Hardware Implementation of the SGASP Cache Memory Prefetcher through HLS

Pablo Sánchez-Cuevas[a,b,*], **Antonio M. Pérez-Peña**[a,b], **Plácido Fernández-Cuevas**[a], **Dagnier A. Curra-Sosa**[a,b], **Santiago Díaz-Romero**[a,b], **Antonio Ríos-Navarro**[a,b,c] [1]

[a] *Robotics and Technology of Computers Lab, Universidad de Sevilla, Spain.*
[b] *Smart Computer Systems Research and Engineering Lab (SCORE), Universidad de Sevilla, Spain.*
[c] *Research Institute of Computer Engineering (I3US), University of Seville, Spain*

* Corresponding author
Emails: {psanchez4, aperez7, pfernandez9, dcurra, sdiaz5, arios}@us.es

Abstract—In a context of rising demand for computing infrastructures for Artificial Intelligence models, the cost and efficiency of deployed systems become key factors. Specific-purpose architectures offer a good solution to this challenge, but like their general-purpose counterparts, they are constrained by memory limitations, which affect both performance and energy cost. To mitigate this issue, this work explores the implementation of a memory prefetcher to hide the latency of accessing external DRAM memory. To this end, we propose a hardware design of the Spatial Greedily Accurate SVM-based Prefetcher (SGASP), a member of the GASP family, which previously achieved results surpassing the state of the art. The components of the prefetcher, including the SVM model optimized for prediction using discrete operations, are efficiently designed and implemented. For the hardware design of the proposed prefetcher, the High-Level Synthesis (HLS) method is used, significantly simplifying the process. The correct operation of the proposed implementations is successfully validated by comparing, for each memory access, the prefetching results obtained using the ChampSim simulator with those obtained by simulating the synthesized hardware. Additionally, through a light configuration of the SGASP prefetcher, very low hardware costs are achieved, demonstrating the adaptability of SGASP to constrained hardware systems and its overall versatility.

Keywords—*Address Prediction, Memory Prefetching, Cache Memory, High Level Synthesis, Machine Learning, Support Vector Machines.*

1. Introduction

In recent years, Computer Science has experienced significant growth, particularly in the field of Artificial Intelligence and Deep Learning, with increasingly widespread applications. However, deploying these models presents major challenges in terms of performance, bandwidth, and energy consumption, which limits the capabilities of computing infrastructures. This affects both the SoCs used in IoT and cloud platforms based on heterogeneous CPU+GPU architectures and FPGA accelerators, making efficiency the main challenge in Computer Architecture.

To address this issue, the use of application-specific hardware such as ASICs, FPGAs, and SoCs has increased, as they offer high performance with low energy consumption, improving the cost-efficiency ratio. Nevertheless, these architectures also face the well-known "Memory Wall," which limits performance due to the slower advancement of memory technologies compared to processors ([7]). To mitigate this problem, cache memories and prefetching techniques are employed to hide memory access latency by bringing data closer to the microprocessor through speculative demand for those memory blocks likely to be accessed by a cache hierarchy level.

Although prefetching is not commonly used in application-specific systems, its adoption could grow in the future due to the increasing use of data-intensive applications in high-performance FPGA-based architectures, such as AMD DPU, CGRA4ML ([15]), or N3H ([13]). In fact, some works like [5], [3], or [10] explore the use of cache memory hierarchies, making memory prefetching a promising solution for applications that utilize external DRAM memory.

To successfully implement prefetching, a prefetcher must exploit memory access locality, which is divided into two types: temporal locality, which assumes that a recently accessed block is likely to be accessed again soon; and spatial locality, which assumes that blocks near a recently accessed block may be accessed in the near future. The latter heuristic has traditionally been the most widely used in state-of-the-art prefetchers, such as BOP ([6]) and AMPM ([4]). In contrast, recent years have seen interesting proposals that leverage Machine and Deep Learning to learn memory access patterns and exploit temporal locality, such as the RAOP prefetcher ([11]) and C-MemMAP ([14]).

In a previous study ([17]), we proposed the Greedily Accurate SVM-based Prefetcher (GASP) family, which stands out for outperforming the state of the art in cost-efficiency. Among other factors, this is due to the accuracy and low cost achieved by the memory access prediction model it incorporates, the SVM For Address Prediction (SVM4AP), which is based on a linear classification SVM model ([16]). The family includes prefetchers based on both temporal locality, such as GASP, and spatial locality, such as SGASP (Spatial GASP).

Given the promising performance in prediction and prefetching shown by the GASP family, as well as its flexibility, this work proposes its hardware implementation targeting FPGAs. Specifically, it presents the design and implementation of the SGASP prefetcher using High-Level Synthesis (HLS). Additionally, it explores design heuristics and how they affect the hardware efficiency of both the prefetcher and its components.

Finally, the SGASP design is validated through a direct comparison between the results obtained by the model in the ChampSim simulator and those obtained by simulating the synthesized hardware.

