Temporal Analysis of Cache Prefetching Strategies for Multimedia Applications

Rita Cucchiara
DSI, University of Modena and Reggio Emilia
Via Vignolese 905
41100 Modena, Italy
rita.cucchiara@unimo.it

Massimo Piccardi
DI, University of Ferrara
via Saragat 1
44100 Ferrara, Italy
mpiccardi@ing.unife.it

Andrea Prati
DSI, University of Modena and Reggio Emilia
Via Vignolese 905
41100 Modena, Italy
andrea.prati@unimo.it

Abstract

Prefetching is a widely adopted technique for improving performance of cache memories. Performances are typically affected by the design parameters, such as cache size and associativity, but also by the type of locality embodied in the programs. In particular multimedia tools and programs handling images and video are characterized by a bi-dimensional spatial locality that could be greatly exploited by the inclusion of prefetching in the cache architecture. In this paper we compare some prefetching techniques for multimedia programs (such as MPEG compression, image processing, visual object segmentation) by performing a detailed evaluation of the memory access time. The goal is to prove that a significant speedup can be achieved by using either standard prefecthing techniques (such as OBL or adaptive prefetching) or some innovative and image-oriented prefetching methods, like the neighbor prefetching described in the paper. Performance are measured with the PRIMA trace-driven simulator.

1. Introduction

Cache memories improve performance in memory access by allowing the exploitation of temporal and spatial data locality. However, spatial locality can actually be exploited only if the storage organization in the cache mirrors the type of data access embodied in programs. This requirement is often not satisfied for multimedia data such as images or video. In fact, image processing computation is characterized by *2D spatial locality* [1,2,3], since, whenever the CPU accesses a single data item, a high probability exists to access logically adjacent data items in both vertical and horizontal direction.

Most algorithms for image compression, noise filtering, and image analysis work on two-dimensional pixel blocks (e.g., MPEG compression uses 8x8 or 16x16 pixels blocks), thus exhibiting both horizontal and vertical spatial locality. For this reason, standard cache architectures are not satisfactory, and in multimedia programs a large amount of compulsory misses in accessing image data has been reported [3,4,5]. Enlarging the cache size is not the right solution, since it decreases only conflict or capacity misses, and even increasing the block size improves exploitation of horizontal locality only [2].

For improving cache performance, *prefetching* techniques have been deeply exploited [6]. Cache prefetching consists in providing data into cache before they are request by the current instruction, so as to eliminate or limit the waiting time due to compulsory misses.

In particular, for data cache prefetching two main directions have been explored: static and adaptive techniques. Static techniques consider prefetching one or more blocks based on a static assumption of probability; the simplest technique, called OBL (*one block lookahead*), prefetches the block following the current one in memory [7,8]. Adaptive techniques analyze history of *strides* (the stride is the difference in address between two consecutive accesses made by a same instruction) to predict the most useful blocks to be prefetched [9]. The simplest adaptive technique computes the last stride by finding it in a history table, adds it to the address of the current block, and prefetches one block at the newly computed address.

The OBL approach achieves high performance on vector data, and this provides a good performance improvement also on image data, since at least the

horizontal locality is used [2]. The second approach is good for programs that work on streams of data with regular data access schemes [4,5]. In [4], Zucker et al. proves the effectiveness of hardware prefetching on MPEG algorithms; the authors propose the use of adaptive techniques based on a Stride Prediction Table (SPT), together with a multiple-way stream memory. In [5], Pimentel et al. evaluate performance of stride prefetching and hybrid prefetching (with stream prefetch instructions, SPT and OBL) for a multimedia case study given by the median algorithm for producing non-interlaced frames from videos with interlaced frames.

Actually, many image processing and multimedia programs are regular: as well as the cited median algorithm, think for instance of a convolution-based filtering, where all the image points are raster scanned and processed by a mask. Nevertheless, some programs are not, since the computation is data-dependent. Typical examples are contour-tracing algorithms, region-growing segmenters or labeling operators, used to extract visual objects from images. In these cases, even stride-based techniques are not adequate.

