A Generic Implementation of a Quantified Predictor Applied to a DRAM Power-Saving Policy

vorgelegt von
Dipl.-Ing.
Gervin Thomas
geb. in Berlin

von der Fakultät IV - Elektotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Stephan Kreutzer
Gutachter:	Prof. Dr. Ben Juurlink
Gutachter:	Prof. Dr. Dietmar Tutsch
Gutachter:	Prof. Dr. Carsten Gremzow

Tag der wissenschaftlichen Aussprache: 23. Juli 2014

Berlin 2014
D 83

A thesis submitted in partial fulfillment of the requirements for the degree of

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

according to the doctoral degree regulations at the Technische Universität Berlin (23.10.2006).

Department of Computer Engineering and Microelectronics
Embedded Systems Architectures (AES)
Technische Universität Berlin
Berlin

Author Gervin Thomas

Thesis period 01.11.2007 - 10.04.2014

Referees Prof. Dr. Ben Juurlink, Technische Universität Berlin, Embedded Systems Architectures (AES)
Prof. Dr. Dietmar Tutsch, Bergische Universität Wuppertal, Automatisierungstechnik / Informatik
Prof. Dr. Carsten Gremzow, Technische Universität Berlin, Rechnertechnologie

Supervisor Dr. -Ing. Ahmed Elhossini, Technische Universität Berlin, Embedded Systems Architectures (AES)

Declaration

I hereby confirm to have written the following thesis on my own, not having used any other sources or resources than those listed.

Berlin, August 12, 2014

Gervin Thomas

Abstract

Predictors are used in many subfields of computer architecture to enhance performance. Accurate estimations of future system behavior allow to develop policies to improve system performance or reduce power consumption. These policies become more efficient if predictors are implemented in hardware and are able to provide quantified forecasts, i.e. providing more than binary forecasts. One of the most important goals of any computer system, from servers to battery-driven hand-held devices, is the reduction of power and energy consumption. To achieve this, the energy consumption of all system components must be reduced. This is especially important for off-chip DRAM, which consumes a significant amount of energy even when it is idle. Hence, DRAMs support different power-saving modes, such as *self-refresh* and *power-down*. However, employing these power-saving modes each time the DRAM is idle, impacts the performance due to their power-up latencies. The *self-refresh* mode offers large power saving potential, but incurs a long power-up latency. The *power-down* mode, on the other hand, has a lower power-up latency but provides less power savings. Using the most efficient mode depends on the length of the idle period, which is normally unknown.

This thesis presents and evaluates a history-based predictor which produces quantified forecasts. A software version and a hardware implementation of the prediction algorithm are implemented and analyzed. A complete design space analysis of the predictor is presented to determine parameter sets achieving an accuracy rate over 96%. Moreover, a generic and fully synthesizeable design is presented in VHDL and implemented on an FPGA. A complete scalability analysis of the hardware predictor shows that the design has a low device utilization and can be clocked by over 210 MHz.

Using the impact of the previous analyses, a predictor-based power-saving policy is presented for the reduction of memory power consumption. This power-saving policy combines the two power-saving modes, *self-refresh* and *power-down*, in order to achieve significant power reductions with marginal performance penalties. The history-based predictor is then used to forecast the duration of idle periods and apply either *self-refresh*, *power-down*, or a combination of both power-saving modes. The policy is evaluated using applications from the multimedia domain. The experimental results exhibit that it reduces the total DRAM energy consumption between 43.4% and 65.8% at a negligible performance penalty between 0.34% and 2.18%.

Zusammenfassung

Prädiktoren werden in vielen Bereichen der Computerarchitektur verwendet, um Performance zu steigern. Ihre genauen Abschätzungen über das Verhalten eines Systems können dazu verwendet werden, um Strategien zu entwickeln, die die Systemperformance steigern oder die Leistungsaufnahme senken. Diese Strategien werden effektiver, falls die Prädiktoren in Hardware implementiert sind und quantifizierte Vorhersagen liefern und nicht nur binäre. Eines der wichtigsten Ziele in jedem Computersystem, von Servern bis hin zu tragbaren batteriebetriebenen Geräten, ist die Reduzierung der Energie bzw. Leistung. Um dies zu erreichen, muss der Energieverbrauch aller Systemkomponenten reduziert werden. Dies ist besonders wichtig für off-chip DRAM, das auch im Leerlauf (*idle*) viel Energie konsumiert. DRAMs unterstützen daher verschiedene Leistungssparmodi wie *Self-Refresh* und *Power-Down*. Werden diese Leistungssparmodi jedes Mal verwendet, wenn der DRAM *idle* ist, sinkt die Performance aber aufgrund ihrer Aufwachzeit. Der *Self-Refresh* Modus bietet große Leistungseinsparungen, hat jedoch eine sehr hohe Aufwachzeit. Auf der anderen Seite ist die Aufwachzeit des *Power-Down* Modus geringer, spart jedoch auch weniger Leistung ein. Der effizienteste Modus hängt daher von der Länge der *Idle*-Periode ab, welche normalerweise unbekannt ist.

Diese Doktorarbeit präsentiert und evaluiert einen historien-basierten Prädiktor für quantifizierte Vorhersagen. Dafür werden eine Softwareversion und eine Hardwareimplementierung eines Vorhersagealgorithmus vorgestellt und analysiert. Ferner wird eine komplette Untersuchung des Ergebnisraumes des Prädiktors durchgeführt, um die Parametersets für eine Genauigkeit von über 96% zu bestimmen. Darüber hinaus wird ein generisches und komplett synthetisierbares Design des Prädiktors in VHDL für FPGAs präsentiert. Eine Analyse der Skalierbarkeit des Hardwareentwurfs zeigt, dass das vorgestellte Design eine geringe Geräteauslastung des FPGAs aufweist und mit über 210 MHz getaktet werden kann.

Unter Verwendung aller vorangegangenen Analysen wird eine vorhersagen-basierte Leistungssparstrategie vorgestellt, die die Leistungsaufnahme eines Speichers reduziert. Diese Leistungssparpolitik verbindet die Vorteile der beiden Leistungssparmodi, *Self-Refresh* und *Power-Down*, um den Verbrauch signifikant zu reduzieren und die Performanceminderung aufgrund von Aufwachzeiten gering zu halten. Zur Realisierung wird der historienbasierte Prädiktor verwendet, um die Länge der Leerlaufzyklen vorherzusagen. Auf Basis dieser Ergebnisse wird der *Self-Refresh* Modus, der *Power-Down* Modus oder eine Kombination von beiden Modi verwendet. Die vorgestellte Leistungssparstrategie wird mit Hilfe von verschiedenen Applikationen aus dem multimedialen Bereich evaluiert. Die Experimente zeigen, dass der Energieverbrauch des DRAMs zwischen 43.4% und 65.8% reduziert werden kann, bei einer vernachlässigbaren Laufzeitverlängerung zwischen 0.34% und 2.18%.

Published Papers

Parts of the thesis are already published as follows:

- Gervin Thomas, Ben Juurlink, and Dietmar 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

- Gervin Thomas, Karthik Chandrasekar, Benny Akesson, Ben Juurlink, and Kees Goossens. A Predictor-Based Power-Saving Policy for DRAM Memories. In *15th Euromicro Conference on Digital System Design (DSD)*, 2012. doi: 10.1109/DSD.2012.11

- Gervin Thomas, Ahmed Elhossini, and Ben Juurlink. A Generic Implementation of a Quantified Predictor for FPGAs. In *GLSVLSI '14: Proceedings of the 24th ACM International Conference on Great Lakes Symposium on VLSI*, Houston, Texas, USA, May 2014. ACM

Contents

List of Tables

List of Figures

1. Introduction

Right now, we are living in a time where the complexity of computer systems and embedded systems is rapidly increasing, due to more computational components being integrated onto smaller and smaller chips. This high complexity allows us to develop embedded systems which have high computational power in small hand-held devices such as tablets and smartphones. However, with a higher degree of complexity, problems arise. These issues include large amounts of data transfers between components inside embedded systems, or the high power consumption, which actually drains the battery of these portable devices. The effects of these problems can be reduced if a good estimate of the prospective behaviors of such components is available. This is the point where predictors come into focus. However, all types of predictors require reoccurring pattern for an accurate prediction. These reoccurring patterns arise from, for example, loops or function calls that have been performed repeatedly. These structures execute the same code several times and, therefore, have a predictive behavior. For example, an array is accessed within a loop. In each loop cycle one array element is loaded from the memory to a register and produces a reoccurring access to the memory. Additionally, in each loop cycle, a branch is performed to execute the loop again and produce a recurring branch. For this type of reoccurrence, branch predictors are used to increase the performance in pipelined microprocessors. The branch predictors predict if a branch is taken or not. A binary decision is perfectly suitable for this case. However, if the problem becomes more complex and a binary decision is not sufficient, more complex predictors are required that can quantify forecasts. Quantified predictors can be used to develop strategies or policies in the field of computer architecture that will improve system's performance. The advantage of quantified forecasts over binary forecasts is that the system is able to react in a broader range of ways and, therefore, they allow for a fine-grained strategy or policy.

The following section introduces two motivational examples for quantified predictors that makes use of the reoccurring pattern. Afterwards, the contributions of this thesis are stated and the general structure is supplied.

1.1. Motivation

The first motivational example shows how a quantified predictor can be used to reduce the memory power consumption by predicting the length of memory idle periods. The second examples shows how the prediction of network traffic can be used to identify possible congestion inside a network.

1.1.1. Memory

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. These requirements do not go hand in hand. On the other hand, high-end server systems also benefit from an energy reduction as it leads to a decrease in the operating costs and cooling effort.

Dynamic random-access memories (DRAMs) contribute significantly to overall system energy consumption. For example, memory energy consumption in mobile devices is up to 20% of the total power [60]. This goes up to 25% in data center servers [43]. The DRAM energy consumption profile in [9] and [10] 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 *self-refresh*.

To make this discussion more concrete, let us consider a 1 GB DDR3-800 MICRON memory [40]. This memory chip draws around 50 mA of current when idle, which corresponds to about 75 mW of power. DDR3 memories support two important power-saving modes, namely the *power-down* mode and *self-refresh* mode. The *power-down* mode reduces the power consumption to 18 mW, while the *self-refresh* mode brings it down to 9 mW [40]. Hence it is possible to reduce the power consumption during idle periods by 76% and 88%, respectively. Figure 1.1 presents an overview of both power-saving modes and presents the possible savings.

Both modes, however, incur a performance penalty, due to their power-up latencies. For the 1 GB MICRON DDR3-800 memory, the *power-down* mode has a relatively small penalty of around 25 ns (10 memory clock cycles), while the *self-refresh* mode has a very

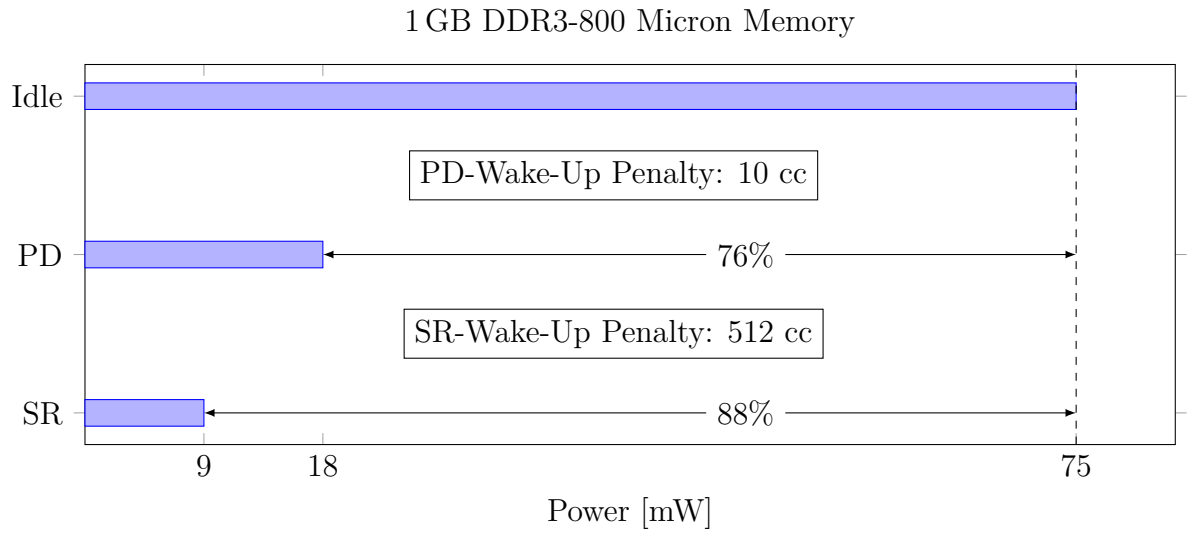


Figure 1.1.: Power Saving Modes

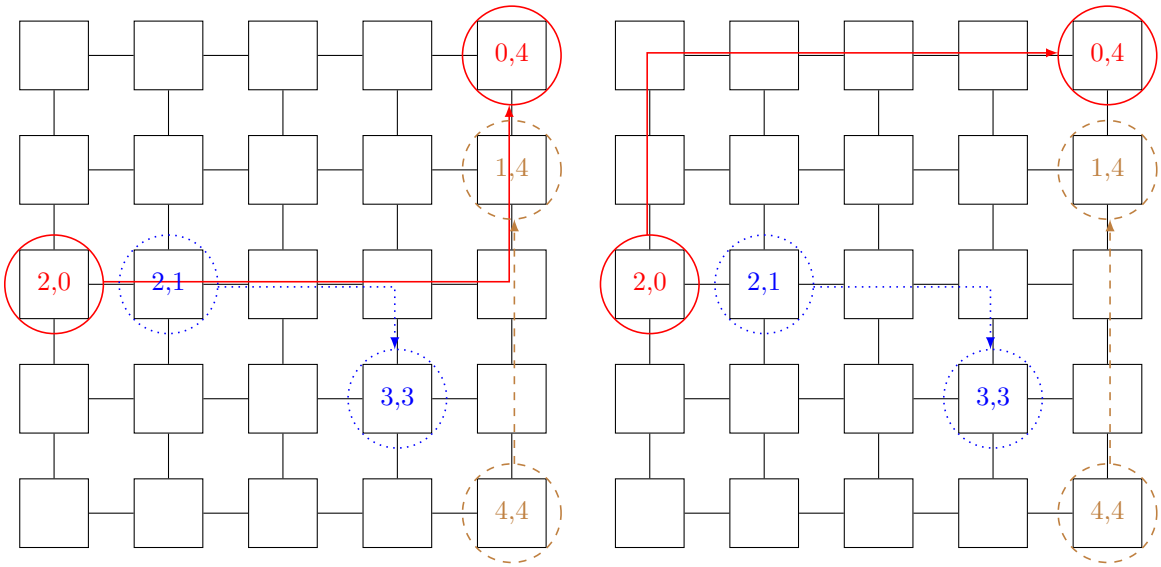
large penalty of about 1280 ns (512 memory clock cycles) [26]. The wake-up penalty cycles are also shown in Figure 1.1. The larger the power saving, the larger the power-up latency and, consequently, the performance penalty. This means that a trade-off needs to be made.

Applications use constructs like loops or functions, which produce reoccurring instructions. These reoccurring instructions produces the same request pattern when accessing a memory and, therefore, have a predictive behavior. Predicting the length of the idle periods can help to avoid, or at least reduce, the penalty cycles but still use the *self-refresh* mode as often as possible to maximize the power reduction.

1.1.2. Network Traffic

Networks are used to transport information from one node to another. However, if several messages are sent at the same time via the same link, congestions which delay the communication occur. Assume, for example, a 5×5 mesh network topology, as depicted in Figure 1.2. In both figures (a, b) it is illustrated that node (2,0) communicates with node (0,4). This is indicated by the solid lines. With dimension-order (xy) routing the communication is accomplished by first routing the message horizontally followed by routing it vertically. Additional messages that are sent simultaneously and that need to cross the same links as the first message cannot reach their destination and, as a result, congestion occurs. Such an example is shown in Figure 1.2a, where an additional message transfer should be established between nodes (2,1) and (3,3), indicated by the dotted line, as well

as the nodes (4,4) and (1,4), indicated by the dashed line. This communication between nodes (4,4) and (1,4) cannot take place until the first communication between nodes (2,1) and (3,3) releases the switching elements.



(a) Congested communication (b) Uncongested communication

Figure 1.2.: Reducing congestion by rerouting communications

With traffic prediction it could be known a priori that the above mentioned nodes want to communicate. With this knowledge a different routing decision could be taken. Alternative routing paths are shown in Figure 1.2b. The first communication between nodes (2,0) and (0,4) is indicated by the solid line. It could be realized by dimension-order (yx) routing which first routes the message vertically and then horizontally. With this new routing decision, the other nodes (2,1) and (3,3), as indicated by the dotted line, as well as (4,4) and (1,4), as indicated by the dashed line, can communicate in parallel. Using a predictor to forecast the amount of data sent over the same link allows to develop a routing policy to realize blocking free communications.

1.2. Contribution

The previous section has presented two examples of systems that benefit from the usage of a quantified predictor. This thesis provides several contributions for how to cope with these problems. The main contributions of this thesis can be summarized as follows:

1. A prediction algorithm is presented to forecast quantified data points. This algorithm is applied to memory idle periods and network traffic based on *message passing interface* (MPI). Two software implementations, Octave and SystemC, are provided. The Octave implementation is used to validate the prediction algorithm and to determine the accuracy. The SystemC model is integrated into a *multiprocessor System-on-Chip* (MPSoC) to forecast memory idleness.

2. A power-saving policy for reducing DRAM power consumption is presented. The predictor is used to forecast the length of memory idle periods to apply a power-saving mode, like *self-refresh* or *power-down*. The predicted length is used to power-up the memory before the next request arrives and avoids or reduces the wake-up penalties. The energy as well as the execution time of different multimedia applications are presented and analyzed to show the effectiveness of the power-saving policy.

3. A *register-transfer level* (RTL) implementation of the predictor in *very high speed integrated circuit hardware description language* (VHDL) is presented to measure and validate power-saving policy, including the predictor itself. A generic and fully synthesizeable implementation is presented and analyzed in detail. Moreover, the prediction algorithm working on floating point numbers is mapped to unsigned integers, to reduce complexity. A complete complexity analysis is presented to determine the device utilization. Furthermore, the maximum frequency for different configurations of the predictor is determined. Additionally, the power consumption of the predictor is investigated and the proposed power-saving policy is validated, which includes the predictor's power consumption.

4. The predictor is applied to MPI network communication to show the predictor's applicability for work on other fields of application. Additionally, the predictor's ability to forecast several data points, which are based on a data point already predicted, is analyzed.

1.3. Organization

This thesis is organized as follows: Section 2 presents an overview of related work, including predictors from several fields, for example, branch predictor, and predictors applied to DRAMs or networks. Afterwards, in Section 3 the proposed predictor is explained in detail by providing the mathematical background. Moreover, a detailed explanation of the required knowledge about DRAMs and their build-in power-saving modes will be provided. Additionally, the MPSoC platform CompSOC [51] is described, as it is used for later analysis. Section 4 provides the software implementation of the proposed predictor. The section will also include a power-saving policy, which uses the predictor to forecast memory idle periods. Additionally, the RTL implementation of the predictor is presented by providing the data path and the control unit separately. Section 5 presents the experimental results by providing an accuracy analysis of the predictor that validates the proposed power-saving policy. Furthermore, this section provides a full complexity analysis of the VHDL predictor, which will include the device utilization, maximum frequency, and the power consumption. Additionally, the predictor is applied to network traffic to validate the predictor's applicability for work on other fields of application. Moreover, the predictor's ability to forecast several data points is tested. Finally, Section 6 concludes this thesis.

2. Related Work

This Chapter presents related work to this thesis. All contributions of this thesis are related to the software and hardware implementation of a history-based predictor. The history-based predictor was used to develop a power-saving policy to reduce the power consumption of DRAMs. For complexity and power analysis, the predictor was implemented on an *field programmable gate array* (FPGA). Additionally, the predictor is used to forecast network traffic so as to show the applicability to other fields of applications. Hence, this chapter generally focuses on predictors; predictors applied to DRAMs or networks, and the implementation of predictors on FPGAs.

2.1. Branch Predictors

Predictions are widely used in computer architecture. The most well known predictors are the branch predictors. In [53], the authors study on branch prediction for maximization of prediction accuracy. They analyzed the accuracy of different static and dynamic branch prediction methods. The best choice was identified as one of the dynamic methods which hashes the branch address so as to access a branch history. The authors in [67] then proposed a dynamic branch predictor, the Two-Level Adaptive Branch Prediction, for achieving substantially higher accuracy. The method used a table-driven approach for branch prediction. Although, the research field of branch predictors is vast, these kind of branch predictors output binary decisions, whether the branch is taken or not. Despite that, using a *branch target buffer* (BTB) offers the possibility to predict the address of a taken or unconditional branch as shown in [33]. The BTB caches the most recently resolved target and uses this data for prospective branches. However, BTB can predict the branch targets only if they have a hit in their buffer. This means that the exact same branch was executed previously. This technique is perfectly suitable for the application on taken branches, but it can never be applied to identify similar branch addresses that are not in the buffer.

2.2. Predictors Applied to DRAMs

Predictors have been used to reduce memory access times for DRAMs connected to MPSoCs. In [54, 66], a predictor was employed to reduce precharges and activations, 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 so as to minimize the average access latency. In [54] a hybrid predictor is used, which closes a row after a timer runs out as well as keeps track of row sequences for deciding which row to open next if a sequence reoccurs. The predictor used by [66] is a two-level access based predictor, similar to a branch predictor [67]. Similarly in [3], a one-level predictor scheme with lower implementation cost is proposed. This predictor is used to track the number of accesses for a given DRAM page to predict DRAM locality so as to make page closing decisions.

The work in [47] proposed a dynamic memory mode control scheme with a predictor to forecast whether the next memory reference causes a page hit or not. This is used to exploit locality and reduce the number of activations and precharges to a bank. The predictor uses a 2 bit saturation counter as also used for branch prediction. However, all the solutions that used predictors targeted active power reduction, not idle power reduction. In [14], the authors proposed reducing idle memory power by using a compiler-directed selective *power-down* as well as a hardware-assisted prediction-based run-time *power-down*. Having said that, 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. Their predictor simply reuses the last value as prospective value.

The hesitation towards 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. The predictor used for these policies are either binary predictors, similar to branch predictors, or quantified predictors. The quantified predictors are simple. They keep track of the last used value or are table-driven to store long sequences in case they reoccur.

2.3. Power Reduction of DRAMs without Predictors

Other papers have proposed DRAM power reduction solutions without using explicit prediction logic. For instance in [36], a DRAM precharge policy based on address analysis is presented. Statistical information from instructions waiting to access the memory are

collected and then analyzed to determine the next bank access and decide on which bank to precharge, thus reducing the average memory latency. In [34], the authors proposed a hardware prefetching technique, that was assisted by static (design time) analysis of data access patterns, for efficient data prefetching. However, this idea would only be useful in improving the performance of caches and can lead to increased main memory power consumption, due to the mispredicted prefetches. The authors in [58] proposed combining read/write multiple times within a single activate-precharge pair so as to obtain significant energy savings and [29] achieve low power consumption in DRAM memories through changing the processor and memory clock frequencies. These solutions also target active power reduction.

Again, using the *self-refresh* mode comes with the risk of high power-up latencies and is also avoided by all the mentioned papers.

2.4. Predictors Implemented on FPGAs

The authors in [7] presented a neural based branch predictor as well as how it is implemented on FPGAs. However, the authors identified that the FPGA implementation of this predictor is not suitable for a long branch history table. Furthermore, they did not analyze the power consumption of the predictor running on an FPGA. In [38], the authors presented an implementation of intelligent predictors for solar irradiation running on FPGA. This predictor make uses of an *artificial neural network* (ANN) to forecast climatic conditions. Huge databases are needed to train the *multilayer perceptron* (MLP) for accurate prediction. The authors claim that their work investigated the possibility of implementing MLP and, therefore, presented only the complexity of their predictor. They did not analyze the maximum frequency or the power consumption. The authors in [37] also presented an embedded hardware architecture to forecast climatic conditions. Their design consists of a microcontroller for data management and controlling as well as an FPGA for data processing. Their predictor performs a data base search and uses the Parzen method [48] to forecast climatic conditions. Therefore, a multidimensional integral of a logarithmic function has to be solved, which requires high computational power. The results of this work focuses on the accuracy of the weather forecast and does not provide any complexity analyses, power measurements, or frequency analyses of their design running on the used FPGA.

Model Predictive Control (MPC) is a method for complex process control mostly used in oil refineries and chemical industry. These controllers have the predictive ability to

calculate the prospective behavior of a process in dependence to the input signals so as to optimize the output signals. These predictive controllers require high computational power because several matrix operations have to be performed. Hence, the authors in [5, 31] presented a hardware/software codesign for their controllers. The arithmetic operations are performed on an FPGA to speed up the calculations. The control is handle by a program running on a microprocessor. Both authors focus on the implementation of their design and how to parallelize or optimize the calculations running on the FPGA. The authors in [5] did not provide any analysis of the design's complexity, maximum frequency, or power consumption. Moreover, [31] did not present any power analysis, but listed the complexity and the maximum frequency. Nevertheless, the authors stated that more analysis has to be done, as the design runs at a low frequency. The predictors needed to speed up their calculation have to solve a system of equations, which are likely highly complex and, therefore not applicable to DRAMs.

The authors in [15] presented a digital fuzzy logic controller running on an FPGA. This controller used a similar technique to the one presented in this thesis, to forecast data points. Again, the authors provided a complexity analysis that included the maximum frequency but did not investigate the power consumption of their predictor.

