memory

[2]Our simulation methodology is described in Sec 8.



(a) Conventional Page Mapping (b) Channel Partitioning

Figure 3: Conceptual example showing benefits of mapping data of low and high row-buffer hit rate memory-intensive applications to separate channels. In both (a) and (b), the top part shows the initial state of the memory request buffer and the bottom part shows the order in which the requests are serviced.

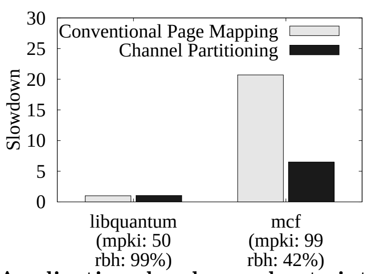


Figure 4: Application slowdowns due to interference between high and low row-buffer hit rate memory-intensive applications

scheduling policy starves *mcf*'s memory requests for a long time by prioritizing *libquantum*'s row-hit requests. On the other hand, if the data of *mcf* is isolated from *libquantum*'s data and given a separate channel, *mcf*'s slowdown drops significantly, to 6.5x from 20.7x. Even though there is still interference between four copies of *mcf* mapped to the same channel, this interference is much smaller than the starvation caused by *libquantum* to *mcf* if they were mapped to the same channel.

Libquantum's small performance loss of 4% shows the trade-off involved in channel partitioning: The drop is due to the loss in bank-level parallelism resulting from assigning only one channel to *libquantum*. In terms of system performance, however, this drop is far outweighed by the reduction in slowdown of *mcf*. We therefore conclude that *isolating applications with low row-buffer locality from applications with high row-buffer locality by means of channel partitioning improves the performance of applications with low row-buffer locality and the overall system.*

Based on these insights, we next develop MCP, an OS-level mechanism to partition the main memory bandwidth across the different applications running on a system. Then, we examine how to best combine memory partitioning and scheduling to minimize inter-application interference and obtain better system performance.

4. MEMORY CHANNEL PARTITIONING

Our Memory Channel Partitioning (MCP) mechanism consists of three components: 1) profiling of application behavior during run time, 2) assignment of preferred channels to applications, 3) allocation of pages to the preferred channel. The mechanism proceeds in periodic intervals. During each interval, application behavior is profiled (Sec 4.1). At the end of an interval, the applications are categorized into groups based on the characteristics collected during the interval, and each application is accordingly assigned a *preferred channel* (Sec 4.2). In the subsequent interval, these preferred channel assignments are applied. That is, when an application accesses a new page that is either not currently in DRAM or not in the application's preferred channel, MCP uses the *preferred channel assignment* for that application: the requested page is allocated in the preferred channel (see Sec 4.3).

In summary, during the Ith interval, MCP applies the preferred channel assignment which was computed at the end of the $(I-1)$th interval, and also collects statistics, which will then be used to compute the new preferred channel assignment to be applied during the $(I+1)$th execution interval.[3] Note that although MCP provides a preferred channel assignment to reduce interference, it does not constrain the memory capacity usage of applications. An application can use the entire DRAM capacity, if it needs so.

4.1 Profiling of Application Characteristics

As shown in Sec 3, memory access intensity and row-buffer locality are key factors determining the level of harm caused by interference between applications. Therefore, during every execution interval, each application's Misses Per Kilo Instruction (MPKI) and Row-Buffer Hit Rate (RBH) statistics are collected. To compute an application's inherent row-buffer hit rate, we use a per-core shadow row-buffer index for each bank, as in previous work [8, 12, 19], which keeps track of the row that would have been present in the row-buffer had the application been running alone.

4.2 Preferred Channel Assignment

At the end of every execution interval, each application is assigned a preferred channel. The assignment algorithm

[3]The very first interval of execution is used for profiling only and we call it the profile interval. We envision it to be shorter than the subsequent execution intervals, and its length is a trade off between minimizing the number of pages that get allocated before the first set of channel preferences are established and letting the application's memory access behavior stabilize before collecting statistics. Sec 9.7 evaluates sensitivity to profile interval length.

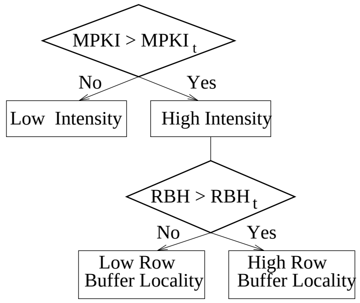


Figure 5: MCP: Application Grouping

is based on the insights derived in Sec 3. The goal is to separate as much as possible 1) the data of low memory-intensity applications from that of high memory-intensity applications, and, 2) among the high memory-intensity applications, the data of low row-buffer locality applications from that of high row-buffer locality applications. To do so, MCP executes the following steps in order:

1. Categorize applications into low and high memory-intensity groups based on their MPKI. (Sec 4.2.1)

2. Further categorize the high memory-intensity applications, based on their row-buffer hit rate (RBH) values into low and high row-buffer hit rate groups. (Sec 4.2.2)

3. Partition the available memory channels among the three application groups. (Sec 4.2.3)

4. For each application group, partition the set of channels allocated to this group between all the applications in that group, and assign a preferred channel to each application. (Sec 4.3)

4.2.1 Intensity Based Grouping

MCP categorizes applications into low and high memory-intensity groups based on a threshold parameter, $MPKI_t$, as shown in Fig 5. $MPKI_t$ is determined by averaging the last level cache Misses Per Kilo Instructions (MPKI) of all applications and multiplying it by a scaling factor. Applications are categorized as low memory-intensity if their MPKI is lower than $MPKI_t$ and high memory-intensity otherwise. The use of the average value of MPKI allows the threshold to adapt to workload memory intensity, while the scaling factor regulates the number of applications in the low memory-intensity group. We empirically found that a scaling factor of 1 provides an effective separation point and good system performance (Sec 9.7).

