DEMM: a Dynamic Energy-saving mechanism for Multicore Memories

Akbar Sharifi[1], Wei Ding[2], Diana Guttman[3], Hui Zhao[4], Xulong Tang[5], Mahmut Kandemir[5], Chita Das[5]
[1]Facebook [2]Qualcomm [3]Intel [4]University of North Texas [5]Pennsylvania State University
Email: {akbarsh, dw841128, alabanzai, hzhao5}@gmail.com, {xzt102, kandemir, das}@cse.psu.edu

***Abstract*—Since main memory system contributes to a large and increasing fraction of server/datacenter energy consumption, there have been several efforts to reduce its power and energy consumption. DVFS schemes have been used to reduce the memory power, but they come with a performance penalty. In this work, we propose DEMM, an OS-based, high performance DVFS mechanism that reduces memory power by dynamically scaling individual memory channel frequencies/voltages. Our strategy also involves clustering the running applications based on their sensitivities to memory latency, and assigning memory channels to the application clusters. We introduce a new metric called Discrete Misses per Kilo Cycle (DMPKC) to capture the performance sensitivities of the applications to memory frequency modulation. DEMM allows us to save power in the memory system with negligible impact on performance. We demonstrate around 25% savings in the memory system energy and 10% savings in the total system energy, with only a 4% loss in workload performance.**

I. Introduction

Dynamically changing the operating voltage and frequency (DVFS) has been employed as an effective strategy to reduce the power and energy consumption of the main memory system in modern multicores [1], [2], [3], [4]. Here, similar to DVFS in the processor domain, the main idea is to reduce the memory power by lowering the operating frequency. However, a slower memory system can impose performance overhead and increase the execution times of the running applications due to increased off-chip access latencies. Performance loss of applications can cause two main problems: first, some applications cannot tolerate the loss in performance; and second, although the power consumption of the memory system decreases by lowering the frequency, it may not be beneficial in terms of the total system energy, since the application execution times increase. Deng et al. [3] proposed a dynamic scheme that adjusts the frequency of the memory system with the goal of saving energy, while considering the imposed performance overhead. In their work, at each time epoch, they compute the performance slack (the tolerable performance loss) for each running application. Then, the memory frequency is reduced based on the smallest slack among all of the running applications. However, three issues can limit the use of this approach: (1) computing the slack for each application requires a reference execution time which may not be easily available; (2) the mathematical model proposed in their work to capture the impact of the frequency scaling on the application performance of in-order cores can be quite complex and highly inaccurate for out-of-order processors and multicores with a large number of cores; and (3) once the slacks are computed for running applications, the smallest slack among all applications should be considered in order to find an appropriate memory frequency. This is because changing the frequencies of the memory channels affects all the running applications, which have different sensitivities to the off-chip access latency. This can be very restrictive in practice since any sensitive application has the potential to become a bottleneck when reducing the memory frequency.

To overcome these limitations, we introduce DEMM, an OS-based, high performance DVFS mechanism for reducing memory system power by scaling memory frequencies/voltages to match application demand. *Our strategy involves partitioning applications at runtime by their sensitivity to memory latency*. Less sensitive applications can tolerate a slower memory channel without loss of performance; however, the most sensitive applications should still be given the maximum memory channel frequency or they will suffer from reduced performance with little energy gain. We introduce a novel metric called *Discrete Misses per Kilo Cycle* (DMPKC) to capture the sensitivity of running applications.

The main **contributions** of this paper are as follows: (1) we introduce DMPKC as a new metric to capture the sensitivities of the running applications to memory frequency scaling; and (2) we propose DEMM as a high-performance, energy-saving DVFS mechanism for the memory system that dynamically partitions the memory channels across the concurrently-running applications and assigns frequency levels to the memory channels. DEMM dynamically adjusts this partitioning to achieve the optimal assignment of applications to channel frequencies at each time slice. Starting from the lowest frequency, it gathers cores (applications) with similar DMPKC values until the channels at that frequency can be fully utilized, and then assigns groups with higher DMPKC values to increasing channel frequencies until all cores have been assigned. In our experiments, DEMM was able to save 25% of memory energy, and 10% of system energy with only a 4% loss of performance. We also report a set of results using multithreaded applications (SPECOMP [5] and SPECJBB[6]).

II. Motivational Results

A. Background

Figure 1 illustrates the high level view of the multicore system that we target in this paper. In our default system, 32 cores execute co-runner applications and 4 on-chip memory controllers manage the flow of the off-chip accesses. The detailed system parameters are given in Table II.

DRAM organization. In Figure 1, the main memory system consists of 4 channels and each of them is controlled by a memory controller (MC). One or more DRAM boards, called dual inline memory modules (DIMMs), are connected to each

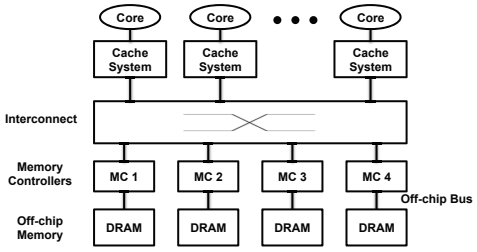


Fig. 1: High level view of our target system.

channel. DRAM chips are placed on the DIMMs and each chip participates in providing a part of the requested data. A *rank* is the subset of the DRAM chips that are activated for a single off-chip access. Data is stored in 2D memory arrays (organized as rows and columns) called *banks*. There are several banks per DRAM chip. Once data from a bank is requested, first the corresponding row is loaded into a buffer called the *row buffer*, and then the requested portion of the row data is selected based on the column address. More details on the DRAM organization can be found elsewhere [7], [8], [9], [10], [11], [12], [13], [14].

Address mapping. In Figure 1, when a miss occurs in the cache system (L1 and L2), an off-chip memory request is sent to one of the MCs via the interconnect module [15]. The target memory controller is determined based on the mapping of physical addresses to channels and MCs. Two commonly-used policies for the channel mapping are *cache line interleaving* and *page interleaving*. In the cache line interleaving policy, two consecutive cache lines are mapped to two consecutive channels, whereas in page interleaving two consecutive OS pages are mapped to consecutive channels. We use the page interleaving policy in most of our experiments, and carry out a sensitivity analysis with line interleaving. Under both mapping policies, the off-chip memory requests generated by an application are distributed across the channels, and each memory controller receives requests from all running applications (we assume one-to-one mapping between applications and cores).

DVFS in main memory. DVFS can be employed to reduce the power and energy consumption of the main memory system in modern multicores [1], [2], [3], [16], [17], [18], [19]. For instance, the JEDEC standard [20] provides such a mechanism in emerging memory systems. This mechanism can be used in three main components of a memory subsystem: memory controllers (MCs), off-chip buses, and DIMM modules. In our study, considering real systems, we assume that hardware lets OS change the frequency of DIMMs, buses and the voltage/frequency of the MCs. Further, to avoid extra synchronization hardware, we assume that DIMMs and the off-chip bus operate at the same frequency and the frequency of the MC is set to double the bus frequency. Note that, the voltage of the MC is adjusted based on the assigned frequency. The same assumptions have also been made by prior studies [3]. Therefore, from now on, we refer to one frequency value for each channel as the channel frequency. Lowering the frequency and voltage of a channel reduces the background power (leakage and refresh) and dynamic power (similar to the CPU DVFS, $power \propto voltage^2 \times frequency$ [3]) of the memory

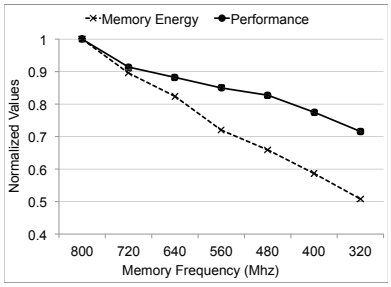


Fig. 2: Memory energy and the workload performance for different memory frequencies. The results are normalized to the base case in which the memory system operates at the maximum frequency (800 Mhz).

system. However, reducing the frequency also increases the DRAM access time, bus data transfer time and the memory request waiting time in MC bank queues [3]. As a result, the off-chip memory access delay increases. This added delay in turn affects the execution times of the running applications. Therefore, although reducing the frequency saves power and energy in the memory system, the resulting performance loss should be taken into account.

