Data-type Dependent Cache Prefetching for MPEG Applications

R. Cucchiara and A. Prati

Dipartimento di Ingegneria dell'Informazione
Università di Modena e Reggio Emilia
Modena, Italy

M. Piccardi

Department of Computer Systems
University of Technology
Sydney, Australia

Abstract

Data cache prefetching is an effective technique to improve performance of cache memories, whenever the prefetching algorithm is able to correctly predict useful data to be prefetched. To this aim, adequate information on the program's data locality must be used by the prefetching algorithm. In particular, multimedia applications are characterized by a substantial amount of image and video processing, which exhibits spatial locality in both the dimensions of the 2D data structures used for images and frames. However, in multimedia programs many memory references are made also to non-image data, characterized by standard spatial locality. In this work, we explore the adoption of different prefetching techniques in dependence of the data type (i.e., image and non-image), thus making it possible to tune the prefetching algorithms to the different forms of locality, and achieving overall performance optimization. In order to prevent interference between the two different data types, a split cache with two separated caches for image and non-image data is also evaluated as an alternative to a standard unified cache. Results on a multimedia workload (MPEG-2 and MPEG-4 decoders) show that standard prefetching techniques such as One-block-lookahead and the Stride Prediction Table are effective for standard data, while novel 2D prefetching techniques perform best on image data. In addition, at a parity of size, unified caches offer in general better performance that split caches, thank to the more flexible allocation of a unified cache space.

1 Introduction

Multimedia computing is a field of increasing importance in IT, up to the point of strongly influencing the design of modern computers. Actually, multimedia processing is spreading not only in strictly multimedia applications such as videoconferencing and virtual environments, but also in many general-purpose applications added with multimedia capabilities such as, for instance, Internet browsers.

MPEG standards played a prime role in the developement of multimedia applications, since they allowed interoperability between different platforms and manufacturers of multimedia components. Amongst MPEG standards, the two most important are MPEG-2 and MPEG-4: MPEG-2 defines compressed video formats over a large variety of frame sizes and rates (covering also the MPEG-1 standard), used massively for movies distribution on the Internet, and for DVDs; the spreading standard MPEG-4 defines instead a more ambitious range of applications including higher compression rates, object-based encoding, interactive multimedia and interactive graphics.

MPEG processing consists mainly of image sequence manipulation; therefore, large image data structures need to be allocated and frequently accessed in MPEG programs. Since images are processed in square pixel blocks (8x8 or 16x16 in size), a relevant *2D spatial locality* dominates memory access to image data. Nevertheless, profiling the memory reference traces shows that many other memory accesses are made to non-image data, interleaved with image accesses. Non-image data are mainly generic data structures which undergo standard (1D) spatial locality. As a consequence, data type exhibiting different spatial locality are mixed in MPEG computation.

Amongst the many techniques proposed in the literature for optimizing memory performance, cache prefetching has been proven one of the most effective [15][19][18]. Many recent works on cache prefetching address in particular multimedia workloads, due to the increasing spread of multimedia computing [19][20][4], proving efficacy of cache prefetching also in this area.

However, none of these proposals has explicitly given evidence to the different form of spatial locality dominating image and non-image memory references. In this paper, we aim to demonstrate that adopting different cache prefetching techniques for different data types can result in a (further) performance

improvement.

To this aim, we propose to use different prefetching techniques for image and non-image data, in order to exploit the 2D spatial locality of the former while preserving the standard locality of the latter. In the rest of the paper we first outline related works (section 2) and the multimedia workload considered (section 3). In section 4 prefetching techniques are described and, in particular, a technique oriented to 2D spatial locality (named neighbor prefetching) is depicted. Section 5 describes results comparing data-dependent prefetching and standard prefetching in terms of memory access time (saved) and speedup, and finally discusses different approaches with unified and split caches.

1.1 Related Works

Cache prefetching has been widely explored to improve effectiveness of cache memories, based on a variety of hardware and software techniques [18][20][4][12]. In the following, we address in particular hardware prefetching techniques rather than software ones since these techniques are less invasive in compilers' design and can take advantage of run-time information [19][17][2]; the larger amount of hardware functionalities required seems not to be a problem anymore, thank to impressive advances in circuit integration technology.

