A Predictor-based Power-Saving Policy for DRAM Memories

Gervin Thomas*, Karthik Chandrasekar†, Benny Åkesson‡, Ben Juurlink* and Kees Goossens‡
* Technische Universität Berlin, Department of Computer Engineering and Microelectronics
Embedded Systems Architecture, Berlin, Germany
† Technische Universiteit Delft, Department of Computer Engineering, Delft, The Netherlands
‡ Technische Universiteit Eindhoven, Department of Electrical Engineering
Electronic Systems, Eindhoven, The Netherlands

***Abstract*—Reducing power/energy consumption is an important goal for all computer systems, from servers to battery-driven hand-held devices. To achieve this goal, the energy consumption of all system components needs to be reduced. One of the most power-hungry components is the off-chip DRAM, even when it is idle. DRAMs support different power-saving modes, such as self-refresh and power-down, but employing them every time the DRAM is idle, reduces performance due to their power-up latencies. The self-refresh mode offers large power savings, but incurs a long power-up latency. The power-down mode, on the other hand, has a shorter power-up latency, but provides lower power savings.**

In this paper, we propose and evaluate a novel power-saving policy that combines the best of both power-saving modes in order to achieve significant power reductions with a marginal performance penalty. To accomplish this, we use a history-based predictor to forecast the duration of an idle period and then either employ self-refresh, or power-down, or a combination of both power saving modes. Significant refinements are made to the predictor to maximize the energy savings and minimize the performance penalty. The presented policy is evaluated using several applications from the multimedia domain and the experimental results show that it reduces the total DRAM energy consumption between 68.8% and 79.9% at a negligible performance penalty between 0.3% and 2.2%.

***Index Terms*—Predictor-based Power Saving Policy, Predictor, DRAM-Memory, Self-Refresh, Power-Down.**

I. Introduction

The power/energy consumption is an important constraint for all kinds of computing systems, not only for battery-powered embedded systems, but also for high-performance servers and any computing system in between. Battery-driven embedded systems, such as cell phones, have limited power budgets as well as high performance requirements, and these requirements do not go hand in hand. High-end server systems, on the other hand, also require the energy to be reduced because it brings down the operating costs and cooling effort.

DRAM memories contribute significantly to the overall system energy consumption. For example, memory energy consumption in mobile devices is up to 20% [22] and in data center servers up to 25% [13]. The DRAM memory energy consumption profile in [3] and [4] show that DRAMs consume significant amounts of power even when they are idle. To reduce DRAM energy consumption during idle periods, different power-saving modes are available, such as *power-down* and *self-refresh*. The drawback of the self-refresh mode is that it takes several clock cycles to power up the DRAM, whereas that of the power-down mode is that it saves much less power than the former.

To make this discussion more concrete, let us consider a $1\,\mathrm{Gb}$ DDR3-800 MICRON memory [12]. This memory draws around $50\,\mathrm{mA}$ of current when idle, which corresponds to about $75\,\mathrm{mW}$ of power. DDR3 memories support two important power-saving modes, namely power-down and self-refresh. The power-down mode reduces the power consumption to $18\,\mathrm{mW}$, while the self-refresh mode brings it down to $9\,\mathrm{mW}$ [12]. Hence it is possible to reduce the power consumption during idle periods by factors of 4.1 (76%) and 8.3 (88%), respectively.

Both modes, however, incur a performance penalty, due to their power-up latencies. For the $1\,\mathrm{Gb}$ MICRON DDR3-800 memory, the power-down mode has a relatively small penalty of around $25\,\mathrm{ns}$ (10 memory clock cycles), while the self-refresh mode has a very large penalty of about $1280\,\mathrm{ns}$ (512 memory clock cycles) [7]. Thus the larger the power saving, the larger the power-up latency and hence the performance penalty, and a trade-off needs to be made.

In order to reduce power dissipation while not incurring a large performance penalty, this paper employs and hones a generic history-based predictor to anticipate the length of the next memory idle period. Depending on the predicted idle period length, the presented power-saving policy employs either power-down, or self-refresh, or a combination of both power-saving modes. Furthermore, in an effort to completely avoid the power-up latency, the power-saving policy uses a conservative prediction to power up the memory just in time before it will be accessed again. The three main contributions of this paper can be summarized as follows:

1) We significantly extend and fine tune a versatile prediction algorithm to be able to apply it to the problem at hand. For example, to be able to apply the prediction algorithm to the problem of reducing DRAM energy consumption, levels of idle period lengths need to be introduced because the idle period lengths vary enormously (up to 4-5 orders of magnitude).
2) We present a novel power-saving policy based on the fine-tuned predictor that, depending on the predicted duration of the idle period, employs either power-down, or self-refresh, or a combination of both power-saving modes.

Furthermore, to avoid powering down the memory during short idle periods, which would hardly save any power but incur a large penalty, a *time-out* strategy is employed.

3) We evaluate the proposed power-saving policy using several applications from the multimedia domain. Experimental results, obtained by employing a trace player that simulates the application behavior in a SystemC model of an MPSoC, show that we save between 68.6% and 79.9% of total memory energy with a marginal increase of execution time between 0.3% and 2.2%.

The rest of this paper is organized as follows, Section II presents a brief overview of related work that employ prediction for different components of an MPSoC, as well as work targeted at reducing DRAM energy consumption. Background information, such as the baseline prediction algorithm and basic DRAM operations and their power-saving modes, is given in Section III. Section IV describes how the baseline prediction algorithm needs to be extended and fine tuned in order to be able to apply it to the problem of reducing DRAM energy consumption. Based on the modified prediction algorithm, the proposed power-saving policy is presented in detail in Section V. The proposed policy is experimentally evaluated in Section VI. Finally, Section VII summarizes and highlights the contributions of this work and presents our final conclusions.

