MicroRefresh: Minimizing Refresh Overhead in DRAM Caches

Nagendra Gulur
Texas Instruments
nagendra@ti.com

R. Govindarajan
Indian Institute of Science
govind@serc.iisc.ernet.in

Mahesh Mehendale
Texas Instruments
m-mehendale@ti.com

ABSTRACT

DRAM memory systems require periodic recharging to avoid loss of data from leaky capacitors. These refresh operations consume energy and reduce the duration of time for which the DRAM banks are available to service memory requests. Higher DRAM density and 3D-stacking aggravate the refresh overheads, incurring even higher energy and performance costs. 3D-stacked DRAM and other emerging on-chip High Bandwidth Memory (HBM) technologies which are widely considered to be changing the landscape of memory hierarchy in future heterogeneous and many-core architectures could suffer significantly from refresh overheads.

Such large on-chip memory, when used as a very large last-level cache, however, provides opportunities for addressing the refresh overheads. In this work, we propose *MicroRefresh*, a scheme for almost eliminating the refresh overhead in DRAM caches. *MicroRefresh* eliminates unwanted refresh of recently accessed DRAM pages; it takes advantage of the relative latency difference between on-chip and off-chip DRAM and achieves a fine balance of usage of system resources by aggressively opportunistically eliminating refresh of *older* DRAM pages. It tolerates any resulting increase in cache misses by leveraging the under-utilized main memory bandwidth. The resulting organization eliminates the energy and performance overhead of refresh operations in the DRAM cache to achieve overall performance and energy improvement.

Across both 4-core and 8-core workloads, *MicroRefresh* eliminates 92% the refresh energy consumed in the baseline periodic refresh mechanism. Further this is accompanied by performance improvements of upto 10%, with average improvements of 3.9% and 3.4% in 4-core and 8-core respectively.

1. INTRODUCTION

Stacked DRAM memory [3] has opened up promising avenues for addressing the performance demands imposed by multi-core processors by provisioning large amounts of DRAM memory at higher bandwidth and lower latency compared to off-chip DRAMs. This is realized by stacking DRAM dies on top of the processor die and data transport implemented using high-bandwidth through-silicon-vias (TSVs). Stacking is expected to offer very large capacity storage - of the order of 100s of megabytes to a few gigabytes.

Researchers have proposed to use the large stacked DRAM storage as a cache [20, 24, 15, 12, 9, 27]. *DRAM Caches* are designed for low latency since any last-level *SRAM* cache (abbreviated *LLSC* in the rest of the paper) miss incurs the latency of a DRAM access. Thus, issues of fast metadata lookup (for cache hit/miss evaluation), set associativity and DRAM Cache block size are important considerations in the design space of DRAM Caches.

Large DRAM density is realized by provisioning a large number of DRAM *pages* (also called *rows*) in each DRAM device. For example, a $1GB$ DRAM with a $2KB$ page size has $512K$ pages. These pages require periodic refreshing of data stored in them as the capacitors that store data in the form of charge lose charge due to circuit leakage. While refreshing is required for functional correctness, these refresh operations incur additional energy and make the DRAM devices unavailable when the refresh is going on.

Compounding this refresh overhead in stacked DRAMs is the issue of its operating temperature. Due to stacking, these DRAM devices operate in the *extended temperature* range [3] (defined to be $85C$ to $95C$ as per the JEDEC specification [14]) due to the heat dissipated by the processor. In this high temperature range, the rate of leakage increases and requires the refresh cycles to become faster. Specifically, the data retention time is halved from $64ms$ (in the normal temperature range) to $32ms$ thereby doubling the refresh overhead. Thus these dual issues of high density and high temperature cause the stacked DRAM organization to suffer from both the energy overhead and performance loss caused by a high rate of refresh operations.

We address the refresh overhead problems in DRAM caches by observing that the latency of on-chip DRAM is about $60-80\%$ of off-chip DRAM and, in the presence of a large on-chip DRAM cache, the off-chip bandwidth is laregly underutilized. Using a simple analytical model and experimentation we demonstrate that the average memory access latency can be reduced by trading off some DRAM cache hits to reduce congestion in the DRAM cache controller. We use this insight to reduce refresh overheads in the DRAM cache by aggressively eliminating refresh to *older* DRAM pages, pages which have not been accessed for more than a threshold amount of time. This decision of not refresh-

MEMSYS 2016 October 3–6, 2016, Washington, DC, USA

DOI: http://dx.doi.org/10.1145/2989081.2989100

ing older pages is based on the load in the DRAM cache controller and the average memory latency/energy experienced. Future requests to such pages, which would result in a DRAM cache miss, are serviced from the main memory thereby achieving the additional advantage of reducing the congestion in the cache controller and improving the off-chip bandwidth utilization. In addition, refreshes to recently accessed DRAM Cache pages are also eliminated. These result in nearly eliminating the entire refresh overhead, relieving the DRAM cache primarily for memory accesses.

As our experimental evaluation in Section 6 shows, these techniques deliver both energy reduction as well as performance improvement. DRAM Cache refresh energy is reduced by an average of 92% across both 4-core and 8-core configurations.This in turn leads to an overall 9% reduction in the energy consumed by on-chip and off-chip DRAM. These energy benefits are obtained along with system performance improvements of 3.9% and 3.4% on average in 4-core and 8-core respectively.

2. BACKGROUND

We first present an overview of DRAM refresh requirements followed by an overview of the DRAM Cache organization that we use as our baseline in this study.

2.1 Refresh Operations

The underlying storage mechanism in DRAM cells is charge stored on capacitors [13]. These capacitors leak charge over time and thus periodic *refresh* operations are performed to restore charge on capacitors. These refresh operations are very similar to normal read operations: a row (page) of DRAM cells are activated and sensed using sense amplifiers that boost the sensed voltage to 0s and 1s. These sensed voltages are then used to restore charge back into these capacitors. Depending on the temperature at which the device is operating, periodic refresh cycles are initiated to ensure that every DRAM cell is replenished with charge within a stipulated *retention window*. Higher temperatues cause faster leakage of stored charge and this requires faster refresh cycles. 3D stacked DRAMs are expected to operate in the *extended* temperature range (85C to 95C) and their cells have a retention window of $32ms$ as stipulated by the JEDEC standard (refer [14]).

Refresh operations are implemented in one of two modes supported by DRAMs:

- *RAS-Only Refresh:* In this mode, the memory controller issues a refresh operation on a specified row of a DRAM bank. The controller supplies the address of the page to be refreshed and is responsible for ensuring that each DRAM page is explicitly refreshed with the *retention window* interval of time since last refresh or access.
- *CAS-Before-RAS Refresh:* Here, the memory controller issues a refresh request to the DRAM banks without specifying the page address. Instead, each DRAM device tracks the next page to be refreshed using its internal counters. This is the more common method of refresh as it keeps the memory controller's implementation simple.

In our work we show that our *MicroRefresh* proposal that uses the *RAS-Only Refresh* technique achieves overall higher energy savings by virtue of eliminating nearly all the refresh operations.

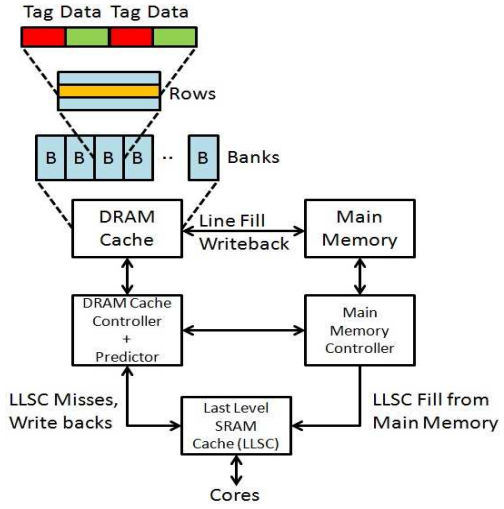

Figure 1: System Organization with DRAM Cache

2.2 DRAM Cache Organization

By virtue of stacking and the inherent density of DRAM, a DRAM cache provides a large capacity (typically 64MB to even gigabytes) offering an unprecedented opportunity to hold critical workload data on chip. The DRAM cache is typically organized as a last level shared cache behind a hierarchy of SRAM caches. In our work, we assume that a shared last level SRAM cache (referred to as the *LLSC*) is in front of the DRAM Cache and sends its misses and writebacks to the DRAM Cache.