The basic prefetching method is known as One-Block-Lookahead (OBL) [15] (also called *always prefetch* in a more recent work from Tse and Smith [17]). With OBL, every time a reference to block i is made, a lookup in the cache is issued for block i + 1; if the block is absent, it is prefetched. The assumption of OBL is that, thank to the spatial locality, block i + 1 has a high probability to be referenced in the near future, and that scheduling its prefetching on the first reference to block i grants adequate timeliness to its load in the cache. Based on similar assumptions of lookahead locality, other more recent proposals have been presented. Stream buffers [19][9][13] prefetch one or more blocks following block i in another level of the memory hierarchy, i.e. the stream buffer. These assumptions are correct, and testified by many simulations, especially when arrays are accessed. They hold for image data too, when pixels of an image or a video frame are evaluated and processed one by one in raster-scan mode (i.e., row-by-row). Nevertheless, when pixels are processed in a data-dependent order such as in MPEG processing, prefetching performance can be improved with respect to lookahead-only prefetching, since locality in the image vertical direction can also be exploited. In contrast with lookahead techniques, other prefetching techniques adapt the block reference prevision to recent references history, achieving effective results for programs with regular memory access schemes. The most basic prefetching algorithm computes the stride, i.e. the address difference between the last two memory references made by a same instruction, and adds it to the address of the last memory reference in order to obtain block prevision. Fu and Patel in [5] proposed the use of an associative memory, called the Stride Prediction Table (SPT), to store stride information. Chen and Baer in [2] proposed the use of a Reference Prediction Table (RPT), similar to the SPT, but with the addition of a state machine to determine if the prediction can be trusted (correct) or not (incorrect). Other proposals combine static and adaptive block reference prevision in order to receive benefits from both the approaches: in [14], a stride predictor is combined with a stream buffer and a voting scheme is used to decide whether the stream buffer or the stride predictor must be used. Eventually, other authors proposed more complicated forms of prevision adaptivity [8], but we judge them to be too demanding in hardware resources and thus still infeasible to be implemented in practice.

In many papers, general-purpose benchmarks have been used for performance evaluation, but, recently, cache prefetching techniques have been evaluated also for some multimedia programs, since they are becoming more and more reference applications [20][12][10][1][3]. In [19] and [20], Zucker *et al.* addressed an MPEG-2 benchmark, proving that the use of an SPT and a stream cache achieves very good performance for a wide range of cache parameters. In [12], Pimentel *et al.* evaluated performance of stride prefetching and hybrid prefetching (with stream prefetch instructions, SPT and OBL) for frame deinterlacing with a multimedia processor. Stride prediction is an attractive solution, since many multimedia image processing programs are regular. Nevertheless, some 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 (visual objects are used in MPEG-4). In these cases, even stride-based techniques are not adequate: as we will show in this paper, their performance can be outperformed by some specific techniques optimized for prefetching image data.

Commonly, programs are characterized by different forms of data locality, and varying the caching strategies for the different data types can result in

substantial performance improvement. In [6], Gonzalez *et al.* proposed the use of a split cache for data with different locality. Similarly, in [11] Milutinovic *et al.* proposed the use of a split cache for data with temporal locality only and with spatio-temporal locality. In this work, we further advance this idea to multimedia programs by proposing the use of a single, unified data cache and different prefetching strategies for image data, exhibiting strong 2D spatial locality, and other, non-image data. Furthermore, a performance comparison between the unified cache and a split cache is analyzed in detail.

2 Description of the MPEG workload

MPEG standards are widely adopted in multimedia for image and video encoding, and are therefore a suitable benchmark for performance evaluation of multimedia programs. In this work, we analyze cache prefetching performance on MPEG-2 and MPEG-4 decoders, since these programs are spread in use nowadays for movie and video playing.