For these reasons, in a previous work we presented a new, 2D-oriented prefetching technique, called *neighbor* prefetching, that better exploits the 2D spatial locality [3]. In [10] we compared static and adaptive cache prefetching for image processing. In [3], we compared neighbor prefetching with other assessed prefetching techniques and proved that it can significantly improve cache performance in terms of eliminated cache misses. In this paper we extend the performance evaluation to temporal analysis, since the ultimate goal is to quantify what reduction can be achieved in the memory access time.

The scope of our research is in multimedia, and therefore we used a suite of programs massively used in multimedia applications, such as MPEG-2 decoding, low-level image processing tasks, and chain-code representations of visual objects (e.g., useful for animation of visual objects extracted by videos).

The simulations presented in this paper assume that a dedicated cache is used to store image data. Although the introduction of a further dedicated cache in modern general-purpose processors is not very likely, this might not be the case of multimedia processors, which could significantly benefit from results achieved in this work.

The paper is structured as follows: next section briefly describes the compared prefetching techniques. Section 3 outlines the multimedia benchmark used for performance evaluation; Section 4 describes the simulation environment and the temporal model; Section 5 presents the simulation results both in terms of cache misses and of speed-up in memory access time. Conclusions briefly recall the relevant aspects of our work and comment the main results achieved.

2. Cache Prefetching Techniques for Multimedia Image Processing

Prefetching techniques ascribe to the two main categories of static and adaptive prefetching techniques. In this paper, we use OBL as representative for the first category [7,8]; for the second, we use the basic prefetching scheme that exploits the last stride, that we call adaptive for short in the following [9]. In [3], we explored also more sophisticated adaptive techniques based on 2-delta filters [11], but performance achieved didn't prove significantly different from that of the basic scheme, and therefore are not reported in this paper.

In OBL prefetching, every time a memory reference is made, a lookup in the cache is performed for the block that follows in memory the currently referenced one. If we call A0 the current *block address* (i.e. the memory address without the LSBs stating the byte offset inside the block), the address of the looked-up block is A0 + 1 (see Table 1). If this block is absent from the cache, its prefetch is issued. Instead, in adaptive prefetching, if we call S the last computed stride, the address of the looked-up block is A0 + S. The rationale of OBL is that useful data may be located in memory following the current address, while adaptive prefetching assumes regularity in address strides; these assumptions are partially valid in typical image processing for multimedia, but do not consider explicitly the vertical locality.

	Current block address	Prefetch address
No Prefetching	A_0	
OBL	A_0	$A_0 + 1$
Neighbor **8-step Neighbor**	A_0	$A_0 + 1$, $A_0 - 1$, $A_0 + NB_{row}$, $A_0 + NB_{row} + 1$, $A_0 + NB_{row} - 1$, $A_0 - NB_{row}$, $A_0 - NB_{row} + 1$, $A_0 + NB_{row} - 1$
Adaptive	A_0	$A_0 + S$

Table 1. The prefetching techniques compared.

In this paper we propose to explicitly explore the 2D spatial locality by *neighbor* prefetching, where all lines containing pixels in the 3 x 3 neighborhood of the currently referenced pixel (called 8-connected lines) are checked for presence in the cache, and

their blocks prefetched if required. Thus, every prefetching stage consists of eight lookups and M issued prefetches, with $M \in [0,8]$. As well as A0 address, the addresses of the looked-up blocks are in Table 1. This technique mirrors the way data are accessed by many image processing algorithms, including both raster-scan and mask-based tasks on streams of data and data-dependent ones, and thus promises a general improvement in spatial locality's exploitation. At the same time, potential drawbacks of this approach are evident: i) it carries a substantial increase of the lookup pressure, and ii) potentially issues a higher number of prefetches.

In order to limit these effects, we made two further modifications to neighbor prefetching:

1. we assume a highly efficient lookup mechanism, with a multiple-port tag directory and pipelining with prefetch issuing.
2. rather than performing a prefetching stage at each memory reference, we limit that to the first reference in a sequence to a same block. The assumption is that if a block is prefetched at this stage, it won't be substituted in the cache in the near future, so it won't be necessary to check for its presence during next references to the block in the same sequence. We call the modified version *optimized neighbor* prefetching.

