CoScale: Coordinating CPU and Memory System DVFS in Server Systems

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

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

Abstract

Recent work has introduced memory system dynamic voltage and frequency scaling (DVFS), and has suggested that balanced scaling of both CPU and the memory system is the most promising approach for conserving energy in server systems. In this paper, we first demonstrate that CPU and memory system DVFS often conflict when performed independently by separate controllers. In response, we propose CoScale, the first method for effectively coordinating these mechanisms under performance constraints. CoScale relies on execution profiling of each core via (existing and new) performance counters, and models of core and memory performance and power consumption. CoScale explores the set of possible frequency settings in such a way that it efficiently minimizes the full-system energy consumption within the performance bound. Our results demonstrate that, by effectively coordinating CPU and memory power management, CoScale conserves a significant amount of system energy compared to existing approaches, while consistently remaining within the prescribed performance bounds. The results also show that CoScale conserves almost as much system energy as an offline, idealized approach.

1. Introduction

The processor has historically consumed the bulk of system power in servers, leading to a rich array of processor power management techniques, e.g. [16, 20, 37]. However, due to their success, and because of increasing memory capacity and bandwidth requirements in multicore servers, main memory energy consumption is increasing as a fraction of the total server energy [2, 24, 29, 39]. In response, many active and idle power management techniques have been proposed for main memory as well, e.g. [8, 10, 11, 12, 22, 34]. In light of these trends, servers are likely to provide separate power management capabilities for individual system components, with distinct control policies and actuation mechanisms. Our ability to maximize energy efficiency will hinge on the coordinated use of these various capabilities [31].

Prior work on the coordination of CPU power and thermal management across servers, blades, and racks has demonstrated the difficulty of coordinated management and the potential pitfalls of independent control [36]. Existing studies seeking to coordinate CPU DVFS and memory low-power modes have focused on *idle* low-power memory states [6, 13, 27]. While effective, these works ignore the possibility of using DVFS for the memory subsystem, which has recently been shown to provide greater energy savings [10]. As such, the coordination of *active* low-power modes for processors and memory in tandem remains an open problem.

In this paper, we propose CoScale, the first method for effectively coordinating CPU and memory subsystem DVFS under performance constraints. As we show, simply supporting separate processor and memory energy management techniques is insufficient, as independent control policies often conflict, leading to oscillations, unstable behavior, or sub-optimal power/performance trade-offs.

To see an example of such behavior, consider a scenario in which a chip multiprocessor's cores are stalled waiting for memory a significant fraction of the time. In this situation, the CPU power manager might predict that lowering voltage/frequency will improve energy efficiency while still keeping performance within a pre-selected performance degradation bound and effect the change. The lower core frequency would reduce traffic to the memory subsystem, which in turn could cause its (independent) power manager to lower the memory frequency. After this latter frequency change, the performance of the server as a whole may dip below the CPU power manager's projections, potentially violating the target performance bound. So, at its next opportunity, the CPU manager might start increasing the core frequency, inducing a similar response from the memory subsystem manager. Such oscillations waste energy. These unintended behaviors suggest that it is essential to coordinate power-performance management techniques across system components to ensure that the system is balanced to yield maximal energy savings.

To accomplish this coordinated control, we rely on execution profiling of core and memory access performance, using existing and new performance counters. Through counter readings and analytic models of core and memory performance and power consumption, we assess opportunities for per-core voltage and frequency scaling in a chip multiprocessor (CMP), voltage and frequency scaling of the on-chip memory controller (MC), and frequency scaling of memory channels and DRAM devices.

The fundamental innovation of CoScale is the way it efficiently searches the space of per-core and memory frequency settings (we set voltages according to the selected frequencies) in software. Essentially, our epoch-based policy estimates, via our performance counters and online models, the energy and performance cost/benefit of altering each component's (or set of components') DVFS state by one step, and iterates to greedily select a new frequency combination for cores and memory. The selected combination trades off core and memory scaling to minimize full-system energy while respecting a user-defined performance degradation bound. CoScale is implemented in the operating system (OS), so an epoch typically corresponds to an OS time quantum.

For comparison, we demonstrate the limitations of fully uncoordinated and semi-coordinated control (i.e., independent controllers that share a common estimate of target and achieved performance) of processor and memory DVFS. These strategies either violate the performance bound or oscillate wildly before settling into local minima. CoScale circumvents these problems by assessing processor and memory performance in tandem. In fact, CoScale provides energy savings close to an offline scheme that considers an exponential space of possible frequency combinations. We also quantify the benefits of CoScale versus CPU-only and memory-only DVFS policies.

Our results show that CoScale provides up to 24% *full-system* energy savings (16% on average) over a baseline scheme without DVFS, while staying within a 10% allowable performance degradation. Furthermore, we study CoScale's sensitivity to several parameters, including its effectiveness across performance bounds of 1%, 5%, 15%, and 20%. Our results demonstrate that CoScale meets the performance constraint while still saving energy in all cases.

2. Motivation and Related Work

Despite the advances in CPU power management, current servers remain non-energy-proportional, consuming a substantial fraction of peak power when completely idle [1]. To improve proportionality, researchers have recently proposed active low-power modes for main memory [7, 10]. CoScale takes a significant step in realizing effective server-wide power-performance tradeoffs using active low-power modes for both cores and memory. Next, we summarize some of the work on CPU and memory power management.

2.1. CPU Power Management

A large body of work has addressed the power consumption of CPUs. For example, studies have quantified the benefits of detecting periods of server idleness and rapidly transitioning cores into *idle low-power states* [30]. However, such states do not work well under moderate or high utilization. In contrast, processor *active low-power modes* provide better power-performance characteristics across a wide range of utilizations. Here, DVFS provides substantial power savings for small changes in voltage and frequency, in exchange for moderate performance loss. Processor DVFS is a well-studied technique [16, 20, 37] that is effective for a variety of workloads.

Processor DVFS techniques typically either rely on modeling or measurements (and feedback) to determine the next frequency to use. Invariably, these techniques assume that the memory subsystem will behave the same, regardless of the particular frequency chosen for the processor(s).

2.2. Memory Power Management

While CPUs have long been a focus of power optimizations, memory power management is now seeing renewed interest, e.g. [7, 9, 10, 38, 41]. As with processors, idle low-power states (e.g., precharge powerdown, self-refresh) have been extensively studied, e.g. [11, 22, 27, 28, 34]. However, past work has shown that active low-power modes are more successful at garnering energy savings for server workloads [9, 10, 31]. In particular, the memory bus is often underutilized for long periods, providing ample opportunities for memory power management.

To harness these opportunities, we recently proposed MemScale, a technique that leverages dynamic profiling, performance and power modeling, DVFS of the MC, and DFS of the memory channels and DRAM devices [10]. David *et al.* also studied memory DVFS [7]. In both these works, memory system scaling was done in the absence of core power management.

2.3. Integrated Approaches and CoScale

Researchers have only rarely considered coordinating management across components [6, 5, 13, 28, 36]. Raghavendra *et al.* considered how best to coordinate managers that operate at different granularities, but focused solely on processor power [36]. Much as we find, they showed that uncoordinated approaches can lead to destructive and unpredictable interactions among the managers' actions.

A few works have considered coordinated processor and memory power management for energy conservation [13, 27]. However, unlike these works, which assume only idle low-power states for memory, we concentrate on the more effective active low-power modes for memory (and processors). This difference is significant for two reasons: (1) Although the memory technology in these earlier studies (RDRAM) allowed per-memory-chip power management, modern technologies only allow management at a coarse grain (e.g., multi-chip memory ranks), complicating the use of idle low-power states; and (2) active memory low-power modes interact differently with the cores than idle memory low-power states. Moreover, these earlier works focused on single-core CPUs, which are easier to manage than CMPs. In a different vein, Chen *et al.* considered coordinated management of the processor and the memory for capping power consumption (rather than conserving energy), again assuming only idle low-power states [6]. Also assuming a power cap, Felter *et al.* proposed coordinated power shifting between the CPU and the memory by using a traffic throttling mechanism [14]. CoScale can be readily extended to cap power with appropriate changes to its decision algorithm and epoch length.