MPEG-2 is targeted to video compression of videos with variable frame rate and size. MPEG-2 decoding consists of several steps, of which the two computationally heaviest are motion compensation and IDCT. These steps operate on image blocks (8 x 8 pixels) and macroblocks (16 x 16 pixels), thus exhibiting a strong 2D data locality. MPEG-2 decoding typically carries a significant number of cache misses [19][16], obviously depending on the frame size and cache parameters. A large part of these misses are compulsory, if no prefetching is used; however, the large 2D locality causes a number of conflict misses as well. The number of misses depends also on the video sequence content (weakly) and on the actual decoding software (strongly): in the experiments, we have used a typical test sequence ("Flower Garden"), with 61 frames of 352 x 288 pixels; the software used is the MPEG-2 video decoder from the MSSG (MPEG Software Simulation Group).

MPEG-4 is a much broader standard than MPEG-2, aiming to support not only video compression but also interactive multimedia and interactive graphics in a large range of bit-rates and qualities. MPEG-4 has introduced the concept of units of audio, visual or audiovisual content, called "media objects". These media objects can be of natural or synthetic origin, are coded separately, and can be described, multiplexed, synchronized and interacted with inside a single scene. Therefore, the complexity of an MPEG-4 decoder is intrinsically much higher than that of an MPEG-2 one. This is confirmed by our tests, where the number of memory references at a parity of frame size proves at least one order of magnitude greater. Obviously, this ratio depends much on the software implementation of decoders and can vary with optimization. In the experiments on MPEG-4, we have used a typical test sequence ("Dancer"), with 250 frames of 352 x 288 pixels; the software used is the MoMuSys MPEG-4 video decoder from the ISO/UNI reference implementation. For both decoders, simulations have been performed by collecting all traces of program execution on a Sparc RISC processor using a tracer [7] able to trace all run-time memory references. 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* (PRefetching Improves Multimedia Applications). The simulator is written in C++ running on a Unix platform, providing support for prefetching, split data caches, and accurate execution time analysis. Due to the large number of memory references, results for the MPEG-4 test sequence have been reported for a shorter sequence of four frames.

3 Data prefetching strategies for multimedia programs

In this paper, we propose a method to improve performance of cache prefetching on the described MPEG benchmark. The main idea is to differentiate the prefetching techniques used for the image data from those used for non-image data. To this aim, in this section we first describe how the compiler can identify image data and transmit this information to the prefetching unit; then we briefly describe the main standard prefetching techniques, pointing out their limitations in exploiting the 2D spatial locality of image data; eventually, a technique able to overcome these limitations and better exploit the 2D spatial locality is proposed.

3.1 The interaction between the compiler and the prefetching unit

The compiler must qualify each memory reference as either addressing image or non-image data, and for each image reference it must provide the prefetching unit with the image row size in bytes. In general, the compiler can extract this information from variable declarations. Although it might be possible for a programmer to use 1-D array variables to store 2-D images, this ought to be considered bad programming style in modern programming languages, such as C, C++, and Java. Therefore, the requested information can in general be extracted by analysis of variable type at declaration.

The extracted information must then be transferred to the prefetching mechanism: this can be done in

different ways, depending whether prefetching is implemented in software or in hardware. In a software implementation, it would be straightforward to insert the prefetch instructions immediately after the image references. However, a software implementation carries the following limitations: first, for data-dependent algorithms, it is possible to compute the memory address of prefetch references only at run-time, and this will slow down the prefetching mechanism; second, if the prefetching policy varies in dependence of the image reference, extra instructions will be needed to differentiate the prefetching policy, carrying further delays which may compromise the overall effectivenes of prefetching, in particular for fast L1 caches. Instead, we propose to implement the prefetching unit in hardware in the following way: every time an image variable is allocated, its address range and row size are stored in a dedicated table in memory by way a procedure call (please notice that this event is rare, reasonably in the order of tens per program execution). At each memory reference, the memory address is compared with those stored in the table, in order to state whether this is an image reference or not, and its row size extracted.

3.2 Standard prefetching techniques

The standard prefetching techniques we used in the experiments to the purpose of comparison are OBL and stride prefetching based on the SPT (SPT for short in the following). OBL is based on a static address prevision, while SPT exploits address prevision adaptation. In OBL prefetching, every time a memory reference is made, a lookup in the cache is performed for the block that follows the currently referenced one in memory. If we call A0 the current address and B(A0) the current *block address* (i.e. the memory address without the LSBs encoding the byte offset inside the block), the block address of the looked-up block is B(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 block address of the looked-up block is B(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 partially hold in typical image processing algorithms for multimedia, but do not consider explicitly the vertical locality, which is instead relevant, as proven by the experimental results.