II. Related work

Predictors have been used in many areas of MPSoC research. In the Networks On Chip domain, [14], [20] used predictors to forecast end-to-end traffic. In [20], a history-based predictor is used to forecast traffic patterns by searching for similar traffic shapes in a history buffer. In [14], a model based on state space representation is used to predict the availability of input buffers in routers.

In DRAM research, predictors have been used to reduce memory access times for DRAMs connected to MPSoCs. In [19], [23], a predictor was employed to reduce precharges and activates and thereby reducing the average DRAM access latency. This forecast is used to determine whether an open DRAM row should be closed or kept open and which row to open next to minimize the average access latency. Similarly in [1], a predictor is used to track the number of accesses to a given DRAM page to predict DRAM locality to make page closing decisions. The work in [15] proposed a dynamic memory mode control scheme with a predictor to predict whether the next memory reference causes a page hit or not. This is used to exploit locality and reduce the number of activates and precharges to a bank. However, all of these solutions that used predictors, targeted active power reduction and not idle power reduction.

Other works have proposed DRAM power reduction solutions without using explicit prediction logic. For instance in [11], a DRAM precharge policy based on address analysis is presented. Statistical information from instructions waiting to access the memory are collected and analyzed to determine the next bank access and decide on which bank to precharge, thus reducing the average memory latency. In [10], the authors proposed a hardware prefetching technique assisted by static (design time) analysis of data access patterns for efficient data prefetching. However, this idea would only be useful in improving performance of caches and can lead to increased main memory power consumption, due to the mispredicted prefetches. [21] proposed combining read/write multiple times within a single activate-precharge pair to obtain significant energy savings and [8] achieves low power consumption in DRAM memories by changing the processor and memory clock frequencies. However, these solutions also target active power reduction.

In [6], the authors proposed reducing idle memory power by extending by using a compiler-directed selective power-down and a hardware-assisted prediction-based run-time power-down. However the former is not suitable for run-time use and the latter only employs power-down mode and does not exploit the self-refresh mode.

The hesitation of using the self-refresh mode stems from the large power-up latency associated with it. As a result, there has been no known effort to combine the use of prediction and self-refresh modes to obtain memory idle energy savings. In this paper, we propose an efficient power-saving policy that with the help of a predictor employs both the self-refresh and power-down modes to reduce idle energy consumption significantly while keeping the penalty negligible.

III. Background

This section introduces the history-based predictor used in this paper and briefly discusses basic DRAM operations and their different power-down modes.

A. Predictor

The generic history-based predictor used in this paper was originally proposed in [20], where it was used to forecast traffic pattern for rerouting in networks. In this paper, we modify and employ this predictor to forecast memory idle periods. The generic predictor is briefly introduced here.

The predictor probes a history of data points, considering a current set of reference data points and searches for similar patterns in the history. This is used to predict the future set of data points. The prediction algorithm considers the latest set of m data points as a *reference pattern* (shown in Figure 1) from the history set $Y(y_0, y_1, \cdots, y_n)$ of $n+1$ data points (where $m < n$) and searches for similar patterns of the reference pattern length in the past. A parameter *width* (w) is used to identify whether a set of past data points fits the reference pattern. In the set of data points from the history, if a particular data point differs by more than $|w/2|$ that pattern is neglected by the predictor to forecast the next data point. The algorithm continues to compare the reference pattern with all data points in the history moving one data point at a time. If the algorithm finds many similar patterns in the history, it forecasts the next data point by considering all these patterns.

The predictor builds up a history before forecasting future data points. The latest set of reference data points between (y_n) and (y_{n-m+1}) are compared with the different patterns from history. The algorithm also defines a parameter *history length*, which gives the limit on the number of past data points

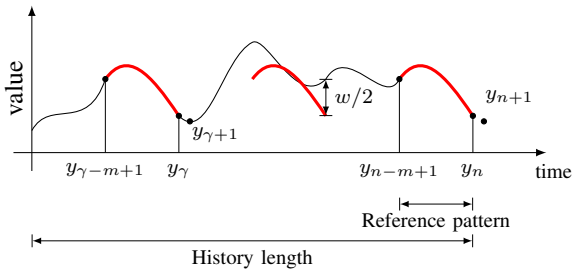


Fig. 1. Working of the Predictor

to be taken into account for this prediction. In Figure 1, the history includes all shown data points, but any size of the history length can be used for the analysis. If there is a pattern of length m in the past that is very similar to the reference pattern, like the pattern between $(y_{\gamma-m+1})$ and (y_γ), the algorithm predicts that the next future data point (y_{n+1}) is very similar to the data point that follows the past pattern $(y_{\gamma+1})$. As stated before, the matching to past data points is not limited to just one reference pattern in the history. If multiple past data patterns are similar to the current one, based on the similarity, the weighted sum of the matching data points is calculated to forecast the next data point [20].

In this paper, we adapt and extend this versatile generic predictor to employ it for our memory power optimization problem. We employ memory idle period lengths as data points and predict lengths of future memory idle periods. The predictor needs further extensions because the statically computed width parameter can produce wrong estimations for memory idle periods with large variations. The improvements and extensions of this predictor are explained in Section IV.