This work is organized as follows: Section 2 describes the original SGASP prefetching model; Section 3 details the proposed hardware design; Section 4 presents the experiments conducted to validate the hardware designs and discusses the associated performance and cost metrics; finally, Section 5 outlines the conclusions and future work.

2. SGASP prefetcher

This section details the SGASP prefetcher model (which was already proposed and described in [17]): (1) its architecture and the components that form it (which come from the SVM4AP prediction model) are briefly described, as well as the steps executed to compute the prefetch; and (2) the operations calculated by the SGASP linear SVM model are discussed.

2.1. SGASP architecture

The SGASP prefetcher is based on spatial locality and, therefore, takes as input the addresses of the cache memory blocks being accessed. From this history, the model can learn, predict which blocks will be demanded in the near future, and finally, issue the prefetch according to a confidence value.

For these tasks, three components are used, which form the SVM4AP prediction model employed by SGASP:

- **Input buffer**. A table that stores a set of entries, which are indexed by the memory region index being accessed. Each entry contains different bit fields used for prediction and prefetching: the last accessed block address, the sequence of last deltas encoded into classes, etc.
- **SVM**. A linear Support Vector Machine ([1]) that performs categorical classification, taking a sequence of classes as input and outputting the predicted class.
- **Dictionary**. A table that stores an entry for each class used by the predictor, where each entry contains a delta (difference between consecutive addresses) and a counter for the application

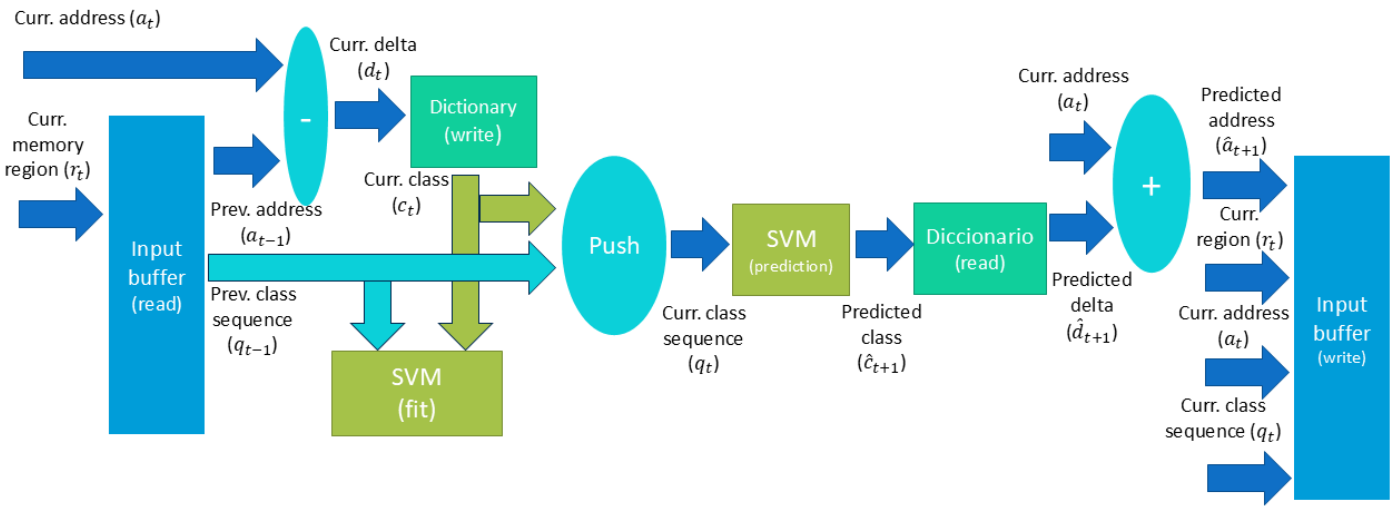

Figure 1. Original SGASP pipeline

of the Least Frequently Used (LFU) replacement policy. Access can be done by class or by delta.

The steps executed by the SGASP prefetcher are detailed below and are illustrated in Figure 1:

1. SGASP receives the block address a_t as input. The memory region index r_t is calculated as a_t shifted right by $\log_2$(#blocks per region) bits. The corresponding entry in the input buffer is accessed using the region r_t, obtaining: (1) the previously accessed block address a_{t-1}, (2) the sequence of classes q_{t-1}, (3) the previously predicted block address $\hat{a}t$, and (4) the previous confidence value $kt-1$. Execution is canceled if the previous address a_{t-1} is equal to the current address a_t. If a miss occurs when accessing the input buffer, a new entry with default values is written (except for a_{t-1}, which is updated with the value a_t) using the Least Recently Used (LRU) replacement policy, and no further steps are taken.
2. If the current address a_t matches the previously predicted address $\hat{a}_t$, the confidence value k_{t-1} is incremented; otherwise, it is decremented. This updated confidence value is called k_t.
3. The difference between the current and the previous address is calculated: $d_t = a_t - a_{t-1}$. Then, the dictionary is queried with the delta d_t: if there is a hit, the class c_t is obtained; in case of a miss, the delta d_t and a default LFU counter value are written into the entry corresponding to the class c_t, chosen via the LFU replacement policy. In the latter case, no prefetch is performed.
4. An adjustment is applied to the SVM model: the previous class sequence q_{t-1} is one-hot encoded as input, while the newly queried dictionary class c_t is taken as the target output (see Section 2.2).
5. The class sequence q_{t-1} is updated by inserting the class c_t, resulting in the class sequence q_t. Then, the class of the delta for the next access $\hat{c}_{t+1}$ is predicted using the sequence q_t as input to the SVM model once it has been one-hot encoded.
6. The predicted delta $\hat{d}_{t+1}$ is read by indexing the predicted class $\hat{c}_{t+1}$ in the dictionary, and the next block address is calculated as: $\hat{a}_{t+1} = a_t + \hat{d}_{t+1}$.
7. For the input buffer entry indexed by r_t, the following are updated: (1) the previously accessed block address with the value a_t, (2) the class sequence with the sequence q_t, (3) the previously predicted block address with the value $\hat{a}_{t+1}$, and (4) the confidence value with the new value k_t.
8. If the confidence value k_t is less than the threshold u_k, no prefetch is applied. Otherwise, the prefetch is issued for the block address $\hat{a}_{t+1}$.

2.2. SVM model computation

As described in [17], the SVM model used in the SGASP prefetcher is implemented using linear and discrete operations thanks to the use of one-hot encoding. This improves accuracy by numerically representing classes without ordinal relationships, increasing the model's dimensionality and improving data separability. Furthermore, it reduces latency and hardware resource consumption.

Let q be the sequence of classes of length n_q that the SVM receives as input, where q_i is the i-th element of the sequence. Each element q_i is a vector $q_{i,j}$ of size n_c, where n_c is the number of classes in the dictionary and $q_{i,j} = 1$ if j corresponds to the class of q_i, or $q_{i,j} = 0$ otherwise.

Therefore, the distance function for each hyperplane of the SVM is described as:

$$f(q,c) = w_c \cdot q - b_c = \left(\sum_{i=0}^{n_q-1} \sum_{j=0}^{n_c-1} w_{c,i,j} \cdot q_{i,j} \right) - b_c \tag{1}$$

, where w_c and b_c are the weight matrix and the bias term for the hyperplane of class c, respectively.

Due to the one-hot nature of q_i, the inner sum can be avoided through multiplexing:

$$mux(q,i,c) = \sum_{j=0}^{n_c-1} w_{c,i,j} \cdot q_{i,j} = w_{c,i,k} \mid q_{i,k} = 1 \tag{2}$$

$$f(q,c) = \left(\sum_{i=0}^{n_q-1} mux(q,i,c) \right) - b_c \tag{3}$$

This reduces the number of operations to just n_q accumulations. Furthermore, when applying gradient descent, the gradients of the weights can only take the value -1 or $+1$ due to the one-hot encoding:

$$\nabla w_{c,i,j} \in \{-1,+1\}, \forall i \in [0,n_q), j \in [0,n_c) \tag{4}$$

Consequently, using a learning rate η equal to a negative power of 2 allows the use of fixed-point numbering and, therefore, the weight adjustment consists of a mere increment or decrement. Thus, the need to use multiplication operations throughout the entire model is avoided.

3. Hardware design and implementation of SGASP

This section explores the various design heuristics adopted for the hardware implementation of the SGASP prefetcher.

To that end, High-Level Synthesis (HLS) has been used— a design method based on the behavioral/algorithmic description of the target hardware. Among other advantages, HLS is chosen due to its versatility for testbenching (allowing an easy integration of complex, trace-based tests) and the lower difficulty of applying a high-level abstraction language to hardware description.

Specifically, in this work, C++ is used to describe the (a) components, (b) operations, and (c) data used in the prefetcher, through: (a) routines and classes; (b) conditional statements, loops, and arithmetic-logic operators; and (c) data structures, arrays, and bit-width configurable variables.

To achieve an efficient and easily verifiable design, the system is divided into five modules: (1) the input buffer, (2) the dictionary, (3) the SVM, (4) the confidence buffer, and (5) the SGASP prefetcher itself, which instantiates the previous modules. These modules are described in the following subsections.

3.1. Input buffer

The input buffer stores one entry/line per memory region, collecting all parameters needed for prediction and prefetching. Given the dynamic nature of this table, it is implemented as an n_w-way associative cache, where each cache line corresponds to an input buffer entry. The index of each line to be accessed is calculated by taking the $\log_2(\#sets)$ least significant bits of the region identifier r_t, and the tag is the remaining bits.

Due to the table's large size, it is implemented using BRAM with dual-port read/write access, resulting in a one-cycle access latency. Note that an alternative implementation using a register file via Look-Up Tables (LUTs) would imply excessive resource consumption (partly due to the cost of multiplexing stored data), and is therefore discarded.

