MultiScale: Memory System DVFS with Multiple Memory Controllers

Qingyuan Deng David Meisner[†] Abhishek Bhattacharjee
Thomas F. Wenisch[‡] Ricardo Bianchini

Rutgers University {qdeng,abhib,ricardob}@cs.rutgers.edu
[†]Facebook Inc. meisner@fb.com
[‡]University of Michigan twenisch@umich.edu

ABSTRACT

The fraction of server energy consumed by the memory system has been increasing rapidly and is now on par with that consumed by processors. Recent work demonstrates that substantial memory energy can be saved with only a small, tightly-controlled performance degradation using *memory* Dynamic Frequency and Voltage Scaling (DVFS). Prior studies consider only servers with a single memory controller (MC); however, multicore server processors have begun to incorporate multiple MCs. We propose MultiScale, the first technique to coordinate DVFS across multiple MCs, memory channels, and memory devices. Under operating system control, MultiScale monitors application bandwidth requirements across MCs. It then uses a heuristic algorithm to select and apply a frequency combination that will minimize the overall *system* energy within user-specified per-application performance constraints. Our results demonstrate that MultiScale reduces system energy consumption significantly, compared to prior approaches, while respecting the user-specified performance constraints.

Categories and Subject Descriptors

C.5 [**Computer System Implementation**]: Miscellaneous

Keywords

Memory system, energy conservation, dynamic voltage and frequency scaling

1. INTRODUCTION

Processors have historically dominated server power consumption. However, main memory power has been growing substantially as multicore servers require increasing memory capacity and bandwidth [4, 18, 21, 31]. Today, main memory accounts for roughly 40% of server energy [31], which is comparable to or slightly higher than processor energy.

Most prior work on memory energy conservation creates memory idleness through scheduling, batching, and layout transformations, so that *idle* low-power modes can be exploited [9, 11, 12, 14, 17, 26]. Techniques that reduce the number of DRAM chips accessed together [2, 33] or change DRAM chip microarchitecture [7, 30] have also been considered. Unfortunately, these prior techniques either provide limited benefit for DDR* memories or require changes to DIMMs (Dual In-line Memory Modules) and/or DRAM chip microarchitecture.

In contrast, *active* low-power modes for main memory have shown great promise in trading memory performance for energy savings [8, 10]. In these techniques, the hardware and the operating system (OS) collaborate to assess the benefits of scaling the voltage and frequency of the MC, and scaling the frequency of the memory channels and devices. The ultimate goal is to provide maximal energy savings under a user-specified performance degradation constraint.

These past memory DVFS proposals select a single performance setting for the memory system and are thus ideal for systems with a single MC. However, chip multiprocessors (CMPs) are increasingly integrating multiple on-die MCs [1, 3, 32]. Furthermore, recent work has demonstrated the benefit of *deliberately* skewing traffic across multiple MCs to preserve fair performance among applications judged likely to interfere (i.e., by placing data such that memory-intensive and non-memory intensive applications access disjoint MCs/channels) [25]. Such asymmetric traffic patterns will call for correspondingly asymmetric DVFS control.

Recent hardware trends also suggest that traffic skew across MCs will grow. For example, as servers increasingly rely on multi-socket configurations, inter-socket MC bandwidth requirements will vary significantly [31]. In addition, the advent of heterogeneous processors incorporating sophisticated superscalar out-of-order cores with simpler in-order cores (e.g., ARM's big.LITTLE architecture [13]), and graphics processing units (e.g., AMD's Fusion and Intel's Sandybridge architectures [28]), will fundamentally increase traffic skew across MCs. Therefore, it is critical to explore novel multi-MC active low-power mode management techniques. The straight-forward extension of prior work—selecting the same frequency for all MCs based on their average bandwidth requirement—will lead to sub-optimal savings under skewed traffic.

Thus, this paper presents MultiScale, a set of software policies and hardware mechanisms for coordinating DVFS across multiple MCs, channels, and devices. Under OS control, MultiScale monitors per-application traffic across MCs and estimates their varying bandwidth and latency requirements. It then uses a heuristic algorithm to quickly select and apply an optimized MC frequency combination. Like prior work on memory active low-power modes [10], MultiScale's goal is to minimize the overall system energy, without degrading performance beyond a user-specified limit. Unlike past work however, MultiScale is able to do so effectively and consistently for multi-MC systems under a variety of traffic skews.

ISLPED'12, July 30–August 1, 2012, Redondo Beach, CA, USA.
Copyright 2012 ACM 978-1-4503-1249-3/12/07 ...$10.00.