Perhaps the most similar work to CoScale is that of Li *et al.* [27], which also seeks to conserve CPU and memory energy subject to a performance bound. Their study investigates the combination of CPU microarchitectural adaptations (but could easily be extended to CPU DVFS) and memory idle low-power states, adapting the delay threshold before a memory device is transitioned to sleep. However, the study considers only a single-core CPU and a memory system with few low-power states. As such, their design is able to employ a policy that experimentally profiles each processor low-power configuration. The policy then profiles different combinations of processor and memory idle threshold configurations. It uses phase detection techniques and a history-based predictor to select the best state combination based on past measurements. Such a profiling-based approach is not viable for a large multicore with per-core and memory DVFS settings, due to the combinatorial explosion of possible states. Moreover, it is unclear how to extend their phase-based prediction for multi-programmed workloads; a proper configuration must be learned for each phase combination across all programs that may execute concurrently. CoScale's most fundamental advance is that it can optimize over a far larger combinatorial space. The large space is tractable because CoScale profiles performance at current settings and then uses simple models to predict power/performance at other settings.

3. CoScale

CoScale leverages three key mechanisms: core and memory subsystem DVFS, and a performance management scheme that keeps track of how much energy conservation has slowed down applications.

Core DVFS. We assume that each core can be voltage and frequency scaled independently of the other cores, as in [21, 40]. We also assume the shared L2 cache sits in a separate voltage domain that does not scale. A core DVFS transition takes a few 10's of microseconds.

Memory DVFS. Our memory DVFS method is based on MemScale [10], which dynamically adjusts MC, bus, and DIMM frequencies. Although it adjusts these frequencies together, we shall simply refer to adjusting the bus frequency. The DIMM clocks lock to the bus frequency (or a multiple thereof), while the MC frequency is fixed at double the bus frequency. Furthermore, MemScale adjusts the voltage

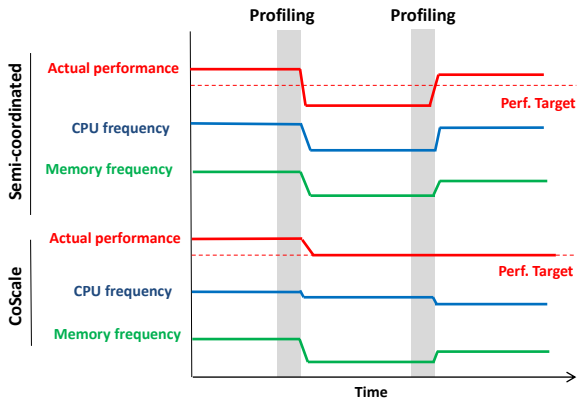

Figure 1: CoScale operation: Semi-coordinated oscillates, whereas CoScale scales frequencies more accurately.

of the MC (independently of core/cache voltage) and PLL/register in the DIMMs, based on the memory subsystem frequency.

Memory mode transition time is dominated by frequency re-calibration of the memory channels and DIMMs. The DIMM operating frequency may be reset while in the precharge powerdown or self-refresh state. We use precharge powerdown because its overhead is significantly lower than that of self-refresh. Most of the re-calibration latency is due to the DLL synchronization time, t_{DLLK} [32]—approximately 500 memory cycles.

Performance management. Similar to the approach initially proposed in [28] and later explored in [9, 10, 11, 34], our policy is based on the notion of program *slack*: the difference between a baseline execution and a target latency penalty that a system operator is willing to incur on a program to save energy. The basic idea is that energy management often necessitates running the target program with reduced core or memory subsystem performance. To constrain the impact of this performance loss, CoScale dictates that each executing program incurs no more than a pre-selected maximum slowdown γ, relative to its execution without energy management ($T_{MaxFreq}$). Thus, $Slack = T_{MaxFreq}(1+\gamma) - T_{Actual}$.

Overall operation. CoScale uses fixed-size epochs, typically matching an OS time quantum. Each epoch consists of a system profiling phase followed by the selection of core and memory subsystem frequencies that (1) minimize *full system* energy, while (2) maintaining performance within the target given by the accumulated slack from prior epochs.

In the system profiling phase, performance counters are read to construct application performance and energy estimates. By default, we profile for 300 μs, which we find to be sufficient to predict the resource requirements for the remainder of the epoch. Our default epoch length is 5 ms.

Based on the profiling phase, the OS selects and transitions to new core and/or memory bus frequencies using the algorithm described below. During a core transition, that core does not execute instructions; other cores can operate normally. To adjust the memory bus frequency, all memory accesses are temporarily halted, and PLLs and DLLs are resynchronized. Since the core and memory subsystem transition overheads are small (tens of microseconds) compared to our epoch size (milliseconds), the penalty is negligible.

The epoch executes to completion with the new voltages and frequencies. At the end of the epoch, CoScale again estimates the accumulated slack, by querying the performance counters and estimating what performance would have been achieved had the cores and the memory subsystem operated at maximum frequency. These estimates are then compared to achieved performance, with the difference used to update the accumulated slack and carried forward to calculate the target performance in the next epoch.

CoScale example. Figure 1 depicts an example of CoScale's behavior (bottom), compared to a policy that does not fully coordinate the processor and memory frequency selections (top). We refer to the latter policy as *semi-coordinated*, as it maintains a single performance slack (a mild form of coordination) that is shared by separate CPU and memory power state managers. As the figure illustrates, under semi-coordinated control, the CPU manager and the memory manager independently decide to scale down when they observe performance slack (performance above target). Unfortunately, because they are unaware of the cumulative effect of their decisions, they over-correct by scaling frequency too far down. For the same reason, in the following epoch, they over-react again by scaling frequency too far up. Such over-reactions continue in an oscillating manner. With CoScale, by modeling the joint effect of CPU and memory scaling, the appropriate frequency combination can be chosen to meet the precise performance target. Our control policy avoids both over-correction and oscillation.

3.1. CoScale's Frequency Selection Algorithm

When choosing a frequency for each core and a frequency for the memory bus, we have two goals. First, we wish to select a frequency combination that maximizes full-system energy savings. The energy-minimal combination is not necessarily that with the lowest frequencies; lowering frequency can increase energy consumption if the slowdown is too high. Our models explicitly account for the system-vs.-component energy balance. Fortunately, the cores and memory subsystem consume a large fraction of total system power, allowing CoScale to aggressively consume the performance slack. Second, we seek to observe the bound on allowable cycles per instruction (CPI) degradation for each running program.

Dynamically selecting the optimal frequency settings is challenging, since there are $M \times C^N$ possibilities, where M is the number of memory frequencies, C is the number of possible core frequencies, and N is the number of cores. M and C are typically on the order of 10, whereas N is in the range of 8-16 now but is growing fast. Thus, CoScale uses the greedy heuristic policy described in Figure 2.

Our gradient-descent heuristic iteratively estimates, via our online models, the marginal benefit (measured as $\Delta power/\Delta performance$) of altering either the frequency of the memory subsystem or that of various groups of cores by one step (we discuss core grouping in detail below). Initially, the algorithm estimates performance assuming all cores and memory are set to their highest possible frequencies (line 1 in the figure). It then iteratively considers frequency reductions, as long as some frequency can still be lowered without violating the performance slack (loop starting in line 2). When presented with a choice between next scaling down memory or a group of cores, the heuristic greedily selects the choice that will produce the highest marginal benefit (lines 3-12). If only memory or only cores can be scaled down, the available option is taken (line 13-19). Still in the main loop, the algorithm computes and records the full-system energy ratio (SER, Section 3.3) for the considered frequency configuration. When no more frequency reductions can be tried without violating the slack, the algorithm selects the configuration yielding the smallest SER (i.e., the best full-system energy savings) (line 21) and directs the hardware to transition frequencies (line 22).

```
1.  Estimate performance with each core and the memory subsystem at their highest frequencies
2.  While any component can be scaled down further without slack violation
3.    If both memory and at least one core can still scale down by 1 step
4.      If the memory frequency has changed since we last computed marginal_memory
5.        Compute marginal utility of lowering memory frequency as marginal_memory
6.      If any core frequency has changed since we last computed marginal_cores
7.        Compute marginal utility of lowering the frequency of core groups (per algorithm in Figure 3)
8.        Select the core group (group_best) with the largest utility (marginal_cores)
9.      If marginal_memory is greater than marginal_cores
10.       Scale down memory by 1 step
11.     Else
12.       Scale down cores in group_best by 1 step each
13.   Else if only memory can scale down
14.     Scale down memory by 1 step
15.   Else if only core groups can scale down
16.     If any core frequency has changed since we last computed marginal_cores
17.       Compute marginal utility of lowering the frequency of core groups (per algorithm in Figure 3)
18.       Select the core group (group_best) with largest marginal utility (marginal_cores)
19.     Scale down cores in group_best by 1 step each
20.   Compute and record the SER for the current combination of core and memory frequencies
21. Select the core and memory frequency combination with the smallest SER
22. Transition hardware to the new frequency combination
```