To allow optimal access to the input buffer, the BRAM is organized in two dimensions. That is, one BRAM submemory is used per cache line in the input buffer, enabling direct access to set and way.

Diverging from the original SGASP design shown in Figure 1, the input buffer does not include the confidence value k_{t-1} or the predicted memory address $\hat{a}_t$ for each line. Instead, these are stored in a separate buffer: the confidence buffer. As explained in Section 3.5, this separation significantly reduces the number of clock cycles between reading and writing to both the input and (new) confidence buffers. This also partially alleviates the magnitude of data hazards caused by Read-After-Write (RAW) dependencies in the pipeline.

To address RAW data hazards, forwarding is applied. A forwarding buffer with N^{IB}_{forw} entries is attached to the input buffer (implemented as a circular buffer via a register file), where each entry includes the following bit fields: (1) memory region identifier r_t, (2) block address a_t, and (3) previous class sequence q_{t-1}. To avoid the 1-cycle latency of accessing the BRAM-based input buffer, this forwarding buffer is queried first using the region r_t. On a hit, data is fetched from the forwarding buffer; otherwise, from the input buffer. By keeping the forwarding buffer updated with values to be written to the input buffer, the cost of RAW hazards is avoided.

3.2. Dictionary

The dictionary maps each class to a unique delta via a dynamic table storing one entry per class. Each entry consists of a delta and an LFU counter.

Unlike the input buffer, this table has very few entries and thus requires little capacity. Note that the number of classes n_c in SGASP is limited to around 8–10. For this reason, the dictionary is implemented as a register file, where each entry is accessible by index (the class) or content (the delta). This setup allows for a parallel lookup of the class associated with a given delta and enables simultaneous updates (increment or decrement) of the confidence values of all entries within the same clock cycle.

3.3. SVM

The SVM design is the most complex module in SGASP. It consists of four components:

- The weight matrix W and the bias vector B of the model's hyperplanes (each hyperplane corresponds to a class).
- The computation unit that calculates the distance from each hyperplane to the input sequence, following Equation 3.
- The prediction unit, which returns the class of the hyperplane closest to the input sequence.
- The update unit, which applies an increment of $+1$ or decrement of -1 to the weights and biases of W and B, respectively.

The weight matrix W is memory-intensive, as it stores every weight $w_{c,i,k} \in W$ where $c \in [0, n_c), i \in [0, n_q), k \in [0, n_c]$. Due to the high resource cost of multiplexing this many values, similar to the input buffer, it is implemented using LUTRAMs. On the other hand, the bias vector B, being much smaller, is implemented as a register file.

Given the three-dimensional shape of the weight matrix W, the design proposes to parallelize the first two dimensions. In other words, W is implemented as $n_c \times n_q$ dual-port LUTRAMs, each storing $n_c + 1$ weights. This allows simultaneous access to all weights required by the distance function in Equation 3, where the function $mux(q, i, c)$ translates into a LUTRAM access using q_i as the index.

The distance computation unit accumulates weights as per Equation 3. To reduce latency, binary summation is used instead of fully sequential addition—summing values in pairs concurrently, reducing propagation time logarithmically. Intermediate values require only $\log_2(n_q)$ extra bits of width.

Similarly, the prediction unit performs a sequential operation—finding the minimum distance using comparisons. Distances are computed using the distance unit, and binary minimum operations (pairwise comparisons) are used to reduce latency.

Finally, the update unit uses the distance unit to calculate each hyperplane's distance to the input q. If the hyperplane produces a false positive, its weights are incremented by $+1$ and bias decremented by -1. For a false negative, weights are decremented and the bias is incremented.

3.4. Confidence buffer

As mentioned in Section 3.1, the confidence buffer is a table with the same number of entries as the input buffer. Each entry stores the confidence value k_{t-1} and the predicted memory address $\hat{a}_t$.

Unlike the input buffer, the confidence buffer is not implemented as a cache but as a table accessed by the same set and way used to access the input buffer. This ensures a one-to-one correspondence between input and confidence buffer entries.

The confidence buffer is implemented similarly to the input buffer: a set of dual-port BRAMs arranged in two dimensions, with one BRAM per entry. To address RAW hazards, the same forwarding mechanism is applied using a forwarding buffer with N^{CB}_{forw} entries (implemented as a circular buffer via a register file). This mechanism follows the same process: first checking the memory region r_t in the forwarding buffer, then in the confidence buffer (see Section 3.1).

3.5. SGASP

The final SGASP prefetcher design differs significantly from the prefetch model described in Section 2 and shown in Figure 1. The updated FPGA-oriented design is depicted in Figure 2, and its differences from the model are described below.

First, to avoid RAW data hazards, the writing to the input buffer is moved as early as possible, placed right after the dictionary update. Similarly, the reading from the confidence buffer is delayed so that writing to it can occur as late as possible.

