Efficient data access management for FPGA-Based image processing SoCs

Zahir Larabi, Yves Mathieu, Stéphane Mancini

Efficient data access management for FPGA-Based image processing SoCs

Zahir Larabi and Yves Mathieu
TELECOM ParisTech,
46, rue Barrault
75634 Paris , France
{zahir.larabi,yves.mathieu}@telecom-paristech.fr

Stéphane Mancini
GIPSA-Lab,
46, Avenue Félix Viallet
38031 Grenoble Cedex, France
stephane.mancini@grenoble-inp.fr

Abstract

In this paper, we propose a low-cost n-dimensional cache (nD-Cache) architecture for FPGA-Based image and signal processing Systems On Chip (SoCs). The architecture allows efficient access to structured data as in 2D or 3D images. We developed a theoretical model for our architecture. It gives a methodology to define cache's practical implementation based on the application and system parameters. Complexity and performance measurements for selected image processing algorithms like jumping snake and 2D Back-Projection are done and compared to classical solutions like associative caches. The architecture is shown to be efficient for tracking algorithm applications by exploiting spacial and temporal locality. Numerical results indicate that 50% improvement in run-time performance can be achieved.

Keywords
FPGA SoC, cache memory, structured data caching, Adaptative Predictive Cache, image processing.

1. Introduction

Modern FPGA allows designing of Programmable SoC (PSoC) using specific blocs such as DSP, CPU, Embedded RAM ... However, this embedded memory is insufficient for complex image and signal processing systems. Thus, the use of external memory is mandatory. Modern technology brings large amount of cheap memories (SDRAM) at the cost of increasing access latency. Typically, cache-memories build from on-chip memories are used to speed-up the system.

There are some cache Intellectual Properties (IPs) for FPGA-embedded microprocessors. These IPs are more or less complex depending on the strategy used for data replacement. The cache IPs proposed by the FPGA vendors are, to our knowledge, direct mapped caches which are too simple to provide desired performance in image processing applications [15, 7, 20]. This is due to the relatively slow operating frequency of the old generation FPGA systems. With new FPGA-SoCs, it is essential to have efficient cache strategies to share memory access between several processing IPs. Nevertheless, some IP vendors provide more complex cache implementation for FPGA, for example, Gaisler company proposes an open source tunable IP set-associative cache in the SoC environment of LEON3 [1].

Traditional caches exploit spatial and temporal locality, but image and signal processing applications process massive amounts of data and temporal locality is not very abundant. For traditional caches, spatial locality occurs in one dimension, a line is fetched on a cache miss. Image processing applications operate on small blocks of 2D data.

This paper focus on the data cache memory for image and signal processing. The architecture we propose (nD-Cache) allows efficient access to structured data as in 2D or 3D images. A theoretical model of this architecture is developed. The cache is suitable for a large class of applications that fetches data from an n-dimensional data structure. It is targeted to be used in the context of application specific hardware on FPGAs where the nature of the algorithms is clearly identified. The prefetching strategy of the cache is independent from the applications and can be tuned with few parameters.

2. Related Work

Cache architectures for general purpose processors have been optimized to deal with structured data and especially for multimedia applications [22, 8]. However, to exploit the parallel computation capacities of FPGA, the rule is to design specific memory access architectures [17].

2.1. Cache architectures for structured data management

Performance's gains may come from a suitable static parameterization of the cache (number of lines, size of line,

replacement policy), prefetching strategies and dynamic reconfiguration of the cache's parameters [19, 16].

The challenge of a prefetching strategy is to estimate the cache lines to prefetch from an analysis of the past fetches, without the knowledge of the initial data structure. The One Block Lookahead (OBL) [21] technique fetches consecutive cache lines based on the reference causing a cache miss. Stride Prediction Table (SPT) [13], used in the Intel-Core processor [11], associates to each load instruction the previous fetched address to compute the stride with the new reference address. SPT prefetches the line a stride ahead the current reference. Although SPT is efficient for high-end micro-processors, it is too complex for specific hardware, FPGA targets and embedded systems because it needs an additional associative memory to store the loaded instructions and the associated tags.

