Gervin Thomas, Ahmed Elhossini, Ben Juurlink

A generic implementation of a quantified predictor on FPGAs

Suggested Citation
Thomas, G.; Elhossini, A.; Juurlink, B.: A generic implementation of a quantified predictor on FPGAs. - In: GLSVLSI '14 Proceedings of the 24th edition of the great lakes symposium on VLSI. - New York, NY: ACM, 2014. - ISBN: 978-1-4503-2816-6. - pp. 255-260. DOI: 10.1145/2591513.2591517. (Postprint version is cited. Page number differs.)

A Generic Implementation of a Quantified Predictor on FPGAs

Gervin Thomas
Technische Universität Berlin
Embedded Systems Architecture
Berlin, Germany
gervin.thomas@tu-berlin.de

Ahmed Elhossini
Technische Universität Berlin
Embedded Systems Architecture
Berlin, Germany
ahmed.elhossini@tu-berlin.de

Ben Juurlink
Technische Universität Berlin
Embedded Systems Architecture
Berlin, Germany
b.juurlink@tu-berlin.de

ABSTRACT

Predictors are used in many fields of computer architectures to enhance performance. With good estimations of future system behavior, policies can be developed to improve system performance or reduce power consumption. These policies become more effective if the predictors are implemented in hardware and can provide quantified forecasts and not only binary ones.

In this paper, we present and evaluate a generic predictor implemented in VHDL running on an FPGA which produces quantified forecasts. Moreover, a complete scalability analysis is presented which shows that our implementation has a maximum device utilization of less than 5%. Furthermore, we analyze the power consumption of the predictor running on an FPGA. Additionally, we show that this implementation can be clocked by over 210 MHz. Finally, we evaluate a power-saving policy based on our hardware predictor. Based on predicted idle periods, this power-saving policy uses power-saving modes and is able to reduce memory power consumption by 14.3%.

Categories and Subject Descriptors

B.0 [**Hardware**]: General

1. INTRODUCTION

Right now, we are living in a time where the complexity of computer systems and embedded systems increase very fast, since more computational components are integrated on smaller and smaller chips. This high complexity allows us to develop embedded systems which have high computational power in small hand-held devices like tablets and smartphones. However, with this high degree of complexity, problems like a high amount of data transfer between components inside these embedded systems or the high power consumption which drains the battery of these portable devices arise. Some of these problems can be reduced or solved with a good estimation of the prospective behavior of such components. This is the point where predictors come into the focus. The well known predictors in the field of computer engineering are usually simple branch predictors which decide if a branch is taken or not. This is a binary decision perfectly suitable for branch predictors. However, if the problem becomes more intricate and a binary decision is not sufficient, more complex predictors are required which can quantify forecasts. These types of predictors allow developing of strategies and policies that can increase system performance or reduce power consumption. For example, in [14] a power-saving policy was presented that reduces the *dynamic random-access memory* (DRAM) power consumption tremendously. The whole power-saving policy is based on a predictor which forecasts the length of the memory idle periods. Using this idle period length, a memory power-saving mode is applied to reduce the memory power consumption and wake up the memory before the next request arrives to avoid wake-up penalties. However, the authors did not present a hardware implementation for this predictor and the power consumption of the predictor itself. In order to fill this gap, we present an implementation of a generic predictor in *very high speed integrated circuit hardware description language* (VHDL). The three main contributions of this paper can be summarized as follows: (1) The *register-transfer level* (RTL) implementation of the generic predictor is introduced in detail. (2) A complete scalability analysis for the RTL implementation of the generic predictor is presented, which includes resource, frequency and power consumption analysis for a *field programmable gate array* (FPGA). (3) We evaluate the hardware predictor by applying the power-saving policy presented in [14]. This shows the efficiency of this policy by predicting the memory idle periods for three different multimedia benchmarks including the power consumption of the predictor itself

This paper is organized as follows: Section 2 presents a brief overview of related work. Afterwards, we present the required background knowledge in Section 3 to understand all following sections. The VHDL implementation of the predictor is described in Section 4 including the whole data path as well as the control unit. Section 5 depicts the complexity, frequency and power consumption analysis as well as the evaluation of the power-saving policy from [14]. Finally, Section 6 summarizes and highlights the contributions of this work.

