Run-Time Power-Down Strategies for Real-Time SDRAM Memory Controllers

Karthik Chandrasekar[1], Benny Akesson[2], Kees Goossens[2]
[1]Computer Engineering, TU Delft, The Netherlands
[2]Electronic Systems, TU Eindhoven, The Netherlands
[1]k.chandrasekar@tudelft.nl
[2]{k.b.akesson, k.g.w.goossens}@tue.nl

ABSTRACT

Powering down SDRAMs at run-time reduces memory energy consumption significantly, but often at the cost of performance. If employed speculatively with real-time memory controllers, power-down mechanisms could impact both the guaranteed bandwidth and the memory latency bounds. This calls for power-down strategies that can hide or bound the performance loss, making run-time memory power-down feasible for real-time applications.

In this paper, we propose two such strategies that reduce memory energy consumption and yet guarantee real-time memory performance. One provides significant energy savings without impacting the guaranteed bandwidth and latency bounds. The other provides higher energy savings with marginally increased latency bounds, while still preserving the guaranteed bandwidth provided by real-time memory controllers. We also present an algorithm to select the most energy-efficient power-down mode at run-time. We experimentally evaluate the two strategies at run-time by executing four media applications concurrently on a real-time MPSoC platform and show memory energy savings of 42.1% and 51.3% for the two strategies, respectively.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Performance, Design, Algorithms

Keywords

SDRAM, Power-Down, Real-Time, Memory Controller

1. INTRODUCTION

Increasing performance demands of modern MPSoCs often reflect poorly in overall system energy consumption. SDRAMs in particular, contribute considerably to the system energy consumption [1] and have the option of powering down [2] at run-time to save energy. However, these power-down mechanisms come at the cost of performance (bandwidth and latency), due to their power-up latencies [10].

Applications with real-time requirements demand worst-case performance guarantees from every component in the system, including the SDRAMs, where these guarantees are at the memory transaction level. Real-time SDRAM controllers provide such guarantees to a memory requester, such as a processor, in terms of a minimum guaranteed bandwidth and/or a maximum latency bound for memory accesses. Real-time SDRAM controllers, such as [3–8], employ predictable *memory arbiters*, such as Round-Robin or Time Division Multiplexing, to schedule memory accesses from different requesters and to provide performance guarantees. If they speculatively employ power-down mechanisms at run-time when the memory is idle, it can affect both the latency and the bandwidth guarantees provided, due to the power-up latencies [10]. Hence, they do not support run-time power-down. However, to design efficient future real-time systems [9], it is essential to reduce memory power consumption while satisfying performance requirements.

This paper proposes *two run-time power-down strategies* that reduce memory power consumption, while preserving the original bandwidth guarantees and also providing memory access latency bounds to guarantee real-time behavior. The first strategy provides significant energy savings without impacting either the maximum latency bounds or the minimum guaranteed bandwidth. The second strategy provides higher energy savings with marginally increased bounds on the memory latency, while still preserving the original guaranteed bandwidth provided by the real-time memory controller. Both these strategies can be employed with any of the real-time memory controllers presented in [3–8].

SDRAMs support different power-down modes viz., *fast exit* and *slow exit*. The former has a short power-up latency and saves some power, while the latter has a longer power-up latency, but saves more power. This paper also proposes an *algorithm to select the most energy-efficient power-down mode at run-time* based on the memory state and the power-down duration, for both the power-down strategies.

We experimentally evaluate the two proposed strategies by concurrently executing four media applications on an MPSoC platform using a real-time SDRAM memory controller. We show around 42.1% memory energy savings using the first power-down strategy and around 51.3% using the second strategy. The second strategy almost reaches the theoretical maximum of 51.4% memory energy savings using power-down modes, but slightly increases the execution time of the applications by 0.25%, due to the marginal increase in latency bounds. We also compare these two strategies against a speculative power-down policy on energy savings and impact on performance guarantees.

The remainder of this paper is organized as follows: Section 2 describes the related work on real-time SDRAM controllers and memory power optimization strategies. Section 3 gives the background on SDRAMs and introduces real-time memory arbitration. Section 4 presents the proposed power-down strategies, followed by deriving the impact of these strategies and a speculative power-down policy

DAC 2012, June 3-7, 2012, San Francisco, California, USA.

on performance guarantees in Section 5. Section 6 describes an algorithm to select the most energy-efficient power-down mode at run-time. In Section 7, we experimentally evaluate our solutions using four media applications and compare against the speculative and theoretical-best power-down options. Section 8 concludes the paper by highlighting the contributions of this work.

2. RELATED WORK

Real-time SDRAM memory controllers like [3–8] employ predictable arbiters, such as Round-Robin or TDM, and provide latency and/or bandwidth (rate) guarantees by bounding the temporal interference between requesters.

