A Control Scheme for Batching DRAM Requests to Improve Power Efficiency

ABSTRACT

The increasing amounts of DRAM in modern servers have already made DRAM power the second largest component of platform power. Therefore, it is of interest to examine techniques for reducing power consumption of DRAM without significant impact on performance. In this paper we examine a technique that specifically controls the performance degradation due to power management of memory ranks and thereby automatically provides the maximum power savings that can be extracted given the current latency sensitivity of the workload. We show that the technique provides power savings over and above that are possible by individual control of memory ranks. Furthermore, the technique is general and can be used in many other contexts involving multiple instances of a resource.

1. INTRODUCTION

It is well known that computer systems waste a large percentage of the power consumed by them due to a variety of reasons including semiconductor leakage power and idle power consumption. Because of the increasing proliferation of information technology (IT), techniques for reducing power consumption of IT equipment is a problem of considerable current interest at all levels. In this paper we focus on the problem of coordinated power management of resources that have multiple instances. In particular, we focus on the main memory, whose relative share of platform power continues to increase. The memory is typically organized in form of multiple DIMMs (dual inline memory module) with each DIMM further organized as a set of "ranks", each of which can be power managed independently. We consider coordinated power management of multiple ranks. The coordination could span across single DIMM, across a memory channel, or even across all channels on a socket.

The technique proposed in this paper is a closed loop control that batches the traffic adaptively to enhance power efficiency while ensuring that the throughput degradation due to power management stays below the desired bound. Such a technique can be applied in other scenarios as well where there are multiple instances of a resource, e.g., multiple solid-state disks (SSD's), multiple sockets in a system, multiple cores of a CPU, etc. In non-storage situations, such as CPU cores or platform sockets, it is possible to switch requests from one instance to another. This is not possible for storage devices such as memory or SSD. (Data rearrangement may make sense at a much coarser granularity, however.) Even in case of non-storage resources such as CPU, request redistribution may have associated overheads such as perturbation to the working set accumulated in the caches, and needs to be considered carefully. In any case, a suitable request distribution mechanism can be combined with batching for even better energy efficiency whenever feasible, but we do not address that aspect here. Similarly, requests arriving at resource instance may be re-ordered for better performance and power efficiency, and such an intra-instance reordering can be combined with inter-instance batching.

The primary uniqueness of this paper is in studying the power efficiency gains in DDR DRAM by exploiting the coordinated use of available inactive or sleep states. This is quite different from the dynamic voltage/frequency switching (DVFS) based power management that has been explored extensively in the context of CPU power management. Memory systems and their power management have a number of idiosyncrasies of their own and hence the CPU power management techniques do not automatically translate to them.

The organization of the rest of the paper is as follows. Section 2 discusses the related work. Section 3 discusses the power management basics and compares isolated vs. coordinated power management schemes. Section 4 delves into the details of our coordinated power management algorithm. Section 5 introduces the experimental setup and briefly describes the simulation model used to obtain results, and section 6 discusses the results. Finally section 7 concludes the discussion.

2. RELATED WORK

The importance of power management in commercial servers, especially the memory systems is well known [15]. Benini, et al. [1] provide a tutorial on power optimization techniques and Venkatachalam et al. [21] provide a survey of power reduction techniques in microprocessors. Luo et al. [16] describe simultaneous active power management of both processor and links. This is a form of coordinated control be-

tween heterogeneous devices that we do not explore in this paper. The coordination technique that we have explored here can be seen as a form of traffic shaping that makes the traffic more batchy, and batching at all levels is a well recognized and effective technique for energy management [19].

Delaluz et al. [4] propose a proactive method for estimate memory device idle times and use it for RDRAM power management. Fan et al. [7] extend this work for systems with multi-level caches. Irani et al. [12] give a theoretical analysis of dynamic power management in memory controllers with multiple low power states. The various thresholds employed by the scheme are difficult to tune since they are system and application dependent. Reordering of memory requests for better performance is a well-known technique that has been examined amply in the past [11]. Hur and Lin also show how such reordering can be used to create longer gaps and thereby make power management more efficient [10]. Such a technique can be used in addition to our batching technique for additional savings.

Compiler-directed techniques for optimizing DRAM power consumption have been considered in several papers including [17] but are orthogonal to the direct techniques and usually of limited value due to the difficulty of accounting for multi-level cache impacts. Operating Systems level mechanisms to control scheduling in order to optimize DRAM power have been explored (e.g., see [5, 9]); however, these operate at a much coarser granularity than the hardware scheme discussed here. For example, Huang et al. [9] consider memory command reordering at page granularity in order to make the accesses more energy efficient.

Felter et al. [8] propose a DRAM power throttling mechanism in order to provide power to CPU when needed but such an approach can result in significant performance degradation. Diniz et al. [6] improve upon such a technique in order to reduce performance degradation by computing for each memory command a complete fine grained power estimate. Such a technique could be quite complex to implement.

Li et al. [14] presents a closed-loop scheme similar to our that measures additional latency due to power management and relates it to the slow-down. In our algorithm we directly and continuously monitor the throughput degradation (instead of attempting to deduce it from latency – which can be very difficult). Isci et al. [13] consider power management of multi-core processors using DVFS (dynamic voltage/frequency switching) under a variety of objectives including prioritization and optimized throughput. The approach in this paper is similar to our approach except that our effort concerns memory system and uses inactive state based power management for DRAM. As pointed out earlier, this is a significant change of context and requires a separate study.

3. POWER MANAGEMENT BASICS

3.1 Active and Passive Power States

Many system components including CPU cores, memory ranks, interconnection links provide at least two forms of power "states": (a) active states where the device continues to operate but at a lower performance level, and (b) passive states where the device can transition to one or more low-power non-operational mode when idle.

Dynamic voltage/frequency switching (DVFS) is the most popular form active power management using active power states. It includes two aspects: (a) operating the device at lower speed (or frequency) and thereby proportionately reducing the active power consumption, and (b) lowering the supply voltage level which reduces the power consumption significantly. In the past, a tremendous amount of work on power management is based on DVFS. Furthermore, a vast majority of this work has focused on CPU because CPU is typically the most power hungry component of the system. DVFS control is also available in all modern processors in form of "P" states along with software to use them transparently. Other components such as links and memory have also been considered for DVFS control, but much less so. Dynamic voltage control is usually not available in these subsystems; even the frequency control is usually provided through BIOS rather than dynamically.

The main attraction of DVFS is the voltage switching part since the power consumed is proportional to V^2. Unfortunately, the voltages are already hovering close to the threshold of reliable operation and in the near future there will be little scope for meaningful levels of lower voltages. This will turn DVFS into primarily a frequency control, which usually doesn't provide much power savings and also involves frequency resynchronization overheads. In view of this, the use of inactive states is becoming quite important. Also, inactive states are already implemented in a variety of devices such as CPU, memory and links, and continue to get enhanced.