Most of the authors either used hardware predictors implemented on FPGAs to speed up their calculations, or presented a feasibility analysis. For their work, it was not necessary to provide a full complexity analysis or to determine the power consumption. This is either because only the speed up compared to the software was an important factor, or to map their problem to FPGAs.

2.5. Predictors Applied to Networks

Another subfield of predictors is to forecast traffic patterns in networks. In [24], the authors proposed a table-driven predictor for predicting the communication patterns in *Network on Chips* (NoCs). They predicted end-to-end traffic without taking the intermediate switches into account. Their method, however, was only able to predict a single future time interval. The predicted amount of communication was either zero or the current quantity. This technique was evaluated by running a modified block LU decomposition kernel on Tilera's TILE64 platform.

A flow control algorithm for the prediction of switch-to-switch traffic is proposed in [44]. This prediction is decentralized and based on the information the routers received directly from their neighbors. The number of packets injected in the network is controlled from

the predictor. In [59] the authors presented a congestion control strategy based on a MPC which controlled the offered load.

In [12] a fuzzy based predictive traffic model was used, so as to avoid congestion at high utilization while maintaining high quality of service in *Asynchronous Transfer Mode* (ATM) networks. Similarly, the authors in [46] used a fuzzy based traffic predictor and applied it to ATM traffic management.

The authors in [30] used a coherence communication prediction in distributed shared-memory systems to detect data that was needed by several processors and to deliver the data as soon as possible. Their approach was also table-driven.

However, all of these mentioned predictors did not present a hardware implementation, or any kind of power analysis for their predictors.

2.6. Other types of Predictors

This section will present predictors that are applied successfully to several fields of application, but are loosely connected to this thesis.

The usage of *dynamic voltage and frequency scaling* (DVFS) have been implemented to increase energy efficiency in modern processors. DVFS allows to switch between operating points (voltage and frequency) at runtime and can, as a result, reduce energy. The authors in [41] presented a DVFS performance prediction for realistic memory systems to choose the optimal operating point. However, the authors applied the DVFS to modern processors to increase power and energy efficiency. The realistic memory system model is used to increase the accuracy for this technique and is not applied to the memory system itself.

Value prediction is a technique used to increase *instruction level parallelism* (ILP) by breaking data dependencies [8]. Depended instructions have to process sequentially, because the output from the first one is needed as input for the following one. Value prediction produces values, which are used speculatively as input for the dependent instructions so as to increase ILP. In [18, 52, 61] different techniques were introduced for value prediction. These predictors used simple forecast techniques. These techniques are, for example, to keep track of the last value of an instruction and the difference to the previous one [18], storing the last 4 values of an instruction in a table [61], or combining both [52]. However, these predictors were adjusted and optimized for this use-case and, therefore, not applicable for other use-cases.

A fuzzy based predictor was also applied successfully to load forecasting for electric

utilities in [45]. The authors presented a software implementation of their predictor using fuzzy logic techniques. However it must be noted that the authors focus only on the accuracy of the prediction algorithm and did not analyze the predictor's complexity or run-time.

2.7. Summary

This chapter provided information about all the types of predictors and polices used to reduce DRAM idle power consumption. However, all types of policies employing predictors to reduce DRAM power consumption did not uses the *self-refresh* mode due to the high wake-up penalties. Furthermore, the DRAM power-saving policies that are without predictor support did not make use of the *self-refresh* mode. Predictors applied to network traffic were shown but never provided any analysis of the complexity or power consumption of the predictor itself, as no hardware implementation was presented. Predictors implemented on FPGAs were always used to either speed up a calculation process, or as a feasibility analysis. However, these predictors were not analyzed on power consumption. In summary, all mentioned related works were either not using the *self-refresh* mode to reduce DRAM power consumption, lacking a hardware implementation of the predictor, or lacking a complete scalability and power analysis. In contrast, this thesis will address all of these points.

3. Background

This chapter provides the background of the proposed predictor, the DRAMs, as well as the platform that validates the proposed power-saving policy. First, the used history-based predictor is introduced, extended and applied in this work. The mathematical background, as well as the underlying algorithm, of the predictor, are explained in detail. Therefore all of needed equations are presented and explained. The algorithm is introduced step by step. These steps are used in later chapters to illustrate the predictor's software and hardware implementation.

This thesis presents a power-saving policy, which employs the introduced predictor. The main goal of this policy is to reduce memory power consumption. Hence, the basics of DRAM will be explained. This includes the general structure of DRAM, its operations, as well as the build in power-saving modes, which are used to implement the power-saving policy. The power-saving modes will be explained in detail and all the necessary specifications are presented. The power-saving modes are compared to one another, so as to determine in which situations each power-saving mode is most efficient for.

Finally, a brief overview of the MPSoC-platform called CompSOC is given. This platform is used later to analyze the presented power-saving policy. Finally, this chapter will conclude with a short summary in Section 3.4.

3.1. Predictor Theory

This section describes the theory of the used prediction algorithm. The history-based predictor is based on [45]. To help better understand the later introduced software and hardware implementations, the theoretical background of the predictor is introduced in this section. The general structure of the predictor is depicted in Figure 3.1.

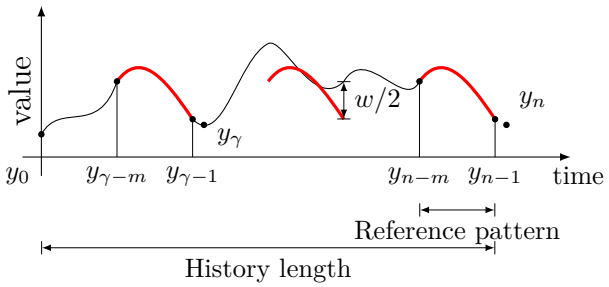


Figure 3.1.: Working of the Predictor

The predictor builds up a history of data points (y_0 to y_{n-1}) before forecasting the next future data point (y_n). Afterward, the predictor probes the history of data points, considering a current set of reference data points between (y_{n-1}) and (y_{n-m}), and it then searches for similar patterns in the history. If a pattern that is similar to the *reference pattern* was found, such as the pattern between ($y_{\gamma-m}$) and ($y_{\gamma-1}$), the algorithm then weighs the next data point (y_γ). This depends on the similarity given by a parameter *width* (w). Matching to past data points is not limited to just one occurring pattern set in the history, in contrast to Figure 3.1. Once the predictor has probed the whole history, the next future data point (y_n) is calculated by considering all weighted data points from the history.

The prediction is done in 5 steps. The steps are explained in detail so as to help understand the implementation of the predictor introduced in the succeeding chapters:

(1) **Build history**: Before the predictor is able to forecast data points, a history of n data points is required. The algorithm defines a parameter *history length* $= n$, which gives the limit on the number of past data points to be taken into account for the prediction. The considered history is given as vector Y.

$$Y = (y_0, y_1, \cdots, y_{n-1}) \tag{3.1}$$

In Figure 3.1, the history includes all shown data points, but any size of the *history length* can be used for the prediction.

(2) **Calculate absolute differences**: Next, the algorithm considers the latest m data points between (y_{n-m}) and (y_{n-1}) as a *reference pattern* that includes *pattern length* $= m$ data points from the history set of n data points (where $m < n$). These *reference patterns* are subtracted iteratively from the history data points.

$$\begin{aligned} D_i =& (d_{i,0}, \cdots, d_{i,m-1}) \\ =& Y\left[n-m-1-i,n-2-i\right] - Y\left[n-m,n-1\right] \\ & i \in [0, n-m-1] \end{aligned} \tag{3.2}$$

This operation calculates the absolute differences between the reference pattern and all possible past data points within the considered history. D_i depicts a difference vector and the $Y[a,b]$ is a subvector from Y that includes all data points between indices a and b. Each iteration produces a set of absolute differences between the *reference pattern* $Y\left[n-m,n-1\right]$ and a set of past data points $Y\left[n-m-1-i,n-2-i\right]$. In total $n-m$ difference vectors are calculated.

(3) **Determine weight/similarity**: A parameter *width* (w), set by the user, is used to identify whether a difference vector D_i fits the reference pattern. Therefore, all elements $d_{i,k}$ of the difference vector D_i are mapped using a triangular function $\mu : X \rightarrow [0,1]$ that is given by Equation (3.3) to a value between 0 and 1. This function gives a degree of similarity, where 1 is a perfect match and 0 depicts no similarity at all.

$$\mu(x) = \begin{cases} 1 - \left|\frac{2 \cdot x}{w}\right|, & \text{if } |x| < |w| \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

If a particular data point differs by more than $|w/2|$, then the weight for this data point is set to zero, otherwise the data point gets a weight depending on the similarity. Equation (3.3) is also illustrated in Figure 3.2.

To determine the weight for each set of absolute differences D_i, all single weights within this difference vector are multiplied with each other.

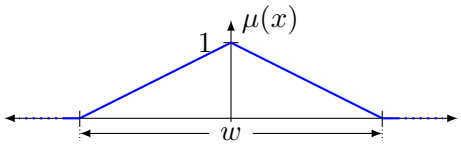


Figure 3.2.: Triangular function to determine similarity

$$\beta_i = \prod_{k=0}^{m-1} \mu\left(d_{i,k}\right) \tag{3.4}$$

$d_{i,k}$ is the k-th element of the difference vector D_i. Each set is reduced to one value that gives the weight for the data point following the considered set. This means, if one value of the set differs more than $|w/2|$ from the reference pattern, the whole weight for the set becomes zero. For example, in Figure 3.1 all data points between $(y_{\gamma-m})$ and $(y_{\gamma-1})$ are subtracted from the reference pattern, weighted by the triangular function and multiplied among each other to calculate the weight for the data point (y_γ).

(4) **Weight past data points**: In the following, two values are determined. The first value N (numerator) is calculated by multiplying each weight β_i with the corresponding data point $y_{n-\gamma-1}$, respectively. After, all products are added to determine the weighted sum of all similar data points. The second value D (denominator) is the sum of all of the weights. Both operations are represented by the Equations (3.5) and (3.6).

$$N = \sum_{\gamma=0}^{n-m-1} \beta_\gamma \cdot y_{n-\gamma-1} \tag{3.5}$$

$$D = \sum_{\gamma=0}^{n-m-1} \beta_\gamma \tag{3.6}$$

The most important effect of both operations is that data points weighted with zero are no longer considered because β_i is zero and, therefore, does not influence N or D. Both operations are required preparations for calculating the weighted mean, which is calculated in the final step (5).

(5) **Forecast data point**: For the final step, the numerator N is divided by the denominator D and gives the next future data point. This calculations is given by the

following equation:

$$y_n = \frac{N}{D} = \frac{\sum_{\gamma=0}^{n-m-1} \beta_\gamma \cdot y_{n-\gamma-1}}{\sum_{\gamma=0}^{n-m-1} \beta_\gamma} \tag{3.7}$$

This operation matches the calculation of the weighted mean.

All steps together calculate the next future data point y_n.

This section presented the mathematical background of the proposed predictor. The predictor needs a total five steps to forecast the next prospective data point. Each step was explained in detail and all equations were presented.

3.2. DRAM

This section provides basic information about DRAM and summarizes the power-saving modes used in this thesis.

3.2.1. DRAM Basics and Power-Saving Modes

DRAM is the most commonly used type of main memory in mobile phones, laptops, gaming consoles and servers. Each bit of data is stored as charge in a capacitor. The capacitors leak the charge over time. Thus, the memory has to be refreshed regularly to avoid losing data. DRAM is volatile memory and data is lost when the memory is turned off. The DRAMs consist of several banks, where bits are stored in rows and columns. A simplified version of a DRAM is depicted in Figure 3.3.

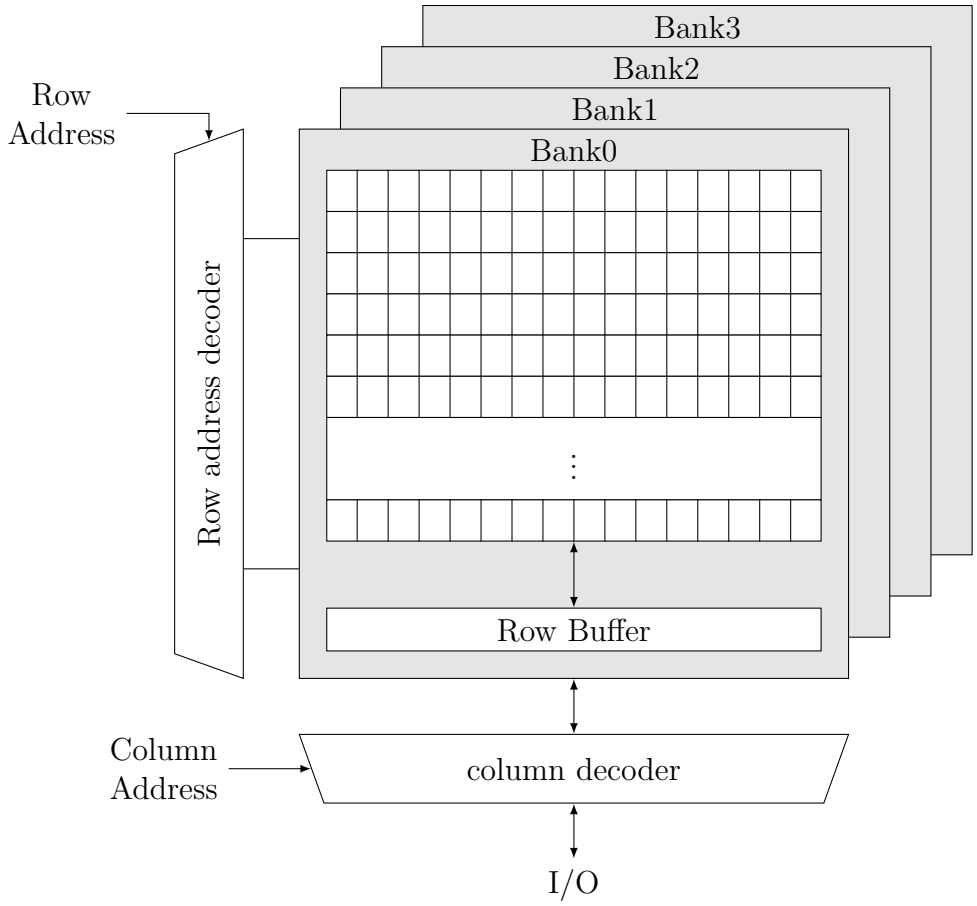


Figure 3.3.: DRAM structure

Reading and writing the DRAM is handled in two steps. First, the data from the addressed row is moved to the row buffer. Second, a column decoder allows the selection

of single data block within the row, before moving it to the I/O buffer to complete the data transfer.

DRAM provides two power-saving modes for the reduction of idle power consumption. These two modes are used to develop a predictor based power-saving policy, which is a contribution of this thesis. The two power-saving modes are called *self-refresh* and *precharge power-down*. Both modes have different characteristics and are, therefore, advantages and disadvantages. Figure 3.4 depicts a simplified *finite-state machine* (FSM) of a DRAM and shows how operations are handled. The full FSM in given in [40].

The states `Init` and `Active Read/Write` are both sub-FSM and possess many sub-states, respectively. For both states, a higher abstraction level was chosen, because more details are not needed to explain the used power-saving modes. The `Init` state consist of states that handle the power on, the reset, the initialization, as well as the calibration behavior of the DRAM. After the `Init` state the memory goes to the `Idle` state and is ready to be used for data transfers. The rows of the memory have to be refreshed at regular intervals so as to contain the data due to the leaking of the capacitors, which is done in the `Refreshing` state.

The power-saving mode *self-refresh* can be triggered from the `Idle` state. However, there are two different *power-down* modes, namely, *precharge power-down* and *active power-down*, reachable from different states that exist within the FSM. Both modes depend on the previous operation. Before a data transfer can be executed, the accessed row has to be moved to the *row buffer*. After the data transfer, there are two possible courses of action: (1) In case data from the same row is needed again, the data can be kept in the row buffer and can later be accessed. If data is retained in the row buffer after the read or write operation is completed, it will keep the memory in the active state and keep the row open. (2) The data inside the *row buffer* is moved back to the memory row and, as a result, closes the row. In this case the memory is moved to the `idle` state after a precharge is done within the `Active Read/Write` state, and therefore, not shown in the FSM. The memory can be idle in either of these two cases and the *precharge power-down* or *active power-down* can be scheduled, as depicted in Figure 3.4. In general, the precharged state *power-down* saves more power than the active state *power-down*. However, in this thesis, only the *precharge power-down* mode, hereinafter referred to as the *power-down* mode, and the *self-refresh* mode are used. For the following analysis, the MICRON 1 GB DDR3-800 is used and therefore more information about the FSM, the power-saving modes, or the memory itself can be obtained from the data sheet [40].

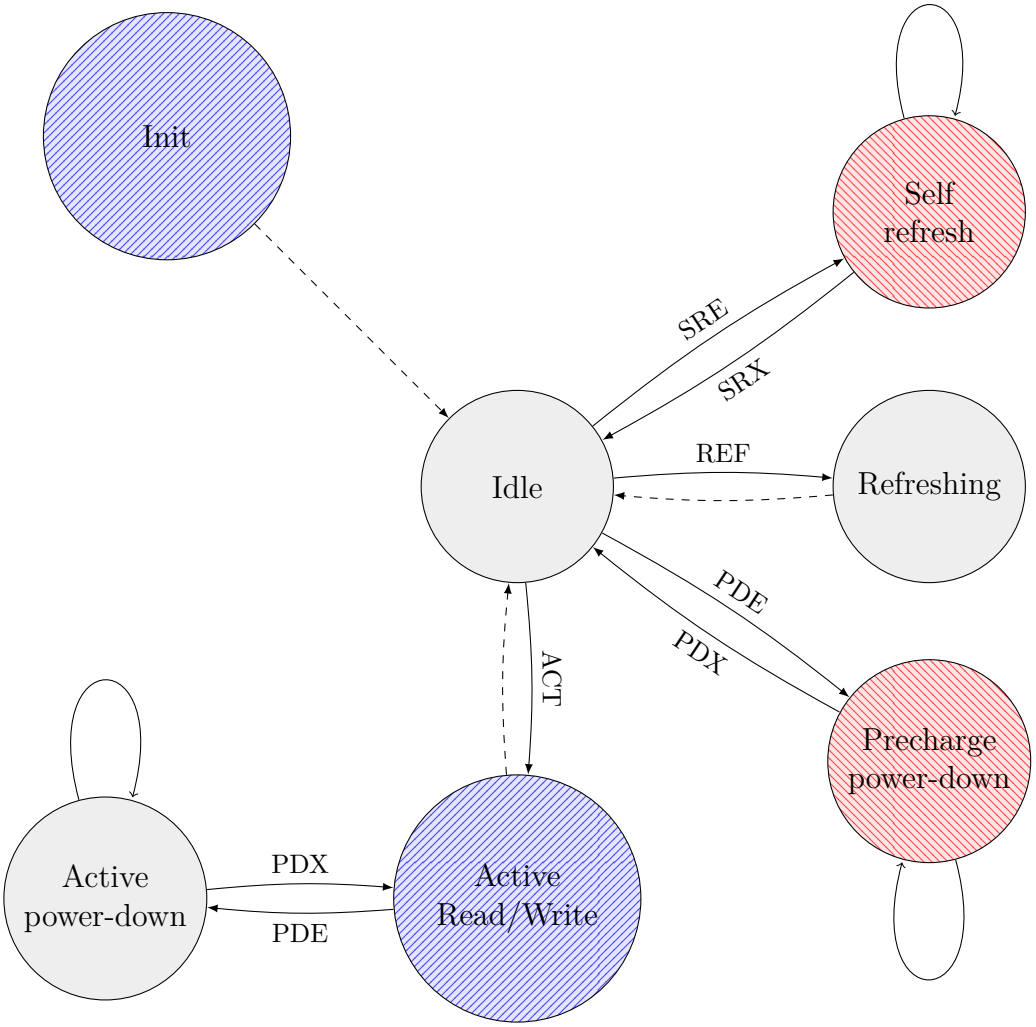


Figure 3.4.: DRAM: simplified FSM

3.2.2. Comparison of self-refresh and power-down

When the memory is on, it is not always in use. This depends on the application as there can be several memory idle periods of varying lengths. The memory still consumes a significant amount of energy during these idle periods [9, 10], which could be reduced using power-saving modes, such as *power-down* or *self-refresh*.

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. For example, the MICRON 1 GB DDR3-800 memory device [40] is used and Table 3.1 summarize the most important parameters for the different power-modes.

Mode	I_{DD}			Power Up Time		
	Abbr.	Value	Unit	Abbr.	Value	Unit
Idle	I_{DD2N}	50	mA	-	-	-
Power up	I_{DD2N}	50	mA	-	-	-
Self-refresh	I_{DD6}	6	mA	X_{SDLL}	512	cc
Power-down	I_{DD2P0}	12	mA	X_{PDLL}	10	cc

Table 3.1.: Micron 1 GB DDR3-800: Specification for different power-modes

For this memory device, the current consumed during the power-up cycles and in the precharged idle mode (denoted by I_{DD2N}) is 50 mA. When in the *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 the 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).

The *self-refresh* mode has the highest power saving ability, so using this mode as often as possible maximizes the power savings. However, to avoid the performance penalty, this thesis presents a power-saving policy that uses the proposed predictor to forecast the length of the idle periods and power-up the memory before the next request arrives. This allows to use *self-refresh* as often as possible and avoid, or at least reduce, the penalty cycles.

3.2.3. Efficient Power-Saving Mode Selection

An overview of both power-saving modes, *self-refresh* and *power-down*, was given in the previous section. 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 1 GB Micron DDR3-800), the *power-down* mode is more efficient, because of the shorter power-up latency compared to the *self-refresh* mode. For longer idle periods, the *self-refresh* mode becomes increasingly useful because of the lower power consumption that compensates for the long power-up latency.

The minimum idle period length at which the *self-refresh* mode begins to save more power than the *power-down* mode (which includes powering-up time) can be calculated. Therefore, the energy consumption when the memory is in the *power-down* or the *self-refresh* mode, as well as the energy consumption during their corresponding power-up cycles, have to be considered. Hereafter, this thesis will refer to this minimum idle duration as the *self-refresh threshold* (SRT). The SRT is defined as the point where the *self-refresh* mode is more efficient than the *power-down* mode.

The Equations (3.8) and (3.9) are used to calculate the energy consumption for the self refresh mode (E_{SR}) and the *power-down* mode (E_{PD}), respectively. Both equations are needed to derive the SRT.

$$E_{SR} = \underbrace{[I_{DD6} \cdot (SRT - X_{SDLL})] \cdot V_{DD} \cdot t_{cycle}}_{\text{self-refresh mode}} + \underbrace{[I_{DD2N} \cdot X_{SDLL}] \cdot V_{DD} \cdot t_{cycle}}_{\text{power-up}} \tag{3.8}$$

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, also known as the power-up latency. During the *self-refresh* period, I_{DD6} current is drawn by the memory, and during the X_{SDLL} cycles, it draws a I_{DD2N} current, as shown in Equation (3.8).

Similarly, when the *power-down* mode is selected, the memory is in the *power-down* state for $SRT - X_{PDLL}$ clock cycles, and it consumes I_{DD2P0} current. During the power-up period of X_{PDLL}, the memory consumes I_{DD2N} current, as given by Equation (3.9). In both equations, V_{DD} corresponds to the supply voltage and t_{cycle} to the clock period of the memory clock.

$$E_{PD} = \underbrace{[I_{DD2P0} \cdot (SRT - X_{PDLL})] \cdot V_{DD} \cdot t_{cycle}}_{\text{power-down mode}} + \underbrace{[I_{DD2N} \cdot X_{PDLL}] \cdot V_{DD} \cdot t_{cycle}}_{\text{power-up}} \tag{3.9}$$

Equating both equations ($E_{SR} = E_{PD}$) and solving for SRT, as shown in Equation (3.10), gives the minimum idle period length (rounded up) when *self-refresh* mode is saving more energy than the *power-down* mode.

$$SRT = \frac{X_{SDLL} \cdot (I_{DD6} - I_{DD2P0}) - X_{PDLL} \cdot (I_{DD2N} - I_{DD2P0})}{I_{DD6} - I_{DD2P0}} \tag{3.10}$$

For the 1 GB Micron DDR3-800 memory discussed in Section 3.2.2 and the specifications from Table 3.1, SRT equates to 3691 cc.

Figure 3.5 presents the energy consumption when the 1 GB Micron DDR3-800 memory is idle, in *self-refresh* mode, or in *power-down* mode. The x-axis is given in clock cycles and the y-axis gives the consumed energy.