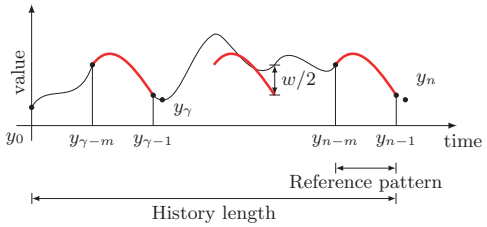

Figure 1: Working of the Predictor

2. RELATED WORK

Predictors are used in many areas of computer architecture. For example in [20] and [13] a predictor was used to reduce average DRAM access latency by forecasting whether an open DRAM row should be closed. Both predictors can therefore only forecast a binary result, whether an action should take place or not. Similarly in [1], a predictor is used to predict DRAM locality to perform page closing decisions.

Another field for predictors is to forecast traffic patterns in networks. In [11] a predictor was used to predict switch-to-switch traffic in a *network on chip* (NoC). A table-driven predictor was presented in [6] to forecast end-to-end traffic. In [4] the authors presented a fuzzy-based predictive traffic model to avoid congestion while maintaining high quality service in *Asynchronous Transfer Mode* (ATM) networks. However, all these mentioned predictors did not present a hardware implementation or any kind of power analysis for their predictors. In [15], a predictor was applied to forecast traffic in NoC, however the authors presented only results based on a software implementation. In [14], the same predictor was used inside a memory controller to snoop memory access. With this data, the predictor forecasts memory idle periods and based on this forecasts a power-saving policy was presented to achieve significant power reductions with only a marginal performance penalty. However, the authors did not present a power analysis for the predictor itself. The authors in [5] presented a fuzzy logic controller running on an FPGA. This controller used a similar technique like [15] and [14] to forecast data points. In [7] the authors presented an implementation of intelligent predictors for solar irradiation running on FPGA. However, both did not provide any analysis on power consumption for their work.

All mentioned predictors were either not able to perform quantified forecasts, the predictor was not implemented in hardware or does not include a complete scalability and power analysis. In contrast, we address all these points in our paper.

3. BACKGROUND

The generic history-based predictor used in this paper was originally proposed in [15], where it was used to forecast traffic pattern for rerouting in networks. In [14] this predictor was used to forecast memory idle periods. Based on this forecast a power-saving strategy was developed to reduce the energy consumption of memory for a *multiprocessor System-on-Chip* (MPSoC). However, the authors did not present a hardware implementation of the predictor. This paper presents a fully synthesizeable VHDL implementation of this predictor. To understand this implementation, the theoretical background of the predictor is introduced in this section.

The general structure of the predictor is depicted in Figure 1. The predictor builds up a history of data points (y_0 to y_{n-1}) before forecasting the next future data point (y_n). Afterwards, 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 searches for similar patterns in the history. If there is a similar pattern in the past that is very similar to the reference pattern, like the pattern between ($y_{\gamma-m}$) and ($y_{\gamma-1}$), the algorithm weights the next data point (y_γ) depending on the similarity. The matching to past data points is not limited to just one occurring pattern set in the history. 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. To understand the RTL implementation of the predictor these steps are explained in more detail:

1) Build history: Before the predictor is able to forecast data points, a set of n data points is required (*history length*).

2) Calculate absolute differences: Next, the algorithm considers the latest m data points between (y_{n-1}) and (y_{n-m}) as *reference pattern*. These reference patterns are subtracted iteratively from the history data points.

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

3) Determine weight/similarity: A parameter *width* (w) is used to identify whether a set of differences fits the reference pattern. A triangular function $\mu(x)$ is applied to all data points to weight all similar ones and set all other which differs by more than $|w/2|$ to zero. To determine the weight for each set of absolute differences, all single weights within this set are multiplied among each other.

$$\beta_i = \prod_{k=0}^{m-1} \mu\left(d_{i,k}\right) \tag{2}$$

4) Weight past data points: In the following, each weight is multiplied with the corresponding data point and summed up. In addition, the sum of all weights is calculated. Both steps are necessary to forecast the next data point and are shown in next step as numerator and denominator.

5) Forecast data point: In the last step N is divided by D and calculates the next future data point.

$$y_n = \frac{N}{D} = \frac{\sum_{\gamma=0}^{n-m-1} \beta_\gamma \cdot y_{n-\gamma-1}}{\sum_{\gamma=0}^{n-m-1} \beta_\gamma} \tag{3}$$

This operation matches the calculation of the weighted mean.