Figure 2: CoScale's greedy gradient-descent frequency selection algorithm.

```
1.  Scan the previous list of cores, removing any that may not scale down further or whose frequency has changed
2.  Re-insert cores with changed frequency, maintaining an ascending sort order by delta performance
3.  For group i from 1 to number of cores on the list
4.    Let delta power of the i-th group be equal to the sum of delta power from first to the i-th core
5.    Let delta performance be equal to delta performance of the i-th core
6.    Let marginal utility of i-th group be equal to delta power over delta performance just calculated
7.  Set the group with the largest marginal utility as the best group (group_best) and its utility as marginal_cores
```

Figure 3: Sub-algorithm to consider core frequency changes by group.

Changing the frequency of the memory subsystem impacts the performance of all cores. Thus, when we compute the $\Delta performance$ of lowering memory frequency, we choose the highest performance loss of any core. Similarly, when computing the $\Delta performance$ of lowering the frequencies of a group of cores, we consider the worst performance loss in the group. The $\Delta power$ in these cases is the power reduction that can be achieved by lowering the frequency of each core in the group.

An important aspect of the CoScale heuristic is that it considers lowering the frequency of cores in groups of 1, 2, 3, ..., N cores (lines 1-6 in Figure 3). The group formation algorithm maintains a list of cores that are eligible to scale down in frequency (i.e., they can be scaled down without slack violation), sorted in ascending order of $\Delta performance$. To avoid a potentially expensive sort operation on each invocation, the algorithm updates the existing sorted list by removing and then re-inserting only those cores whose frequency has changed (lines 1-2). N possible core groups are considered, forming groups greedily by first selecting the core that incurs the smallest delta performance from scaling (i.e., just the head of the list), then considering this core and the second core, then the third, and so on. This greedy group formation avoids combinatorial state space explosion, but, as we will show, it performs similarly to an offline method that considers all combinations. Considering transitions by group is needed to prevent CoScale from always lowering memory frequency first, because the memory subsystem at first tends to provide greater benefit than scaling any one core in isolation. Failing to consider group transitions may cause the heuristic to get stuck in local minima.

Our algorithm is run at the end of the profiling phase of each epoch (5ms by default). Because of core grouping, the complexity of our heuristic is $O(M + C \times N^2)$, which is exponentially better than that of the brute-force approach. Given our default simulation settings for M (10), C (10), and N (16), searching once per epoch has negligible overhead. Specifically, in all our experiments, searching takes less than 5 microseconds on a 2.4GHz Xeon machine. Our projections for larger core counts suggest that the algorithm could take 83 and 360 microseconds for 64 and 128 cores, respectively, in the worst case (4 microseconds in the best case). If one finds it necessary to hide these higher overheads, one can either increase the epoch length or dedicate a spare core to the algorithm.

3.2. Comparison with Other Policies

The key aspect of CoScale is the efficient way in which it searches the space of possible CPU and memory frequency settings. For comparison, we study five alternatives. The first, called "MemScale", represents the scenario in which the system uses only memory subsystem DVFS. The second alternative, called "CPUOnly", represents the scenario with CPU DVFS only. To be optimistic about this alternative, we assume that it considers all possible combinations of core frequencies and selects the best. In both MemScale and CPUOnly, the performance-aware energy management policy assumes that the behavior of the components that are not being managed will stay the same in the next epoch as in the profiling phase.

The third alternative, called "Uncoordinated", applies both MemScale and CPU DVFS, but in a completely independent fashion. In determining the performance slack available to it, the CPU power manager assumes that the memory subsystem will remain at the same frequency as in the previous epoch, and that it has accumulated no CPI degradation; the memory power manager makes the same assumptions about the cores. Hence, each manager believes that it alone influences the slack in each epoch, which is not the case. The fourth alternative, called "Semi-coordinated", increases the level of coordination slightly by allowing the CPU and memory power managers to share the same overall slack, i.e. each manager is aware of the past CPI degradation produced by the other. However, each

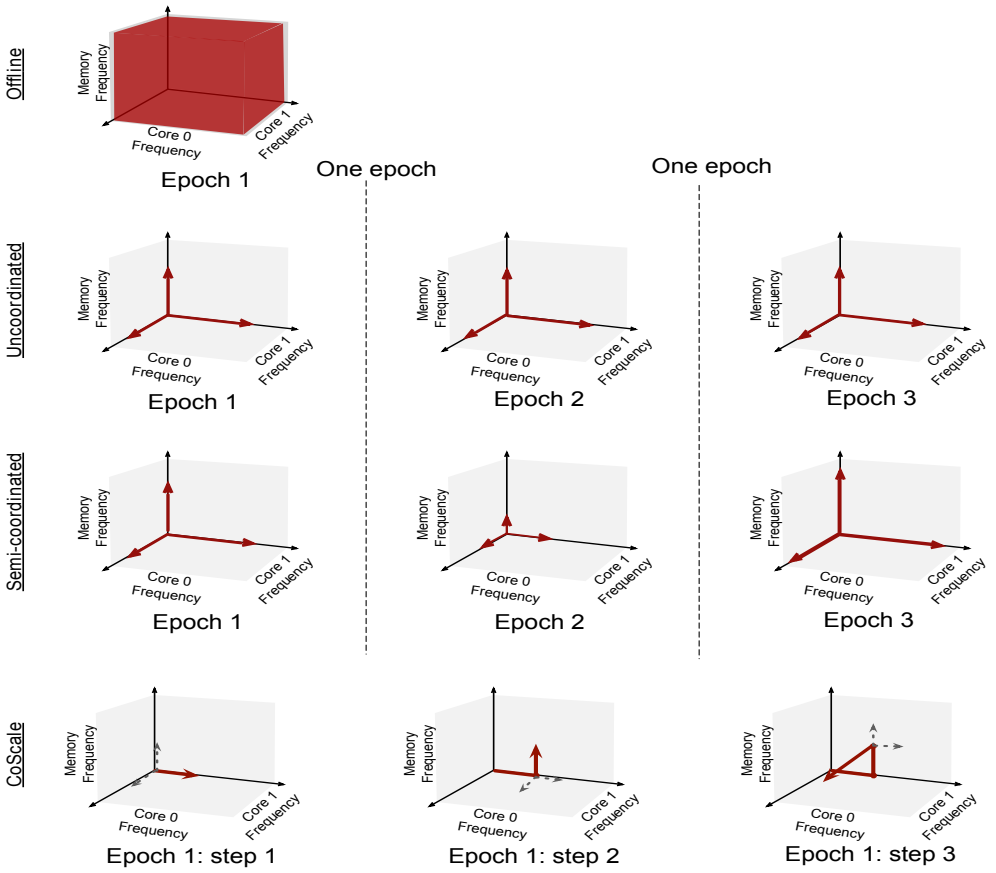

Figure 4: Search differences: CoScale searches the parameter space efficiently. Uncoordinated violates the performance bound and Semi-coordinated gets stuck in local minima.

manager still tries to consume the entire slack independently in each epoch (i.e., the two managers account for one another's past actions, but do not coordinate their estimate of future performance).