B. DRAM Basics and Power-Saving Modes

Dynamic Random Access Memory (DRAM) is the most used type of main memory in mobile phones, laptops, gaming consoles and servers. DRAMs consist of several banks, where data are stored in rows and columns. When reading or writing data from or to the memory, the data from any given row is moved to the row buffer, and then to the I/O buffers to complete the data transfer. If data is retained in the row buffer after the operation is finished, it keeps the memory in the active state. If it is moved back to the memory row, it moves the memory to the precharged state. The memory can be idle in either of these two states.

Each bit of data is stored as charge in a capacitor. The capacitors leak the charge over time. The memory has to be refreshed at regular intervals to avoid losing data. Therefore, DRAM is a volatile memory and data is lost when the memory is turned off. However, when the memory is on, it is not used all the time and depending on the application there can be several memory idle periods of varying lengths. The memory consumes a significant amount of energy during these idle periods, which can be reduced using power-saving modes, such as power-down or self-refresh. The power-down mode can be employed in either the active or precharged state, while the self-refresh mode can be employed only in the precharged state. In general, the precharged state power-down saves more power than the active state power-down. For simplicity in this paper, we make sure the memory is in the precharged state at the end of every read or write transaction, making it easier to employ the precharge power-down and the self-refresh modes.

Comparing these two modes, the power-down mode saves less power than the self-refresh mode, but the memory can power-up from the power-down mode much faster than from the self-refresh mode. The goal of this work is to use self-refresh as often as possible to maximize the power savings while keeping the performance penalty low.

For our analysis, we consider a MICRON 1 Gb DDR3-800 memory device [12]. For this memory, the current consumed during power-up cycles and in the precharged idle mode (denoted by I_{DD2N}) is 50 mA. When in self-refresh mode (SR), the memory draws a current of 6 mA (denoted by I_{DD6}) and needs X_{SDLL} clock cycles (equal to 512 cc) to power-up the memory. In precharged power-down mode (PD), 12 mA of current is consumed (denoted by I_{DD2P0}) and the power-up latency is given by X_{PDLL} (equal to 10 cc).

IV. Extending the Predictor

This section extends and fine tunes the generic predictor from [20] to serve two purposes: (1) For efficient selection of power-saving modes for any given idle period length, (2) To apply the predictor to forecast idle periods in DRAMs.

A. Efficient Power-Saving Mode Selection

Selecting the best power-saving mode depends on the length of the idle period. For short idle periods (up to a few thousand clock cycles for DDR3-800), the power-down mode is more gainful because of its short power-up latency compared to self-refresh. For longer idle periods, the self-refresh mode becomes more gainful because of its lower power consumption that compensates for its long power-up latency.

To estimate the minimum idle period length at which the self-refresh mode saves more power compared to power-down (including powering-up time), we need to consider the energy consumption when the memory is in the power-down or the self-refresh mode, in addition to the energy consumption during their corresponding power-up cycles. This minimum idle duration defines the *self-refresh threshold* (*SRT*), employed by the prediction algorithm in this paper.

To derive the *SRT*, we estimate energy consumption when employing the self-refresh (E_{SR}) and power-down (E_{PR}) modes (including their power-up latencies) in SRT clock cycles. The self-refresh mode keeps the memory in the self-refresh state for $SRT - X_{SDLL}$ clock cycles, and powers-up the memory during X_{SDLL} clock cycles (its power-up latency). During the self-refresh period, I_{DD6} current is drawn by the memory, and during the X_{SDLL} cycles, it draws I_{DD2N} current, as shown in Equation (1). Similarly, when the power-down mode is selected, the memory is in the power-down state for $SRT - X_{PDLL}$ clock cycles and consumes I_{DD2P0} current. During its power-up period of X_{PDLL}, it consumes I_{DD2N} current, as given by Equation (2). In these equations, V_{DD} corresponds to the supply voltage and clk to the clock

period of the memory clock.

$$E_{SR} = [I_{DD6} \cdot (SRT - X_{SDLL})] \cdot V_{DD} \cdot clk + [I_{DD2N} \cdot X_{SDLL}] \cdot V_{DD} \cdot clk \quad (1)$$

$$E_{PD} = [I_{DD2P0} \cdot (SRT - X_{PDLL})] \cdot V_{DD} \cdot clk + [I_{DD2N} \cdot X_{PDLL}] \cdot V_{DD} \cdot clk \quad (2)$$

Equating them and solving for SRT, as shown in Equation (3), gives the minimum idle period length (rounded up) when self-refresh saves more energy than power-down. For the 1 Gb Micron DDR3-800 memory discussed in Section I, *SRT* equates to 3691 clock cycles.

$$SRT = \frac{X_{SDLL} \cdot (I_{DD6} - I_{DD2P0})}{I_{DD6} - I_{DD2P0}} - \frac{X_{PDLL} \cdot (I_{DD2N} - I_{DD2P0})}{I_{DD6} - I_{DD2P0}} \quad (3)$$

B. Applying the Predictor to DRAMs

This section describes how we significantly extend and fine tune the generic predictor described in Section III-A.

To adapt the predictor to predict idle periods in DRAM memories, we define a set of values for the history length, the reference pattern length and the width parameters (see Section III-A). For this, we use the inferences of the impact of these parameters on prediction accuracy from [20]. We do a design-time analysis using a combination of useful values for these parameters and statically select the ones with the best prediction accuracy for a particular application.

After selecting the parameters for the problem at hand, we observed that the width parameter, which defines the allowed difference between the reference pattern and the patterns from the history, has an adverse effect on the prediction accuracy if there are large variations in the idle period lengths. This is due to the fact that for a set of highly variable idle periods, a small width ignores idle patterns in the history even for a relatively small variation. To resolve this issue, we propose an extension to the predictor to be able to accurately predict length of idle periods even when there are large variations.