An important consideration with DRAM Caches is where the cache metadata (tags, recency, valid, dirty and coherence bits) is stored: on DRAM itself or on a dedicated SRAM store. Recent proposals are divided on this issue with some works (see [20, 24, 12, 9]) favoring tags on DRAM while others (see [15, 16]) favoring tags on SRAM. Further, tags-on-SRAM designs use larger block sizes (1KB–2KB) to reduce SRAM overhead. In either case the intent is to achieve a low-latency tag look-up at low SRAM overhead.

For our work, we use as our baseline the organization presented in [24]. It uses a tags-on-DRAM organization with 64B blocks and direct-mapped sets. The metadata is stored interleaved with data in the same DRAM cache rows permitting fast access to tag and data using a larger burst.

Figure 1 presents an overview of the system organization employing a DRAM Cache. Both the DRAM Cache and main memory have their own memory controllers. The *LLSC* misses and writebacks are first evaluated in a hit/miss predictor. If it is a predicted hit, then the data access is made to the corresponding DRAM Cache bank. If it is a predicted miss, then a DRAM cache fill request is issued to the main memory controller[1]. Note that in either case, predictions are verified to ensure that no bad/stale accesses are made.

3. MOTIVATION

[1]We model a write-allocate DRAM Cache and thus both read and write misses cause line fills to the DRAM Cache.

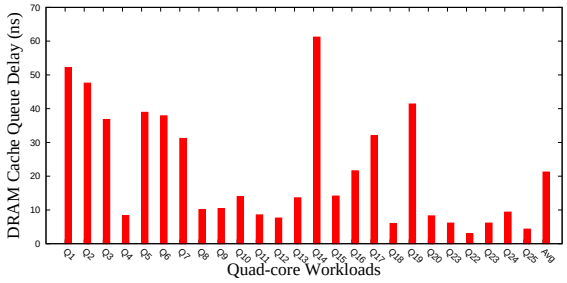

Figure 2: Queuing Delays (ns) at the DRAM Cache Banks

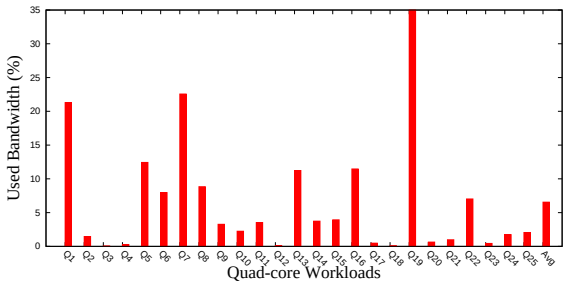

Figure 3: Main Memory Bandwidth Utilization

3.1 Under-utilized Off-Chip Memory Bandwidth

The large capacities of DRAM caches ensure that they capture the bulk of the *LLSC* miss traffic. Thus in systems employing large DRAM Caches, the main memory bandwidth often goes under-utilized. We measured the main memory bandwidth utilization achieved in 4-core and 8-core configurations employing $128MB$ and $256MB$ DRAM Caches respectively. The peak main memory bandwidth is $12.8GBPS$ and $25.6GBPS$ respectively (refer Section 5 for details of our experimental set-up) in 4-core and 8-core. Figure 3 plots the fraction of off-chip bandwidth used by 4-core workloads, with average utilization of only $\approx 6\%$. The average for 8-core is 9% (details not shown due to space limitation). At the same time, the DRAM Cache is heavily congested and pending requests incur large queueing delays. This is demonstrated in Figure 2 wherein we plot the average waiting time experienced by requests at the DRAM Cache controller in our baseline. The average waiting time is $\approx 21ns$ with some workloads experiencing as much as $50ns$, which is $1.5X - 3X$ compared to the average DRAM Cache access time of $\approx 15ns$.

These observations motivate us to consider utilizing the large available main memory bandwidth to reduce the load on the DRAM Cache. As discussed next, in workloads that stress the DRAM Cache, allowing a fraction of these requests to be serviced from main memory actually improves performance.

3.2 Trading DRAM Cache Hits

We present an admittedly simple analytical model to illustrate the importance of leveraging main memory bandwidth. We model both the cache and main memory as M/M/1 queues [26]. The DRAM Cache has a service time of $7.5ns$ per request (i.e, service rate $\mu_{Cache} = 0.1333$ requests per ns)and that of main memory is 4× the DRAM Cache service time[2]. These service times correspond to typical service times seen in our detailed simulations (see Table 2). We can now compute the average times L_{Cache} and L_{Mem} that requests spend in the cache and main memory using the M/M/1 queuing theory result for each server which is given by $\frac{\lambda}{(\mu*(\mu-\lambda))} + \frac{1}{\mu}$ where λ and μ are the arrival and service rates respectively.

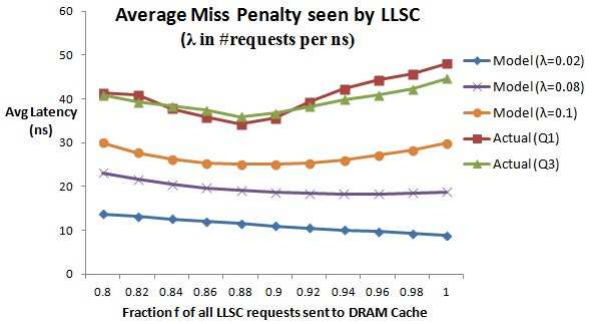

Figure 4: Average *LLSC* Latency with varying fraction f

Now let us assume that a fraction f of all *LLSC* requests are serviced by the DRAM Cache while the remaining requests are diverted directly to the main memory. Thus the cache sees an arrival rate $f\lambda$ and the main memory an arrival rate of $(1-f)\lambda$ where λ is the overall request rate issued by the *LLSC*. The average access latency seen by the *LLSC* is the weighted average given by: $L_{Avg} = fL_{Cache} + (1-f)fL_{Mem}$. Figure 4 plots L_{Avg} as a function of f ranging from 0.8 to 1 for different values of λ. It also plots the actual latencies observed in detailed simulations of two quad-core workloads ($Q1$ and $Q3$). In this experiment, only some fraction of the *LLSC* requests are sent to DRAM Cache and the others are directly sent to main memory (assuming these are for clean DRAM Cache blocks).

The average latency L_{avg} observed in actual simulations shows over 25% reduction at $f^* = 0.88$ compared to the baseline latency (no bypass). The analytical model also predicts a similar behavior (see Figure 4). At high values of λ, even though the cache has an access time which is 4× faster than main memory, there is an optimal fraction $f^* < 1$ at which the overall average latency experienced by *LLSC* misses is minimized. For example, at $\lambda = 0.1$, the minimum average latency experienced by *LLSC* misses occurs at $f^* \approx 0.87$. The performance benefits come — despite an increased (4x) access time of the off-chip memory — essentially from utilizing the under-utilized off-chip memory bandwidth effectively. On the other hand, under light load, it does not make sense to employ cache bypassing. This simple model demonstrates that issuing a small fraction of *LLSC* requests to the main memory is often beneficial for overall latency reduction. It should be emphasized here that the simple analytical model is used only for motivating the benefit of leveraging main memory bandwidth. The work in [10] presents a detailed performance model of DRAM Caches and provides a similar insight.

3.3 Are Periodic Refreshes Useful?

[2]The 4× increase in service time comes from a 2× slower clock and halved channel width.