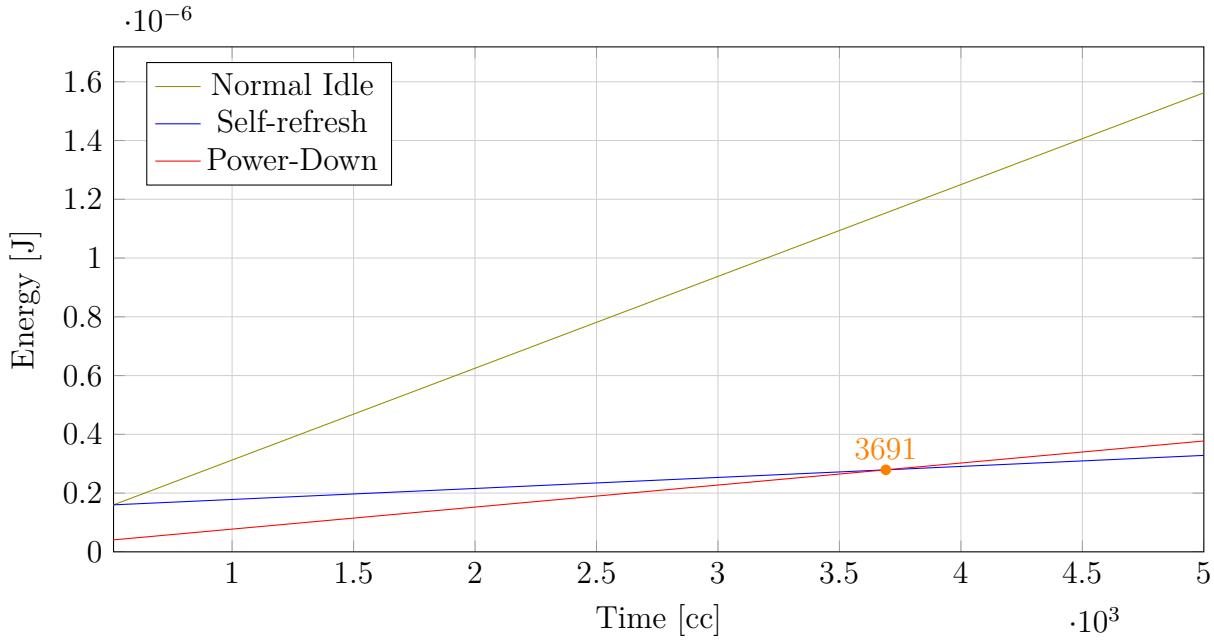


Figure 3.5.: Energy consumption for different power-modes (1 GB Micron DDR3-800)

The intersection between the energy of the *self-refresh* mode and *power-down* mode is the one calculated by Equation (3.10).

3.3. CompSOC

The proposed predictor-based power-saving policy is analyzed and validated using the CompSOC platform [51]. CompSOC was developed by The Eindhoven University of Technology and The Delft University of Technology as research platform [17]. The goal of the CompSOC platform is to reduce system complexity by using two techniques; *composability* and *predictability* [17]. In general, a platform is considered composable if a running application cannot change the behavior of another application in value and time domains [22]. The advantage of composable systems are that applications can be developed and verified independently. Moreover, these applications can be integrated in a larger system without any interference to other applications or the system itself. Predictability guarantees a useful lower bounds on performance given to an application [23].

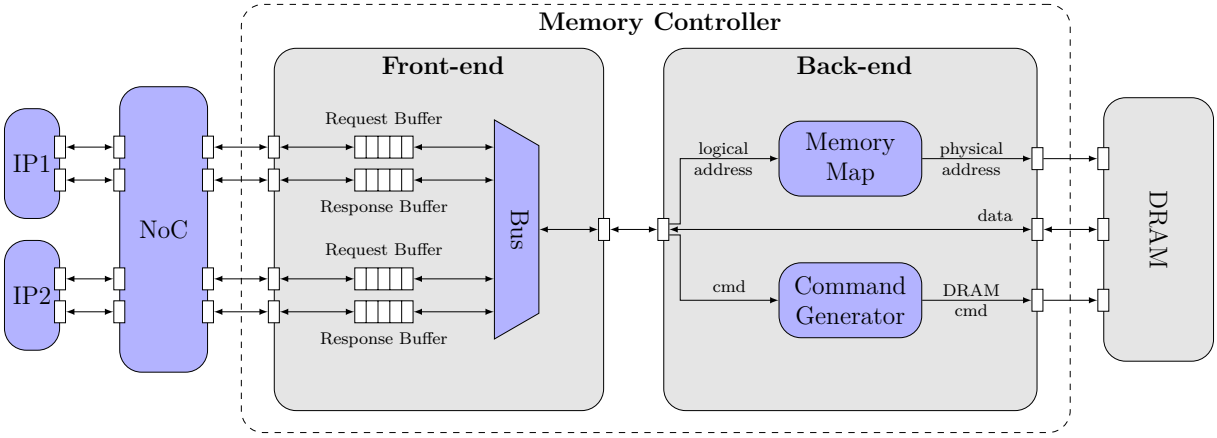


Figure 3.6.: Simplified overview of the CompSOC platform

A simplified overview of the CompSOC platform is depicted in Figure 3.6. The CompSOC platform consists of multiple IPs, which could be MicroBlazes and DMAs for example. All IPs are connected to a NoC, called Aethereal [21, 23]. The DRAM memory controller [2, 50] is highlighted in more details, as this is the place were the predictor will be integrated for the proposed power-saving policy. The controller is divided into the front-end and back-end. The front-end contains the buffering as well as arbitration, while the back-end interfaces the memory.

This section presented an overview of the CompSOC platform, as well as references that provide further information on the platform.

3.4. Summary

This section has presented all required basics underlying the thesis. First the used prediction algorithm was presented. The algorithm was introduced in 5 steps and explained in detail by presenting the mathematical background. Based on the predictor, this thesis will propose a power-saving policy for DRAMs, applying two power-saving modes provided by the memory device. Therefore, the basic structure and functionality of DRAMs were presented. Additionally, the two power-saving modes were introduced as was a discussion of their advantages and disadvantages. This thesis also incorporated an analysis so as to show the preferable power-saving mode for a specific idle period length. Finally, the CompSOC platform was presented as it will be later used for the experimental analysis.

4. Implementation

This Chapter introduces the software and hardware implementation of the predictor, an extensions of the predictor to be applicable for DRAMs, and a proposed power-saving policy.

This section will begin by introducing the software implementation of the predictor. Therefore, all equations presented in the background section are mapped to code for an executable program. The predictor needs an extension to be applicable on DRAMs idle cycles. Therefore, the concept of predicting *levels* is introduced as an extension to the prediction algorithm. Predicting *levels* allow a wide range of idle periods to be covered. Afterward, the proposed power-saving policy is presented using the extended predictor to reduce power consumption. This policy uses the predictor to forecast the length of the idle period. Depending on the length, the power-saving policy employs the *self-refresh* mode, the *power-down* mode, or a combination of both to maximize the power reduction. Moreover, the predicted length allows the memory to wake-up before the next request arrives so as to reduce the wake-up penalty. Next, the hardware implementation of the predictor in VHDL is presented for later analysis on power consumption. Therefore, the data path and control unit are introduced separately. The prediction algorithm is mapped to hardware components to implement the data path. The prediction algorithm, working on floating point numbers, is also mapped to unsigned integers to reduce hardware complexity. The predictor control unit is implemented as a FSM to carry out all calculations in the correct order. The predictor's runtime is calculated by determining the runtime for each component. Finally, this chapter is concluded with a summary of the entire implementation process.

4.1. Software Implementation

This section introduces the software implementation of the prediction algorithm presented in Section 3.1. The purpose of the software implementation is twofold: (1) The prediction algorithm can be tested and validated with manageable effort and (2) to integrate the predictor in the CompSOC platform [51].

The first software implementation was realized in GNU Octave [16, 27], which is a high-level interpreted language. GNU Octave is, for the most part, compatible to the proprietary MATLAB, but it is distributed under the terms of the GNU General Public License. The GNU Octave implementation was used to verify and analyze the influence of the different parameters on the accuracy. The second software implementation was implemented in SystemC [1] to integrate the predictor into the CompSOC platform so as to validate the proposed power-saving policy, which will be introduced in Section 4.3. The CompSOC platform is prototyped on Xilinx FPGA boards, and (partially) in SystemC. However, due to licensing restrictions, only the SystemC implementation of the CompSOC platform was available.

Both implementations, GNU Octave and SystemC, are similar and they differ only by the specifications of the programming languages. Therefore, the prediction algorithm is presented as a pseudo-code and it is depicted in Algorithm 1. The implementation is based on the algorithm theory from Section 3.1.

In this pseudo code, small letters like hl,pl,w, depict variables which contain a single number, while capital letters like Y,RP,D represent a vector and include several data points. The input parameter of the prediction algorithm are (line 1): the history of past data points (Y), the *history length* (hl), the number of data points used as *reference pattern*(pl) as well as the *width*(w) to define similarity. The number of past data points in Y can be different from the *history length* (hl). However, it is important to guarantee that the length of Y is greater or equal to hl before invoking the algorithm PREDICTOR. Next, in line 2 the variable *num* and *den* are initialized and the length of the vector Y is determined and stored in n. Using n and pl, the *reference patterns* are determined and stored in the vector RP (line 4). Afterward, the algorithm loops over the calculation steps from line 5 to line 10. The first calculation inside the loop (line 6) is a subtraction of a set from past data points and the *reference pattern* (RP) so as to determine a difference vector, which is stored (D). The difference vector D, as well as the *width* parameter w are passed over to the function WEIGHT in line 7. This function has two objectives:

Algorithm 1 Pseudo code of the prediction algorithm

```
1:  function PREDICTOR(Y,hl,pl,w)
2:      num,den ← 0                                   ▷ Initialize
3:      n ← GETLENGTH(Y)                              ▷ n gives the number of data points
4:      RP ← Y[n − pl,n − 1]                          ▷ RP includes the reference pattern
5:      for i ← 0,i ≤ n − pl − 1,i ← i + 1 do
6:          D ← Y[n − pl − i − 1,n − 2 − i] − RP       ▷ subtract RP from past data points
7:          β ← WEIGHT(D,w)                           ▷ weight and multiply all data points
8:          num ← num + β · y_{n−i−1}                 ▷ calculate numerator
9:          den ← den + β                             ▷ calculate denominator
10:     end for
11:     return y_n = num/den                          ▷ calculate and return result
12: end function
```

1. Each data points inside the vector D is weighted by a triangular function and is given a value between 0 and 1 so as to define the similarity based on the *width* w to the *reference pattern*. The triangular function is given by Equation (3.3). It is presented in Section 3.1.

2. All single weighted data points are multiplied among each other to calculate a single weight β which provides the weight for the data point following the considered set of past data points y_{n-1-i}. This operation equals the one given by Equation (3.4).

In line 8 the weighted sum of the considered data point and β is calculated (num). In line 9, the sum of all weights β is determined and stored in den. Both operations correspond to the Equations (3.5) and (3.6), respectively. After finishing the loop, the predicted value y_n is calculated by dividing the variables num and den (line 11). This line of code represents the calculation given by Equation (3.7).

4.2. Extending the Predictor

This section extends and fine tunes the generic predictor introduced in Section 3.1 and later implemented in Section 4.1. This serves two purposes: (1) To allow for the efficient selection of power-saving modes for any given idle period length, (2) To apply the predictor to forecast idle periods in DRAMs.

Before applying the predictor to a problem, such as the prediction of memory idle periods, a parameter set that includes the *history length*, the *reference pattern* length, and the *width* has to be chosen. Selecting such a set of parameters means setting up the predictor with a certain configuration, which is not changeable during run-time. This fixed configuration leads to a problem situation where the described predictor cannot be directly used to predict DRAM memory idle periods. The problem arises from the parameter *width*, which defines the allowed difference between the *reference pattern* and the patterns from the history. For a set of idle periods with a maximum length of up to a few hundred clock cycles, the *width* can be set to a small number, since the relative variation in the idle periods is small. However, most applications have large variations of idle period length which ranges from just a few clock cycles, to several hundreds of thousands clock cycles. If the *width* is set to a small number, nearly matching pattern in higher ranges of idle periods (several thousands) might not be taken into account, as small relative variations appear large and hence, are not to be considered matches. A similar problem arises if the *width* is set to a larger number, because small idle periods (a few hundred) begin to match even though they have no similarity. To resolve this issue, an extension to the predictor that can predict the length of idle periods even under circumstances where there are large variations, is presented.

The concept of *prediction on levels* is introduced as an extension to the prediction algorithm. *Levels* are used to classify the different idle period lengths within a set of predefined ranges. The different *levels* of the idle periods, which represent their lengths, are used as data points in the generic prediction algorithm, as introduced in Section 3.1. The *width* parameter is now applied on to the allowed difference in *levels* and not on to the exact idle period lengths. Therefore, a small value for the *width* parameter is sufficient. Figure 4.1 presents an illustrative example of how to use *levels* to predict idle period lengths. The figure shows two requests depicted with cross-hatched bars, as well as an idle period in between. All idle periods in the range from $[x_{i-1}, x_i]$ are grouped together to *level* l_i.

In Section 3.2.3 the two power-saving modes, the *self-refresh* mode and the *power-down*

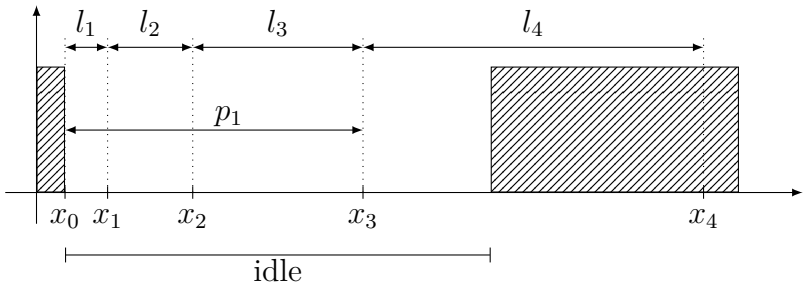


Figure 4.1.: Idleness Prediction on *levels*

mode, were analyzed. The SRT was derived to define when the *self-refresh* mode becomes more efficient than the *power-down* mode, specifically from which clock cycle period on. The SRT for the Micron 1 GB DDR3-800 memory [40] is at 3691 cc. The goal of the later introduced power-policy is to reduce power consumption through the application of the *self-refresh* mode as often as possible. To assure that the *self-refresh* mode is always more efficient 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 of the current one. Therefore, *level 3* includes idle period lengths from 7382 cc to 14 763 cc. Table 4.1 presents the length of the idle periods for the first ten *levels*.

Level	Min	Max
1	1	3690
2	3691	7381
3	7382	14763
4	14764	29527
5	29528	59055
6	59056	118111
7	118112	236223
8	236224	472447
9	472448	944895
10	944896	1889791

Table 4.1.: Minimum and maximum length of idle periods using *levels*

Based on these *levels*, *self-refresh* is not profitable for *level 1* idle periods. For all other *levels*, the *self-refresh* mode is the favored power-saving mode. The introduction of *levels* reduces the variation in idleness to a smaller range from *level 1* to *level 10* instead of from 0 cc to 1 889 797 cc. Hence, a small *width* parameter can be successfully employed. However, since the prediction is performed on *levels*, as is shown in Figure 4.1, predicting

a *level* l_i equates to a conservative idle period length of x_{i-1}. This is because the lower bound of the *level* range l_i is used as the predicted value. The history now consists of different *levels* of idle periods in the past, and the pattern comparison is based on the *levels*. Note that since *levels* represent ranges of idleness, all predictions may not be accurate at the single clock cycle *level*. Moreover, because the predictor forecasts conservatively, 100% of all of the idle cycles within some of the idle periods may not be exploited by the prediction.

4.3. Power Saving Policy

In this section, a novel power-saving policy is proposed. This policy employs the prediction algorithm presented and extended in Section 4.2 in combination with a time-out strategy for identifying memory idleness. This policy enables the use of either the *self-refresh* mode, the *power-down* mode, or both, while avoiding or reducing the penalty cycles. The standard Time-Out strategy, briefly discussed in Section 4.3.1, is used to weed out any speculative usage of the *self-refresh* mode. Additionally, the prediction algorithm is employed multiple times in every idle period to allow for the effective use of the *self-refresh* mode, described in Section 4.3.2. Moreover, the policy is also employing the *power-down* mode speculatively when some idle cycles are not exploited using the *self-refresh* mode (described in Section 4.3.3).

4.3.1. Time-Out

The standard time-out strategy is a widely used technique in the field of computer science and engineering, such as in [13, 25, 35, 49]. This strategy waits for a pre-defined time-out interval before the system reacts to an event.

In this thesis, the standard time-out strategy is used to avoid the unnecessary use of the *self-refresh* mode. A pre-defined time-out interval is used before applying a power-saving mode. The idle periods of the memory can vary between a few and several thousand clock cycles, as already explained in Section 4.2. However, the goal of the power-saving policy is to use the *self-refresh* mode as often as possible so as to reduce power consumption. The *self-refresh* mode is not efficient for short idle periods because of its long power-up latency, as shown in Section 3.2.2. For short idle periods, this 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 by filtering short idle periods.

4.3.2. Prediction for Self-refresh

This subsection analyzes the effectiveness of the prediction algorithm in efficiently employing *self-refresh* in combination with the time-out strategy described earlier.

The prediction is done on *levels* as described in Section 4.2. The predictor conservatively forecasts idle periods longer than the SRT in order to avoid the penalty cycles when employing the *self-refresh* mode. Based on the predicted *level*, the memory powers-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 so as to weed out very short idle periods. If the predictor forecasts an idle period shorter than the SRT, which is equivalent to *level 1*, these idle periods are neglected and the *self-refresh* mode is not used. However, it is possible that the predictor under or over-estimates the idle period length. The former is more probable, because the predictor always provides a conservative estimate for the idle period length. This is due to the lower bound of the predicted *level* being used. The solution to the misestimation problem is explained in the following two subsections.

Reasons for Estimation Problems

If the predictor forecasts idle period lengths shorter than the actual idle period, it can be for two reasons. The first is due to the fact that the prediction is done in terms of *levels*. It provides the lower bound of a particular *level* as the prediction value, and therefore, may neglect some idle clock cycles in the corresponding idle period. The second reason for under-estimating the length of the idle periods is that very long idle periods can be rare and far apart.

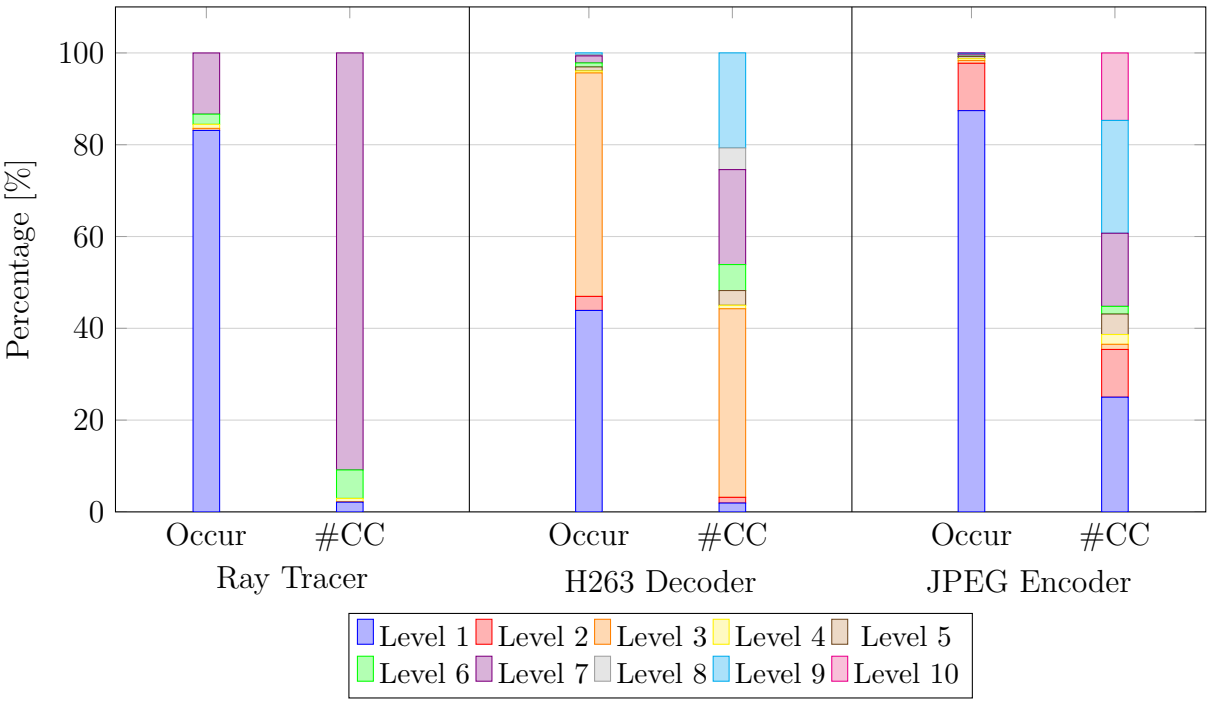


Figure 4.2.: Histogram of idleness for different multimedia benchmarks

Figure 4.2 depicts this problem by providing an analysis of the occurrence of idleness for three different multimedia benchmarks, which are later used to verify the proposed

power-saving policy. The left bar for each benchmark shows the occurrence of each *level* in percentages, respectively. For example, for the ray tracer benchmark, 83% of all *levels* are assigned to *level 1*. The right bar of each benchmark gives the total number of the idle cycles, which are depicted as *#CC* within a specific *level*. Again, a closer look at the ray tracer benchmark reveals that only 2% of all idle cycles are included in *level 1*. Additionally, Table 4.2 provides the total number of *level* occurrences as well as the total number of clock cycles assigned to the corresponding *level*, respectively. As can be seen in the ray tracer and JPEG encoder benchmark, most of the idle periods are assigned to *level 1*. However the total number of idle cycles within this *level* are low. On the other hand, the *levels* with higher numbers, like 7 to 10, occur rarely but include most of the idleness. The H263 decoder benchmark has a slightly different distribution of idleness. Nevertheless, over 40% of idle periods are assigned to *level 1*, which contributes less than 2% to the total idleness. Again, long idle period rarely occur (*level 7* to *10*).

	Ray Tracer		H263 Decoder		JPEG Encoder	
Level	#Levels	#CCs	#Levels	#CCs	#Levels	#CCs
1	187	140904	201	109886	1314	1618824
2	0	0	14	70202	155	673966
3	1	8864	223	2333868	9	72630
4	2	46244	2	42944	6	138806
5	0	0	4	182446	7	289214
6	5	412364	4	321740	1	108396
7	30	6047792	7	1175118	7	1032272
8	0	0	1	268112	0	0
9	0	0	2	1176242	3	1592350
10	0	0	0	0	1	952480

Table 4.2.: Histogram

Using the information from Figure 4.2 and Table 4.2 the problem of underestimating long idle periods can be better understood. The predictor is history-based and if a long idle period has not been predicted before or if it is already out of the limited history buffer, then as a result, the predictor under-estimates. This is due to the missing large value that is similar in the history buffer.

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 within the idleness.

Solution for Estimation Problems

To solve the under-estimation problem, this section proposes to employ *multiple predictions* in a given idle period. This extends the use of the *self-refresh* mode if it is viable, and powers-up the memory as late as is possible, just in time to be accessed again. This policy is depicted in Figure 4.3.

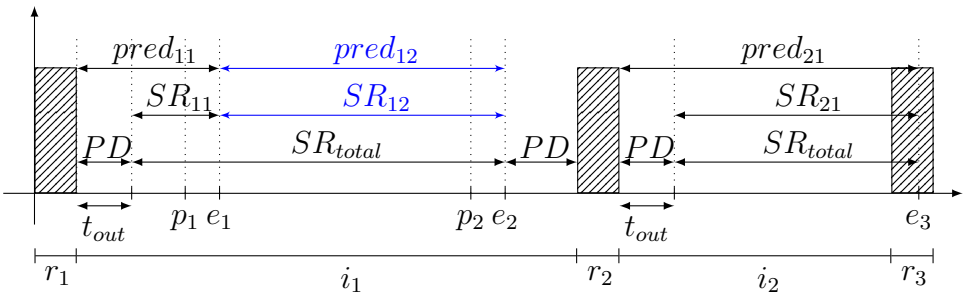


Figure 4.3.: Multiple predictions for Self-Refresh

In this figure, the requests are shown as hatched bars and denoted as r_x and idle periods occur 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 to see if it is greater than *level 1* (by at least SRT cycles) and if so, a *self-refresh* (SR_{11}) is scheduled. The memory is expected to start to power up at p_1 so as 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 profitable for an additional *self-refresh*, the memory continues to power up. However, if the predicted value is still profitable ($pred_{12}$), then 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 made 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 profitable or a misprediction occurs. As already explained, the prediction is done conservatively on *levels* and therefore, 100% of the exploitation of each idle cycles in every idle period are not possible when using *self-refresh*.

Figure 4.3 also shows the idle period i_2 which has an over-estimation problem. After an initial time-out, a *self-refresh* (SR_{21}) is scheduled based on the length of the predicted value ($pred_{21}$). The predicted value becomes 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 number of cycles the memory are already powering-up.

4.3.3. Proposed Power-Saving Policy

As stated in Section 4.3.2, by performing multiple predictions per idle period, it is possible to exploit most of the idle cycles using the *self-refresh* mode. However, because the prediction is done on *levels*, a 100% exploitation of idle cycles is not possible for all of the idle periods. To resolve this issue for the unexploited idle cycles, multiple predictions for the *self-refresh* mode combined with the speculative use of the *power-down* mode is proposed 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 it is less than the savings from *self-refresh*. However, it is 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) a combination of the two modes to exploit most idle cycles by *self-refresh* and the rest by the *power-down* mode. All of these features exist 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 profitable (*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 uses a prediction for the *self-refresh* mode and also schedules a speculative *power-down* for the idle clock cycles not covered by *self-refresh*, thereby covering 100% of the idle cycles by either of the two power-saving modes.

Figure 4.3 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.

The proposed power-saving policy is based on the assumption that the predictor can also provide a result for the next prospective data point. However, it is also possible that the predictor is not capable of forecasting. This can be the case if there exist no similarity between the *reference pattern* and all sets of past data points. Hence, the result of the prediction can be either, a `hit`, a `miss`, or `no result`, as depicted in Figure 4.4. In

case the predictor is not capable of calculating a result, the proposed power-saving policy assumes a *level-1* prediction and does not apply *self-refresh* to avoid penalty cycles.