B. Motivational Results

In this section, we present and discuss the experiments we ran to motivate our proposed scheme. For these experiments, we use workload-1 in Table III. We ran this workload on our default 32-core target system with 4 memory channels, seven times, each time with a different memory frequency/voltage. We assume that all 4 memory channels have the same frequency. Figure 2 plots the memory energy consumption and the workload performance under different memory frequencies. Note that the values are *normalized* to the base case in which the memory system operates at its maximum frequency (800 Mhz in this case). We refer to the "average speedup" of the workload applications (in terms of IPC) as the "workload performance". As expected, the memory energy decreases with lowering the memory frequency. However, this energy reduction comes with a performance penalty. For instance, although the memory energy is reduced about 50% when the frequency is set to 320 Mhz, there is a 30% performance loss, implying that the execution times of the applications have increased about 30%.

As mentioned above, the performance values presented in Figure 2 are the "average values" across all the applications in the workload. However, *different applications may have different sensitivities to modulations in the memory frequency*. Figure 3 plots the individual performances of the applications on this workload, when the memory frequency is set to 720 Mhz. We see that reducing the memory frequency affects the performance of applications to varying degrees. For instance, this reduction slows down the application running on core 13 significantly, whereas core 29 remains largely unaffected.

In the experiment above, we assume that all the channels operate at the same frequency. However, any of the cores/applications may face performance degradation even when only the frequency of one of the channels it uses is reduced, even if all other channels operate at the maximum

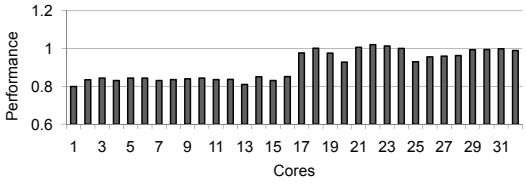


Fig. 3: The slow-downs of individual applications when the memory frequency is set to 720 Mhz.

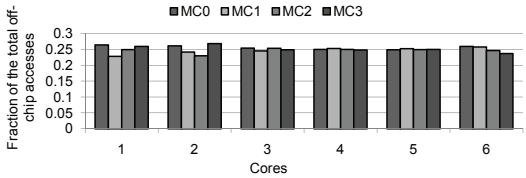


Fig. 4: Distribution of the off-chip accesses across 4 memory channels in Figure 1 for 6 cores.

frequency. This is because, as mentioned before, the memory accesses of different cores are typically distributed across all memory channels according to the address mapping policy employed by the hardware and, consequently, reducing the frequency of one of the channels may make the memory requests coming to that channel become a bottleneck for the performance of the running applications. Figure 4 plots the distribution of the memory requests of 6 cores across the 4 memory channels in the previous experiment. As can be observed, the memory accesses issued by these six cores are almost evenly distributed across the 4 MCs (each MC receives around 25% of the total off-chip accesses issued by the same core).

Based on our observations, next we perform our final motivational experiment, where we first select a subset of applications that are not sensitive to the memory frequency scaling (performance loss $< 2\%$ in Figure 3). We find 10 applications in this category, and then map all the pages of these applications to MC0, one of our memory controllers. We also map the pages that belong to the other applications to MC1, MC2 and MC3. By doing this, the off-chip memory requests are clustered into two groups: MC0 receives the requests from the non-sensitive applications, and MC1, MC2 and MC3 receive the rest of the requests. In the next step, we assign frequencies to memory controllers. Recall that, in Figure 3, the frequencies of all the 4 channels were set to 720 Mhz. However, in this experiment, the frequency of MC0 is set to 480 Mhz and the frequencies of MC1, MC2 and MC3 are set to the maximum value (800 Mhz). The goal here is to reduce the memory energy consumption by reducing the memory frequency/voltage and, at the same time, avoid the potential performance loss of sensitive applications by mapping them to the MCs that operate at the maximum frequency available. Figure 5 compares the workload energy and performance in two cases: (1) when no clustering is used and the frequencies of all the channels are reduced (Figure 2; 720 Mhz), and (2) when clustering is used. It is clear from this figure that we achieve better performance and save more energy (about 7%) by clustering (note that the values are *normalized* to the case in which all the memory channels operate at the maximum frequency). With clustering,

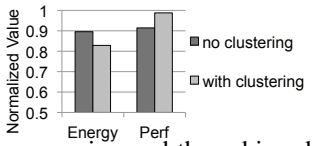


Fig. 5: The energy saving and the achieved performance loss reduction by employing our application clustering.

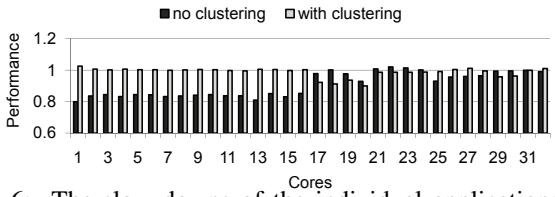


Fig. 6: The slow downs of the individual applications when clustering is employed.

the power consumption is reduced by lowering the memory frequency/voltage, and at the same time the performance loss experienced by the sensitive applications is limited. As can be observed from Figure 6, clustering improves the performance of those applications that lose significant performance when no clustering scheme is employed (see Figure 3 for comparison).

These results clearly show that clustering of applications can be used with memory frequency scaling to save energy without performance penalty. However, there are at least three issues to be addressed: (1) how to partition applications into clusters, (2) how to assign memory controllers to clusters; and (3) how to set frequencies of memories.

III. DEMM

The previous section shows the potential benefits of using clustering to maximize the benefits of memory system DVFS. Comparing Figures 3 and 6 clearly highlight the importance of clustering. However, the reality is a bit more complex than this, as illustrated in Figure 7. This figure shows, for two applications from our experimental suite, the ideal memory frequency they would prefer over a portion of their execution (measured in epochs). We observe that, the frequency preferred by the two applications varies significantly over time depending on the phase behavior. Consequently, a clustering strategy that sets memory frequencies/voltages to save energy and minimize potential performance loss should do it *dynamically*. Further, we note from this plot that different applications prefer different frequencies in most of the epochs and, if they are to share the same memory controller(s) (which may actually be the case depending on other applications in the workload), the frequency/voltage of the corresponding memory (memories) should be selected very carefully to strike the right balance between energy saving and performance loss. Considering these observations, we propose DEMM, which is a *dynamic* scheme that regulates the memory channel frequencies (and, correspondingly, voltages) and application clusters over the course of execution with the goal of saving as much memory energy as possible with minimal performance impact. Our approach takes the system performance counters as input and gives the application clustering decisions and the corresponding channel frequencies as output. The OS runs the DEMM algorithm over the execution at fixed time epochs (intervals) and enforces the output decisions. The hardware

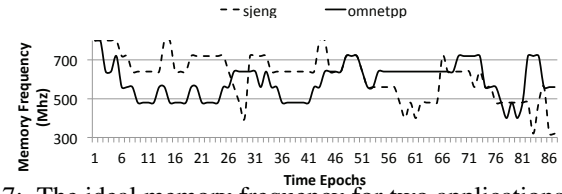


Fig. 7: The ideal memory frequency for two applications from our experimental suite.

provides the system performance counters which are used to identify the sensitivities of the running applications to memory frequency/voltage scaling.