[4] employs Round-Robin arbitration and provides upper bounds on delays for different memory accesses. Similarly, [6] employs Round-Robin arbitration and uses worst-case response time to bound memory access latency. [3] adopts a budget-based static-priority arbitration and provides bounds on latency and guarantees a minimum bandwidth for every memory requester. It also supports Round-Robin or TDM arbiters. [7] uses TDM arbitration and provides bandwidth guarantees and a worst-case execution time for memory accesses. In [5], weighted Round-Robin arbitration is used to provide both bandwidth guarantees and latency bounds. [8] uses static scheduling and provides predictable memory accesses. However, none of these real-time memory controllers support power-down at run-time, due to the impact of power-up latencies on performance guarantees.

When it comes to work on SDRAM memory power minimization, there exists no generic run-time SDRAM power-down solution for real-time systems. For instance, [15] proposed to reduce idle power consumption by using a compiler-directed selective power-down and a hardware-assisted run-time power-down. However, the former is not suitable for run-time use and the latter can incur large performance penalties due to mis-predictions of future idleness. [16] proposed history-based scheduling and an adaptive memory throttling mechanism to allow memory to remain in the idle mode for longer periods of time to employ power-down longer. However, these methods also incur performance penalties and cannot be used for real-time applications.

In short, real-time memory controllers do not currently support power-down mechanisms, and existing power-saving solutions are not applicable at run-time and cannot be used with real-time memory controllers. This paper bridges this gap and provides run-time power-down strategies for real-time SDRAM memory controllers.

3. BACKGROUND

This section discusses SDRAM organization, operation and power-down options. This is followed by an introduction to predictable arbiters and how they guarantee latency and bandwidth (rate) in a real-time memory controller.

3.1 SDRAM Essentials

SDRAMs are organized as a set of memory banks that include memory elements arranged in rows and columns. A row buffer also resides in every bank to store contents of the currently accessed memory row. The banks in an SDRAM operate in a parallel and pipelined fashion, although only one bank can perform an I/O operation (data transfer) at a particular instance in time, due to the shared data bus. To read contents from the memory, an *Activate* command is first issued by the memory controller to the SDRAM, which opens the requested row and brings data from the SDRAM cells into the row buffer. Then, any number of *Read* or *Write* commands can be issued to read out or write into specific columns of data in the row buffer. Subsequently, a *Precharge* command is issued and the contents of the row buffer are stored back into the corresponding memory row. Reads and writes can also be issued with an *auto-precharge flag* to automatically precharge as soon as the request completes. If any row is active, the memory is said to be in the *active* state, else it is in the *precharged* state. Switching between a read and a write transaction, or vice versa, takes a few clock cycles. Further, to retain data in the memory, all rows in the SDRAM need to be refreshed at regular intervals, which is done by issuing a *Refresh* command. The number of words of data transferred in a single read/write command is called a burst, and its size is given by the Burst Length (BL) (usually 8 words for DDR3). A memory controller may serve a single request by issuing a number of read/write bursts per bank (defined by the Burst Count (BC) parameter) and interleaving over more than one banks (given by the degree of Bank Interleaving (BI)). The BL, BC and BI parameters determine the data *access granularity* with which the memory controller accesses the memory and have a large impact on performance and power consumption [18].

3.2 Power-Down Options in SDRAMs

It is possible to power down the SDRAM memory at run-time to reduce power consumption if it is not in use. SDRAMs support different power-down modes, such as *fast exit* and *slow exit* [10, 11]. The former has shorter power-up latency but saves less power, while the latter has longer power-up latency, but saves more power. These power-down modes can be entered either in the active or precharged state, based on certain timing constraints and the type of memory being used. For DDR2 memories, an active power-down can be used in the fast or slow exit mode, but only in the slow exit mode for DDR3. Conversely, a precharged power-down can be used in the fast or slow exit mode for DDR3, but only in the slow exit mode for DDR2.

When transitioning into and out of these power-down (PD) modes, certain timing constraints must be respected, as shown in Figure 1. In the figure, the t_{TRANS} parameter gives the *transition in* timing constraint before the memory can switch to a power-down mode after a read/write command is issued. The t_{PD} parameter gives the *power-down time*, which may vary from a minimum of *tCKE* (Clock Enable pulse width) to a maximum of $9 \times$ *tREFI* (refresh interval). The t_{PUP} parameter gives the *power-up* timing constraint before the next command can be issued. This t_{PUP} parameter for the fast exit mode is given by t_{XP} for DDR3 and t_{XARD} for DDR2 memories. For the slow exit mode, the same power-up timing constraints are applicable if the next issued command is an ACT/PRE/REF. However, before a Read/Write command is issued after powering-up, the timing constraints of t_{XPDLL} for DDR3 and t_{XARDS} for DDR2 must be satisfied, for the DLL to be activated before issuing these commands.