We evaluate MultiScale using detailed simulation on a diverse set of workload mixes constructed from the SPEC benchmark suite [29]. We quantify MultiScale's benefits across a range of traffic-skew patterns, showcasing its consistently higher energy efficiency versus past single voltage/frequency approaches [10].

This work is the first to study techniques to apply memory system active low-power modes to multiple MCs in a coordinated manner. First, we develop a set of low-overhead, yet effective software policies and hardware mechanisms that monitor per-application, per-MC bandwidth/latency requirements. Our readily-implementable performance counters, allied with low-overhead OS support, can be used to realize MultiScale's performance and energy models. Second, we quantify MultiScale's ability to exploit user-defined per-application performance degradation constraints across a range of traffic skew patterns. Our results show that MultiScale's ability to coordinate per-MC DVFS based on the workloads' dynamic memory bandwidth requirements allows it to achieve up to 4.5× greater energy savings than prior approaches. MultiScale is effective even in situations when performance constraints are tight; for example, when the allowable degradation is capped at just 1%, MultiScale can still achieve energy savings over 9% whereas past work achieves savings of merely 2%.

2. BACKGROUND AND RELATED WORK

We summarize DRAM operation and architecture; more details can be found in [23, 24]. In single-MC CMPs, a single controller services memory requests to all addresses, which are interleaved over one or more memory channels. Each channel has its own data and address bus and can be independently accessed. A channel connects to one or more DIMMs, each of which usually includes 16 or 18 DRAM chips. Each memory access targets a rank, a set of chips that collectively respond to the access. Each rank comprises multiple banks, each of which is a two-dimensional DRAM cell array. A channel's banks can be accessed in parallel, though they share the channel's address and data buses.

Several recent commercial processors integrate MCs on-chip. For example, Nehalem [15] integrates four cores and one MC with three channels to DDR3 memory. However, MC complexity and resource contention limit the number of channels a single controller can effectively manage. As systems accommodate higher core counts, multiple on-chip MCs will become the norm. For example, Power7 uses two MCs connected to eight channels [31], whereas Tile64 uses four MCs to service 64 cores [32].

2.1 Memory Power Management

Increasing memory capacity demands and growth in the fraction of overall system power attributable to memory [4, 6, 21, 31] have motivated studies of memory low-power modes. Numerous studies investigated the use of idle low-power modes (e.g., precharge powerdown, self-refresh) [9, 11, 12, 14, 17, 26]. However, past work has shown that active low-power modes are generally more successful at saving energy under performance constraints, while being readily implementable [8, 10]. These techniques observe that while server workloads are often highly sensitive to memory access latency, only rarely do they demand peak memory bandwidth [22]. In our prior work [10], we used this insight to propose MemScale, an energy management technique for the memory system. MemScale leverages dynamic profiling, performance and power modeling, DVFS of the MC, and DFS of the memory channels and DRAM devices to realize significant energy savings under user-specified performance degradation constraints.

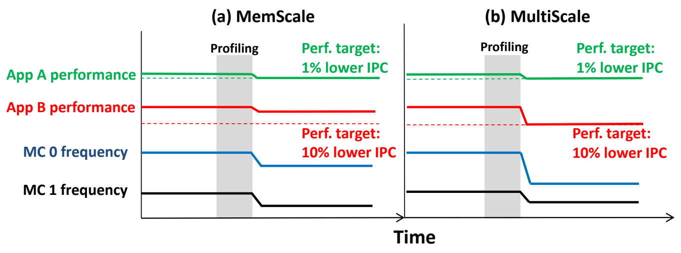

Figure 1: Because it independently manages each MC, MultiScale can select a lower frequency for MC 0 and thus save more energy than MemScale while remaining within the prescribed performance bounds.

2.2 Cross-MC Traffic Skew

MemScale selects a single performance setting for the memory system based on aggregate application requirements. As such, it is unable to exploit time-varying or asymmetric traffic patterns across MCs in multiple-MC systems. Figure 1 illustrates a scenario where two applications, A and B, run on a dual-MC system. Suppose the memory allocation is skewed such that 80% of application A's accesses are directed to MC 1, whereas application B's accesses are predominantly to MC 0. Further, suppose that application A is under a tight performance constraint, and can tolerate only a 1% degradation, whereas application B can tolerate up to 10% slowdown. Application A's tight performance constraint will require MC 1 at a high frequency. However, there might be substantial opportunity to slow MC 2 without violating the performance constraint of either application. MemScale (Figure 1 left) selects only a single frequency for both MCs, based on the tighter performance constraint of application A, limiting energy savings. In contrast, MultiScale (Figure 1 right) independently assesses the bandwidth requirements of each MC and selects an appropriate frequency/voltage to maximize energy savings while respecting each applications' degradation constraint.