A. The DMPKC Metric

DEMM first clusters the running applications into different groups based on the application sensitivities and then assigns memory channels to the application clusters. At the same time, the algorithm also determines the values of the frequencies (and corresponding voltages) assigned to the memory channels. Recall that Figure 3 shows the sensitivities of applications to memory frequency scaling. We generated this graph by running our workload twice (once with the maximum memory frequency as a baseline and once with the reduced memory frequency) and comparing the results. However, such a comparison cannot be performed at runtime. Since DEMM is a dynamic mechanism, the application sensitivities need to be estimated based on a parameter that can be measured over the execution. In DEMM, we use a novel metric called *Discrete Misses Per Kilo Cycles (DMPKC)* to predict the sensitivity of each application. DMPKC of an application is defined as *the number of discrete groups of off-chip memory requests (per thousand cycles) that belong to the same application*. A subset of memory requests is considered a "group" if the difference between the issue times of any two of the requests in that group is less than a threshold (*DMPKC-th*). The DMPKC value is incremented by 1 for each off-chip memory request group. This is because our goal is to consider off-chip requests issued by the same application in parallel in order to estimate the application sensitivities to the off-chip delay increase coming from the memory frequency reduction. Figure 8 illustrates how the DMPKC is defined. In this figure, each dot represents a cache miss for a given application, and the figure also shows the time that each last level cache miss occurs. As can be seen, the MPKC (Misses Per Kilo Cycle) value of this example is 10, since there are ten misses over 1K cycles. However, only two groups can be identified in Figure 8 and therefore the DMPKC value is 2 (the issue times of any two of the requests in each group is less than the threshold). In DEMM, we assume that hardware provides the values for this metric. For the hardware implementation, there is a counter for each core that counts the number of discrete misses for each application. To count the number of discrete misses, the hardware monitors the arrival times of the memory requests and increments the counter for each request group (*not* for each request).

In our experiments, we observed that, *if two applications have the same number of misses per thousand cycles, the application with higher off-chip access parallelism is less sensitive to the memory frequency reduction* since the added off-chip access delays have less impact on the application

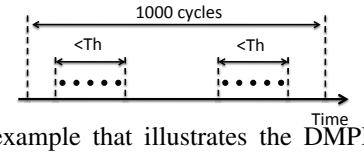
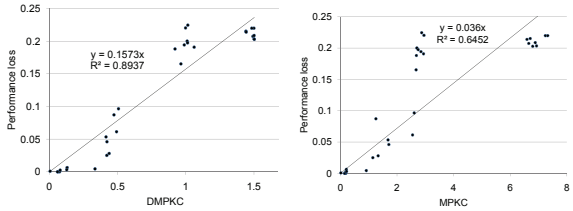


Fig. 8: An example that illustrates the DMPKC definition. Each dot represents a cache miss for a given application.

performance if the memory requests of the application are issued in parallel. Figure 9a plots the correlation between the average DMPKC values of the 32 applications in our motivational example (in Section II-B) and their performance loss when the memory frequency is scaled down (each dot in this plot represents an application). As can be seen, *applications with higher DMPKC values lose more performance*. Figure 9b plots the correlation as in Figure 9a but the x-axis is *Misses Per Kilo Cycles* (MPKC) instead of DMPKC. The correlation now is much weaker, without considering the off-chip access parallelism.



(a) DMPKC (b) MPKC

Fig. 9: The correlation between the average DMPKC and MPKC values of 32 applications and their performance loss when the memory frequency is scaled down to 720 Mhz (each dot represents an application).

B. Details of the Algorithm

At each time interval, the OS runs the DEMM algorithm using the DMPKC values of the running applications. Algorithm 1 gives the pseudo-code for DEMM. A brief description of the parameters used in this algorithm is given in Table I. Algorithm 1 takes the DMPKC values as input and gives the frequency of the memory channels ($CHANNEL\_FREQ$) as well as the channel mappings ($CORE\_MAP$) as output. As an example, if the algorithm maps core i to two memory channels (channel-0 and channel-1, i.e, $CORE\_MAP[i][0] = 1$ and $CORE\_MAP[i][1] = 1$), this means that the OS pages accessed by core i, from this point on, will be mapped to channel-0 and channel-1. The main loop of the algorithm (Lines 5-27) terminates once all the channels in the memory system are assigned to the cores. The algorithm starts working with the minimum available memory frequency ($freq\_level$) and a minimum $DMPKC$ threshold ($dmpkc\_level$). First, it counts the number of the cores that have $DMPKC$ values less than $dmpkc\_level$ (Lines 7-11). Then, $get\_num\_channels(.)$ decides how many channels can be allocated to that number of cores (we will later discuss the policy we employ for that). Based on the number decided by $get\_num\_channels(.)$, the cores selected at Lines 7 to 11 are mapped to the available channels with the $freq\_level$ frequency. Finally, $dmpkc\_level$ and $freq\_level$ values are incremented for the next iteration of the main loop. In summary, each $dmpkc\_level$ has a corresponding $freq\_level$, and DEMM assigns one or more channels (with $freq\_level$) to the cores with that $dmpkc\_level$ (if

Algorithm 1 DEMM Algorithm

Input: $DMPKC[1..C]$
Output: $CHANNEL\_FREQ[1..Ch]$, $CORE\_MAP[1..C][1..Ch]$
1: $dmpkc\_level = DMPKC\_th$
2: $freq\_level = FREQ\_BASE$
3: $current\_channel = 0$
4: $core\_done[1..C] = 0$
5: **while** $current\_channel < TOTAL\_CHANNELS$ **do**
6: $counter = 0$
7: **for** $i = 0$ to C **do**
8: **if** $core\_done[i] == 0$ and $DMPKC[i] < dmpkc\_level$ **then**
9: $counter++$
10: **end if**
11: **end for**
12: $ch\_num = get\_num\_of\_channels(counter, C, Ch, freq\_level)$
13: **for** $j = 0$ to ch_num **do**
14: **for** $i = 0$ to C **do**
15: **if** $core\_done[i] == 0$ or $core\_done[i] == freq\_level$ **then**
16: **if** $DMPKC[i] < dmpkc\_level$ **then**
17: $CORE\_MAP[i][current\_channel] = 1$
18: $CHANNEL\_FREQ[current\_channel] = freq\_level$
19: $core\_done[i] = freq\_level$
20: **end if**
21: **end if**
22: **end for**
23: **end for**
24: $current\_channel = current\_channel + ch\_num$
25: $inc(dmpkc\_level)$
26: $inc(freq\_level)$
27: **end while**

C	number of cores
Ch	number of memory channels
$DMPKC\_th$	a constant threshold value (default : 0.6)
$FREQ\_BASE$	minimum channel frequency (320 Mhz)
$CHANNEL\_FREQ$	frequency of each memory channel
$CORE\_MAP$	$CORE\_MAP[i][j] = 1$ shows that the OS pages of core i can be mapped to channel j
$core\_done$	$core\_done[i] = 0$ shows that DEMM has not taken of core i so far
$inc(dmpkc\_level)$	increments $dmpkc\_level$ by 1 level (default : 0.1)
$inc(freq\_level)$	increments $freq\_level$ by 1 level

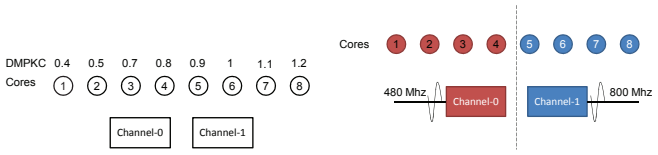
TABLE I: Parameters/functions used by DEMM.

they are sufficient in number to be allocated channels). We employ the policy used in the channel partitioning scheme proposed in [21] to determine the number of memory channels at line 12. This policy is based on the number of cores in the subset, the total number of cores, and the total number of channels. If C is the total number of cores and Ch is the total number of channels, $\lfloor \frac{n}{C} \times Ch \rfloor$ gives the number of channels that will be allocated to the n selected cores. The goal of this policy is to avoid high contention on the memory channels. For example, if there are 32 applications in a workload, 2 out of 4 memory channels are assigned to a set with 16 applications. We want to emphasize that, *our experimental results include all the performance overheads incurred by the memory contentions caused by DEMM.*

C. Example Application of the Algorithm

We now go over an example to show how DEMM works in practice. Suppose that we have an 8-core machine with 2 memory channels and, at the k^{th} time interval, the $DMPKC$ values of the running applications are assumed to be as shown in Figure 10a. We further assume that our base $DMPKC$ is 0.6 [1] and the available memory frequencies are 320, 400, 480, 560, 640, 720 and 800 Mhz. Therefore, $dmpkc\_level$ and $freq\_level$ values start at 0.6 and 320 Mhz, respectively.