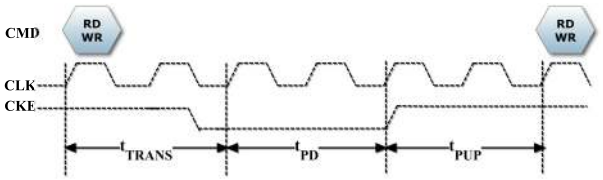

Figure 1: Power-Down Transitions

3.3 Arbiters and Latency-Rate Servers

Real-time SDRAM memory controllers employ predictable arbiters to provide latency and bandwidth (rate) guarantees to applications (requesters) accessing the memory. These arbiters use scheduling algorithms like Round-Robin and TDM, and can be analyzed using the Latency-Rate ($\mathcal{LR}$) server model [12], to characterize their performance guarantees. These guarantees are provided to a requester in terms of a *minimum rate of service* (ρ) and a *maximum initial service latency* (Θ), whenever it is *busy* (requesting a higher rate of service on average than allocated to it). Figure 2 depicts this rate guarantee and the initial service latency

bound provided by $\mathcal{LR}$ arbiters. As shown in the figure, a *busy period* for a requester corresponds to a time interval when its *requested service rate* is above the *busy line*, else, it is considered to be *not busy*. In the figure, the *allocated service line* indicates the minimum rate guarantee (ρ) given to a requester. ρ corresponds to the fraction of the net memory bandwidth that is provided as the bandwidth guarantee (β) to that requester. The maximum initial service latency (Θ) gives the maximum duration a requester has to wait after its arrival, to start getting served by the memory at the guaranteed rate (ρ). As can be noticed in the figure, the actual *provided service* may be higher than the allocated rate, if the system has the capacity to support it.

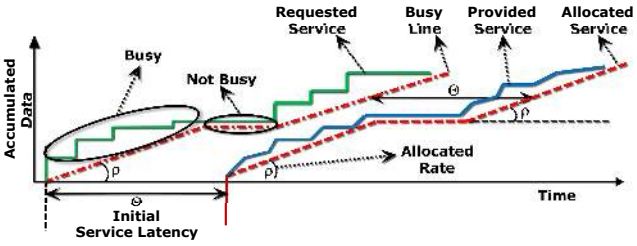

Figure 2: Latency-Rate Server

In short, a Latency-Rate ($\mathcal{LR}$) arbiter provides a busy requester, a guaranteed bandwidth β in the form of a guaranteed rate of service ρ after an initial service latency (Θ). These guarantees can be used for formal verification of an application's real-time behavior.

3.4 $\mathcal{LR}$ Arbiters and Memory Controller Guarantees

This section describes how latency and bandwidth (rate) guarantees are derived for a real-time SDRAM controller.

The initial service latency bound (Θ) of a requester can be intuitively seen as the duration between the time of arrival of a request at the arbiter and the time at which the request is accepted by the memory for service. It is given by the sum of the service time of the request currently being served and that of other interfering requesters, including refreshes (if any), as discussed later in Section 5. The service time for any given request is defined as the *service cycle length* (SCL) of that request. This can be highly variable depending on whether the request is a read or a write and if there is switching time involved (from read to write or vice versa) between the last and the next request. Hence, we use the longest SCL denoted by *maximum service cycle length* (*max_SCL*) to derive a conservative worst-case initial service latency bound Θ (as will be shown in Section 5.1).

As stated before, once the request is accepted for service by the memory after Θ, it is guaranteed a minimum rate of service, ρ. This rate of service (ρ) defines the bandwidth guarantee (β), based on the net memory bandwidth (*net_BW*). This net memory bandwidth is predominantly defined by the request size and the *max_SCL* for the particular request size. The size of a request can be defined as a multiple of the *access granularity* parameter (described in Section 3.1), which is the minimum size of data accessed by the memory controller. For efficient memory access, the access granularity should be of the same size as the request size, although it can be smaller. In this work, we assume all requests to be of one size, and the access granularity to be of the same size as the requests, for efficient memory access and simplicity of analysis. The *max_SCL* for the given access granularity is then used to compute the net memory bandwidth (*net_BW*) and along with ρ, is used to provide the bandwidth guarantee β (also shown in Section 5.1).

An $\mathcal{LR}$ arbiter employs a *scheduling interval* parameter, to schedule different requesters to memory. This parameter gives the duration after which, in every service cycle, a subsequent requester is selected to be scheduled at the end of the current request. This scheduling interval is statically defined as the minimum service cycle length among all requests (*min_SCL*), since this is the minimum period after which, the next requester could be scheduled.