Second, to avoid RAW hazards in accessing and updating the SVM weight matrix, two parallel SVM models are used. Specifically: (1) one model performs prediction using the input sequence q_t; (2) the

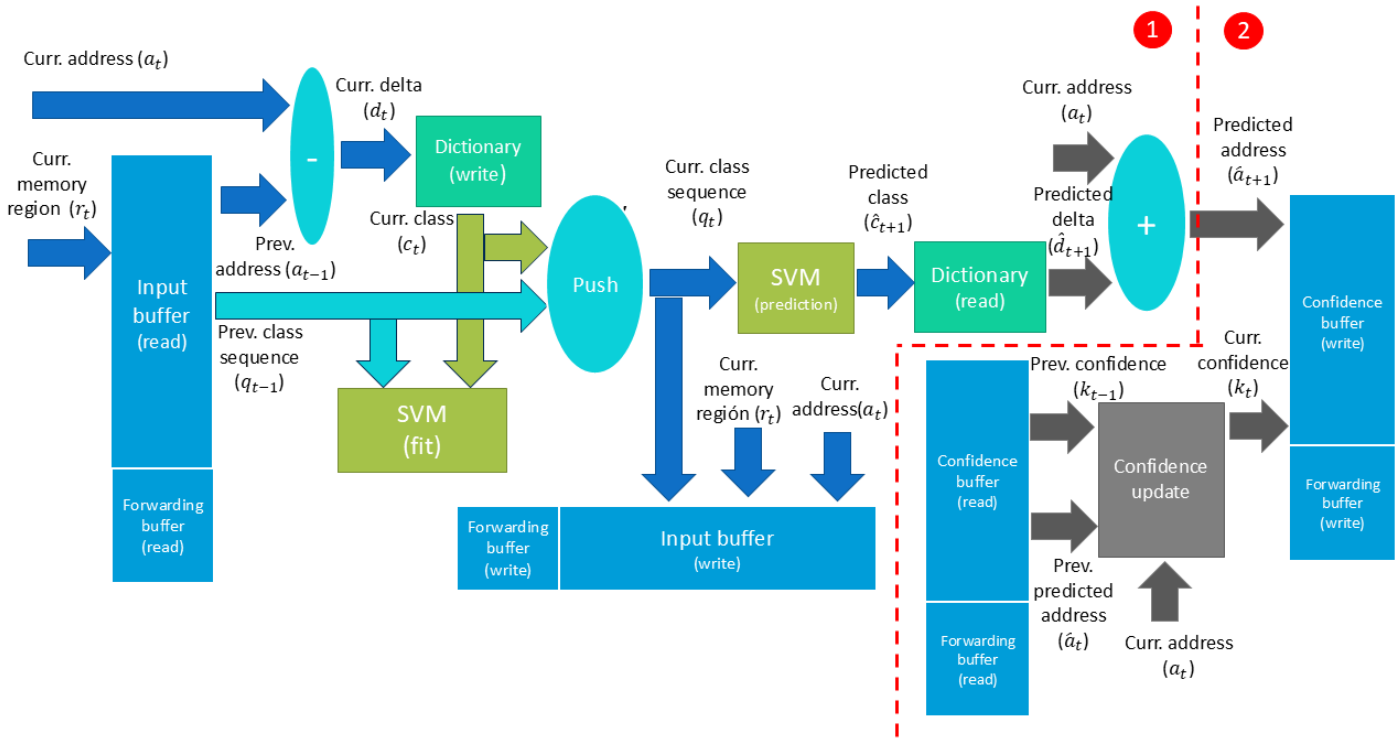

Figure 2. Final hardware design of the SGASP prefetcher. For the *Dataflow* heuristic, the design is split (dashed red line) into two phases (red circles "1" and "2").

other performs updates using the sequence q_{t-1} and class c_t. The updated weights and biases are then copied from the second model to the first.

Finally, to evaluate the control unit's impact on the prefetcher's performance, two HLS-defined operation scheduling methods are applied:

- *Pipeline*. All SGASP operations are part of a single pipeline, with a centralized control unit.
- *Dataflow*. SGASP operations are partitioned into two concurrent pipelines: (1) the pipeline that performs block address prediction and SVM adjustment; and (2) the pipeline that, after consulting the confidence buffer, determines whether the predicted address is reliable enough to be prefetched. Each pipeline has its own control unit and communicates through buffers, allowing for data-driven operation scheduling.

Table 1. SGASP prefetcher configuration.

Component	Configuration
Input buffer	64 sets, 2 ways. 64 memory blocks per region. 1-bit LRU counter
Dictionary	7 classes. 7-bit LFU counter.
SVM	Input sequences of length 6. Learning rate of 2^{-6}.
SGASP	64-bit memory addresses. Memory blocks of 64 bytes. Max. confidence of 15. Threshold of 7. Incr. and decr. of +1 and -4, resp.