Finally, the fifth alternative, called "Offline", relies on a perfect offline performance trace for every epoch, and then selects the best frequency for each epoch by considering *all* possible core and memory frequency settings. As the number of possible settings is exponential, Offline is impractical and is studied simply as an upper bound on how well CoScale can do. However, Offline is not necessarily optimal, since it uses the same epoch-by-epoch greedy decision-making as CoScale (i.e., a hypothetical oracle might choose to accumulate slack in order to spend it in later epochs).

Figure 4 visualizes the difference between CoScale and other policies in terms of their search behaviors. For clarity, the figure considers only two cores (X and Y axes) and the memory (Z axis), forming a 3-D frequency space. The origin point is the highest frequency of each dimension; more distant points represent lower per-component frequencies. CPUOnly and MemScale search subsets of these three dimensions, so we do not illustrate them.

We can see from the figure that the Offline policy (top illustration) examines the entire space, thus always finding the best configuration. Under the Uncoordinated policy (second row), the CPU power manager tries to consume as much of the slack as possible with cores 0 and 1, while the memory power manager gets to consume the same slack. This repeats every epoch. Semi-coordinated (third row) behaves similarly in the first epoch. However, in the second epoch, to correct for the overshoot in the first epoch, each manager is restricted to a smaller search space. This restriction leads to over-correction in the third epoch, resulting in a much larger search space. The resulting oscillation may continue across many epochs. Finally, CoScale (bottom row) starts from the origin and greedily considers steps of memory frequency or (groups of) core frequency, selecting the move with the maximal marginal energy/performance benefit. From the figure, we can see that in step 1, CoScale scaled core 0 down by one frequency level; then it scaled the memory frequency down in step 2; and finally scaled core 1 down by two frequency levels in step 3. The search then terminates, because the performance model predicts that any further moves will violate the performance bound of at least one application. CoScale's greedy walk is shorter and produces better results than the other practical approaches.

Although CoScale provides no formal guarantees precluding oscillating behavior, this behavior is unlikely and occurs only when the profiling phases are consistently poor predictions of the rest of the epochs, or the performance models are inaccurate. On the other hand, the Semi-coordinated and Uncoordinated policies exhibit poor behavior due to their design limitations.

3.3. Implementation

We now describe the performance counters and performance/power models used by CoScale.

Performance counters. CoScale extends the performance modeling framework of MemScale [10] with additional performance counters

that allow it to estimate core power (in addition to memory power) and assess the degree to which a workload is instruction throughput vs. memory bound.

- **Instruction counts** – For each core, CoScale requires counters for *Total Instructions Committed* (`TIC`), *Total L1 Miss Stalls* (`TMS`), *Total L2 Accesses* (`TLA`), *Total L2 Misses* (`TLM`), and *Total L2 Miss Stalls* (`TLS`). CoScale uses these counters to estimate the fraction of CPI attributable to the core and memory, respectively. These counters allow the model to handle many core types (in-order, out-of-order, with or without prefetching), whereas MemScale's model (which required only `TIC` and `TMS`) supports only in-order cores without prefetching.
- **Memory subsystem performance** – CoScale reuses the same seven memory performance counters introduced by MemScale, which track memory queuing statistics and row buffer performance. We refer readers to [10] for details.
- **Power modeling** – To estimate core power, CoScale needs the L1 and L2 counters mentioned above and per-core sets of four *Core Activity Counters* (`CAC`) that track committed ALU instructions, FPU instructions, branch instructions, and load/store instructions. We reuse the memory power model from MemScale, which requires two counters per channel to track active vs. idle cycles and the number of page open/close events (details in [10]).

In total, CoScale requires eight additional counters per core beyond the requirements of MemScale (which requires two per core and nine per memory channel, all but five of which already exist in current Intel processors).

Performance model. Our model builds upon that proposed in [10], with two key enhancements: (1) we extend it to account for varying CPU frequencies, and (2) we generalize it to apply to cores with memory-level-parallelism (e.g., out-of-order cores or cores with prefetchers).

The performance model predicts the relationship between CPI, core frequency, and memory frequency, allowing it to determine the runtime and power/energy implications of changing core and memory performance. Given this model, the OS can set the frequencies to both maximize energy-efficiency and stay within the predefined limit for CPI loss.

CoScale models the rate of progress of an application in terms of CPI. The average CPI of a program is defined as:

$$E[\text{CPI}] = (E[TPI_{\text{CPU}}] + \alpha \cdot E[TPI_{\text{L2}}] + \beta \cdot E[TPI_{\text{Mem}}]) \cdot F_{\text{CPU}} \quad (1)$$

where $E[TPI_{\text{CPU}}]$ represents the average time that instructions spend on the CPU (including L1 cache hits), α is the fraction of instructions that access the L2 cache and stall the pipeline, $E[TPI_{\text{L2}}]$ is the average time that an L1-missing instruction spends accessing the L2 cache while the pipeline is stalled, β is the fraction of instructions that miss the L2 cache and stall the pipeline, $E[TPI_{\text{Mem}}]$ is the average time that an L2-missing instruction spends in memory while the pipeline is stalled, and F_{CPU} is the operating frequency of the core. The value of α can be calculated as the ratio of `TMS` and `TIC`, whereas β is the ratio of `TLS` and `TIC`.

The expected CPU time of each instruction ($E[TPI_{\text{CPU}}]$) depends on core frequency, but is insensitive to memory frequency. Since we keep the frequency (and supply voltage) of the L2 cache fixed, the expected time per L2 access that stalls the pipeline ($E[TPI_{\text{L2}}]$) does not change with either core or memory frequency (we neglect the secondary effect of small variations in L1 snoop time). The expected time per L2 miss that stalls the pipeline ($E[TPI_{\text{Mem}}]$) varies with memory frequency. We decompose the latter time as in [10]: $E[TPI_{\text{Mem}}] = \xi_{\text{bank}} \cdot (S_{\text{Bank}} + \xi_{\text{bus}} \cdot S_{\text{Bus}})$, where ξ_{bus} represents the average number of requests waiting for the bus; ξ_{bank} are requests waiting for the bank; 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.

The above counters and model assume single-threaded applications, each running on a different core. To tackle multi-threaded applications, CoScale would require additional counters and a more sophisticated performance model (one that captures inter-thread interactions). To deal with context switching, CoScale can maintain the performance slack independently for each software thread.

Full-system energy model. 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 as a function of the frequencies of the cores and memory subsystem.

For frequency f^i_{core} for core i and memory frequency f_{mem}, we define the *system energy ratio* (SER) as:

$$\text{SER}(f^1_{\text{core}}, ..., f^n_{\text{core}}, f_{\text{mem}}) = \frac{T_{f^1_{\text{core}},...,f^n_{\text{core}},f_{\text{Mem}}} \cdot P_{f^1_{\text{core}},...,f^n_{\text{core}},f_{\text{Mem}}}}{T_{\text{Base}} \cdot P_{\text{Base}}} \quad (2)$$

Here, T_{Base} and P_{Base} are time and average power at a nominal frequency (e.g., the maximum frequencies). $T_{f^1_{\text{core}},...,f^n_{\text{core}},\text{Mem}}$ is the time estimate for an epoch at frequencies $f^1_{\text{core}}, ..., f^n_{\text{core}}$ for the n cores and frequency f_{Mem} for the memory subsystem. This time estimate corresponds to the core with the highest CPI degradation compared to running at maximum frequency.

$$P_{f^1_{\text{core}},...,f^n_{\text{core}},f_{\text{Mem}}} = P_{\text{NonCoreL2OrMem}} + P_{\text{L2}} + P_{\text{Mem}}(f_{\text{Mem}}) + \sum_{i=1}^{n} P^i_{\text{Core}}(f^i_{\text{core}}). \quad (3)$$

In this formula, $P_{NonCoreL2OrMem}$ accounts for all system components other than the cores, the shared L2 cache, and the memory subsystem, and is assumed to be fixed. P_{L2} is the average power of the L2 cache and is computed from its leakage and number of accesses during the epoch. $P_{\text{Mem}}(f)$ is the average power of L2 misses and is calculated according to the model for memory power in [33]. We find that this average power does not vary significantly with core frequency (roughly 1-2% in our simulations); workload and memory bus frequency have a stronger impact. Thus, our power model assumes that core frequency does not affect memory power. $P^i_{\text{Core}}(f)$ is calculated based on the cores' activity factors using the same approach as prior work [3, 18]. We also find that the power of the cores is essentially insensitive to the memory frequency.