4.2.2 Row-Buffer Locality Based Grouping

MCP further classifies the high memory-intensity applications into either low or high row-buffer locality groups based on a threshold parameter, RBH_t, as shown in Fig 5. For every application i, if its RBH_i is less than RBH_t, then it is classified as a low row-buffer locality application. In this case, we do not take an average or use a scaling factor, as we observe that inter-application interference due to row-buffer locality differences are more pronounced between applications with very low and high row-buffer localities, unlike memory-intensity where there is interference across the continuum. We empirically observe that an RBH_t value of 50% provides effective interference reduction and good performance (Sec 9.7).

4.2.3 Partitioning of Channels between Application Groups

After thus categorizing applications into three groups, MCP partitions the available memory channels between the groups. It is important to note that at this stage of the algorithm, memory channels are assigned to application groups and *not* to individual applications. MCP handles the preferred channel assignment to individual applications in the next step (Sec 4.2.4). Channels are first partitioned between low and high memory-intensity groups. The main question is how many channels should be assigned to each group. One possibility is to allocate channels proportional to the total bandwidth demand (e.g., sum of applications' MPKIs) of each group (bandwidth-proportional allocation). This amounts to balancing the total bandwidth demand across the available channels. Alternatively, channels could be allocated proportional to the number of applications in that group (application-count-proportional allocation). In the former case, the low memory-intensity applications which constitute a very low proportion of total bandwidth demand might be assigned no channels. This fails to achieve their isolation from high memory-intensity applications, leading to low system performance. In contrast, the latter scheme may result in bandwidth wastage as the low memory-intensity applications seldom generate requests and the bandwidth of the channels they are assigned to would have been better utilized by the high memory-intensity applications. We found that the isolation benefits of application-count-proportional allocation outweighs the potential bandwidth wastage caused by allocating low-intensity applications to one or more channels.[4] Therefore, we use the application-count-proportional channel allocation strategy for MCP. However, bandwidth wastage caused by potentially allocating very low intensity applications dedicated channels remains. We will show that eliminating this wastage by handling these applications in the scheduler in an integrated scheduling and partitioning mechanism is beneficial (Sec 5).

The channels allocated to the high memory-intensity group are further partitioned between the low and high row-buffer locality application groups. The applications in the high memory-intensity group are bandwidth sensitive, meaning they each need a fair share of bandwidth to make progress. To ensure this, MCP assigns a number of channels to each of these two groups proportionately to the bandwidth demand (sum of MPKIs) of the group.

4.2.4 Preferred Channel Assignment within an Application Group

As a final step, MCP determines which applications within a group are mapped to which channels, when more than one channel is allocated to a group. Within a group, we balance the total bandwidth demand across the allocated channels. For each group, we maintain a ranking of applications by memory-intensity. We start with the least intensive application in the group and map applications to the group's first allocated channel until the bandwidth demand allocated to it (approximated by sum of $MPKI_i$ of each application i allocated to it) is $\frac{\text{Sum of MPKIs of applications in the group}}{\text{Number of channels allocated to the group}}$. We then move on to the next channel and allocate applications

[4]We found that bandwidth-proportional allocation results in a 4% performance loss over the baseline since it increases memory interference.

to it. We repeat this procedure for every group and eventually, each application is assigned a *preferred channel*.

4.3 Allocation of Pages to Preferred Channel

Once each application is assigned a preferred channel, MCP attempts to enforce this preference. For each page, there are two possibilities. First, the accessed page may not be resident in any memory channel (page fault). In this case, the page fault handler attempts to allocate the page in the preferred channel. If there is a free page in the preferred channel, the new page is allocated there. Otherwise, a modified version of the CLOCK replacement policy, as described in [7] is used. The baseline CLOCK policy keeps a circular list of pages in memory, with the hand (iterator) pointing to the oldest allocated page in the list. There is a Referenced (R) bit for each page, which is set to '1' when the page is referenced. The R bits of all pages are cleared periodically by the operating system. When a page fault occurs and there are no free pages, the hand moves over the circular list until an unreferenced page (a page with R bit set to '0') is found. The goal is to choose the first unreferenced page as the replacement. To allocate a page in the preferred channel, the modified CLOCK algorithm looks ahead N pages beyond the first replacement candidate to potentially find an unreferenced page in the preferred channel. If there is no unreferenced page within N pages, the first unreferenced page in the list across all channels is chosen as the replacement candidate. We use an N value of 512.

Second, the accessed page may be present in a channel other than the preferred channel, which we observe to be very rare in our workloads, since application behavior is relatively constant within an interval. In this case, dynamically migrating the page to the preferred channel could be beneficial. However, dynamic page migration incurs TLB and cache block invalidation overheads as discussed in [3]. We find that less than 12% of pages in all our workloads go to non-preferred channels and hence migration likely does not gain much performance over allowing some pages of an application to potentially remain in the non-preferred channels. Thus, our default implementation of MCP does not use page migration. However, if needed, migration can be incorporated into our mechanism.

5. INTEGRATED MEMORY PARTITIONING AND SCHEDULING

MCP aims to solve the inter-application memory interference problem entirely with the system software's page mapper (with the support of additional hardware counters to collect MPKI and RBH metrics for each application). It does not require any changes to the memory scheduling policy. This approach is in contrast to the various existing proposals, which try to solve the problem purely in hardware. These proposals aim to reduce memory interference entirely in the memory controller hardware using sophisticated scheduling policies (e.g., [11, 12, 19, 20]). The question is whether either extreme alone (i.e., page mapping alone and memory scheduling alone) can really provide the best possible interference reduction. Based on our observations below, the answer is no. Specifically, we devise an integrated memory partitioning and scheduling (IMPS) mechanism that aims to combine the interference reduction benefits of both the system software page mapper and the memory request scheduling hardware.