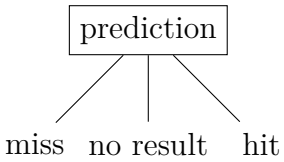


Figure 4.4.: Possible predictor results

4.4. Hardware Implementation

This section introduces the hardware implementation of the predictor explained in Section 3.1 and realized as software version in Section 4.1. To implement the predictor in VHDL, several constraints are made. The introduced algorithm works on rational numbers, so that the result can be any number with several position after the decimal point. This is difficult to realize in hardware and requires a lot of effort as, all operations must be performed on floating point numbers. However, the proposed power-saving policy, presented in Section 4.3, needs unsigned integers as input. Then the predictor makes internal calculations with floating point numbers and provides unsigned integer as output. The internal calculation with floating point numbers was used to reduce the effort for the software implementation. However, a hardware implementation favors the use of unsigned integers to reduce the hardware complexity, because components like adder and multiplier are less complex compared to the equivalent components operating on floating point numbers. Hence, the hardware predictor is implemented to work on an unsigned integer.

The hardware implementation of the prediction algorithm is done in two steps: First, the data path is introduced to realize the prediction algorithm from Section 3.1. Therefore, the equations from the five calculations steps, namely (1) build history, (2) calculate absolute differences, (3) determine weight/similarity, (4) weight past data points, and (5) forecast data point, are implemented in hardware. Second, the control unit is introduced and explained in detail to complete the full predictor unit.

4.4.1. Data Path

Figure 4.5 gives an overview of the data path from the predictor unit including all of the components necessary to realize the predictor algorithm.

In the first step, described in Section 3.1, the predictor builds up a history before starting to forecast the next data point. The *historybuffer* component is realized as a generic *First In First Out* (FIFO) buffer, where each element is also placed on the output. This component is generic in terms of the maximum elements stored in the buffer and equals the parameter *history length (hl)*. Moreover, the maximum number of bits per element can be set by the parameter *register size (rs)*.

Step (2), which calculates absolute differences, and step (3), which determines weight/similarity, are both performed in the component *Predictor Core*. Figure 4.6 shows the inner design of this component and presents the *historybuffer* on top of this component, addi-

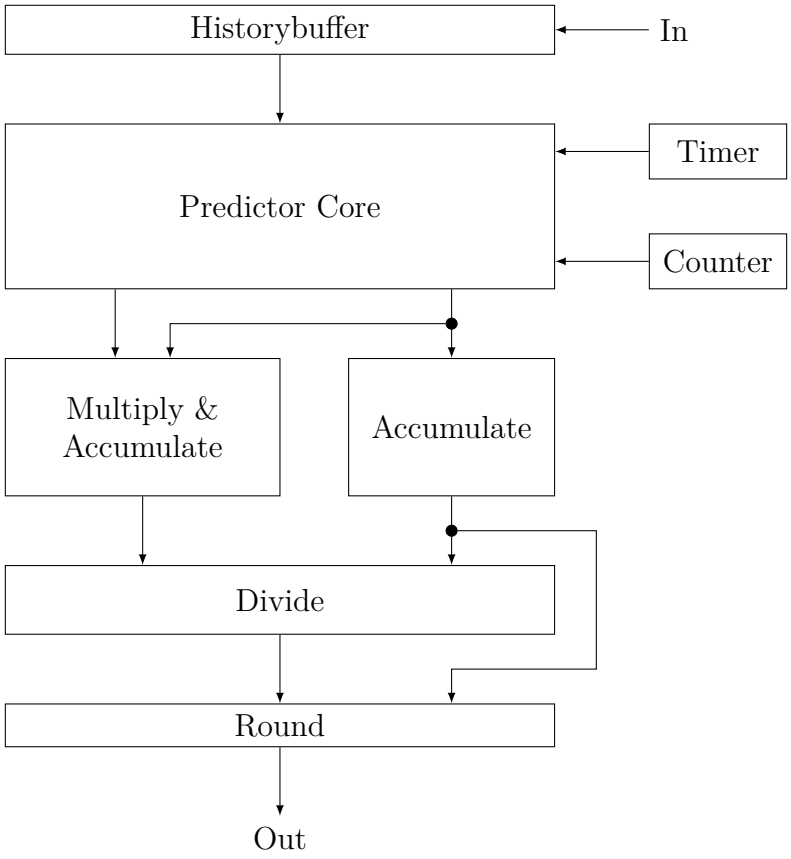


Figure 4.5.: Components in the data path of the predictor

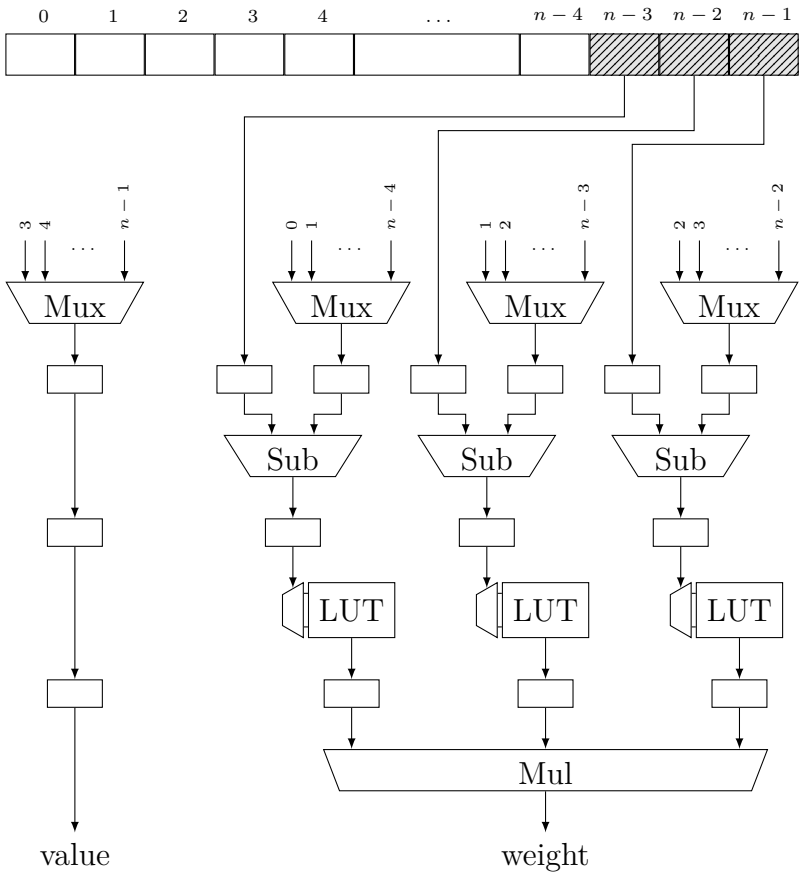
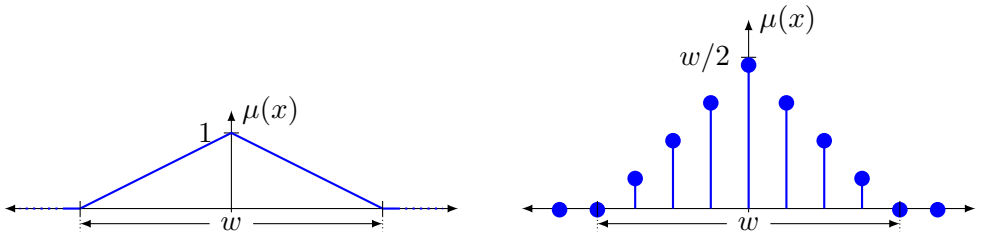


Figure 4.6.: Predictor Core in detail

tionally. The latest entries of the *history buffer* are the *reference pattern* and are depicted as crosshatched lines. However, the predictor is also generic in terms of the number of elements used as *reference pattern* and can, therefore, be chosen by the user. For sake of simplicity, the number of elements for the *reference pattern* is set to three in this figure. Each element of the *reference pattern* is connected to a *Subtractor* (Sub), respectively. The other entry of each Sub is provided by a *Multiplexer* (MUX), which can select one arbitrary past data point out of the *history buffer*. The counter shown in Figure 4.5 is used to select one set of past data points and sets each of these to each of the Subs. For example, assume that the counter is set to 0, then the difference vector $D_i = Y[n-3,n-1] - Y[n-4,n-2]$ is calculated. The counter operates clockwise, so that for every clock cycle, another set of absolute differences between the *reference pattern* and the past data points is calculated until the entire history is probed.



(a) continuous (b) discrete

Figure 4.7.: Triangular functions for floating point numbers and unsigned integers

In step (3) each absolute difference is weighted based on the similarity to the *reference pattern*. Therefore, a triangular function, like the one depicted in Figure 4.7a, was used to set all non-similar values to zero (non similarity) and weight then the rest with a value between zero and one. The parameter *width* (w) was used to define which values should be taken into account so as to forecast the next data point and which data points are negligible. This triangular function works with floating point numbers. However, the hardware implementation of the predictor can only handle unsigned integers, and therefore, the triangular function must be adapted. By extending the Y-range from [0,1] to $[0, w/2]$ all differences can still be weighted due to their similarity using only unsigned integers. This can be achieved by using a discrete triangular function like the one depicted in Figure 4.7b. The weighting is implemented in hardware as *lookup table* (LUT) which is generated during synthesis. After weighting each absolute difference, the entire set of considered past data points have to be weighted. For that reason, all weighted data points are multiplied among each other to generate a single weight for the whole set,

as shown in Figure 4.6. The *Predictor Core* then provides both, the *weight* as well as the corresponding *value*, as outputs. Moreover, after each stage (MUX, Sub, LUT), an intermediate register is inserted to pipeline the *Predictor Core* and increase the clock frequency. This was done, because after the first RTL implementation was finished, a short analysis showed that the critical path was within this component.

As already mentioned, the predictor is also generic in terms of the number of data points used as *reference pattern* and is given by the parameter *pattern length (pl)*. This parameter defines the implemented number of calculation chains (MUX, Sub, LUT).

In Section 4.3 the power-saving policy is presented, which is using a standard time-out strategy to weed out short idle periods. Therefore, as an additional component, a timer is connected to the *Predictor Core* as depicted in Figure 4.5. The timer is used to delay the prediction process by a user defined amount of clock cycles.

To calculate the denominator D as explained in step (4), all single weights forwarded from the *Predictor Core* have to be accumulated. A standard accumulator taken from [63] is used for this task. To determine the numerator N, each single weight has to be multiplied with the corresponding past data point. Furthermore, these products must be accumulated. Again, a standard component (multiply-accumulate) was taken from [63].

Finally, the numerator is divided by the denominator in step (5) to forecast the next data point. A radix-2 divider, presented in [42], was modified and extended to handle 64-bit numbers and with the option to interrupt the division in progress, so that the divider is usable for the predictor.

Unfortunately, the divider produces as output quotient and remainder and does not round up or down. However, the result of the predictor must be rounded. A value greater or equal to .5 should be rounded to the next unsigned integer. Values smaller than .5 should, in turn, be rounded to the previous unsigned integer. An additional *rounding unit* (see Figure 4.5) was implemented. This rounding unit computes based on the quotient, remainder, and divisor a rounded unsigned integer as a result. The inner structure of the *rounding unit* is depicted in Figure 4.8. The rounding unit distinguishes whether the divisor is an even or odd number. The results of the division are rounded up, if the divisor divided by 2 is less or equal to the remainder for even numbers, or if the divisor divided by 2 is less than the remainder for odd numbers. To implement this calculation, the divisor is shifted right by one position (division by 2) and compared in parallel to see if this value is less or less or equal to the remainder. Based on an even or odd divisor, one of the two comparators forwarded the result by a MUX to an adder. The output of both comparators are an 1 bit value and they indicate whether or not the quotient must be

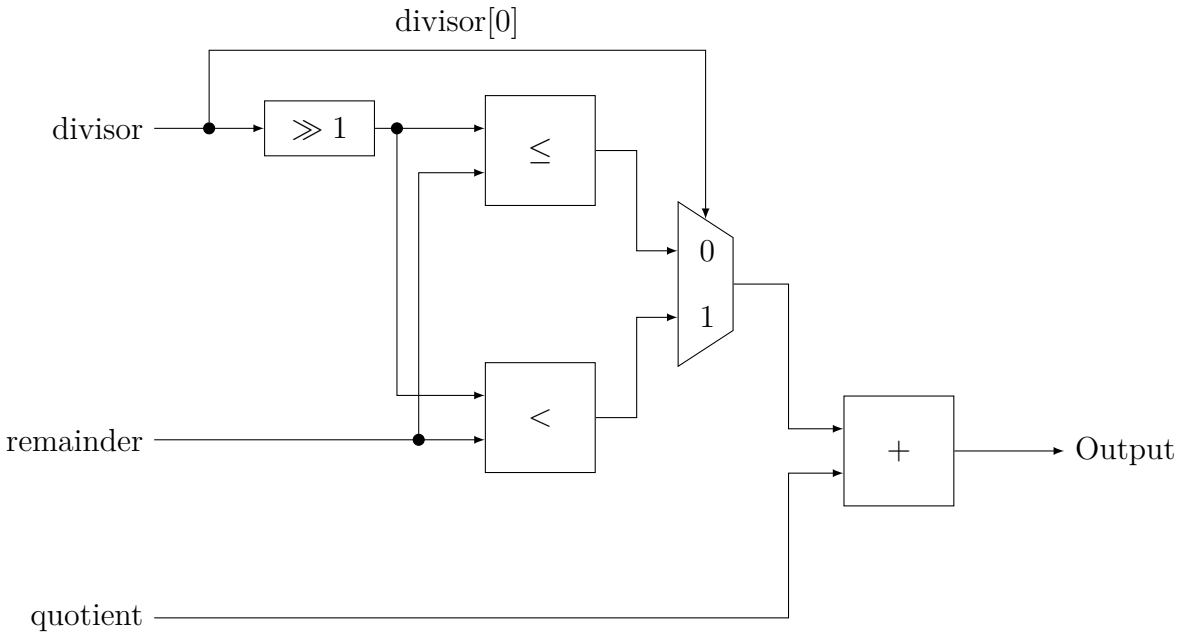


Figure 4.8.: Rounding unit

rounded up. Finally, the adder increases the quotient by 1 depending on the comparators output (rounded up), or it leaves the quotient as calculated (rounded down).

4.4.2. Control Unit

The component depicted in Figure 4.5 shows only the data path of the predictor. A control unit is needed so that all single components work together and forecast the next prospective data point. The predictor control unit has to handle all components depicted in Figure 4.5 so that all operations, given by Equations (1) to (5) from Section 3.1, carried out in the correct order. Additionally, the predictor needs the option to delay the prediction process by an amount of clock cycles so as to realize the time-out strategy presented in Section 4.3.1. Moreover, if new data arrives, then the prediction in progress must be interrupted to update the historybuffer, because the history has changed and the calculation in progress becomes invalid. Afterwards, the prediction must start over again to forecast the new value based on the updated history. The *predictor control unit* is realized as FSM and is depicted in Figure 4.9.

The FSM consists of 5 states, namely: `idle`, `data`, `wait`, `accu` and `div`. Starting with the `idle` state, the FSM waits for data input so as to build up a history. Every time a new value arrives (`new_data`) the FSM changes the state to `data`, adds the value to the historybuffer (`done`), and returns to the `idle` state. As long as the historybuffer is not

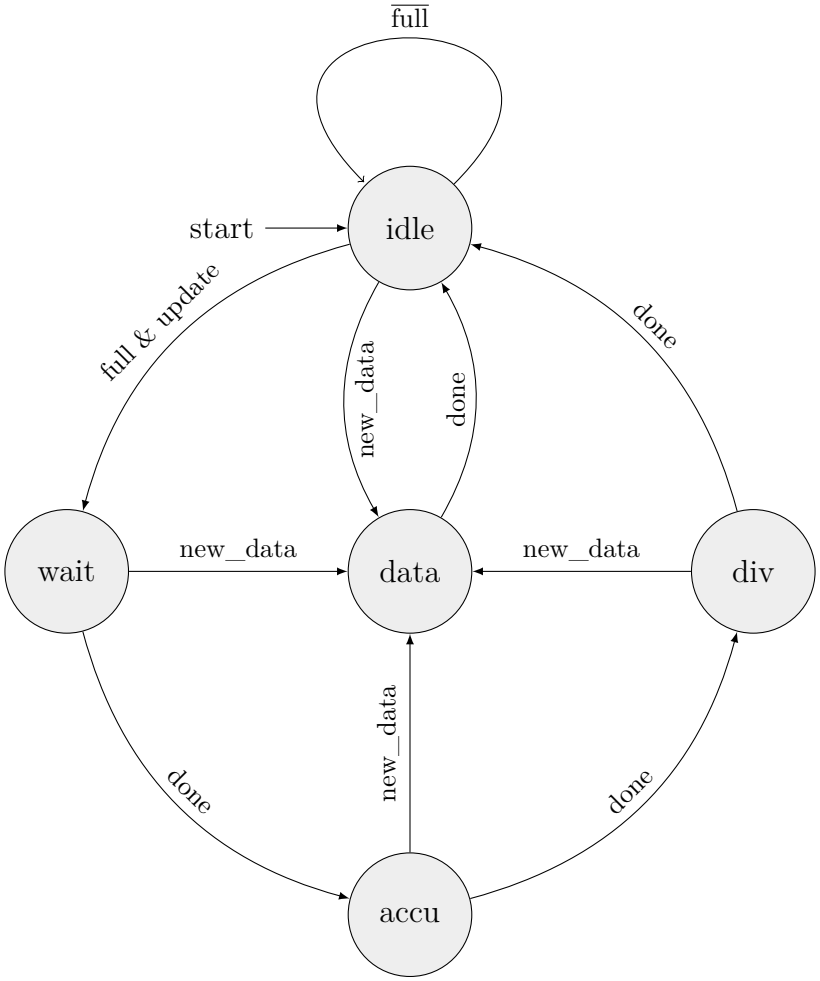


Figure 4.9.: Predictor FSM

full, depicted by the control signal `full`, the FSM stays in `idle` state. Moreover, a second control signal `update` is used to prevent a new prediction after the first one has finished. This signal only changes in case the history is updated. In the `data` state, the control signal `update` is set after the history is updated. Once the historybuffer of the predictor is full and the signal `update` is set, the FSM changes to the `wait` state. In this state a counter is started to delay the prediction process for an amount of clock cycles, so as to realize the already mentioned time-out strategy. To directly start the prediction and skip the time-out, the counter can be also set to zero. In the case of new data, the countdown is interrupted and the state changes back to the `data` state. If the counter runs out, the FSM goes directly to state `accu`, which triggers the *Predictor Core* to start calculating the weight for a set of past data points each clock cycle. The outputs of the *Predictor Core* are handed over to the accumulator and the multiply-accumulator so as to calculate the numerator and denominator, respectively. Once the whole history is processed, the FSM changes to state `div`. If new data arrives, the FSM immediately turns back to the `data` state so as to process the new input and then start over again. Once, in `div` state the final result is calculated by performing the division process. Again, if new data arrives during this process, the FSM returns immediately to state `data` for processing the new data and start over again. After the prediction process has finished, the FSM stays in the `idle` state until new data arrives and the history buffer has been updated, depicted by the control signal `update`.

4.4.3. Predictor Runtime

All components of the hardware predictor have a predetermined runtime based on four parameters. These parameters are: the register size (rs) of each element inside the history buffer, the *history length* (hl) - which defines the number of data points stored inside the history buffer, the number of data points used as *reference pattern*, given by the *pattern length* (pl), and the *width* (w) - which defines the similarity between the *reference pattern* and past pattern. The parameters are already explained in Section 3.1. Based on the runtime for each component, the overall runtime of the predictor can be calculated. There are two main contributors to the total latency of the predictor: (1) The *Predictor Core* with the following accumulation components, and (2) the divider. Both contributors are dependent and work sequentially.

The first contributor probes the *reference pattern* with a set of past data points from the history buffer each clock cycle. Moreover, each single probe is forwarded to the accumulator components to calculate the numerator and denominator clockwise. The

number of total probes is given by Equation (4.1).

$$\#probes = hl - pl + pipeline_{core} - 1 \tag{4.1}$$

This equation defines the number of clock cycles needed by the components *Predictor Core*, *Accumulate* and *Multiply & Accumulate*. Since, the *Predictor Core* is pipelined the constant $pipeline_{core}$ is added and set to three, because of the three pipeline stages.

To calculate the final result, the second contributor, the divider, performs a final division. The runtime of the used radix-2 divider [42] equals to the maximum number of bits for either the dividend or the divisor. When comparing both inputs, the dividend is always greater or equal to the divisor. The dividend is calculated by the *Multiply & Accumulate* component. The output size of this component is given by Equation (4.2). For each clock cycle, the *Multiply & Accumulate* component multiplies the output values, *weight* and *value*, that are calculated within the *Predictor Core* (see Figure 4.6), and adds them up until the entire history is probed. The size of a multiplication is the sum of the size of both inputs. In the worst-case, each accumulation step can increase the output of the result by 1 bit. In total there are $hl - pl$ additions and the bit size of *value* equals the register size *rs*.

$$output_{size} = weight + value + hl - pl \tag{4.2}$$

$$value = rs \tag{4.3}$$

The bit size of the *weight* depends on two factors: the number of data points used for the *reference pattern* (*pl*), and the bit size of the output from the LUT used inside the *Predictor Core* (as depicted in Figure 4.6). There are in total $pl \times$ LUTs and all outputs are multiplied among each other to determine the output value *weight*. The maximum output size of each LUT is the number of bits to code the maximum number of the discrete triangular function, as can be seen in Figure 4.7b. The bit size of *weight* is given by Equation (4.4).

$$weight = \lceil \log_2(\frac{w}{2} + 1) \rceil \cdot pl \tag{4.4}$$

Using Equations (4.2), (4.3), and (4.4) results in the bit size of the output value from

the *Multiply & Accumulate* component and is, therefore, the runtime of the divider. The bit size is given by Equation (4.5).

$$output_{size} = \lceil \log_2(\frac{w}{2} + 1) \rceil \cdot pl + rs + hl - pl \tag{4.5}$$

Finally, having analyzed the runtime of the two main contributors, the total predictor runtime can be determined by using the runtime of all other components. The total runtime of the predictor is given by Equation (4.8). An additional constant *wait* is added to represent the optional time-out strategy.

$$\begin{aligned} pred_{cc} &= \#probes + output_{size} + wait && (4.6)\\ &= hl - pl + pipeline_{core} - 1 + \lceil \log_2(\frac{w}{2} + 1) \rceil \cdot pl + rs + hl - pl + wait && (4.7)\\ &= \lceil \log_2(\frac{w}{2} + 1) \rceil \cdot pl + rs + 2 \cdot (hl - pl) + 2 + wait && (4.8) \end{aligned}$$

4.5. Summary

This chapter has depicted the implementations presented in this thesis. First, a software implementation of the prediction algorithm was introduced. The algorithm was implemented twice, first in Octave to analyze the accuracy and second in SystemC for later integration in the CompSOC platform. To apply the predictor to forecasting memory idle periods, an extension was presented and the concept of predicting on *levels* was introduced. The proposed power-saving policy was then introduced and explained in detail. This power-saving policy combines an initial time-out, the prediction for *self-refresh* mode, as well as the speculative use of the *power-down* mode. Moreover, the concept of multiple predictions within an idle period was introduced so as to exploit most of the idleness by the *self-refresh* mode. Finally, a hardware implementation of the predictor in VHDL was presented for a power and complexity analysis. The data path and the control unit were introduced separately. Finally, the runtime of the hardware implementation was discussed.

5. Experimental Evaluation

This chapter provides several experimental analysis to verify the proposed predictor and power-saving policy. First, the accuracy of the predictor is validated in Section 5.1. Therefore, a design space exploration with different parameter sets for the predictor are analyzed. This analysis identifies the parameter sets with the highest accuracy, which are used in the following sections. Afterward in Section 5.2, the power-saving policy for DRAMs, presented in Section 4.3, is analyzed and validated.

A time-out strategy using a speculative *self-refresh* is analyzed. The impact of these results are combined with a prediction of the idle period lengths so as to power-up the memory before a wake-up penalty can arise. Moreover, a speculative *power-down* is added to this strategy for the final predictor based power-saving policy. This policy exploits all idle cycle with either *self-refresh*, *power-down* or a combination of both. All of these strategies, as well as the final power-saving policy are validated by analyzing the power consumption and the execution time.

Section 5.3 presents a complexity analysis, a frequency analysis and a power analysis of the predictor running on an FPGA. These analyses allow the verification of the proposed power-saving policy for DRAMs.

After analyzing the power-saving policy, the predictor is also tested on network traffic in Section 5.4, investigating its applicability to other fields of applications. The predictor is tested on forecasting several data points ahead. Therefore, the predictor receives already predicted data points as input.

Finally, this chapter is concluded with a summary of all results in Section 5.5.

5.1. Accuracy Analysis

This section analyzes the accuracy of the predictor to the three main parameter introduced in Section 3.1. The three parameters are:

1. the *history length* (hl), which defines how many past data points are taken into account when searching for similar patterns
2. the *pattern length* (pl), which defines the number of the latest data points taken as the *reference pattern*
3. the *width* (w), which defines the similarity between the *reference pattern* and a set of past data points

This section analyzes the impact of all three parameters and determines the best sets of parameters with the highest accuracy for a certain predictor configuration.

5.1.1. Experimental Setup

To analyze the impact of the three parameters, different applications from the multimedia domain, like MediaBench [32], are used. From this set of benchmarks three different multimedia applications are used: (1) H263 decoder, (2) Ray Tracer, and (3) JPEG encoder. The software implementation of the predictor (Octave [27]) is used for this analysis. To determine the set of parameters with the highest accuracy, a design space exploration is performed. The design space as depicted in Table 5.1 was used.

