Energy-aware Memory Management through Database Buffer Control

Chang S. Bae
Dept. of EECS
Northwestern University
Evanston, IL 60208
cbae@u.northwestern.edu

Tayeb Jamel
Intel Corporation
211 NE 25th Ave
Hillsboro, OR 97124 USA
jamel.tayeb@intel.com

ABSTRACT

Energy consumption in modern data center trends to be increasing, which gives pressure to limit power consumption in servers. There is an opposite side trend that needs more memory resources to load increasing application data. Once a system is designed with capacity for peak time usage, it inefficiently consumes more energy in an idle state. So in this paper we propose a dynamically adaptie application-driven memory management scheme. Our example is on a database application with an online transactional workload. We design and implement a simple heuristic scheme, which determines the state of either expanding or shrinking the size of memory. Our experimental results show that significant amount of energy saving (4–8%) is achieved without any visible performance penalty. Although we acknowledge that the overall amount of energy reduction varies to the amount of memory in a system, we believe that this is equally achievable over high-end server machines that are similar to our test machine.

1. INTRODUCTION

Energy-aware memory management has importance and gives significant opportunity to reduce energy consumption in a system. As there are more computation resources available in a modern machine, more data is loaded to increase system throughput. Thus, memory modules become one of major sources of energy consumption. However the amount of data being loaded varies on its demands. So, it is quite inefficient to maintain whole memory slots on-line when there is significantly less data needed. For this reason, it should be useful to power down or power off some of DIMMs when 1) they are not used and 2) removal of them does not affect performance.

In this paper, we present our application-level approach to dynamically adjust amount of on-line memory. It addresses how to find both two conditions in a very straightforward way. Particularly, we claim that application-level approach can be best to efficiently and accurately capture the amount of data it needs rather than lower-level approach.

We designed and applied a simple heuristic approach to one of most popular enterprise workloads, an online transactional workload with database engine. This type of application spends most memory resource caching data to memory (buffer) from disk. So our scheme is essentially required to guide the amount of data being cached. To do so, we measures two key metrics, *Bhit* – percentage of buffer block hit per request and *Butil* – amount of buffer currently used. Based on threshold value of empirically measured data, state transitions are triggered. Each state is defined with the attribute for buffer size. If an adjustment of buffer size is needed, the application automatically changes its memory usage and accordingly the OS reconfigures the power mode for DIMMs. In this way, we conduct our experiments and the result shows significant energy-savings while it maintains sufficient performance (response time).

The structure of this paper is as follows: Section 2 clarifies our motivation. Section 3 argues our application driven approach. Section 4 describes a simple heuristic design for adaptive buffer control. Sections 5 and 6 illustrate experimental setups and the results for each.

2. MOTIVATION

The Intel Nehalem EX platform offers a feature called dynamic memory power management [10, Chapter 11.2.1]. The system can manage the physical DIMM's power draw. We conduct a preliminary study to check the opportunity of energy savings from power-down mode in memory. In *Default* case, the power mode of the DIMM is configured as default where we expect no power savings in memory. The other case, *Powerdown*, drives power-down mode for half of the DIMMs in a test system, as shown in Figure 4. Then we compare the energy and performance of these two cases, while it runs online transaction workload, TPCC-UVa (more detail can be found in Section 5.2). As Figure 1 shows, *Powerdown* case achieves about 8% energy savings over *Default*. However, its average response time under *Heavyweight* takes longer than *Default* case. Detailed descriptions on this can be found in Sections 4.2 and 5. It is clear that unconditional reductions of on-line memory size is not always the correct approach. From this test, the following two points can be summarized.