The available low power memory states depend on the memory technology and to some extent on the vendor. Here we describe the states relevant for DDR3 technology, which is beginning to go mainstream. However, for a clear explanation of these, it is essential to briefly review the memory architecture.

In modern systems, each CPU socket (or package) includes an integrated memory controller (MC) which usually supports multiple *channels* that transport data between DRAM chips and memory controller. Each channel can support one or more memory DIMMs. Furthermore, each DIMM is divided into '*ranks*, with 1, 2 or 4 ranks per DIMM. A "rank" is a set of memory devices that can independently provide the entire 64 bits (8 bytes) needed to transfer a chunk over the memory channel. A rank is further divided up into banks (e.g., 8 banks per ranks for DDR3), which means that it is possible to simultaneously schedule multiple requests per rank. Note that the data from all ranks of a DIMM must flow over the same channel; therefore, the throughput is usually limited by the channel bandwidth. From a power management perspective, this implies that individual ranks could be very lightly utilized even if the channel is quite busy.

A DRAM read can involve up to 4 phases: (a) RAS or row access strobe which selects a DRAM row, (b) CAS or column address strobe which selects a column and reads a page from DRAM, (c) placement of DRAM data onto the memory channel, and (d) page close which makes the page unavail-

able in the DRAM. A write operation follows a similar sequence with obvious changes related to the direction of data movement. DRAMs also support open page and adaptive page close policies but we do not delve into those here.

The basic unit of memory power management is a rank, although some aggregate power states involve entire DIMM or all DIMMs on a channel. In the list below, the first two power states apply to individual ranks, whereas others apply to larger aggregates.

- **Fast CKE:** In this state (denoted CKEf) the clock enable (CKE) signal for a rank is de-asserted and I/O buffers, sense amplifiers, and row/column decoders are all deactivated. However, the DLL (digital locked loop) is left running. Fast CKE exit latency is 6 ns and it consumes about 60% of idle power. The entry time is only 1 cycle, but once entered, the rank must stay in CKE for at least 4 cycles.
- **Slow CKE:** In this state (denoted CKEs), the DLL is also turned off, resulting in lower power consumption (about 40% of idle power) but at a much higher exit latency (25 ns). The entry characteristics are same as for fast CKE.
- **Register Mode:** If all ranks of a DIMM are in slow CKE mode, the DIMM register can also be turned off thereby leading to even lower power consumption w/o any additional latency. Further circuitry can be turned off if all DIMMs on a channel are in CKEs mode. We denote these modes as CKEsr. Supporting more than one DIMM per channel is becoming difficult and thus the two modes may be identical.
- **Self-refresh:** In this mode much of the DRAM circuitry is placed in an inactive low power mode and the required refresh is done by the DRAM itself (rather than by the memory controller). Self-refresh is usually too slow (0.75us exit latency) to be entered during active use of the memory. Self-refresh is usually appropriate when the CPU is in a deep "C" state (discussed next).

For completeness, we also briefly mention low power states for CPU and interconnection links. In case of CPU, each core can be in one of several states, often denoted as CC0, CC1, ... where CC0 is the active state and others are inactive states with decreasing power consumption but increasing exit latencies. In addition, there are "package states" denoted as C0, C1, ... which refer to the state of the entire socket. The socket C state is normally selected as the highest power state among all the cores. The interconnection links (e.g., inter-socket links, PCI-Express links, etc.) usually have three power states, namely L0, L0s and L1 where L0 is the operational state, L0s is the higher power inactive state which allows independent control of each direction of the link, and L1 is deeper sleep state requiring coordination between the two sides.

3.2 Isolated vs. Coordinated Low Power State Management

An isolated power management means that each resource unit performs its own independent transition between the various power states. In case of single inactive power state, a power state management algorithm has to make two simple decisions: (a) when to go into low-power state, and (b) when to exit it [20]. Because of the finite entry/exit time to/from low-power mode, it is undesirable to go into low-power mode for small gaps (or idle periods). Since the gap duration is not known in advance, the general technique is to monitor it for some period, henceforth called *runway*, and if the resource is still idle, start transition into the low-power mode.

The exit from low power state could be either *reactive* or *proactive* [4]. A reactive exit is directly driven by the arrival of requests. It is trivial to implement but results in full exit latency hit every time. A proactive exit is driven by a prediction model of future packet arrivals and initiation of exit in anticipation of an arrival. The motivation for a proactive exit is to minimize exit latency; however, its success depends on how well future events can be predicted based on the past history. Reference [2] presents a hybrid algorithm suitable for high speed hardware implementation.

It is important to note here that in order to achieve a fine granularity power control that can take advantage of idle periods of the order of 10's or 100's of underlying clock cycles, the power management must be implemented in special purpose HW using a rather small number of gates and primarily local information. More sophisticated software based controls can be useful, but must operate at much coarser time granularities. For example, in case of DRAM, it is possible to segregate "hot" and "cold" data so that the fine-grain HW power management can work more effectively.

We now discuss coordinated power management. One form of coordination is between different devices in the system such as between CPU and memory [8, 6]. Such a coordination is clearly important in order to minimize the performance degradation of the system due to power management of various component. In this paper, however, we consider coordination among multiple instances of the same resource type, such as multiple CPU cores, DRAM ranks, platform sockets, etc.

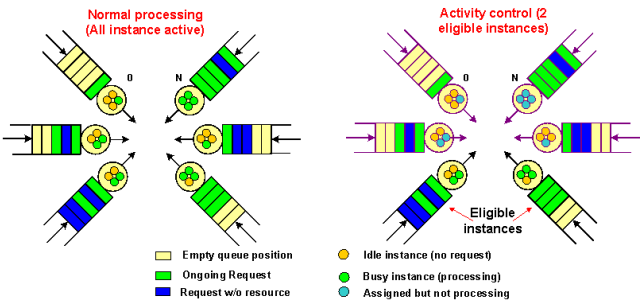

Figure 1: Illustration of coordinated instance control

4. COORDINATED INSTANCE CONTROL

Figure 1 illustrates the coordinated control pictorially. We have a total of N distinct resource instances, each having its own queue that holds the pending and active requests for that instance. Each instance is, in general, a multi-server

system, as shown in the figure by small circles (4 servers per instance in this case). For a memory rank, the servers correspond to the banks of a rank (8 in case of DDR3 technology). In case of CPU sockets, the servers correspond to the CPU cores or hardware threads per socket. Normally, each time a request arrives for an instance, the request is either scheduled immediately (if a suitable server is available) or when such a server becomes free. This is shown on the LHS of Fig. 1. (In case of memory, each request needs to go to a specific bank, hence a request can be scheduled only on the server corresponding to that bank.)