The key observation underlying IMPS is that applications with very low memory-intensity, when prioritized over other applications in the memory scheduler, do not cause significant slowdowns to other applications. This observation was also made in previous work [11, 12]. These applications seldom generate memory requests; prioritizing these requests enables the applications to quickly continue with long computation periods and utilize their cores better, thereby significantly improving system throughput [11, 12]. As such, scheduling can very efficiently reduce interference experienced by very low memory-intensity applications. In contrast, reducing the interference against such applications purely using the page mapper is inefficient. The mapper would have to dedicate one or more channels to such low-memory-intensity applications, wasting memory bandwidth, since these applications do not require significant memory bandwidth (yet high memory-intensity applications would likely need the wasted bandwidth, but cannot use it). If the mapper did not dedicate a channel to such applications, they would share channels with high memory-intensity applications and experience high interference with an unmodified memory scheduler.

The basic idea and operation of IMPS is therefore simple. First, identify very low memory-intensity applications (i.e., applications whose MPKI is smaller than a very low threshold, 1.5 in most of our experiments (Sec 9.7)) at the end of an execution interval, prioritize them in the memory scheduler over all other applications in the next interval, and allow the mapping of the pages of such applications to any memory channel. Second, reduce interference between all other applications by using memory channel partitioning (MCP), exactly as described in Sec 4. The modification to the memory scheduler is minimal: the scheduler only distinguishes the requests of very low memory-intensity applications over those of others, but does not distinguish between requests of individual applications in either group. The memory scheduling policy consists of three prioritization rules: 1) prioritize requests of very low memory-intensity applications, 2) prioritize row-hit-first requests, 3) prioritize older requests.

Note that MCP is still used to classify the remaining applications as low and high memory-intensity, as only the very low memory-intensity applications are filtered out and prioritized in the scheduler. MCP's channel partitioning still reduces interference and consequent slowdowns of the remaining applications.

6. IMPLEMENTATION

Hardware support. MCP requires hardware support to estimate MPKI and row-buffer hit rate of each application, as described in Sec 4.1. These counters are readable by the system software via special instructions. Table 1 shows the storage cost incurred for this purpose. For a 24-core system with 4 memory controllers (each controlling 4 memory banks and 16384 rows per bank), the hardware overhead is 12K bits. IMPS requires an additional bit per each request (called *low-intensity bit*) to distinguish very low-memory-intensity applications' requests over others, which is an additional overhead of only 512 bits for a request queue size of 128 per MC. IMPS also requires small modifications to the memory scheduler to take into account the *low-intensity bits* in prioritization decisions. Note that, unlike previous application-aware memory request schedul-

Storage	Description	Size (in bits)
Storage Overhead for MCP - per-core registers		
MPKI-counter	A core's last level cache misses per kilo instruction	$N_{core} \times log_2 MPKI_{max} = 240$
Storage Overhead for MCP - per-core registers in each controller		
Shadow row-buffers	Row address of a core's last accessed row	$N_{core} \times N_{banks} \times log_2 N_{rows} = 1344$
Shadow row-buffer hit counters	Number of row-hits if the application were running alone	$N_{core} \times N_{banks} \times log_2 Count_{max} = 1536$
Additional Storage Overhead for IMPS - per request register in each controller		
Very low memory-intensity indicator	Identify requests from very low-intensity applications	$1 \times Queue_{max} = 128$

Table 1: Hardware storage required for MCP and IMPS

ing policies, IMPS 1) does not require each main memory request to be tagged with a thread/application ID since it does not distinguish between individual applications' requests, 2) adds only a single new bit per request for the memory scheduler to consider, 3) does not require application ranking as in [11, 12, 20] – ranking and prioritization require hardware logic for sorting and comparisons [18]. As such, the hardware complexity of IMPS is much lower than previous application-aware memory scheduling policies.

System software support. MCP and IMPS require support from system software to 1) read the counters provided by the hardware, 2) perform the preferred channel assignment, at the end of each execution interval, as already described. Each application's preferred channel is stored as part of the system software's data structures, leading to a very modest memory overhead of $N_{AppsInSystem} \times N_{MemoryChannels}$ bits. The page fault handler and the page replacement policy are modified slightly, as described in Sec 4.3. The proposed mechanisms do not require changes to the page table. The time overhead of the modified replacement policy is small. First, only on a page fault are up to N (512) entries in the circular page list scanned to find a suitable replacement candidate. Next, compared to the baseline cost of a page fault (on the order of millions of cycles), the extra overhead of scanning the list is extremely small. A quick back-of-the-envelope calculation shows that, assuming a page table entry size of 4 bytes, bringing in 512 entries (i.e., 2KB) from memory to cache (to check their Referenced bits) takes less than 10,000 cycles.

7. RELATED WORK AND QUALITATIVE COMPARISONS

To our knowledge, this paper is the first to propose and explore memory page mapping mechanisms as a solution to mitigate inter-application memory interference and thereby improve system performance.

Memory Scheduling. The problem of mitigating interference has been extensively addressed using application-aware memory request scheduling. We briefly describe the two approaches we compare our mechanisms to in Section 9. ATLAS [11] is a memory scheduling algorithm that improves system throughput by prioritizing applications based on their attained memory service. Applications that have smaller attained memory service are prioritized over others because such threads are more likely to return to long compute periods and keep their cores utilized. Thread cluster memory scheduling (TCM) [12] improves both system performance and fairness. System performance is improved by allocating a share of the main memory bandwidth for latency-sensitive applications. Fairness is achieved by shuffling scheduling priorities of memory-intensive applications at regular intervals to prevent starvation of any application. These and other application-aware memory schedulers [17, 19, 20, 22, 24, 18] attempt to reduce inter-application memory interference purely through memory request scheduling. As a result, they require significant modifications to the memory controller's design. In contrast, we propose 1) an alternative approach to reduce memory interference which does not require changes to the scheduling algorithm when employed alone, 2) combining our channel partitioning mechanism with memory scheduling to gain better performance than either can achieve alone. Our quantitative comparisons in Section 9 show that our proposed mechanisms perform better than the state-of-the-art scheduling policies, with no or minimal changes to the memory scheduling algorithm.

Application-unaware memory schedulers [10, 21, 25, 32], including the commonly-employed FR-FCFS policy [25, 32], aim to maximize DRAM throughput, These policies do not attempt to reduce inter-application interference and therefore, lead to low system performance in multi-core systems, as shown in previous work [11, 12, 17, 19, 20, 22].