However, another drawback of this technique (as well as for the basic neighbor), is that a number M of prefetches will be issued (with $M \in [0,8]$) on the first reference to a new block. This will bring to a huge latency to complete these prefetches. The next improvement (that we call *8-step neighbor*) consists of distributing the M prefetches along time, during the sequence of references to the same memory block.

Let us call block *i* one of the blocks neighboring the currently referenced one, with $i = 1 \ldots 8$ (i equal to 1 for the horizontal direction and increasing in clockwise order). The *8-step neighbor* prefetching stage consists of the following steps:

1. starts with i = *lastdirection*
2. if $i \leq 8$, performs a lookup for block *i*; if $i = 8$ goes to step 4
3. if the lookup result is

   hit: datum is present in the cache → set $i = i + 1$ and goes to step 2,

   miss: datum is not present in the cache → load the datum then go to step 4
4. *lastdirection* = i, end of current prefetching stage

A prefetching stage is issued at each memory reference; lookups at step 2 are allowed with a parallel implementation likewise those of the neighbor prefetching, and thus are not inefficient. The value *lastdirection* is set to 1 at the first reference of the sequence, then it increases spontaneously throughout the prefetching stages of the sequence.

Therefore, for each reference of a sequence to a same block, the 8-step neighbor prefetching performs a number N of lookups (with N ranging between 0 and 8) until a miss occurs; the number of lookups in the same block cannot trespass the number of 8, like in optimized neighbor prefetching.

Moreover, the 8-step algorithm performs at most one prefetch for each reference, thus avoiding long prefetching latencies.

3. Multimedia benchmark

Performance of prefetching schemes in multimedia image processing can be influenced by several factors, such as the program type and the input images. The program type is the most critical factor since the type of locality embedded in the program is the main cause of success of a prefetching technique. For this reason, we focused on a realistic application including video decompression and image manipulation, while at the same time covering a reasonably large spectrum of memory address schemes from commonly used algorithms. A possible data flow is reported in Fig.1. The video decompression part involves on-line decompression of an MPEG-2 stream, and image manipulation is performed on the decompressed frames in order to extract a minimum set of geometrical properties from objects (in Fig. 1, an example is reported where visual objects are extracted from a frame sequence in order to provide an MPEG-4-like coding of visual information). The algorithms considered are: a) image convolution, b) chain code computation, c) MPEG-2 decoding.

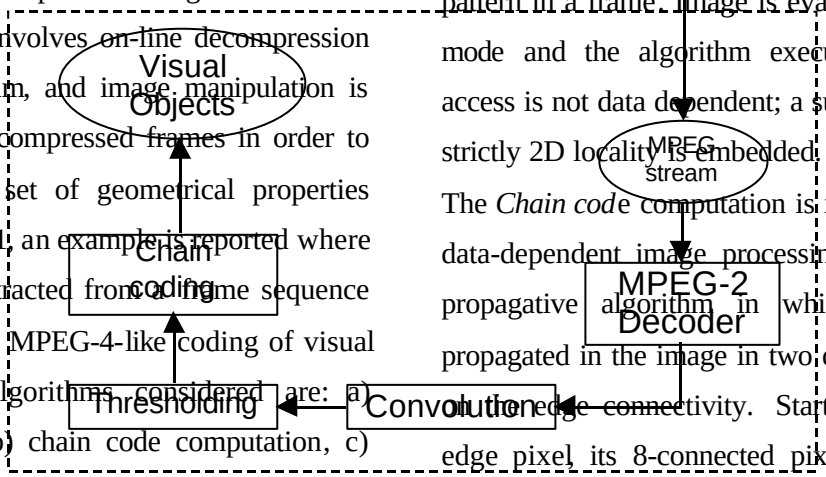

Fig. 1. Example of the data flow in video decompression and analysis.