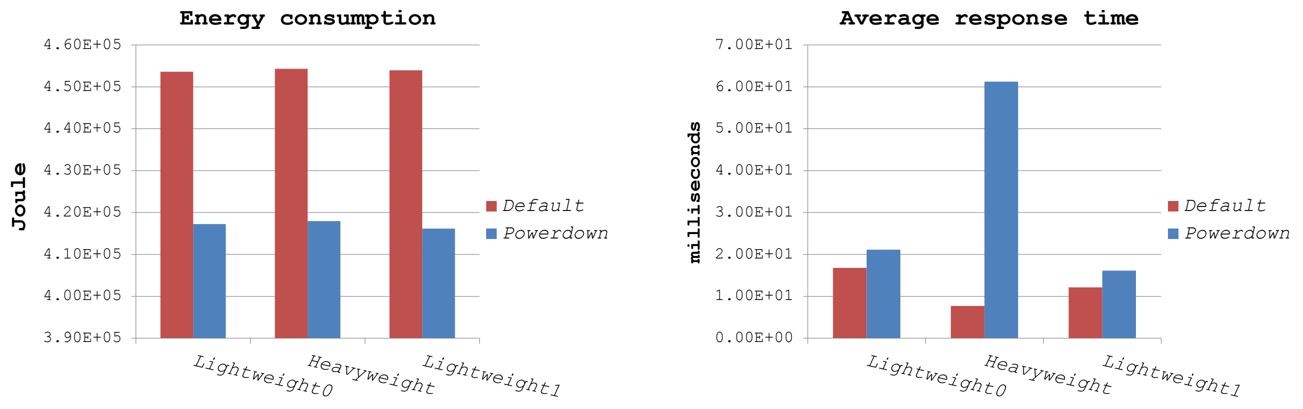


Figure 1: Although *Powerdown* mode achieves significant energy saving, it shows significant performance degradation.

- Considering our test machine is one of the currently available server machines, the results show that there is a compelling opportunity to reduce system energy consumption from memory power-down mode.
- At the same time, the results give motivation to investigate the autonomous control scheme for managing power mode of memory. We expect it first figures out memory requirement for the workload and then adjusts the memory size. Accordingly, it then finally controls power mode for each DIMM.

3. APPLICATION DRIVEN POLICY

Hardware vs Software. To save energy, software can easily leverage its knowledge of the work it does and the data it is processing, while it still needs the actuator supported by hardware. This hardware feature reacts upon the system's power draw. In many cases, this hardware-based logic needs measurements to figure out what happens in software. However, whatever hardware can detect through measurements, it is quite possible that it causes delay, where the software action may have already elapsed. In some instances, software-based policies can be more accurate than hardware-based approaches.

Application-level approach. Furthermore, we argue that an application-level would be better than a lower-level in the software layer. It is possible to infer memory requirements from middleware or the OS, however this is less efficient since memory utilization is one of key knowledges the application knows best. This idea is also similar to [9], which gives lessons on database design and implementation using hardware more directly because paging caches and schedulers make it hard to design efficient database system. While allowing the application to decide memory power mode, the programmer does not have to be aware of the physical layout of data in memory because we can isolate direct access using a marshalling layer and/or a wrapping API. We also assume one dominant application runs in a server system. Therefore, in this work, we are going to demonstrate a case with design and implementation of application-level adaptive scheme in following Sections.

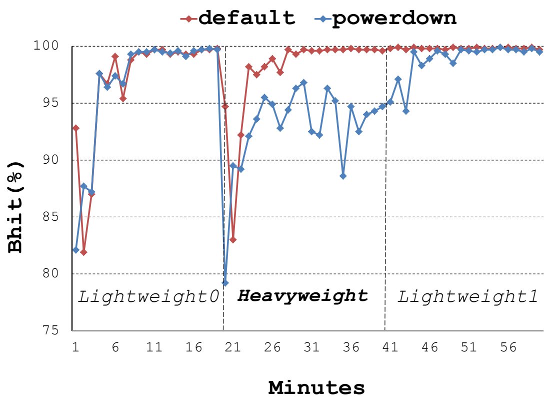
4. BUFFER POOL CONTROL

Database (DB) applications allocate most of its memory usage for buffer that caches DB data. So controlling of the size of the buffer directs this application's memory consumption. To get the best knowledge on appropriate size of buffer, we use a couple of metrics, buffer hit rate (*Bhit*) and buffer utilization (*Butil*). Having these, we designed and implemented a simple heuristic approach, as shown in Section 4.2. Again, it is important to remind that our motivation comes from the effort to preserve the performance while achieving energy savings. This heuristic tries to provide sufficient resources to the workload as required. We also admit that further extension, discussed in Section 7, is possible for its wide applicability.

4.1 Metrics

Each metric hints at a different direction of resizing, increasing or decreasing the buffer pool. Again, the buffer is expected to cache data to reduce the cost of IO operations. So how many IO operations executed and how many of cached blocks are reused provide the basis for configuring the amount of buffers.