OS Thread Scheduling. Zhuravlev et al. [31] aim to mitigate shared resource contention between threads by co-scheduling threads that interact well with each other on cores sharing the resource, similar to [26]. Such solutions require enough threads with symbiotic characteristics to exist in the OS's thread scheduling pool. In contrast, our proposal can reduce memory interference even if threads that interfere significantly with each other are co-scheduled in different cores and can be combined with co-scheduling proposals to further improve system performance.

Page Allocation. Page allocation mechanisms have been explored previously. Awasthi et al. [3] use page allocation and migration to balance load across memory controllers (MCs) in an application-unaware manner, to improve memory bandwidth utilization and system performance in a network-on-chip based system where a core has different distances to different memory channels. Our proposal, in comparison, performs page allocation in an application-aware manner with the aim of reducing interference between different applications. We compare our approach to an adaptation of [3] to crossbar-based multicore systems where all memory controllers are equidistant to any core (in Section 9.3) and show that application-aware channel partitioning leads to better system performance than balancing load in MCs. However, concepts from both approaches can be combined for further performance benefits.

In NUMA-based multiprocessor systems with local and remote memories, page allocation mechanisms were used to place data close to corresponding computation node [5, 29]. The goal was to reduce the average latency of accessing data from memory, while still leveraging the memory capacity provided by remote nodes, for a single application. Our goal is completely different: to map data to different channels to mitigate interference between different applications.

Sudan et al. [27] propose to colocate frequently used chunks of data in the same rows, thereby improving row-buffer locality, by modifying OS page mapping mechanisms. Lebeck et al. [13] and Hur et al. [10] propose page allocation mech-

Processor Pipeline	128-entry instruction window (64-entry issue queue, 64-entry store queue), 12-stage pipeline
Fetch/Exec/Commit Width	3 instructions per cycle in each core; 1 can be a memory operation
L1 Caches	32 K-byte per-core, 4-way set associative, 32-byte block size
L2 Caches	512 K-byte per core, 8-way set associative, 32-byte block size
DRAM controller (on-chip)	128-entry request buffer, 64-entry write buffer, reads prioritized over writes, row interleaving
DRAM chip parameters	DDR2-800 timing parameters, t_{CL}=15ns, t_{RCD}=15ns, t_{RP}=15ns, BL/2=10ns, 4 banks, 4KB row-buffer
DIMM Configuration	Single-rank, 8 DRAM chips put together on a DIMM to provide a 64-bit wide memory channel
DRAM Capacity	4KB/row * 16K rows/bank * 4 banks/channel * 4 channels = 1GB
Round-trip L2 miss latency	For a 32-byte cache line uncontended: row-buffer hit: 40ns (200 cycles), closed: 60ns (300 cycles), conflict: 80ns (400 cycles)
Cores and DRAM controllers	24 cores, 4 independent DRAM controllers, each controlling a single memory channel

Table 2: Default processor and memory subsystem configuration

No.	Benchmark	MPKI	RBH	No.	Benchmark	MPKI	RBH	No.	Benchmark	MPKI	RBH
1	453.povray	0.03	85.2%	10	445.gobmk	0.6	71%	19	482.sphinx3	24.9	85.4%
2	400.perlbench	0.13	83.6%	11	435.gromacs	0.7	84.4%	20	459.GemsFDTD	25.3	28.8%
3	465.tonto	0.16	91%	12	464.h264	2.7	92.3%	21	433.milc	34.3	93.2%
4	454.calculix	0.20	87.2%	13	401.bzip2	3.9	53.8%	22	470.lbm	43.5	95.2%
5	444.namd	0.3	95.4%	14	456.hmmer	5.7	35.5%	23	462.libquantum	50	99.2%
6	481.wrf	0.3	91.9%	15	473.astar	9.2	76.2%	24	450.soplex	50.1	91.3%
7	403.gcc	0.4	73.2%	16	436.cactusADM	9.4	18%	25	437.leslie3d	59	82.6%
8	458.sjeng	0.4	11.5%	17	471.omnetpp	21.6	46%	26	429.mcf	99.8	42.9%
9	447.dealIII	0.5	81.2%	18	483.xalancbmk	23.9	73.2%				

Table 3: SPEC CPU2006 benchmark characteristics

anisms to increase idleness and thus decrease energy consumption in DRAM ranks/banks. Phadke et al. [23] propose a heterogeneous memory system where each memory channel is optimized for latency, bandwidth, or power and propose page mapping mechanisms to map appropriate applications' data to appropriate channels to improve performance and energy efficiency. None of these works consider using page allocation to reduce inter-application memory interference, and therefore they can be potentially combined with our proposal to achieve multiple different goals.

Source Throttling. Ebrahimi et al. [8, 9] propose mechanisms to reduce inter-application interference by throttling the request injection rate of those applications that slow down others the most, at the source (core). Though our goal is also to reduce inter-application interference, our approach is complementary: channel partitioning. Source throttling and channel partitioning can be employed synergistically to further mitigate inter-application interference and we aim to investigate this interaction as part of future work.

8. EVALUATION METHODOLOGY

Simulation Setup. MCP requires the MPKI and RBH values to be collected for each application. These per-application hardware counters, though easy to implement, are not present in existing systems. Also, our evaluation requires different system configurations with varying architectural parameters and comparison to new scheduling algorithms. For these reasons, we are unable to evaluate MCP on a real system and use an in-house cycle-level x86 multi-core simulator. The front end of the simulator is based on Pin [15]. Pin supplies virtual addresses. Our simulator models a virtual-to-physical address translator that allocates a physical frame to a virtual page when the virtual page is first touched. This simulator also models the memory subsystem of a CMP in detail. It enforces channel, rank, bank, port and bus conflicts, thereby capturing all the bandwidth limitations and modeling both channel and bank-level parallelism accurately. The memory model is based on DDR2 timing parameters [16]. We model the execution in a core, including the instruction-window. Unless mentioned otherwise, we model a 24-core system with 4 memory channels/controllers. Table 2 shows major processor and memory parameters.