4. Validation

Once the SGASP prefetcher has been designed and implemented using HLS, the goal is to validate its behavior. To achieve this, this section presents a methodology based on prior study ([17]) to verify whether the behavior of the hardware developed in this work matches that of the simulated version used in that previous study (baseline).

4.1. Experimental setup

Three key resources have been used to conduct the experiments presented here: the execution dataset of various applications, a general-purpose CPU simulator with a cache memory hierarchy and virtual memory support, and an HLS-based development environment for the synthesis and co-simulation of the described FPGA hardware.

The chosen dataset is a collection of memory traces from the 3rd Data Prefetching Championship (DPC-3), which records the instruction stream executions of several SPEC06 ([2]) and SPEC17 ([8]) applications of different natures. These traces were collected via instrumentation using Intel Pin. Furthermore, among the full execution stream of these applications, the referenced traces correspond to selected regions determined using SimPoint ([9]), which identify memory-intensive or pattern-rich sections. For our validation purposes, and since the goal is solely to verify a hardware design, only a subset of three traces (first 5 million memory accesses) is selected: *imagick*, *lbm*, and *nab*.

The selected simulator for the general-purpose CPU is ChampSim ([12]), a well-known trace-driven simulator (i.e., it simulates program execution based on pre-recorded trace files) used in the latest two editions of the DPC competition (DPC2 and DPC3). ChampSim supports both single-core and multi-core CPU simulation. In this work, using its default configuration, we implement SGASP as a prefetcher in the L1 data cache of a single-core processor. Thus, we obtain the set of memory accesses generated by the CPU to the memory hierarchy.

Using ChampSim and running the SPEC traces, we collect input and output traces for the SGASP prefetcher and its individual components. Specifically, for each memory access, we capture the inputs and outputs of: (1) the input buffer, (2) the dictionary, (3) the SVM, and (4) the SGASP prefetcher itself. These trace sets allow for module-wise

Table 2. Validation results of the SGASP prefetcher and its components.

						Hit rate diff.			Matching rate		
Component	Latency	II	# LUTs	#FFs	#BRAMs	imagick	lbm	nab	imagick	lbm	nab
Input buffer	1	1	750	299	11	0.00%	0.47%	0.00%			
Dictionary	0	1	2439	470	0	-24.16%	-0.03%	0.00%			
SVM	2	1	6233	2546	0				99.80%	95.60%	97.38%
Pipeline SGASP	**6**	1	**15345**	**7272**	15				81.98%	75.26%	**91.97%**
Dataflow SGASP	8	1	16191	8801	15				**94.8%**	**85.1%**	85.9%

validation through testbenching, as described in Section 4.2.

The configuration of the SGASP prefetcher simulated in ChampSim (baseline) is shown in Table 1. Accordingly, for fair validation, the HLS-based hardware implementation of SGASP follows the same configuration.

The selected hardware development environment is Vitis HLS (version 2022.1), which provides tools for C++ code simulation, synthesis, and co-simulation (i.e., simulating synthesized hardware using a software testbench). The repository used for storing the source code is located in [1]. For the experiments presented here, the Ultra96-V2 platform (incorporating the Zynq UltraScale+ MPSoC ZU3EG A484) is used as the target FPGA platform for synthesis.

4.2. Metrics

Validation is performed for both the overall SGASP prefetcher and its individual components. Specifically, the following metrics are used to validate the designs:

1. **Hit rate difference.** Measures the difference in hit rate (percentage of successful accesses) between the baseline and the hardware implementation. This is applied to the input buffer and the dictionary.
2. **Matching rate.** Measures the percentage of output values from the hardware implementation that match the baseline. This is measured for the SVM (based on matching predicted classes) and SGASP (based on matching prefetch addresses).

Additionally, to assess the performance and resource cost of the implemented hardware, the following metrics are collected: latency, initiation interval (II), number of LUTs, number of Flip-Flops (FFs), and number of BRAMs.

4.3. Results

This section presents the experimental results, which consist of: (1) collecting the input/output traces of the components simulated in ChampSim (baseline), and (2) performing co-simulation using the hardware implementation of those components with the collected traces as testbench inputs.

The results are discussed in two parts. First, the metrics are analyzed for individual modules, validating that the prefetcher components have been correctly designed and implemented. Second, the same analysis is applied to the full SGASP prefetcher, for both *Pipeline* and *Dataflow* designs. This allows comparing the two designs and evaluating system correctness as well as the impact of operation timing, which can cause RAW-type hazards.

Results for SGASP components (input buffer, dictionary, and SVM) are listed in Table 2. All components are shown to be validated successfully, with hit rate differences for the input buffer and dictionary being close to zero (except for *imagick*, where the hardware dictionary surpasses the baseline hit rate by 24.16%), and SVM matching rates exceeding 95%. Thus, the hardware components behave identically to their simulated counterparts in ChampSim.

[1] https://github.com/Hematies/PredicMem25

Table 3. SGASP low-cost prefetcher configuration.