The dynamic tuning of the cache memory tries to optimize the efficiency of applications for which memory access patterns may vary in time. [3] proposes to reconfigure a tunable cache when a phase transition is detected at fixed intervals. The reconfiguration process needs an exhaustive search of the available cache parameters to reduce the miss rate.

Some knowledge about the pattern access may lead to efficient prefetching mechanisms. As an example, texture caching for 3D rendering, benefits of some assumptions about the access pattern [18, 6]. Some information about the size of an image can also be used to exploit 2D locality and perform neighbor prefetching [9]. [14] reports satisfying results of a Markov predictor based prefetching but the important memory's need to store the matrix of transition probability makes it impracticable.

Specific caching hardware can be implemented, more or less tighten to the application. The most obvious strategy is to pipeline computations and memory accesses. But it makes little use of the fetch coherency and parallelization is difficult. Similar to pipelining, deterministic caching [10] analyzes a part of the fetch sequence to compute the needed data. It may be of low overhead but some memory is necessary to store the fetch sequence and the corresponding intermediate internal variables. On-line cache accesses with a prefetch mechanism is the most efficient way to reach a high throughput with a low pipeline latency.

2.2. Optimization of applications for cache efficiency

Applications have to be transformed in such a way that they produce fetches in a cache friendly way: the next iteration of a loop has to produce a fetch at an address close to the previous one. The main results we can find in the literature are about the transformation of nested loop when data indexes are affine functions of loop indexes [5]. Tiling is another popular optimization which decomposes a loop into a higher level loop to produce tiles and an inner loop in each tile [12]. Furthermore, the combination of the transformations of an application together with a re-mapping of the data structure in the memory can lead to a high cache efficiency of a direct mapped cache, which is of low hardware cost [5]. These solutions are shown to be efficient at the expense of a lack of genericity.

3. A model of targeted data access applications

Our cache is intended for applications with multidimensional data. For example in 2D image processing, by analyzing the accessed pixels addresses, the cache predict the 2D block of memory elements to pefetch. Successive access fetches represent displacements in the data structure. A typical example is shown in figure 1, we assume that the displacement in one dimension is the sum of a low speed global displacement and a high speed local displacement. The sequence is characterized by three parameters:

- N : the number of dimensions in the data structure.
- v^n: the maximum speed of the global displacement for the n^{th} dimension.
- A^n: the magnitude of the local displacement for the n^{th} dimension.

We assume that displacements in different dimensions are independent. The tracking mechanism is performed in each dimension independently.

An ideal cache should be able to contain the data corresponding to local displacements and predict the global displacement in order to update the cached data. Targetted applications should fit this model or should be transformed to comply with it.

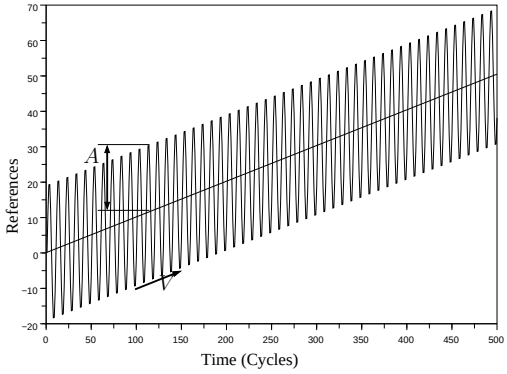


Figure 1. A model of targeted application in one dimension

4. The Cache architecture

The principle of the cache is to copy n-Dimension (nD) block elements of the main memory in the cache memory to speed up memory access by prefetching the element the processing unit would use in advance. Doing so, we reduce cache miss while moving in the n-Dimensions. To predict the cached zone position we measure the mean of the addresses issued by the processing unit. Low-pass IIR filters are used to compute these means. The cache is updated each time the mean value drift to much from the cache center (the center of the cached block). Therefore, a virtual zone is defined around the current cache center called “guard zone”. The cache does not move if the computed mean is inside this guard zone. The cache center is updated with a fixed step when the estimated mean crosses a border of that zone. At each movement, only the needed data are updated. The prefetching mechanism, is called a *tracker* hereinafter. Figure 2 shows the cache zones for a one dimension example.

- $2T$ is the size of the cached zone,
- 2Γ is the size of the guard zone,
- Δ is the tracker step displacement.