3.3 Neighbor prefetching

In this paper we propose to exploit the 2D spatial locality on image data by neighbor prefetching. By this technique, cache blocks neighboring the currently referenced block (called 8-connected blocks) are checked for presence in the cache, and prefetched if required. The block addresses of the looked-up blocks with reference to the current address A0 are shown in Table 1 (NB_{row} is the number of bytes of an image row). This technique mirrors the way data are accessed by many image processing algorithms, including both raster-scan and data-dependent ones, thus promising a general improvement in 2D spatial locality's exploitation.

	Curr. block address	**Prefetch address**
No pref.	$B(A_0)$	-
OBL	$B(A_0)$	$B(A_0)+1$
SPT	$B(A_0)$	$B(A_0+S)$
Neighbor	$B(A_0)$	$B(A_0)+1$, $B(A_0)-1$, $B(A_0+NB_{row})$, $B(A_0+NB_{row})+1$, $B(A_0+NB_{row})-1$, $B(A_0-NB_{row})$, $B(A_0-NB_{row})+1$, $B(A_0-NB_{row})-1$

Table 1: The prefetching techniques compared

However, looking up all the 8-connected blocks and potentially performing more than one prefetch at each memory reference may turn into a performance bottleneck. Therefore, two modifications are introduced: first, a highly efficient lookup mechanism, using a multiple-port tag directory and in pipelining with prefetching, limits the lookup overhead; second, the prefetching of blocks does not block execution, but is instead distributed along a sequence of memory references. Let us call block i one of the blocks neighboring the currently referenced one, with $i = 18$ (i equal to 1 for the forward direction and increasing in clockwise order), and let us call *sequence* a sequence of memory references made consecutively to a same cache block.

Fig. 1 reports the algorithm which is executed at each memory reference (please notice that there is no need for explicit initialization of the *lastdirection* variable, since at the first execution the "then" branch is always taken first). At each memory reference, the algorithm checks if the current reference is either the first or a subsequent reference in the sequence. If this is the first, block 1 is looked up (like in OBL), since it is the block with the highest static probability of access. If block 1 is absent, it is prefetched and the algorithm stops. Otherwise, the block counter i is incremented and a new lookup is performed, until one prefetch is actually issued, or all the 8-connected

blocks have been looked up. If this is a subsequent reference, lookups start from the last direction explored at the previous memory reference. In this way, all the 8-connected blocks will be looked up during the sequence and their prefetching accomplished if needed, but these activities will be distributed along the sequence itself, thus limiting access bottlenecks. The clockwise direction is a heuristic rule tuned to raster scan processing, where blocks 1 and 2 should be absent from the cache and thus their prefetching ought to be scheduled first.

```
{ if first_reference (A0 block) = TRUE then i ← 1;
    else i ← lastdirection + 1;
    hit ← TRUE;
    while (i ≤ 8 AND hit = TRUE) do
          { hit ← lookup (block i);
            if (hit = TRUE) then i ← i+1;
            else prefetch (i); }
    lastdirection ← i; }
```

Figure 1: The neighbor prefetching algorithm

The algorithm shown in Fig. 1 is given in a sequential form for the sake of clarity. However, while we still assume that prefetches can be issued only sequentially, parallel hardware implementation of lookups is allowed, which can be outlined as follows: multiple computation of block addresses and lookups can be performed in parallel via dedicated adders and the multi-ported tag directory (reasonably, from two to four). If one of these lookups will miss, the first block missing will be prefetched in the cache. If further lookups miss, their blocks will be prefetched on the next memory references; extra status bits are required to store the lookup hit/miss conditions and the computed block addresses. In case none of these lookups will miss, the next groups of lookups will be scheduled, and so on, if needed.