While the OS maps virtual to physical addresses, the interleaving of physical addresses has the greatest impact on the traffic skew across MCs. Cache line and page interleaving seek to distribute traffic evenly. While such interleaving balances traffic across MCs, there are many strong arguments and hardware/software trends pointing toward addressing schemes that skew traffic across channels. For example, recent work by Muralidhara and co-authors has demonstrated the benefit of deliberately skewing traffic across multiple MCs to preserve fair performance among applications judged likely to interfere [25]. Under this scheme, data from memory-intensive and non-memory-intensive applications are mapped to disjoint MCs/channels. This isolates non-memory-intensive workloads from interference effects of intense memory traffic, while clever scheduling techniques manage access among the memory-intensive workloads. Similarly, Awasthi and co-authors propose mechanisms to place data in multi-MC systems to improve performance [3]. Their approach, which balances the benefits of allocating application data to the MC closest to the corresponding core against queuing delays, on-chip latencies, and row buffer hit rates, often results in traffic skew. Traffic skew is also common in heterogeneous platforms with accelerators, such as graphics processing units and sophisticated out-of-order cores coupled with simpler in-order cores, which might impose differing bandwidth requirements across MCs.

MultiScale adapts better than MemScale to traffic skew across MCs. So, in characterizing MultiScale, we have two primary goals: (1) We demonstrate its effectiveness across a range of traffic patterns. To this end, we quantify

MultiScale's operation across a spectrum of traffic skews. (2) We aim to showcase MultiScale's effectiveness at realizing energy savings even under the tightest performance degradation constraints. To this end, we study the benefits of MultiScale across a range of degradation constraints, as small as 1% allowable slowdown.

3. MULTISCALE DESIGN

MultiScale seeks to maximize energy savings while adhering to per-application user-specified performance degradation constraints. We now detail the hardware mechanisms and software policies that make this possible.

3.1 Hardware and Software

Hardware mechanisms. For each MC, MemScale adjusts its frequency and voltage and the frequency of its associated memory channels and DIMMs; for expediency, we shall refer to these operations collectively as "adjusting the MC frequency". The DIMM clocks lock to the bus frequency (or a multiple thereof), while the MC frequency is fixed at double the bus frequency. While MultiScale can readily be applied to systems where each MC controls multiple channels at different frequencies, assessing the hardware overheads of such an approach is beyond the scope of this work. We therefore treat each MC, together with all its memory channels, as the unit for frequency selection. A frequency change requires the system to briefly suspend operation and reconfigure to run at the new target [10]. Our experiments model the associated recalibration delays.

Similar to [10], MultiScale frequency scales the MCs, buses, and DIMMs. Voltage scaling is restricted to the MCs, and is set according to the selected frequencies.

Performance counter monitoring. Our management policies require input from a set of performance counters implemented on each core and on-chip MC. Specifically, we require counters tracking the amount of work pending at each MC's memory banks and channels. Counters similar to those we require already exist in most modern architectures, and are accessible through the CPU's performance-monitoring interface. For further details on the exact performance counters used, we refer the reader to [10] as MultiScale uses identical counters.

Energy management policy. Our goal is to minimize overall system energy consumption without degrading performance beyond user-specified bounds. Therefore, as in previous schemes [10, 20, 26], MultiScale exploits the notion of performance *slack*: the difference between a baseline execution and a target slowdown that the each application may incur to save energy. Our control algorithm exploits this allowable slack to reduce memory system performance and save energy. The per-application performance target is defined such that the application incurs no more than a pre-selected maximum slowdown relative to its execution without energy management (i.e., at maximum frequency). Formally, the slack is the difference in time of the program's execution (T_{Actual}) from the target (T_{Target}).

$$\begin{aligned} \text{Slack} &= T_{\text{Target}} - T_{\text{Actual}} \\ &= T_{\text{MaxFreq}} \cdot (1+\gamma) - T_{\text{Actual}} \end{aligned} \tag{1}$$

where γ defines the target maximal execution time increase.