Although each queue itself may have some maximum limit, the number of pending requests is primarily a function of the request issuers. Thus, in case of memory ranks, the CPU cores or hardware threads issuing memory requests would typically run out of work after issuing a few memory requests and would "stall" until one of the requests completes. The number of requests that can be issued before a stall depends on the inherent parallelism of the software being executed and the extent to which that parallelism is exploited by the compiler.

The basic idea of the coordinated scheme is to allow only a subset all instances to schedule new requests as shown on the RHS of Fig. 1. The eligible subset remains so for a certain *dwell period* after which another subset becomes eligible. The policies for moving from one eligible subset to another should be such that starvation of any given instance is avoided. New requests can continue to arrive to the ineligible (or non-dwelling) subset and must queue up normally. This is not a problem in case of memory since pending memory requests are managed by the memory controller rather than by the DRAM.

An instance may have nonzero ongoing requests when it is marked as ineligible. (In fact, if all the servers of the instance are busy, it could even have some waiting requests at the time of being marked as ineligible). All ongoing requests will finish normally on an ineligible instance; the only difference is that no new requests are scheduled on the freed up servers. When all the servers become idle, the resource instance can be transitioned to a low power state. Notice that the instance could still have some waiting requests, and the instance will go to low power state in spite of them. Doing so requires some additional hardware capabilities but these are not difficult to implement in current memory controllers.

The batching and elongation of idle periods enforced by this scheme is what leads to enhanced power savings at the cost of some degradation in performance. The performance degradation results from forcing some servers to stay idle in spite of pending requests and due to the overhead of low-power mode transitions. The goal of the proposed mechanism is to maximize power savings subject to some acceptable limit on the performance degradation.

The set of ranks included in the access coordination may extend up to a DIMM, a channel, or an entire socket. For example, with dual rank DIMMs, 2 DIMMs per channel, and 2 channels per socket, the number of ranks managed together could be 2 (DIMM level), 4 (channel level), or 8 (socket level). Larger groups provide more opportunity for coordination and thus can provide more power savings for the same performance degradation but at the expense of somewhat more complex implementation. Extending coordination across natural boundaries (e.g., coordination across more than one memory controller) may become impractical. For this reason, we shall consider coordination among all ranks for each socket separately.

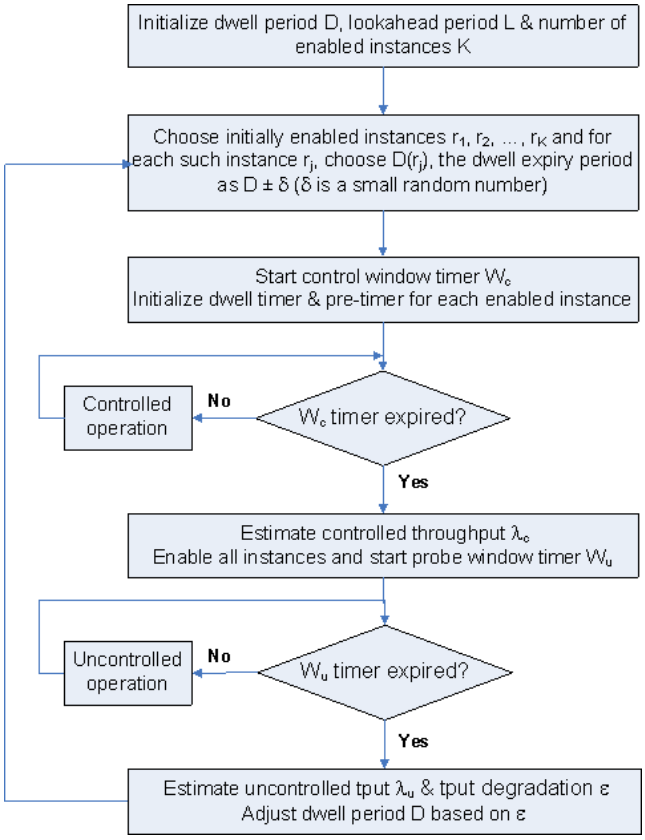

Figure 2: Flow chart for top level description of the algorithm

4.1 Overview of Algorithm

Before stating the algorithm, we start with some notations. We denote the number of instances to be kept eligible at a time as K, and D as the dwell-period, i.e., the time after which we need to make a change to the eligible subset. In our algorithm, D is not a constant, but a parameter that is adjusted dynamically in order to keep the throughput degradation within the acceptable bound ε_t. Generally, ε_t will be in the range of a few percent.

To see why D affects performance degradation, notice that if D is reduced down to zero, each time a request completes on some rank, the memory controller will be able to immediately schedule another request (if any) on that rank. This corresponds to the normal operation of the memory controller. On the other hand, if D is large, the requests waiting on ineligible ranks will experience large delays and hence substantial stalls for the CPU threads.

The memory access behavior of a workload is generally quite complex and any attempt to characterize it based on an of-

fline trace analysis is usually inadequate for the kind of fine-grain control that we are seeking here. Therefore, in order to make our control algorithm applicable irrespective of the workload, we use an online monitoring approach to estimate and react to the throughput degradation. The algorithm is designed specifically to allow inexpensive HW implementation and thus attempts to keep things as simple as possible. In fact, we shall choose parameters such that multiplications and divisions implied in the equations can be implemented via a few shifts.

For the purposes of online monitoring, we divide time into successive windows of size $W_c + W_u$ where W_c denotes the period during which dwell control is effected and W_u the period during which it is not. We call W_u as the *probe period* during which we probe for the full (or un-degraded) throughput λ_c. The choice of W_u is crucial: it should be long enough to let the memory system recover and provide the unrestricted throughput, but it should be short enough so that the power savings are maximized. Generally, we expect W_u to be in the range of a few percent of W_c. W_c itself needs to be chosen carefully. Generally, a longer W_c is desirable for reliable estimate of throughput degradation, but it should not be too long, else the control may lag the workload behavior. Generally, W_c in the range of 1-10 ms is quite reasonable.

Fig. 2 shows a simplified view of the overall algorithm. The notion of lookahead and pre-timer will be explained shortly and can be ignored for now. The dwell-timer implements the dwell-period D discussed above. The outer loop in the figure shows the actions performed over one complete cycle of duration $W_c + W_u$. At the beginning of this cycle, a set of K ranks is chosen as the eligible ranks. Successive cycles rotate this set in order to keep it fair for all the ranks involved. The detailed dwell handling within a cycle is represented by the box "controlled operation" which is depicted in Fig. 3 and is discussed in the next subsection. During a probe period, the behavior is represented by the box "uncontrolled operation". Since this corresponds to the default memory controller operation, it is not explained any further.