[1]The starting value 0.6 is determined based on the average $DMPKC$ values we observed in our experiments for the very low sensitive applications.



(a) Initial state (b) The output of the DEMM algorithm

Fig. 10: An example illustrating how DEMM works.

Processors	32 out-of-order cores with private L1 data and instruction caches. instruction window size: 128, LSQ size: 64
Private L1 D&I–Caches	Direct mapped, 32KB, 64 bytes block size, 3 cycle access latency
L2 Cache	64 bytes block size, 10 cycle access latency
L2 Cache Bank Size	512KB per core
Number of Banks Per Memory Controller	16
Memory Configuration	DDR-800, Memory Bus Multiplier: 5, Bank Busy Time: 22 cycles, Rank Delay: 2 cycles, Read-Write Delay: 3 cycles, Refresh Period: 3120, Memory CTL latency: 20 cycles
Epoch length	$5M$ cycles
Simulation length	$100M$ cycles

TABLE II: Baseline configuration.

At the first iteration, the algorithm finds two cores that have $DMPKC$ values less than 0.6 (cores 1 and 2). Based on our employed policy, two cores are not sufficient to be allocated a channel in this case. So, $dmpkc\_level$ and $freq\_level$ are increased by two more levels to 0.8 and 480 Mhz, respectively, at which point four cores (cores 1 to 4) have $DMPKC$ values less than 0.8 and channel-0 is assigned to them. We apply the same strategy to the rest of the cores and channel-1 is assigned to them with a frequency of 800 Mhz. Figure 10b shows the final mapping in our example. However, in this example, if we assume that the DMPKC values of cores 3 and 5 are 1, as per the DEMM algorithm, the frequencies of both of the channels are set to 800 Mhz, since the number of non-sensitive applications (with low DMPKC) is not high enough to deserve a channel. In that case, most of the applications are sensitive and DEMM does not reduce memory frequencies. However, we can save power by increasing the aggressiveness of DEMM as will be discussed later in our sensitivity experiments.

D. Discussion

When a running application tries to access a page in the memory, there are three possible scenarios: (1) The page is not already allocated and a page fault occurs. In this case, the page will be allocated in one of the preferred channels determined by DEMM. (2) The page is already allocated and resides in one of the preferred channels suggested by DEMM. No action is taken here, and the page is simply accessed by the application. (3) The page is already allocated in the memory but is not located in one of the preferred channels (due to the changing of the channel mapping over the execution). In this case, there are two options. One option is to migrate the page to one of the preferred channels; this leads to TLB and cache block invalidation, and incurs performance overhead [21]. The other option is not performing any migration and the page is accessed from its current location (channel). We chose the second option due to the performance and power overheads caused by migration (also quantified in our experimental evaluation). In DEMM, the initial page mapping is the page

interleaving policy in which consecutive pages are mapped to consecutive channels. Although at the beginning, some pages of sensitive applications may be mapped to slow channels by employing DEMM, as the application makes progress, the number of accesses to the old pages reduces over time and the new pages are allocated to the preferred channels. Also, the old pages (residing potentially in non-preferred channels) are not accessed much as the execution progresses (due to *page locality*). Further, the sensitivity (behavior) of the applications does not change very frequently over the course of execution and therefore, the channel partitioning does not change very often. Consequently, the overheads caused by the mentioned problem are not much when we look at the whole execution. Note also that we avoid unbalanced core-to-memory channel mapping in our algorithm. Specifically, if the number of cores is not sufficient to be allocated to a channel, we increase $dmpkc\_level$ and $freq\_level$ to include more cores. Finally, since in our scheme an application's data are mapped to a dedicated set of memory channels, some bandwidth may be lost; but the impact of it is included in our experimental results.

IV. Experimental Evaluation

Setup: We use GEMS [22] as our simulation framework. For calculating the memory system energy, we employ the power model provided by Micron for DDR3 memory technology [23]. The Micron power model takes the activity and frequency as input and estimates the power consumption of the memory system. Energy numbers are calculated based on the obtained power numbers and the execution times of the running applications. Table II gives our baseline configuration. Our baseline system has 32 cores that run 32 applications and 4 DDR3-800 memories (channels) connected to 4 memory controllers. More details on the timing and implementation of the memory system can be found in [24]. We factor the performance impact of the memory frequency regulation as the variation in the access and bus data transfer time. Note that, DEMM can be employed in the multicores with or without shared caches, and it captures the sensitivities of the running applications in both cases by monitoring the *off-chip accesses*.

Metrics: We compare DEMM against a "base scheme" in which *no* clustering is employed and the entire memory system frequency is varied to obtain the best performance, i.e., all the MCs operate at the same frequency. Note that this base scheme is similar to the scheme proposed in [3]) in the sense that the frequency of all the MCs are changed with the goal of improving the memory power/performance efficiency. Note also that, allowing the MCs to operate at different frequencies (similar to the approach used in [4]) does not improve the efficiency of our base scheme significantly since, as can be seen from Figure 4, the memory accesses are *uniformly* distributed across the MCs and the memory frequency scaling of different MCs will have the same impact on the performance of running applications. Also, although not evaluated in this work, it is possible to integrate DEMM with core and NoC [25] voltage scaling techniques to further increase energy savings.

We evaluate the following metrics: memory energy, full-system energy, performance, and Energy-Performance-Ratio (EPR). Note that, all the energy, performance, and EPR values presented in this section are *normalized* to the case where all the memory channels operate at the maximum frequency (800 Mhz). To estimate the full system-energy saving, we make the same assumption as in [3], [26] that, on average, 40% of the system energy is consumed in the memory system. We use *fair speedup* as our metric for performance evaluation. The Fair Speedup (FS) metric is defined as the *harmonic mean* of per application speedup with respect to the baseline:

$$FS = \frac{N}{\sum_{i=1}^{N} \frac{IPC_{base}(i)}{IPC_{scheme}(i)}},$$

where N is the number of applications, $IPC_{base}(i)$ is the Instructions-Per-Cycle of application i under assumption that all the channels operate at the maximum frequency, and $IPC_{scheme}(i)$ is the IPC of the same application when our scheme is employed. FS can be used to assess the overall improvement in IPC values across different schemes. Energy-Performance-Ratio (EPR) is a metric that captures the ratio between the amount of energy that is saved and the performance loss of the employed mechanism, and is defined as $EPR = \frac{Energy}{Performance}$. As an example, suppose that we have two schemes, Scheme-1 and Scheme-2, and both save 20% energy but Scheme-1 incurs 10% performance overhead while Scheme-2 incurs 20%. The EPR values of Scheme-1 and Scheme-2 would be 0.89 and 1. A smaller EPR value indicates a better performance-energy efficiency since it implies that we can save more energy with less performance loss. This metric in a sense represents a "tradeoff" between energy and performance; in general, a scheme with an EPR greater than 1 does not represent a good tradeoff, meaning that the energy savings of that scheme may not worth performance costs.

Workloads: Most of our experiments are performed using the workloads composed from the applications in SPEC2006 benchmark suite [27]. Table III gives the 30 workloads that we used in our experiments on our 32-core system (the numbers in parentheses represent the number of copies for each application in the workload). The workloads listed in Table III cover *all* applications and are categorized into three different groups based on the "memory intensity" of the applications: (1) Workloads 1 through 10 (mixed workloads): in these workloads, half of the applications are memory intensive (applications with high MPKI) and the remaining ones are memory non-intensive. (2) Workloads 11 through 20: all applications in this category are memory intensive. (3) Workloads 21 through 30: none of the applications in this group is memory intensive. MPKI values representing the memory intensities of the applications SPEC2006 can be found in [28] (for these applications, we use 1-to-1 application-to-core mapping, as they are single threaded). In addition to these workloads, we also performed a set of experiments with multithreaded HPC workloads and SpecJBB [6] (in this case, the application is parallelized over all available cores).