4. IMPLEMENTATION

To implement the predictor in VHDL, several constraints are made. The introduced algorithm works on rational numbers. However, in [14] the predictor was used on unsigned integer and was, nevertheless, able to predict all necessary values. Moreover, the realization for unsigned integer decreases the complexity, since components like adder and multiplier are easier to build. Hence, the predictor is implemented to work on unsigned integer.

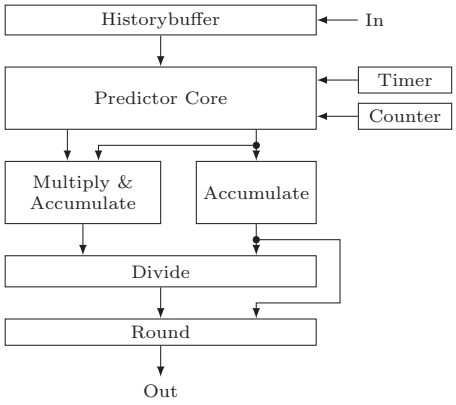

Figure 2: Components in the data path of the predictor

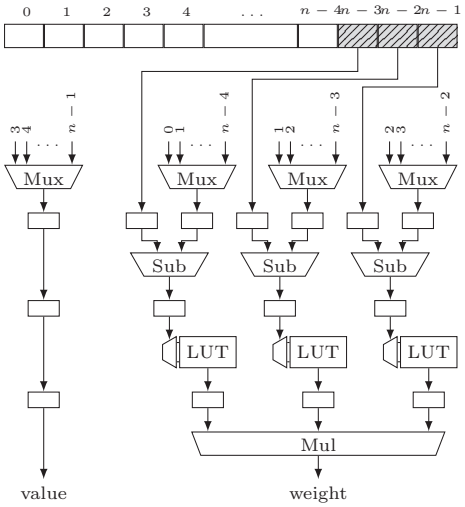

Figure 3: Predictor Core in detail

Figure 2 gives an overview of all components necessary to realize the data path of the predictor. First, a description of all components is provided to realize steps (1) to (5) from Section 3. Afterwards, the control unit is introduced and explained in detail to complete the full predictor unit.

4.1 Data Path

To build up a history as described in Section 3, a generic FIFO buffer is used, where each element is also placed on the output. Step (2), calculate absolute differences, and (3), determine weight/similarity, are both performed in the component *predictor core* which is depicted in Figure 3. The latest entries are the *reference pattern* and are depicted as crosshatched lines. The predictor is generic in terms of the *reference pattern* but set to three in this figure. Each element of the reference pattern is connected to a *subtractor* (sub), respectively. The other input of the subtractors are all possible past data points connected via *multiplexer* (mux) which is controlled by the counter shown in Figure 2. Every clock cycle another set of absolute differences between the reference pattern and the past data points is calculated.

The triangular function from step (3) must be adapted to handle unsigned integers. By extending the Y-range from $[0, 1]$ to $[0, w/2]$ all differences can still be weighted due to their similarity using unsigned integer. The weighting is realized as *lookup table* (LUT). Next, all single weighted

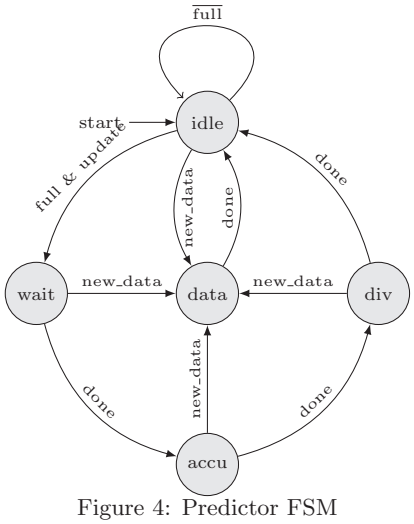

Figure 4: Predictor FSM

values are multiplied among each other to weight the corresponding past data point named as *value*. The design is pipelined to increase clock frequency. As an additional component, a timer is connected to the *predictor core* as depicted in Figure 2. The timer can be used to delay the prediction process by a user defined amount of clock cycles. In [14] this functionality was needed to apply a standard time-out strategy. For step (4), a standard accumulator, taken from [17], is used to sum up all weight and calculate the denominator D. Additionally, a standard multiply and accumulate component was taken from [17] to determine the numerator N by summing up the product of each single weight and the corresponding past data point. Finally for step (5), the devision is realized by using a modified and extended version of the radix-2 divider, presented in [10]. An additional *rounding unit* was implemented which computes, based on the quotient and the remainder, a rounded unsigned integer as result.