As an extension, we introduce a new parameter to the prediction algorithm, called *levels*. *Levels* are used to classify the different idle period lengths within a set of pre-defined ranges. We employ the different levels of the idle periods (representing their lengths) as data points in the generic prediction algorithm, introduced in Section III-A. The width parameter is now applied on the allowed difference in levels and not on exact idle period lengths and hence, a small value for the width parameter is sufficient. We show an illustrative example of using levels to predict idle period lengths in Figure 2.

As can be seen in the figure, the range from $[x_{i-1}, x_i]$ refers to level l_i. Assume $SRT = 3691$ as derived in Section IV-A. To assure that that self-refresh mode is always better than employing a power-down, level 1 includes all idle periods of lengths from 0 cc to 3690 cc ($SRT - 1$ cc). Level 2 includes all idle periods from 3691 cc (SRT) to 7381 cc. The bounds of every sub-sequent level is derived by doubling the length

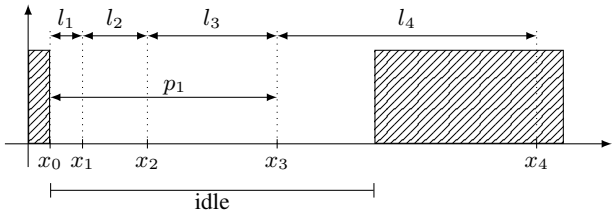


Fig. 2. Idleness Prediction on levels

of the current one. Therefore, level 3 includes idle period lengths from 7382 cc to 14762 cc. By choosing levels in this manner, we employ level 1 to indicate that self-refresh is not gainful and all other levels where self-refresh is the favoured power-saving mode. The introduction of levels reduces the variation in idleness to range from level 1 to level 7 instead of 0 cc to 236162 cc. Hence, a small width parameter can be successfully employed. However, since the prediction is performed on levels, as shown in Figure 2, predicting a level l_i equates to a conservative value x_{i-1}, since we use the lower bound of the range l_i as the predicted value, to reduce the misprediction penalty. The history now consists of different levels of idle periods in the past and the pattern comparison is done on the basis of levels. Note that since levels represent ranges of idleness, all predictions may not be accurate at the single clock cycle level. Also since the predictor forecasts conservatively, 100% of all the idle cycles in some idle periods may not exploited by the prediction.

V. Power Saving Policy

In this section, we propose a novel power-saving policy that employs the prediction algorithm form [20] in combination with a time-out strategy to identify memory idleness. This enables the use of either the self-refresh mode or the power-down mode or both, while avoiding or reducing the penalty cycles. The standard Time-Out strategy [16], briefly discussed in Section V-A, is used to weed out any speculative usage of the self-refresh mode. Additionally, we employ the prediction algorithm multiple times in every idle period for effective use of the self-refresh mode, described in Section V-B, and also use the power-down mode speculatively when some idle cycles are not exploited using the self-refresh mode (described in Section V-C).

A. Time-Out

To avoid unnecessary usage of the self-refresh mode, the standard time-out strategy is used that waits for a system to be idle for a pre-defined time-out interval before powering it down. The idle periods of the memory can vary between a few and several thousand clock cycles, as explained in Section IV-B. The self-refresh mode is not gainful for short idle periods because of its long power-up latency. For short idle periods, it always results in a performance penalty and no energy gain. Hence, using a time-out interval can avoid the unnecessary power-up penalty for short idle periods.

B. Prediction for Self-refresh

In this subsection, we analyse use of the prediction algorithm for efficiently employing self-refresh in combination

with the time-out strategy described earlier.

The predictor conservatively predicts idle periods longer than *SRT* in order to avoid the penalty cycles when employing the self-refresh mode. This allows the memory to power-up expectantly before the next request arrives. Furthermore, by employing the time-out strategy, the predictor forecasts only idle periods larger than the pre-defined time-out period. If the predictor forecasts an idle period shorter than *SRT*, these idle periods are neglected and the self-refresh mode is not used. It is possible that the predictor under or over-estimates the idle period length. The former is more probable, since the predictor always provides a conservative estimate for the idle period length. How to solve the under/over-estimation problem is explained in the following two subsections.

1) Reasons for Estimation Problems: If the predictor forecasts idle period lengths shorter than the actual idle period, it can be for two reasons. One, since the prediction is done in terms of levels and provides the lower bound of a particular level as the prediction value, it may neglect some idle clock cycles in the corresponding idle period. The second reason for under-estimating the length of idle periods is that very long idle periods can be rare and far apart. If a long idle period has not been predicted before or it is already out of the limited history buffer, the predictor under-estimate because of the missing similar large value in the history. It is also possible that the predictor over-estimates the idle period length due to a mismatch in the history or an unexpected extreme variation in the idleness.

2) Solution for Estimation Problems: To solve the under-estimation problem, we propose to employ *multiple predictions* in a given idle period. This extends the use of the self-refresh mode if possible, and powers-up the memory as late as possible, just in time before it is accessed again. This policy is depicted in Figure 3.