As an example, consider 64-byte requests from a real-time memory controller accessing a 1Gb DDR3-800 memory with a BC of 4 interleaving over 1 bank. The SCLs of read and write requests (including any switching) corresponding to the 64-byte access granularity are shown in Figure 3. As can be noticed, the SCLs vary depending on the request type (read/write). The shortest SCL (*min_SCL*) is 26 clock cycles for a read transaction and this defines the *scheduling interval* for all service cycles and the length of an idle service cycle. The longest SCL *max_SCL* is 37 cycles and it includes write SCL (t_{WR}) and read to write switching time (t_{RTW}).

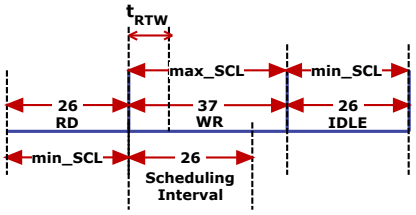

Figure 3: Scheduling Interval & SCLs

4. REAL-TIME POWER-DOWN STRATEGIES

Existing real-time SDRAM controllers employ predictable $\mathcal{LR}$ arbiters, such as Round-Robin and TDM, to provide latency and/or bandwidth guarantees, but do not address power optimization. In this section, we propose two run-time power-down strategies for such memory controllers; one a conservative latency-bandwidth-neutral strategy and the other an aggressive bandwidth-neutral strategy. These strategies can be employed whenever the memory is idle to reduce memory power consumption, while preserving the original guaranteed bandwidth, and providing bounds on memory latencies. The analysis here is restricted to DDR3 memories, although it is easily adaptable for DDR2 as well.

4.1 Conservative Latency-Bandwidth-Neutral Strategy

The first strategy involves triggering a special *power-off* request whenever an arbiter service cycle is idle. This power-off request is designed to power down the memory and power it back up within the scheduling interval, thus hiding the power-up transition latencies within the idle service cycle. This ensures that the scheduling of memory access requests is not disturbed and the power-down mechanism is effectively hidden from the requesters. This latency-bandwidth-neutral strategy provides significant energy savings and preserves both the guaranteed initial service latency bounds and the bandwidth, as is shown later in Section 7.

4.2 Aggressive Bandwidth-Neutral Strategy

The second strategy is more aggressive, since it checks for new requests before powering up the memory. It involves issuing a *power-down* request when there are no pending requests at the arbiter, and snooping the arbiter inputs for new requests before the end of the current idle service cycle. If there are any new pending requests, a *power-up* request is issued to the memory to power it up by the end of the idle service cycle (thus maintaining scheduling interval). To implement this strategy, we introduce a *Snooping Point* at a pre-defined time instance, before the end of the scheduling interval, as shown in Figure 4. This snooping point can be derived by subtracting the worst-case power-up time (t_{PUP_max}) (given by Equation (1)) from the scheduling interval (given by Equation (2)).

$$t_{PUP\_max} = max(t_{XP}, t_{XPDLL} - t_{RCD}) \quad (1)$$

$$t_{SNOOP} = t_{SCHED\_INTERVAL} - t_{PUP\_max} \quad (2)$$

This strategy assures that the memory powers-up in time and following request is scheduled on-time, as in the conservative case, if it arrives before this snooping point.

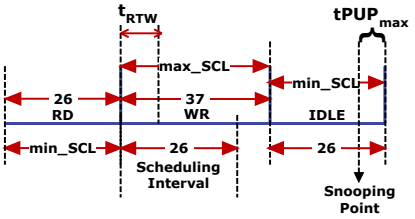

Figure 4: Snooping Point in Aggressive Power-Down

As can be noticed in Equation (1), t_{PUP_max} considers the minimum timing constraint between an ACT and a RD/WR command (t_{RCD}) [10], besides the fast exit (t_{XP}) and slow exit (t_{XPDLL}) power-up timing constraints. The rationale behind it is as follows: In DDR3 memories, the slow exit power-down can be employed only in the precharged state, which implies that every read/write transaction has to end with a precharge and begin with an ACT command. The power-up timing constraint before issuing an ACT after a slow-exit power-down is given by t_{XP}, which is shown as the first constraint in Equation (1). For efficient memory access, the first RD/WR command is scheduled immediately after t_{RCD} is satisfied, after an ACT command is issued. However, since a RD/WR can be issued only after a duration of t_{XPDLL}, after the memory begins to power-up, the corresponding ACT in the transaction can be issued after $max(t_{XP}, t_{XPDLL} - t_{RCD})$ is satisfied.

Now consider the scenario, when the next request arrives after the snooping point but before the end of scheduling interval. In this case, no power-up will be issued and the memory continues in the power-down mode. This results in the next request missing a service cycle and getting scheduled in the following service cycle, if no other interfering requesters show up. However, as will be shown in Section 5, this only increases the latency bounds (Θ) by the power-up transition time (in the worst-case) and does not impact the maximum service cycle length (max_SCL) and therefore the bandwidth guarantee (β). In short, this strategy is bandwidth-neutral and provides marginally increased latency bounds, thus guaranteeing real-time memory performance. The advantage of this strategy is that all contiguous idle periods are combined into one large idle period, thereby avoiding frequent powering-up of memory (every idle service cycle), as in the conservative strategy, to save more energy.