Table 5.1.: Design space for the accuracy analysis

Parameter	Range
History length	10,20,30,40,50
Pattern length	2,3,4,5
Width	2,4,6,8

The predictor is used for the power-saving policy, as introduced in Section 4.3, to forecast memory idle cycles. However, this policy does not operate on the real length of the idle period, but on *levels* as explained in Section 4.2. For that reason, the accuracy analysis is also done on *levels* which represents the idle period length. Moreover, this analysis should identify the best parameters for the later used power-saving policy. Three possible results can arise by using the predictor to forecast the next prospective data

point: (1) `miss`, (2) `no result` and (3) `hit`. A `miss` is obvious and means the predictor forecasts an idle period that is longer than the actual one. `No result` simply means that the predictor has found no similarity between the *reference pattern* and the pattern in the history buffer. A `hit` depicts that the predicted idle period is equal or less than the actual one. However, this hit has to be subdivided in a `perfect hit` and a `short hit`. A `perfect hit` arises when a predicted idle period is equal to the actual one. The `short hit` indicates that the predicted idle period is shorter than the actual one. All possible results are depicted in Figure 5.1, which is an extension of Figure 4.4. Hence, the hit is defined in that way because in both cases no penalty can arise for the power-saving policy explained in Section 4.3. Moreover, `short hits` can be used for multiple prediction within the same idle period to exploit more idle cycles with *self-refresh*. This is explained in Section 4.3.2.

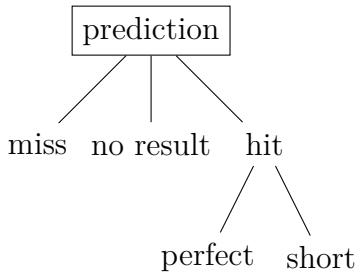
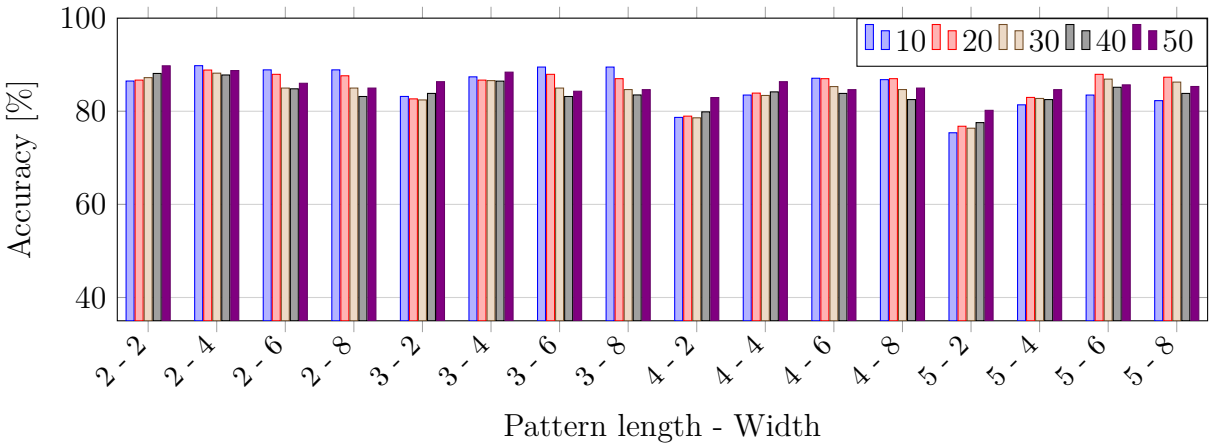


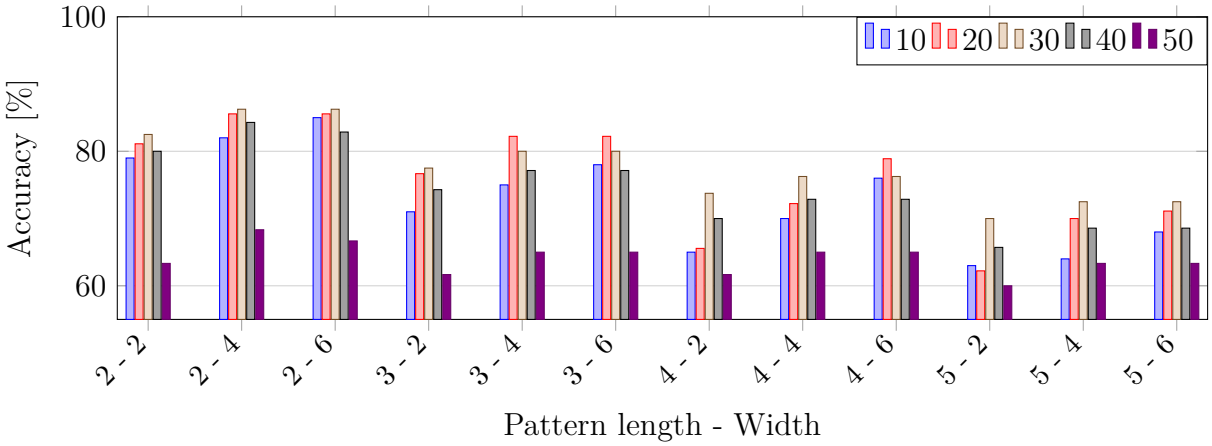
Figure 5.1.: Possible predictor results

5.1.2. Experimental Analysis

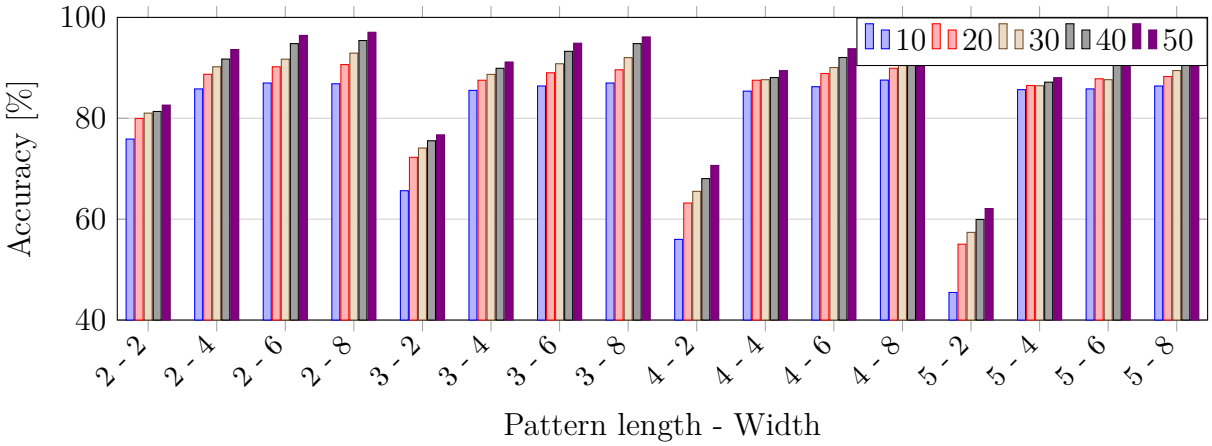
A complete design space exploration was performed and the accuracy was calculated for each set of parameters as well as for each used application, respectively. The results are depicted in Figure 5.2. The figure shows the hit rate in percentages for all applications on the y-axis. The hit rate is defined as sum of *perfect hits* and *short hits*. The output value *no result* is neglected for this analysis, because this type of result is converted by the power-saving policy to a *level 1* prediction, as explained in Section 4.3.3. The hit rate results will be discussed in more detail, later in this section along with the distribution of `perfect hits`, `short hits` and the occurrence of `no result`. Each bar of the same color represents a predictor configuration with the same *history length*. Moreover, the x-axis gives for each set of bars, the *pattern length* as first number and the *width* as second number. For example a set of bars with an x-axis label of 3-4 defines a predictor configuration with a *pattern length* of three and a *width* of four.



(a) H263 decoder



(b) Ray Tracer



(c) JPEG encoder

Figure 5.2.: Accuracy analysis

Having a closer look at Figure 5.2 reveals that for each application a different behavior of the accuracy arises, and that the accuracy of the prediction for an application depends on a certain predictor configuration. Hence, the three applications will be analyzed independently.

Figure 5.2a depicts the accuracy for the H263 decoder. Increasing the *pattern length* results in lower accuracy. This can be seen by comparing the sets of bars with equal *width* but different *pattern length* (e.g. 2-2 compared to 3-2, 4-2 and 5-2). This identifies a *pattern length* of 2 as the best parameter for the highest accuracy. Analyzing the accuracy in relation to the history length shows that for the majority of sets, the accuracy decreases with the increasing history length.

However, there are some exceptions e.g. 2-2, 3-2, 4-2, 5-2, where the accuracy increases with longer history length. Using a *width* of 2, means that only perfect matches between the *reference pattern* and the past data points are used for the prediction, as differences greater or equal ± 1 are already neglected (see Section 3.1). For perfect matches a longer history length increases the chance to find a similar pattern and therefore results in a higher accuracy. However, these exceptions have a lower accuracy, like the ones with a *width* that is greater than 2. For that reason, short history length results in a higher accuracy and identifies a history of 10 as best parameter. A *width* of 4 is identified as best parameter, because using values greater or equal to 6 already results in a decreased accuracy. The best parameters for the H263 decoder are summarized in Table 5.2.

The design space for the Ray Tracer application reveals that the parameters have a different influence on the accuracy. This is presented in Figure 5.2b. First, the *history length* is analyzed. For almost all sets of bars, the accuracy increases with the *history length* until a maximum of 30 and afterwards, begins to decrease. For some configurations, like 3-4, 3-6, 4-6, the maximum is already achieved with a *history length* of 20. However, all of these sets already have a lower accuracy than the ones with a shorter *pattern length*. This identifies a *history length* of 30 as the best parameter for this specific application. Analyzing the influence of the *width* requires the consideration of the *pattern length* as fixed number. As can be seen in all sets of bars, the maximum accuracy is obtained with a *width* of 4. Considering the *width* as a fix number reveals, that the accuracy decreases with increasing *pattern length* and identifies, therefore, a *pattern length* of 2 as best parameter. Results for a *width* of 8 are not included in this figure as the accuracy is much lower. Again, the best set of parameters for the highest accuracy is summarized in Table 5.2.

Figure 5.2c depicts the accuracy analysis for the JPEG encoder and presents again a different effect of the parameters. For this application, a longer *history length* results in a

higher accuracy and reaches its maximum by a length of 50. The influence of *width* can be directly seen in this figure. The accuracy increase up to a *width* of 6 and remains the same for 8, which identifies a *width* of 6 as the best parameter. Again, by increasing the pattern length, the accuracy of the predictor decreases, which identifies a *pattern length* of 2 as the best parameter. The best parameter set is depicted in Table 5.2.

Table 5.2.: Parameter set for highest accuracy

Application	hl	pl	w	accuracy [%]	perfect [%]	short [%]	no result [%]
H263 decoder	10	2	4	89.79	89.84	10.16	4.8
Ray Tracer	30	2	4	86.25	84	16	7.5
JPEG encoder	50	2	6	96.43	71.27	28.73	0.31

Analyzing all of the parameters in Table 5.2 reveals that all applications have a *pattern length* of 2. It leads to the conclusion that the reoccurring patterns of all used applications are very short and consist of only two data points. Both, the H263 decoder and the Ray Tracer, have a *width* of 4. This allows only perfect matches as well as matches that differ by ± 1. The reoccurring patterns for these two applications have a high degree of similarity. Compared to the JPEG encoder, their degree is lower, as a higher *width* is needed to achieve high accuracy. Each application possesses a different *history length*. Consequently, the reoccurring patterns of the H263 decoder (shortest *history length*) occur more often compared to the JPEG encoder. The JPEG encoder needs a larger history to detect the reoccurring patterns.

For all previous analysis, a hit was defined as predicted idle period length that is shorter or equal to the actual idle period, because no penalty arise when the predictor is used for the power-saving policy. However, to distinguish between the perfect hit and short hit, Table 5.2 also provides the values for both types of hits. As can be seen, the perfect hit rate is above 71% for all application, with a maximum of 89.84% for the H263 decoder application. Moreover, all previous analysis does not include how often the predictor calculates *no result*. This case can occur if their are no similarities between the *reference pattern* and all sets of past data points. The incidence of this case is also depicted in Table 5.2 and ranges between 0.31% and 7.5%. However, *no result* does not influence the predictor based power-saving policy, as it is assumed that the *self-refresh* mode is not efficient in this case. This means that the *self-refresh* mode is not used to avoid performance penalties, as already discussed in Section 4.3.3.

5.1.3. Summary of the Accuracy Analysis

To conclude this section, the predictor accuracy depends strongly on the set of used parameters for the predictor and is dependent on the used application. The consequence is that the predictor can only be used in an application specific manner. For each application a preliminary analysis has to be performed to identify a set of parameters for the highest accuracy. The parameter sets with the highest accuracy for each application will be used to validate the power-saving policy in Section 5.2 and the analysis of the power consumption of the predictor itself in Section 5.3. A summary of these parameter sets is depicted in Table 5.2.

5.2. Power-Saving Policy Analysis

The goal of the proposed power-saving policy, introduced in Section 4.3, is to reduce memory energy consumption while only marginally increasing execution time due to the power-up latencies. In Section 5.2.1, the experimental setup and the usage of the predictor within the system setup are introduced. In Section 5.2.2, three setups are analyzed:

1. The impact of the time-out strategy by speculatively employing the *self-refresh* mode
2. Using multiple predictions for *self-refresh* in an idle period
3. Employing the *power-down* mode speculatively for the idle cycles not exploited by the prediction in the *self-refresh* mode

For these three evaluations, the same applications, as well as the parameter sets determined in Section 5.1, are used.

5.2.1. Experimental Setup

For the experiments, a 1 GB Micron DDR3-800 [40] memory is employed and the proposed predictor-based power-saving policy is evaluated, with respect to both energy savings and the impact on execution time. The three different multimedia applications from the Mediabench [32] are used: (1) H263 decoder, (2) Ray Tracer, and (3) JPEG encoder. To evaluate the proposed power-saving policy for Section 4.3, each application runs on the Simplescalar simulator [6] with a 16 kB L1-Data cache, 16 kB L1-Instruction cache, a 256 kB shared L2 cache and 256 B cache line configuration. Afterwards, all L2 cache misses are filtered out and the transactions are obtained to the DRAM memory. Next, a trace player is employed to simulate the application behavior in a SystemC simulation model of the CompSOC platform, introduced in Section 3.3. These transactions from the trace player are forwarded to the DRAM memory controller [50] in the platform, which is modified to employ the predictor and the power saving policy logic.

The predictor is placed in the front-end of the memory controller, adjacent to the arbiter-bus. An overview of the CompSOC platform that includes the predictor and the memory controller is depicted in Figure 5.3.

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. Additionally, the predictor records the time stamps at the end of the last transaction before every idle

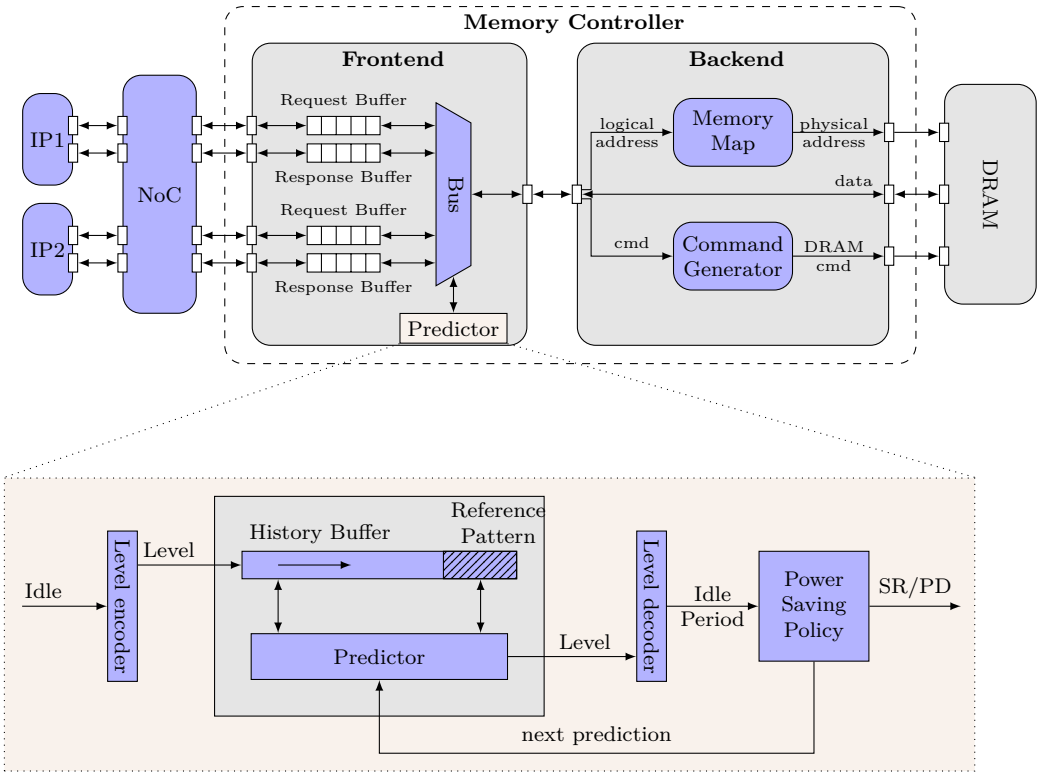


Figure 5.3.: CompSOC overview including the predictor

period. The length of the idle periods is calculated and encoded to *levels*, as explained in Section 4.2. 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 4.2.

Using the inferences from the accuracy analysis that were presented in Section 5.1 on the predictor, the sets of the parameter with the highest accuracy are used. These sets of parameters for the history buffer length (hl), the *reference pattern length* (pl), and the predictor *width* parameter (w) are depicted in Table 5.2 (Section 5.1). Additionally, two new parameters are introduced:

1. The best initial time-out (to) used for the time-out strategy (see Section 4.3.1)

2. The maximum number of predictor invocations (pi) used to limit the number of multiple predictions per idle period (see Section 4.3.2)

The impact of these two parameters are explained in more detail in the next section.

Table 5.3.: Parameter set for all application including time-out cycles and predictor invocations

Application	hl	pl	w	time-out [cc]	invocations
H263 decoder	10	2	4	300	70
Ray Tracer	30	2	4	300	220
JPEG encoder	50	2	6	300	90

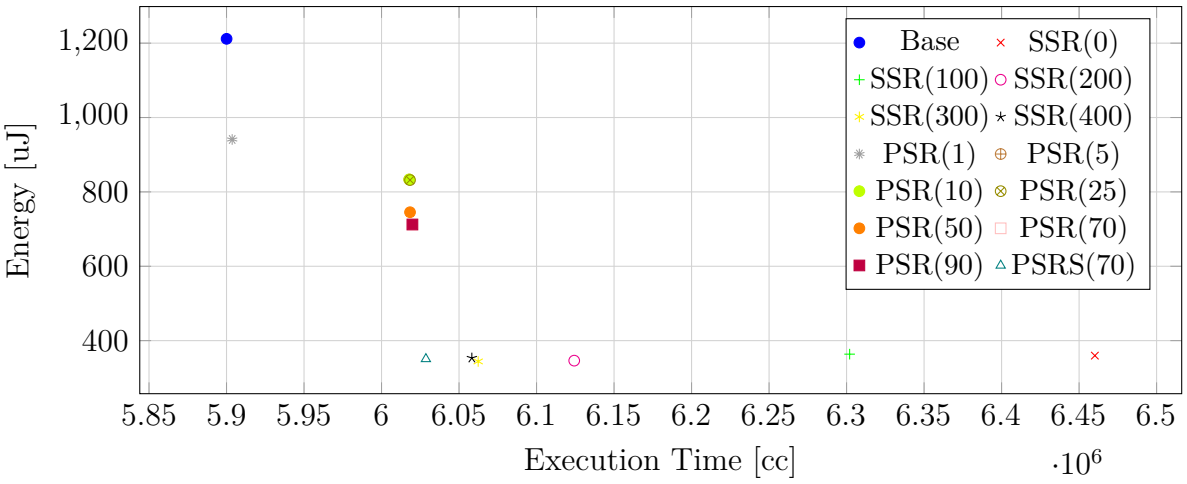
For all power analysis, the open-source DRAM energy estimation tool [11] based on the power model presented in [10] is employed. Current and voltage numbers are obtained from the DRAM vendor's datasheets [40].

However, the analysis is performed with the SystemC model of the CompSOC platform using the SystemC implementation of the predictor, Section 4.1. This SystemC model does not allow for the determination of the power consumption of the predictor itself. These results are obtained in the upcoming Section 5.3 and they validate against the measurements presented in this section.

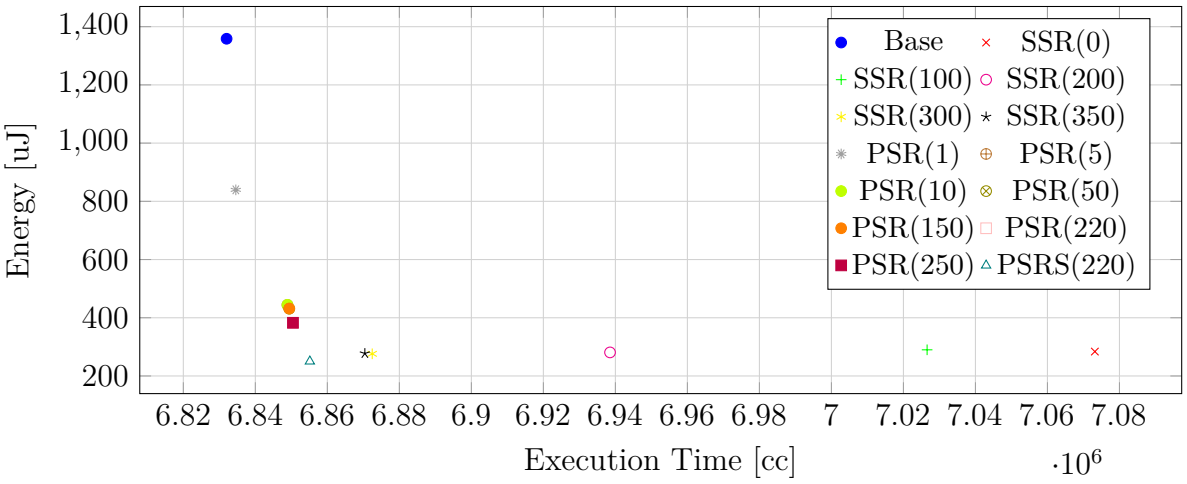
5.2.2. Evaluation of the Proposed Policy

In this section, the impact of (a) the time-out strategy (Section 4.3.1) on the speculative use of the *self-refresh* mode, (b) using multiple predictions for the *self-refresh* mode (Section 4.3.2, and (c) employing speculative the *power-down* mode (Section 4.3.3) for idle cycles not exploited by prediction is evaluated. The Pareto plots in Figure 5.4 show the total memory energy consumption and the performance penalty as total execution time for the media applications. This figure depicts the impact 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. Figure 5.4a presents the results for the H263 decoder, Figure 5.4b for the Ray Tracer, and Figure 5.4c for the JPEG encoder, respectively. Additionally, Table 5.4 explicitly presents the energy reduction and the increased execution time resulting from power-up penalties.

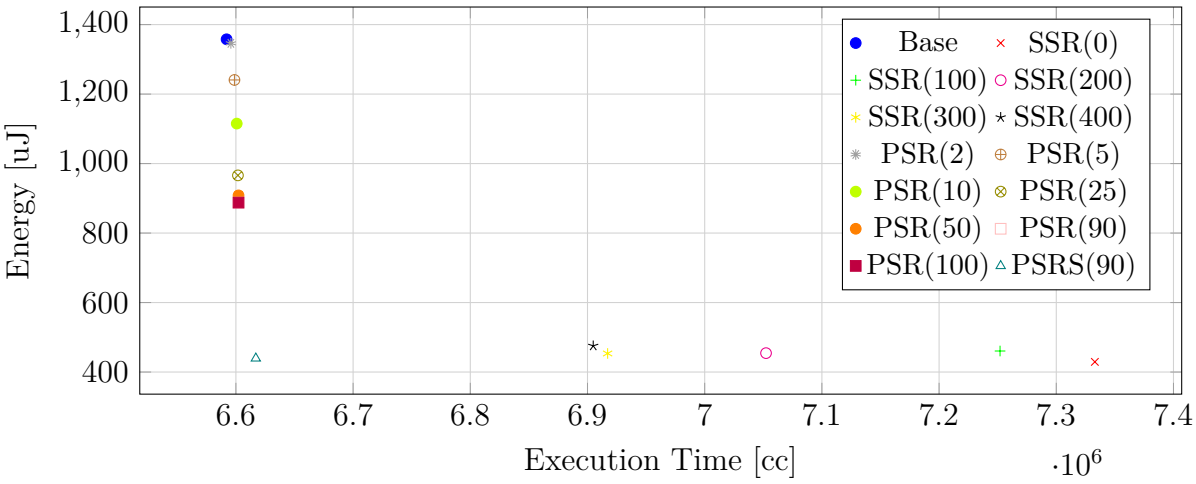
In both Figure 5.4 and Table 5.4, `Base` corresponds to the baseline results when no prediction or power-saving mode is employed. Therefore, `Base` has the lowest execution time (no penalty) and the highest power consumption.



(a) H263 decoder



(b) Ray Tracer



(c) JPEG encoder

Figure 5.4.: Pareto plot of energy and execution time (Highest accuracy)

Time-Out Strategy

In the first experiment, the solution is compared to the speculative use of the *self-refresh* mode when the memory is idle. This is depicted in Figure 5.4 and Table 5.4 by `SSR(time)` (Speculative Self-Refresh), where time gives the time-out threshold that is employed. As can be noticed in for all applications, by increasing the time-out threshold up to a defined limit, the speculative *self-refresh* gives improved energy savings and a lower performance penalty. However, for all applications a time-out of 300 cc results in the best power-savings. Longer time-outs already result in less energy reduction. This confirms the usefulness of using the time-out strategy for these applications. There are many idle periods shorter than the time-out threshold value and they are ignored by the speculative *self-refresh* by increasing the time-out thresholds. For all applications, the use of the time-out strategy with the speculative *self-refresh* reduces the energy consumption between 66.6% and 79.6%. However, the speculative use of *self-refresh* without employing prediction always results in high penalties and an increase in execution time by up to 4.9%. In short, high energy savings are achieved because most of the idle cycles, except those filtered out by the 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, the speculative use of the *self-refresh* mode results in much higher penalties, as can be seen in all figures at data point `SSR(0)`.