4.2 Control Unit

The component depicted in Figure 2 shows the data path of the predictor. A control unit is needed so that all single components work together. The *predictor control unit* is realized as *finite-state machine* (FSM) and is depicted in Figure 4.

The FSM consists of 5 states, namely `idle`, `data`, `wait`, `accu` and `div`. Starting with the state `idle` the FSM is waiting for data input. Independent from the state, every time a new value arrives the FSM changes the state to `data`, add the value to the historybuffer and returns to the `idle` state. As long as the historybuffer is not full and no new data arrived the FSM stays in `idle`. Once the historybuffer is full the FSM changes the state to `wait` to delay the prediction process for a user chosen amount of clock cycles. Following, the FSM goes to state `accu` which triggers the *predictor core* as well as the accumulators. Once the whole history is processed, the FSM changes to state `div` to calculate the final result. After the prediction process has finished, the FSM stays in state `idle` until new data arrives.

5. RESULTS

In this section, first the experimental setup is presented. Afterwards, experiments are shown to analyze the predictors

Table 1: 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

scalability due to the variation of the different predictor parameters. Finally, we apply the generic hardware predictor to analyze the power consumption for the power-saving policy introduced in [14] using real multimedia benchmarks. For more information about the predictor like the accuracy, we refer to [15], since this analysis was already done in detail.

5.1 Experimental Setup

The predictor is implemented as a generic and fully synthesizeable VHDL implementation. As described in Section 3 the predictor has four parameters which have a huge influence on the predictor performance. The parameters are *history length (hl)*, *register size (rs)*, *pattern length (pl)* and *width (w)*. For the rest of this paper, we refer to an instance of the predictor with fixed set of parameters 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 1 is used. The predictor is set up with a parameter set, synthesized with the help of the Xilinx ISE Design Suite 14.6 [18] as well as placed and routed. As target device a Spartan-6 FPGA [19] is used. After the place and route, the scalability of the design is analyzed. To determine the maximum frequency, the design is iteratively synthesized using a constraint on the frequency. Section 5.2 presents the results for the device utilization, run time and maximum frequency.

In Section 5.3 the power consumption is analyzed using the lowest frequency (50 MHz) achievable for all configurations within the design space as well as the same 4 bit input sequence to achieve comparable results. The power consumption is determined by simulating each configuration with Mentor Graphics ModelSim 6.6SE [8] and analyzed afterwards using the XPower Analyzer [18].

Finally, in Section 5.4 results are presented by applying the predictor to real multimedia benchmark. In [14] a power-saving policy was presented to reduce memory power consumption for DRAMs applying the same predictor on memory idle cycles. Based on the predicted length of these idle cycles one of two memory power-saving modes is used to reduce power consumption with a negligible performance penalty. However, this work does not consider the power consumption of the predictor itself. The determined data from [14] are taken and validated against the power numbers from a predictor configuration running on an FPGA.

5.2 Device Utilization Analysis

The parameter w has only a minor influence on the predictor complexity and is therefore not shown in any figure. Increasing the width influences the complexity of all following components only marginally. As shown in previous works [14, 15] the parameter w ranges between 4 to 6 in most cases which gives a maximum bit size of 3. We analyze all *predictor configurations* with a *width* of 4.

Figure 5 depicts the device utilization of all analyzed *predictor configurations* by giving the number of used slice registers and slice LUTs. 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 stacked bar gives the device utilization by depicting the slice register (R) in the lower and the slice LUTs (L) in the upper part for a certain *history length*.

Table 2: Usage of slice registers/LUTs to implement the predictor on an Spartan-6 FPGA

	Usage	Spartan-6	Utilization[%]
#LUT	$245 \dots 1278$	27288	$0.9 \dots 4.68$
#Register	$186 \dots 891$	54576	$0.34 \dots 1.63$

For a given *pl* and *rs*, the number of used slice registers and slice LUTs grows with the increasing *history length*. This reflects the growing history buffer since a longer *history length* requires more registers inside the history buffer to store the past data values. However, the *predictor core* and the following computational components do not become larger, since the *history length* has no influence on these components.