3.4. Hardware and Software Costs

We now consider CoScale's implementation cost. Core DVFS is widely available in commodity hardware, although each voltage domain may currently contain several cores. Though CPUs with multiple frequency domains are common, there have historically been few voltage domains; however, research has shown this is likely to change soon [21, 40].

Our design also may require enhancements to performance counters in some processors. Most processors already expose a set of counters to observe processing, caching and memory-related performance behaviors (e.g., row buffer hits/misses, row pre-charges).

Table 1: Workload descriptions.

Name	MPKI	WPKI	Applications (x4 each)			
ILP1	0.37	0.06	vortex	gcc	sixtrack	mesa
ILP3	0.27	0.07	sixtrack	mesa	perlbmk	crafty
ILP2	0.16	0.03	perlbmk	crafty	gzip	eon
ILP4	0.25	0.04	vortex	mesa	perlbmk	crafty
MID1	1.76	0.74	ammp	gap	wupwise	vpr
MID3	1.00	0.60	apsi	bzip2	ammp	gap
MID2	2.61	0.89	astar	parser	twolf	facerec
MID4	2.13	0.90	wupwise	vpr	astar	parser
MEM1	18.2	7.92	swim	applu	galgel	equake
MEM3	7.93	2.55	fma3d	mgrid	galgel	equake
MEM2	7.75	2.53	art	milc	mgrid	fma3d
MEM4	15.07	7.31	swim	applu	sphinx3	lucas
MIX1	2.93	2.56	applu	hmmer	gap	gzip
MIX3	2.55	0.80	equake	ammp	sjeng	crafty
MIX2	2.34	0.39	milc	gobmk	facerec	perlbmk
MIX4	2.35	1.38	swim	ammp	twolf	sixtrack

In fact, the latest Intel architecture exposes many MC counters for queues [25]. However, the existing counters may not conform precisely to the specifications required for our models.

When CoScale adjusts the frequency of a component, the component briefly suspends operation. However, as our policy operates at the granularity of multiple milliseconds, and transition latencies are in the tens of microseconds, the overheads are negligible. As mentioned above, the execution time of the search algorithm is not a major concern.

Existing DIMMs support multiple frequencies and can switch among them by transitioning to powerdown or self-refresh states [19], although this capability is typically not used by current servers. Integrated CMOS MCs can leverage existing DVFS technology. One needed change is for the MC to have separate voltage and frequency control from other processor components. In recent Intel architectures, this would require separating last-level cache and MC voltage control [17]. Although changing the voltage of DIMMs and DRAM peripheral circuitry is possible [23], there are no commercial devices with this capability.

4. Evaluation

We now present our methodology and results.

4.1. Methodology

Workloads. Table 1 describes the workload mixes we use. We construct the workloads by combining applications from the SPEC 2000 and SPEC 2006 suites. We use workloads exhibiting a range of compute and memory behavior, and group them into the same mixes as [10, 41]. The workload classes are: memory-intensive (MEM), compute-intensive (ILP), compute-memory balanced (MID), and mixed (MIX, one or two applications from each other class). The rightmost column of Table 1 lists the application composition of each workload; four copies of each application are executed to occupy all 16 cores.

We run the best 100M-instruction simulation point for each application (selected using Simpoints 3.0 [35]). A workload terminates when its slowest application has run 100M instructions. Table 1 lists the LLC misses per kilo-instruction (MPKI) and writebacks per kilo-instruction (WPKI). In terms of the workloads' running times, the memory-intensive workloads tend to run more slowly than the CPU-intensive ones. On average, the numbers of epochs are: 46 for MEM workloads, 32 for MIX, 15 for MID, and 10 for ILP.

Simulation infrastructure. Our evaluation uses a two-step simulation methodology. In the first step, we use M5 [4] to collect memory access traces (consisting of L1 cache misses and writebacks), and per-core activity counter traces. In the second step, we feed the memory traces into our detailed LLC/memory simulator of a 16-core CMP with a shared L2 cache (LLC), on-chip MC, memory channels, and DRAM devices. We also feed core activity traces, along with the run-time statistics from the L2 module, into McPAT [26] to dynamically estimate the CPU power. Overall, our infrastructure simulates in detail the aspects of cores, caches, MC, and memory devices that are relevant to our study, including memory device power and timing, and row buffer management.

Table 2: Main system settings.

Feature		Value
CPU cores		16 in-order, single thread, 4GHz Single IALU IMul FpALU FpMulDiv
L1 I/D cache (per core)		32KB, 4-way, 1 CPU cycle hit
L2 cache (shared)		16MB, 16-way, 30 CPU cycle hit
Cache block size		64 bytes
Memory configuration		4 DDR3 channels, 8 2GB ECC DIMMs
Time	tRCD, tRP, tCL	15ns, 15ns, 15ns
	tFAW	20 cycles
	tRTP	5 cycles
	tRAS	28 cycles
	tRRD	4 cycles
	Refresh period	64ms
Current	Row buffer read, write	250 mA, 250 mA
	Activation-precharge	120 mA
	Active standby	67 mA
	Active powerdown	45 mA
	Precharge standby	70 mA
	Precharge powerdown	45 mA
	Refresh	240 mA

Table 2 lists our default simulation settings. We simulate in-order cores with the Alpha ISA. Each core is allowed one outstanding LLC miss at a time. Like [10], we compensate for the lower memory traffic of these assumptions by simulating prefetching in Section 4.2.4. In the same section, we investigate an optimistic out-of-order design.

Table 2 also details the memory subsystem we simulate: 4 DDR3 channels, each of which populated with two registered, dual-ranked DIMMs with 18 DRAM chips each. Each DIMM also has a PLL device and 8 banks. Timing and power parameters are taken from Micron datasheets for 800 MHz devices [32].

Our simulated MC exploits bank interleaving and uses closed-page row buffer management, which outperforms open-page policies for multi-core CPUs [38]. Memory read requests (cache misses) are scheduled using FCFS, with reads given priority over writebacks until the writeback queue is half-full. More sophisticated memory scheduling is unnecessary for our single-issue workloads, as opportunities to increase bank hit rate via scheduling are rare.

We assume per-core DVFS, with 10 equally-spaced frequencies in the range 2.2-4.0 GHz. We assume a voltage range matching Intel's Sandybridge, from 0.65 V to 1.2 V, with voltage and frequency scaling proportionally, which matches the behavior we measured on an i7 CPU. We assume uncore components, such as the shared LLC, are always clocked at the nominal frequency and voltage.

As in [10], we scale MC frequency and voltage, but only frequency for the memory bus and DRAM chips. The on-chip 4-channel MC has the same voltage range as the cores, and its frequency is always double that of the memory bus. We assume that the memory bus and DRAM chips may be frequency-scaled from 800 MHz to 200 MHz, with steps of 66 MHz. We determine power at each frequency using Micron's calculator [32]. Transitions between bus frequencies are assumed to take 512 memory cycles plus 28 ns, which accounts for a DRAM state transition to fast-exit precharge powerdown and

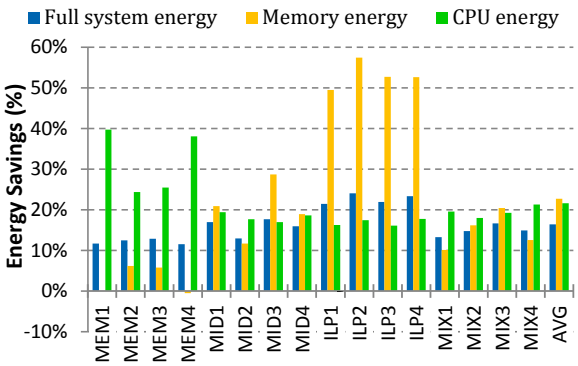

Figure 5: CoScale energy savings. CoScale conserves up to 24% of the full-system energy.