Although the algorithm estimates a single value of dwell period D for all ranks, the actual dwell period used by different ranks is perturbed by a few cycles in order to break up the synchronization between the ending of dwell periods of all eligible ranks. Note that this perturbation is required only for the very first dwell in a cycle – subsequent dwells could well use the precise value D. In fact, if the same perturbation were to be maintained for subsequent dwells, it will cause unfairness (i.e., some ranks with longer dwell than others), which is undesirable.

Let N_c denote the number of transactions completed at the resource during the window W_c, and N_u those during the probing window W_u. Then the overall throughput is $\lambda_c = N_c/W_c$, and the unperturbed throughput is $\lambda_u = N_u/W_u$. Given maximum tolerable degradation of ε_t, the remaining problem is then to adjust the dwell period such that the error $\varepsilon = (1 - \lambda_c/\lambda_u) < \varepsilon_t$. In order to reduce jitter in measured throughput, we found it useful to exponentially smooth N_c and N_u values over successive cycles. The flowchart in Fig. 2 does not show this detail for simplicity.

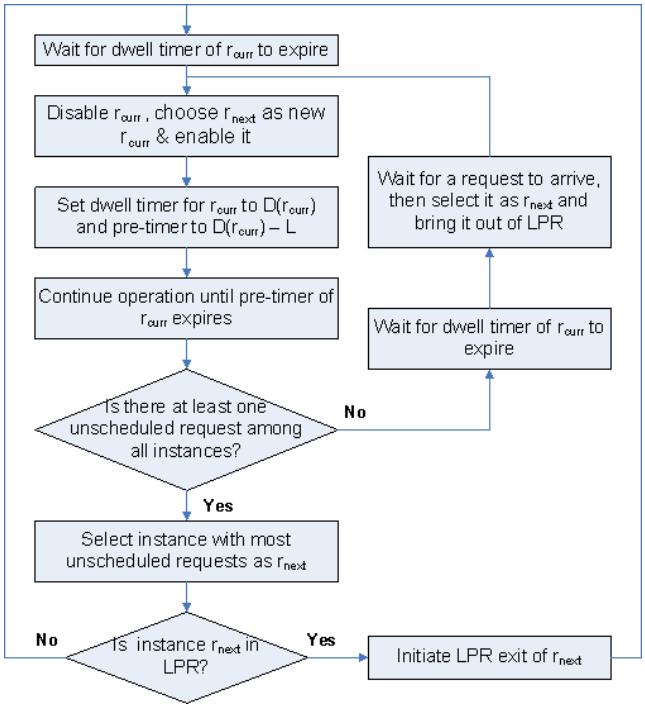

Figure 3: Flow chart for controlled operation of instances

4.2 Dwell Period Adjustment

As stated above, the behavior of the algorithm is controlled by dynamically adjusting the dwell period D. The throughput impact of changing D depends on a variety of factors including inter-dependencies between various requests. For example, an additional delay in completing some memory transactions could stall progress of several other CPU threads, and thereby significantly affect the application throughput. Since our algorithm controls the throughput degradation directly, it effectively converts the memory path latency tolerance of the application into power savings. In other words, the more latency sensitive the application, more power savings it will be able to generate.

In general, the net impact of increasing the dwell period includes several impacts which we discuss here:

1. Accumulation of new requests at non-eligible instances, which amounts to a traffic shaping mechanism that elongates low power periods and thus makes the power management more effective. It also has the side effect of pushing up the concurrency at the time an instance is made eligible for switching. This effect can make the active ranks work more effectively.

2. Reduction in overall instance switching overhead. Dwelling longer on a few instances is likely to lower the rate of switching between ranks and thus reduce the overhead. For example, the memory controller incurs 2 cycle of overhead in switching from one rank to another on the same channel.

3. Since the servers freed up on an ineligible instance must stay idle even if there is a pending request for

them, the net result is reduced processing rate.

One interesting side effect of the dwell scheme is its ability to reduce the impact of low power mode exit latency. Suppose that an instance I_{old} is about to end its dwell period, and we want to replace it with another instance I_{new}. Now if I_{new} happens to be in a low power mode, it needs to exit this mode. Let η denote the exit latency. If I_{old} is supposed to end its dwell at time t, and we initiate the low-power exit for I_{new} at time $t - \eta$, the impact of exit latency is completely hidden! We call this a *lookahead* scheme. To implement this idea, we introduce another timer called *dwell pre-timer* which is also started along with the *dwell-timer*. The dwell pre-timer duration is set to $D - \eta$ where D is the dwell period. When this pre-timer fires, we can select the new instance I_{new} based on the information at that time and start its low-power exit (if it is indeed in a low-power mode).

While *lookahead* is indeed a useful concept, it does have some downside. First, the selection of the new instance I_{new} is based on somewhat stale information. This is usually not an issue unless η happens to be large. The second problem arises when the exit latency η is not a constant. This is always a problem when multiple low power states are involved, eg., CKEf and CKEs in case of memory. Since we need to set the pre-timer before we know about I_{new}, it is not clear if the chosen η should for CKEf (6 ns) or for CKEs (25 ns). The second problem is that the exit latency could even depend on the context. For example, the CKEs exit latency differs depending upon whether the next operation is a read or write. Because of these complications, the lookahead amount (denoted L) an adjustable parameter in the algorithm, and is usually set to the smallest value for all the cases. Fig. 3 shows various details concerning the operation during the control window W_c including lookahead. As stated earlier, this portion of the algorithm represents the "controlled operation" part in Fig. 2 and is excuted once for each dwell period.

Next we discuss the details of adjusting the dwell period D based on the computed throughput degradation value ε at the end of a probe window. The dwell period can be adjusted in a variety of ways, and indeed some experimentation was done before settling on the scheme proposed here. The proposed scheme is motivated by 3 requirements: (a) simplicity of implementation, (b) oscillation (or flip-flop) avoidance, and (c) need for quick adjustments when the workload characteristics change substantially in a short period of time.

In general, the dwell control problem can be viewed as the traditional control system design problem where one could consider P (proportional), PD, PI or PID controllers. However, in the interests of a simple design that even attempts to avoid multiplications and divisions, we instead allow for only two types of adjustments: (a) "normal" – to correct small errors, and (b) "large" – to correct large errors in the control parameter. In particular, the controller has the following 3 rules:

1. Dwell increase: If the throughput degradation is much smaller than the target value, i.e., $\varepsilon < 0.75\varepsilon_t$, then increase dwell by the "normal" amount d_N.
2. Normal dwell decrease: If the throughput degradation is not too far from the target, i.e., $\varepsilon_t < \varepsilon \leq \varepsilon_u$, then decrease dwell by the "normal" amount d_N.
3. Large dwell decrease: If the degradation is becomes too high, i.e., $\varepsilon > \varepsilon_u$, then decrease dwell by the large amount d_H.

where $\varepsilon_u > \varepsilon_t$ and $d_H > d_N$. In all cases, the dwell is not allowed to become negative. The main idea is to reduce the error quickly when it is large. Note that d_N itself should be small enough to avoid ping-ponging around the acceptable range.