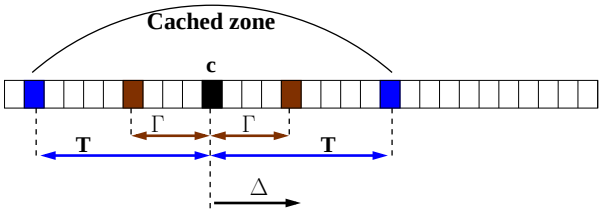


Figure 2. The Cache zones

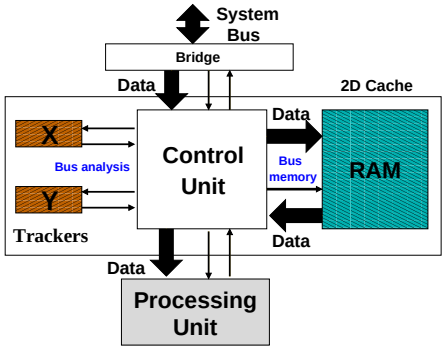


Figure 3. The cache architecture for 2D signal

The cache architecture as illustrated in the figure 3, is composed of of:

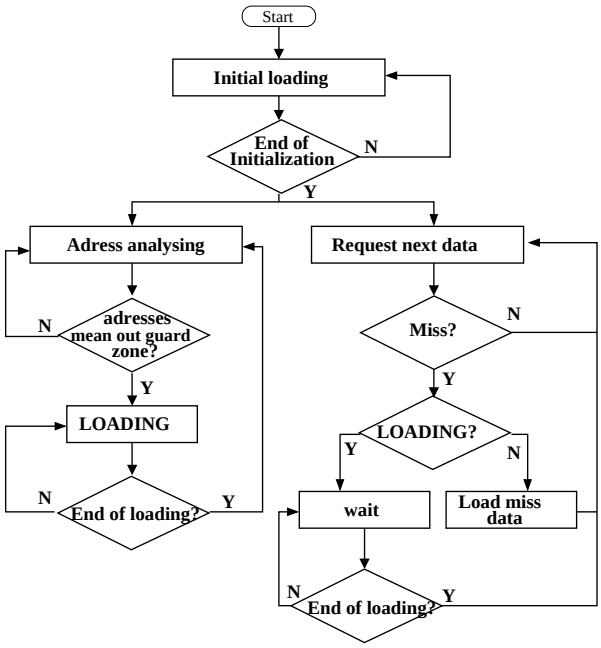


Figure 4. The cache algorithm flow

- **Trackers** : They estimate the zone of data to cache and prefetch.
- **Control unit** : It performs the memory mapping of indexes into addresses and, loads zones of data upon requests of trackers.
- **An embedded double port memory** : thanks to this memory the cache updates are performed concurrently with the cache accesses, this kind of memory is available in numerous FPGAs [2].
- **External bus interface**: The cache provides a virtual interface to the processing unit that issues multidimensional indexes in the data structure. The cache performs the memory mapping between indexes and the external memory addresses by an interface to standard bus like Avalon, CoreConnect or AMBA.

Figure 4 illustrates the cache algorithm flow. It starts by *Initial loading* which corresponds to either “cold start” or “ shift in the sequence bigger than the cache size”. Thereafter, we read (*request next data*) and analyze the addresses (*address analyzing*) simultaneously. If the computed mean of addresses crosses the limit of the guard zone, we shift the cached zone (*loading*). The cache can be read concurrently with the *loading* process. If there is a cache *miss* outside the *loading* phase, we read back the missed data from the external memory (*load miss data*), but, if the miss occurs during the *loading* phase, the cache will *wait* until the *end of loading*.

5. The theoretical model of the Cache

To compute the runtime parameters of the cache (T, Γ and Δ), we developed a theoretical model of the cache. This model is a starting point to understand how the cache can be dynamically set. The entry of the model is a part of a fetch sequence and more precisely v the speed of the global displacement and A the magnitude of the local displacement. To set the cache parameters, we need also the system characteristics ie. memory latency in cycles (Lat) and bus throughput in data by cycle (m).