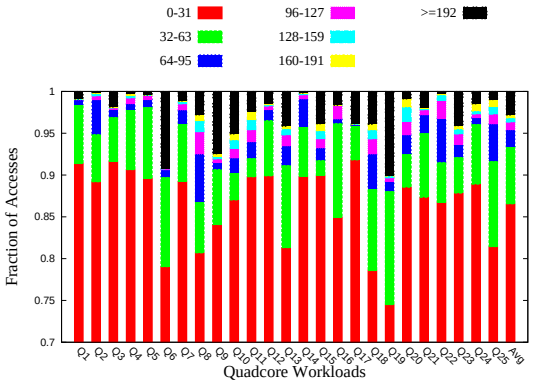

Figure 5: Quad-Core: Fraction of Requests Falling in Different Refresh Intervals

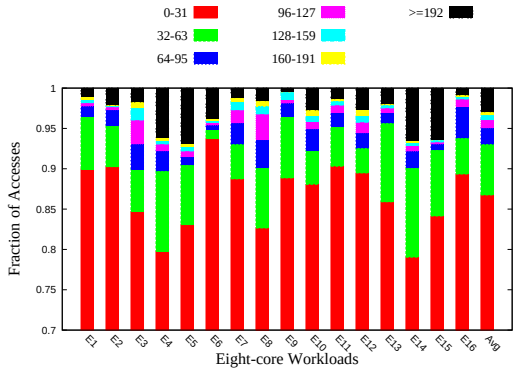

Figure 6: 8-Core: Fraction of Requests Falling in Different Refresh Intervals

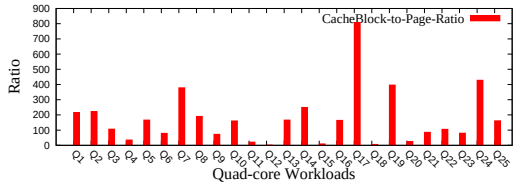

Figure 7: Quad-Core: Ratio of Time Gaps in Accesses to the Same Cache Blocks and Same DRAM Pages

Our discussions so far in this section have demonstrated that off-chip memory bandwidth is under-utilized, and there is a benefit in exploiting this, even by trading some DRAM cache accesses. Can we use this effectively to overcome the refresh overheads of DRAM caches? Before we answer this question, we first ask, to what extent is periodic refresh useful? Figures 5 and 6 plot the fractions of accesses that occur to the same pages, within multiples of refresh intervals (of time duration R) $(0, R)$, $(R, 2R)$, $(2R, 3R)$, $(3R, 4R)$, $(4R, 5R)$, $(5R, 6R)$ and $\geq 6R$ in 4-core and 8-core workloads. For example, at $R = 32ms$ if the next access to a given page is $20ms$ after the previous access, then it is counted in the $(0, R)$ bin. The experimental setup uses the baseline configuration given in Section 5. On average, $\approx 86\%$ of accesses are seen to occur in the $(0, R)$ bin. That is, a majority of pages are accessed atleast once within a refresh interval. This suggests that periodic refreshes are largely unnecessary as the misses from the *LLSC* are themselves causing a majority of the pages to get refreshed.

Given that more than 80% of the pages are accessed frequently, and do not need any refresh, what happens if we eliminate refresh completely? Do the misses caused by pages which are not accessed within a refresh interval cause a performance degradation? To confirm this, we simulated an oracle configuration in which *LLSC* accesses are issued to the cache if the oracle knows that the corresponding cache pages are valid (i.e., refreshed from recent accesses); if not, the accesses are sent to the main memory. Despite incurring additional DRAM Cache misses due to pages which are not refreshed, this oracle configuration results in an average performance gain of 2.9% for 4-core workloads. This suggests that while most pages get refreshed by accesses, any left over accesses are successfully handled by the main memory with resulting energy and performance benefits. Next we explain what causes this performance improvement.

3.4 Refresh Tracking Granularity

Refrint [1] proposes to reduce refreshes by a combination of treating recent accesses as refreshes and invalidating cache lines that have not been accessed for long periods of time. Specifically, *Refrint* addresses the problem in eDRAM (embedded DRAM) memory organized as a cache. It uses per-cache line counters to track the number of refreshes performed since the last read and last write. In the recommended $WB(32, 32)$ configuration, a clean line is invalidated if it receives 32 refresh cycles since the last read, and a dirty line is written back (but not invalidated) if it receives 32 refresh cycles since the last write. *Refrint* addresses eDRAM caches and as such makes two key assumptions: (i) cache block and page size are the same (64B in that work), and (ii) per-cache-block counters are acceptable overhead. In the stacked DRAM Cache organizations, these assumptions are unrealistic - DRAM Cache sizes are much larger, making the overhead prohibitively expensive. Second the cache banks have page sizes in the range $1KB$ to $4KB$ thus mapping multiple cache blocks to the same page. Thus tracking the refresh at cache block granularity (as opposed to tracking at page granularity) essentially loses out a lot of opportunity.

To quantify this observation, we measured the average time gaps between accesses to the same cache blocks as well as to the same DRAM pages. Figure 7 plots the ratio of these times for quad-core workloads run on a 256MB DRAM cache with 64B blocks. A large value indicates that the DRAM pages are accessed much more frequently than cache blocks. With the exception of a few workloads (Q12, Q15, Q18) the ratios are very high (as high as 800 for Q17). Thus we track DRAM cache pages instead of cache blocks for evaluating refresh requirements.

In terms of storage overhead, *Refrint* stores two five-bit counters for tracking when to invalidate/writeback a cache block, a 2-bit access time-stamp and a valid bit. For example, for a cache of size $512MB$ with a block size of $64B$, the storage overhead is $\approx 13MB$ that incurs significant additional access and leakage energy[3]. Thus we argue that it is infeasible to maintain such metadata on a per-cache block

[3]This storage overhead may even exceed the size of the last-level SRAM cache

granularity in future large-sized caches.

MicroRefresh eliminates tracking at a cache block granularity. Instead it tracks page usage and makes invalidation decisions based on access recency.

4. DESIGN

First we present an overview of *MicroRefresh* operation followed by detailed discussions of how pages are categorized, refreshed, or bypassed.

4.1 Overall *MicroRefresh* Operation

MicroRefresh is based primarily on two observations: (i) most DRAM Cache pages are accessed atleast once in a refresh interval, and (ii) a significant part of the memory bandwidth is left unutilized. Combining these two, *MicroRefresh* categorizes DRAM Cache pages as *hot* (accessed within a refresh interval), *dead* (not accessed for a long time, greater than a certain threshold T_{Dead}) and *live* (other pages). *MicroRefresh* can eliminate refresh to all hot and dead pages (ensuring dirty pages are written back before they are allowed to be dead). Misses incurred on dead pages are serviced using the under-utilized memory bandwidth and faster than an overloaded DRAM cache.

The *MicroRefresh* scheme offers the following benefits:

- Refreshes are almost entirely eliminated thereby saving refresh energy
- The cache is not tied up with refresh cycles periodically, allowing DRAM Cache requests to be serviced faster
- Bypassing to main memory has the benefit of bandwidth load-balancing, and reducing the queuing delay at the DRAM Cache controller resulting in an overall latency reduction

4.2 Categorizing DRAM Cache Pages

As mentioned earlier, *MicroRefresh* categorizes DRAM Cache pages into 3 types based on their access history. Intuitively, *hot* pages are refreshed by virtue of recent accesses and thus need not be explicitly refreshed periodically. *Dead* pages are those that have not received any access for a time exceeding a threshold T_{Dead}. Such pages become candidates for skipping refreshes (ofcourse while ensuring that any modified data they hold is written back and the cache blocks marked invalid). A larger value of T_{Dead} allows more pages to stay live in the cache longer. Thus this parameter controls the aggressiveness with which refreshes can be omitted on a fraction of the DRAM Cache contents. Note that when a request to a cache block residing in a *dead* page is received, it is treated as a DRAM Cache miss and is sent to main memory. Such requests may or may not initiate cache line fills depending on whether the load on the DRAM Cache is low or high.