Buffer hit rate (Bhit). This metric, similar to the processor cache hit/miss ratio, indicates the percentage of successful look-ups in buffers for requested data (the higher the better). So, if hit rate (*Bhit*) goes down like *Powerdown* in Figure 2 (left), it is good to increase amount of buffers to avoid IO



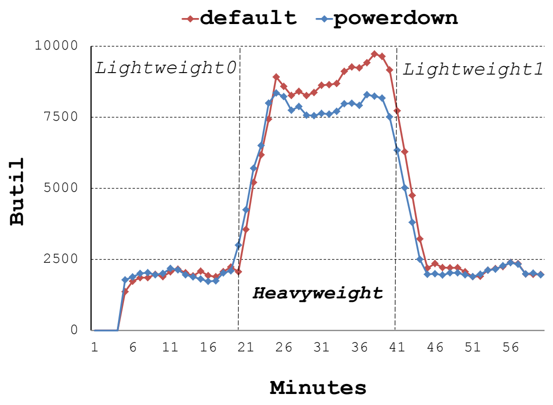


Figure 2: *Bhit*(left) represents buffer hit ratio per request. *Butil*(right) shows 5-point moving average of samples and each sample captures the number of buffer being accessed per minute, so this metric reflects the usage of buffer pool

operations. In this case where half of memory is unused (mentioned in Section 2), phase of *Heavyweight* load does not fit into allocated buffer size. It is useful to increase buffer at this point.

However this metric does not address when is affordable to reduce buffer size. In *Default* case, performance-wise there is no problem maintaining the same buffer size over *Heavyweight* and *Lightweight1* phases. Considering that *Powerdown* mode recovers its *Bhit* during *Lightweight1* phase, it is advisable to reduce the buffer size to save energy. Analyzing only this metric does not give sufficient information when to decrease buffer size, while it is useful to indicate when to increase them.

Buffer utilization (Butil). We use another metric, presents the usage of buffer blocks. It is defined to count number of buffer blocks accessed for request over a certain period of time. It can capture when the cached blocks are significantly less used. At this point, it is possible to free unused buffers. As expected, Figure 2 (right) reflects the transition between *Heavyweight* and *Lightweight1*. It is important to note that this is not detectable from *Bhit* alone.

4.2 Heuristics

Shrink vs Expansion. In our heuristic approach, to make it simple, we assume that there are only two set of buffer sizes, Buf_{shrink} and $Buf_{expansion}$. Each of buffer size is mapped to two high-level states, **Shrink** and **Expansion** as shown in Figure 3. These two defined buffer sizes are configured to have the following relationship with the amount of data load that the DB has during two phases, as shown in Figure 5. Note that $Load_{light}$ and $Load_{heavy}$ represent amount of data loaded during *Lightweight* and *Heavyweight* separately.

$$Load_{light} \leq Buf_{shrink} < Load_{heavy} \leq Buf_{expansion}$$

Unstable vs Stable. Each phase in Figure 2 (left) shows the fluctuation of hit rates at the beginning of each phase. Changing the data set suffers multiple temporal buffer misses. Eventually it gets stabilized, since the DB engine places frequently accessed blocks in its buffer pool. However, note that *Powerdown* under *Heavyweight* load still has quite a large fluctuation due to frequent misses, although it is less than at the beginning of this phase. Based on this observation, in each of the high-level states, it begins with `Unstable` state. Once it is determined to be stabilized, it moves to `Stable` state.

Threshold based triggering. Overall, transitions over different states are triggered by the threshold value of two metrics, (*Bhit* and *Butil*). Conditions for state transitions in Figure 3 are described below.

1. (`Unstable`→`Stable`)/**Shrink**
2. (`Unstable`→`Stable`)/**Expansion**

   Both transitions (1 and 2) are driven by this condition, $\Delta Bhit < 0.1\%$
3. `Stable`/**Shrink**→`Unstable`/**Expansion**: *Bhit* $< 99\%$
4. (`Stable`→`Pre-shrink`)/**Expansion**: buffer size is assumed to be enough for current load, if *Bhit* $\geq 99\%$
5. `Pre-shrink`/**Expansion**→`Unstable`/**Shrink**: shrunk buffer size, if *Butil* $< threshold$,
6. `Stable`/**Shrink**: $threshold \leftarrow max($*Butil*$)$, while staying in `Stable`/**Shrink**

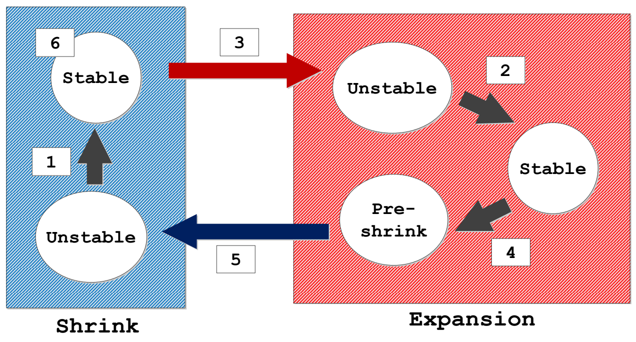


Figure 3: Control diagram for heuristic approach.