Evaluation Metrics. We measure the overall throughput of the system using *weighted speedup* [26]. We also report *harmonic speedup*, which is a combined measure of performance and fairness.

$$SystemThroughput = WeightedSpeedup = \Sigma_i \frac{IPC_i^{shared}}{IPC_i^{alone}};$$
$$HarmonicSpeedup = \Sigma_i \frac{N}{\frac{IPC_i^{alone}}{IPC_i^{shared}}}.$$

Our normalized results are normalized to the baseline with FR-FCFS memory scheduler, unless stated otherwise.

Workloads. We use workloads constructed from the SPEC CPU2006 benchmarks [1] in our evaluations. We compiled the benchmarks using gcc with the O3 optimization flag. Table 3 shows benchmarks' characteristics. We classify benchmarks into two categories: high memory-intensity (greater than 10 MPKI) and low memory-intensity (less than 10 MPKI). We vary the fraction of high memory-intensity benchmarks in our workloads from 0%, 25%, 50%, 75%, 100% and construct 40 workloads in each category. Within each memory-intensity category, we vary the fraction of high row-buffer hit rate benchmarks in a workload from low to high. We also create another category, $VeryLow(VL)$ consisting of 40 workloads. All benchmarks in these workloads have less than 1 MPKI. We consider VL for completeness, although these workloads have little bandwidth demand. For our main evaluations and some analyses, we use all 240 workloads and run each workload for 300M cycles. For sensitivity studies, we use the 40 balanced (50% memory-intensive) workloads, unless otherwise mentioned, and run for 100M cycles to reduce simulation time.

Parameter Values. The default MPKI scaling factor and RBH_t values we use in our experiments are 1 and 50% respectively. For the *profile interval* and *execution interval*, we use values of 10 million and 100 million cycles, respectively. We later study sensitivity to these parameters.

9. RESULTS

We first present and analyze the performance of MCP and IMPS on a 24-core 4-memory-controller system. Figure 6 shows the system throughput and harmonic speedup

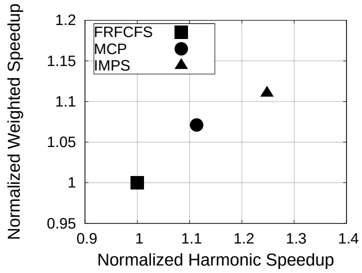


Figure 6: MCP and IMPS performance (normalized) across 240 workloads

averaged over all 240 workloads. The upper right part of the graph corresponds to better system throughput and a better balance between fairness and performance. MCP improves system throughput by 7.1% and harmonic speedup by 11% over the baseline. IMPS provides 4% better system throughput (13% better harmonic speedup) over MCP, and 11% better system throughput (24% better harmonic speedup) over the baseline. We observe (not shown) that the scheduling component of IMPS alone (without partitioning) gains half of the performance improvement of IMPS. We conclude that interference-aware channel partitioning is beneficial for system performance, but dividing the task of interference reduction between channel partitioning and memory request scheduling provides better system performance than employing either alone.

Effect of Workload Memory-Intensity. Figure 7 shows the system throughput benefits of MCP and IMPS, for six workload categories with different memory intensities.[5] As expected, as workload intensity increases (from left to right in the figure), absolute system throughput decreases due to increased interference between applications.

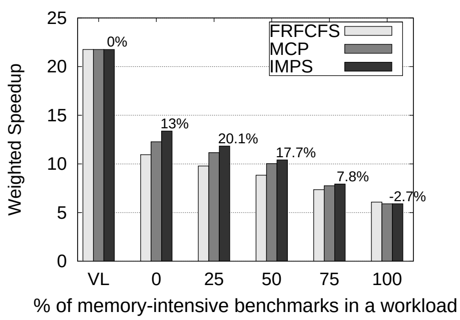


Figure 7: MCP and IMPS performance across memory-intensity categories. % gain values of IMPS over FR-FCFS are labeled.

We make three major conclusions. First, MCP and IMPS improve performance significantly over FR-FCFS in most of the memory-intensity categories. Specifically, MCP avoids interference between applications of both dissimilar and similar intensities by isolating them to different channels, enabling benefits mostly regardless of workload composition. Second, IMPS's performance benefit over MCP is especially significant in the lower-intensity workloads. Such workloads have a higher number of very low memory-intensity applications and IMPS prioritizes them in the scheduler, which is more effective for system performance than reducing interference for them by assigning them to their own channels, which wastes bandwidth as done by MCP. As the workload memory-intensity increases, IMPS' performance benefit over MCP becomes smaller because the number of low memory-intensity applications becomes smaller. Third, when the workload mix is very non-intensive or very intensive, MCP/IMPS do not provide much benefit. In the VL category, load on memory and as a result interference is very low, limiting the potential of MCP/IMPS. When 100% of applications in the workload are intensive, the system becomes severely memory bandwidth limited and conserving memory bandwidth by exploiting row-buffer locality (using simple FR-FCFS) provides better performance than reducing inter-application interference at the expense of reducing memory throughput. Any scheduling or partitioning scheme that breaks the consecutive row-buffer hits results in a system performance loss. In fact, it can be observed that MCP/IMPS degrade performance only by 2.7% as compared to previously proposed scheduling policies that degrade performance by 40-50% (shown in Figure 9). Therefore, we conclude that MCP and IMPS are effective for a wide variety of workloads where contention exists and the system is not severely limited by memory bandwidth.

[5]All categories except VL place a significant load on the memory system, as the intensity cut off used to classify an application as intensive is 10 MPKI, which is relatively large.