In exploiting this slack, MultiScale's control algorithm divides execution into fixed-sized epochs. We typically associate an epoch with an OS time quantum. MultiScale splits each epoch into four distinct phases. First, applications are profiled by collecting statistics from the performance counters. We find that profiling for 300 μs in an epoch of 5 ms suffices. Second, the OS uses the profiling information to select new MC frequencies (as detailed in the next subsection). Third, each MC, its channels, and DRAM devices are transitioned to their new frequency. Finally, the epoch completes at this new frequency configuration. At the end of the epoch, we again query the counters and estimate the performance that would have been achieved had the memory system operated at maximum frequency. The difference between this estimate and the achieved performance is used to update the slack and is carried forward to calculate the target performance in the next epoch.

3.2 Performance and Energy Models

Performance model. Our control algorithm utilizes a performance model extended from [10] to account for per-MC frequencies and the traffic directed by each application to each MC. The performance model predicts the relationship between CPU *cycles per instruction* (CPI) of an application, and the per-MC frequency. There are three steps in our modeling approach. First, we estimate the memory-boundedness of each running application, and thus estimate the target access latency that satisfies the performance target of each application. Second, we estimate each MC's contribution to this average latency (which requires the MCs to be aware of the hardware thread that issued each access). Finally, based on the first two steps, we calculate per-MC frequencies. Next, we detail each step.

Step 1: Under our performance model, the runtime of a program is defined as: $t_{\text{total}} = t_{\text{CPU}} + t_{\text{Mem}} = I_{\text{CPU}} \cdot E[TPI_{\text{CPU}}] + I_{\text{Mem}} \cdot E[L_{\text{Mem}}]$, where I_{CPU} represents the number of instructions, and I_{mem} is the number of last-level cache (LLC) misses stalling the pipeline. TPI_{CPU} represents the average time that instructions spend on the CPU (including L1 cache hits, L1 cache misses, and L2 cache hits in a two-level cache hierarchy), and L_{Mem} is the average memory latency of each application.

Since runtime is not known a priori, we model the rate of progress of an application in terms of CPI. The average CPI of a program is defined as: $E[\text{CPI}] = (E[TPI_{\text{CPU}}] + \alpha \cdot E[L_{\text{Mem}}]) \cdot F_{\text{CPU}}$, where α is the fraction of instructions that miss in the L2 cache and stall the pipeline, and F_{CPU} is the operating frequency of the core. The value of α can easily be calculated as the ratio of instruction to LLC miss counts, accessible through performance counters. Given a target CPI, we can compute $E[L_{\text{Mem}}]$ assuming other components in the above equation are constant. By substituting α into the equation and dividing by the frequency, we can compute the target average per-application latency needed to compute per-MC latency.

Step 2: Having calculated the target average memory access latency per application, we now focus on the latency breakdown per MC. To understand this, consider a simple scenario where there are two applications, A and B, and two MCs, MC 0 and MC 1. Suppose that for A, $Perc_{A0}$% of total memory accesses go to channels under MC 0, while $Perc_{A1}$% go to channels under MC 1. Further, assume that $Perc_{B0}$% of accesses from B go to channels under MC 0, and $Perc_{B1}$% of accesses go to channels under MC 1. We have:

$$\begin{cases} Perc_{\text{A0}}\% + Perc_{\text{A1}}\% = 100\% \\ Perc_{\text{B0}}\% + Perc_{\text{B1}}\% = 100\% \end{cases} \tag{2}$$

Assume that the average access latency to channels under MC 0 is L_0, and to channels under MC 1 is L_1. Furthermore, denote the average memory access latency of A and B is $E[L_A]$ and $E[L_B]$, respectively. We then have:

$$\begin{cases} E[L_{\text{A}}] = Perc_{\text{A0}} \cdot L_0 + Perc_{\text{A1}} \cdot L_1 \\ E[L_{\text{B}}] = Perc_{\text{B0}} \cdot L_0 + Perc_{\text{B1}} \cdot L_1 \end{cases} \tag{3}$$

We can now use these equalities to calculate per-MC memory access latencies. Using the information from step (1) on each application's $E[L_{\mathrm{Mem}}]$, we can cap each application's latencies so as to ensure the correct performance targets. Specifically, assuming that $L_{\mathrm{Target\_A}}$ and $L_{\mathrm{Target\_B}}$ are the threshold latencies that guarantee the performance targets of A and B, we have the following inequalities.

$$\begin{cases} E[L_{\mathrm{A}}] \leq L_{\mathrm{Target\_A}} \\ E[L_{\mathrm{B}}] \leq L_{\mathrm{Target\_B}} \end{cases} \tag{4}$$

Solving this system provides L_0 and L_1 values which can then be used as input to our next step.