To avoid multiplications, both ε_t and ε_u must be chosen appropriately. In particular, we make ε_t a negative power of 2 and choose $\varepsilon_u = 1.5\varepsilon_t$. Also, the condition such as $\varepsilon > \varepsilon_u$ is evaluated as $\lambda_u - \lambda_c > \varepsilon_u \lambda_u$. Both W_c and W_u are also chosen as powers of 2.

4.3 Changing Eligible Set

When a dwell period is about to end, we need to changeover from the current eligible set S of instances to a new one, say, S'. In the above, we specifically avoided any synchronization between the ending of dwell time of currently eligible instances by adding perturbation to dwell amounts. Therefore, S and S' differ by just one instance. In more general cases, it suffices to select new instances of S' one by one. There are several simple ways to choose the new instance from among the currently ineligible ones:

1. Round-robin. Here we simply pick the next higher numbered instance (modulo N) that is currently ineligible. The motivation for this scheme is extreme simplicity. If the chosen instance has no waiting requests, it is skipped over.
2. Longest waiting. Here we pick the instance that has remained ineligible the longest. The motivation here is to avoid starvation as much as possible. If the chosen instance has no waiting requests, it is skipped over.
3. Instance with longest queue of waiting requests. The idea here is to give preference to most heavily used instance. If the chosen instance has no waiting requests, the current eligible set is retained for another dwell window.
4. Instance with most number of schedulable requests. This scheme differs from the last one in that only the requests that are currently schedulable are counted. The motivation here is to maximize concurrency of scheduling. If the chosen instance has no waiting requests, the current eligible set is retained for another dwell window.

It is easy to see that the listed schemes are increasingly more sophisticated. Detailed simulation results (not shown here) clearly indicate that the round-robin scheme does not work well and can be ignored. The longest waiting scheme generally works well since the longest waiting rank is likely to

have the most requests accumulated; however, this is not guaranteed. For long dwell windows, the second and third schemes behave similarly; however, for shorter windows explicitly choosing an instance with more requests works better. Finally, the last scheme is expected to be theoretically optimal, since it serves the instance with the most requests that can start immediately. However, it is complex to implement and its performance is only marginally better than that of the third scheme. Because of this, the results shown in this paper uniformly use the third scheme, as shown in Fig. 3. In order to ensure that no instance is ever starved, we do associate a *starvation timer* with every request. If this timer goes off, the corresponding instance is ineligible, that instance will be picked up at the end of the next dwell period.

5. EXPERIMENTAL SETUP AND MODEL

5.1 Modeling Methodology

Although we described the instance coordination problem in queuing theoretic terms, it is not possible to analyze it mathematically even if the idiosyncrasies of DDR3 memory system and CPU thread complexities are ignored. In particular, the multi-server queuing system shown in Fig. 1 does not represent bank conflicts, memory channel usage by various ranks, and other complexities. Also, no such implementations or detailed cycle-accurate simulators were available or usable for evaluating such a system. In view of this, we evaluated the algorithm by implementing it in a detailed simulator of a 2-socket SMP (symmetric multiprocessor) system called LMPWR [3]. This simulator was designed specifically to study memory and CPU-memory interconnect power management issues, and is quite well suited for this study. This simulator has been used for a number of other power management studies in our organization.

LMPWR has a fairly sophisticated model of link (modeled after Intel's QuickPath^{TM} link) which implements flow control and various transaction types (read, write, remote snoop, remote cache invalidation) along with both reactive and proactive link power control algorithms [2]. It also has a rather sophisticated DDR3 memory model that explicitly models DRAM channels, ranks, banks, interleaving schemes, rank refresh, etc. It also accounts for a variety of latencies inherent in DRAM access including RAS and CAS latencies, asynchronous ODT (on-die termination) overhead, IBT (input bus termination) overhead, rank switch overhead, read-write switch overhead, bank conflicts, channel queuing and usage, transaction blocking for ranks that are being refreshed, etc. The model further implements, in detail, power management of ranks including fast and slow CKE (clock enable) states and switching overheads between them. It also implements both DIMM and channel level register modes.

In order to capture the impact of link and memory subsystem latencies on application performance, it is necessary to model the extent to which the HW CPU threads will stall due to memory path delays. This part is modeled somewhat simplistically, but is quite adequate for relative performance comparisons. Each HW thread executes for a while (depending on the modeled offered load), and then issues a memory transaction which is placed in the thread-level transaction buffer. This transaction stays in the transaction buffer until the memory access request (which could be either to the local memory in the current socket or to the remote memory in the other socket) is completed. If the buffer is not full, the thread still proceeds with execution and subsequently generates another memory request. However, if the transaction buffer becomes full, the thread stalls until at least one of the pending memory transactions is done. Note that in case of a memory write, the transaction buffer is released as soon as the transaction is posted to the (local or remote) memory. That is, writes occupy the transaction buffer for shorter periods than reads.

With the above modeling, the size of the transaction buffer represents the *inherent parallelism* of the workload. In reality, the number of transactions that can be issued before the thread must stall varies depending on the phase of the computation. The model allows for this behavior directly: there is a notion of phase duration and at the end of a phase, both a new phase duration and a new buffer size are chosen according to the specified distributions. The net effect of this model is that the latency sensitivity of the workload can be controlled by appropriately choosing the buffer size distribution. In other words, if we know the latency sensitivity of the workload (as is the case with popular benchmarks such as TPC-C), it is possible to dial that into the model.

The model implements both individual and coordinated control of memory ranks as described here and carefully handles a variety of special situations that arise in handling of memory transactions. For example, whenever the refresh timer of a rank goes off, we either wait for the rank to empty out naturally or if that does not happen soon enough, we stop scheduling more requests so that the refresh can be started without too much delay.

5.2 Model Calibration

We illustrate the coordinated control by using the concrete example of power management of memory ranks across all channels of a socket. We assume DDR3 memory running at 800MHz (i.e., DDR3-1600), with 4 channels per socket and one dual-rank DIMM per channel. We assume a workload with characteristics similar to TPC-C traffic, except that the workload is generated analytically in order to have better control over its properties. The workload assumes that memory transactions arrive according to Zipf distribution (interarrival times are integers) with $\alpha = 2.25$. That is, the probability mass function of interarrival time in terms of underlying clock cycles n goes as $n^{-2.25}$, which is rather heavy tailed.