Image convolution is the basic algorithm for image processing; it consists of processing each image pixel by convolving pixels of its bidimensional neighborhood with a coefficient mask. In many papers addressing performance evaluation in image processing [2,10,12], convolution is included in the basic benchmark, since it is very common. Examples of programs based on convolution are filtering for noise cleaning, image enhancement, edge detection and template matching for searching a known pattern in a frame. Image is evaluated in raster-scan mode and the algorithm execution and the data access is not data dependent; a substantial amount of strictly 2D locality is embedded.

The *Chain code* computation is included as a typical data-dependent image processing program. It is a propagative algorithm in which data access is propagated in the image in two directions depending on the edge connectivity. Starting from an initial edge pixel, its 8-connected pixels are checked: if there is an edge pixel, it is linked to the edge chain and a defined code is computed for storing the

direction change of the edge; then, the previous step is repeated from the linked pixel until the initial pixel is reached [13]. This algorithm has been included as an example of image computation that potentially does not benefit from standard cache techniques since it exhibits unusual and non-predictable spatial locality. Since computation is data dependent, prefetching performance varies with the input image. Chain codes are useful for describing object edges in image compression and image description; they are possible shapes coding techniques for MPEG-4.

Finally, *MPEG-2* decoding is the typical benchmark for multimedia processors, since it is currently the most spread multimedia program [1,3,4,12]. MPEG-2 decoding consists of several steps, but the most interesting aspect for data access is the inverse DCT for computing original frames that operates on image blocks (8 x 8 pixels) or macroblocks (16 x 16 pixels), thus exhibiting a strong 2D data locality. MPEG-2 decoding typically carries a high number of cache misses [4,12]: obviously, a large part are compulsory misses, if no prefetching is used; moreover, due to the relatively large mask size, the large 2D locality causes a number of cache conflicts as well. The number of misses depends on the image format and compression; for this reason, results over many different examples of MPEG videos have been compared.

4. Temporal Analysis

In this section we analyze the temporal performance of the different prefetching techniques. Preliminarily, in the first sub-section the simulation tools used to obtain time and efficacy results are described. The second sub-section describes in detail the temporal model used for the simulations.

4.1. The simulation environment

In this work we used trace-driven simulation to evaluate cache performance. The simulation environment is reported in Fig. 2.

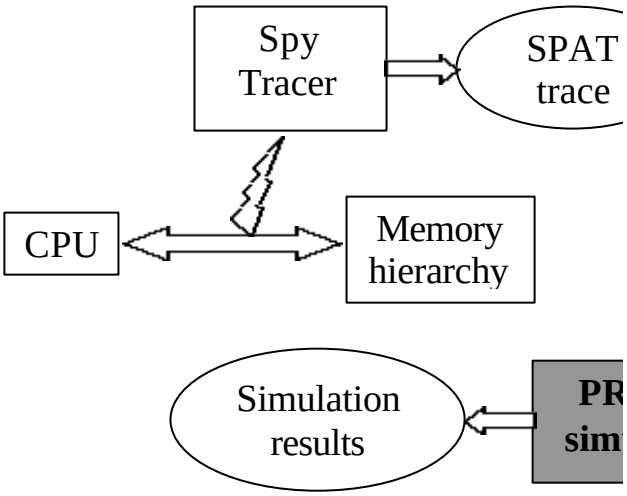

Fig. 2. The simulation environment.

Simulations have been performed by collecting all traces of program execution on the Sparc RISC processor using a tracer [14] able to trap signals in order to catch all memory references at *run time*. Traces are compressed on-the-fly into a compressed trace format (size of trace files is a critical issue), and then processed by using our simulator, called **PRIMA** (**Pr**efetching **Im**proves **M**ultimedia **A**pplications). The simulator is written in C++ running on a Unix platform, and is evolved from ACME [15]: we integrated a support for dealing with prefetching and 2D caches (PRIMA 1.0); then

we extended it to perform temporal simulation (PRIMA 2.0; see http://www.dsi.unimo.it/staff/st36/imagelab/prima.html for further details).