5. IMPACT ON LATENCY-BANDWIDTH BOUNDS

In this section, we first derive the initial service latency bounds and bandwidth guarantees offered by real-time memory controllers. These guarantees are conservative and simpler than those presented in [14], for ease of understanding. We then analyze the impact of the conservative, aggressive and speculative power-down strategies on these bounds.

5.1 Latency and Bandwidth Guarantees

As stated in Section 3.4, the bandwidth guarantee (β) depends on net memory bandwidth (net_BW), which can be derived using the max_SCL for a given access granularity. The worst-case bound for max_SCL can be computed as the maximum of: (1) the service time of a read (t_{RD}) and the time to switch to a read after a write (t_{WTR}), and (2) the service time of a write (t_{WR}) and the time to switch to a write after a read (t_{RTW}) (given by Equation (3)).

$$max\_SCL = max(t_{WTR} + t_{RD}, t_{RTW} + t_{WR}) \quad (3)$$

To compute the net memory bandwidth, let us assume that a requester is busy throughout a refresh interval (t_{REFI}), when it is interrupted by a refresh. For a worst-case estimate, we consider every service cycle during t_{REFI} to be as long as max_SCL. Hence, the total number of service cycles (num_SCL) in t_{REFI} is given by Equation (4), where t_{Ref} is the length of a single refresh request in the refresh interval.

$$num\_SCL = \lfloor (t_{REFI} - t_{Ref}) / max\_SCL \rfloor \quad (4)$$

The total data transfer during this period, assuming an access granularity of G is $num\_SCL \times G$. Hence, the next memory bandwidth is given by Equation (5).

$$net\_BW = num\_SCL \times G / tREFI \quad (5)$$

Consider a use-case with 'x' requesters accessing the memory through a real-time memory controller using Round-Robin arbitration. Each requester is guaranteed a service rate of ρ = '1/x' in the form of *1 out of x Round-Robin time slots*, if it is *busy*. Hence, the minimum bandwidth guaranteed to each requester (β), is given by Equation (6).

$$\beta = net\_BW \times \rho \quad (6)$$

Next, we derive the initial service latency (Θ) for a requester (in Equation (7)), considering the max_SCL from Equation (3)), in the presence of 'x' interfering requesters. We also consider a refresh request length t_{tRef} for any interference from a refresh during the busy period. In addition, Θ would also include the SCL of the currently scheduled request and a waiting period t_{wait} equal to the difference between the max_SCL and the scheduling interval (min_SCL).

$$\Theta = t_{wait} + t_{Ref} + (max\_SCL \times (x + 1)) \quad (7)$$

In Equation (7), the '1' corresponds to service cycle of the currently scheduled request. The t_{wait} period corresponds to the difference between the time of scheduling a request and the end of the current SCL (max_SCL in the worst-case).

These conservative guarantees are applicable to most real-time memory controllers discussed in this paper.

5.2 Impact of Conservative and Aggressive Power-Down Strategies

To explain the implications of both the conservative and aggressive strategies, we illustrate a use-case with four requesters (R1, R2, R3 and R4) connected to a Round-Robin arbiter, as shown in Figure 5. Each requester is provided with a initial service latency bound (Θ) and a bandwidth guarantee (β) based on a rate guarantee (ρ).

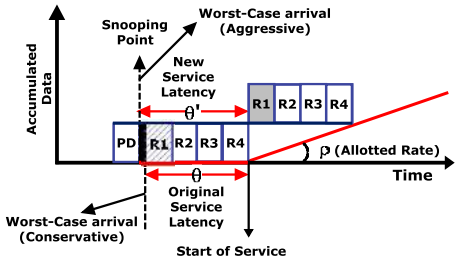

Figure 5: Impact on Latency and Rate Guarantees

We first re-visit the condition for deriving the worst-case initial service latency bound for requester R1 (given by Θ). Consider that R1 arrives at the arbiter one clock cycle after the end of the current scheduling interval, depicted in Figure 5 as worst-case arrival. In this case, it would miss out on being scheduled in its first service slot (indicated by the striped slot). It would eventually get serviced after waiting four service cycles, at the next allotted service slot (indicated by the shaded slot), with a guaranteed rate of service ρ at a guaranteed bandwidth β. Hence, its worst-case latency bound, Θ, includes four max_SCLs, apart from the t_{Ref} and t_{wait} discussed in Equation (7).

In the case of conservative power-down, the SCLs are not modified, since power-up is always completed within the scheduling interval, which is given by the shortest service cycle. Hence, max_SCL remains unaltered and the bandwidth guarantee (β) does not change. The initial service

latency bound is also maintained as is, since any request arriving before the end of scheduling interval in the idle service cycle is scheduled as before, when no power-down was used.