Table 1 details the accesses and updates chronology of the cache. In the phase 1 the mean e_i is out of the guard zone $[c_i - \Gamma, c_i + \Gamma]$, we make the assumption that the next fetch will be in the direction of the crossed border. Phase 2, the actual center c_{i+1} is then updated to $c_i + \Delta$ and the zone $[c_i - T, c_i - T + \Delta]$ is invalidated. In the phase 3, the cache loads the needed data $[c_i+T, c_i+T+\Delta]$ concurrently with the cache accesses. After $Lat + \frac{\Delta}{m}$ cycles, the cache is updated (phase 4) and the new available cached zone is $[c_i - T + \Delta, c_i + T + \Delta]$ (phase 5). In the phase 6, the mean is once again out the guard zone so we return to phase 1.

5.1. The cache constraints

From table 1, to avoid cache misses, we must check some constraints:

1. We must not ask for new cached zone before the end of the current loading to avoid conflicts:

$$\Delta + \Gamma > \Gamma + v.(Lat + \frac{\Delta}{m}) \tag{1}$$

2. While loading, we must not have cache miss, to avoid waiting time. The loading time is $Lat + \frac{d}{m}$ cycles, the mean moves from the position $c_i + \Gamma$ to the position $c_i + \Gamma + v(Lat + \frac{\Delta}{m})$. The second constraint is:

$$\begin{cases} \Gamma + v(Lat + \frac{\Delta}{m}) + A & < & T \\ \Gamma - A & > & -T + \Delta \end{cases} \tag{2}$$

3. To avoid the cache center oscillations, after updating the cache, we must have the computed mean in the guard zone:

$$\Gamma > \frac{\Delta}{2} \tag{3}$$

From the equations (1), (2) and (3), the constraints to check are:

$$\begin{cases} \Delta & > & \frac{v.Lat}{1 - \frac{v}{m}} \\ \Gamma & > & \frac{\Delta}{2} \\ T & > & \Gamma + v(Lat + \frac{\Delta}{m}) + A \end{cases} \tag{4}$$

5.2. The efficiency model

There are several ways to measure a cache efficiency depending on the target specifications. One can measure the ratio between the number of total memory references N_r and the number of clock cycles to get all the data sequence, the hit rate, the bus occupancy, the power consumption, etc... In this paper, we focus on the timing performance given by: Efficiency $= \frac{N_r}{\#\text{cycles}}$. This efficiency takes into account the initialization time of the cache N_{init}.

$$N_{init} = Lat + \frac{2T}{m} \text{ cycles.} \tag{5}$$

If equ. 4 are respected, the cache achieves maximum efficiency which is given by the equ. 6:

$$\text{Efficiency} = \frac{N_r}{N_{init} + N_r} \tag{6}$$

For a given configuration of the cache (T, Γ and Δ), we can compute the maximum latency Lat_{max} (equ. 8) that offers the optimum efficiency. Above that latency Lat_{max}, misses appear during the loading time. In that case, the efficiency can be written as:

$$\text{Efficiency} = \frac{N_r}{N_{init} + (Lat + \frac{\Delta}{m})\frac{vN_r}{\Delta}} \tag{7}$$

$$Lat_{max} = \min\{\frac{T - \Gamma - A}{v} - \frac{\Delta}{m}, \frac{\Delta}{v} - \frac{\Delta}{m}\} \tag{8}$$

Figure 5 represents the efficiency model for a cache with minimal size for $Lat_{max} = 36$.

5.3. Tuning the model

For a given image processing algorithm, from which, we can extract the parameters (v and A) and for a given FPGA-SoC system from which we can extract the mean access latency; we can find the minimal dimensions of the cache using the equ. 4 giving optimum efficiency. We should take into account a safety margin, given the rapid collapse of the performance after Lat_{max}

phase	Access number (cycle)	Mean position (e_i)	Cached zone available	Cache state
1	i^-	$c_i + \Gamma$	$[c_i - T, c_i + T]$	out of the guard zone
2	i^+	$c_i + \Gamma$	$[c_i - T + \Delta, c_i + T]$	zone invalidation
3	$\vdots$	$\vdots$	$[c_i - T + \Delta, c_i + T]$	Loading $[c_i + T, c_i + T + \Delta]$
4	$i + Lat + \frac{\Delta}{m}$	$c_i + \Gamma + v(Lat + \frac{\Delta}{m})$	$[c_i - T + \Delta, c_i + T + \Delta]$	End of loading
5	$\vdots$	$\vdots$	$[c_i - T + \Delta, c_i + T + \Delta]$	cache updated
6	$n = i + \frac{\Delta}{v}$	$c_i + \Delta + \Gamma$	$[c_i - T + 2\Delta, c_i + T + \Delta]$	new out of guard zone