The simulator processes each trace entry, detecting the type of memory reference (instruction fetch or data read/write) and the reference address; the simulator manages the reference sequence, by collecting all statistics required. The system is fully expandable with other prefetching techniques and statistics.

4.2. The temporal model

In a previous work [3], we have shown the reduction in terms of number of misses achievable with the neighbor prefetching technique. In this paragraph, we describe the temporal model used to validate such results by means of temporal analysis. Our temporal simulation is based on a reference memory architecture with split-caches [16]. A specific cache for image data (named *2D cache*) as well as the standard data cache (named *Scalar cache*) is considered with the following assumptions:

- the 2D cache doesn't allow multiple access, i.e. only one miss or prefetch can be sustained at a time;
- *Misses* are *blocking* (or synchronizing) *events* for execution, meaning that when a miss happens, execution must wait the miss line fill completion before starting again. Moreover, any prefetching activity scheduled is completed before serving the miss. Hence, if the prefetching completion loads in the cache the data required by the miss, no miss line fill is then issued. Thus *Lookahead miss inquire* is defined, i.e. when a miss occurs, current prefetches queue is tested and a line fill is performed only if any prefetch is loading data which cause the miss (e.g. the * labeled line fill in Fig. 3 is not performed if the data is already loaded by prefetch).
- Instead, *prefetches* are obviously *non-blocking event*s for execution, i.e. can be accomplished in parallel with execution.
- Access times in the cache: T_{HIT} = 1 Tck and
- T_{MISS} = 8 Tck to for line fill (this miss penalty is realistically assumed independent from line size, as in [18]). The lookup time is assumed null, since lookup can be done in parallel with other activities (see [18]).

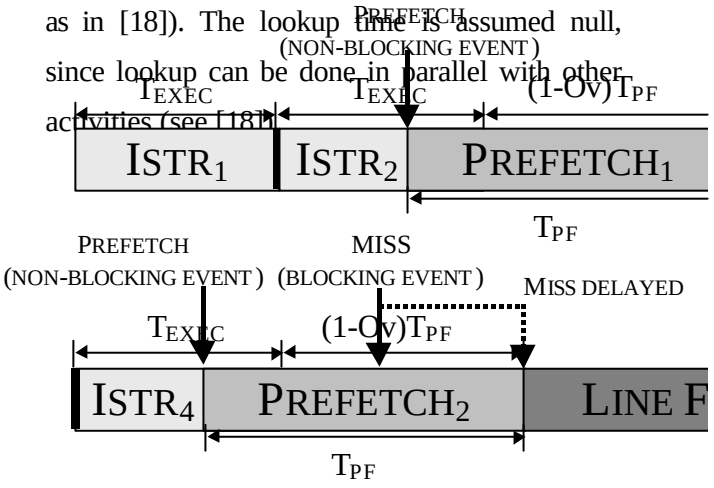

Fig. 3. Temporal model behavior.

Thus considering miss as blocking event and prefetch as non-blocking event, with reference to Fig.3, we consider four possible situations. We call T_{EXEC} the basic instruction execution time for completing a register-register instruction or a hit memory access (as for $ISTR_1$ in Fig3). If the

instruction (as $ISTR_2$) is partially overlapped to a prefetch issued by itself or pending from previous issuing, the execution time is T_{EXEC} + ((1-Ov)T_{PF}). Ov is the percentage of T_{PF} covered by overlapping. If the instruction causes a miss (as $ISTR_3$), we have an execution time T_{EXEC} + T_{MISS}. Finally, if instructions are partially overlapped to a prefetch and cause a miss (as $ISTR_4$), we can have two different situations: execution time is given by T_{EXEC} + (1-Ov)T_{PF} + T_{MISS}, or T_{EXEC} + (1-Ov)T_{PF} only, if the reference is in the prefetching queue.

This reference model can be used for defining the performance model in terms of improvement both in memory miss number and memory access time.