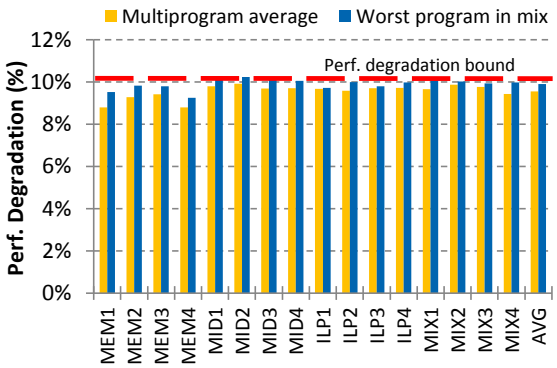

Figure 6: CoScale performance. CoScale never violates the 10% performance bound.

DLL re-locking [19, 10]. Some components' power draws also vary with utilization. Specifically, register and MC power scale linearly with utilization, whereas PLL power scales only with frequency and voltage. As a function of utilization, the PLL/register power ranges from 0.1 W to 0.5 W [10, 15, 17], whereas the MC power ranges from 4.5 W to 15 W.

We do not model power for non-CPU, non-memory system components in detail; rather, we assume these components contribute a fixed 10% of the total system power in the absence of energy management (we show the impact of varying this percentage in Section 4.2.4).

Under our baseline assumptions, at maximum frequencies, the CPU accounts for roughly 60%, the memory subsystem 30%, and other components 10% of system power.

4.2. Results

4.2.1. Energy and Performance We first evaluate CoScale with a maximum allowable performance degradation of 10%. We consider lower performance bounds in Section 4.2.4.

Figure 5 shows the full-system, memory, and CPU energy savings CoScale achieves for each workload, compared to a baseline without energy management (i.e., maximum frequencies). The memory energy savings range from -0.5% to 57% and the CPU energy savings range from 16% to 40%. As one would expect, the ILP workloads achieve the highest memory and lowest CPU energy savings, but still save at least 21% system energy.

The memory energy savings in the MID and MIX workloads are lower but still significant, whereas the CPU energy savings are somewhat higher (system energy savings of at least 13% for both workload classes). Note that CoScale is successful at picking the right energy saving "knob" in the MIX workloads. Specifically, it more aggressively conserves memory energy in MIX3, whereas it more aggressively conserves CPU energy in MIX1, MIX2, and MIX4.

The MEM workloads achieve the smallest memory and largest CPU energy savings (system energy savings of at least 12%), since their greater memory channel traffic reduces the opportunities for memory subsystem DVFS.

Figure 6 shows the average and maximum percent performance losses relative to the maximum-frequency baseline. The figure shows that CoScale never violates the performance bound. Moreover, CoScale translates nearly all the performance slack into energy savings, with an average performance loss of 9.6%, quite near the 10% target.

In summary, *CoScale conserves between 13% and 24% full-system energy for a wide range of workloads, always within the user-defined performance bounds.*

4.2.2. Dynamic Behavior To provide greater insight, we study an example of the dynamic behavior of CoScale in detail. Figure 7 plots the memory subsystem and core frequency (for milc in MIX2) selected by CoScale over time. For comparison, we also show the behavior of the Uncoordinated and Semi-coordinated policies.

Figure 7(a) shows that, in epoch two, CoScale reduces the core and memory frequencies to consume the available slack. In this phase, milc has low memory traffic needs, but the other applications in the mix preclude lowering the memory frequency further. Near epoch 10, another application's traffic spike results in a memory frequency increase, allowing a reduction of core frequency for milc. Near epoch 14, milc undergoes a phase change and becomes more memory-bound.

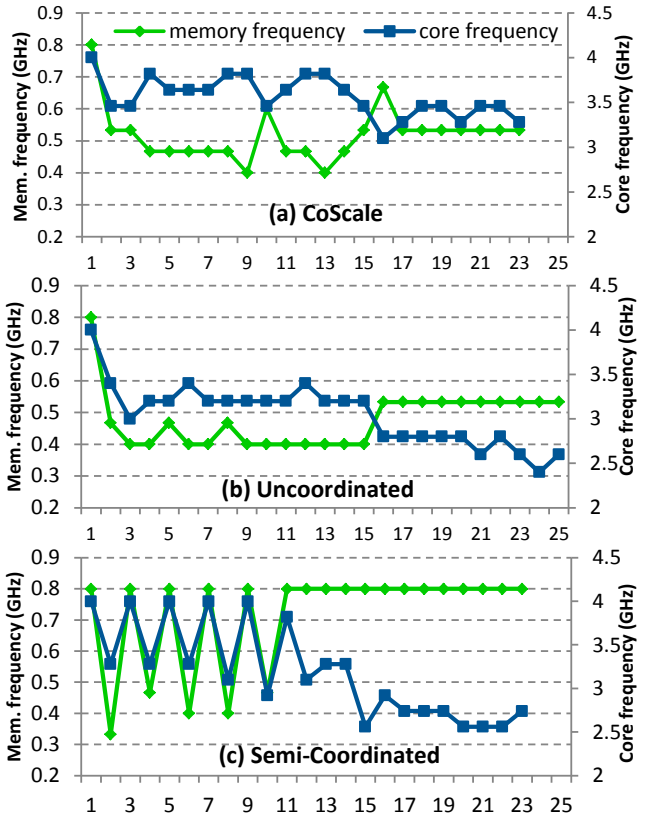

Figure 7: Timeline of the milc application in MIX2. Milc exhibits three phases. CoScale adjusts core and memory subsystem frequency precisely and rapidly in response to the phase changes. The other techniques do not.

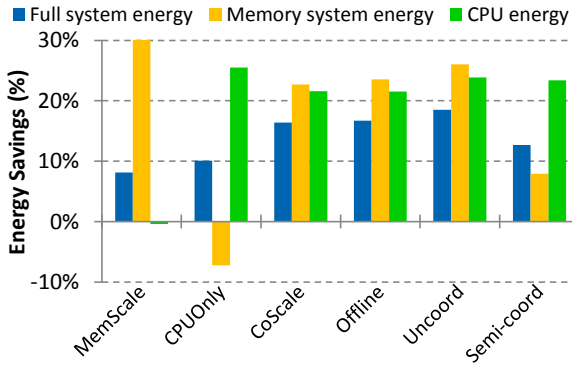

Figure 8: Energy savings. CoScale provides greater full-system energy savings than the practical policies.

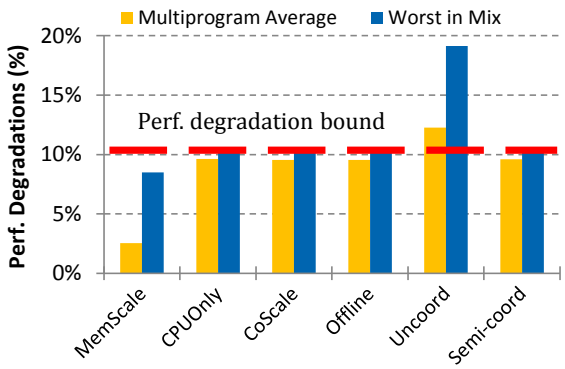

Figure 9: Performance. Uncoordinated is incapable of limiting performance degradation.

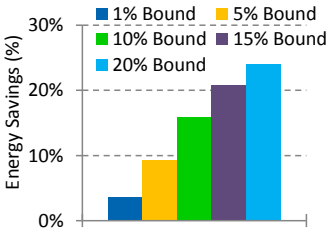

Figure 10: Impact of performance bound. Higher bound allows more savings without violations.

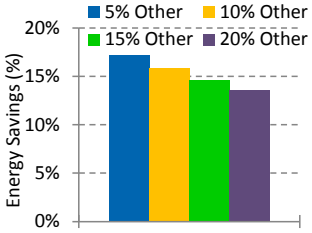

Figure 11: Impact of rest-of-system power. Savings still high for higher rest-of-system power.

As a result, CoScale increases the memory frequency, while reducing the core frequency.