Although this simple example assumes two applications and two MCs, this approach can be generalized to any number of applications and MCs. In general, this entails solving a linear programming (LP) problem where the number of MCs is likely smaller than the number of running applications. (MultiScale only needs to deal with the applications running during the next epoch.) Standard LP solvers can be used to calculate these latencies efficiently. The overhead of this computation is negligible for realistic numbers of MCs, since it only occurs once per epoch. For our setup (4 MCs and 16 cores), the overhead is less than 50 μs on a Xeon 5520 machine.

Step 3: Having solved for L_0 and L_1 and the latencies of all other MCs (which we collectively denote as L_{Mem}), we model the relationship between channel frequency and memory access latency. We take the approach of [10]: $E[L_{\mathrm{Mem}}] = \xi_{\mathrm{bank}} \cdot (S_{\mathrm{Bank}} + \xi_{\mathrm{bus}} \cdot S_{\mathrm{Bus}})$, where ξ_{bus} represents the average number of requests waiting for the bus and is approximated by the counters capturing the queuing impact of waiting for bus transfers; ξ_{bank} represents the average number of requests waiting for the bank and is approximated by the counters capturing per bank queuing; S_{Bank} is the average time, excluding queueing delays, to access a bank (including precharge, row access and column read, etc); and S_{Bus} is the average data transfer (burst) time across the bus. Since the values of S_{Bank}, ξ_{bank}, and ξ_{bus} can be obtained by profiling performance counters, we can calculate S_{Bus}, which is a function of the frequency. Finally, we can calculate the target frequency from S_{Bus}.

Full-system energy model. Simply meeting the CPI loss target for a given workload does not necessarily maximize energy efficiency. In other words, though additional performance degradation may be allowed, it may save more energy to run faster. To determine the best operating point, we construct a model to predict full-system energy usage. For memory frequency $f_{\mathrm{MC\_1}}, f_{\mathrm{MC\_2}}, \ldots, f_{\mathrm{MC\_N}}$, we define the *system energy ratio* (SER) as:

$$\mathrm{SER}(f_{\mathrm{mem}}) = \frac{T_{f_{\mathrm{Mem}}} \cdot (\sum_i P_{f_{\mathrm{MC\_i}}} + P_{NonMem})}{T_{\mathrm{Base}} \cdot P_{\mathrm{Base}}} \tag{5}$$

$T_{f_{\mathrm{Mem}}}$ is the performance estimate for an epoch at frequency $f_{\mathrm{MC\_1}}$ through $f_{\mathrm{MC\_N}}$. Memory power is calculated with the memory power model in [24], and P_{NonMem} accounts for all non-memory system components and is assumed to be fixed. T_{Base} and P_{Base} are corresponding values at a nominal frequency. At the end of each epoch's profiling phase, we calculate SER for all memory frequencies that can meet the performance constraint given by the slack, and select the frequency that minimizes the SER, from the range calculated by the latency model as described earlier in this section.

4. EVALUATION

In this section, we demonstrate the efficacy of MultiScale over a range of traffic skews relative to MemScale.

Table 1: Workloads.

Name	MPKI	WPKI	Applications (x4 each)			
MIX1	2.93	2.56	applu	hmmer	gap	gzip
MIX2	2.34	0.39	milc	gobmk	facerec	perlbmk
MIX3	2.55	0.80	equake	ammp	sjeng	crafty
MIX4	2.41	1.41	lucas	vpr	h264ref	eon
MIX5	2.35	1.38	swim	ammp	twolf	sixtrack
MIX6	2.91	1.57	libquantum	twolf	vpr	sjeng
MIX7	3.12	1.48	mcf	astar	gzip	sixtrack
MIX8	1.83	0.77	mgrid	fma3d	crafty	eon

Table 2: Simulation parameters.

Feature	Value
CPU cores	16 in-order, single thread, 4GHz
	Single IALU IMul FALU FMulDiv
L1 I/D cache (per core)	64KB, 2-way, 1 CPU cycle hit
L2 cache (shared)	16MB, 16-way, 10 CPU cycle hit
Memory configuration	4 MCs, 1 channel/MC, 8 2GB DIMMs
tRCD/tRP/tCL	15ns, 15ns, 15ns
tFAW/tRTP/tRAS/tRRD	20/5/28/4 cycles
tXP/tXPDLL/Refresh	6ns/24ns/64ms
Row buffer read, write	250mA, 250mA
Activation-precharge	120 mA
Active standby	67 mA
Active powerdown	45 mA
Precharge standby	70 mA
Precharge powerdown	45 mA
Refresh	240 mA
VDD	1.575 V

4.1 Methodology

Simulator and workloads. Our evaluations are based on a two-step simulation methodology. First, we use M5 [5] to collect memory access traces (consisting of L1 cache misses and writebacks), and per-core activity traces. Second, we feed these traces into a detailed simulator modeling a 16-core CMP with a shared L2 cache (LLC), on-chip MCs, memory channels, and DRAM devices. We also feed core activity traces, with the run-time statistics from the L2 module, into McPAT [19] to dynamically estimate the CPU power. Overall, we simulate in detail all aspects of core microarchitecture, caches, memory controller and memory devices relevant to our study, including memory device power and timing, and row buffer management.