5. Performance results

Reduction of the miss number is an important parameter for estimating the efficacy of a prefetching method. Performances are computed with respect to the case without prefetching, on a same cache architecture. If we aim to observe only the miss reduction, an expressive measure is the percentage of miss reduction, called *efficacy* of the i-th prefetching method in [3], defined as the ratio

$$h_i = (N_{MISS} - N_{MISS\_PFi}) / N_{MISS} \quad (1)$$

Efficacy ranges between 0 and 1 (could be even negative in case of ineffective prefetching) and tends to 1 when prefetching achieves high performance, that is its N_{MISS_PFi} (i.e. the number of non-eliminated misses) tends towards 0.

Table 2 and Table 3 show, respectively, the number of misses and the efficacy results for the four considered prefetching techniques on the test programs, for a 2-way set-associative cache of 64 KB with 32 bytes per lines. These results confirm that exploiting horizontal spatial locality only (as in OBL) is not an effective solution since the number of miss with OBL prefetch remains not negligible. Instead, while good results are achieved by adaptive techniques, the best performance is reached by the neighbor-based techniques, which efficacy very close to 1. For a more detailed analysis of neighbor (not optimized) technique refer to [3].

	Convolution 5x5	**Chain Mpeg**	**MJPEG Frisco**	**MJPEG Hulahoop**	**MJPEG MJ**	**MJPEG Pirates**	
No prefetching	16370	4646	15924	12738	44374	81018	
OBL	6	428	3951	3154	11001	20074	
Adaptive	32	23	990	773	2840	5213	
Opt. neighbor	2	2	35	30	85	150	
8-step neighbor	2	3	35	30	85	150	

Table 2. Number of misses.

	Convolution 5x5	**Chain Mpeg**	**MJPEG Frisco**	**MJPEG Hulahoop**	**MJPEG MJ**	**MPEG Pirates**	**F**
OBL	99,9633%	90,7878%	75,1884%	75,2394%	75,2085%	75,2228%	87
Adaptive	99,8045%	99,5050%	93,7830%	93,9315%	93,5999%	93,5656%	89
Opt. Neighbor	99,9878%	99,9570%	99,7802%	99,7645%	99,8084%	99,8149%	96
8-step neighbor	99,9878%	99,9354%	99,7802%	99,7645%	99,8084%	99,8149%	95

Table 3. Efficacy h_i (in percentage).

Convolution 5x5	**Chain Mpeg**	**MJPEG Frisco**	**MJPEG Hulahoop**	**MJPEG MJ**	**MPEG Pirates**	**M Fl**
OBL 99,9633%	90,7878%	75,1884%	75,2394%	75,2085%	75,2228%	**85,**

9,8045%	**78,3981%**	93,7830%	93,9315%	93,5999%	93,5656%	**84,1505%**	**86,3875%**
9,9878%	**99,8466%**	99,7802%	99,7645%	99,8084%	99,8149%	**86,8022%**	**82,5357%**
9,9878%	99,9354%	99,7802%	99,7645%	99,8084%	99,8149%	**93,5526%**	**90,2296%**

Table 4. Time-based efficacy

However the miss number is not sufficient, since prefetching schedules many other bus transfers due to prefetching issues and if the prefetch number is too high, or if prefetch is not enough overlapped with instruction execution (since is a non-blocking event), the gain achieved by the miss reduction is lost by the prefetching costs [8].

Thus, we measure not only the number of misses but also the *Memory Access Delay Time* **MADT**, as proposed in [17]. We refer to accesses to 2D data cache only since we simulate prefetching only on this memory.

In [17], MADT is used to denote the average delayed cycle time incurred to access the memory for instructions other than cache hits during the program execution. MADT is the ratio between the total amount of memory delay accessing instructions and the number of memory reference instructions executed (N_{REF}). The effective access time for a 2D memory instruction is equal to the value of T_{EXEC}+MADT.