Also the influence of the *register size* on the history buffer can be seen in Figure 5. A larger *register size* results in an increasing complexity, not only because of the history buffer becoming more complex but because of the *predictor core* and the computational components doing so as well.

The increase of the *pattern length* at fixed size *hl* and *rs* cause also a growing complexity, since the *pl* equals the number of chains (mux, sub, LUT) inside the predictor core as can be seen in Figure 3. However, the bit size of the calculated value *weight* increases and causes an increased complexity of all following components, since more values have to be multiplied . The available and used slice registers/LUTs for the Spartan-6 FPGA are shown in Table 2. As can be seen, most of the FPGA is not used and the device utilization is always below 5%.

The predictor has a certain latency to finish the forecast, which ranges between 28 cc for smaller configurations and 120 cc for the maximum configuration. There are two main contributors to the total latency: (1) The historybuffer, since all elements need to be probed clockwise and therefore the number of clock cycles equals the *hl*. (2) The latency of the divider equals the maximum bit size of both input vectors, which is identical to the output vector of the *Multiply & Accumulate* component. The size of this vector can be estimated by the following formula:

$$\mathrm{div}_{\mathrm{cc}} = r + \lceil \log_2 \left(\frac{w}{2} \right) \rceil \cdot pl + (hl - pl)$$

The total latency for the predictor can be estimated as sum of both main contributors.

We also determined the maximum frequency for each predictor configuration. The frequency ranges between 210 MHz and 85 MHz depending on the configuration's complexity. With growing complexity the maximum frequency drops, since operations performed by the predictor's components increase, too.

5.3 Power Consumption Analysis

To achieve comparable results for the power consumption all predictor configurations run with the same frequency and the same test pattern as described in Section 5.1. Figure 6 depicts the power consumption for different *predictor configurations*. One stacked bar depicts the total power consumption for a certain *history length*, *pattern length* and *register*