Live pages are those that have not received recent accesses but they hold valid cache data and are not too old. Our design issues explicit refreshes to only *live* pages.

4.3 Page Valid Table

Next we describe how *MicroRefresh* identifies *live* and *dead* pages. It associates a *Valid* bit and an *Accessed* bit with each DRAM Cache page. Initially the *Valid* and *Accessed* bits of all pages are reset to 0. The *Valid* bit of the page is set when a cache line fill occurs that brings data to that page. At the beginning of every T_{Dead} interval, the *Accessed* bits are reset. Access to any cache block in a DRAM Cache page during this interval sets the corresponding *Accessed* bit of the page. At the end of the T_{Dead} interval, whichever pages have their *Accessed* bits still reset are deemed old and invalidated (with writebacks if the invalidated page holds dirty cache data). The *Valid* bit associated with each such page is also reset. Inspecting the *Accessed* bits and invalidating pages are not in the critical path of program execution and could be performed at a lower priority than normal *LLSC* requests.

The two-bit overhead per page is quite small and easily accommodated in a small SRAM table called the *Page Valid Table (PVT)*. For example, for a 512MB cache with 2KB pages, the overhead is 64KB, and is independent of the DRAM Cache block size.

4.4 Run-Time Adaptation of T_{Dead}

Initially, T_{Dead} is set to a high value. In our setup, we set it equal to twice the refresh interval (ie, $64ms$). At the end of the configured T_{Dead} interval, a new value may be assigned to it depending on the load experienced by the cache and memory. If the cache is highly loaded and memory is idle, then it is lowered to enable more pages to get invalidated out of the cache in the next interval.

The adaptation algorithm uses estimates of the "energy-delay" products (EDP) of both the cache and main memory. Specifically, we define $EDP_{cache} = E_{cache\_access} \times L_{cache\_access}$ where E_{cache_access} and L_{cache_access} represent the estimated energy and latency per DRAM Cache access respectively. EDP_{Mem} is similarly defined. L_{cache_access} includes the queuing delays/waiting time before the request gets serviced. Using EDP ensures that the trade-off between cache and memory load-balancing takes into account not just the bandwidth imbalance but also the correspoding energy implications. In this respect, our metric differs from the one employed in MCC [27].

At the beginning of each interval if $EDP_{Cache} > EDP_{Mem}$, then we set $T_{Dead} = T_{Dead} - 8$ (T_{Dead} is lowered in steps of 8 ms upto a low thresold of 8ms). If $EDP_{Cache} < EDP_{Mem} + \Delta$ then we set $T_{Dead} = T_{Dead} + 8$. It should be observed that the run-time adaptation interval can be lower than the periodic refresh cycle. In fact, this aggressive invalidation of pages is required to achieve timely load balance when the cache is heavily congested. In our workloads, we observed that the system lowered T_{Dead} to values below the refresh interval for an average of 3.9% of the total execution time (with congested workloads $Q1, Q3$ and $Q14$ lowering T_{Dead} below the refresh interval for $11\%, 9.7\%$ and 12.2% of their execution times respectively).

Note that our adaptation method does not rely on accurate estimates of energy and latency but only on relative values of cache and memory estimates to be correct. It estimates L_{Cache} and L_{Mem} by monitoring the number of requests in the cache and memory controller queues and scaling them by average cache and memory access times. For energy estimates, we simply set $E_{Mem} = 2 \times E_{Cache}$ using the energy estimates obtained from Micron power calculator [21].

4.5 Accesses from the *LLSC*

All the accesses (misses and writebacks) from the *LLSC*

are first evaluated in the PVT to determine if the corresponding DRAM cache page is valid or not. If the corresponding *Valid* bit is set, then a normal cache tag lookup and hit/miss resolution is carried out as in the baseline case.

If the *Valid* bit is reset or the access evaluated to a cache miss (after tag lookup), then the request is sent to main memory. At the same time, the cache controller determines whether the data response bypasses the cache or not. This is also done using the EDP estimate discussed above. If $EDP_{Cache} < EDP_{Mem}$ then the response causes a cache line fill; else the cache is bypassed. This is summarized in Figure 8.

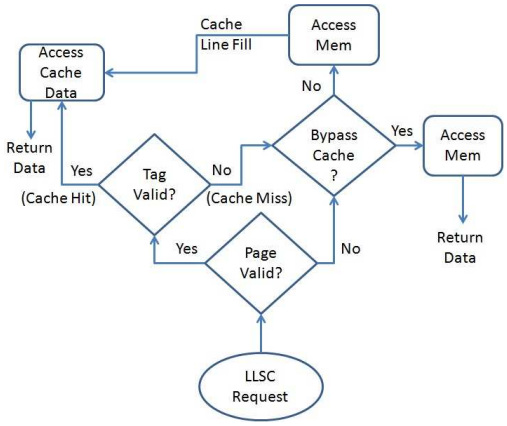

Figure 8: Cache Accesses in *MicroRefresh*

4.6 Support for Tracking Recent Accesses

MicroRefresh leverages recent accesses to skip periodic refreshes to *Hot* pages. A $32ms$ refresh interval is split into 4 quarters of $8ms$ duration each. 2-bit counters associated with each page track the quarter in which the last access was made. At the beginning of each quarter, only those entries that have the matching timestamp are refreshed. Thus for example, a page with timestamp "01" is refreshed at the beginning of the 2nd quarter in the next refresh interval. Whenever an access occurs, its timestamp is updated to the current quarter. Thus such a page does not get refreshed for at least $24ms$ until the start of the corresponding quarter in the next refresh interval occurs. This is illustrated in Figure 9. The last access to X updates its timestamp to "01" and thus gets refreshed at the beginning of the 2^{nd} quarter in the next refresh interval. There are two accesses to Y in quarters "00" and "11" in the same refresh interval. The second access sets the timestamp to "11" causing the periodic refresh operation to skip refreshing Y in quarter "00" in the next interval. Thus frequent accesses to a page cause refreshes to get postponed. In practice, this mechanism almost entirely avoids periodic refresh overhead as demonstrated by our experimental evaluation in Section 6. Our proposal of using timestamps to identify *Hot* pages is similar to works [1] and [8]. Together with the tracking of *Dead* pages, we require a total of 4 bits per page.

4.7 Eliminating Periodic Refreshes

In the *MicroRefresh* design, we eliminate periodic *CAS-Only* refreshes entirely. Instead, we issue explicit refreshes only to valid pages held in the PVT whenever they are close

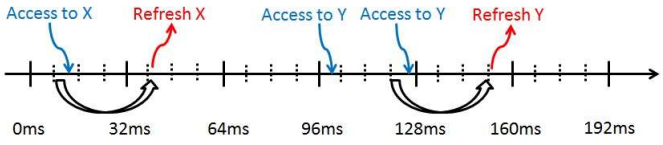

Figure 9: Eliminating Periodic Refreshes

Quad-Core Workloads
*Q1:(462,459,470,433), *Q2:(429,183,462,459), *Q3:(181,435,197,473), Q4:(429,462,471,464), *Q5:(470,437,187,300), *Q6:(462,470,473,300), *Q7:(459,464,183,433), Q8:(410,464,445,433), Q9:(462,459,445,410), *Q10:(429,456,450,459), Q11:(181,186,300,177), Q12:(168,401,435,464), Q13:(434,435,437,171), *Q14:(444,445,459,462), Q15:(401,410,178,177), Q16:(300,254,255,470), *Q17:(171,181,464,465), Q18:(464,450,465,473), *Q19:(453,433,458,410), Q20:(462,471,254,186), Q21:(462,191,433,437), Q22:(197,168,179,187), Q23:(401,473,435,177),Q24:(416,429,454,175) Q25:(254,172,178,188)
Eight Core Workloads
E1:(462,459,433,456,464,473,450,445), *E2:(300,456,470,179,464,473,450,445), *E3:(168,183,437,401,450,435,445,458), *E4:(187,172,173,410,470,433,444,177), E5:(434,435,450,453,462,471,164,186), E6:(416,473,401,172,177,178,179,435), *E7:(437,459,445,454,456,465,171,197), E8:(183,179,433,454,464,435,444,458), *E9:(183,462,450,471,473,433,254,168), *E10:(300,173,178,187,188,191,410,171), *E11:(470,177,168,434,410,172,464,171), E12:(459,473,444,453,450,197,175,164), E13:(471,462,186,254,465,445,410,179), *E14:(187,470,401,416,433,437,456,454), *E15:(300,458,462,470,433,172,191,471), E16:(183,473,401,435,188,434,164,427)

Table 1: Workloads

to completing a refresh interval of time since the last access or since the last refresh cycle. This is accomplished by reading timestamp counters every quarter and if the current quarter matches the timestamp of a page, then a refresh is issued.