4.3 Detailed implementation

Given the policy to determine buffer size, it needs to: 1) reallocating memory for buffer and 2) adjust memory power mode accordingly.

Back-end buffer resize. We modified the backend of PostgreSQL since this DB engine does not support resizing buffer pools during runtime. We made it possible to attach/detach extensional shared memory to/from pre-allocated buffers, Buf_{shrink}. DB server begins with Buf_{shrink} as the size of buffer. With adaptive scheme, when the state changes to **Expansion** from **Shrink** state, then it connects additional buffers to the end of the current buffer pool. Now its overall buffer size is equal to $Buf_{expansion}$.

Control power mode with exported counter. The control of physical DIMMs power level is not performed directly from the DB server. Hence, it is necessary to export the value of the current state in adaptive scheme so that the OS or middleware can reactively control the power mode accordingly. We export the state value by instrumenting PostgreSQL with Intel Energy Checker SDK [10].

5. EXPERIMENTAL SETUP

5.1 Hardware

Out test machine, described in Figure 4, is an Intel Nehalem EX platform which offers a feature called dynamic memory power management. This hardware feature is an essential element to power draw savings. Here is detailed description on this technology.

DIMM power control support. System having this technology can manage their physical DIMMs' power mode. In short, the BIOS and the hardware allow the positioning of DIMMs' power draw to various levels, including an unpowered state. Note that, in our experiment, we use two levels, active mode and power-down mode, although it is possible to save more energy by using unpower mode. In fact, there are some drawbacks when this feature is used, [10, Chapter 11.2.1]. In the scope of our transactional workload, the delay from changing power mode is quite small. It does not translate into any performance degradation. However, it is also true that the lower the power mode, the longer the resume time is. For applications where memory latency is important, delay can be mitigated by avoiding power-off mode and/or by pre-fetching data before accessing the memory.

Processor	Intel Xeon X7560 - 2.26GHz 64 cores/4 chips/16 cores per chip 4 sockets
Chipset	Emerald Ridge Gold Baseboard - Quanta QSSC-S4R BIOS - BIOS 26, BMC 17, FRU 10, HSC 2.14, ME 1.83
BUS	Intel QPI 6.4 GT/s
Memory	Hynix PC3-8500R DDR3 ECC Memory 128GB (32x4) 64 DIMM slots (8 on each of 8 memory risers)
Power supply	Delta Electronics 850W DPS-850FB
Measurement	Yokogawa WT210

Figure 4: Features of test machine. Intel Software Development Platform was used.

OS modifications: To manipulate memory address space easier, we need to use a modified kernel so that page allocator can linearly allocate memory pages to physical DIMMs. In a NUMA machine, this modified kernel allocates consecutive virtual address linear to physical address in a given node. Assuming there is one dominant application running in a server system, side effects of sequential allocation such as fragmentation are expected to be limited.

5.2 Software

As mentioned in Section 5.2, we select an online transactional workload, TPCC-UVa [8], ver 1.2.3. Interfacing this workload, PostgreSQL [2], ver 8.4.6, runs as a database server. Buffer management policy and mechanism as described in Section 4 are applied to PostgreSQL. Along with this workload, RedHat Enterprise Linux 5.5 runs on our test machine. As mentioned, we used modified kernel version 2.6.28.8. Lastly, to measure energy consumption, a Yokogawa WT210 Power Analyzer and Intel Energy Checker SDK [10] for online logging are used.