Main Results: Figure 11a plots the memory energy consumption for the workloads given in Table III for the base scheme and DEMM. Note that the energy numbers are *normalized* to the case in which all memory channels operate at the

maximum frequency. The reflections of these memory energy savings on the system total energy are shown in Figure 11b. As can be observed from Figures 11a and 11b, the base scheme and DEMM save, on average, 17% and 24% memory energy (7% and 10% system energy), respectively, for our mixed workloads (workloads 1 to 10) by reducing the memory frequencies. That is, DEMM saves 7% more memory energy than the base scheme. Recall that our base scheme is similar to a recently proposed one [3] (the entire memory system frequency is varied). As shown in Figure 12a, DEMM saves this energy while incurring, on average, only about 5% performance loss in the case of the mixed workloads; in comparison, the average performance loss incurred by the base scheme is around 13%. This is because DEMM reduces the performance overhead coming from the memory frequency scaling by clustering the running applications and partitioning the memory channels over the execution. DEMM's clusters can take advantage of the varied memory demands of applications in the mixed workload, while the base scheme must subject all applications to the same channel frequency. Figure 12b plots the average memory access latency per request when the base scheme and DEMM are employed for each workload. As can be seen, DEMM imposes less overhead on memory accesses as compared to the base scheme. These changes in the off-chip latency cause the performance variations seen in Figure 12a.

Note that, a major portion of the performance loss caused by DEMM is because of the memory contention caused by assigning memory intensive applications to the same channels. However, as our experimental result show, this performance overhead is less than the case (base scheme) where no channel partitioning is performed and the contention caused by lowering the memory frequency affects all the running applications (sensitive and non-sensitive).

In the case of memory-intensive workloads (workloads 11 to 20), as can be observed in Figures 11a and 11b, the energy consumption values are not reduced by employing the base scheme. This is because, *there is a high load and pressure on the memory system*; even a slight decrease in the speed of the memory system (by reducing the frequency) can lead to a significant increase in the memory queue latencies and ultimately to performance loss. Therefore, although reducing the memory frequency reduces the power consumption, the resulting performance loss increases the total energy consumption for the base scheme. Again, DEMM saves 7% and 3%, memory and system energy respectively, by avoiding the performance loss caused by the frequency scaling. However, even DEMM could not save energy for some of the workloads, such as workloads 15 and 17, since no non-sensitive application (access pattern) could be found in any execution epoch.

In general, memory non-intensive applications are less sensitive to the memory frequency regulations. Therefore, more energy can be saved by scaling the memory system frequency. As can be observed from Figures 11a and 11b, DEMM on average saves 43% and 17% memory and system energy with 5% performance loss. However, in most of these (non-intensive) workloads, the same amount of energy and performance is also saved by employing the base scheme.

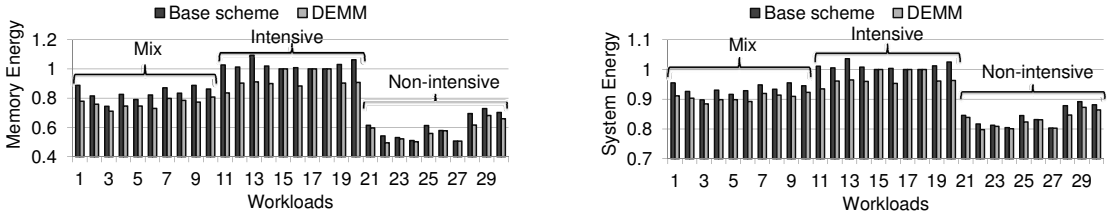
MIXED	workload-1	libquantum(4), leslie3d(4), GemsFDTD(4), soplex(4), astar(4), gobmk(4), calculix(4), bzip2(4)
	workload-2	lbm(4), leslie3d(4), sphinx3(4), soplex(4), perlbench(4), astar(4), sjeng(4), namd(4)
	workload-3	mcf(4), xalancbmk(4), leslie3d(4), soplex(4), sjeng(4), namd(4), gobmk(4), gamess(4)
	workload-4	mcf(4), libquantum(4), leslie3d(4), soplex(4), omnetpp(4), perlbench(4), h264ref(4), dealII(4)
	workload-5	lbm(4), libquantum(4), leslie3d(4), GemsFDTD(4), astar(4), gromacs(4), gamess(4), bzip2(4)
	workload-6	lbm(4), libquantum(4), leslie3d(4), leslie3d(4), povray(4), namd(4), dealII(4), bzip2(4)
	workload-7	mcf(4), lbm(4), libquantum(4), leslie3d(4), omnetpp(4), astar(4), gromacs(4), bzip2(4)
	workload-8	mcf(4), lbm(4), leslie3d(4), GemsFDTD(4), perlbench(4), astar(4), perlbench(4), gamess(4)
	workload-9	xalancbmk(4), leslie3d(4), GemsFDTD(4), soplex(4), omnetpp(4), gromacs(4), gamess(4), bzip2(4)
	workload-10	mcf(4), lbm(4), xalancbmk(4), leslie3d(4), omnetpp(4), sjeng(4), povray(4), bzip2(4)
MEM INTENSIVE	workload-11	mcf(8), libquantum(8), leslie3d(8), GemsFDTD(8)
	workload-12	mcf(8), lbm(8), libquantum(8), leslie3d(8)
	workload-13	xalancbmk(8), libquantum(8), GemsFDTD(8), soplex(8)
	workload-14	lbm(8), xalancbmk(8), GemsFDTD(8), soplex(8)
	workload-15	lbm(8), libquantum(8), leslie3d(8), GemsFDTD(8)
	workload-16	lbm(8), libquantum(8), leslie3d(8), soplex(8)
	workload-17	xalancbmk(8), libquantum(8), leslie3d(8), GemsFDTD(8)
	workload-18	mcf(8), lbm(8), leslie3d(8), sphinx3(8)
	workload-19	xalancbmk(8), leslie3d(8), GemsFDTD(8), soplex(8)
	workload-20	mcf(8), lbm(8), xalancbmk(8), libquantum(8)
MEM NON-INTENSIVE	workload-21	astar(4), omnetpp(4), perlbench(4), sjeng(4), povray(4), h264ref(4), calculix(4), bzip2(4)
	workload-22	astar(4), omnetpp(4), perlbench(4), sjeng(4), gobmk(4), gcc(4), gamess(4), bzip2(4)
	workload-23	astar(4), perlbench(4), astar(4), namd(4), gromacs(4), gobmk(4), gamess(4), bzip2(4)
	workload-24	omnetpp(4), perlbench(4), omnetpp(4), astar(4), gobmk(4), gcc(4), dealII(4), bzip2(4)
	workload-25	astar(4), omnetpp(4), perlbench(4), h264ref(4), gromacs(4), gobmk(4), dealII(4), bzip2(4)
	workload-26	perlbench(4), astar(4), omnetpp(4), astar(4), gamess(4), calculix(4), bzip2(4), bzip2(4)
	workload-27	omnetpp(4), astar(4), namd(4), namd(4), h264ref(4), gromacs(4), bzip2(4), bzip2(4)
	workload-28	omnetpp(4), astar(4), omnetpp(4), sjeng(4), namd(4), h264ref(4), gromacs(4), gamess(4)
	workload-29	omnetpp(4), omnetpp(4), astar(4), namd(4), namd(4), gamess(4), dealII(4), bzip2(4)
	workload-30	omnetpp(4), sjeng(4), povray(4), namd(4), namd(4), h264ref(4), gromacs(4), calculix(4)

TABLE III: Workloads used in our 32-core experiments. The numbers within parentheses indicate the number of running instances of the corresponding application.