9.1 Comparison to Previous Scheduling Policies

Figure 8 compares MCP and IMPS with previous memory scheduling policies, FR-FCFS [25], PAR-BS [20], ATLAS [11] and TCM [12] over 240 workloads. Two major conclusions are in order. First, most application-aware scheduling policies perform better than FR-FCFS, and TCM performs the best among the application-aware scheduling policies, consistent with previous work [20, 11, 12]. Second, MCP and IMPS outperform TCM by 1%/5%, with no/minimal changes to the scheduler.

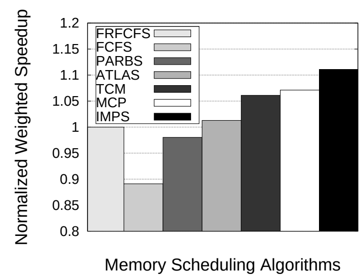


Figure 8: MCP and IMPS performance (normalized) vs. previous scheduling policies avg. across 240 workloads

Figure 9 provides insight into where MCP's and IMPS' performance benefits are coming from by breaking down performance based on workload intensity. As the workload memory intensity (thus contention) increases, MCP and IMPS become more effective than pure memory scheduling approaches. For low-intensity workloads (VL, 0%, 25%), TCM performs slightly better than IMPS because TCM is able to distinguish and prioritize between each individual application in the memory scheduler (not true for MCP/IMPS), leading to reduced interference between low and medium intensity applications. For high memory-intensity workloads (50%, 75%, 100%), reducing interference via channel partitioning is more effective than memory scheduling: both MCP and IMPS outperform TCM, e.g. by 40% in the 100%-intensity workloads. In such workloads, contention for memory is very high as many high-intensity applications contend. Channel partitioning completely eliminates interference between some applications by separating out their access streams to different channels, thereby reducing the number of applications that contend with each other. On the other hand, TCM or a pure memory scheduling scheme tries to handle contention between high-intensity workloads purely by prioritization, which is more effective at balancing interference but cannot eliminate interference as MCP/IMPS does since all applications contend with each other. We conclude that IMPS is a more effective solution than pure memory scheduling especially when workload intensity (i.e., memory load) is high, which is the expected trend in future systems.

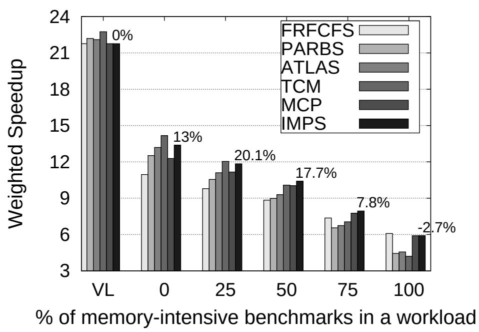


Figure 9: MCP and IMPS performance vs previous scheduling policies across memory-intensity categories. % values are gains of IMPS over FR-FCFS.

Note that IMPS's performance benefits over application-aware memory schedulers come at a significantly reduced hardware complexity, as described in Section 6.

9.2 Interaction with Previous Scheduling Policies

Figure 10 compares MCP and IMPS, when implemented on top of FR-FCFS, ATLAS and TCM as the underlying scheduling policy. When IMPS is implemented over ATLAS and TCM, it adds another priority level on top of the scheduling policy's priority levels: very-low-intensity applications are prioritized over others and the scheduling policy's priorities are used between very-low-intensity applications and between the remaining applications.

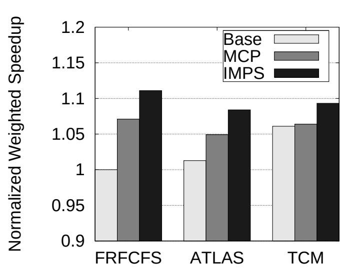


Figure 10: MCP and IMPS performance over different scheduling policies (240 workloads)

Several conclusions are in order. First, adding MCP/IMPS on top of any previous scheduling policy improves performance (IMPS gains 7% and 3% over ATLAS and TCM respectively) showing that our proposal is complementary to the underlying memory scheduling policy. Second, MCP/IMPS over FR-FCFS (our default proposal) provides better performance than MCP/IMPS employed over TCM or ATLAS. This is due to two reasons: 1) channel partitioning decisions MCP makes are designed assuming an FR-FCFS policy and not designed to take into account or interact well with ATLAS/TCM's more sophisticated thread ranking decisions. There is room for improvement if we design a channel partitioning scheme that is specialized for the underlying scheduling policy. We leave this for future work. 2) MCP/IMPS isolates groups of similar applications to different channels and ATLAS/TCM operate within each channel to prioritize between/cluster these similar applications. However, ATLAS and TCM are designed to exploit heterogeneity between applications and do not perform as well when the applications they prioritize between are similar. We found that prioritizing similar-intensity applications over each other as done in ATLAS/TCM creates significant slowdowns because the applications are treated very differently. We conclude that MCP/IMPS can be employed on top of any underlying scheduler to gain better performance over using the scheduler alone. However, it performs best when employed over an FR-FCFS baseline for which it is designed.

9.3 Comparison with Balancing Memory Load Across Multiple Memory Controllers

In [3], Awasthi et al. propose two page allocation schemes to balance the load across multiple memory controllers: 1) page allocation on first touch (Adaptive First Touch, AFT), 2) Dynamic Page Migration (DPM). AFT attempts to balance load by allocating a page to a channel which has the minimum value of a cost function involving channel *load*, row-buffer hit rate, and the *distance* to the channel. DPM proposes to migrate a certain number of pages from the most loaded to the least loaded channel at regular intervals, in addition to AFT. In our adaptation of AFT, we consider both channel load and row-buffer hit rate but do not incorporate the channel distance, as we do not model a network-on-chip. Figure 11 compares MCP/IMPS performance to that of AFT and DPM on 40 workloads with 50% of applications memory intensive. First, AFT and DPM both improve performance by 5% over the baseline, because they reduce memory access latency by balancing load across different channels. The gains from the two schemes are similar as the access patterns of the applications we evaluate do not vary largely with time, resulting in very few invocations of dynamic page migration. Second, our proposals outperform AFT and DPM by 7% (MCP) and 12.4% (IMPS), as they proactively reduce inter-application interference by using application characteristics, while AFT and DPM are not interference- or application-aware and try to reactively balance load across memory controllers. We conclude that reducing inter-application interference by page allocation provides better performance than balancing load across memory controllers in an application-unaware manner.