5. EXPERIMENTAL METHODOLOGY

We model both the DRAM Cache energy as well as the main memory energy using the Micron power calculator [21]. The power calculator uses a combination of DDR3 configuration (the "DDR3 Config" sheet) as well as the system load (the "System Config" sheet) to determine the memory power in terms of Activate, Read-Write and Standby power. For the baseline, we set $t_{REFI} = 3.9us$ to model the extended temperature range power consumption in the *CAS-Then-RAS* refresh mode. Since *MicroRefresh* uses *RAS-Only* refresh, we model this by removing the periodic refresh power component from the power calculation. Instead, we include the refresh-related *RAS-Only* commands as additional system load. The estimated power is converted to energy by combining it with the simulated run-time of our 4-core and 8-core configurations.

We evaluated the performance benefits of the proposed cache architecture using the GEM5 [2] simulation infrastructure to which we integrated detailed models of stacked DRAM caches and off-chip memory. The memory models

Processor	3.2 GHz OOO Alpha ISA
L1I Cache	32kB private, 64B, Direct-mapped, 2 cycles per hit
L1D Cache	32kB private, 64B, 2-way, 2 cycles per hit
L2 Cache	For 4/8 cores: 4MB/8MB, 8-way/16-way, 128/256 MSHRs, 64B, 7/9 cycles per hit
DRAM Cache (Baseline)	For 4/8 cores: 128MB/256MB, Direct-mapped, 64B blocks, Tags and Data interleaved on rows, 8/16 DRAM Cache banks, 2KB page, 128-bit bus width, 1.6GHz, CL-nRCD-nRP=9-9-9 Refresh: T_{REFI} of 3.9us and T_{RFC} of 280nCK
MicroRefresh	Same as baseline + PVT (Page Valid Table) + Page Timestamp Counters + EDP based bypass
Memory Controller	For 4/8 cores: 1/2 off-chip data channels Each MC: 64-bit channel, 256-entry command queue FR_FCFS scheduling [25], open-page policy Address-interleaving: row-rank-bank-mc-column
Off-Chip DRAM	For 4/8 cores: 4GB/8GB main memory using: DDR3-1600H, BL (cycles)=4, CL-nRCD-nRP=9-9-9 in 4/8 ranks, 32/64 banks

Table 2: CMP configuration

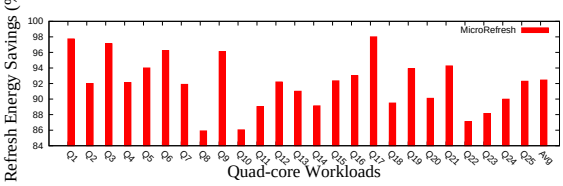

Figure 10: Refresh Energy Savings in *MicroRefresh*

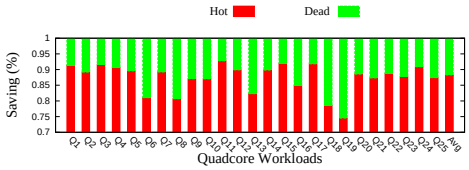

Figure 11: Refresh Energy Savings Break-up

faithfully account for all the significant timing and functional characteristics including hierarchical DRAM organization, key memory timing parameters (including refresh), data bus widths, clock frequencies and memory controller parameters.

For 4-core workloads, timing simulations were run for 1 billion instructions on each core after fast-forwarding the first 10 billion instructions to allow for sufficient warm-up. As is the norm, when a core finishes its timing simulation, it continues to execute until all the rest of the cores have completed[4]. In case of 8-core workloads, due to the amount of simulation time required, we collected statistics on timing runs of $500M$ instructions per core. In all cases, the total instructions simulated across all the cores amount to more than 4B. In addition, we explored several architectural knobs of interest (such as cache hit rate, refresh overhead, bandwidth, and the T_{Dead} parameter setting) using a trace-based DRAM cache simulator. Traces were collected from GEM5 simulations running for 75B instructions on each core. This has resulted in 120M – 450M accesses to the DRAM cache, with an average of $310M$ DRAM cache accesses per workload.

Our workloads are comprised of programs from SPEC 2000 and SPEC 2006 benchmark [11] suites. The 4, and 8-core multiprogrammed workloads are listed in Table 1. These benchmarks were carefully combined to create high, moderate and low levels of memory *intensity*[5] in the chosen workloads to ensure a representative mix. Workloads marked with a "*" in Table 1 have high memory intensity ($LLSC$ miss rate $\geq$ 10%). We also measured the footprints of these workloads in terms of distinct $64B$ blocks accessed. The average memory footprint in 4-core is 7.5× the cache size (128MB) and 8.2× in 8-core (with a cache size of 256MB). Further, these workloads have sufficiently large working sets to fully utilize available cache capacity[6].

[4]The statistics however are collected only during the first 1 Billion instructions.

[5]Intensity was measured in terms of the last-level SRAM cache miss rate.

[6]This was verified by measuring the performance improvement when running 4-core workloads with a 256MB cache and 8-core with a 512MB cache. These larger cache configurations showed average performance improvements of 4.3% and 5.1% over the respective baseline cache sizes.

System performance is measured using the $ANTT$ [6] metric, defined as: $ANTT = \frac{1}{n}\sum_{i=1}^{n}\frac{C_i^{MP}}{C_i^{SP}}$, where C_i^{MP} and C_i^{SP} denote the cycles taken by the i^{th} program when running in a multi-programmed workload and when running standalone, respectively. We present all performance results in terms of improvements in $ANTT$ relative to baseline. For example, the improvement in $ANTT$ achieved by *MicroRefresh* is measured as:

$$ANTT_{Improvement} = \frac{(ANTT_{Baseline} - ANTT_{MicroRefresh})}{ANTT_{Baseline}} \quad (1)$$

In this paper arithmetic mean is used for reporting average improvements in $ANTT$ and energy.

The baseline architecture used in our evaluation [24] and variants of *MicroRefresh* explored are listed in Table 2.

6. RESULTS

6.1 Energy

Figure 10 presents the refresh energy in DRAM Cache saved by *MicroRefresh* in 4-core workloads compared to the baseline. In each workload we observe savings of $\geq$ 85% with average savings of 92%. Similar results (not reported in the paper) are observed in eight-core. To understand the sources of these savings, Figure 11 plots the break up of the periodic refreshes eliminated from *hot* and *dead* pages. While the majority of the savings comes from the refresh reduction achieved on *hot* pages (average of 88% in quad-core), several workloads (notably $Q6, Q8, Q18$ and $Q19$) benefit from the refresh reduction achieved on *dead* pages. Thus *MicroRefresh* successfully eliminates the bulk of the refresh overhead from the DRAM Cache by a combination of periodic refresh elimination and adaptive cache bypass.