In the case of aggressive power-down, a power-down (PD in Figure 5) was issued during the idle period, preceding R1's request for service. If R1 arrives before the snooping point, it is scheduled during its first available service slot (striped slot). If R1 arrives after the scheduling interval, as discussed above, it is scheduled after waiting four service cycles. However, if R1 arrives after the snooping point and before the end of scheduling interval, it also misses out on its first service slot, since the next slot is already scheduled to be in power-down. But, after waiting over the next four service cycles and any memory-generated refresh, R1 would get serviced at the next allotted service slot (indicated by the shaded slot), with the guaranteed rate of service ρ at a guaranteed bandwidth β. This is the same, as the case when R1 arrives one clock cycle after the scheduling interval, as discussed above. The only difference in this scenario is that, the worst-case initial service latency bound Θ would increase marginally by the worst-case power-up transition time t_{PUP_max}, as shown in Equation (8).

$$\Theta' = \Theta + t_{PUP\_max} \tag{8}$$

Since powering-up of memory is not allowed beyond the snooping point, the power-up is always completed by the end of scheduling interval in the idle service cycle. Hence, the worst-case bound on *max_SCL* (shown in Equation (3)) is not affected by the power-up and hence, the net memory bandwidth (*net_BW*) and the bandwidth-guarantee (β) do not change. To quantify the increase in Θ (shown in Equation (8)), we consider service cycle lengths from the illustration in Figure 3 with request size of 64 bytes. The original Θ for requester R1 in the presence of three interfering requesters (R2, R3 and R4) is derived as 203 clock cycles (cc) using Equation (7), where t_{Ref} is 44 cc for 1Gb DDR3-800. The increased Θ' is calculated as 208 cc (t_{PUP_max} = 5 cc), thus, showing marginal increase in latency bounds (2.4%).

In conclusion, the initial service latency hit of t_{PUP_max} is observed only once per busy period and only by the requester waking up the memory from power-down. Also, there is no impact on the requester's bandwidth guarantee.

5.3 Impact of Speculative Strategies

In this subsection, we derive the impact of speculative power-down policies on latency and bandwidth guarantees.

A *speculative power-down strategy* can be defined as one that powers-down the memory whenever it is idle and *allows it to power-up even after the snooping point in the idle service cycle.* In the worst-case, a request may arrive at the last clock cycle of the idle service cycle and hence, the power-up transition time (t_{PUP_max}) gets added to the SCL of the following request, which may originally have been *max_SCL* in length. This impact on *max_SCL* as a result of a speculative power-up, is shown in Equation (9). This reduces the net memory bandwidth (*net_BW*) and thereby, the bandwidth guarantee (β) provided by the memory controller, as shown in Equations (4), (5) and (6). It also increases Θ in the presence of 'x' interfering requesters, by $t_{PUP\_max} \times (x + 1)$.

$$max\_SCL' = max(t_{PUP\_max} + t_{RD}, t_{PUP\_max} + t_{WR}, max\_SCL) \tag{9}$$

Using the SCLs from Figure 3, *max_SCL* increases to 42 cc from 37 cc (by 13.5%). As a result, the service latency bound increases to 228 cycles, showing a larger increase (around 12.3% using Equation (7)) compared to the aggressive strategy. Most importantly, the net memory bandwidth reduces from 681 MB/s to around 599 MB/s and the bandwidth guarantee (β) reduces from around 170.27 MB/s to 149.72 MB/s (around 12.1% using Equation (5)), which is unacceptable for real-time memory controllers, since it results in an inefficient use of the already scarce memory bandwidth.

Moreover, the bandwidth and latency impact of the speculative policy depends on the number of requesters accessing the memory and gets worse with an increase in the same.

6. POWER-DOWN MODE SELECTION

In this section, we present a power-down mode selection algorithm that determines the most appropriate mode of power-down (fast exit or slow exit) based on the state of the memory and idle service cycle length. Using the power-down equations presented in [13], and the current and voltage numbers from SDRAM datasheets [17], the algorithm evaluates the different power-down modes (fast exit or slow exit, active or precharged) and selects the best power-down mode with the least energy consumption.

To employ Algorithm 1, we derive a 'power-off' (t_{Off}) period for the entire power-down request including the transitions in and out of the power-down mode ($t_{TRANS} + t_{PD} + t_{PUP}$ in Figure 1), equal to the idle service cycle length (*min_SCL*). We then forward this information to the algorithm, along with the memory state information (precharged or active), which then selects the most energy-efficient power-down mode for the given system configuration. If there can be no energy savings with any of the power-down modes, the algorithm opts for no power-down (No_PD).