Table 1 lists the main characteristics of our 8 workloads. The workloads are formed by combining applications from the SPEC 2000 and SPEC 2006 suites. We analyze the best 100M-instruction simulation point for each application (selected using Simpoints 3.0 [27]). The workload terminates when the slowest application has executed 100M instructions. We report the LLC misses per kilo instruction (MPKI) and LLC writebacks per kilo instruction (WPKI) of each workload in Table 1.

Simulation parameters. Table 2 details our baseline memory system with 4 MCs, each connected to one DDR3 channel. Each channel is populated with two registered, dual-ranked DIMMs with 18 DRAM chips each. We further assume the 16 cores are interconnected using as 4x4 mesh with the MCs on the corners. Every set of 4 cores has a *local* MC, which is at the adjacent corner.

Table 2 also shows the timing and power parameters of the DRAM chips, register, PLL, and MC [24], assuming a frequency of 800 MHz. We also consider frequencies of 733, 667, 600, 533, 467, 400, 333, 267, and 200 MHz. Frequency transitions take 512 memory cycles plus 28ns [16]. MultiScale further assumes that the MC frequency is double the channel frequency, which in turn is double the DRAM device frequency. The voltage range of each MC is the same as that of the cores (0.65V-1.2V). Each MC can be set at a different voltage. Each MC's power ranges from 2W to 4W, depending on the voltage, frequency, and utilization. Based on these values, our memory system (including MCs) accounts for 44% of the total system power on average.

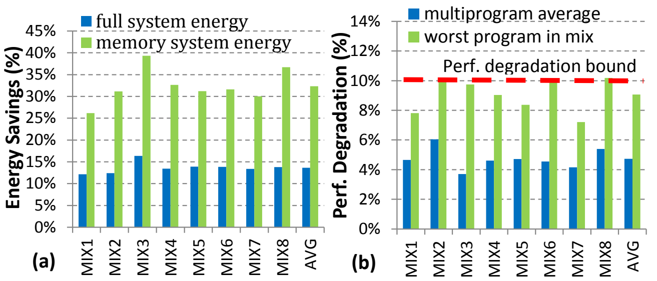

Figure 2: (a) MultiScale's energy savings assuming that 80% of an application's pages are mapped to its local MC and a 10% performance loss bound; (b) MultiScale's actual performance loss in this scenario.

Experiments. We compare MultiScale to MemScale for different traffic distributions and performance degradation bounds. We distribute traffic by controlling how many pages of each application are allocated to its local MC. In our experiments, we vary this from 100% (the extreme form of channel partitioning proposed in [25]) to 80%, 60%, 40%, 25%, and 20%. Non-local MC pages are allocated randomly across the remote MCs. Thus, the 25% case represents the scenario in which each MC is responsible for (roughly) the same number of pages.

For each of these cases, we investigate 1%, 3%, 5%, 7%, and 10% allowable performance slowdowns.

4.2 Results

Figure 2(a) shows MultiScale's memory and full-system energy savings across the workload mixes, assuming maximum allowable performance degradations of 10%, and a distribution where 80% of an application's pages are allocated to its local MC. The figure indicates that MultiScale is successful in saving energy across all workload mixes under skewed traffic. Although the exact savings vary across the workloads, on average, MultiScale saves 13% of the baseline *full-system* energy.

MultiScale's energy savings do not come at the cost of excessive performance degradation. Figure 2(b) shows the average and worst-case performance degradation across all applications in a workload. The results indicate that MultiScale saves energy without exceeding the 10% performance loss constraint across all workload mixes.

Figure 3 depicts the MultiScale and MemScale energy savings across the entire spectrum of performance loss bounds and page allocation schemes. The horizontal axis plots the page allocation scheme, whereas the vertical axis captures the full-system energy savings of each approach. For every allocation scheme and approach, there are five bars, each illustrating the full-system energy savings for a different performance loss bound.