Component	Configuration
Input buffer	64 sets, 2 ways. 16 memory blocks per region. 1-bit LRU counter
Dictionary	5 classes. 7-bit LFU counter.
SVM	Input sequences of length 4. Learning rate of 2^{-5}.
SGASP	32-bit memory addresses. Memory blocks of 16 bytes. Max. confidence of 15. Threshold of 7. Incr. and decr. of +1 and -4, resp.

Regarding performance, all components achieve pipelined execution (II=1) with low cycle latencies. Hardware resource usage is mainly concentrated in the SVM module, where the weight matrix and prediction/update mechanisms account for the highest usage of FFs and LUTs respectively. These costs can be significantly reduced in smaller SGASP configurations, particularly by reducing the number of classes n_c and the sequence length n_q.

Finally, the results for the full SGASP prefetcher under both *Pipeline* and *Dataflow* designs are shown in Table 2. The resource usage is similar for both heuristics, with the *Dataflow* design being slightly more expensive. The additional cost and the two-cycle latency difference compared to the *Pipeline* version are attributed to the interface buffers used between the two stages of the *Dataflow* architecture.

The absolute cost in LUTs, FFs, and BRAMs is high. However, it must be noted that the implementations analyzed in this section correspond to configurations compatible with a 64-bit general-purpose CPU as simulated in ChampSim. As previously mentioned, SGASP's cost can be significantly reduced by targeting smaller configurations and using 32-bit address widths (as shown in Section 4.4), which are more common in FPGA-based application-specific architectures.

Regarding the matching rate, Table 2 shows differences between the *Pipeline* and *Dataflow* heuristics. Specifically, the *Dataflow* design achieves a higher match with the baseline results, except for the *nab* trace. However, in general, both heuristics yield a matching rate above 75% across all traces (up to 85% for *Dataflow*), indicating that both implementations successfully replicate the behavior of the SGASP prefetcher as implemented in ChampSim. Therefore, these designs are expected to provide potential performance acceleration in systems that are highly dependent on memory access behavior.

4.4. Hardware costs for specific-purpose setups

Finally, in this section, we explore the reported hardware costs of the SGASP when configured for a specific-purpose system. As such,

Table 4. Hardware costs of the low-cost SGASP prefetcher and its components.

			Synthesis (Vitis HLS)			Synthesis (Vivado)			Implementation (Vivado)				
Design	Latency	II	# LUTs	#FFs	#BRAMs	# LUTs	#FFs	#BRAMs	# LUTs	#FFs	#LUTRAMs	#BRAM36s	#BRAM18s
Low-cost Pipeline SGASP	5	1	8013	3501	9	3031	2567	5	2826	2215	5	2	6
Low-cost Dataflow SGASP	6	1	11253	5676	9	3260	3180	5	2805	2457	5	2	5

we aim for a low-cost implementation that can be integrated with FPGA-compatible architectures such as MicroBlaze or RISC-V.

Both *Pipeline* and *Dataflow* designs are here implemented following the low-cost configuration that is featured in Table 3. We can highlight the 32-bit memory addresses, the 5 dictionary classes, and the class sequence length 4 as the main changes in terms of overall hardware costs reduction.

In Table 4, the hardware costs of both designs is shown. Due to the significant optimizations that are applied after the hardware implementation step (a process that incorporates place-and-route operations), the table includes both after-synthesis and after-implementation hardware costs. Synthesis and implementation were carried out by Vitis HLS and Vivado (both with version 2022.1), respectively. That is, firstly, the design is developed and synthesised using Vitis HLS; then, it is exported as an IP core to be imported in a Vivado project; finally, synthesis and implementation are carried out following the Vivado workflow.

The low-cost implementations of SGASP achieve much lower hardware costs than the versions adapted for ChampSim (see Tables 2, 4), both via Vitis HLS synthesis. Furthermore, when it comes to a realistic deployment after the place-and-route process for the target FPGA, it can be seen that both designs (*Pipeline* and *Dataflow*) feature low hardware costs of $\sim$ 2800 LUTs ($\sim$ 1.0% of all LUTs) and $\sim$ 2300 FFs ($\sim$ 0.4% of all FFs). Hence, this demonstrates that the proposed hardware designs of the SGASP prefetcher can be effectively applied to low-cost constrained hardware systems.