Algorithm 1 Power-Down Mode Selection

Require: mode_select(t_{Off}, *mem_state*)
1: **if** $t_{Off} > t_{CKE} + t_{XPDLL} - t_{RCD}$ **then**
2: {Comment: Minimum PRE Slow Exit Duration}
3: **if** *mem_state* == *PRE* **then**
4: *Mode* ←Min_Mode(E(t_{Off}, *S_PRE*), E(t_{Off}, *F_PRE*))
5: **else**
6: *Mode* ← *F_ACT*
7: **end if**
8: **else if** $t_{Off} > t_{CKE} + t_{XP}$ **then**
9: {Comment: Mimimum ACT/PRE Fast Exit Duration}
10: **if** *mem_state* == *PRE* **then**
11: *Mode* ←Min_Mode(E(t_{Off}, *F_PRE*), E(t_{Off}, *No_PD*))
12: **else**
13: *Mode* ← *F_ACT*
14: **end if**
15: **else**
16: *Mode* ← *No_PD*
17: **end if**
18: **return** (*Mode*)

It should be noted that the algorithm presented here is for DDR3 memories. For DDR2, the appropriate power-down modes and timings described in [11], must be used.

7. EXPERIMENTS AND RESULTS

In our experiments, we employ a 1Gb Micron DDR3-800 [17] memory and four common media applications: (1) H.263 encoder, (2) EPIC Encoder, (3) JPEG Encoder and (4) MPEG2 Decoder. Using these applications, we evaluate the two proposed real-time power-down strategies and a speculative power-down policy with respect to energy savings, average execution time, initial service latency bounds and bandwidth guarantees. We also derive the theoretical best-case energy savings when using power-down by performing memory trace post-processing. Once the trace is obtained, we manually insert a power-down request at the start of every idle period and power-up the memory in-time for the next transaction to be served, in order to avoid any impact on execution times and performance guarantees.

7.1 System and Experiments Setup

We executed the four test applications independently on the Simplescalar simulator [20] with a 16KB L1 D-cache, 16KB L1 I-cache, 128KB shared L2 cache and 64-byte cache line configuration. We filtered out the L2 cache misses, and obtained a trace of the transactions meant for the SDRAM

memory. To simulate four requesters in our experimental setup (similar to the illustration in Section 5.2), we employed the traces from these four applications on four trace players in a SystemC model of our real-time MPSoC platform [19]. We forwarded these transactions to our real-time SDRAM memory controller [3], fitted with a Round-Robin arbiter. To obtain 64-byte access granularity for DDR3-800, we used a BL of 8 words, each 2 bytes long, with a BC of 4, interleaving over 1 bank, for most power-efficient memory accesses [13]. For all our power analysis, we employed our open-source DRAM energy estimation tool [21] based on the power model presented in [13], and the current and voltage numbers from SDRAM datasheets [17].

7.2 Results and Analysis

In our first experiment, we analyze the impact of the different power-down policies on total memory energy consumption when executing the four application traces concurrently. In doing so, we also observe the average-case impact on total execution time, due to these power-down policies. We compare the conservative (CPD) and aggressive (APD) power-down strategies against the theoretical best-case power-down, no power-down (No PD) and the speculative power-down (Spec PD) policies, as depicted in the graph in Figure 6. We observed that the conservative strategy saves around 42.1% of total memory energy, compared to using no power-down, without impacting the execution times. We also observed that the aggressive power-down strategy saves 51.3% of the total memory energy (very close to the theoretical best-case of 51.4%), at a marginal increase of 0.25% (approximately 510 μs) in the total execution time. The speculative power-down policy saves around 51.1% of the memory energy consumption, but at an increase of about 1.32% (approximately 2640 μs) in the total execution time.

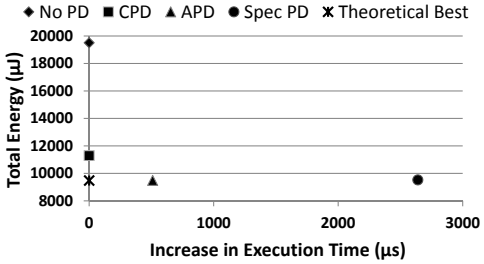

Figure 6: Total Energy & Penalties using Strategies

In our second experiment, we analyze the difference in the energy savings, and the bandwidth guarantee (β) and initial service latency bound (Θ) of the different policies, as depicted in the graph in Figure 7. The data labels indicate the energy consumption, and the bandwidth and latency guarantees of the different policies.