Figure 7(b) shows a similar timeline for Uncoordinated. On the whole, the frequency transitions follow the same trend as in CoScale. However, both frequencies are markedly lower. Because there is no coordination, both CPU and memory power managers try to consume the same slack. These lower frequencies result in a longer running time (23 vs 25 epochs), violating the performance bound.

Figure 7(c) plots the timeline for Semi-coordinated. Initially, it incurs frequency oscillations until the traffic spike at epoch 10 causes memory frequency to become pegged at 800MHz. At that point, the CPU frequency for milc is also lowered considerably to consume all remaining slack. Unlike Uncoordinated, Semi-coordinated is successful in meeting the performance bound as slack estimation is coordinated among controllers. However, both the oscillations and the local minima selected after epoch 12 result in lower energy savings relative to CoScale. Altering the CPU and memory power managers to make their decisions half an epoch out of phase reduces oscillation, but the system gets stuck at local minima even sooner (around the 7th epoch). Making decisions an entire epoch out of phase produces similar behavior.

4.2.3. Energy and Performance Comparison Figure 8 contrasts average energy savings and Figure 9 contrasts average and worst-case performance degradation across polices. These results demonstrate that MemScale and CPUOnly are of limited use. Although they save considerable energy in the component they manage (MemScale conserves 30% memory energy, whereas CPUOnly conserves 26% CPU energy), gains are partially offset by higher energy consumption in the other component (longer runtime leads to higher background/leakage energy for the unmanaged component). These schemes save at most 10% full-system energy.

Uncoordinated conserves substantial memory and CPU energy, achieving the highest full-system energy savings of any scheme. Unfortunately, it is incapable of keeping the performance loss under the pre-defined 10% bound. In some cases, the performance degradation reaches 19%, nearly twice the bound. On the other hand, Semi-coordinated bounds performance well because the managers share the slack estimate. However, because of frequent oscillations and settling at sub-optimal local minima, Semi-coordinated consumes up to 8% more system energy (2.6% on average) than CoScale. Reducing oscillations by having the power managers make decisions out of phase does not improve results (0.3% lower savings with the same performance).

CoScale is more stable and effective than the other practical policies at conserving both memory and CPU energy, while staying within the performance bound. CoScale does almost as well as Offline. These results show that our heuristic for selecting frequencies is almost as effective as considering an exponential number of possibilities with prior knowledge of each workload's behavior.

4.2.4. Sensitivity Analysis To illustrate CoScale's behavior across different system and policy settings, we report on several sensitivity studies. In every case, we vary a single parameter at a time, leaving the others at their default values. Given the large number of potential experiments, we usually present results only for the MID workloads, which are sensitive to both memory and core performance.

Acceptable performance loss. In Figure 10, we vary the maximum *allowable* performance degradation, showing energy savings. Recall that our other experiments use a bound of 10%. As one would expect, 1% and 5% bounds produce lower energy savings, averaging 4% and 9%, respectively. Allowing 15% and 20% degradations saves more energy. In all cases, CoScale meets the configured bound, and provides greater percent energy savings than performance loss, even for tight performance bounds.

Rest-of-the-system power consumption. Figure 11 illustrates the effect of doubling and halving our assumption for non-memory, non-core power. When this power is doubled, CoScale still achieves 14% average full-system energy savings, whereas the savings increase to 17% when it is halved. In all cases performance remains within bounds (not shown).

Ratio of memory subsystem and CPU power. We also consider the effect of varying the ratio of memory subsystem to CPU power. Recall that, under our baseline power assumptions, CPU accounts for 60%, while memory accounts for 30% of total power at peak frequency (a CPU:Mem ration of 2:1). In Figure 12, we consider

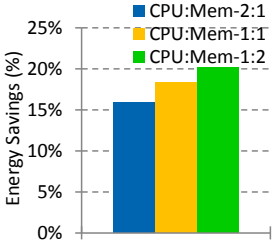

Figure 12: Impact of CPU:mem power, MID. Savings increase as memory power increases.

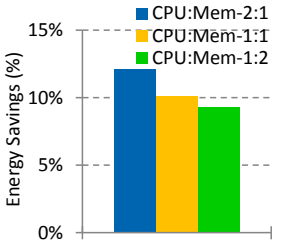

Figure 13: Impact of CPU:mem power, MEM. Savings decrease as memory power increases.

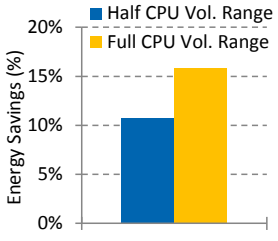

Figure 14: Impact of CPU voltage range. Smaller voltage ranges reduce energy savings.

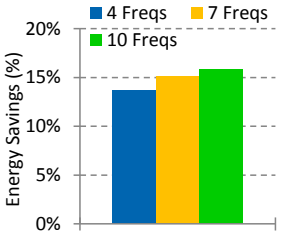

Figure 15: Impact of number of frequencies. Savings decrease little when fewer steps are avalaible.

1:1 and 1:2 ratios. CoScale achieves greater energy savings when the fraction of memory power is higher for the MID workloads. Interestingly, this trend is reversed for our MEM workloads (Figure 13), as most savings come from scaling the CPU.

CPU voltage range. We next consider the impact of a narrower CPU (and MC) voltage range, which reduces CoScale's ability to conserve core energy. Figure 14 shows results for a half-width range (0.95 1.2v) relative to our default assumption (0.65 1.2v). When the marginal utility of lowering CPU frequency decreases, CoScale scales the memory subsystem more aggressively and still achieves 11% full-system energy savings on average.

Number of available frequencies. By default, we assume 10 frequencies for both the CPU and the memory subsystem. Figure 15 shows results for 4 and 7 frequencies as well. As expected, the energy savings decrease as the granularity becomes coarser. However, CoScale adapts well, conserving only slightly less energy with fewer frequencies. With 4 frequencies the maximum performance loss is slightly lower than 10%, because the coarser granularity limits CoScale's ability to consume the slack precisely.

Prefetching. Next, we consider the impact of the increase in memory traffic that arises from prefetching. We implement a simple next-line prefetcher. This prefetcher is effective for these workloads, always decreasing the LLC miss rate. However, the prefetcher is not perfect; its accuracy ranges from 52% to 98% across our workloads. On average, it improves performance by almost 20% on MEM workloads, 8% on MIX, 4% on MID, and 1% for ILP. At the same time, it increases the memory traffic more than 33% on MEM, 20% on MID, 33% on MIX, and 13% on ILP. As one might expect, the higher memory traffic and instruction throughput result in higher memory and CPU power.

Figure 16 shows the full-system energy per instruction of three designs (Base+prefetching, Base+CoScale, and Base+prefetching+CoScale) normalized to our baseline (Base). We can see that the energy consumptions of Base+prefetching and Base are almost the same, except for the MEM workloads, since higher power and better performance roughly balance from an energy-efficiency perspective. Again except for MEM, the energy consumptions of Base+CoScale and Base+prefetching+CoScale are almost exactly the same, since average memory frequency is lower but CPU frequency is higher. For the MEM workloads, the performance improvement due to prefetching dominates the average power increase, so the average energy of Base+prefetching is 7% lower than Base. In addition, Base+prefetching+CoScale achieves 17% energy savings, compared to 12% from Base+CoScale. These results show that CoScale works well both with and without prefetching.

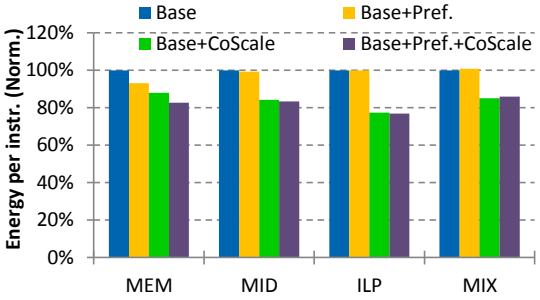

Figure 16: Impact of prefetching. CoScale works well with and without prefetching.