From this figure, we make the following observations. First, MultiScale saves energy across *every* considered allocation and performance bound scenario. The exact energy savings depend upon the amount of traffic skew across MCs and the performance bound. As expected, greater skew and performance bounds allow MultiScale to save more energy. For example, with 100% local page allocation and 10% performance bound, MultiScale saves over 14% of full-system energy. In contrast, with 40% of pages allocated to the local MC and a 5% bound, the energy savings are 8%.

Second, MultiScale conserves at least as much energy as MemScale on *every* considered scenario. MultiScale's advantage increases with greater traffic skew across MCs. This is expected since MultiScale is better able to detect the traffic pressure on each MC and adjust each MC to an appropriate frequency. Interestingly, Figure 3 also shows that MultiScale outperforms MemScale substantially when the performance bounds are low. For example, at a 1% performance bound and 100% local MC allocation, MultiScale can still achieve energy savings of over 9%, whereas MemScale manages merely 2%. The reason is that MultiScale provides finer-grained control of the required memory system performance for a given slack; as such, it is easier for MultiScale to exploit any available slack.

Finally, we consider the performance loss that MultiScale and MemScale incur across the same spectrum of page allocations and loss bounds. Figure 4 depicts these data, showing the performance loss of the *most* degraded application. The figure demonstrates that MultiScale degrades the performance of the worst-hit application slightly more than MemScale. These differences are most pronounced (but still lower than 2%) with lower traffic skew and higher performance bounds. In fact, MultiScale slightly violates the bound in a few cases, but always by less than 0.23%. These slight violations occur because MultiScale pushes *all* applications to the edge of their performance bounds, whereas MemScale pushes only the application that is most sensitive to memory performance to its bound. Thus, MultiScale is more prone to (slight) violations.

5. CONCLUSION

We proposed MultiScale, a set of hardware mechanisms and software policies for using active low-power modes to manage multiple MCs in a coordinated fashion and under performance constraints. MultiScale yields greater energy savings than the best previous approach, MemScale, by gauging the traffic requirements of each MC and setting it to the appropriate DVFS level. As our results demonstrate, MultiScale is particularly effective in scenarios where traffic is skewed across MCs and when the allowable performance degradation is low. As such, MultiScale is an ideal energy management approach for multi-MC systems with scheduling and allocation policies that promote traffic skew.

6. ACKNOWLEDGEMENTS

This research was partially supported by Google and the National Science Foundation under grants #CCF-0916539, #CSR-0834403, and #CCF-0811320.

7. REFERENCES

[1] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. Lipasti. Achieving Predictable Performance Through Better Memory Controller Placement in Many-Core CMPs. In *ISCA*, 2009.

[2] J. H. Ahn, J. Leverich, R. Schreiber, and N. P. Jouppi. Multicore DIMM: An Energy Efficient Memory Module with Independently Controlled DRAMs. In *IEEE Computer Architecture Letters*, 2009.

[3] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the Problems and Opportunities Posed by Multiple On-Chip Memory Controllers. In *PACT*, 2010.

[4] L. A. Barroso and U. Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, Jan. 2009.

[5] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The M5 Simulator: Modeling Networked Systems. In *IEEE Micro*, 2006.

[6] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-Aware Scheduling on Multicore Systems. In *ACM Trans. Comput. Syst.*, 2010.

[7] E. Cooper-Balis and B. Jacob. Fine-grained Activation for Power Reduction in DRAM. In *IEEE Micro*, 2010.

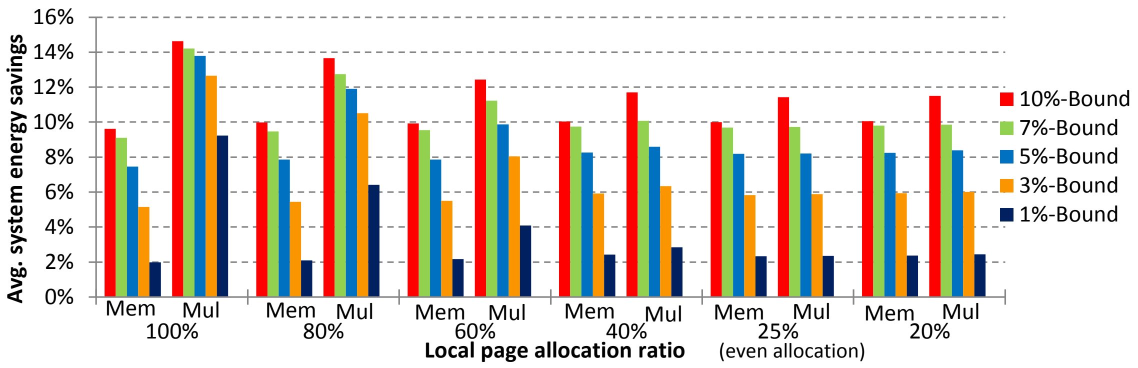

Figure 3: MultiScale's energy savings versus MemScale across a spectrum of traffic skews and performance degradation bounds. MultiScale consistently provides greater energy savings than MemScale.