Figure 12 plots the total energy savings in the DRAM Cache + memory system for quad-core workloads. *MicroRefresh* achieves average energy savings of 9.5% (peak savings of 23% in $Q3$) over the baseline due to a combination of refresh reduction and performance improvement. To understand the sources of energy savings, the plot in Figure 13 provides a break-up of the energy savings as well as additional energy incurred by *MicroRefresh* in 4-core workloads. *MicroRefresh* saves energy of refresh as well as the background cache and memory energy by virtue of improving

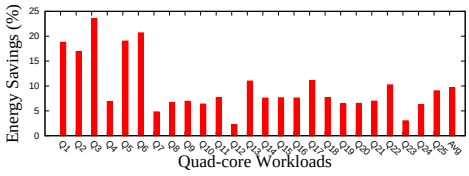

Figure 12: Total Energy Savings in Quad-Core

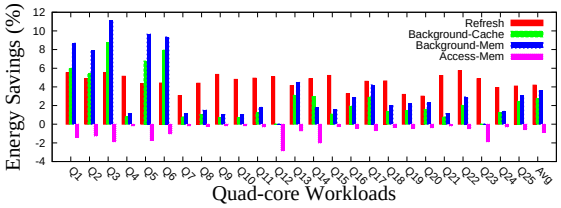

Figure 13: Total Energy Savings Break-up in 4-core

performance (first three bars in each workload). It incurs additional main memory energy due to increased accesses (last bar in each workload). The increase in energy is more than offset by the savings in the other components. In workloads such as $Q1, Q3$ and $Q5$, their performance improvement leads to substantial savings. In all the workloads, refresh reduction is a significant contributor to the overall energy savings. Due to the adaptive nature of *MicroRefresh*, no workload sees a degradation in energy consumption (nor performance as shown later) by limiting the extent of cache bypass. Similar results are observed in 8-core with average total energy savings of 9% (not reported in the paper due to space constraints).

6.2 Performance

In this section, we discuss the performance (ANTT) improvement achieved by *MicroRefresh*. We demonstrate that *MicroRefresh* achieves a moderate improvement in performance due to a combination of freeing up the DRAM Cache from refresh overhead as well as diverting a small fraction of requests to the main memory. Figures 14 and 15 plot the performance improvements achieved in 4-core and 8-core workloads respectively.

MicroRefresh achieves average performance gains of 3.9% and 3.4% over the baseline in 4-core and 8-core respectively. Further, in workloads with high memory intensity (such as $Q1, Q2, Q3, Q14$, $E2, E11$ and $E15$), leveraging the main memory bandwidth improves performance by over 8% over the baseline. As an illustrative example, the queuing delay in workload $Q1$ reduces from $51ns$ (refer Figure 2) to $19ns$ leading to an overall 10% system performance improvement.

We also implemented *MicroRefresh* on top of Footprint-

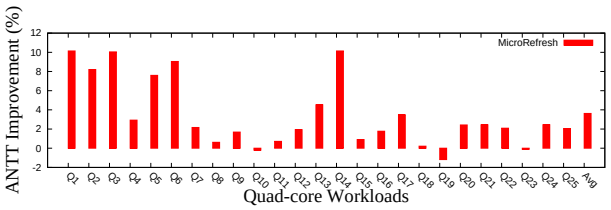

Figure 14: Performance Improvement in 4-core

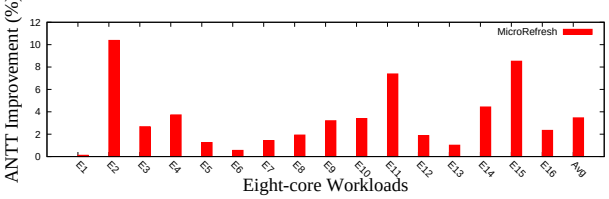

Figure 15: Performance Improvement in 8-core

Cache [15]. Footprint-Cache keeps tags on SRAM and uses large block sizes to reduce tag overhead. As expected, *MicroRefresh* exhibits significant refresh reduction (average in quad-core: 89%) along with an average performance improvement of 2.9% over this baseline.

6.2.1 *Impact on Cache Hit Rate and Off-Chip Bandwidth*

Since our scheme invalidates dead cache blocks early, it may lead to lower cache hit rates. The decrease in cache hit rate depends on the fraction of pages that are invalidated and the access characteristics of the workload. Figure 16 plots the observed hit rates in the baseline and *MicroRefresh*.

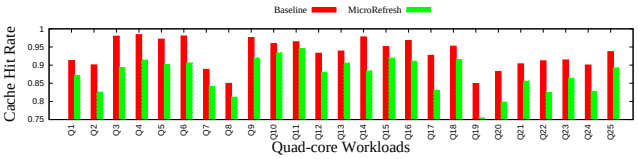

Figure 16: Cache Hit Rates in *MicroRefresh*

We observe that *MicroRefresh* suffers, on an average, 10% lower cache hit rate over the baseline for all workload. Despite this and the resultant increase in main memory access, *MicroRefresh* does not cause any degradation in performance in any of the workloads. Instead, as predicted by our model in Section 3.2, these additional misses are serviced by the under-utilized main memory resulting in an overall minor performance improvement.

6.2.2 *Comparison to a* Zero-Cycle Refresh *Cache*

To understand what contributes to the performance improvements of *MicroRefresh*, we compare the performance of *MicroRefresh* with that of a DRAM cache that performs refreshes at *zero* overhead - all the pages are kept refreshed, the refreshes are assumed to take zero cycles and they do not affect open row-buffers. Such a cache performs better than the baseline by $\approx 1.6\%$ on average in 4-core workloads as shown in Figure 17. *MicroRefresh* achieves an additional improvement of 2.3% over the baseline by not only eliminating the refresh component but also by leveraging the available main memory bandwidth to improve performance further. This effect is pronounced in memory intensive workloads such as $Q1$, $Q3$ and $Q5$. In 8-core, the results are similar - on average, the *Zero-Cycle Refresh* cache and *MicroRefresh* achieve performance improvements of 1.5% and 3.4% respectively over the baseline.

6.3 Comparison With *Refrint* and *MCC*

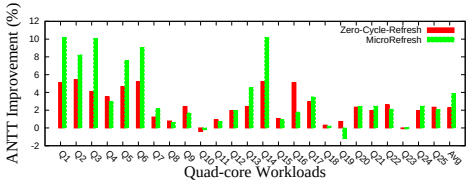

Figure 17: Comparison to Zero-Cycle Refresh

Mostly Clean Cache (MCC) [27] proposes a dynamic self-balancing dispatch mechanism to balance the bandwidth demands on the DRAM cache and main memory. It monitors the cache and main memory request queues and dynamically bypasses the cache (even if it would have resulted in a cache hit) to reduce queuing delays at the DRAM cache.

MicroRefresh proposes a similar mechanism for a different motivation: to nearly eliminate the periodic refresh overhead in the DRAM cache. In MCC, bypassed cache blocks continue to get refreshed thereby incurring energy and performance losses. *MicroRefresh* not only eliminates this but also ensures that the self-regulating mechanism is energy-aware. While MCC uses only the queuing delays at the cache and memory to decide whether to bypass the cache, *MicroRefresh* uses the energy-delay product of accessing the cache and memory to decide on the bypass. Thus, in a scenario where main memory accesses are energy-expensive, *MicroRefresh* is able to throttle the extent of invalidation of cache pages suitably.

In this section, we compare the performance and refresh energy gains achieved by *MicroRefresh* with those of *Refrint* [1] and *MCC* [27]. We also simulated a configuration that simultaneously used both *Refrint* and *MCC* (referred to as *Refrint+MCC*). Figures 18 and 19 plot the performance and refresh energy gains observed by all four configurations in quad-core workloads over the baseline. While *MicroRefresh* achieves performance gains similar to *MCC*, it performs significantly better than *Refrint* or *Refrint+MCC*. In *Refrint+MCC*, i.e., when *MCC* is combined with *Refrint* (for energy reduction), the combined scheme results in significantly lower performance gains. In some workloads (such as Q2, Q3, Q14), *MicroRefresh* gains slightly higher performance than *MCC* by eliminating the periodic refresh component in the DRAM cache. In some others (such as Q1, Q5, Q7), *MCC* achieves better performance than *MicroRefresh* due to the difference in the metric used. While *MCC* examines only queuing congestion, *MicroRefresh* looks at both queuing congestion and energy-per-access to determine whether to bypass. Thus *MicroRefresh* chooses not to bypass even under some congestion if the overall *EDP* of the cache remains lower than that of memory.