Table 1. Cache accesses chronology

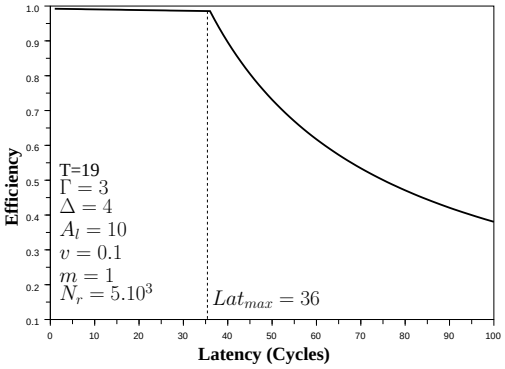


Figure 5. The efficiency model

6. Application and results

In this section, we present measurement of the cache efficiency and complexity. The cache is designed both in VHDL RTL for synthesis and SystemC for high speed simulation. It has been successfully implemented on a SoPC (System On Programmable Chip) prototype and validated with a Avnet Virtex II Pro Development board.

6.1. Complexity

Table 2 gives the complexity results of the 2D Cache for a typical applications and an unconstrained logical synthesis. The synthesis tool reports a 170 MHz frequency for the Virtex 4 FX target.

The hardware complexity and timing of the cache control are almost independent of the size of the embedded memory, contrary to a standard cache (set-associative caches).

Unit	Virtex 4
Control Unit	853 FG, 280 DFF
Tracker	216 FG, 49 DFF

Table 2. The 2D cache complexity

6.2. Experimental results

Performances are measured for several applications such as: “Jumping Snake”, “2D & 3D Backprojection” used in medical imaging [4], “Ray Casting” algorithm used for 3D visualization (lines that propagate in a 3D grid) and “2D tile based video rendering” used in image transformation and composition.

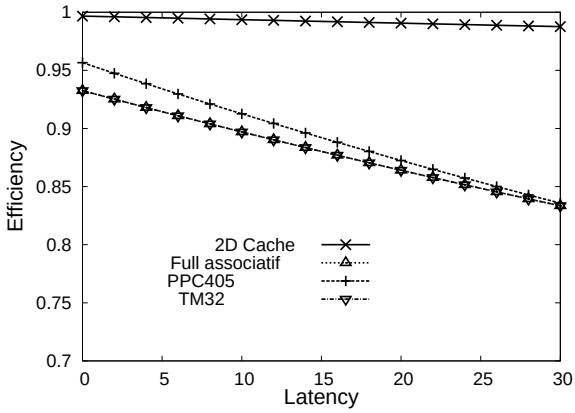
Figure 6 gives the curves of the cache efficiency depending on the system bus (32 bit bus) latency, for the aforesaid applications. These results are given by the cache parameters computed with the method from section 5.The embedded memory cache size is also given for each measure. The 2D Cache efficiency is compared with an ideal model of the following caches:

- Full Associative, 16K, 256 lines of 16 words.
- TM32 cache, 16K, 2 way set-associative, 256 lines of 16 words.
- PowerPC 405 cache , 16K, 8 way set-associative, 512 lines of 8 words.

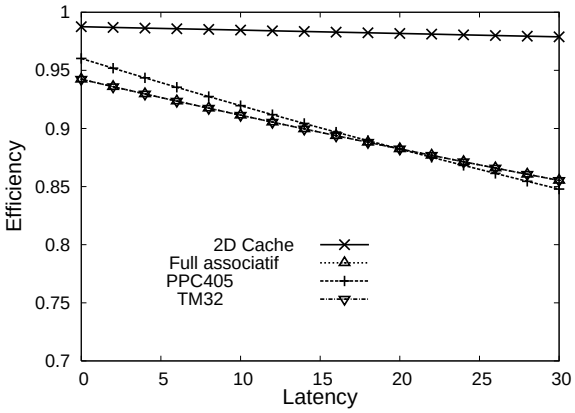
The results demonstrate that the 2D Cache is better in terms of cache efficiency than a standard cache. The model presented in section 5 gives satisfying results.