5. Conclusions

In this work, the hardware design of the SGASP prefetcher has been proposed and thoroughly described, based on the successful results of a previous study. To achieve this, the HLS (High-Level Synthesis) development flow was used, allowing for an algorithmic/behavioral description of the prefetcher to design complex hardware mechanisms. Specifically, this work introduces significant design changes aimed at implementing SGASP on an FPGA platform, such as the introduction of forwarding mechanisms and the SVM model's compute units based on discrete operations. Moreover, the design supports implementation using either a single control unit (Pipeline) or two concurrent control units (Dataflow). The proposal was successfully validated by comparing the output of the SGASP simulated in ChampSim (baseline) with the output generated by the proposed hardware implementation. For the full system, a matching rate above 75% was achieved for the Pipeline implementation and over 85% for the Dataflow implementation. Therefore, it is confirmed that the hardware implementation behaves identically to the SGASP model simulated in ChampSim, and it is expected that the hardware will deliver similar performance acceleration results to those previously reported in ChampSim. Finally, a low-cost configuration for both *Pipeline* and *Dataflow* designs was applied in order to fit in specific-purpose systems. After applying synthesis and implementation (via Vitis HLS and Vivado), both SGASP designs entailed very low hardware costs ($\sim$ 1.0% of LUTs and $\sim$ 0.4% of FFs), demonstrating SGASP's capability to adapt to a wide range of hardware budgets.

References

[1] M. Farrens, B. Culpepper, and M. Gondree, "SVMs for improved branch prediction. report for ECS201A computer architecture", Jan. 2004.

[2] D. Ye, J. Ray, C. Harle, and D. Kaeli, "Performance characterization of spec cpu2006 integer benchmarks on x86-64 architecture", in *2006 IEEE International Symposium on Workload Characterization*, 2006, pp. 120–127. DOI: 10.1109/IISWC.2006.302736.

[3] H. Devos, J. Van Campenhout, and D. Stroobandt, "Building an application-specific memory hierarchy on fpga", in *2nd HiPEAC Workshop on Reconfigurable Computing*, 2008, pp. 53–62.

[4] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for high performance data cache prefetch", *J. Instr. Level Parallelism*, vol. 13, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:8375218.

[5] O. Arcas-Abella, A. Armejach, T. Hayes, *et al.*, "Hardware acceleration for query processing: Leveraging fpgas, cpus, and memory", *Computing in Science & Engineering*, vol. 18, no. 1, pp. 80–87, 2016. DOI: 10.1109/MCSE.2016.16.

[6] P. Michaud, "Best-offset hardware prefetching", in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 469–480. DOI: 10.1109/HPCA.2016.7446087.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017, ISBN: 0128119055.

[8] J. Bucek, K.-D. Lange, and J. V. Kistowski, "SPEC CPU2017 : Next-generation compute benchmark", in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18, Berlin, Germany: Association for Computing Machinery, 2018, pp. 41–42, ISBN: 9781450356299.

[9] Q. Wu, S. Flolid, S. Song, J. Deng, and L. K. John, "Invited paper for the hot workloads special session hot regions in SPEC CPU2017", in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 71–77.

[10] T. Pacini, E. Rapuano, G. Dinelli, and L. Fanucci, "A multi-cache system for on-chip memory optimization in fpga-based cnn accelerators", *Electronics*, vol. 10, no. 20, 2021, ISSN: 2079-9292.

[11] P. Zhang, A. Srivastava, B. Brooks, R. Kannan, and V. K. Prasanna, "RAOP: Recurrent neural network augmented offset prefetcher", in *The International Symposium on Memory Systems*, ser. MEMSYS 2020, Washington, DC, USA: Association for Computing Machinery, 2021, pp. 352–362, ISBN: 9781450388993.

[12] N. Gober, G. Chacon, L. Wang, *et al.*, "The championship simulator: Architectural simulation for education and competition", *ArXiv*, vol. abs/2210.14324, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:253117130.

[13] Y. Gong, Z. Xu, Z. He, *et al.*, "N3h-core: Neuron-designed neural network accelerator via fpga-based heterogeneous computing cores", in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22, Virtual Event, USA: Association for Computing Machinery, 2022, pp. 112–122, ISBN: 9781450391498.

[14] P. Zhang, A. Srivastava, T.-Y. Wang, C. A. F. De Rose, R. Kannan, and V. K. Prasanna, "C-memmap: Clustering-driven compact, adaptable, and generalizable meta-lstm models for memory access prediction", *International Journal of Data Science and Analytics*, vol. 13, no. 1, pp. 3–16, Jan. 2022, ISSN: 2364-4168.

[15] G. Abarajithan, Z. Ma, Z. Li, S. Koparkar, R. Munasinghe, and F. Restuccia, "Cgra4ml: A framework to implement modern neural networks for scientific edge computing", Aug. 2024. DOI: 10.48550/arXiv.2408.15561.

[16] P. Sanchez-Cuevas, F. Diaz-del-Rio, D. Casanueva-Morato, and A. Rios-Navarro, "Competitive cost-effective memory access predictor through short-term online svm and dynamic vocabularies", *Future Generation Computer Systems*, vol. 164, p. 107 592, 2025, ISSN: 0167-739X.

[17] P. Sanchez-Cuevas, F. Diaz-del-Rio, E. Piñero-Fuentes, *et al.*, "Gasp: A cost-effective svm-based cache prefetcher for mono- and multi-core processors", *Pending peer-review in the Journal of Parallel and Distributed Computing*, 2025.