Workload modification. We actually modified the workload, TPCC-UVa, because this benchmark originally targets performance measurement with fixed amount of data load. Rather than this, we configured two sets of data load, *Lightweight* and *Heavyweight*, by varying the number of warehouses. So in runtime the workload has three different phases as shown in Figure 5. It is important to know that, in our scenario, higher query rates bring more data accesses, since each of warehouse accordingly has 10 accessing terminals and 100MB of data size. This setup comes from our basic assumption that amount of data load changes during time of day, however, we do not assume a linear relationship between the query rate and the amount of data accessed. Nonetheless, asymptotic linearity appears in the real world. For metrics, we chose the average response time as performance metric, since performance (transaction per second) between two cases are naturally different. Note that performance under *Powerdown* mode is distinctly degraded. In *Powerdown* mode in Figure 2, there is more than 5 percent of consistent buffer misses which is enough to impact even the 95th percentile response time of *Powerdown* mode over *Default* mode.

Threshold and window size. As shown in Figure 2, we measure both buffer hit rate (*Bhit*) and the number of buffers being touched (*Butil*) once each minute. Measuring more often than this, a high degree of variations are observed. Even though, we realize that *Butil* needs smoother by taking a 5 minutes moving average. $threshold$ value should be taken in runtime, but in this case as Figure 2 its value went around 2500.

NUMA affinity. In its resource mapping of our setup we set affinity for DB manager, PostgreSQL, to a single socket. The rest of the resources in other sockets are assigned for TPCC-UVa workload which is interfacing PostgreSQL, and both are used for evaluations. In the meantime, we do not consider any NUMA-awareness in our design.

Buffer size expansion. As described earlier, we modified PostgreSQL to dynamically support additional buffer pool as an extension. Once this additional shared memory space is added, it incrementally touches the memory of the extended buffer pool as it uses them. In fact real allocation is made by Linux when it actually touches memory. However, when state changes to Shrink from Expansion, the DB application needs to free all of the expanded buffers at once. Hence, this expansion region should be configured small enough for this sudden deallocation behavior. For this reason, the amount of data for this increased number of warehouses is also limited to be around 1GB. Note that following paragraph describes where each buffer pool is located within the memory system. Meanwhile, if having multiple expansion regions, each of which is large enough to tolerate sudden deallocation, our scheme and implementation is applicable for large amounts of memory as well.

Memory allocation. Within a single node, Buf_{shrink} is placed in the first half of memory address and additional extended buffer(s) are located in next half of memory. Each of this half of memory in a socket is equal to the amount of memory in one memory riser shown in Figure 4. Note that Buf_{shrink} is suite for only one warehouse data, 100MB, and

Phase	Duration (mins)	Number of warehouse
Lightweight0	30	1
Heavyweight	30	11
Lightweight1	30	1

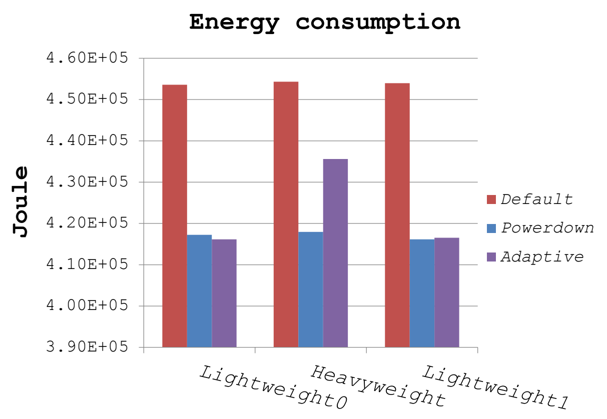
Figure 5: Configuration of workload.

the modified kernel allocates memory sequentially. Hence, we need to fill out the first half of memory in a socket with dummy data (all 0 or 1 value) so that extended buffer pool goes into very next half. Nonetheless, we still believe that it is scalable with multiple expansions. One possible extension of our scheme will be discussed in Section 7. We changed the power mode in the second memory riser of each and every socket(s), while the power mode of DIMMs in the first memory riser are not changed ever.

Turning off page caching. Besides the buffer pool, the DB engine takes advantage of page caching managed by the OS. If there are any misses in buffer, DB process requests searched blocks to the OS. Then the OS looks up on page caching, if found, the found blocks from page caching are returned, preventing IO accesses, unless it needs to conduct IO processes to draw searched blocks from disks. In our experiments, we discard this page caching in memory for simplicity and focus on the effects of buffer controlling. To our knowledge, unless the kernel source is modified, it is only possible either to turn off or to leave the OS allocating whole remained memory for paging cache. In fact turning off option we applied lets OS to keep flushing page caching continuously.