This is because the sensitivities of all the applications in such workloads are almost the same, and DEMM does not bring much additional benefit by clustering the applications.

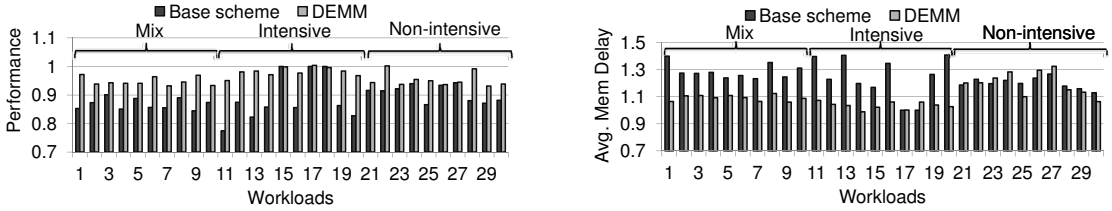
Figures 13a and 13b plot the memory and system EPRs, which capture both the energy values and performance loss (shown in Figures 11a, 11b and 12a) together in a single metric. We see from these plots that, DEMM achieves 11%, 13% and 6% system level EPR improvements for the mixed, intensive and non-intensive workloads, respectively, as compared to the base scheme. Note that, although the EPR is improved by 13% for intensive workloads as compared to the base scheme, the actual value is close to 1, which means we lose about 1% performance to save 1% system energy. It should also be observed that non-intensive workloads have the best EPR savings, indicating that they present the best opportunity to reduce the memory frequency to save power without losing much performance.

DEMM is a "dynamic scheme" and varies the application clusters, channel partitioning and frequencies during execu-



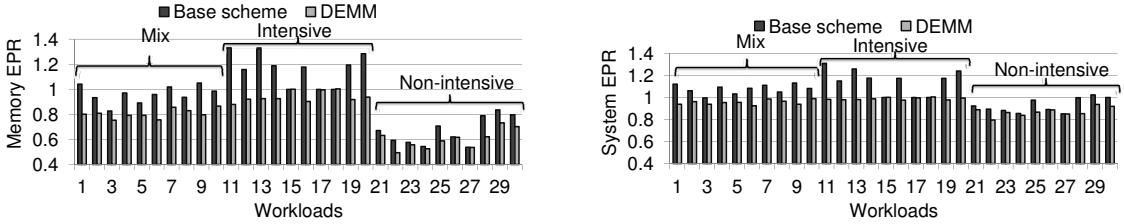
(a) Memory Energy (b) System Energy

Fig. 11: Memory and system energy consumption when DEMM and the base scheme are employed.



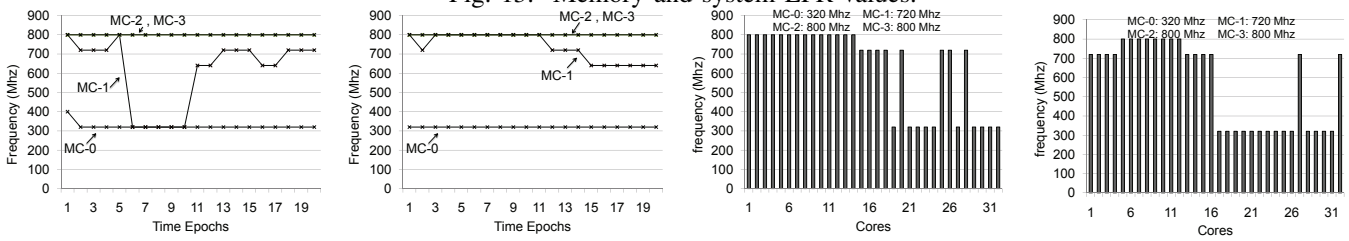
(a) (b)

Fig. 12: (a) Achieved performance (fair speedup) and (b) average memory access latency for each off-chip request (the values are *normalized* to the case where all the memory channels operate at the maximum frequency (800 Mhz)).



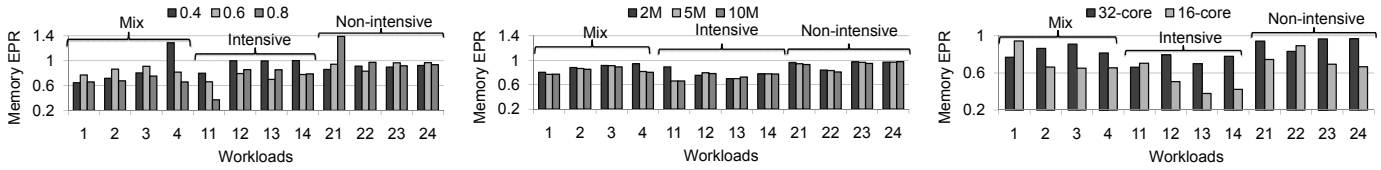
(a) Memory EPR (b) System EPR

Fig. 13: Memory and system EPR values.



(a) Memory controller frequency over time (workload-1). (b) Memory controller frequency over time (workload-2). (c) Epoch 4: memory frequency assignments (workload-1). (d) Epoch 15: memory frequency assignments (workload-1).

Fig. 14: Memory controller cluster/frequency assignments over time.



(a) $DMPKC\_th$ (b) Epoch length (c) Results with a 16-core system

Fig. 15: Sensitivity analysis.

tion. To illustrate its dynamic behavior, Figure 14a plots the frequencies assigned by DEMM to our 4 memory channels during runtime when workload-1 is running, and Figure 14b shows the same results for workload-2. Figures 14c and 14d plot two snapshots of how the channels are partitioned across the running applications at the 4th, and 15th time epochs for workload-1. For instance, the frequency shown for core 1 at the 4th time epoch (Figure 14c) is 800 Mhz and, as shown in this figure, only MC-2 and MC-3 operate at that frequency, therefore, core 1 is mapped to MC-2 and MC-3 at the 4th time interval. At the 15th epoch (Figure 14d), core 1 is assigned to 720 Mhz but only MC-1 is operating at this frequency, so core 1 is in the cluster mapped to MC-1.

Sensitivity Results. For the following experiments, we picked the first 4 workloads from each category (mixed, intensive, and non-intensive) given in Table III. $DMPKC\_th$ in the DEMM algorithm (Algorithm 1) is the main parameter to tune the aggressiveness in reducing the memory frequency. This parameter is the initial threshold used to identify the sensitivities of the applications. To better illustrate the impact

of $DMPKC\_th$ on the aggressiveness of DEMM, consider our example in Figure 10a. The $DMPKC\_th$ value in this example is assumed to be 0.6. Therefore, $dmpkc\_level$ and $freq\_level$ are increased by two more levels and reach 0.8 and 480 Mhz, respectively, where four cores (cores 1 to 4) have $DMPKC$ values less than 0.8 and channel-0 is assigned to them. However, if $DMPKC\_th$ is set to be 0.8 instead of 0.6 in this example, at the first iteration in the main loop in Algorithm 1, channel 0 will be assigned to the first four cores with the frequency of 320 Mhz. Increasing the $DMPKC\_th$ value would result in a lower frequency assignment for channel 0. Generally speaking, $DMPKC\_th$ can be used as a knob to control the aggressiveness of DEMM such that more power is saved by increasing the value of this parameter (since the levels of the assigned frequencies are reduced) with the cost of losing more performance. Figure 15a plots the memory EPR values achieved by DEMM under three $DMPKC\_th$ values: 0.4, 0.6 (default) and 0.8. As can be observed from this graph, the impact of changing the $DMPKC\_th$ values is not the same across different workloads. For instance, workload-4 benefits from increasing the $DMPKC\_th$ value (the memory EPR decreases), whereas this increase is harmful for workload-21 since the performance overhead is a dominant factor in this workload. $DMPKC\_th$ can be used to search for an optimal point for each workload by adjusting the aggressiveness of DEMM. In other words, this parameter helps us to explore alternate solutions that are in the neighborhood of the solution returned by our algorithm.