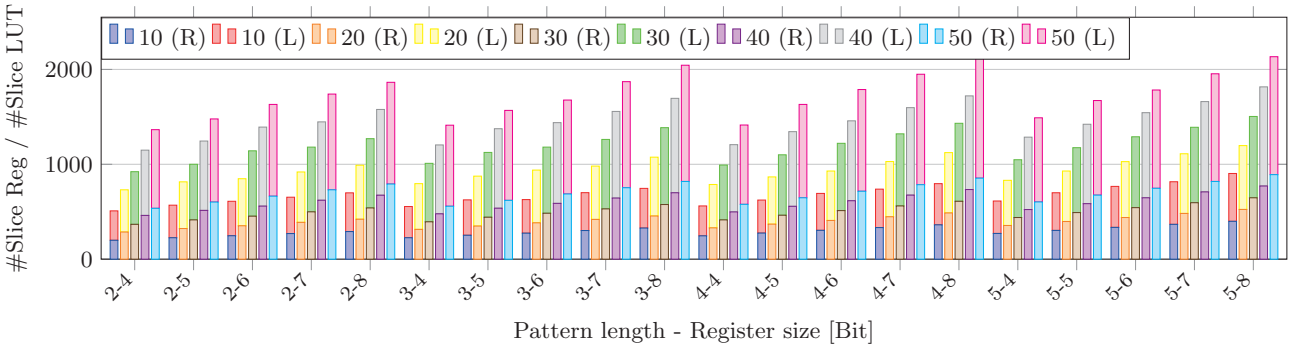

Figure 5: Number of used slice registers and slice LUTs

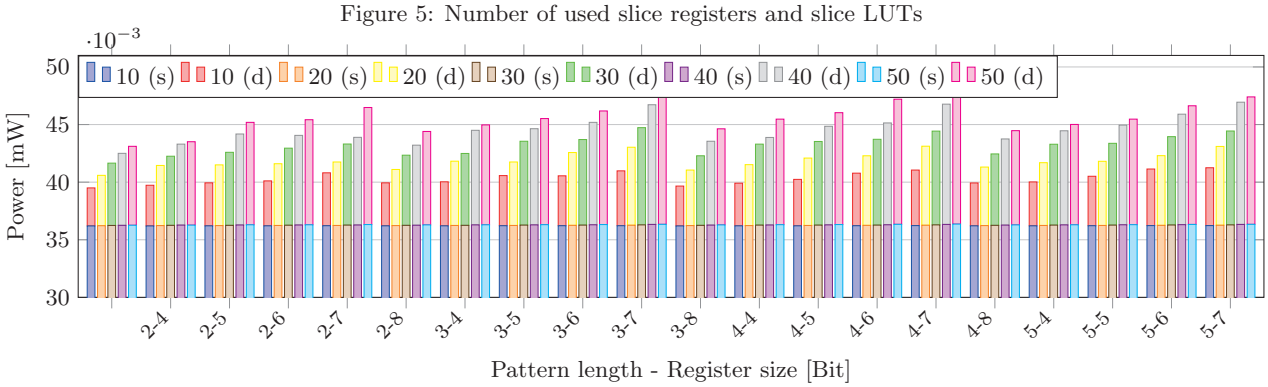

Figure 6: Spartan-6 FPGA Power Consumption

size in Watts. The lower part of each bar gives the static power consumption and the upper half the dynamic one. For better presentation, the y-axis starts at 30 mW. Again, the x-axis shows the *pattern length* and the *register size*.

It can be seen clearly that the static power consumption for each configuration is almost constant. There are some variations for each configuration, however this can not be seen in the figure, since they are in ranges around ±0.1 mW. The reason for 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 [16], since the whole FPGA has to be powered. Moreover, our design contributes to the static power only by approximately 6 mW.

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

However, it is important to point again to the high static power consumption compared to our small predictor design as shown in the previous section. Around 85% of total static power consumption is needed to power the whole FPGA and only the remaining 15% is consumed by our design. This power can be decreased tremendously if a smaller or more power efficient FPGA is used. Moreover, also the realization as *application-specific integrated circuit* (ASIC) would decrease the power consumption.

5.4 Memory Idle Prediction

In this section, we evaluate the impact of the predictor forecasting on memory idle periods, based on the approach presented in [14]. The predictor was integrated in the memory controller and used to forecast the length of memory idle periods. Based on this prediction the used memory (Micron DDR3-800 [9]) was set to one of two power-saving modes (power-down or self-refresh). However, selecting the best power-saving mode depends on the length of the idle period, the power-down mode is more gainful for short periods and self-refresh for longer one. Both modes have a wake-up penalty if the memory was not powered up in time before the next request arrives. A power-saving policy was presented that combines the best of both power-saving modes.

To validate this power-saving policy [14], the authors presented results for three multimedia benchmarks (H263 decoder, Ray Tracer and JPEG encoder) running on the CompSOC platform [12] and using [2, 3] to analyze power. The predictor was set up with the following configuration [14]: $hl = 50$, $pl = 2$, $w = 4$ and $rs = 4$ bit. The authors reduced the energy consumption between 68.8% and 79.9% with only a marginal increase in execution time form 0.3% up to 2.2%, but did not consider the power consumption of the predictor itself.

The power saving policy used in CompSOC system [14] required the forecast of the idle time to be delivered after a specific period of time after the last activity of the memory. In order to meet this requirement, the predictor presented in this paper was configured as described earlier, synthesized and implemented to target operation frequency of 180 MHz. A test bench was used to insert the same multimedia benchmarks. To analyze the power consumption the same setup presented in Section 5.1 is used.

Figure 7 depicts the results of the power analysis for the three different multimedia benchmarks. The y-axis gives the power numbers and the x-axis the different benchmarks. The first bar (Base) in each set gives the power consumption of the memory without predictor for the corresponding benchmark and is considered as baseline. The second bar for

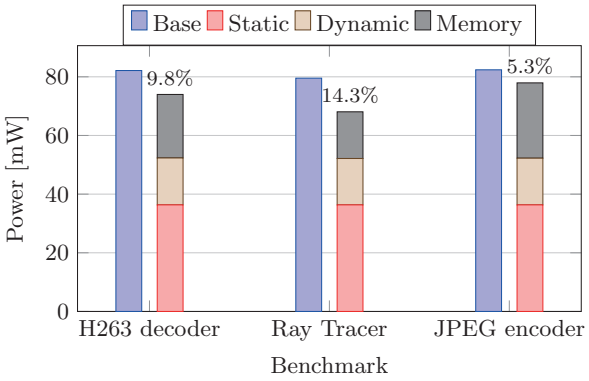

Figure 7: Power consumption analysis for multimedia benchmarks running on the CompSOC platform

each benchmark gives the power consumption of the memory using the power-saving policy from [14] (Memory) as well as the static (Static) and dynamic (Dynamic) power consumption of the predictor itself.