In absence of prefetching, this quantity is:

$$\mathbf{MADT_{NO\_PF}} = (N_{MISS}\ T_{MISS}) / N_{REF} \quad (2)$$

In the case of prefetching (let us considered i^{th} prefetch technique and $T_{PFi}=T_{MISS}$) this becomes:

$$\mathbf{MADT_{PFi}} = (N_{MISS\_PFi}\ T_{MISS} + (N_{PFi} - OV_{PFi})\ T_{MISS}) / N_{REF} \quad (3)$$

In this case, N_{MISS_PFi} accounts for the effectively issued memory accesses due to miss (i.e., excluding the ones attempted but not issued due to the lookahead miss inquire). N_{PFi} is the total number of issued prefetches while $OV_{PFi}T_{MISS}$ is the total amount of time gained by the overlap between prefetches and instructions execution (is the sum of all OV T_{PF} contributes shown in Fig.3 for a single instruction).

N_{MISS_PFi} is function of the cache size, block size, prefetching type and tested program (and data too, if the computation is data-dependent). N_{PFi} is a parameter of the prefetching techniques: in OBL and adaptive techniques is at maximum given by the number of memory reference ($N_{PFi} \leq N_{REF}$ since the prefetches issued are less than the prefetch attempted; note that we are considering prefetching *on reference*). Instead, in case of neighbor-based prefetching, as discussed in section 2, this number could be even higher than N_{REF} and can not be statically defined. The $OV_{PFi}\ T_{MISS}$ quantity is a time measure depending on the memory architecture too. All parameters has been measured on the selected benchmark by the PRIMA simulator with the temporal analysis, known the instruction execution time, the memory access time and the temporal sequence of blocking and non blocking events on 2D cache.

The overlap measure is a parameter often used for evaluating the efficiency of prefetching. For instance

in [18], Hsu and Smith (analyzing the performance for the instruction cache) define the same measure as prefetch efficiency *PE*: for each prefetched line, they keep track of the number of instructions N_{INSTR} between the time when a prefetch was started and the time the line was first referenced. If the prefetch is completely hidden by the N_{INSTR} instruction, the PE is set to 1, otherwise is set to $1.0 - \frac{T_{REMAINING}}{T_{MISS}}$ where $T_{REMAINING}$ is the time not overlapped.

Therefore we aim at measuring the efficacy of the prefetching not only in terms of miss number as in (1), but in terms of MADT. We define this time-base efficacy as

$$h^T i = (MADT_{NO\_PF} - MADT_{PFi}) / MADT_{NO\_PF} \quad (4)$$

Results are in Table 4 and they are very similar to the ones of Table 3, but differs from the cost of prefetching as non-overlapped memory access. In fact, by substituting in Eq. (4) the definition of Eq. (2) and (3), we have that

$$h^T i = h_i - ((N_{PFi} - OV_{PFi}) / N_{MISS}) \quad (5)$$

The second term is the cost of prefetching for the non-overlapped memory operations. By comparing Tables 3 and 4, we can note that in some tests the prefetching memory accesses are totally hidden (and $h^T i = h_i$), e.g. in the case of convolution where the data re-use is wide. Instead in most of the cases this cost is not negligible both for adaptive methods and for neighbor ones. These cases are reported in bold in Tables 4 where it is possible to note, for example in the last two columns, that prefetching cost heavily decreases the efficacy for the basic neighbor technique (and for adaptive too), but is negligible for the 8-step neighbor. For this reason, amongst the prefetching algorithms compared, neighbor prefetching exhibits the best performance: the explicit exploitation of the 2D data locality allows a better prediction of the data to be prefecthed, together with a good timeliness of the prefetching activity. In general, the 8-step version seems more effective in scheduling the prefetching activity.

6. Conclusions

This paper has proposed a temporal analysis of different cache prefecthing techniques on a multimedia benchmark. The benchmark consists of a set of image processing programs characterized by different memory access schemes to 2D data, including both raster-scan and data-dependent accesses.

FlowerG.img	**4 x 8K**	**8 x 16 K**	**16 x 32 K**	**32 x 64 K**	**64 x 128 K**
No Prefetching	60633	28813	13576	4646	1321
OBL	9933	4060	2057	428	2
Adaptive	747	236	86	23	**1**
Opt. neighbor	**5**	**4**	**6**	**2**	**1**
8-step neighbor	41	9	8	3	**1**

Table 5. Number of misses for chain code with different cache sizes.