As mentioned before, the OS runs the DEMM algorithm over the execution at fixed time epochs (intervals) and enforces the output decisions. Our default value for the length of the epochs is 5M cycles (as indicated in Table II). Figure 15b gives the results for a set of experiments in which we vary the epoch lengths (this figure shows the results for the epoch length of 2M, 5M and 10M cycles). As one can observe from Figure 15b, the EPR values for some of the workloads such as workload-4 and 11 increases when the epochs are set to be smaller (2M cycles). This is mainly because, in these cases, the epochs are not long enough for DEMM to capture the sensitivities of the applications based on the $DMPKC$ values and the sensitive applications may be mapped to slow channels due to the inaccuracy in estimating the sensitivity. Figure 15b also shows that most of our workloads are not very sensitive to the epoch lengths employed. In cases where DEMM has the same performance for different time epochs such as workload-14, it would be better to set the epoch length to the largest value, in order to reduce the overhead of the algorithm (since the algorithm is run less frequently).

In another experiment, we chose half of the applications in each of our sensitivity analysis workloads to run on a 16-core multicore with 4 MCs. Figure 15c plots the memory EPR values achieved by DEMM for 32-core and 16-core systems. As can be observed from this figure, in most of the workloads, more memory EPR is saved with the 16-core system, since, as the memory load decreases, DEMM has the opportunity to cluster the running applications and reduce the channel frequencies with more flexibility.

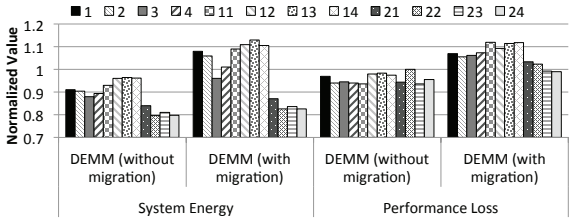


Fig. 16: Impact of page migration (normalized w.r.t base scheme).

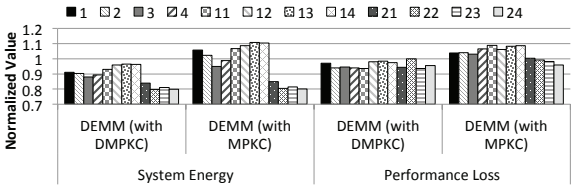


Fig. 17: Results with MPKC (normalized w.r.t base scheme).

Impact of Page Migration. Recall that our default implementation of DEMM does *not* migrate pages across memory channels during execution. Intuitively speaking, this may be an issue since, as memory channels assigned to an application and frequency assigned to those channels change over time, some of the already-placed pages may prefer different channels. In Section 3.3, we defended of the policy of not migrating pages, arguing that (1) the number of references to old page gets reduced over time and (2) page migration is a costly activity. Nevertheless, we also tested an alternative implementation of DEMM that does perform page migrations. More specifically, in this new implementation, whenever a page is not in its preferable channel, it is migrated to its preferable channel. Figure 16 plots the total system energy and performance numbers with migration for the same 12 workloads used above in our sensitivity experiments (each bar denotes a workload). One can see that activating page migrations increases energy consumption as well as performance losses, and therefore, leaving old pages in their original locations seems to be a better option.

Results with MPKC. We showed earlier in Figure 9 that DMPKC correlates much better with performance losses compared to MPKC. Figure 17 plots the energy and performance numbers if we use MPKC instead of DMPKC . These results clearly show that MPKC performs much worse than DMPKC, and therefore, considering off-chip access parallelism is key to achieving best power-performance tradeoffs. Note that DMPKC is proposed based on two important observations: (1) having a slower memory affects the memory aggressive applications (with high MPKC) more significantly since the frequency of issuing memory accesses is high, and the execution progress of these applications depends more on the off-chip access latency, and (2) an application with higher memory level parallelism (and with the same MPKC), suffers less from slower memories. DMPKC considers these two factors at the same time, whereas MPKC does not, explaining the difference between the results.

Results with Line-Level Interleaving. Recall that the results presented so far used page-level interleaving of physical accesses across memory channels. We also conducted

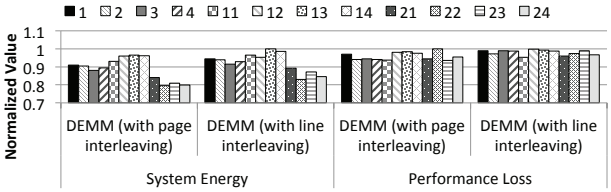


Fig. 18: Results with cache level interleaving (normalized w.r.t base scheme).

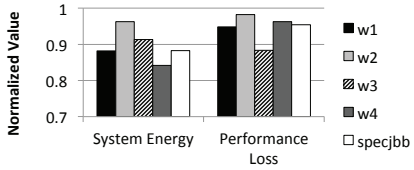


Fig. 19: Results with other benchmarks (normalized w.r.t base scheme).

experiments with line-level interleaving and the total system energy and performance loss results are plotted in Figure 18. Although the memory energy savings are not as high as in the case of page-level interleaving (due to the fact that memory accesses are less localized under line interleaving and that in turn prevents DEMM from employing very low frequencies), they are still very good, indicating that our approach works well under both types of address interleavings.

Results with Multithreaded Benchmarks. In our final set of experiments, we tested the effectiveness of DEMM when using applications other than SPEC2006. Figure 19 presents results with five additional workloads. The first four of these (w1 through w4 in the graph) are workloads drawn from SPECOMP [5], each having 4 multithreaded applications, running on 8 cores of our default configuration. The second is SPECJBB [6], which is a benchmark used to test the Java server performance. In particular, it evaluates the performance of server side Java by emulating a three-tier client/server system, with an emphasis on the middle tier. In this application, each thread represents a user that initiates transactions within a warehouse and we modeled a system with 32 clients (threads). The results plotted in Figure 19 show that our approach works well with multithreaded applications too (detailed discussion is omitted due to lack of space).

V. Related Work

Meisner et al. [29] proposed a scheme in which the load on a server determines if the server operates in active or idle mode. There also exist several works on memory energy management, with the goal of reducing energy consumption [30], [31], [32], [33], limiting peak power [32], or avoiding high memory temperatures [34], [35]. [36] and [37] proposed schemes to mitigate the latency overhead incurred by the transitions between different power modes. However, as the number of cores and concurrently running applications increases in multicores, finding the long enough memory idle periods becomes harder. Further, the performance overhead of changing the memory power mode needs to be considered [38] and, as mentioned before, this overhead may differ across different applications based on their sensitivities to the latencies seen by the off-chip memory requests. Yan et al. [39] proposed a scheme where sensitive applications are executed on a group of processors fed by an on-chip regulator.

DVFS has been employed as a scheme to reduce the power and energy consumption of main memory systems in modern multicores [1], [2], [3], [40], [41]. In [2], [42], DVFS is used for the cores and memory system together to achieve the specified power budget [43] and minimize the full system energy consumption [2], [42]. The memory DVFS schemes employed in these prior studies affect *all* co-runner applications when there are multiple MCs in the system and no channel partitioning is used. In comparison, DEMM considers the sensitivity of each application individually and attempts to improve the performance and the efficiency of the energy saving achieved by the memory DVFS. The scheme proposed in [4], unlike the DVFS mechanism proposed in [8] (and discussed in Section I), assigns different frequencies to different MCs. However, no channel partitioning is done in this work and therefore lowering the frequency of one channel affects the performance of all the running applications (sensitive and non-sensitive). This is because, as shown in Figure 4, each MC receives memory accesses from all of the applications. Other prior works such as [44], [45] proposed new DRAM organizations/architectures and data mapping schemes to improve the energy efficiency of the main memory. Note that DEMM does not change the DRAM architecture and can be employed in parallel with these schemes.