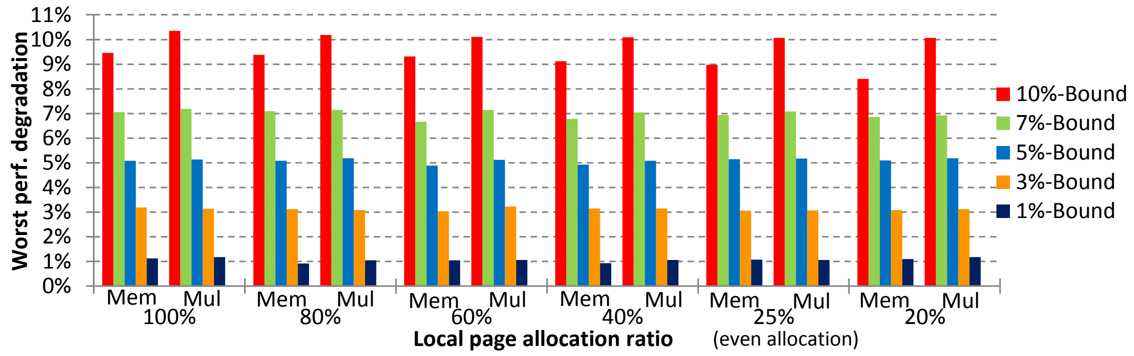

Figure 4: MultiScale's worst performance degradation versus MemScale across a spectrum of traffic skews and performance degradation bounds. MultiScale leads to slightly higher degradations than MemScale.

[8] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory Power Management via Dynamic Voltage/Frequency Scaling. In *ICAC*, 2011.

[9] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. Hardware and Software Techniques for Controlling DRAM Power Modes. In *IEEE Transactions on Computers*, 2001.

[10] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. MemScale: Active Low-Power Modes for Main Memory. In *ASPLOS*, 2011.

[11] B. Diniz, D. Guedes, W. Meira Jr, and R. Bianchini. Limiting the Power Consumption of Main Memory. In *ISCA*, 2007.

[12] X. Fan, C. Ellis, and A. Lebeck. Memory Controller Policies for DRAM Power Management. In *ISLPED*, 2001.

[13] P. Greenhalgh. big.LITTLE Processing with the Cortex-A15 and Cortex-A7 Processors, 2011.

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

[15] Intel. Intel Xeon processor 5600 Series, 2010.

[16] JEDEC. DDR3 SDRAM Standard, 2009.

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

[18] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy Management for Commercial Servers. *IEEE Computer*, 36(12), 2003.

[19] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Micro*, 2009.

[20] X. Li, Z. Li, F. M. David, P. Zhou, Y. Zhou, S. V. Adve, and S. Kumar. Performance-Directed Energy Management for Main Memory and Disks. In *ASPLOS*, 2004.

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

[22] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Webber, and T. F. Wenisch. Power Management of Online Data-Intensive Services. In *ASPLOS*, 2011.

[23] Micron. 1Gb: x4, x8, x16 DDR3 SDRAM, 2006.

[24] Micron. Calculating Memory System Power for DDR3, July 2007.

[25] 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.

[26] V. Pandey, W. Jiang, Y. Zhou, and R. Bianchini. DMA-Aware Memory Energy Management. In *HPCA*, 2006.

[27] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for Accurate and Efficient Simulation. In *SIGMETRICS*, 2003.

[28] S. Sawant, U. Desai, G. Shamanna, L. Sharma, M. Ranade, A. Agarwal, S. Dakshinamurthy, and R. Narayanan. A 32nm Westmere-EX Xeon Enterprise Processor. In *ISSCC*, 2011.

[29] Standard Performance Evaluation Corporation. SPEC CPU 2006.

[30] A. N. Udipi, N. Muralimanohar, N. Chatterjee, Rajeev Balasubramonian, A. Davis, and N. P. Jouppi. Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores. In *ISCA*, 2010.

[31] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. Rubio, F. Rawson, and J. Carter. Architecting for Power Management: The IBM POWER7 Approach. In *HPCA*, 2010.

[32] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 2007.

[33] 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.