Multiple Predictions for Self-Refresh

In the next experiment, the use of multiple predictions per idle period for the employment of the *self-refresh* mode, in combination with the previously mentioned time-out strategy, is evaluated. 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 5.4, 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 5.4, where using the prediction results in only a small increase of the execution time compared to the `Base` mode and also reduces the energy consumption. However, using multiple prediction for the H263 decoder, see Figure 5.4a, results in a sudden increase of execution time between the data points `PSR(1)` and `PSR(5)`. This behavior arises due to an accumulated number of misprediction

between these two points.

For all applications, the number of multiple predictions can be limited to a maximum number of invocations, because the reduction of energy goes into saturation as can be seen in Figure 5.4. There are no more reductions beyond this maximum number of predictor invocations. This limit indicates the number of predictor invocations when the predictor forecasts that it is no longer profitable to continue in *self-refresh*. The best values for the limit parameter for the different applications are shown in Table 5.3. Using the prediction for employing *self-refresh* reduces the energy consumption significantly across the different applications between 34.6% and 71.84%). These savings are lesser than the speculative usage of the *self-refresh* mode, because 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 2.03%. The energy savings and the increase of execution time are shown in Table 5.4.

Multiple Predictions for Self-Refresh and Speculative Power-Down

The next experiment evaluates the proposed power-saving policy, which combines the time-out strategy and the predictions for *self-refresh*. Additionally, the power-saving policy schedules a speculative *power-down* to maximize power savings, but is 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, in turn, is used for predictions of *level 1* and cycles ignored due to the time-out threshold. 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*) in Figure 5.4 and Table 5.4, 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 high energy savings for all applications and it reduces the energy consumption between 67.6% and 81.58%. This policy also results in a low performance penalty between 0.34% and 2.18%. The 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 a considerably lower performance penalties compared to using the *self-refresh* mode speculatively.

Table 5.4.: Power savings and penalty cycles

	H263 decoder		Ray Tracer		JPEG encoder	
	Inc. Execution Time [%]	Savings [%]	Inc. Execution Time [%]	Savings [%]	Inc. Execution Time [%]	Savings [%]
Base	0	0	0	0	0	0
SSR	2.75	71.6	0.59	79.7	4.9	66.6
PSR	2.03	41.2	0.27	71.84	0.15	34.6
PSRS	2.18	71.1	0.34	81.58	0.38	67.6

Complete Design Space Exploration

The whole analysis of the power-saving policy was performed using the sets of parameters with the highest accuracy determined in Section 5.1. A complete design space exploration was done with all sets of parameters depicted in Table 5.1. This was done to verify if the sets of parameters with the highest accuracy results also in the highest possible power-savings. Analyzing the Ray Tracer and the JPEG encoder confirms that the set with the highest accuracy also have the highest savings. However, the H263 decoder application has higher savings with a different configuration. Both parameter sets are depicted in Table 5.5. Compared with the parameter set with the highest accuracy, the set with the highest savings has a lower accuracy of 86.01%. Additionally, Figure 5.5 depicts the results for the H263 decoder with the highest savings. The figure reveals that the analysis done for the set with the highest accuracy is also applicable for the one with the highest savings. Moreover, Table 5.6 compares both sets by giving the results for all previous analyzed power-saving policies.

Table 5.5.: Parameter set for H263 decoder with highest savings

Application	hl	pl	w	time-out [cc]	invocations	Accuracy
H263 decoder (accuracy)	10	2	4	300	70	89.79
H263 decoder (savings)	50	2	6	300	110	86.01

Comparing the speculative *self-refresh* mode `SSR` for both parameter sets shows lower savings and increased execution time for the one with the higher accuracy. However, these results have nothing to do with the accuracy of the predictor, because the predictor is not used for the speculative *self-refresh* mode. The difference between both arise from the different *history lengths* and an adverse implementation of the power-saving policy. The set with the highest accuracy has a history length of 10, while the other one has a length

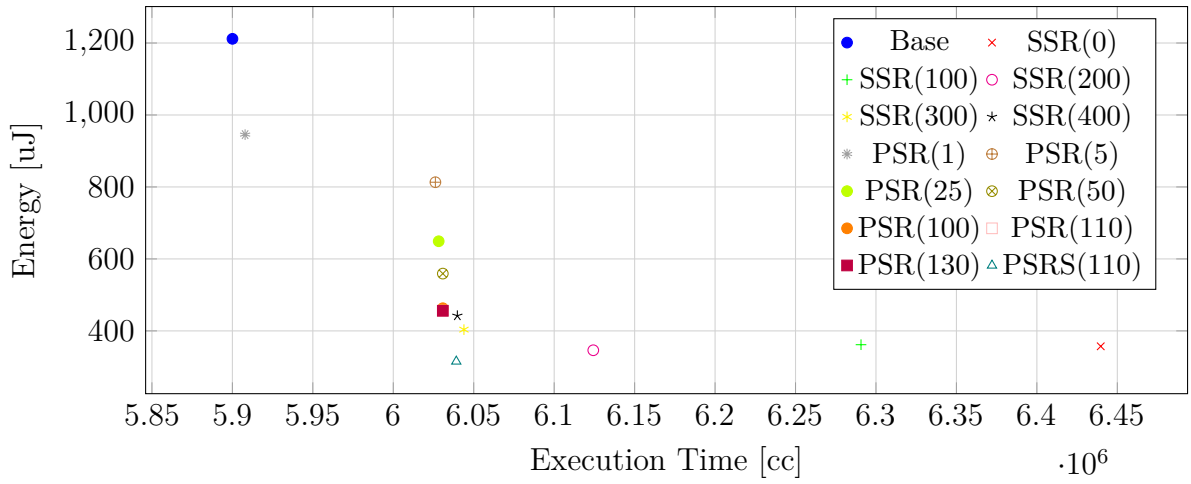


Figure 5.5.: H263 decoder with higher savings parameter set

Table 5.6.: Power savings and penalty cycles

	H263 decoder - highest accuracy		H263 decoder - highest savings	
	Inc. Execution Time [%]	Savings [%]	Inc. Execution Time [%]	Savings [%]
SSR	2.75	71.6	2.44	66.7
PSR	2.03	41.24	2.22	62.38
PSRS	2.18	71.1	2.36	74

of 50, a difference of 40 data points. `SSR` uses an initial time-out of 300 cc. In addition, the module *Power Saving Policy*, depicted in Figure 5.3, is needed for this time-out. `SSR` is applied after the history buffer is full but is neglecting the predictor. As a result, these 40 data points are not used for `SSR` and, therefore, produce lesser savings for the set of parameters with longer history. The main difference between these two sets can be seen in the strategy *prediction for self-refresh* (`PSR`). The set with the lower accuracy causes a slightly higher run-time, as more mispredictions occur. However, the parameter set with the higher savings exploits more idle cycles for the *self-refresh* mode, as can be seen by the number of invocations in Table 5.5. Using an additional speculative *power-down* (`PSRS`) increases the already achieved savings.

5.2.3. Summary of the Power-Saving Policy Analysis

This section provided an analysis of the different power-saving strategies and the proposed predictor based power-saving policy. This policy leverages DRAM idle periods so as to put the DRAM either in the *self-refresh* mode, the *power-down* mode, or a combination

of both 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 the *self-refresh* mode, provided that the predicted idle period length is sufficient to save power. Otherwise, it exploits the idle period by scheduling a speculative *power-down*. For the predictor, a set of parameters that provide the highest accuracy were used. Experimental results for several multimedia benchmarks have shown that this policy 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 67.6% and 71.1%) at a marginal performance penalty (between 0.34% and 2.18%). Moreover, a full design space exploration was done to verify that the sets of parameters with the highest accuracy provides the highest savings. This was proved in two out of three cases. For the third case, a set with lower accuracy was identified to provide higher savings. The accuracies differ only slightly (89.79% vs. 86.01%) and have therefore a low influence on the savings (71.1% vs 74%). The set with the lower accuracy has a slightly higher execution time but exploits more idle cycles due to a higher number of invocations of the predictor. To conclude this section, a set of parameters with high accuracy results in high savings. However, due to multiple predictor invocations, the set with the lower accuracy can provide higher savings. Nevertheless, both sets still have very high savings and differ only by around 3%. Note, that all of the analysis done in this section does not include the power consumption of the predictor itself, which will be determined in the next section.

5.3. Scalability and Power Analysis on Hardware

This section provides the power analysis of the predictor itself for validating the power-savings policy presented in the previous section. First the experiential setup is presented in detail. Afterward, experiments are shown to analyze the scalability of the hardware predictor, due to the variation of the different predictor parameters, which influence the device utilization, speed, and power consumption. Finally, the generic hardware predictor is applied to the proposed power-saving policy so as to analyze the power consumption using the same multimedia benchmarks as in Section 5.2. .

5.3.1. Experimental Setup

The predictor is implemented as a generic and fully synthesizeable VHDL implementation. As described in Section 3.1 and Section 4.4, the predictor has parameters which have a huge influence on the predictor's accuracy and performance.

The parameter *history length (hl)* controls how many past values are stored and, therefore, the number of registers inside the history buffer. Moreover, the maximum unsigned integer number that can be stored in one register defines the size of these registers. This parameter is called the *register size (rs)*. The parameter *pattern length (pl)* controls how many chains of MUXs, Subs and LUTs are needed inside the *Predictor Core* (Figure 4.6) so as to calculate the weight for the past values. This is because the number of chains equals the *pattern length*, as already discussed in Section 4.4. The parameter *width (w)* defines the similarity between the current and past pattern, and it influences the size of the LUTs inside the *Predictor Core* (Figure 4.6). Again, an instance of the predictor with a fixed set of parameters is referred as *predictor configuration*.

To analyze the influence of these four mentioned predictor parameters on the scalability, power consumption and maximum frequency, the design space depicted in Table 5.7 is used. The predictor is set up with a parameter set, synthesized with the help of the Xilinx ISE Design Suite 14.6 [64], as well as placed and routed. As target device a Spartan-6 FPGA [65] is used. After it is placed and routed, the scalability of the design is analyzed. Section 5.3.2 presents the results for the device utilization analysis.

In Section 5.3.3 the power consumption of the predictor is analyzed. For each synthesized *predictor configuration*, a post-place and route simulation model was generated and combined with a test bench that inputs a test-pattern to the predictor. The lowest frequency (50 MHz) achievable for all configurations within the design space and the same 4 bit input sequence are used to achieve comparable results between different con-

Table 5.7.: Parameter configuration for the predictor scalability

Parameter	Range
History length (hl)	10,20,30,40,50
Pattern length (pl)	2,3,4,5
Width (w)	2,4,6,8
Register size (rs)	4,5,6,7,8 bit

figurations. The power consumption is determined by simulating each configuration with Mentor Graphics ModelSim 6.6SE [39]. It is then analyzed afterwards using the XPower Analyzer from the Xilinx ISE Design Suite [64] to determine the static and dynamic power consumption for all of the configurations.

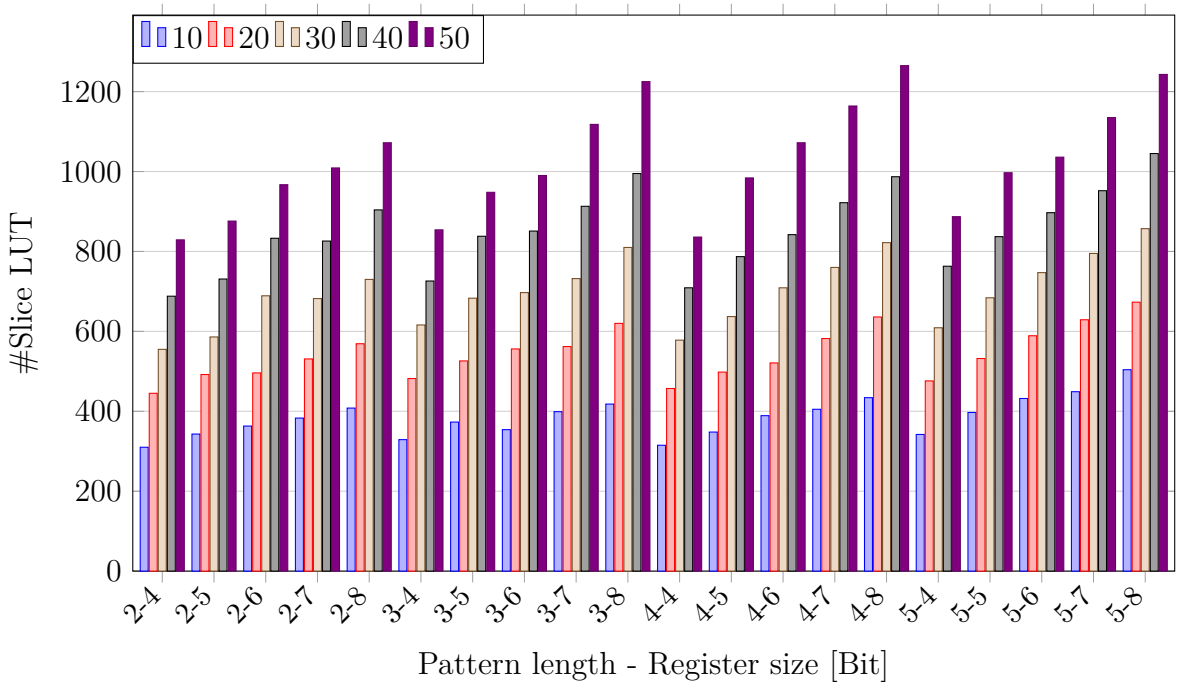
To determine the maximum frequency of each *predictor configuration*, the VHDL model is synthesized with the ISE Design Suite 14.6 [64] that uses, a user constraint file (ucf) which defines the maximum frequency. As long as this synthesis is successful, the constraint is tightened until the synthesis fails, so as to determine the maximum frequency. The frequency is analyzed by increasing the frequency in steps of 10 MHz. The analysis to determine the maximum frequency for a certain predictor configuration is presented in Section 5.3.4.

Finally, in Section 5.3.5 the results are presented by applying the predictor to the same multimedia benchmarks used in the previous section to validate the proposed power-saving policy, including the predictor's power consumption. The determined data from Section 5.2 are used and validated against the power numbers from a predictor configuration running on an FPGA. The power analysis is done as already described.

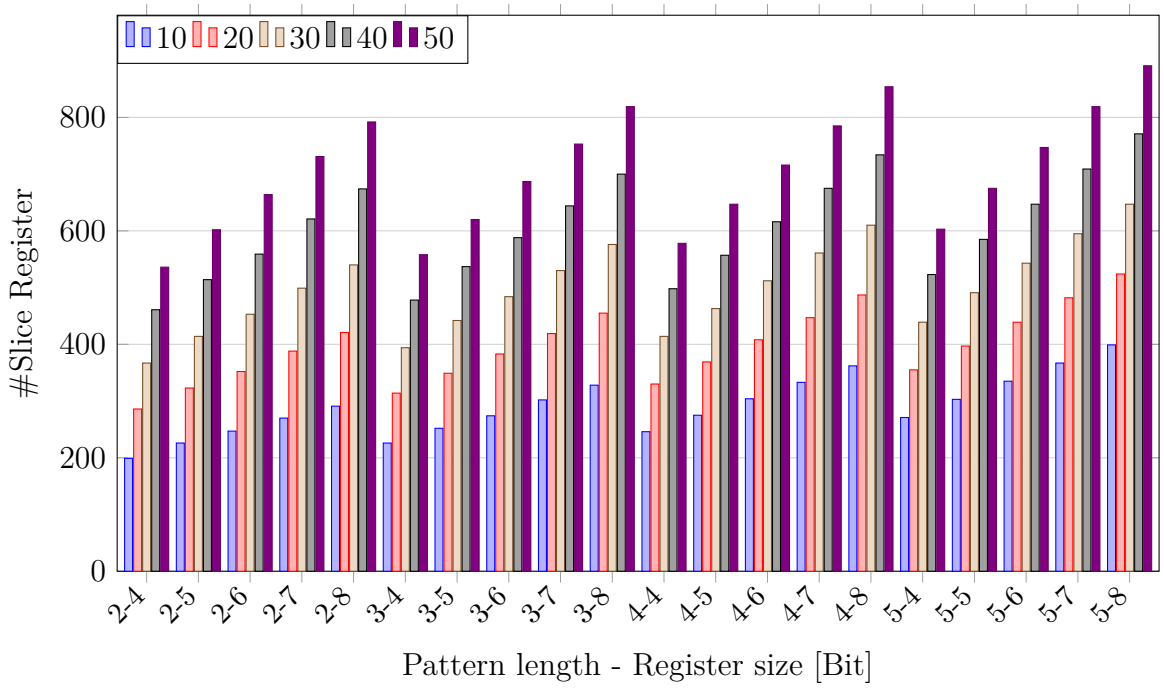
5.3.2. Device Utilization Analysis

The parameter *width* has only a minor influence on the predictor complexity. The *width* defines the similarity between the recent and past pattern and is used inside the *Predictor Core* by the LUTs, see Figure 4.6. Increasing this number influences the complexity of all following components only marginally. As shown in previous Sections 5.1 and 5.2, the highest accuracy was achieved by the *width* of 4 and 6, which gives both numbers a maximum bit size of 3. Higher numbers results in a lower accuracy. All *predictor configurations* are analyzed with a *width* of 4, as this is the same complexity as for a *width* of 6, and higher numbers result in a lower accuracy.

Figure 5.6 depicts the device utilization of all analyzed *predictor configurations* by giving the number of used slice registers in Figure 5.6a as well as the number of slice LUTs in



(a) Number of used slice LUTs



(b) Number of used slice registers

Figure 5.6.: Predictor's device utilization

Figure 5.6a. The x-axis gives the *pattern length* and *register size* (e.g. 2-4 equals a *pattern length* of 2 and each register in the history buffer has 4 bit). Each bar shows the device utilization by depicting the number slice register and the slice LUTs for a certain *history length*.

For a given *pattern length* and *register size*, the number of used slice registers and slice LUTs grows with the increasing *history length*. This reflects the growing history buffer as a longer *history length* requires more registers inside the history buffer to store the past data points. However, the *Predictor Core* and the following computational components does not become larger, as the *history length* has no influence on these components.

The influence of the *register size* on the history buffer can be seen in Figure 5.6. A larger *register size* at a constant *history length* and *pattern length* results in an increasing complexity. The number of slice registers and LUTs increases because not only does the history buffer becomes more complex, but the *Predictor Core* and the computational components does as well, because they have to handle numbers with a larger bit size.

The increase of the *pattern length* at a fixed size *history length* and *register size* cause a growing complexity. The *pattern length* equals the number of chains (MUX, Sub, LUT) inside the *Predictor Core*, as can be seen in Figure 4.6. However, the bit size of the calculated value *weight* increases since more values must be multiplied. This causes an increased complexity of all of the following components, like the accumulators and divider, as depicted in Figure 4.5.

The available slice registers/LUTs of the Spartan-6 FPGA, as well as the range of the used slice registers/LUTs, are shown in Table 5.8. As can be seen, most of the FPGA is not used, as the predictor design is very small and therefore the device utilization is always below 5%.

Table 5.8.: Usage of slice registers/LUTs to implement the predictor on a Spartan-6 FPGA

	Predictor usage	Spartan-6	utilization
#Slice LUT	245 . . . 1278	27288	0.9% . . . 4.68%
#Register	186 . . . 891	54576	0.34% . . . 1.63%

5.3.3. Power Consumption Analysis

To achieve comparable results for the power consumption, all predictor configurations ran with the same frequency and the same test pattern as described in Section 5.3.1. Figure 5.7 depicts the power consumption for different *predictor configurations*. The

upper Figure 5.7a depicts the static power consumption of the Spartan-6 FPGA and the lower Figure 5.7b the dynamic power consumption. Each bar represents a predictor configuration at a certain *history length*, *pattern length* and *register size* in Watts. Again, the x-axis shows the *pattern length* (first number) and the *register size* (second number).

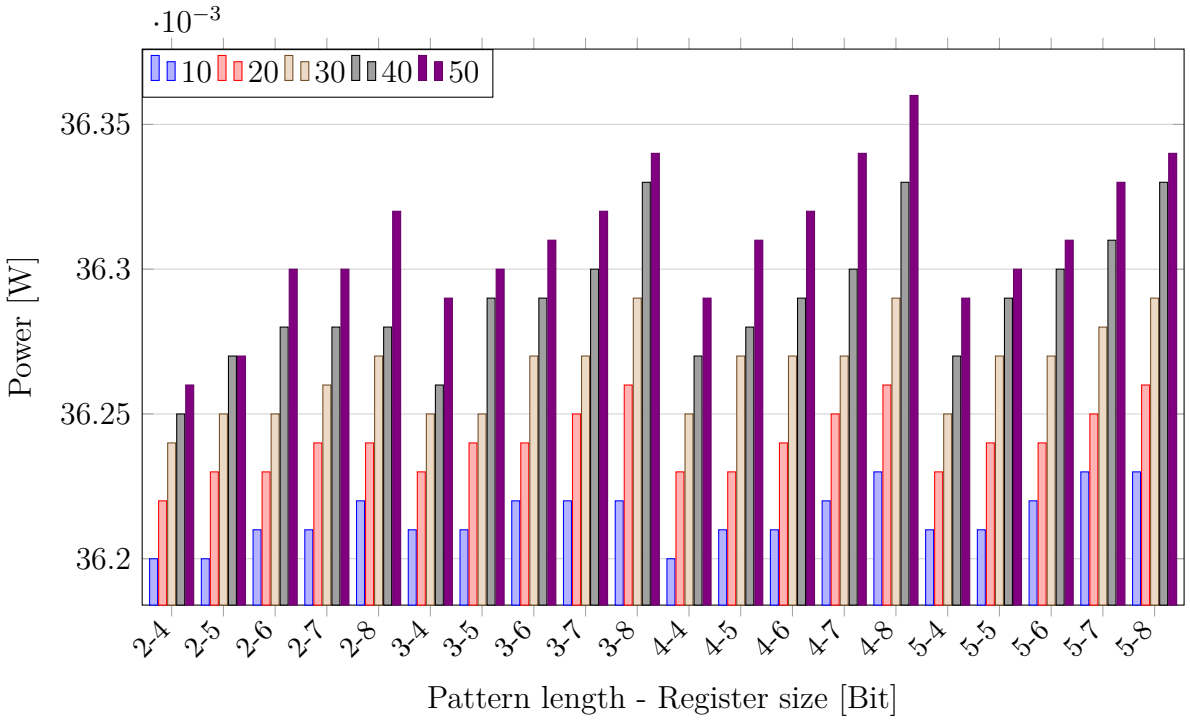
The static and the dynamic power consumption is influenced by the *predictor configuration*. The greater the complexity of the design, like longer *history length*, higher *pattern length*, or a larger *register size*, the higher the power consumption.

Analyzing the static power consumption for each configuration shows that it is almost constant for all sets. The minimum static power is at 36.2 mW for the smallest configuration and the maximum at 36.38 mW for the largest. Minimum and maximum power consumption differs only by 0.18 mW. The reason for the almost constant static power is that the predictor is running on an FPGA. Independent from the size of the predictor design, the basic static power consumption for the Spartan-6 FPGA is at 31 mW [62] since the whole FPGA has to be powered. Moreover, the predictor contributes to the static power by only approximately 5.2 mW for the smallest configuration and by 5.38 mW for the largest. Around 85% of total static power consumption is needed to power the whole FPGA and only the remaining 15% is consumed by the predictor design. This power can be decreased tremendously if a smaller or more power efficient FPGA is used. Moreover, the implementation as an *application-specific integrated circuit* (ASIC) would decrease the power consumption.

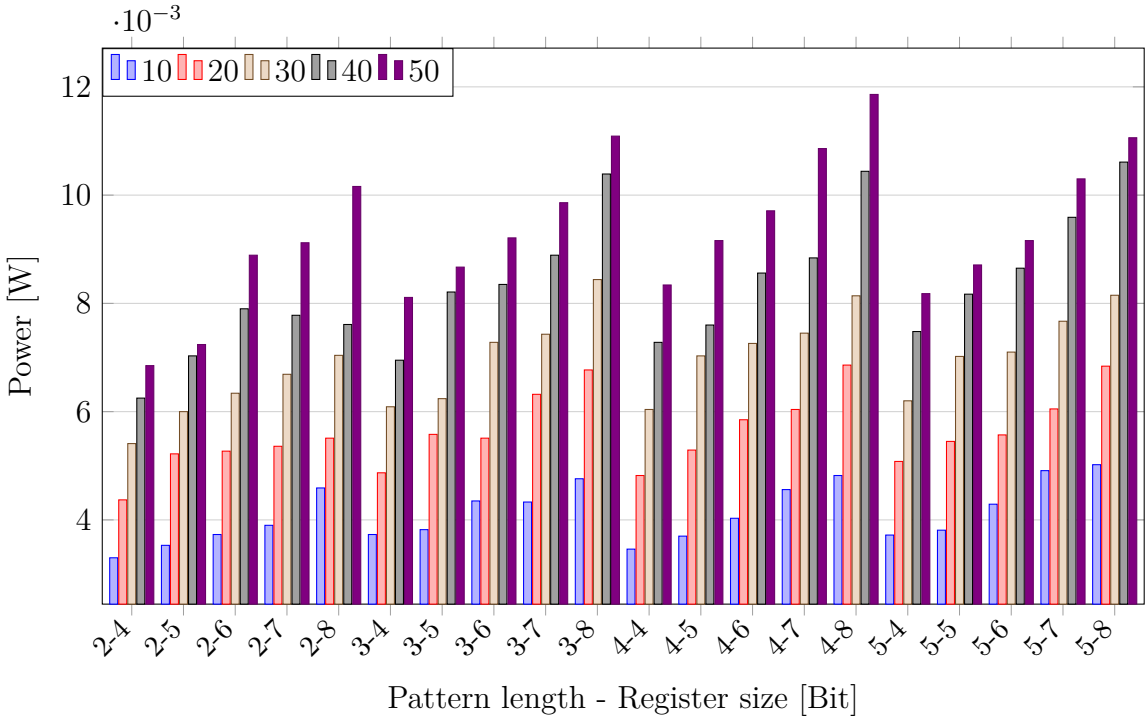
5.3.4. Frequency Analysis

In this section the maximum frequency for different predictor configurations is analyzed. Figure 5.8 depicts the maximum frequency for different *predictor configurations*. Again, the x-axis shows the parameters *pattern length* (first number), and *register size* (second number). Each set of five bars for the given values of fixed *pattern length* and *register size* represent the influence of the parameter *history length*.