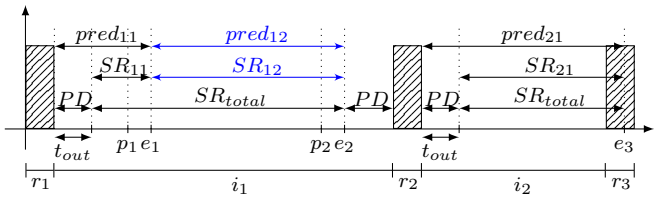


Fig. 3. Multiple predictions for Self-Refresh

In the figure, requests are shown as hatched bars and denoted as r_x and idle periods occurs as i_x. The initial time-out is denoted as t_{out}. After this time-out, the predicted value marked as $pred_{11}$ is checked if it is greater than or equal to level 1 (minimum *SRT* cycles) and if so, a self-refresh (SR_{11}) is scheduled. The memory is expected to start to power up at p_1 to assure that it powers up completely at e_1. However, at p_1 the history is temporarily updated with the elapsed idle cycles and an additional prediction is invoked. If the predictor forecasts that the new prediction is not gainful for an additional self-refresh, the memory continues to power up. However, if the predicted value is still gainful ($pred_{12}$) the memory stays in self-refresh. All predicted self-refresh periods (SR_{11}, SR_{12}) are combined into a longer continuous self-refresh period (SR_{total}). At the end of the real idle period, the temporarily set history value is overwritten by the real idle period length. At every p_i, an additional prediction can be done to extend the self-refresh period. The maximum number of predictions per idle period can be limited to avoid unnecessary penalty due to over-estimations. However, the actual number of predictions is lower than the defined maximum when either the predictor forecasts that a self-refresh is not gainful or a misprediction occurs. As already explained, the prediction is done conservatively on levels and therefore a 100% exploitation of idle cycles in every idle period is not possible using self-refresh.

Figure 3 also shows the idle period i_2 which has an over-estimation problem. After an initial time-out, a self-refresh (SR_{12}) is scheduled based on the length of the predicted value ($pred_{21}$). The predicted value is larger as the actual idle period and a wake-up penalty arises. The wake-up penalty can be either the total power-up latency or a fraction of it. The latter occurs if the memory has already started powering-up when the next request arrives. In this case, the penalty is the difference between the power-up latency and the cycles the memory is already into the power-up.

C. Proposed Power-Saving Policy

As stated in Section V-B, by performing multiple predictions per idle period, it is possible to exploit most of the idle cycles using the self-refresh mode. However, since the prediction is done on levels, a 100% exploitation of idle cycles is not possible for all idle periods. To resolve this issue for the unexploited idle cycles, we propose to combine the multiple predictions for self-refresh with speculative use of the power-down mode, whenever all idle cycles in any given idle period are not exploited by self-refresh. The power-down mode saves a considerable amount of power, though lower than the savings from self-refresh, but also at a much lower power-up penalty (10 cc against 512 cc for self-refresh for DDR3-800). Thus, the proposed policy uses (a) self-refresh, when the prediction exploits all idle cycles of an idle period, (b) power-down, when the prediction forecasts idle periods shorter than SRT clock cycles (level 1) and (c) combination of the two modes to exploit most idle cycles by self-refresh and the rest by the power-down mode, all at a nominal performance penalty.

The idle cycles not covered by the self-refresh prediction include (1) the short idle periods where using the self-refresh mode is not gainful (level 1), (2) the cycles spent during the initial time-out, and (3) the idle cycles not exploited by the prediction and self-refresh due to the conservative estimates on levels. The proposed power-saving policy thus uses prediction for self-refresh and also schedules a speculative power-down for idle clock cycles not covered by self-refresh, thereby covering 100% of the idle cycles by either of the two power-saving modes.

Figure 3 also depicts this proposed power-saving policy. All cycles exploited by power-down are donated as PD. For all idle periods not exploited by self-refresh, the power-down mode is used and only marginally increases the execution time. In other words, scheduling a speculative power down results in a minor penalty for a considerable energy gain.

In the next section, we compare the proposed power-saving policy, which uses multiple predictions for self-refresh or speculative power-down or a combination of both modes against a naive speculative usage of the self-refresh mode using several applications from the multimedia domain.

VI. Experimental Evaluation

The goal of the proposed power-saving policy is to reduce memory energy consumption while only marginally increasing the execution time due to the power-up latencies. In this section, we first briefly introduce the experimental setup and the usage of the predictor in the system setup in Section VI-A.

Later, in Section VI-B, we analyse the impact of (a) the time-out strategy by speculatively employing the self-refresh mode, (b) using multiple predictions for self-refresh in an idle period, and (c) employing the power-down mode speculatively for the idle cycles not exploited by the prediction in the self-refresh mode. For these evaluations, different applications from the multimedia domain like MediaBench [9] are used.

A. Experimental Setup

In our experiments, we employ a 1 Gb Micron DDR3-800 [12] memory and evaluate the proposed predictor-based power-saving policy with respect to energy savings and the impact on execution time. Three different multimedia applications are used: (1) H263 decoder, (2) Ray Tracer, and (3) JPEG encoder. To evaluate the proposed power saving policy, we run each application on the Simplescalar simulator [2] with a 16 kB L1-Data cache, 16 kB L1-Instruction cache, a 256 kB shared L2 cache and 256 B cache line configuration. We filter out the L2 cache misses and obtain the transactions to the DRAM memory. We then employ a trace player to simulate the application behavior in a SystemC simulation model of the CompSOC platform [18]. We forward these transactions from the trace player to a DRAM memory controller [17] in platform, which is modified to employ our predictor and the power saving policy logic.

For all our power analysis, we employed our open-source DRAM energy estimation tool [5] based on the power model presented in [4]. Current and voltage numbers are obtained from the DRAM vendors datasheets [12].