Amongst the prefetching techniques compared, neighbor prefetching (originally proposed in [3]) proved the most effective in terms of miss elimination, even for caches of limited size such as those typical of low-cost multimedia processors, as reported in Table 5 that shows number of misses for variable cache sizes and line sizes in the case of chain code. In addition, the 8-step version of neighbor prefetching achieved the maximum reduction of execution time, by effectively distributing the prefetching activity over time and hiding the prefetching latency in overlap with the execution activity. This result is not obvious, since prefetching can even worsen the memory access time, in the case that the prefetching activity cannot be completely accomplished in overlap with execution.

These results qualify neighbor prefetching as an effective solution for prefetching of 2D data in image processing programs for multimedia.

References

[1] Kuroda, I. and Niscitani, T., "Multimedia processors", Proceedings of IEEE, v. 86, n. 6, 1998, pp. 1203-1221.

[2] Cucchiara, R. and Piccardi, M. and Prati, A., "Exploiting Cache in Multimedia", Proc. of IEEE International Conf. on Multimedia Computing and Systems (ICMCS 99), v. 1, 1999.

[3] Cucchiara, R. and Piccardi, M. and Prati, A., "Hardware Prefetching Techniques for Cache Memories in Multimedia Applications", Proc. of International Workshop on Computer Architectures for Machine Perception (CAMP 2000).

[4] Zucker, D. and Flynn, M.J. and Lee, R., "A comparison of hardware prefetching techniques for multimedia benchmark", Proc. of IEEE Multimedia 96, pp. 236-244.

[5] Pimentel, A.D. and Hertberger, L.O. and Struik, P. and Van Der Wolf, P., "Hardware versus hybrid data prefetching in multimedia processors", Proc. of 20th IEEE International Performance, Computing and Communications Conference (IPCCC 99), pp. 525-531

[6] Chen, T.F. and Baer, J.L., "A performance study of hardware and software data prefetching schemes", Proc. of 21st Int. Symp. Computer Architecture (ISCA '94), pp. 223-232.

[7] Smith, A.J., "Cache memories", Computing Surveys, v. 14, n. 3, 1982, pp. 473-530.

[8] Tse, J. and Smith, A.J., "CPU cache prefetching: timing evaluation of hardware implementation", IEEE Trans. on Computer, v. 47, n. 5, 1995, pp. 509-526.

[9] Chen, T.F. and Baer, J.L., "Effective hardware-based data prefetching for high-performance processors", IEEE Trans. on Computer, v. 44, n. 5, 1995, pp. 609-623.

[10] Cucchiara, R. and Piccardi, M., "Exploiting Image Processing Locality in Cache Pre-fetching", Proc. of 5th International Conference on High Performance Computing (HiPC '98).

[11] Eickemeyer, R. J. and Vassiliadis, S., "A load instruction unit for pipelined processor", IBM Journal of Research and Development, vol. 37, pp. 547-564, July 1993.

[12] Soderquist, P. and Leeser, M., "Memory traffic and data cache behavior of an MPEG-2 software decoder", preprint for ICCD '97, 1997.

[13] Kanedo, T. and Okudaira, M., "Encoding of arbitrary curves based on the chain code representation", IEEE Trans. on Communication, vol. COM-33, pp. 697-707, 1985.

[14] Irlam, G., "Spa", personal communication, 1992.

[15] Hunt, B., Acme cache simulator, http://tracebase.nmsu.edu/acme/acs.html, 1996.

[16] Milutinovic, V. and Markovic, B. and Tomasevic, M. and Tremblay, M., "The split temporal/spatial cache: initial performance

analysis", Proc. of the SCIzzL-5, Santa Clara, CA, USA, 1996

[17] Park, G. and Kwon, O. and Han, T. and Kim, S. and Yang, S., "An Improved Lookahead Instruction Prefetching", Proc. of the High-Performance Computing on the Information Superhighway, HPC-Asia '97, pp. 712-715

[18] Hsu, W. and Smith, J.E., "A performance study of instruction cache performance methods", IEEE Trans. On Computers, v. 47, n. 5, May 1998, pp. 497-508