Figure 5.8 shows that the parameter *pattern length* has great influence on the maximum frequency. Increasing this parameter results in a decreased maximum frequency. For example with a *pattern length* of 2 the maximum frequency ranges approximately from 165 MHz up to over 200 MHz for all predictor configurations. However, if the *pattern length* is increased to 3 the maximum frequency range is reduced to approximately 130 MHz and 160 MHz. The same reduction can be observed when increasing the *pattern length* to 4 or 5. This is due to the fact, that a higher pattern length will require more components in the *Predictor Core* (see Figure 4.6) to calculate the weight. The critical step is the final



(a) Static power



(b) Dynamic power

Figure 5.7.: Spartan-6 FPGA Power Consumption

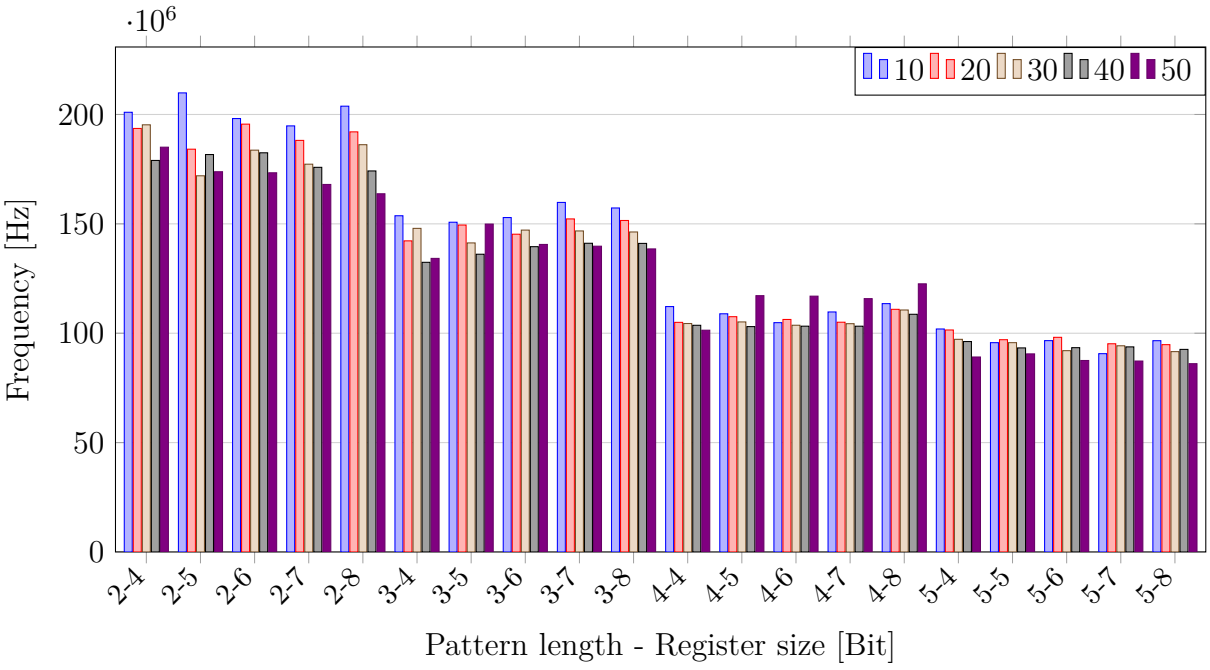


Figure 5.8.: Frequency analysis

multiplication which becomes slower when handling input values with a larger bit size.

The influence of the parameter *history length* on the maximum frequency can be seen by analyzing a configuration set with a constant *pattern length* and *register size*. In most sets of configurations, the frequency decreases slightly with the increased *history length*. However, for some configurations, like 2-5, 4-5 or 4-6, the determined maximum frequency fluctuates. These sets fluctuate by ±10 MHz depending on the initial constraint of the place and route implementation done by the Xilinx ISE Design Suite.

Figure 5.8 also depicts that the influence of the parameter *register size* can be neglected. Changing this parameter results in a frequency variation of ±5 MHz.

5.3.5. Memory Idle Prediction

In Section 5.2, the proposed power-saving policy was analyzed, neglecting the power consumption of the predictor. This section closes that gap by analyzing the impact of the predictor applied to memory idle periods. To validate this power-saving policy, the same three multimedia benchmarks (H263 decoder, Ray Tracer and JPEG encoder) are used. The predictor was set up with the configurations with the highest accuracy determined in Section 5.1 and already applied in Section 5.2. All configurations are summarized in Table 5.9. However, for H263 decoder, two parameter sets are identified: one with the

highest accuracy and another with highest savings. Therefore, both configurations are analyzed and can be distinguished by the abbreviations (ac), which is the parameter set with the highest accuracy and by (sa), which is the parameter set with the highest savings.

Table 5.9.: Parameter set for power analysis

Application	hl	pl	w	rs [bit]	x_{tout} [cc]	x_{pred} [cc]	t_{min} [ns]	f_{min}[MHz]
H263 decoder (ac)	10	2	4	4	300	26	28.8	34.67
H263 decoder (sa)	50	2	6	4	300	106	7.07	141.3
Ray Tracer	30	2	4	4	300	66	11.36	88
JPEG encoder	50	2	6	4	300	106	7.07	141.3

Each predictor configuration is running with the lowest possible frequency, since the frequency is directly proportional to the power consumption. The minimum frequency is determined by Equation (5.1).

$$t_{min} = \frac{x_{tout}}{x_{pred}} \cdot t_{cyc} \quad \Rightarrow \quad f_{min} = \frac{1}{t_{min}} \tag{5.1}$$

The predictor has to deliver the forecast before the time-out runs out, since the *self-refresh* mode is scheduled from that time on, as can be seen in Figure 4.3. This value is given by x_{tout} in clock cycles. This parameter was determined in Section 5.2 and is equal to 300 cc for all applications. The runtime of the predictor in clock cycles, x_{pred}, was derived in Section 4.4.3 and is given by Equation (4.8). The predictor was placed in the front-end of the memory controller, as depicted in Figure 5.3. The memory controller runs with a clock frequency from 400 MHz giving a cycle time of $t_{cyc} = 2.5$ ns. Table 5.9 depicts also the time-out length, predictor runtime as well as the lowest frequency calculated from (5.1).

To determine power numbers for the predictor, the afore mentioned configuration was synthesized as well as placed and routed with the help of the Xilinx ISE Design Suite 14.6 [64]. A VHDL test bench was used to insert the same multimedia benchmarks to the hardware predictor, running at the lowest possible frequency. To analyze the power consumption, the same setup as presented in Section 5.3.1 is used.

Figure 5.9 depicts the results of the power analysis for the three different multimedia benchmarks. The y-axis gives the power consumption, and the x-axis the different benchmarks. The first bar (Base) in each set gives the total power consumption without the predictor for the corresponding benchmark, and is considered as baseline. The second bar

for each benchmark consist of four parts: (1) the reduced power consumption using the power-saving policy (Memory), (2) the dynamic power consumption of the predictor, (3) the static power consumption of the predictor, and (4) the static power consumption of the FPGA. However, as already mentioned in the previous section, the whole FPGA has to be powered even if only a small part is used for the design. For that reason the static power is divided into the power consumed by the predictor and the power consumed by FPGA. The static power consumption of the FPGA is at 31 mW [62]. For that reason the third bar represents the power consumption of the predictor design without the static FPGA power consumption. This bar represents the best possible power consumption in case of an FPGA, which fits the design perfectly without the overhead.

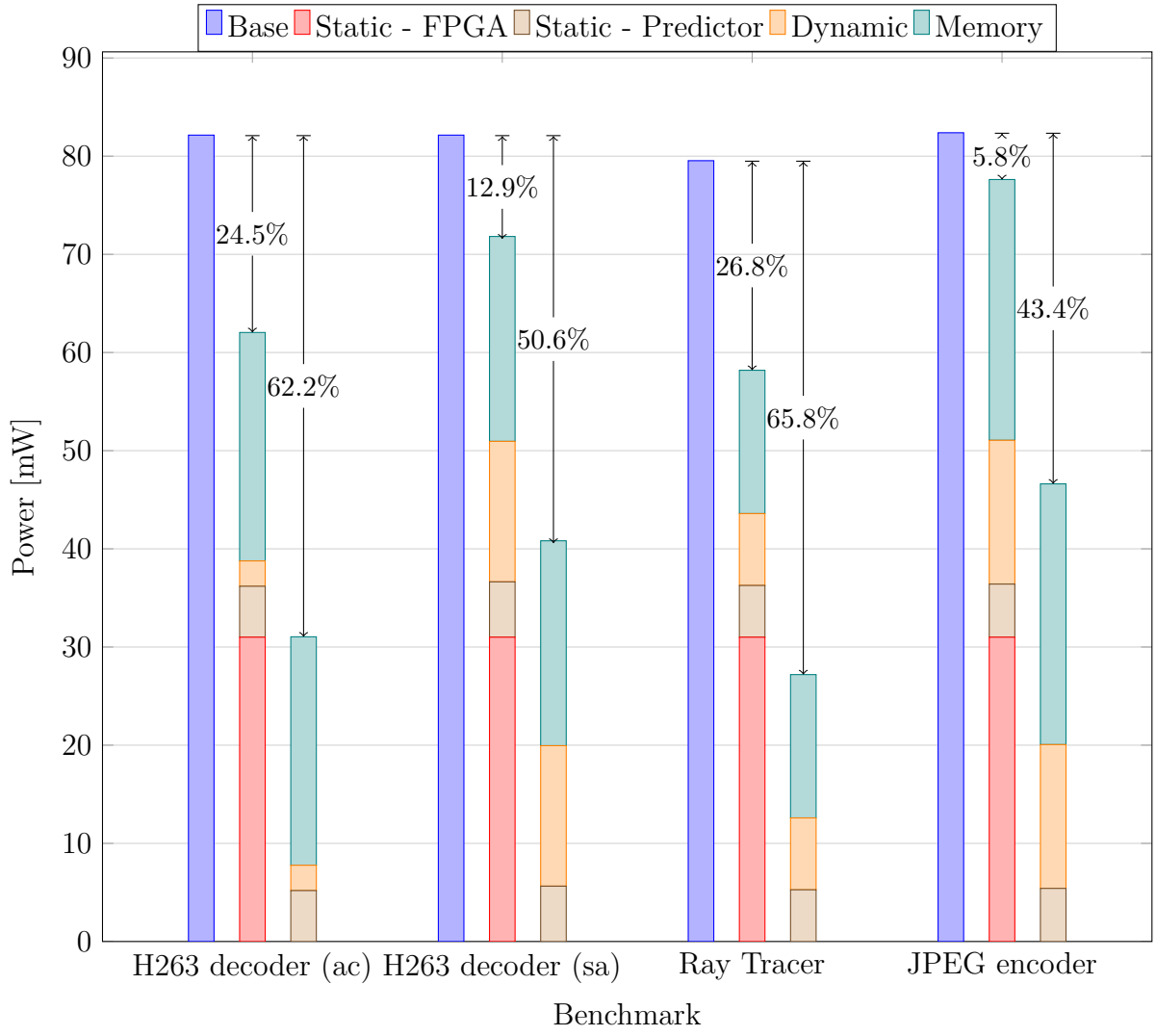


Figure 5.9.: Power consumption analysis for multimedia benchmarks running on the CompSOC platform

Figure 5.9 shows that over 50% of the total power consumption is caused by the static power of the FPGA (second bar). Despite this high static power consumption, the presented power-saving policy produces gainful results, since the power consumption of the memory can be reduced by 5.8% up to 24.5%. However, neglecting the wasted power due to the static power consumption of the FPGA (third bar), the actual saving range between 43.4% and 65.8%.

Comparing both sets of parameters for the H263 decoder reveals that the configuration with the highest accuracy (ac) does not result in the highest power reduction. The parameter set with the highest savings (sa) reduces the power consumption of the memory by a higher amount compared to the set with the highest accuracy. However, this comes at the cost of a more complex hardware design. Additionally, the more complex design needs a higher frequency to meet the timing constraints due to the time-out. Therefore, the static and dynamic power consumption is higher and results in a not beneficial trade-off.

For all analysis, the parameters are set statically. However, a run-time reconfiguration is not implemented but conceivable, in case the change of applications is detectable. To realize the run-time reconfiguration, the history buffer has to be set up with the highest *history length* and *register size*, since the history buffer has to provide capacity for a changing history. The number of data points consider in the *history buffer* is controlled by the counter depicted in Figure 4.5. Changing it is equal to change the *history length*. The *pattern length* is identical for all applications and does not have be changed. The *width* can be changed by loading a new triangular function to the LUTs used inside the *Predictor Core* (Figure 4.6).

5.3.6. Summary of the Scalability and Power Analysis on Hardware

This section has analyzed the hardware predictor in detail. First, the device utilization was determined for an FPGA in terms of slice LUTs and slice registers. This analysis shows that the whole VHDL implementation uses less than 5% of the resources of an FPGA. Moreover, a general power consumption analysis was done to achieve comparable results for different predictor configurations. The static power consumption of the predictor design ranges between 5.2 mW and 5.38 mW. Moreover, as part of this analysis, the high static power consumption of the whole FPGA was identified at 31 mW. A frequency analysis was presented and showed that the predictor's maximum frequency ranges between 86 MHz and 210 MHz and depends on the hardware complexity. Finally, the power-saving policy was evaluated with different multimedia benchmarks. Using the generic hardware predictor and the proposed power-saving policy reduces the memory

power consumption of a DDR3-800 memory between 43.4% and 65.8%, neglecting the static power consumption of the FPGA.

5.4. Analysis of Network Traffic

All results shown in the previous sections were aimed at verifying the predictor in software and hardware. Moreover, the proposed power-saving policy was analyzed and validated. However, this section provides additional results for the predictor using another type of input traffic and analysis at the same time if the predictor is able to forecast several data points ahead. Therefore, the software implementation of the prediction algorithm is used, see Section 4.1, and validate with input data from a real MPI application.

5.4.1. Experimental Setup

To validate the proposed algorithm on network traffic patterns, traffic traces from real applications are needed. To generate these traces, the application Meep [28] is used. Meep is a free *finite-difference time-domain* (FDTD) simulation software package developed at MIT to model electromagnetic systems. The program has the ability to parallelize a problem with the MPI which is used for the communication between processes. For this experiment Meep is used with OpenMPI [19]. The application Meep has been running on a server with 2 processors with 4 cores each.

Every communication is recorded using the MPItrace tool [20]. This tool traces the basic activities in an MPI program and generates a Paraver [4] trace file. This trace file includes information about thread states and communication. For this experiment, only the communication information between MPI processes is relevant. Therefore a short script has been implemented that separates the communication information from all other information. However, a drawback of the MPItrace tool is that the start and end time of every communication corresponds to the time when the MPI functions, send and receive, are called, respectively. This is called the logical communication. It is not possible to determine when the real (physical) communication takes places. How this problem can be solved is described in the following.

The dashed bars in Figure 5.10 depicts the amount of data sent by an arbitrary core over time. The figure shows that the gap between two transferred messages, as well as the amount of data that is sent varies. This leads to a prediction problem with two unknown variables, since the problem is not only to predict how much data is sent but also at which time. There is a large difference between not knowing if a certain data point in time exists or to know that there is a data point whose value may be zero. This means that the time between two communications in the time domain are not equidistant, which introduces problems for the prediction algorithm. The introduced prediction algorithm cannot deal

with this problem because there are too many unknown variables. The algorithm can predict only one unknown variable over an equidistant scale, for example, the amount of data that is sent at a fixed point in time. For that reason the problem must be reduced in order to apply the proposed algorithm.

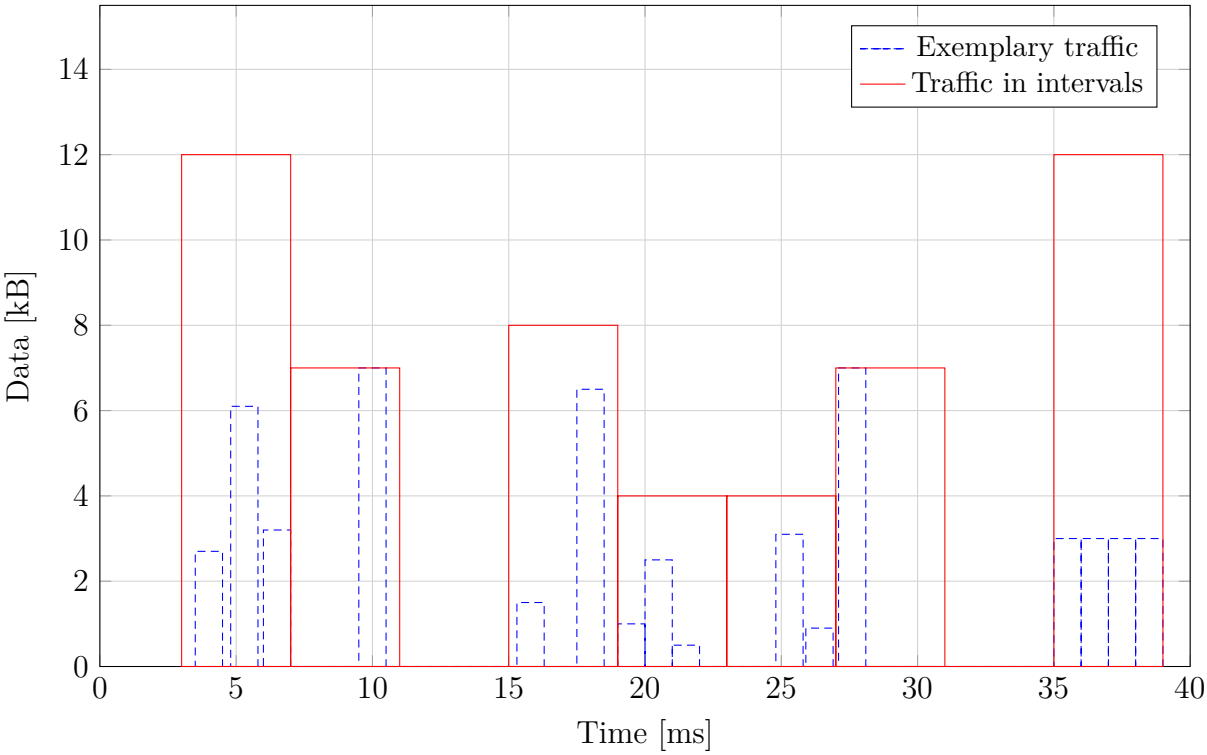


Figure 5.10.: Non-equidistant MPI communication patterns become equidistant

To reduce this problem, time is divided into fixed intervals. In every time interval, the amount of data that is sent is summed up. The solid bars in Figure 5.10 depict the amount of data sent in each time interval. The advantage of time intervals is that the time axis does not have non-equidistant time steps. Therefore the previous mentioned problem with two unknown variables has been reduced to a problem with one unknown, namely to predict the amount of data that will be sent. With this technique the problem of the tracing tool only providing traces of logical communication information is also reduced. It can be assumed that the physical communication will take place shortly after the logical so the assumption is made that most communications will start in the corresponding time interval, provided the size of the interval is not too short. For communications that take place at the end of a time interval, it cannot be determined if they are assigned to the correct time interval. This side effect will be neglected.

The predictor needs an application specific set of parameters to be set up, as shown

in Section 5.1. Therefore, a design space exploration was performed to identify the best set of parameters. Moreover, the Octave implementation of the algorithm is used. This version is not limited to unsigned integers and can handle floating point numbers as input data, output data, and also for the parameter set. The design space exploration identifies a *history length* of 300, a *pattern length* of 7 and a *width* of 5.5 as the best parameter set. To forecast several data points, the predictor calculates the next prospective data point and then uses this predicted point again as an input for the next prediction.

5.4.2. MPI Analysis

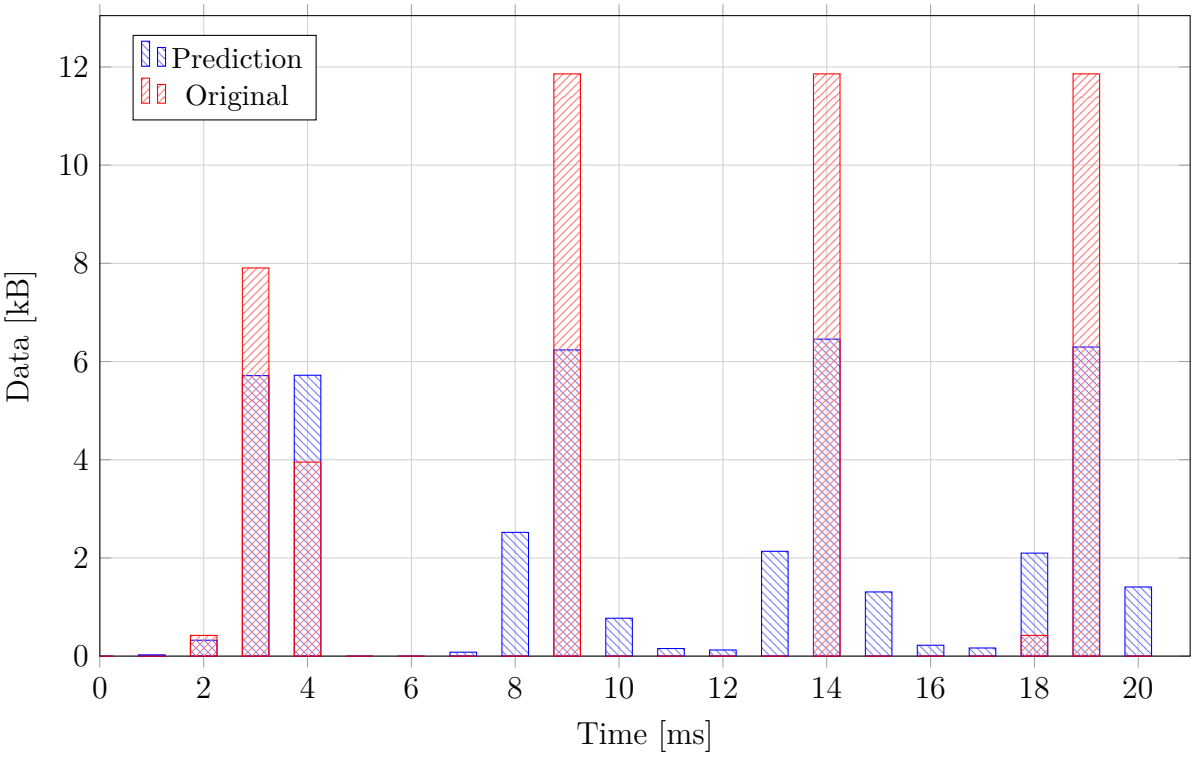
The proposed algorithm should potentially predict up to 20 prospective time intervals. Figure 5.11 depicts two typical behaviors by comparing the real data and the predicted data. The first data point numbered zero in both figures is the real one. Therefore the measured and predicted values match perfectly. The prediction starts in time interval 1. In time intervals where no bars are visible the communication volume during this interval is zero. Both figures show the amount of data sent in each time interval. The dashed bars depict the real data and the solid bars the predicted.

When predicting the MPI communication traces, two behaviors have been observed (depicted in Figure 5.11a and Figure 5.11b). The first behavior shows that the time at which a peak communication takes places is predicted with high accuracy, but the predicted amount of data is below the actual amount. The second behavior is depicted in Figure 5.11b and reveals that the amount of transmitted data is predicted better than in the other case. The algorithm, however, predicts communication peaks where there are no peaks. This can be seen in time interval 12.