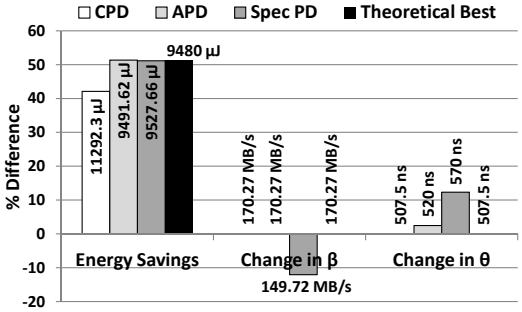

Figure 7: Energy Savings & Latency-Rate Impact

As can be observed, the aggressive and speculative power-down strategies save almost as much energy as the theoretical best power-down solution. However, in the case of the aggressive power-down strategy, the service latency bound (Θ) increases only marginally by around 2.4% to 520 ns, whereas it increases by around 12.3% to 570 ns in the case of the speculative power-down policy. Moreover, the bandwidth guarantee (β) of the speculative power-down policy takes a large of around 12.1% and reduces to 149.72 MB/s from the initial 170.27 MB/s, while it is maintained by the aggressive power-down policy at 170.27 MB/s. This decrease in bandwidth-guarantee occurs for any speculative power-down policy that decides to power-up the memory after the snooping point, since it increases *max_SCL* by 13.5% and decreases *net_BW* by 12.1%, as shown in Section 5.3. Both the proposed bandwidth-neutral strategies can be employed at run-time by any real-time SDRAM memory controller.

8. CONCLUSION

This paper presented two run-time power-down strategies that reduced SDRAM memory energy consumption and yet guaranteed real-time memory performance. The conservative strategy provided significant energy savings of around 42.1% when running traces from four media applications, without impacting the guaranteed latency bound and bandwidth. The aggressive strategy provided higher energy savings of around 51.3% (close to the theoretical best of 51.4%) and only marginally increased the latency bounds by 2.4%, while still preserving the original guaranteed bandwidth. This paper also showed that a speculative power-down policy cannot do any better than the aggressive strategy in terms of energy savings and would also increase the latency bounds by 12.3% and reduce the guaranteed bandwidth by 12.1%, which is unacceptable for real-time memory controllers. Finally, this paper also presented an algorithm to select the most energy-efficient power-down mode at run-time for both the strategies, thereby providing a complete power-down solution for all real-time memory controllers using LR arbiters.

9. REFERENCES

[1] O. Vargas, *Achieve minimum power consumption in mobile memory subsystems*, EE Times Asia, March 2006.

[2] B. Jacob et al., *Memory Systems: Cache, DRAM, Disk*, Morgan Kaufmann Publishers, 2007.

[3] B. Akesson et al., *Architectures and Modeling of Predictable Memory Controllers for Improved System Integration*, In Proc. DATE 2011.

[4] M. Paolieri et al., *An Analyzable Memory Controller for Hard Real-Time CMPs*, IEEE Embd. Sys. Letters, Vol.1, No.4, 2009.

[5] A. Burchard et al., *A Real-Time Streaming Memory Controller*, In Proc. DATE 2005.

[6] S.A. Edwards et al., *A Disruptive Computer Design Idea: Architectures with Repeatable Timing*, In Proc. ICCD 2009.

[7] C. Pitter, *Time-predictable memory arbitration for a Java chip-multiprocessor*, In Proc. JTRES 2008.

[8] J. Reineke et al., *PRET DRAM Controller: On the Virtue of Privatization*, In Proc. CODES+ISSS 2011.

[9] ITRS, *http://www.itrs.net*

[10] JEDEC, *DDR3 SDRAM Standard*, JESD79-3E, 2010.

[11] JEDEC, *DDR2 SDRAM Standard*, JESD79-2F, 2009.

[12] D. Stiliadis et al., *Latency-rate servers: a general model for analysis of traffic scheduling algorithms*, IEEE Trans. on Netw., Vol. 6, No. 5, 1998.

[13] K. Chandrasekar et al., *Improved Power Modeling of DDR SDRAMs*, In Proc. DSD 2011.

[14] B. Akesson et al., *Memory Controllers for Real-Time Embedded Systems*, Springer, 2011.

[15] V. Delaluz et al., *Hardware and Software Techniques for Controlling DRAM Power Modes*, IEEE Trans. on Comp., Vol.50, No.11, 2001.

[16] I. Hur et al., *A comprehensive approach to DRAM power management*, In Proc. HPCA 2008.

[17] Micron, *1Gb: X4, X8, X16 DDR3 SDRAM*, Rev. 2010.

[18] S. Goossens et al., *Memory-Map Selection for Firm Real-Time Memory Controllers*, In Proc. DATE 2012.

[19] A. Hansson et al., *CoMPSoC: A template for composable and predictable multi-processor system on chips*, ACM TODAES, Vol. 14, No. 1, 2009.

[20] D. Burger et al., *The SimpleScalar tool set, Version 2.0*, ACM SIGARCH Comp. Arch. News, Vol. 25, No. 3, 1997.

[21] K. Chandrasekar et al., *DRAMPower: Open-source DRAM power & energy estimation tool*, www.es.ele.tue.nl/drampower