6. RESULTS

With our heuristic scheme, we expect it is possible to adaptively adjust memory usage efficiently with decent performance. The case with adaptive control is named as *Adaptive* mode. Figures 7 and 6 show the overall results and they satisfy our goal. *Adaptive* mode holds energy saving as much as 8 percentage from *Default* mode, which is the same amount that *Powerdown* mode saves over all phases. *Heavyweight* in *Adaptive* mode is the case adaptive scheme autonomously extending buffers by turning on one memory riser in a socket, which saves 4% of energy consumption. However, contrast to *Powerdown* mode, *Adaptive* mode maintains almost same response time as *Default* mode. Performance target for *Adaptive* mode is near to the performance of *Default* mode that has enough memory. Again, it is important to know that this adaptive scheme is very applicable for energy savings, although the overall energy savings vary by the portion of energy consumed by memory modules in the system.



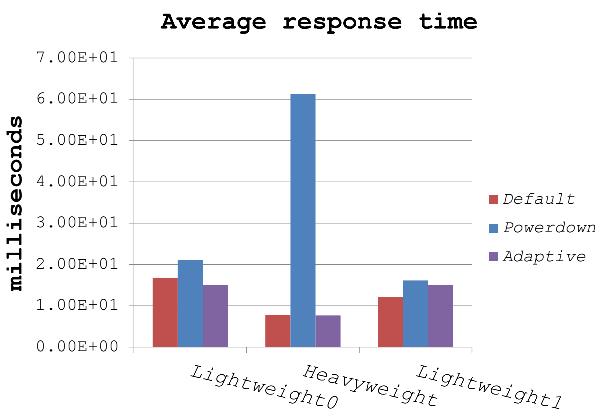


Figure 7: Experimental result by *Adaptive*: it achieves significant energy saving, while maintain the same response time as *Default*

Phase	*Powerdown*	*Adaptive*
Lightweight0	-8.02 %	-8.25 %
Heavyweight	-8.00 %	-4.12 %
Lightweight1	-8.33 %	-8.25 %

Figure 6: Energy reduction percentage over *Default*

7. DISCUSSION AND FUTURE WORK

Some practical amendments can help a smooth deployment in real enterprise systems. In this sense, we discuss some possible issues and bring up potential future works.

Opportunities on energy-saving. Since we gave first priority to the performance, particularly response time, opportunities to reduce energy consumption may be limited. If there are consistent and well distributed memory accesses, it is hard to unload even one portion. However, given that some users tolerate and allow the penalty in terms of increase in response time under peak loads, other opportunities to save energy are possible.

Comparison to OS- or hardware-based techniques. We need to compare our application driven approach to other techniques for other levels such as OS or hardware level inference. It helps to understand each technique in practical applications and demonstrate which is best. In particular, quantification of benefits over exhaustive inferences leverages its knowledge when best to apply.

Extension for applicability. Although the result is very satisfactory, following immediate extensions are commensurate.

Stuck at unstable state: One extension would be drawing direct transition to Unstable state of Expansion from Unstable state of Shrink, when it is considered stuck at Unstable state. Once this transition is made with omitting learning phase to set $threshold$ value, inference work should proceed to find the $threshold$ while staying Expansion state. Simply, when the order of phase for the load switches to go to *Heavyweight* first, it hangs in Unstable of Shrink state. Whereas, it should be fine to remain in Unstable state under the maximum size of the buffer pool that the system memory affords. Inference for $threshold$ is challenging work. Still, *Butil* is valid metric to monitor, but ideally it should do at least one oscillation of shrink and expansion to estimate appropriate the $threshold$ value.

Multiple level of expansions/shrinks: To have scalability in large amounts of memory, it helps to maintain multiple levels of expansion in buffer pools. Continuous low buffer hit ratios enforce further buffer allocations through multiple expansion stages. If it goes through Stable states in certain level of buffer expansion, the $threshold$ for return is explicit. Otherwise, potential inference should be investigated as mentioned above.

Oscillation: Oscillation behavior can possibly happen if the query rate changes in a nondeterministic way, and more importantly if repeated expansion and shrinkage equal additional overhead to pay. One way to control this is through adjustment of $threshold$ values to conservatively shrink the buffer pool. Oscillation behavior is sensitive to the distance between $threshold$ values in each level when maintaining multiple levels, which is also related to this matter, what is appropriate granularity.