We assume a 2-socket SMP system with an embedded memory controller in each socket. Each socket generates 50% of the traffic, and 25% of the traffic from each socket is directed to the remote memory. The traffic mix is assumed to be 70% read and 30% write. The two sockets are connected via a 19.2 GB/sec link and have six core CPUs on each side. Each core has two hardware threads. The memory power management capabilities used are fast and slow CKE along with register modes enabled. Because of significant overhead of switching over between fast and slow CKE modes (12 cycles), the CKE mode to use is selected up-front when deciding to put a rank in low power mode. Basically, long idle periods in recent past result in selecting slow CKE as opposed to fast CKE.

The algorithm described above is further refined via a *starvation timer*. Basically, if a memory request is left waiting for more than 240 ns, its rank is immediately made eligible and a *victim* rank is taken out of the eligible set. The victim rank selection is round-robin to avoid punishing any specific rank in particular. The default window size W_c is set as 1024us and the default probe window W_u is set as 32us. Thus the probe window occupies only about 3% of the entire window. The normal dwell change amount d_N is 4 DRAM cycles, and the high dwell change amount d_H is 16 DRAM cycles. By default, the target throughput degradation ε_t was assumed to be 1%. For simplicity, the lookahead amount used was fixed at 4 cycles (exit latency of fast CKE).

6. EXPERIMENTAL RESULTS

Latency sensitivity of the workload is the key factor in determining the power savings potential of power management, whether done independently for each instance or in a coordinated fashion. Consequently, we created three workloads for our testing with "low", "medium" and "high" latency sensitivity. Here latency sensitivity refers to the extent that an additional 1% latency in the memory access path affects the overall throughput. Latency sensitivity was dialed in via the transaction buffer size distribution as described above. The "high" latency sensitivity basically corresponds to TPC-C like workload, which is known to be extremely latency sensitive (roughly 0.5% throughput degradation for 1ns increase in latency). The "low" latency sensitivity corresponds to web workloads that essentially are 10-times as latency insensitive as TPC-C. The "medium" latency sensitivity corresponds to workloads in the middle. It was found that "high" latency sensitivity traffic does not allow for any significant power savings over and above the savings achievable by isolated control. Therefore, we do not show results for this case.

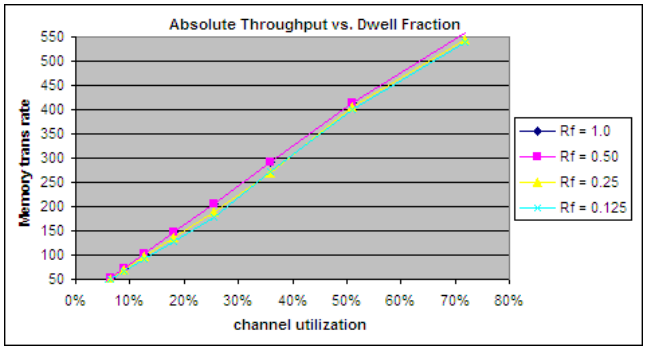

Figure 4: Throughput vs. dwell fraction : Low latency sensitity

As stated earlier, we control all ranks of each socket as a group. We define the quantity *eligible fraction* R_f as the fraction of ranks are kept eligible at a time. Given a total of 8 ranks per socket, the suitable values of R_f in a HW implementation are 1/8, 1/4, 1/2 and 1. $R_f = 1/8$ means that only one rank in a socket is eligible for scheduling at any given time, and thus this represents the situation with the most stringent coordinated control. The value 1 corresponds to no coordinated control whatsoever. Since we are attempting to demonstrate the advantages of coordinated control, the baseline for comparison is the $R_f = 1$ case.

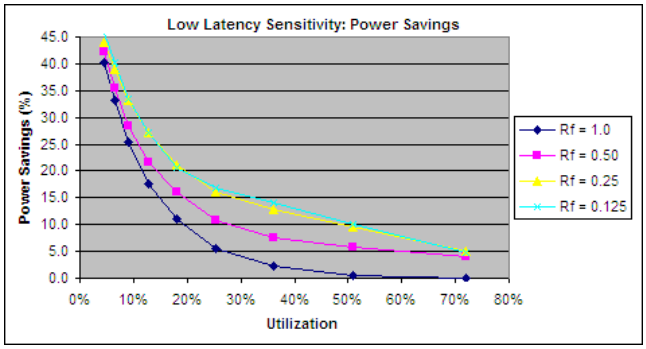

Figure 5: Power savings vs. dwell fraction : Low latency sensitity

Let us start with the low latency sensitivity case. Fig 4 and 5 show the throughput and power savings as a function of channel utilization with R_f as a parameter. At each utilization level, the power saving is calculated by dividing the saved power in watts by the total power consumption (at that utilization) without any power management. It is seen that in all cases, the throughput is maintained close to the optimal throughput. The power savings increase with decreasing R_f. For example, at 25% channel utilization, $R_f = 1/2$ gives 11% power savings whereas $R_f = 1$ (i.e., uncoordinated control) gives only 5% savings. With the unmanaged power consumption of 24 watts in this case, this means that uncoordinated control saves 1.2 watts whereas coordinated control saves 2.64 watts of power. With $R_f = 1/4$, the power savings go up to 16%, or 3.84 watts. At 50% channel utilization, the uncoordinated control comes up with almost no power savings (0.6%) whereas with $R_f = 1/2$ and $R_f = 1/4$ we get respectively 5.8% and 9.2% savings. With unmanged power of about 32 watts, this corresponds to savings of 1.86 watts and 3.04 watts respectively.

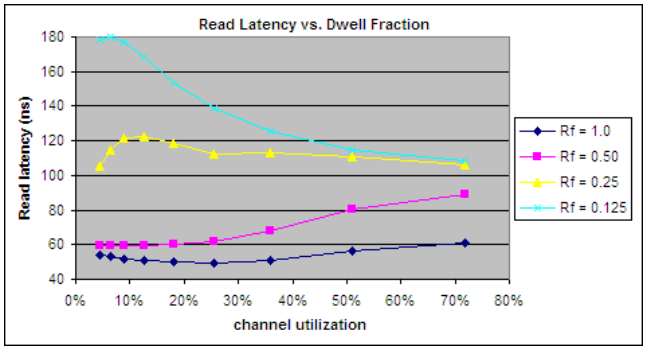

Figure 6: Read Latency vs. dwell fraction : Low latency sensitity

It is thus clear that coordinated control provides additional power savings over and above what is possible with isolated control. As expected, when the channel utilization is rather low, there isn't much difference between the various curves, since in all cases we are able to keep most of the ranks in low power mode. However, at higher utilizations, the coordinated control really shines. Notice that even at 70% channel utilization, coordinated control can provide a substantial advantage over the isolated control. The reason for

this somewhat nonintuitive result is that because of multiple ranks per channel (in this case 2 per channel), the rank utilization is generally much lighter than the channel utilization, and a coordinated control can squeeze out power savings. The final point to note from these figures is that $R_f = 1/8$ and $R_f = 1/4$ behave almost identically. This is because even with $R_f = 1/4$, much of the available power savings have already been squeezed out, and reducing R_f does not help.