The predictor is placed in the front-end of the memory controller adjacent to the arbiter-bus. An overview of the CompSOC platform including the predictor and the memory controller is depicted in Figure 4.

The predictor monitors the inputs of the bus for incoming requests and records the time stamps of the beginning of the first transaction after every idle period and the end of the last transaction before every idle period. The length of the idle periods are encoded to levels. Using these levels, it builds up a history of levels representing the lengths of idle periods in a history buffer. The predictor uses the contents of the history buffer to forecast the prospective levels, which are decoded to conservative lengths of future idle periods, as described in Section IV-B. Using the inferences from our previous work [20] on the predictor and an initial design-time analysis, we set the history buffer length (hl), the reference pattern length (pl), and the predictor width parameter (w) as

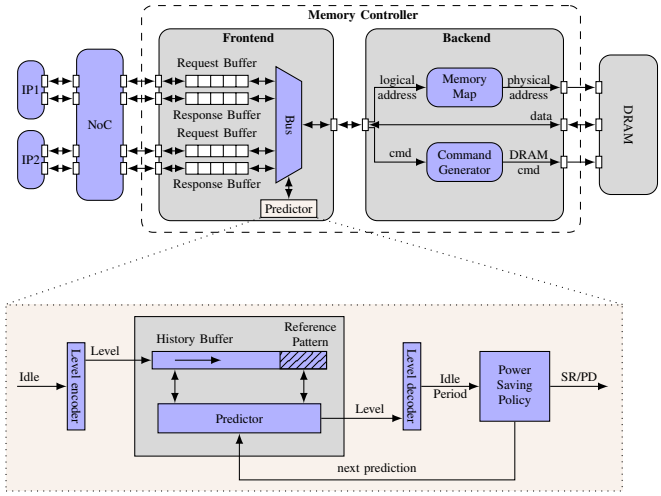


Fig. 4. CompSOC overview including the predictor

TABLE I
Predictor parameters

Application.	hl	pl	w	to	pi
H263 decoder	50	2	4	230	150
Ray Tracer	50	2	4	250	40
JPEG encoder	50	2	4	0	200

depicted in Table I. The table also includes the best initial time-out (to) parameter (see Section V-A) as well as the maximum number of predictor invocations (pi). These two parameters are explained in more detail in Section VI-B.

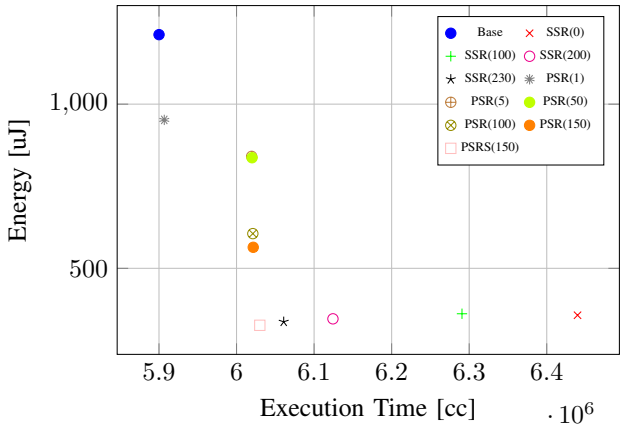
B. Evaluation of the Proposed Policy

In this section, we evaluate the impact of (a) the time-out strategy (Section V-A) on speculative use of the self-refresh mode, (b) using multiple predictions for self-refresh (Section V-B, and (c) employing speculative power-down (Section V-C) for idle cycles not exploited by prediction.

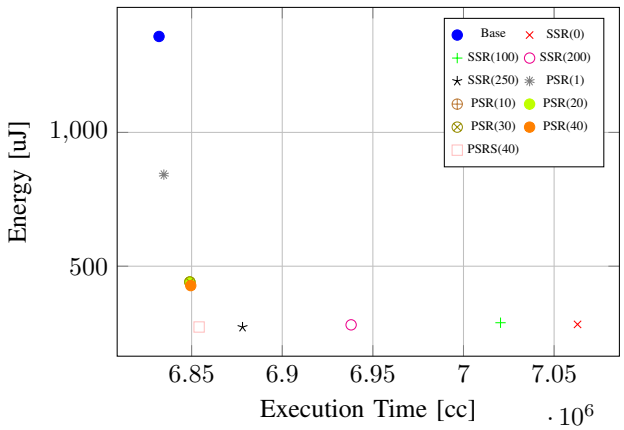
The pareto plots in Figure 5 show the total memory energy consumption and the performance penalty (in the form of impact on execution time) for the different media applications when employing the time-out strategy, multiple predictions with self-refresh, and the proposed power-saving policy that selects between self-refresh, power-down or a combination of both modes for different idle periods. Figure 5a presents the results for the H263 decoder, Figure 5b for the Ray Tracer, and Figure 5c for the JPEG encoder, respectively. Additionally, Table II explicitly presents in percentage as well as factor by which the energy consumption reduces and the execution time increases due to the power-up penalties.

In both Figure 5 and Table II, `Base` corresponds to the baseline results when no prediction or power-saving mode is employed. Therefore, `Base` has the lowest execution time (no penalty) and the highest power consumption.