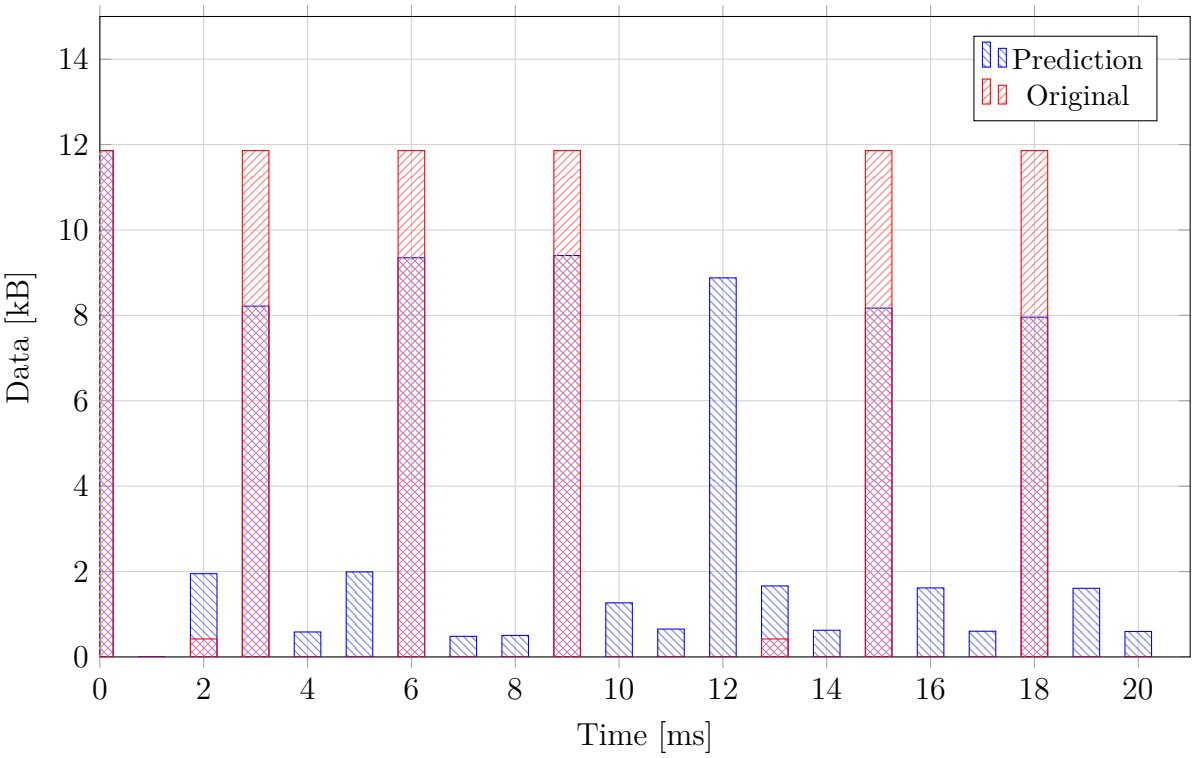
Table 5.10 shows the absolute average error in kB for both figures after several predictions steps. It is not possible to present the relative error because several data points are zero which would lead to a divisor of zero. Table 5.10 shows that a 1-step prediction can be performed with high accuracy for both behaviors. For the first behavior, a 2-step prediction has a small error as well. After that the error increases, but stays nearly constant up to step 10. Thereafter the error increases more and more because of error propagation.

Table 5.10.: Absolute average error in kB

Steps	1	2	5	10	20
Avg. Err. (Fig. 5.11a)	0.008	0.768	1.551	1.498	1.920
Avg. Err. (Fig. 5.11b)	0.246	1.596	2.442	2.427	2.656



(a) First behavior



(b) Second behavior

Figure 5.11.: Prediction of MPI communications

The mispredictions for both shown behaviors arise from the relative distance between two data points in contiguous time intervals, especially when a peak communication is followed by the absence of communication or vice versa. Such jumps mislead the proposed algorithm so that a wrong estimation is produced. The communication patterns from the history are weighted incorrectly, so an error arises which influences coming data points. Data points that occur far into the future are predicted based on previous predictions, and leads to a very fast error propagation.

5.4.3. Summary of the Analysis on Network Traffic

The prediction algorithm was tested on real traffic patterns obtained by tracing an MPI application on a multiprocessor system. Additionally, the algorithm was used to predict several data points based on previous predicted data points. This section demonstrates that the prediction algorithm can be applied to commutation patterns between several cores communicating with MPI. Hence, the algorithm needs an application specific parameter set. The data packages sent over the network ranged between 0 kB and 12 kB. The algorithm accurately predicted the next prospective data point, with a marginal error of maximum 0.246 kB. However, applying the algorithm to predict several data points, two behaviors were identified. The first behavior shows that the communication peaks were predicted at the correct time interval, but with an underestimated amount of sent data. The second behavior shows that the amount of sent data was identified with a higher accuracy, but wrong communication peaks were predicted. It is therefore possible to predict two data points, but in both cases a higher error rate occurs compared to a 1-step prediction. Due to error propagation, the accuracy drops as the number of predicted data points increases. This leads to the conclusion that the prediction algorithm is only applicable to a 1-step prediction.

5.5. Summary

This chapter presented four different types of analysis, all using the proposed predictor. (1) An accuracy analysis to identify the best parameter sets for multimedia benchmarks, which were used in following analysis. (2) An analysis of the proposed predictor-based power-saving policy by providing the maximum possible power reduction as well as the execution time of the multimedia benchmarks running on an MPSoC. (3) A detailed analysis of the predictor's hardware implementation to show the whole impact of the predictor itself. (4) The predictor was applied to network traffic to verify its applicability to other types of application. Additionally, the predictor's capability of forecasting multiple data points based on previously predicted data points is tested.

The accuracy analysis reveals, that the predictor's accuracy strongly depends on the predictor's configuration. Before applying the predictor to forecast the behavior of an application, an exploration of design space exploration to identify the best set of parameters for the *history length*, the *pattern length*, and the *width*. However, this analysis shows that the idleness of the three used multimedia benchmark (H263 decoder, ray tracer and JPEG encoder) can be determined by an accuracy ranging between 86.25% and 96.43%.

The validation of the proposed predictor-based power-saving policy showed, that high energy savings between 67.6% and 71.1% at very marginal performance penalty between 0.34% and 2.18% are possible.

The hardware analysis of predictor presented the device utilization for an FPGA and showed that the VHDL implementation uses less than 5% of the resources. Moreover, a power analysis demonstrated that static power consumption of the FPGA is at 31 mW and that it consumes around six times more power than the design of the predictor, which is at maximum 5.38 mW. Additionally, the maximum frequency of the predictor was identified as ranging between 86 MHz and 210 MHz, depending on the complexity of the predictor. Using the impact of all previous analysis, the proposed power-saving policy was investigated, including the power consumption of the predictor itself. Power savings between 5.8% and 24.5% were achieved, however, these results include the high static power consumption of the FPGA (31 mW). When considering only the static and dynamic power of the design itself, the savings range between 43.4% and 65.8%.

The last experiment demonstrated, that the predictor is also applicable to other fields of application, like communications between the cores of a multiprocessor system that uses MPI. The capability of the predictor to forecast several steps ahead, based on previous predicted steps, was presented. However, the error rate increased with each subsequent

prediction due to error propagation. Therefore, the predictor is only recommended for one step predictions.

6. Conclusions

In this thesis, a prediction algorithm was proposed to optimize system performance in embedded systems. This prediction algorithm is applied to forecast the length of idle cycles of DRAMs in order to reduce memory power consumption. The results are used to put system memory in a power-saving mode and powering it up before the next request arrives to avoid, or at least, reduce performance penalties.

Therefore, this thesis presented two software implementations of the predictor. One was written in Octave for initial analysis, while the other was coded in SystemC for integration within an MPSoC. The Octave version was used for design space exploration, determining the predictor's accuracy, which is influenced different parameters. Three multimedia applications, an H263 decoder, a Ray Tracer, and a JPEG encoder, were used for testing the approach. The behavior of these applications while accessing memory was recorded to determine memory idleness, and was used as input for the predictor. The accuracy analysis showed that the predictor needs a special set of parameters for each application. In other words, before applying the predictor to forecast the application's behavior, an exploration of design space is needed to determine the setup of the predictor. Afterwards, the predictor is able to forecast the length of the application's idleness with an accuracy rate ranging between 86.25% and 96.43%.

Based on this predictor, the thesis presented a power-saving policy to reduce idle memory power consumption. DRAM supports two power-saving modes, namely *power-down* and *self-refresh*, which were used for the power-saving policy. *Self-refresh* has high power-savings but also a high wake-up penalty. Once the memory is in *self-refresh*, it needs a certain number of clock cycles to power-up before handling the next request. On the other hand, the *power-down* mode achieves lower savings, but comes with a shorter wake-up penalty. The proposed power-saving policy leverages DRAM idle periods to put the DRAM either in the *self-refresh* or the *power-down* mode, or a combination of both in a given idle period depending on its predicted duration. The goal of this power-saving policy is to use *self-refresh* as often as possible to maximize power reduction and therefore uses this mode every time the predictor forecasts that *self-refresh* is efficient. Idle cycles that

are not exploited by *self-refresh* are exploited by scheduling a speculative *power-down*.

The SystemC version of the predictor was integrated into an MPSoC called CompSOC to apply the proposed power-saving policy. Therefore, the results of the accuracy analysis were used to set up the predictor to provide high accuracy. The same multimedia benchmarks were used to show the effectiveness of the power-saving policy. The analysis presented the energy consumption, as well as the total execution time of each multimedia benchmark to show the savings and the performance penalty. The proposed policy achieves significant energy savings, ranging between 81.58% and 71.1% at marginal performance penalties between 0.34% and 2.18%.

To measure the power consumption of the predictor itself, this thesis presents an RTL implementation of the predictor in VHDL. This implementation is fully synthesizeable and at the same time generic, because the algorithm depends on application specific parameters. The data path and the control unit were introduced and explained in detail. The thesis presented a full complexity analysis of the hardware predictor that included the device utilization, the maximum frequency, and the power consumption. The device utilization on a Xilinx Spartan-6 FPGA was less than 5% in terms of slice LUTs and slice registers, respectively. The power analysis examined the dynamic and static power. Depending on the predictor's configuration, the static power of the design ranges between 5.2 mW and 5.38 mW. However, the FPGA's high static power consumption of 31 mW is added, although only 5% of the device is used. Additionally, a frequency analysis was done, and this showed that the predictor's maximum frequency depends on the predictor's configuration, ranging between 86 MHz and 210 MHz. Using the impact of the hardware analysis, the power-saving policy was evaluated, including the power consumption of the predictor itself. Power savings for the multimedia benchmarks ranged between 5.8% and 24.5%. Even considering only the power consumption of the design, and neglecting the high static power consumption of the FPGA at 31 mW, the power savings were ranging between 43.4% and 65.8%.

Applying the predictor to memory idleness in order to reduce power consumption is the main focus of this thesis. However, the prediction algorithm was also tested on real traffic patterns, which are obtained by tracing a MPI application on a multiprocessor system. This experiment showed that the prediction algorithm is also applicable to other fields of application, for example to forecast the amount of data sent in a time interval between processor cores. Moreover, using this input data, the predictor was used to forecast several data points ahead. These predictions were based on already predicted data points. This experiment showed, that the predictor can be applied to forecast two data points ahead,

unfortunately with a high error rate. Furthermore, predicting more than two data points leads to an increasing error rate due to error propagation. Hence, the prediction algorithm is only applicable to forecast one step ahead.

A. Glossary

Acronyms

ANN Artificial Neural Network

ASIC Application-specific Integrated Circuit

ATM Asynchronous Transfer Mode

BTB Branch Target Buffer

DRAM Dynamic Random-access Memory

DVFS Dynamic Voltage And Frequency Scaling

FDTD Finite-difference Time-domain

FIFO First In First Out

FPGA Field Programmable Gate Array

FSM Finite-state Machine

ILP Instruction Level Parallelism

LUT Lookup Table

MLP Multilayer Perceptron

MPC Model Predictive Control

MPI Message Passing Interface

MPSoC Multiprocessor System-on-Chip

MUX Multiplexer

NoC Network On Chip

RTL Register-transfer Level

SRT Self-refresh Threshold

Sub Subtractor

VHDL Very High Speed Integrated Circuit Hardware Description Language

Bibliography

[1] Accellera Systems Initiative. SystemC. Online, December 2013. URL `http://www.accellera.org/home/`.

[2] Benny Akesson and Kees Goossens. *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*. Springer, 2012 edition, 9 2011. ISBN 9781441982063.

[3] Manu Awasthi, David W. Nellans, Rajeev Balasubramonian, and Al 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.

[4] *Paraver - Parallel Program Visualization and Analysis tool*. Barcelona Supercomputing Center - Centro Nacional de Supercomputación, version 3.1 edition, October 2001. URL `http://www.bsc.es/plantillaA.php?cat_id=493`.

[5] L.G. Bleris, P.D. Vouzis, M.G. Arnold, and M.V. Kothare. A co-processor FPGA platform for the implementation of real-time model predictive control. In *American Control Conference, 2006*, pages 6 pp.–, June 2006. doi: 10.1109/ACC.2006.1656499.

[6] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH Comput. Archit. News*, 25:13–25, June 1997. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/268806.268810. URL `http://doi.acm.org/10.1145/268806.268810`.

[7] Oswaldo Cadenas, Graham Megson, and Daniel Jones. A New Organization for a Perceptron-Based Branch Predictor and Its FPGA Implementation. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design*, ISVLSI '05, pages 305–306, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2365-X. doi: 10.1109/ISVLSI.2005.11. URL `http://dx.doi.org/10.1109/ISVLSI.2005.11`.

[8] B. Calder, G. Reinman, and D.M. Tullsen. Selective value prediction. In *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, pages 64–74, 1999. doi: 10.1109/ISCA.1999.765940.

[9] Ramazan Can, Frank Koch, Insu Choi, Insuk Suh, Huichung Byun, and Peyman Blumstengel. Save power and improve efficiency in virtualized environment of datacenter by right choice of memory. Technical report, Microsoft Technology Center & Samsung Semiconductor, 2011.

[10] 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. doi: 10.1109/DSD.2011.17.

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

[12] Bor-Sen Chen, Yu-Shuang Yang, Bore-Kuen Lee, and Tsern-Huei Lee. Fuzzy Adaptive Predictive Flow Control of ATM Network traffic. *Fuzzy Systems, IEEE Transactions on*, 11(4):568 – 581, 2003. ISSN 1063-6706. doi: 10.1109/TFUZZ.2003.814860.

[13] K.C. Claffy, H.-W. Braun, and G.C. Polyzos. A parameterizable methodology for Internet traffic flow profiling. *Selected Areas in Communications, IEEE Journal on*, 13(8):1481–1494, 1995. ISSN 0733-8716. doi: 10.1109/49.464717.

[14] 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. doi: 10.1109/12.966492.

[15] K.M. Deliparaschos, F.I. Nenedakis, and S.G. Tzafestas. Design and Implementation of a Fast Digital Fuzzy Logic Controller Using FPGA Technology. *Journal of Intelligent and Robotic Systems*, 2006. doi: 10.1007/s10846-005-9016-2. URL `http://dx.doi.org/10.1007/s10846-005-9016-2`.

[16] John W. Eaton, David Bateman, and Soren Hauberg. *Gnu Octave Version 3.0.1 Manual: A High-Level Interactive Language For Numerical Computations*. CreateSpace Independent Publishing Platform, 1 edition, 3 2009. ISBN 9781441413000.

[17] Eindhoven University of Technology and the Delft University of Technology. CompSOC - Homepage. Online, November 2012. URL `http://www.compsoc.eu/`.

[18] Tien fu Chen and Jean loup Baer. Effective Hardware-based Data Prefetching for High-performance Processors. *IEEE Transactions on Computers*, 44:609–623, 1995.

[19] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[20] Harald Servat Gelabert and Germán Llort Sánchez. MPItrace - User Guide Manual, 2010. URL `http://www.bsc.es/plantillaA.php?cat_id=492`.

[21] K. Goossens and A. Hansson. The aethereal network on chip after ten years: Goals, evolution, lessons, and future. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 306–311, 2010.

[22] Andreas Hansson. *A Composable and Predictable On-Chip Interconnect*. PhD thesis, Technische Universiteit Eindhoven, 2009.

[23] Andreas Hansson and Kees Goossens. *On-Chip Interconnect with aelite: Composable and Predictable Systems (Embedded Systems)*. Springer, 2011 edition, 10 2010. ISBN 9781441964960.

[24] Y.S.C. Huang, K.C.-K. Chou, Chung-Ta King, and Shau-Yin Tseng. NTPT: On the End-to-End Traffic Prediction in the On-Chip Networks. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 449 –452, 2010.

[25] Gianluca Iannaccone, Christophe Diot, Ian Graham, and Nick McKeown. Monitoring Very High Speed Links. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, IMW '01, pages 267–271, New York, NY, USA, 2001. ACM. ISBN 1-58113-435-5. doi: 10.1145/505202.505235. URL `http://doi.acm.org/10.1145/505202.505235`.

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

[27] John W. Eaton. GNU Octave. Online, December 2013. URL `http://www.gnu.org/software/octave/`.

[28] Steven G. Johnson, John D. Joannopoulos, and Marin Soljačić. Meep, 2006. URL `http://ab-initio.mit.edu/wiki/index.php/Meep`.

[29] Yongsoo Joo, Yongseok Choi, and Hojun Shim. Energy exploration and reduction of SDRAM memory systems. In *Proc. 39th Design Automation Conf*, pages 892–897, 2002. doi: 10.1109/DAC.2002.1012748.

[30] S. Kaxiras and C. Young. Coherence communication prediction in shared-memory multiprocessors . In *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, pages 156 –167, 2000. doi: 10.1109/HPCA.2000.824347.

[31] B.F. Wu K.V. Ling and J.M. Maciejowski. Embedded Model Predictive Control (MPC) using a FPGA. In *Proceedings of the 17th World Congress. The International Federation of Automatic Control (IFAC)*, Seoul, Korea, 2008.

[32] Chunho Lee, Miodrag Potkonjak, and William 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. ISBN 0-8186-7977-8. URL `http://dl.acm.org/citation.cfm?id=266800.266832`.

[33] J.K.F. Lee and A.J. Smith. Branch Prediction Strategies and Branch Target Buffer Design. *Computer*, 17(1):6–22, Jan 1984. ISSN 0018-9162. doi: 10.1109/MC.1984.1658927.

[34] Junghee Lee, Chanik Park, and Soonhoi 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. doi: 10.1109/ASPDAC.2003.1194988.

[35] L. Libman and A.. Orda. Optimal retrial and timeout strategies for accessing network resources. *Networking, IEEE/ACM Transactions on*, 10(4):551–564, 2002. ISSN 1063-6692. doi: 10.1109/TNET.2002.801412.

[36] Chiyuan Ma and Shuming 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. doi: 10.1109/DSD.2007.4341475.

[37] T. Meindl, W. Moniaci, D. Gallesio, and E. Pasero. Embedded hardware architecture for statistical rain forecast. In *Research in Microelectronics and Electronics, 2005 PhD*, volume 1, pages 133–136 vol.1, July 2005. doi: 10.1109/RME.2005.1543011.

[38] A. Mellit, H. Mekki, A. Messai, and S.A. Kalogirou. FPGA-based implementation of intelligent predictor for global solar irradiation, Part I: Theory and simulation. *Expert Systems with Applications*, 2011. ISSN 0957-4174. doi: http://dx.doi.org/10.1016/j.eswa.2010.08.057. URL `http://www.sciencedirect.com/science/article/pii/S0957417410008535`.

[39] Mentor Graphics. ModelSim, 11 2013. URL `http://www.mentor.com/products/fpga/simulation/modelsim`.

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

[41] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N. Patt. Predicting Performance Impact of DVFS for Realistic Memory Systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 155–165, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4924-8. doi: 10.1109/MICRO.2012.23. URL `http://dx.doi.org/10.1109/MICRO.2012.23`.

[42] Lothar Miller. Division in VHDL, February 2009. URL `http://www.lothar-miller.de/s9y/archives/29-Division-in-VHDL.html`.

[43] L. Minas and B. Ellison. *Energy Efficiency for Information Technology: How to Reduce Power Consumption in Servers and Data Centers*. Intel Press, 2009. ISBN 9781934053201.

[44] Umit Y. Ogras and Radu 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. ISBN 1-59593-381-6. doi: http://doi.acm.org/10.1145/1146909.1147123. URL `http://doi.acm.org/10.1145/1146909.1147123`.

[45] Peter Otto and Tobias Schunk. Fuzzybasierte Zeitreihenvorhersage. *Automatisierungstechnik*, 48:327–334, 2000. In: German.

[46] Qixiang Pang, Shiduan Cheng, and Peng Zhang. Adaptive fuzzy traffic predictor and its applications in ATM networks. In *Communications, 1998. ICC 98. Conference*

Record.1998 IEEE International Conference on, volume 3, pages 1759 –1763 vol.3, June 1998. doi: 10.1109/ICC.1998.683131.

[47] Seong-Il Park and In-Cheol Park. History-based memory mode prediction for improving memory performance. In *Proc. Int. Symp. Circuits and Systems ISCAS '03*, volume 5, 2003. doi: 10.1109/ISCAS.2003.1206226.

[48] Emanuel Parzen. On Estimation of a Probability Density Function and Mode. *The Annals of Mathematical Statistics*, 33(3):pp. 1065–1076, 1962. ISSN 00034851. URL `http://www.jstor.org/stable/2237880`.

[49] Massoud 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. ISBN 0-7803-6634-4. doi: http://doi.acm.org/10.1145/370155.370333. URL `http://doi.acm.org/10.1145/370155.370333`.

[50] Benny Åkesson and Kees 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.

[51] Benny Åkesson, Anca Molnos, Andreas Hansson, Jude Ambrose Angelo, and Kees Goossens. Composability and Predictability for Independent Application Development, Verification, and Execution. In Michael Hübner and Jürgen Becker, editors, *Multiprocessor System-on-Chip — Hardware Design and Tool Integration*, chapter 2. Springer, December 2010. ISBN 978-1-4419-6459-5.

[52] Glenn Reinman and Brad Calder. Predictive Techniques for Aggressive Load Speculation. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 127–137, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press. ISBN 1-58113-016-3. URL `http://dl.acm.org/citation.cfm?id=290940.290969`.

[53] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, ISCA '81, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press. URL `http://portal.acm.org/citation.cfm?id=800052.801871`.

[54] V.V. Stankovic and N.Z. 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. doi: 10.1109/TELSKS.2005.1572183.

[55] Gervin Thomas, Ben Juurlink, and Dietmar 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.

[56] Gervin Thomas, Karthik Chandrasekar, Benny Akesson, Ben Juurlink, and Kees Goossens. A Predictor-Based Power-Saving Policy for DRAM Memories. In *15th Euromicro Conference on Digital System Design (DSD)*, 2012. doi: 10.1109/DSD.2012.11.

[57] Gervin Thomas, Ahmed Elhossini, and Ben Juurlink. A Generic Implementation of a Quantified Predictor for FPGAs. In *GLSVLSI '14: Proceedings of the 24th ACM International Conference on Great Lakes Symposium on VLSI*, Houston, Texas, USA, May 2014. ACM.

[58] Jelena Trajkovic, Alexander V. Veidenbaum, and Arun Kejariwal. Improving SDRAM access energy efficiency for low-power embedded systems. *ACM Trans. Embed. Comput. Syst.*, 7:24:1–24:21, May 2008. ISSN 1539-9087. doi: http://doi.acm.org/10.1145/1347375.1347377. URL `http://doi.acm.org/10.1145/1347375.1347377`.

[59] J.W. van den Brand, C. Ciordas, K. Goossens, and T. Basten. Congestion-Controlled Best-Effort Communication for Networks-on-Chip. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1 –6, 2007. doi: 10.1109/DATE.2007.364415.

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

[61] Kai Wang and Manoj Franklin. Highly Accurate Data Value Prediction Using Hybrid Predictors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 281–290, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7977-8. URL `http://dl.acm.org/citation.cfm?id=266800.266827`.

[62] Xilinx. *Xilinx Power Estimator 14.3*, October 2012. URL `http://www.xilinx.com/products/design_tools/logic_design/xpe.htm`.

[63] *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices*. Xilinx, 14.3 edition, October 2012. URL `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/xst_v6s6.pdf`.

[64] Xilinx. ISE Design Suite 14.6, 11 2013. URL `http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm`.

[65] Xilinx. Spartan-6 FPGA Family, 11 2013. URL `http://www.xilinx.com/products/silicon-devices/fpga/spartan-6/`.

[66] Ying Xu, Aabhas S. Agarwal, and Brian 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. ISBN 978-3-642-03137-3. doi: http://dx.doi.org/10.1007/978-3-642-03138-0_14. URL `http://dx.doi.org/10.1007/978-3-642-03138-0_14`.

[67] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 451–461, New York, NY, USA, 1998. ACM. ISBN 1-58113-058-9. doi: http://doi.acm.org/10.1145/285930.286004. URL `http://doi.acm.org/10.1145/285930.286004`.

Acknowledgment

I would like to thank some people, which helped me a lot to finish my thesis.

Thomas Flik already worked for over 35 years at the university when I started as teaching assistant. Starting as colleague, he became a good friend of mine very quickly. We shared the workload of one of the most time consuming lectures and so I was able to work in parallel on my research topic. Without him, I would have been lost between teaching and administrative work and never succeeded in finishing this thesis. Unfortunately, Thomas died in December 2013. Thank you Thomas, I owe you a lot. Rest in peace.

I was lucky to spend some time at the Technische Universiteit Eindhoven in the Electronic Systems group during my time as PhD student. I worked together with Karthik Chandrasekar supervised by Benny Akesson and Kees Goossens. It was a great time and we published a joint paper, which boosted my thesis a lot. Thank you all and especially to Karthik, I really enjoyed the time and learned a lot from all of you.

Ahmed Elhossini joined the group when I was already in the late stage of my thesis. We had some overlap in research, so I ask him if we can collaborate and he agreed. We published a paper together and he became not only my supervisor for the last stage of my thesis but also a friend. During this time, I was twice at the point to leave the group without PhD, however he always encouraged me to go one. Thank you Ahmed for encouraging me and please remember, I do not want to put pressure on you.

I like to thank Ben Juurlink and Dietmar Tutsch who supervised me and were part of my PhD committee. I learned a lot from you and I try to apply it for the next step in my life. I am also very grateful to Carsten Gremzow who was part of my PhD committee. You always had an open ear to discuss problems and helped me a lot. I also thank Stephan Kreutzer who was chairman of my PhD committee.

Special thanks to Angela Pohl, for helping me within the last days before I submitted my thesis.