Out-of-Order. Although our trace-based methodology does not allow detailed out-of-order (OoO) modeling, we can approximate the latency hiding and additional memory pressure of OoO by emulating an instruction window during trace replay. We make the simplifying assumption that all memory operations within any 128-instruction window are independent, thereby modeling an upper bound on memory-level parallelism (MLP). Note that we still model a single-issue pipeline, hence, our instruction window creates MLP, but has no impact on instruction-level parallelism. Figure 17 compares the average CPI of the in-order and OoO designs, with and without CoScale, normalized to the in-order result. At one extreme, OoO drastically improves MEM, as memory stalls can frequently overlap. At the other extreme, ILP gains no benefit, since the infrequent L2 misses do not overlap frequently enough to impact performance. Note that, in the OoO+CoScale cases, performance remains within 10% of the OoO case; that is, CoScale is still maintaining the target degradation bound. Although we do not show these results in the figure, similar to the in-order case, Semi-coordinated on OoO meets the performance requirement, whereas Uncoordinated on OoO does not – Uncoordinated on OoO degrades performance by up to 16%, on a 10% performance loss bound.

Figure 18 shows average energy per instruction normalized to In-order. As we do not model any power overhead for OoO hardware structures (only the effects of higher instruction throughput and memory traffic), OoO always breaks even (ILP and MIX) or improves (MEM and MID) energy efficiency over In-order. Across the workloads, CoScale provides similar percent energy-efficiency gains for OoO as for In-order. The MEM case is the most interesting, as OoO has the largest impact on this workload. OoO increases memory bus utilization substantially (35% on average and up to 50%) and also results in far more queueing in the memory system (43% on average). The increased memory traffic balances with a reduced sensitivity

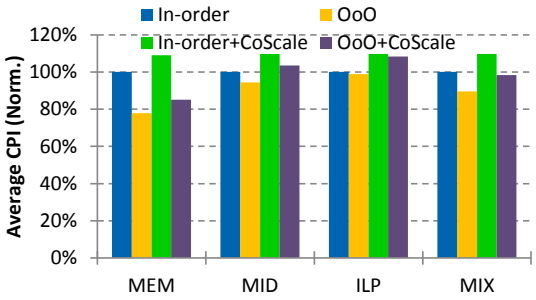

Figure 17: In-order vs OoO: performance. CoScale is within the performance bound in both in-order and OoO.

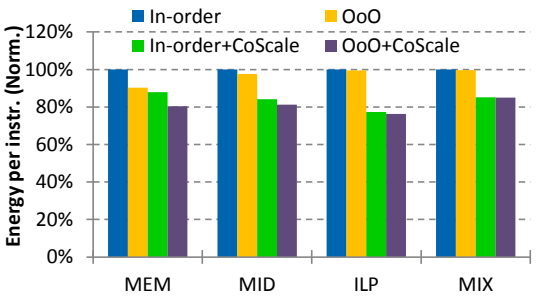

Figure 18: In-order vs OoO: energy. CoScale saves similar percent of energy in in-order and OoO.

to memory latency, and CoScale selects roughly the same memory frequencies under In-order and OoO. Interestingly, because of latency hiding, the MEM workload is more CPU-bound under OoO, and CoScale selects a slightly higher CPU frequency (5% higher on average). Again, we do not show results for Semi-coordinated and Uncoordinated on OoO in the figure, but their results are similar to those on an in-order design. Semi-coordinated on OoO causes frequency oscillation and leads to higher (up to 8%, and 4% on average) energy consumption than CoScale. Uncoordinated on OoO saves a little more energy (1% on average) than CoScale, but it violates the performance target significantly as mentioned above.

Summary. These sensitivity studies demonstrate that CoScale's performance modeling and control frameworks are robust—across the parameter space, CoScale always meets the target performance bound, while energy savings vary in line with expectations. Although the results in this subsection focused mostly on the MID workloads, we observed similar trends with the other workloads as well.

5. Conclusion

We proposed CoScale, a hardware-software approach for managing CPU and memory subsystem energy (via DVFS) in a coordinated fashion, under performance constraints. Our evaluation showed that CoScale conserves significant CPU, memory, and full-system energy, while staying within the performance bounds; that it is superior to four competing energy management techniques; and that it is robust over a wide parameter space. We conclude that CoScale's potential benefits far outweigh its small hardware costs.

Acknowledgements

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

References

[1] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12):33–37, 2007.

[2] 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*, 2009.

[3] F. Bellosa. The Benefits of Event-Driven Energy Accounting in Power-Sensitive Systems. In *SIGOPS European Workshop '00*, 2000.

[4] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, G. Saidi, and S. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4), July 2006.

[5] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*, 2008.

[6] M. Chen, X. Wang, and X. Li. Coordinating Processor and Main Memory for Efficient Server Power Control. In *ICS*, 2011.

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

[8] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. Hardware and Software Techniques for Controlling DRAM Power Modes. *IEEE Transactions on Computers*, 50(11), 2001.

[9] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. MultiScale: Memory System DVFS with Multiple Memory Controllers. In *ISLPED*, 2012.

[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. M. Jr, and R. Bianchini. Limiting the Power Consumption of Main Memory. *ISCA '07: International Symposium on Computer Architecture*, 2007.

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

[13] X. Fan, C. S. Ellis, and A. R. Lebeck. The Synergy between Power-aware Memory Systems and Processor Voltage Scaling. In *PACS*, 2003.

[14] W. Felter, K. Rajamani, T. Keller, and C. Rusu. A Performance-Conserving Approach for Reducing Peak Power Consumption in Server Systems. In *ICS*, 2005.

[15] E. Gorbatov, 2010. Personal communication.

[16] S. Herbert and D. Marculescu. Analysis of Dynamic Voltage/Frequency Scaling in Chip-Multiprocessors. In *ISLPED*, 2007.

[17] Intel. Intel ® Xeon ® Processor 5600 Series, 2010.

[18] C. Isci and M. Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *MICRO*, 2003.

[19] JEDEC. DDR3 SDRAM Standard, 2009.

[20] S. Kaxiras and M. Martonosi. Computer Architecture Techniques for Power-Efficiency. *Synthesis Lectures on Computer Architecture*, 2009.

[21] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks. System Level Analysis of Fast, Per-Core DVFS Using On-Chip Switching Regulators. In *HPCA*, 2008.

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

[23] H.-W. Lee, K.-H. Kim, Y.-K. Choi, J.-H. Shon, N.-K. Park, K.-W. Kim, C. Kim, Y.-J. Choi, and B.-T. Chung. A 1.6V 1.4 Gb/s/pin Consumer DRAM with Self-Dynamic Voltage-Scaling Technique in 44nm CMOS Technology. In *ISSCC*, 2011.

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

[25] D. Levinthal. Performance Analysis Guide for Intel ® Core ™ i7 Processor and Intel ® Xeon ™ 5500 processors, 2009.

[26] 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.

[27] X. Li, R. Gupta, S. Adve, and Y. Zhou. Cross-component energy management: Joint adaptation of processor and memory. In *ACM Trans. Archit. Code Optim.*, 2007.

[28] 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.

[29] 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.
[30] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: Eliminating Server Idle Power. In *ASPLOS*, 2009.
[31] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power Management of Online Data-Intensive Services. In *ISCA*, 2011.
[32] Micron. 1Gb: x4, x8, x16 DDR3 SDRAM, 2006.
[33] Micron. Calculating Memory System Power for DDR3, July 2007.
[34] V. Pandey, W. Jiang, Y. Zhou, and R. Bianchini. DMA-Aware Memory Energy Management. In *HPCA*, 2006.
[35] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for Accurate and Efficient Simulation Erez Perelman. In *SIGMETRICS*, 2003.
[36] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No "Power" Struggles: Coordinated Multi-level Power Management for the Data Center. In *ASPLOS*, 2011.
[37] D. Snowdon, S. Ruocco, and G. Heiser. Power Management and Dynamic Voltage Scaling: Myths and Facts. In *Power Aware Real-time Computing*, 2005.
[38] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-Pages: Increasing DRAM Efficiency with Locality-Aware Data Placement. In *ASPLOS*, 2010.
[39] 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.
[40] 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.
[41] H. Zheng, J. Lin, Z. Zhang, and Z. Zhu. Decoupled DIMM: Building High-Bandwidth Memory System Using Low-Speed DRAM Devices. In *ISCA*, 2009.