On memory system (on-chip and off-chip DRAM) energy, *MicroRefresh* achieves higher savings than the other 3 schemes. Gains over *Refrint* and *Refrint+MCC* are a result of improved performance (resulting in shorter execution time and energy) while the gain over *MCC* is due to refresh energy reduction in the DRAM cache.

6.4 Sensitivity Studies

6.4.1 *Sensitivity:* LLSC *Size*

We explore the refresh energy savings and performance gains achieved with different *LLSC* sizes. Since larger *LLSC* sizes reduce traffic to the DRAM cache, they reduce the opportunity for cache bypassing and at the same time, allow more DRAM cache pages to turn dead and to get invalidated. *MicroRefresh* invalidates them and uses the idle main memory bandwidth effectively. Figure 20 plots the refresh energy savings achieved by *MicroRefresh* over baselines at respective *LLSC* cache sizes (16MB and 32MB). It shows that *MicroRefresh* scales well with larger *LLSC* caches with average savings of 94%. Correspondingly, there is a small gain in performance (average: 2.6%).

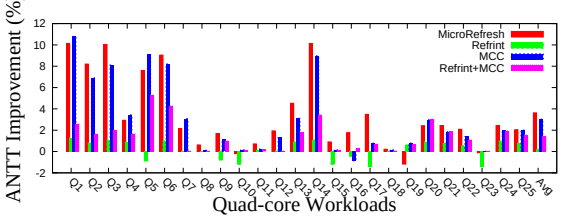

Figure 18: Comparison of *MicroRefresh* Performance with *Refrint*, *MCC* and *Refrint+MCC*

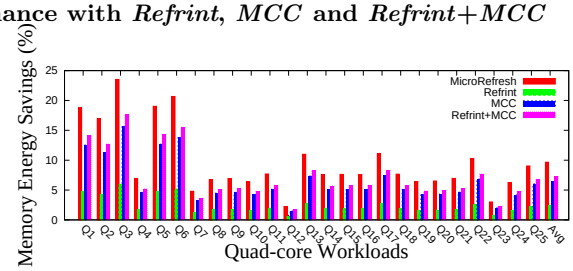

Figure 19: Comparison of *MicroRefresh* Energy with *Refrint*, *MCC* and *Refrint+MCC*

6.4.2 *Sensitivity: DRAM Cache Size*

DRAM Cache sizes are expected to support very large capacity and we conduct a study of the effectiveness of *MicroRefresh* in large sized caches. Larger caches have more DRAM pages and thus the (average) access time gaps between consecutive accesses to the same pages tend to be higher (as the requests distribute out to more pages). This causes a greater fraction of pages to be classifed *dead*. However by use of the *EDP* metric, *MicroRefresh* restricts the extent of bypass by adjusting the T_{Dead} interval setting. As reported in Figure 21, larger DRAM caches continue to benefit from *MicroRefresh* with average refresh savings of 89% and 84% in 512MB and 1GB caches respectively, accompanied by an average performance improvement of 2.9%.

6.4.3 *Sensitivity: 64ms Periodic Refresh*

If the periodic refresh rate is halved to $64ms$, then the performance and energy overheads drop correspondingly. Fig-

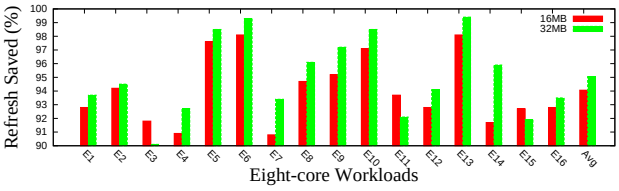

Figure 20: Refresh Savings at different *LLSC* sizes

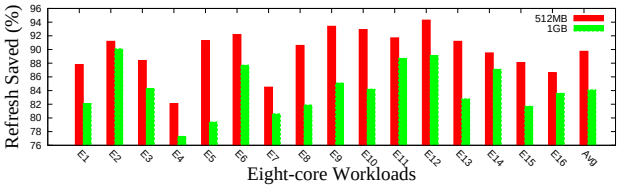

Figure 21: Refresh Energy Saved at 2 Cache sizes

ures 22 and 23 plot the performance and energy improvements in *MicroRefresh* over a 64ms baseline periodic refresh scheme. *MicroRefresh* eliminates nearly all of the refresh overhead (average: 91%) with an average performance improvement of 3%. In congested workloads, *MicroRefresh* continues to utilize the off-chip bandwidth effectively (for example, in $E11$ which achieves a performance gain of 4.2%).

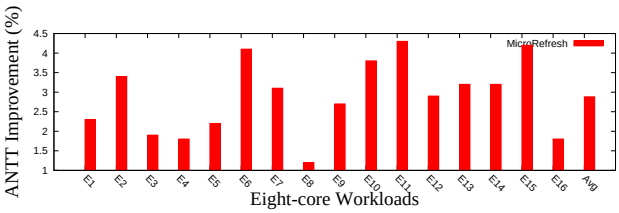

Figure 22: Perf. Gain over 64ms Refresh

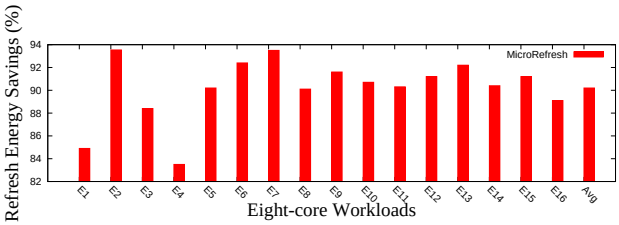

Figure 23: Refresh Energy Gain over 64ms Refresh

6.4.4 *Sensitivity: Non Volatile Main Memory*

An important consideration in the context of non-volatile main memories (NVMs) is write performance. NVMs (Phase Change Memory [32], for example) suffer from high write latency and significantly higher energy owing to the underlying data storage mechanisms. Thus it is desirable to reduce writes to such memories. With our proposal, the DRAM Cache may write back dirty pages more often (whenever a such a page is classified *dead*).

We evaluated the energy and performance impact of *MicroRefresh* when the main memory is a non-volatile memory such as PCM [32]. These memories have very low background energy (mostly consumed by peripheral circuitry) since they store data using a non-volatile medium such as resistance values. Also they have higher write latency and energy since write operations require changing the resistance level of the storage medium. For our evaluation, we used PCM timing and energy parameters as in [32]. We simulated *MicroRefresh* on 4-core workloads using PCM as main memory. An average performance gain of 1.7% is observed in 4-core workloads with resulting average energy improvement of $\approx 6.4\%$[7]. The slower main memory limits the extent of cache bypass and thereby the performance gains as compared to DDR based main memory. The increase in writebacks is $< 3\%$ and incurs only a small increase in write energy.

[7]These improvements are relative to a PCM main memory baseline.

7. RELATED WORK

Several recent works address the issue of growing refresh overhead in DRAM systems [1, 8, 22, 29, 19, 31, 30, 5]. As cache, DRAM refresh offers additional flexibility - pages need not even be refreshed without any correctness issues[8]. Thus our proposal - tailored for DRAM Caches - achieves high energy savings while simultaneously improving performance.