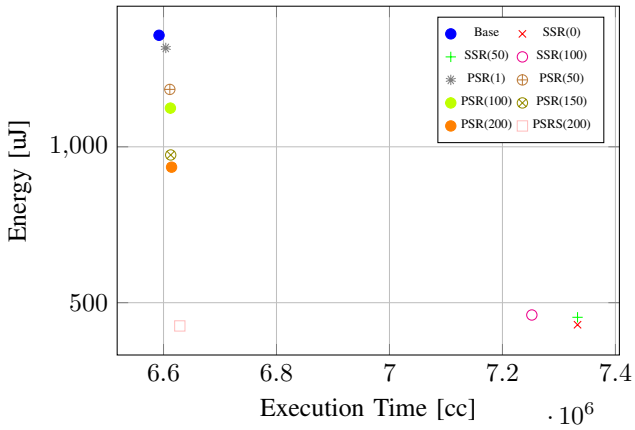
In our first experiment, we compare our solution against speculative use of the self-refresh mode when the memory is idle. This is depicted in Figure 5 and Table II by `SSR(time)` (Speculative Self-Refresh), where time gives the time-out threshold employed. Table I indicates the best time-out threshold values for the three applications. As can



(a) H263 decoder



(b) Ray Tracer



(c) JPEG encoder

Fig. 5. Pareto plot of energy and execution time

be noticed in Figure 5a and 5b, by increasing the time-out threshold up to a defined limit, the speculative self-refresh gives improved energy savings and lower performance penalty. This confirms the usefulness of using the time-out strategy for these two applications, as there are many idle periods shorter than the time-out threshold values and are ignored by the speculative self-refresh by increasing the time-out thresholds. However, in the case of the JPEG encoder (Figure 5c), the

TABLE II
POWER SAVINGS AND PENALTY CYCLES

	H263 decoder		Ray Tracer		JPEG encoder	
	Inc. Execution Time [%]	Savings [%] / Factor	Inc. Execution Time [%]	Savings [%] / Factor	Inc. Execution Time [%]	Savings [%] /Factor
Base	0	0 / 1	0	0 / 1	0	0 / 1
SSR	2.72	72.2 / 3.6	0.67	79.9 / 5	11.2	68.3 / 3.2
PSR	2.04	53.5 / 2.2	0.25	68.5 / 3.2	0.33	31.2 / 1.6
PSRS	2.2	73.1 / 3.7	0.32	79.9 / 5	0.55	68.6 / 3.2

design-time effort identifies that a time-out threshold value greater than zero increases the energy consumption and only marginally reduces the penalty and therefore results in a poor trade-off. Hence, the time-out strategy is not employed for this application. For all applications, the use of the time-out strategy with the speculative self-refresh reduces the energy consumption by factors between 3.2 (68.3%) and 5 (79.9%). However, the speculative use of self-refresh without employing prediction results in the very high penalties and an increase in execution time by up to 11.2%. In short, high energy savings are achieved because most of the idle cycles, except those filtered out during time-out, are covered by the self-refresh mode. The power-up penalties are unavoidable because the self-refresh mode is used speculatively. Note that without employing the time-out strategy speculative use of self-refresh would result in a much higher penalties.

In our next experiment, we evaluate the use of multiple predictions per idle period for employing the self-refresh mode in combination with the previously mentioned time-out strategy. This is represented by `PSR(limit)` (Prediction for Self-Refresh), where the parameter `limit` gives the maximum number of invocations of the predictor in a single idle period. As can be seen in Figure 5, increasing the number of predictions per idle period exploits more idle cycles using the self-refresh mode and reduces the energy consumption. At the same time the prediction is used to wake-up the memory before the next request arrives and therefore avoids most of the penalty observed when using self-refresh speculatively. This can be noticed in Figure 5, where using prediction results in only a small increase of the execution time compared to the `Base` mode and also reduces the energy consumption.

Using design-time analysis, a limit is derived on the maximum number of predictor invocations per idle period, beyond which employing additional predictions is not gainful. This limit indicates the number of predictor invocations when the predictor forecasts that it is no longer gainful to continue in self-refresh or when the predictor starts over-estimating the idleness that results in an increase in performance penalty. The best values for the limit parameter for the different applications are also shown in Table I. Using the prediction for employing self-refresh reduces the energy consumption significantly across the different applications by a factor between 1.6 (31.2%) and 3.2 (68.5%). These savings are lesser than the speculative usage of the self-refresh mode, since the

predictions are done conservatively on levels and therefore cover lesser number of idle cycles using the self-refresh mode compared to the speculative self-refresh. On the other hand, the use of the predictor avoids a lot of penalty cycles compared to the speculative self-refresh mode, which results in a marginal increase of execution time up to 2.04%. The energy savings and the increase of execution time are shown in Table II.

In our final experiment, we evaluate the proposed power-saving policy, which combines the time-out strategy and the predictions for self-refresh and additionally schedules a speculative power-down to maximize power savings, but still avoids most of the power-up penalties. Using this policy, the self-refresh mode is used when the prediction is above level 1 and covers all idle cycles in that idle period, the power-down mode is used for predictions of level 0 and cycles ignored due to the time-out threshold and a combination of both the modes is used when the prediction is inadequate in exploiting all the idle cycles in that idle period using self-refresh alone. The policy is denoted by `PSRS(limit)` (Prediction for Self-Refresh with Speculative power-down), where `limit` corresponds to the maximum number of predictor invocations. Using this policy all idle periods are completely exploited using either the self-refresh mode, or the power-down mode or a combination of both power saving modes. The proposed policy has the highest energy savings for all applications and reduces the energy consumption by a factor between 3.2 (68.6%) and 5 (79.9%). This policy also results in a low performance penalty between 0.32% and 2.2%.This policy harnesses the benefits of the predictor and efficiently combines both the self-refresh and the power-down modes to get maximum energy savings at considerably lower performance penalties compared to using the self-refresh mode speculatively.