Power proportionality and granularity of management. Managing fine-grained domain of buffer pool itself provides a degree of power proportionality [3] and further benefit, while coarse-grained memory deallocation happens abruptly. Although, it has to figure out a reasonable size of domain that does not trigger frequent oscillation as mentioned above and should be better if aligned with power domain as piece of hardware (such as DIMM) in the memory system.

Meanwhile, provided with a dedicate core to release memory and additional power saving techniques in Section 8, certain level of a coarse-grained buffer management is also a possible choice. While releasing chuck of memory is being done by one dedicated core, operation of service may proceed. The result of abrupt energy reduction can be relaxed by employing other power saving techniques as well.

Page caching control and NUMA awareness. The OS needs energy-aware page caching controllability. Applications might need to export some meaningful counters down to the OS, so that it can dynamically manage page caching size. NUMA awareness is another point to consider. In a NUMA machine, there is additional hierarchy in the memory system which makes it complicated when managing energy-aware buffer pool control. So it incurs reconsideration of our design. Nevertheless, above extensions are not always required for deployment in such a case when continuous flushing on page caching is taken as sufficient option and server consolidation goes on NUMA machine with virtualization.

8. RELATED WORK

Hot plug memory [1] implements a functionality in Linux kernel to attach or detach regions of memory during runtime. While this idea is very close to and helpful for our work, we use our modified Linux kernel as mentioned in Section 5.2, because in some cases its off-line functionality fails. So this is not yet completed work as far as we know. [7] proposes adaptive history-based scheduling in the memory controller. This work focuses on architecture level memory power management. We did not directly compare this to our work. However our hypothesis is that hardware cannot know what software knows. Even a smart DIMM will always react to history data or some data provided by the OS. Only software can project itself at best and tell what it does and when it happens at minimum. [6] addresses energy efficiency in DRAM by reshaping DRAM access pattern. This work is another interesting approach to save DRAM power. Although, we still believe it is more compelling to unpower or deep power-down memory selectively. [5, 4] point out a DVFS-based memory power management scheme. This approach is also related to ours in terms of its power mode control. Note that we aim at aggressive unpower mode rather than power-down mode, although our results are from power-down mode because the test machine does not support unpower mode when we conducted our experiments.

9. CONCLUSION

In our work, we first indicated a chance to save energy by changing the memory power mode, and identified the opportunity of an adaptive scheme to control memory usage. In our approach, we claim that an application driven style has increased efficiency over a bottom up inference. Having this, we take a heuristic approach and design/implement an adaptive scheme. Our experiments show one case where it effectively manages memory with energy- and performance-awareness, which validates our approach. Although only one example is shown in this work, we believe it is quite possible to use this application driven energy-saving approach to other workloads as well.

10. REFERENCES

[1] Linux memory hotplug support. webpage http://lhms.sourceforge.net/.

[2] Postgresql. webpage http://www.postgresql.org/.

[3] Dennis Abts, Michael R. Marty, Philip M. Wells, Peter Klausler, and Hong Liu. Energy proportional datacenter networks. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 338–347, New York, NY, USA, 2010. ACM.

[4] Xi Chen, Chi Xu, and R.P. Dick. Memory access aware on-line voltage control for performance and energy optimization. In *Computer-Aided Design (ICCAD), 2010 IEEE/ACM International Conference on*, pages 365 –372, nov. 2010.

[5] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. Memscale: active low-power modes for main memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 225–238, New York, NY, USA, 2011. ACM.

[6] Hai Huang, Kang G. Shin, Charles Lefurgy, and Tom Keller. Improving energy efficiency by making dram less randomly accessed. In *Proceedings of the 2005 international symposium on Low power electronics and design*, ISLPED '05, pages 393–398, New York, NY, USA, 2005. ACM.

[7] Ibrahim Hur and Calvin Lin. A comprehensive approach to dram power management. In *International Symposium on High-Performance Computer Architecture*, pages 305–316, 2008.

[8] Diego R. Llanos. TPCC-UVa: an open-source tpc-c implementation for global performance measurement of computer systems. *SIGMOD Rec.*, 35:6–15, December 2006.

[9] Michael Stonebraker. Operating system support for database management. *Commun. ACM*, 24:412–418, July 1981.

[10] Jamel Tayeb, Chang S. Bae, Cong Li, and Stevan Rogers. *Intel Energy Checker SDK User Guide*. Intel, rev2 edition, December 2010.