2D Backprojection and Ray Casting provide almost an ideal performance. For a wide range of memory latencies, the prefetch realized by the cache corresponds exactly to the need of the application ($Lat_{max} > 30$). Excellent results are achieved, in part,thanks to the high rates of data reused by these algorithms.

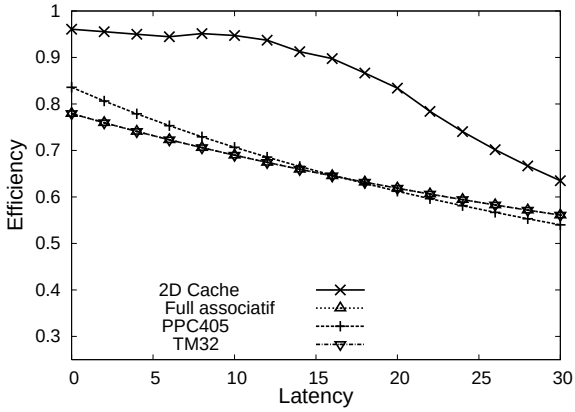
The case of the Snake shows the limitations of the proposed tracking and inefficient configuration of the parameters of the cache in one dimension for latencies over 15. The residual oscillations of the filter imposes a large guard zone. That limits the prediction’s performance and makes it more sensitive to memory latency. This seems to be related to the phase shift of the low pass filter that prevents the tracker to predict the next references on time.



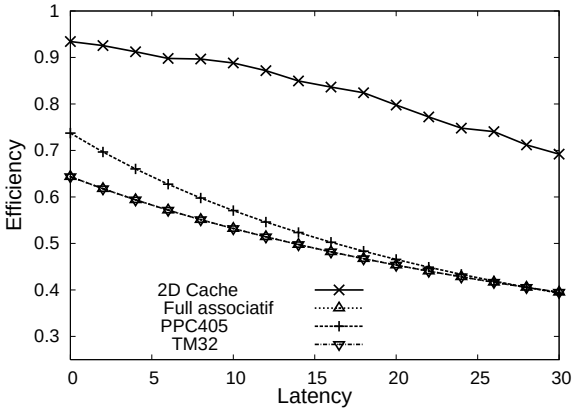
(a) 2D Backprojection: cache size=0.5 KB



(b) Ray Casting: cache size=1 KB



(c) Snake: cache size=0.5 KB



(d) Video rendering: cache size=4 KB

Figure 6. Cache efficiency of a single 2D Cache with automatic setting of the parameters

Finally, an interesting result is the video rendering that is an IP that was designed previously prior to the 2D Cache by another team. The 2D Cache acts as a 2nd level cache and appears to be efficient. The reuse of data is relatively low (high speed movement of the cache center) which makes the cache much more sensitive to memory latency. However, the performance remains more efficient than a standard cache(50% improvement).

If we consider only the tracking algorithm, figure 7 shows that the hit rate of the nD-Cache is near perfect. The efficiency as defined before is therefore a more objective way of comparing the nD-Cache to other architectures.

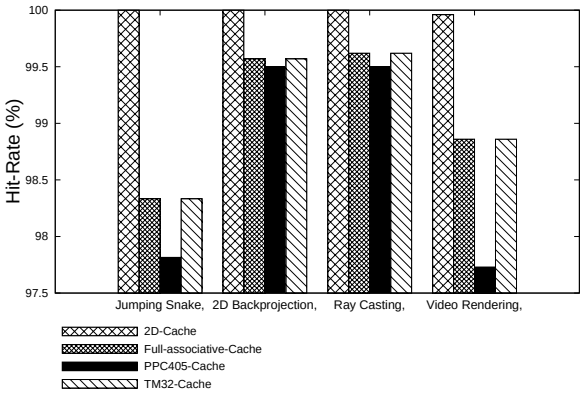


Figure 7. Measured hit rate

7. Conclusion & perspectives

This paper presents the nD-Cache architecture and an associated methodology to compute its runtime parameters. The nD-Cache is a new trade-off between the hardware complexity of the control unit, the size of embedded memory and the cache efficiency. Several prefetching mechanisms and models of fetch sequence are available and the system designer can choose the one that fits its application

best. The tracker presented in this paper can be automatically tuned and is shown to be efficient for several applications.