Chasapis et al. [46] address manufacturing variability in power-constrained NUMA nodes. Page allocation has been employed as a mechanism to improve the opportunity of saving memory power and energy [47], [48]. However, page migration incurs performance overheads in multicores and, in prior work they do not propose a method to allocate newly-accessed OS pages.

VI. Conclusions

We proposed a novel OS-based DVFS technique, DEMM, which utilizes the DMPKC parameter for monitoring the memory sensitivity of applications for dynamically partitioning the applications into different clusters, assigning clusters to memory channels, and modulating the frequencies of memory channels for optimizing energy conservation at the cost of minimal performance penalty. With the default values of our simulation parameters, DEMM provides around 25% saving in the memory system energy and 10% saving in the total system energy, with only a 4% loss in workload performance. Our ongoing work is on comparing DEMM with memory power capping. We are also interested in testing our approach under SMT cores.

Acknowledgment

We thank the anonymous reviewers for their feedback. This research is supported in part by NSF grants #1205618, #1213052, #1212962, #1302225, #1302557, #1313560, #1320478, #1320531, #1409095, #1409723, #1439021, #1439057, #1526750, #1629129 and #1629915, and a grant from Intel.

REFERENCES

[1] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in *ICAC*, 2011.

[2] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "CoScale: Coordinating CPU and memory system DVFS in server systems," in *MICRO*, 2012.

[3] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini, "Memscale: active low-power modes for main memory," in *ASPLOS*, 2011.

[4] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "MultiScale: memory system DVFS with multiple memory controllers," in *ISLPED*, 2012.

[5] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, "Specomp: A new benchmark suite for measuring parallel computer performance," in *Proceedings of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, ser. WOMPAT '01, 2001.

[6] "http://www.spec.org/jbb2005/."

[7] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary DRAM architectures," in *ISCA*, 1999.

[8] B. T. Davis, "Modern DRAM architectures," Ph.D. dissertation, 2001.

[9] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO*, 2007.

[10] M. Kandemir, H. Zhao, X. Tang, and M. Karakoy, "Memory Row Reuse Distance and Its Role in Optimizing Application Performance," in *SIGMETRICS*, 2015.

[11] X. Tang, M. Kandemir, P. Yedlapalli, and J. Kotra, "Improving Bank-Level Parallelism for Irregular Applications," in *MICRO*, 2016.

[12] O. Kislal, J. Kotra, X. Tang, M. Taylan Kandemir, and M. Jung, "POSTER: Location-Aware Computation Mapping for Manycore Processors." in *PACT*, 2017.

[13] J. B. Kotra, N. Shahidi, Z. A. Chishti, and M. T. Kandemir, "Hardware-Software Co-design to Mitigate DRAM Refresh Overheads: A Case for Refresh-Aware Process Scheduling," in *ASPLOS*, 2017.

[14] J. B. Kotra, M. Arjomand, D. Guttman, M. T. Kandemir, and C. R. Das, "Re-NUCA: A Practical NUCA Architecture for ReRAM Based Last-Level Caches," in *IPDPS*, 2016.

[15] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the problems and opportunities posed by multiple on-chip memory controllers," in *PACT*, 2010.

[16] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities," in *PACT*, 2016.

[17] O. Kayiran, A. Jog, A. Pattnaik, R. Ausavarungnirun, X. Tang, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "uC-States: Fine-grained GPU Datapath Power Management," in *PACT*, 2016.

[18] V. Adhinarayanan, I. Paul, J. L. Greathouse, W. Huang, A. Pattnaik, and W. c. Feng, "Measuring and modeling on-chip interconnect power on real hardware," in *IISWC*, 2016.

[19] A. Jog, O. Kayiran, A. Pattnaik, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Exploiting Core Criticality for Enhanced Performance in GPUs," in *SIGMETRICS*, 2016.

[20] "JEDEC. DDR3 SDRAM Standard, 2009."

[21] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *MICRO*, 2011.

[22] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, 2005.

[23] "Micron, calculating memory system power for DDR3, july 2007."

[24] "http://www.micron.com/products/dram/ddr3-sdram."

[25] A. Sharifi, A. K. Mishra, S. Srikantaiah, M. Kandemir, and C. R. Das, "Pepon: performance-aware hierarchical power budgeting for noc based multicores," in *PACT*, 2012.

[26] C. Lefurgy, K. Rajamani, F. L. R. III, W. M. Felter, M. Kistler, and T. W. Keller, "Energy management for commercial servers," *IEEE Computer*, 2003.

[27] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, 2006.

[28] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *HPCA*, 2010.

[29] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: eliminating server idle power," in *ASPLOS*, 2009.

[30] V. Delaluz, M. T. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin, "Hardware and software techniques for controlling DRAM power modes," *IEEE Trans. Computers*, 2001.

[31] I. Hur and C. Lin, "A comprehensive approach to DRAM power management," in *HPCA*, 2008.

[32] M. S. Ware, K. Rajamani, M. S. Floyd, B. Brock, J. C. Rubio, F. L. R. III, and J. B. Carter, "Architecting for power management: The IBM POWER7^{TM} approach," in *HPCA*, 2010.

[33] G. Liu, H. An, W. Han, X. Li, T. Sun, W. Zhou, X. Wei, and X. Tang, "FlexBFS: A Parallelism-aware Implementation of Breadth-first Search on GPU," in *PPoPP*, 2012.

[34] S. Liu, B. Leung, A. Neckar, S. O. Memik, G. Memik, and N. Hardavellas, "Hardware/software techniques for DRAM thermal management," in *HPCA*, 2011.

[35] S. Liu, Y. Zhang, S. O. Memik, and G. Memik, "An approach for adaptive DRAM temperature and power management," *IEEE Trans. VLSI Syst*.

[36] M. Bi, R. Duan, and C. Gniady, "Delay-hiding energy management mechanisms for DRAM," in *HPCA*, 2010.

[37] K. T. Malladi, I. Shaeffer, L. Gopalakrishnan, D. Lo, B. C. Lee, and M. Horowitz, "Rethinking DRAM power modes for energy proportionality," in *MICRO*, 2012.

[38] X. Li, Z. Li, Y. Zhou, and S. Adve, "Performance directed energy management for main memory and disks," *Trans. Storage*, 2005.

[39] G. Yan, Y. Li, Y. Han, X. Li, M. Guo, and X. Liang, "Agileregulator: A hybrid voltage regulator scheme redeeming dark silicon for power efficiency in a multicore architecture," in *HPCA*, 2012.

[40] W. Ding, X. Tang, M. Kandemir, Y. Zhang, and E. Kultursay, "Optimizing Off-chip Accesses in Multicores," in *PLDI*, 2015.

[41] X. Tang, A. Pattnaik, H. Jiang, O. Kayiran, A. Jog, S. Pai, M. Ibrahim, M. T. Kandemir, and C. R. Das, "Controlled Kernel Launch for Dynamic Parallelism in GPUs," in *HPCA*, 2017.

[42] X. Li, R. Gupta, S. V. Adve, and Y. Zhou, "Cross-component energy management: Joint adaptation of processor and memory," *ACM Transactions on Architecture and Code Optimization*.

[43] M. Chen, X. Wang, and X. Li, "Coordinating processor and main memory for efficient server power control," in *Proceedings of the international conference on Supercomputing*, 2011.

[44] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Rethinking DRAM design and organization for energy-constrained multi-cores," in *ISCA*, 2010.

[45] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu, "Mini-rank: Adaptive DRAM architecture for improving memory power efficiency," in *MICRO*, 2008.

[46] D. Chasapis, M. Casas, M. Moretó, M. Schulz, E. Ayguadé, J. Labarta, and M. Valero, "Runtime-guided mitigation of manufacturing variability in power-constrained multi-socket numa nodes," in *ICS*, 2016.

[47] H. H. 0002, K. G. Shin, C. Lefurgy, and T. W. Keller, "Improving energy efficiency by making DRAM less randomly accessed," in *ISLPED*, 2005.

[48] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis, "Power aware page allocation," in *ASPLOS*, 2000.