9.4 Effect of Cache Line Interleaving

We study the effect of MCP/IMPS on a system with a restricted form of cache line interleaving that maps consecutive cache lines of a page across banks within a channel. Figure 12 shows that MCP/IMPS improve the performance of such a system by 5.1% and 11% respectively. We observed (not shown) that unrestricted cache line interleaving across channels (to which MCP/IMPS cannot be applied) improves

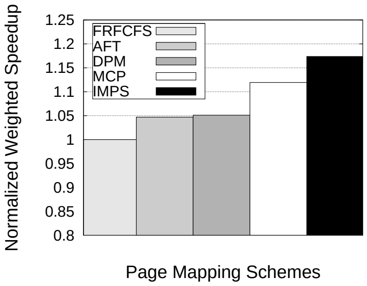


Figure 11: MCP and IMPS performance vs. load balancing across memory controllers [3] (40 workloads)

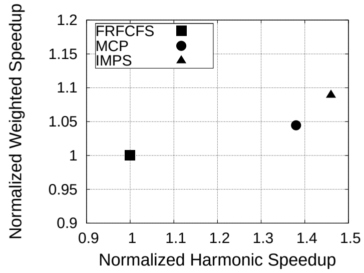


Figure 12: System throughput and harmonic speedup with cache line interleaving (240 workloads)

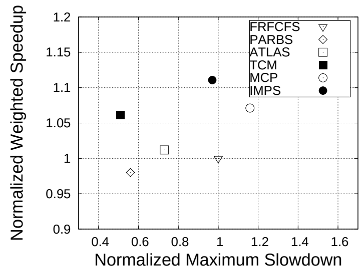


Figure 13: Performance and fairness compared to previous scheduling policies (240 workloads)

performance by only 2% over restricted cache line interleaving. Hence, using channel partitioning with MCP/IMPS outperforms cache line interleaving across channels. This is because the reduction in inter-application interference with MCP/IMPS provides more system performance benefit than the increase of channel-level parallelism with unrestricted cache-line interleaving. We conclude that MCP/IMPS are effective independent of the interleaving policy employed, as long as the interleaving policy allows the mapping of an entire page to a channel (which is required for MCP/IMPS to be implementable).

9.5 Effect of MCP and IMPS on Fairness

The fairness metric we use, the maximum slowdown of a workload, is defined as the maximum of the slowdowns (inverse of speedups) of all applications [11, 12, 28]; lower maximum slowdown values are more desirable. Figure 13 shows performance vs. fairness of previously proposed scheduling policies and our proposed schemes. IMPS has slightly better fairness (3% lower maximum slowdown) than FR-FCFS. While MCP and IMPS provide the best performance compared to any other previous proposal, they result in higher unfairness. Note that this is expected by design: MCP and IMPS are designed for improving system performance and not fairness. They make the conscious choice of placing high memory-intensity (and high row-buffer locality) applications onto the same channel(s) to enable faster progress of lower-intensity applications, which sometimes results in the increased slowdown of higher-intensity applications. Channel partitioning based techniques that can improve both performance and fairness are an interesting area for future work.

9.6 Effect of MCP and IMPS on Page Faults

We observe that MCP/IMPS do not increase and in fact can reduce page faults due to two reasons. First, MCP does not hard partition memory capacity between applications. Hard partitioning would cause fragmentation across channels, which could lead to page faults in a channel whose capacity is exhausted. However, MCP introduces only soft partitions, so that if the preferred channel is exhausted, an empty page from a non-preferred channel is allocated. Second, we use a modified page replacement policy proposed by Das et al. [7] (see Section 4.3). As also analyzed and observed in [7], this replacement policy reduces interference between pages allocated to different channels by preferentially picking replacement candidates within the same channel, actually leading to a reduction in page faults.

9.7 Sensitivity Studies

Sensitivity to MCP/IMPS algorithm parameters. We first vary the profile interval length to study its impact on MCP and IMPS' performance (Figure 14). A short initial profile interval length leads to less stable MPKI and RBH values, and hence potentially inaccurate estimation of application characteristics. In contrast, a long profile interval length causes a larger number of pages to be allocated prior to computing channel preferences. A profile interval length of 10M cycles balances these downsides of short and long intervals and provides the best performance.

We also experimented with different execution interval lengths (Figure 15). A shorter interval leads to better adaptation to changes in application behavior but could potentially result in more pages being on non-preferred channels. In contrast, a longer interval might miss changes in the behavior of applications. Though execution interval lengths of 100M and 200M cycles both provide similar (and the best) system performance, because an interval length of 100M cycles captures application phase changes better, we use that.

Figure 16 shows the sensitivity of MCP/IMPS to the $MPKI_t$ scaling factor. As the $MPKI_t$ scaling factor is increased beyond 1, more medium and high memory-intensity applications get into the low memory-intensity group, thereby slowing down the low-intensity applications and resulting in lower throughput. We also varied RBH_t, the row-buffer hit rate threshold and the very low memory-intensity threshold used in IMPS. System performance remains high and stable over a wide range of these values, with the best performance observed at an RBH_t value of 50% and a very low memory-intensity threshold value of 1.5.

Sensitivity to core count, cache sizes, MCs and banks. Table 4 shows the performance of IMPS as number of cores, L2 cache size, number of MCs and number of banks are varied. The rest of the system remains the same and we do not tune system/mechanism parameters. Even without any parameter tuning, IMPS' benefits are significant across all configurations. IMPS' performance gain in general increases when the system is more bandwidth constrained, i.e., with higher number of cores and smaller number of MCs. MCP shows similar trends as IMPS (not shown).