In *SmartRefresh* [8], 2-bit or 3-bit timeout counters are associated with each DRAM page. Whenever a page is accessed or refreshed, the counter is reset to its maximum value. Every $16ms$ (for a $64ms$ refresh interval), the counters are decremented and whenever a counter reaches 0, a refresh is scheduled. In order to avoid a burst of refreshes at $16ms$ boundaries, it also proposes a staggered countdown technique. While conceptually similar, our proposal has 3 key benefits: i) in our scheme, the timestamps are updated only on access, not periodically, ii) at any time timestamps are maintained only for a subset of all pages, and iii) unused main memory bandwidth is used to gain performance. By updating the timestamps of only the cached pages only on access, our scheme avoids the energy and complexity of updating *all* the counters periodically while simultaneously improving performance. In [22], the new fine granularity refresh (FGR) feature of DDR4 memories is leveraged to develop an adaptive mechanism that determines the granularity and rate of each refresh operation. In [29], periodic refresh operations are suitably delayed to allow higher priority CPU requests to be serviced first, leveraging the tolerance permitted by the JEDEC DDR specification [14]. In contrast, our work does not use the in-built *CAS-Then-Ras* refresh mechanism and instead uses explicit *RAS-Only* refreshes.

The work in [19] observes that different DRAM pages may have different rate-of-refresh requirements and thus calibrates all the pages into different bins. It then issues periodic refreshes to all the pages as per the bins they belong to. While this mechanism is orthogonal to all of the other schemes, it requires the DRAM device to operate outside the JEDEC specification and also has to ensure reliable operation in the presence of run-time variations due to temperature and leakage issues arising from inter-page electrical interference. In the work in [30], a software approach is used to allocate pages with longer data retention time first before pages with smaller retention times are allocated. This approach is suitable for main memory wherein the large capacity may not get used up. With DRAM Caches, however, this technique does not mitigate the refresh overhead as all of the cache quickly gets filled up and remains in use. The works in [31] and [5] propose use of error codes to detect if a DRAM page has decayed. These proposals come with additional code storage overhead and access complexity.

There are a number of works related to on-chip SRAM energy reduction (for example, see [7, 17]) wherein counter

[8]As long as the main memory has the latest data for such pages.

or timeout based mechanisms are used to determine when to invalidate cache lines or move them to lower power states. In contrast, our work addresses DRAM caches wherein both the book-keeping overhead as well as the granularity of low power modes drive a different set of design choices.

Bandwidth partitioning [28, 33, 23, 18] addresses the issue of load balancing across multiple memory systems/channels to ensure overall high throughput. The work in [28] proposes a dynamic mechanism to issue requests to the cache or to the main memory based on the instantaneous load on the cache. It uses a hybrid write-through/write-back policy with additional book-keeping of dirty pages to ensure functional correctness. In contrast, our proposal driven by the motivation to reduce refresh overhead proposes a simple tracking of valid pages to allow dirty data to reside in the cache as long as the page is not classifed as *dead*. The work in [33] explores mechanisms to reduce write-backs in PCM based main memories. The works in [23, 18] are aimed at balancing multiple memory channels/controllers for high throughput. While stacked DRAMs offer high bandwidth, when used as a large cache a significant fraction of this bandwidth gets consumed for cache management operations. The work in *BEAR* [4] discusses this overhead and proposes techniques to mitigate it.

8. CONCLUSIONS

In this paper, we propose *MicroRefresh*, a DRAM refresh organization that almost entirely eliminates refresh operations in DRAM caches by exploiting a fine balance between utilizing off-chip memory bandwidth and trading DRAM cache hits, and thereby significantly reducing the refresh energy incurred. By classifying pages as *hot*, *dead* and *live* and explicitly refreshing only *live* pages, we are able to eliminate over 90% of the periodic refreshes on average. Further, by leveraging the available main memory bandwidth, we improve overall performance by as much as 10% over the baseline, with average improvements of 3.9% and 3.4% in 4-core and 8-core configuratons respectively.

9. REFERENCES

[1] A. Agrawal, P. Jain, A. Ansari, and J. Torrellas, "Refrint: Intelligent refresh to minimize power in on-chip multiprocessor cache hierarchies," in *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013*, 2013, pp. 400–411. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2013.6522336

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[3] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, "Die stacking (3d) microarchitecture," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 469–479. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.18

[4] C. Chou, A. Jaleel, and M. K. Qureshi, "Bear: Techniques for mitigating bandwidth bloat in gigascale dram caches," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 198–210. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750387

[5] P. G. Emma, W. R. Reohr, and M. Meterelliyoz, "Rethinking refresh: Increasing availability and reducing power in dram for cache applications." *IEEE Micro*, vol. 28, no. 6, pp. 47–56, 2008.

[6] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads." *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.

[7] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: Simple techniques for reducing leakage power," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ser. ISCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 148–157. [Online]. Available: http://dl.acm.org/citation.cfm?id=545215.545232

[8] M. Ghosh and H.-H. S. Lee, "Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40. Washington, DC, USA: IEEE Computer Society, 2007, pp. 134–145. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2007.38

[9] N. Gulur, M. Mehendale, R. Manikantan, and R. Govindarajan, "Bi-modal dram cache: Improving hit rate, hit latency and bandwidth," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, Dec 2014, pp. 38–50.

[10] N. Gulur, M. Mehendale, and R. Govindarajan, "A comprehensive analytical performance model of dram caches," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '15. New York, NY, USA: ACM, 2015, pp. 157–168. [Online]. Available: http://doi.acm.org/10.1145/2668930.2688044

[11] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[12] C.-C. Huang and V. Nagarajan, "Atcache: Reducing dram cache latency via a small sram tag cache," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 51–60. [Online]. Available: http://doi.acm.org/10.1145/2628071.2628089

[13] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[14] JEDEC, "Ddr3 sdram specification," 2010.

[15] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram

caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 404–415. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485957

[16] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "Chop: Adaptive filter-based dram caching for cmp server platforms." in *HPCA*, M. T. Jacob, C. R. Das, and P. Bose, Eds. IEEE Computer Society, 2010, pp. 1–12.

[17] S. Kaxiras, Z. Hu, and M. Martonosi, "Cache decay: Exploiting generational behavior to reduce cache leakage power," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ser. ISCA '01. New York, NY, USA: ACM, 2001, pp. 240–251. [Online]. Available: http://doi.acm.org/10.1145/379240.379268

[18] F. Liu, X. Jiang, and Y. Solihin, "Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance." in *HPCA*, M. T. Jacob, C. R. Das, and P. Bose, Eds. IEEE Computer Society, 2010, pp. 1–12.

[19] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "Raidr: Retention-aware intelligent dram refresh," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337159.2337161

[20] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 454–464. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155673

[21] MICRON, "Micron ddr3 power calculator," 2009. [Online]. Available: http://www.micron.com/products/support/power-calc

[22] J. Mukundan, H. Hunter, K.-h. Kim, J. Stuecheli, and J. F. Martínez, "Understanding and mitigating refresh overheads in high-density ddr4 dram systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 48–59. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485927

[23] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 374–385. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155664

[24] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 235–246. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.30

[25] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 128–138. [Online]. Available: http://doi.acm.org/10.1145/339647.339668

[26] S. M. Ross, *Introduction to Probability Models, Ninth Edition*, 2006.

[27] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A mostly-clean dram cache for effective hit speculation and self-balancing dispatch," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 247–257. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.31

[28] ——, "A mostly-clean dram cache for effective hit speculation and self-balancing dispatch," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 247–257. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.31

[29] J. Stuecheli, D. Kaseridis, H. C.Hunter, and L. K. John, "Elastic refresh: Techniques to mitigate refresh penalties in high density memory," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 375–384. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2010.22

[30] R. K. Venkatesan, S. Herr, and E. Rotenberg, "Retention-aware placement in dram (rapid):software methods for quasi-non-volatile dram," in *In Proceedings of the Twelfth Annual Symposium on High Performance Computer Architecture*, 2006, pp. 155–165.

[31] C. Wilkerson, A. R. Alameldeen, Z. Chishti, W. Wu, D. Somasekhar, and S.-l. Lu, "Reducing cache power with low-cost, multi-bit error-correcting codes," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 83–93. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815973

[32] H.-S. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec 2010.

[33] M. Zhou, Y. Du, B. Childers, R. Melhem, and D. Mossé, "Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 53:1–53:21, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2086696.2086732