VII. Conclusions

In this paper, we have significantly extended and fine tuned a generic prediction algorithm to be able to employ it for reducing DRAM power/energy consumption. Furthermore, based on the prediction algorithm, we have proposed a novel power-saving policy that leverages DRAM idle periods to put the DRAM either in the self-refresh mode or the power-down mode, or a combination of both power-saving modes in a given idle period depending on the predicted duration of the idle period. The power-saving policy, referred to as *prediction for self-refresh with speculative power-down* (PSRS), places the memory in self-refresh mode provided the predicted idle period length is sufficient to save power. Otherwise it exploits the idle period by scheduling a speculative power-down. If the predicted idle period length is shorter than the actual idle period length, it schedules a speculative power-down for the clock cycles not exploited by the prediction. This policy hence exploits all the idle cycles in all idle periods. Experimental results for several multimedia benchmarks have shown that this policy significantly reduces the total DRAM energy consumption with negligible performance penalties when compared to using the self-refresh mode speculatively. The proposed policy results in very high energy savings (between 68.6% and 79.9%) at very marginal performance penalty (between 0.32% and 2.2%).

References

[1] M. Awasthi, D. W. Nellans, R. Balasubramonian, and A. Davis. Prediction Based DRAM Row-Buffer Management in the Many-Core Era. In *Proc. 20th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 183–184, Galveston Island, Texas, October 2011. Poster Track.

[2] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH Comput. Archit. News*, 25:13–25, June 1997.

[3] R. Can, F. Koch, I. Choi, I. Suh, H. Byun, and P. Blumstengel. Save power and improve efficiency in virtualized environment of datacenter by right choice of memory. Technical report, Microsoft Technology Center & Samsung Semiconductor, 2011.

[4] K. Chandrasekar, B. Åkesson, and K. Goossens. Improved Power Modeling of DDR SDRAMs. In *14th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*, pages 99 –108, 2011.

[5] K. Chandrasekar et al. DRAMPower: Open Source DRAM Power & Energy Estimation Tool. *www.es.ele.tue.nl/drampower*, 2012.

[6] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. Hardware and software techniques for controlling DRAM power modes. *IEEE Transaction on Computers*, 50(11):1154–1173, 2001.

[7] JEDEC SST Association. *DDR3 SDRAM Standard*, 2010. JESD79-3E.

[8] Y. Joo, Y. Choi, and H. Shim. Energy exploration and reduction of SDRAM memory systems. In *Proc. 39th Design Automation Conf*, pages 892–897, 2002.

[9] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proc. 30th ACM/IEEE International symposium on Microarchitecture*, MICRO 30, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.

[10] J. Lee, C. Park, and S. Ha. Memory access pattern analysis and stream cache design for multimedia applications. In *Proc. Asia and South Pacific Design Automation Conf the ASP-DAC 2003*, pages 22–27, 2003.

[11] C. Ma and S. Chen. A DRAM Precharge Policy Based on Address Analysis. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*, pages 244–248, 2007.

[12] Micron Technology Inc. *DDR3 SDRAM 1Gb Data Sheet*, 2006.

[13] L. Minas and B. Ellison. *Energy Efficiency for Information Technology: How to Reduce Power Consumption in Servers and Data Centers*. Intel Press, 2009.

[14] U. Y. Ogras and R. Marculescu. Prediction-based Flow Control for Network-on-Chip Traffic. In *Proc. 43rd Design Automation Conference*, DAC '06, pages 839–844, New York, NY, USA, 2006.

[15] S.-I. Park and I.-C. Park. History-based memory mode prediction for improving memory performance. In *Proc. Int. Symp. Circuits and Systems ISCAS '03*, volume 5, 2003.

[16] M. Pedram. Power optimization and management in embedded systems. In *Proc. Asia and South Pacific Design Automation Conference*, ASP-DAC '01, pages 239–244, New York, NY, USA, 2001. ACM.

[17] B. Åkesson and K. Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *Proc. Design, Automation, and Test in Europe (DATE)*, pages 851–856, 2011.

[18] B. Åkesson, A. Molnos, A. Hansson, J. Ambrose Angelo, and K. Goossens. Composability and Predictability for Independent Application Development, Verification, and Execution. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-Chip — Hardware Design and Tool Integration*, chapter 2. Springer, Dec. 2010.

[19] V. Stankovic and N. Milenkovic. DRAM Controller with a Complete Predictor: Preliminary Results. In *Proc. 7th International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Services*, volume 2, pages 593 –596, sept. 2005.

[20] G. Thomas, B. Juurlink, and D. Tutsch. Traffic Prediction for NoCs using Fuzzy Logic. In *Proc. 2nd International Workshop on New Frontiers in High-performane and Hardware-aware Computing (in Conjunction with HPCA-17)*, pages 33–40, San Antonio, Texas, USA, February 2011. KIT Scientific Publishing.

[21] J. Trajkovic, A. V. Veidenbaum, and A. Kejariwal. Improving SDRAM access energy efficiency for low-power embedded systems. *ACM Trans. Embed. Comput. Syst.*, 7:24:1–24:21, May 2008.

[22] O. Vargas. Achieve minimum power consumption in mobile memory subsystems. Technical report, Infineon Technologies AG, 2006.

[23] Y. Xu, A. S. Agarwal, and B. T. Davis. Prediction in Dynamic SDRAM Controller Policies. In *Proc. 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '09, pages 128–138, Berlin, Heidelberg, 2009. Springer-Verlag.