10. CONCLUSION

We presented the concepts and mechanisms of 1) application-aware memory channel partitioning (MCP), a fundamentally new approach to reducing inter-application interference at the memory system, by mapping the data of

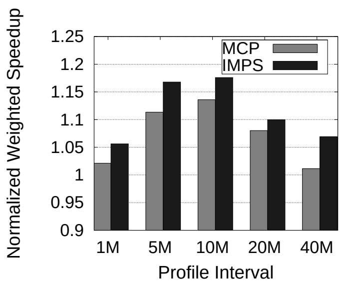


Figure 14: Performance vs profile interval (40 workloads)

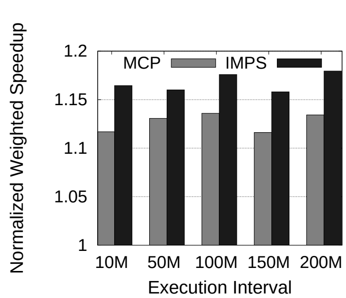


Figure 15: Performance vs execution interval (40 workloads)

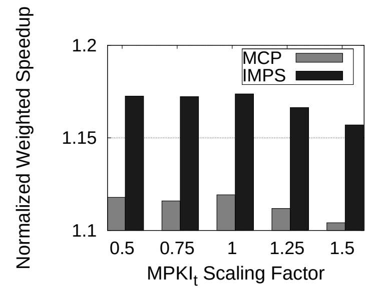


Figure 16: Performance vs $MPKI_t$ scaling factor (40 workloads)

No. of Cores			Private L2 Cache Size			No. of MCs				No. of Banks		
16	24	32	256KB	512KB	1MB	2	4	8	16	4	8	16
15.8%	17.4%	31%	16.6%	17.4%	14.3%	18.2%	17.1%	10.7%	6.5%	17.1%	12.6%	4.6%

Table 4: IMPS system throughput improvement with different system parameters (averaged over 40 workloads)

interfering applications to separate channels, 2) integrated memory channel partitioning and scheduling (IMPS), that effectively divides the work of reducing inter-application interference between the system software and the memory scheduler. Our extensive qualitative and quantitative comparisons demonstrate that MCP and IMPS both provide better system performance than the state-of-the-art memory scheduling policies, with no or minimal hardware complexity. IMPS provides better performance than channel partitioning or memory scheduling alone. We conclude that inter-application memory interference is best reduced using the right combination of page allocation to channels and memory request scheduling, and that IMPS achieves this synergy with minimal hardware complexity.

ACKNOWLEDGMENTS

We gratefully acknowledge Yoongu Kim, Vivek Seshadri, members of the SAFARI research group at CMU and the Microsystems Design Lab at PSU for insightful discussions on this work. We acknowledge the support of our industrial sponsors, AMD, Intel, Oracle, and Samsung. This research was partially supported by grants from the Gigascale Systems Research Center, the National Science Foundation (CAREER Award CCF-0953246, 1147397, 1147388, 1152479, 1017882, 0963839), Intel Corporation ARO Memory Hierarchy Program, Microsoft Corporation and Carnegie Mellon CyLab.

REFERENCES

[1] *SPEC CPU2006*. http://www.spec.org/spec2006.

[2] The AMD processor roadmap for industry standard servers, 2010.

[3] M. Awasthi et al. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *PACT-19*, 2010.

[4] J. Casazza. First the tick, now the tock: Next generation intel microarchitecture (Nehalem). In *Intel White Paper*, 2009.

[5] R. Chandra et al. Scheduling and page migration for multiprocessor compute servers. In *ASPLOS*, 1994.

[6] V. Cuppu et al. A performance comparison of contemporary DRAM architectures. In *ISCA-26*, 1999.

[7] R. Das et al. Application-to-core mapping policies to reduce interference in on-chip networks. In *Carnegie Mellon SAFARI Technical Report No. 2011-001*, 2011.

[8] E. Ebrahimi et al. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *ASPLOS-15*, 2010.

[9] E. Ebrahimi et al. Prefetch-aware shared resource management for multi-core systems. In *ISCA-38*, 2011.

[10] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *MICRO-37*, 2004.

[11] Y. Kim et al. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA-16*, 2010.

[12] Y. Kim et al. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *MICRO-43*, 2010.

[13] A. Lebeck et al. Power aware page allocation. In *ASPLOS-9*, 2000.

[14] C. J. Lee et al. Improving memory bank-level parallelism in the presence of prefetching. In *MICRO-42*, 2009.

[15] C. Luk et al. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI, 2005.*

[16] Micron. *1Gb DDR2 SDRAM Component: MT47H128M8HQ-25.*

[17] T. Moscibroda and O. Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX Security*, 2007.

[18] T. Moscibroda and O. Mutlu. Distributed order scheduling and its application to multi-core DRAM controllers. In *PODC*, 2008.

[19] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO-40*, 2007.

[20] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA-35*, 2008.

[21] C. Natarajan et al. A study of performance impact of memory controller features in multi-processor server environment. In *WMPI-3*, 2004.

[22] K. Nesbit et al. Fair queuing memory systems. In *MICRO-39*, 2006.

[23] S. Phadke and S. Narayanasamy. MLP aware heterogeneous memory system. In *DATE*, 2011.

[24] N. Rafique et al. Effective management of DRAM bandwidth in multicore processors. In *PACT*, 2007.

[25] S. Rixner et al. Memory access scheduling. In *ISCA-27*, 2000.

[26] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS-9*, 2000.

[27] K. Sudan et al. Micro-pages: increasing DRAM efficiency with locality-aware data placement. In *ASPLOS-15*, 2010.

[28] H. Vandierendonck and A. Seznec. Fairness metrics for multi-threaded processors. *IEEE Computer Architecture Letters*, Feb 2011.

[29] B. Verghese et al. Operating system support for improving data locality on CC-NUMA compute servers. In *ASPLOS-7*, 1996.

[30] G. L. Yuan et al. Complexity effective memory access scheduling for many-core accelerator architectures. In *MICRO-42*, 2009.

[31] S. Zhuravlev et al. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS-15*, 2010.

[32] W. Zuravleff and T. Robinson. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. 1997.