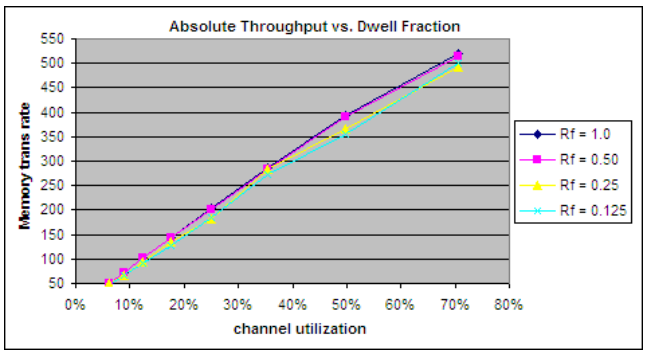

Figure 7: Throughput vs. dwell fraction : Medium latency sensitity

Fig 6 shows the read latency as a function of channel utilization under different values of R_f. It is seen that a tight control (i.e., small R_f) can increse the read latencies significantly. At low utilizations, the impact is most significant and the figure shows that $R_f = 1/8$ results in huge increase in read latency without providing any power savings (as seen earlier). The large latency still does not affect the throughput very much because there is enough slack in the system to tolerate the latencies.

Fig. 6 shows some non-intuitive behavior: for small R_f, the read latency actually goes *down* with increasing utilization. This is a direct result of the coordinated control. At high utilizations, the dwell period is decreased in order to control the throughput degradation and consequently the read latencies actually go down. However, decreasing read latency with utilization is a sign of overcontrol, which again indicates that $R_f = 1/8$ is not desirable.

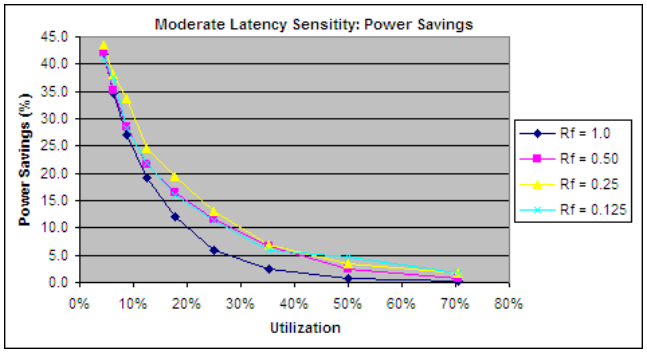

Figure 8: Power savings vs. dwell fraction : Medium latency sensitity

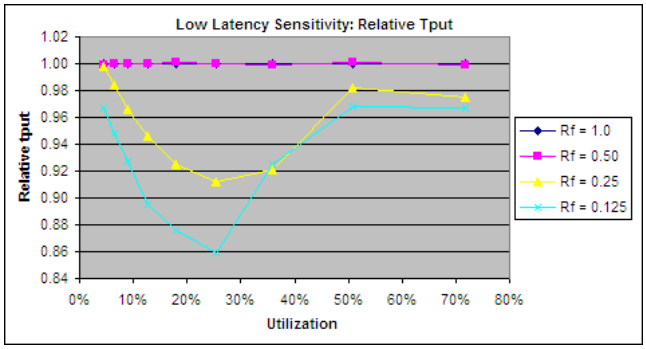

Figure 9: Relative throughput vs. dwell fraction : Medium latency sensitity

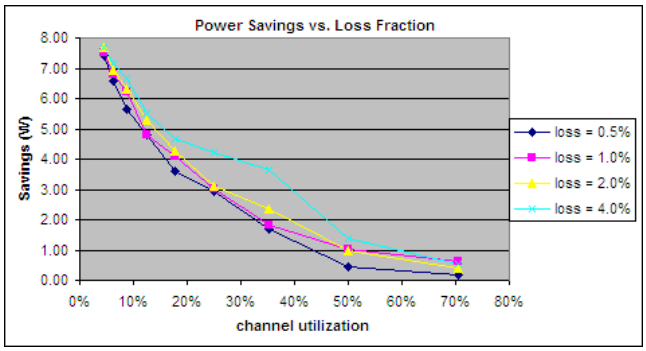

Figure 10: Power Savings vs. loss fraction : Medium latency sensitity

Now we consider similar plots for the medium latency sensitivity case. Fig 7 and 8 show the throughput and power savings as a function of channel utilization with R_f as a parameter. It is seen that in all $R_f = 1/8$ and $R_f = 1/4$ show a clear case of overcontrol and are neither able to maintain the specified throughput degradation, nor provide any additional power savings. Basically, $R_f = 1/2$ in this case has already squeezed out all the power saving that coordinated control could provide, and reducing R_f only does harm. This can be more clearly seen in Fig. 9 which shows relative throughput as a function of channel utilization. The reason that the algorithm is unable to keep the throughput degradation within specified bounds is that the probe window W_u is inadequate to provide adequate recovery of the throughput. Increasing W_u in this case does help bring the throughput degradation closer to the target value (not shown).

Until now we used target throughput degradation as 1%. Fig 10 shows the power savings curves for the medium latency sensitivity case. As before, the x-axis is still the channel utilization, however, various curves are for different throughput degradation targets. For these curves we used $R_f = 1/2$ to ensure that there is no significant overcontrol and the target degradation can indeed be maintained. Not surprisingly, as the target degradation increases, we are able to save more power. In fact, with the emphasis in computer systems shifting from performance centric to power centric, a 4.0% throughput degradation is not unreasonable, and can provide significant additional power savings.