Figure 7 shows that approx. 50% of the total power consumption is caused by the static power of the FPGA. Despite this high static power consumption, the presented power-saving policy [14] still produces beneficial results, since the power consumption of the memory can be reduced by up to 14.3%. However, as already mentioned in previous sections, the whole FPGA has to be powered and therefore 31 mW are needed [16], even if only a small part is used for the design. In our case, this predictor configuration needs less than 5% of the FPGA and consumes only approx. 6 mW. This high basic static power consumption from 31 mW can be reduced by using a more power efficient or smaller FPGA. Moreover, another option to reduce the static power consumption is the realization of predictor as an ASIC which can reduce the power consumption drastically.

6. CONCLUSIONS

In this paper, we have presented a RTL implementation of a generic predictor in VHDL. This predictor is able to produce quantified forecasts and not only binary ones. Furthermore, we have shown how the data path as well as the control unit is implemented. Therefore, we have shown how an algorithm working on real numbers is mapped to hardware working only on unsigned integer numbers. Moreover, we demonstrated that the whole VHDL implementation uses less than 5% resources of an FPGA and still runs with over 210 MHz. We presented power analyses which show the distribution of dynamic and static power when running on an FPGA. Finally, we evaluated a power-saving policy with different multimedia benchmark to reduce the memory power consumption. Using the generic hardware predictor, this power-saving policy reduces the memory power consumption of a DDR3-800 memory by up to 14.3%.

7. REFERENCES

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

[2] K. Chandrasekar, B. Åkesson, and K. Goossens. Improved Power Modeling of DDR SDRAMs. In *14th Euromicro Conference on Digital System Design (DSD)*, 2011.

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

[4] B.-S. Chen, Y.-S. Yang, B.-K. Lee, and T.-H. Lee. Fuzzy Adaptive Predictive Flow Control of ATM Network traffic. *IEEE Transactions on Fuzzy Systems*, 2003.

[5] K. Deliparaschos, F. Nenedakis, and S. Tzafestas. Design and implementation of a fast digital fuzzy logic controller using fpga technology. *Journal of Intelligent and Robotic Systems*, 2006.

[6] Y. Huang, K.-K. Chou, C.-T. King, and S.-Y. Tseng. NTPT: On the End-to-End Traffic Prediction in the On-Chip Networks. In *47th Design Automation Conference (DAC)*, 2010.

[7] A. Mellit, H. Mekki, A. Messai, and S. Kalogirou. Fpga-based implementation of intelligent predictor for global solar irradiation, part i: Theory and simulation. *Expert Systems with Applications*, 2011.

[8] Mentor Graphics. ModelSim, 11 2013.

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

[10] L. Miller. Division in VHDL, February 2009.

[11] U. Y. Ogras and R. Marculescu. Prediction-based Flow Control for Network-on-Chip Traffic. In *43th Design Automation Conference (DAC)*, New York, NY, USA, 2006. ACM.

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

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

[14] G. Thomas, K. Chandrasekar, B. Akesson, B. Juurlink, and K. Goossens. A Predictor-Based Power-Saving Policy for DRAM Memories. In *15th Euromicro Conference on Digital System Design (DSD)*, 2012.

[15] G. Thomas, B. Juurlink, and D. Tutsch. Traffic Prediction for NoCs using Fuzzy Logic. In *2nd International Workshop on New Frontiers in High-performane and Hardware-aware Computing*, San Antonio, USA, February 2011. KIT Scientific Publishing.

[16] Xilinx. *Xilinx Power Estimator 14.3*, October 2012.

[17] Xilinx. *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices*, 14.3 edition, October 2012.

[18] Xilinx. ISE Design Suite 14.6, 11 2013.

[19] Xilinx. Spartan-6 FPGA Family, 11 2013.

[20] Y. Xu, A. S. Agarwal, and B. T. Davis. Prediction in Dynamic SDRAM Controller Policies. In *Proc. 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2009.