As already seen in the snake sequence, the two major drawbacks of the simple filters already used for the center tracking are residual oscillations and prediction delay. Auto tunable trackers investigation will permit to dynamically compute the nD-Cache parameters. This preliminary work is still on-going and the evaluation in realistic environment has now to be undertaken.

References

[1] Grlib ip core users manual. *http://www.gaisler.com/*.

[2] Virtex-5 fpga user guide. *http://www.xilinx.com/*.

[3] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. A dynamically tunable memory hierarchy. *IEEE Transactions on Computers*, October 2003.

[4] B. Bendriem and D. W. Townsend. *The Theory & Practice of 3D PET*. Kluwer Academic Publishers, 1998.

[5] F. Catthoor, K. Danckart, C. Kulkarn, and al. *Data Access and Storage Management for Embedded Programmable Processors*. Kluwer Academic Publishers, 2002.

[6] S. Cho, C.-H. Yu, and L.-S. Kim. An efficient texture cache for programmable vertex shaders. In *IEEE ISCAS 2006*.

[7] R. Cucchiara, M. Piccardi, and A. Prati. Exploiting cache in multimedia. *Multimedia Computing and Systems, 1999. IEEE International Conference on*, 1:345–350 vol.1, Jul 1999.

[8] R. Cucchiara, M. Piccardi, and A. Prati. Exploiting cache in multimedia. *Multimedia Computing and Systems, 1999. IEEE International Conference on*, 1:345–350 vol.1, Jul 1999.

[9] R. Cucchiara, M. Piccardi, and A. Prati. Improving data prefetching efficacy in multimedia applications. *Multimedia Tools and Applications*, 20(3):159–178, 2003. Kluwer Academic Press.

[10] C. Cunat, J. Gobert, and Y. Mathieu. A coprocessor for real-time mpeg4 facial animation on mobiles. In *Proc. of ESTIMedia*, 2003.

[11] J. Doweck. Intel smart memory access: Minimizing latency on intel coretm microarchitecture. *Technology @Intel Magazine*, Sept. 2006.

[12] H. Dutta, F. Hannig, and J. Teich. Hierarchical partitioning for piecewice linear algorithms. In *Proceedings of the International Symposium on PARELEC*, pages 153–160, 2006.

[13] J. Fu, J. Patel, and B. Janssens. Stride directed prefetching in scalar processors. *Microarchitecture, 1992. MICRO 25., Proceedings of the 25th Annual International Symposium on*, pages 102–110, Dec 1992.

[14] D. Joseph and D. Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121 – 133, 1999.

[15] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Computer Architecture, 1990. Proceedings. 17th Annual International Symposium on*, pages 364–373, 28-31 May 1990.

[16] D. Kim, R. Managuli, and Y. Kim. Data cache and direct memory access in programming mediaprocessors. *Micro, IEEE*, 21:33–42, 2001.

[17] V. Mariatos, K. Adaos, and G. Alexiou. A novel system-on-chip architecture for efficient image processing. *Rapid System Prototyping, 2008. RSP '08. The 19th IEEE/IFIP International Symposium on*, pages 165–171, June 2008.

[18] S.-J. Park and al. A reconfigurable multilevel parallel texture cache memory with 75-gb/s parallel cache replacement bandwidth. *IEEE Journal of Solid-State Circuits*, May 2002.

[19] D. Patterson and J. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, 2nd ed. edition, 1996.

[20] J. Rivers and E. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. *Parallel Processing, 1996., Proceedings of the 1996 International Conference on*, 1:154–163 vol.1, Aug 1996.

[21] A. J. Smith. Caches memories. *Computing Surveys*, 14:473–530, September 1982.

[22] S. Wong, S. Cotofana, and S. Vassiliadis. General-purpose processor huffman encoding extension. *Information Technology: Coding and Computing, 2000. Proceedings. International Conference on*, pages 158–163, 2000.