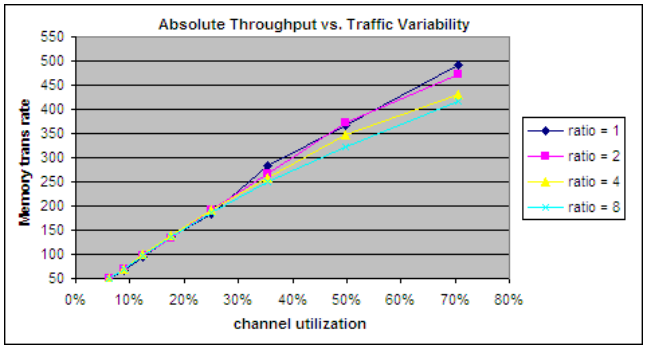

Figure 11: Throughput vs. traffic variability : Medium latency sensitity

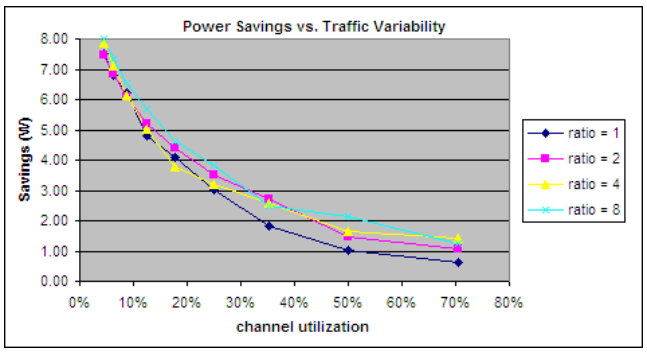

Figure 12: Power savings vs. traffic variability : Medium latency sensitity

Finally, we examine the impact of traffic characteristics on the performance of the coordinated control. For this, we used a phase type arrival process model with 3 "macro-states", each with a different traffic intensity. We controlled the traffic burstiness via a parameter called "ratio", which is the ratio of traffic intensity in successive macro-states. Obviously, a ratio=1 represents the normal case where the traffic intensity does not vary across macro states. The case with ratio=8 represents a situation where the ratios of traffic intensities in the 3 states are 1:8:64. Obviously, this is an extremely bursty traffic. It is seen that a highly bursty traffic makes the throughput control difficult since a small probe window W_u will be unable provide adequate recovery to reflect true unperturbed throughput. Obviously, a larger W_u will help here. In terms of power savings however, higher burstiness does show a somewhat better behavior, although we should be a bit careful in the interpretation. For $ratio = 2$, the throughput is well maintained and the increased power savings are depicted correctly; however for lower R_f, the results are somewhat spurious since the throughput degradation exceeds our target.

7. CONCLUSIONS

In this paper, we examined the coordinated control of multiple instances of resources by using a simple algorithm that attempts to control throughput degradation and thereby converts the latency insensitivity of the workload into power savings. Because of the closed loop control, the algorithm does not require any a priori knowledge of the workload characteristics and can, in fact, easily handle workloads whose characteristics change dynamically. It is seen that the coordinated control can provide power savings over and above those provided by isolated power control. It was shown that the algorithm does need to be parameterized carefully so it is able to keep the throughput within the specified limits. This means that (a) the eligible fraction should not be made too small – a value of 1/2 works reasonably well, and (b) the probe window W_u should be large enough to allow the workload to be able attain unperturbed throughput.

The results show that it may be desirable to "tune" the eligible instance fraction R_f to the characteristics of the workload. We expect changes in R_f to be made based on measurements over a rather long period so that the algorithm does not experience instability. These and other parameter settings for the algorithm are interesting things to explore in the future. Other things to examine are coordinated control of dissimilar resources and an architecture for controlling them.

8. REFERENCES

[1] L. Benini and G. Micheli, " System-Level Power Optimization: Techniques and Tools", ACM Transactions on Design Automation of Electronic Systems, Vol. 5, No. 2, pp. 115-192, 2000.

[2] — and —, "Proactive vs. Reactive Idle Power Control", Proc. of DTTC, Aug 2008.

[3] —, "LMPOWER – A Comprehensive Link-Memory Power Management Simulator", Available from author (web-site not disclosed for anonymity).

[4] V. Delaluz, M. Kandemir, et al., "DRAM energy management using software and hardware directed power mode control", Proc. of HPCA 2001, pp159-170.

[5] V. Delaluz, A. Sivasubramaniam, M. Kandemir, et al., "Scheduler-based DRAM energy man- agement", Proc. of the 39th Conference on Design Automation, pp697-702, 2002.

[6] B. Diniz, D. Guedes, J. Wagner Meira, and R. Bianchini, "Limiting the power consumption of main memory", Proc. of ISCA, pp290-301, 2007.

[7] X. Fan, C. Ellis, and A. Lebeck, "Memory controller poli- cies for DRAM power management", Proc. of 2001 Intl. symp. on Low-Power Electronics and Design, pp129-134, 2001.

[8] W. Felter, K. Rajamani, T. Keller, and C. Rusu, "A performance-conserving approach for reducing peak power consumption in server systems", Proc. of 19th Intl. conf. on Supercomputing, pp293-302, 2005.

[9] H. Huang, K. G. Shin, C. Lefurgy, and T. Keller, "Improving energy efficiency by making DRAM less randomly accessed", Proc. of 2005 Intl. symp. on Low-Power Electronics and Design, pp393-398, 2005.

[10] I. Hur and C. Lin, "A Comprehensive Approach to DRAM Power Management", Proc. of HPCA 2008.

[11] I. Hur and C. Lin, "Adaptive History-Based Memory Schedulers", Proc. of 37th Annual IEEE/ACM Intl. symp. on Microarchitecture (MICRO'04), pp343-354, 2004

[12] S. Irani, S. Shukla, and R. Gupta, "Online strategies for dy- namic power management in systems with multiple power-saving states", ACM trans. on

Embedded Computing Systems, 2(3), pp325-346, 2003.

[13] C. Isci, A. Buyuktosunoglu, C-Y Cher, et al., "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget", Proc. of IEEE/ACM Intl. symp. on Microarchitecture (MICRO'06), 2006, pp.347-358,

[14] X. Li, Z. Li, F. David, et al., "Performance directed energy management for main memory and disks", Proc. of 11th ASPLOS conf, pp271-283, 2004.

[15] C. Lefurgy, K. Rajamani, F.L. Rawson, et al., "Energy Management for Commercial Servers", IEEE Computer, Vol. 36, No. 12, pp. 39-48, Dec 2003.

[16] J. Luo, N. Jha, Li-S Peh, "Simultaneous dynamic voltage scaling of processors & communication links in real-time distributed embedded systems", IEEE Trans on VLSI, 15, 4, April 2007.

[17] C.-G. Lyuh and T. Kim, "Memory access scheduling and binding considering energy minimization in multi-bank memory systems", Proc. of the 41st Annual Conf on Design Automation, pp81-86, 2004.

[18] S.A. McKee, W.A. Wulf, et. al., "Dynamic Access Ordering for Streamed Computations," IEEE Trans. on Computers, 49, 11, pp1255-1271, Nov. 2000,

[19] A. Papathanasiou and M. Scott, "Energy Efficiency through Burstiness", Proc of the 5th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'03), pp. 44-53, Oct 2003.

[20] V. Raghunathan, M.B. Srivastava, and R.K. Gupta, "A survey of techniques for energy efficient on-chip communication", Proc. of 40th Conference on Design Automation, 2003.

[21] V. Venkatachalam and M. Franz, "Power Reduction Techniques for Microprocessors", ACM computing surveys, Vol 37, NO 3, Sept 2005, pp 195-237. (http://www.ics.uci.edu/ vvenkata/finalpaper.pdf)