3.4 Combining prefetching for image and non-image references

Neighbor prefetching aims to exploit 2D spatial locality better than other existing prefetching techniques. However, being tailored to 2D image data, it might only casually match the locality of non-image data. Therefore, differentiating the prefetching strategy for these different data types may achieve the best overall performance. To this aim, we propose to adopt a standard prefetching technique such OBL or SPT for non-image data, and the neighbour prefetching for image data. Results are reported in the next section, and compared with those achievable without prefetching and with a single prefetching technique for all data.

4 Performance analysis of cache prefetching

In this section, performance are reported in terms of the memory access delay time (MADT). MADT is the total amount of time due to delayed memory cycles, i.e., cache misses and prefetching completion, in case prefetching does not completely overlap with execution. MADT is defined as follows:

$$MADT_{PF_i} = N_{MISS\_PF_i} T_{MISS} + (N_{PF_i} - OV_{PF_i}) T_{PF} \quad (1)$$

In the equation, $N_{MISS\_PF_i}$ is the number of misses that occurs when using the i-th prefetching technique, and N_{PF_i} is the number of prefetches issued with this same i-th prefetching technique; T_{MISS} is the time required to serve a miss, and T_{PF} is the time required to complete a prefetch. Prefetches, in general, only partially overlap with execution; OV_{PF_i} represents the amount of overlap and is computed as follows: for each of the N_{PF_i} prefetches, the fraction ($\in [0,1]$) of prefetching time overlapped with execution is measured during the simulation, and then all fractions added up to obtain OV_{PF_i}. Consequently, OV_{PF_i} ranges between 0 (worst case) and N_{PF_i} (best case - complete overlap). Therefore, the total amount of time caused by prefetching can be expressed as $(N_{PF_i} - OV_{PF_i})T_{PF}$, ranging between 0 (best case) and $N_{PF_i}T_{PF}$ (worst case). The total amount of time caused by misses is simply given by $N_{MISS\_PF_i}T_{MISS}$.

MADT is an effective way to measure performance of cache memories, since it covers all aspects influencing cache performance, including actual system timings, while at the same excludes all those which do not relate with cache performance (non-ideality of the pipeline, time to execute hits and non-memory instructions,...); as a consequence of its definition, MADT is zero in the case of an ideal cache (hits only). The speedup with the i-th prefetching technique can be expressed as:

$$Speedup_{PF_i} = \frac{MADT_{no\_prefetching}}{MADT_{PF_i}} \quad (2)$$

First, we report results for image data only, in order to prove the effectiveness of neighbor prefetching. Then we show results for the combined image/non-image approach; the reference architecture is a 64 KB 2-way set-associative data cache with 32 bytes per block. Eventually we compare results with a 32 KB + 32 KB split cache of where image and non-image data are stored separately.

4.1 Performance results on the MPEG-2 decoder

As a first test, we analyze MADT results on a trace generated on MPEG-2 decoder for image data only. Misses on non-image data are omitted, assuming that non-image data always hit the cache. In this manner, the effective impact of the access to 2D data structure is accounted. In Table 2 results are reported (in 10^3 clock cycles) for a 32KB cache size. The MADT with "no prefetching" techniques accounts for miss penalty only.

	No pref.	**OBL**	**SPT**	**Neighbor**
MADT	5054	748	931	359
Speedup	1	6.75	5.42	14.07

Table 2: MPEG-2 decoder results. MADT (in 10^3 clock cycles) on image data only with a cache size of 32 KB (non-image data are assumed always hitting the cache)

The high speedup obtained with whichever prefetching techniques demonstrates that a good prevision of the future reference blocks allows a timely prefetching without pollution. Since MPEG-2 works basically with blocks of data with a regular access, both OBL and SPT techniques achieve good performance and exhibit similar speedups. Neighbor prefetching, however, outperforms the others with a MADT speedup of 14.07 on image data.

MPEG-2 decoder uses a large amount of non-image data too and therefore we simulated the cache behaviour on a unified cache of 64 KB including both types of data. In Table 3 we show the performance assuming data-type dependent cache prefetching.

Non-image Image	**No pref.**	**OBL**	**SPT**
No pref.	5166 (1)	5098 (1.01)	5104 (1.01)
OBL	1097 (4.71)	1002 (5.16)	1003 (5.15)
SPT	1083 (4.77)	956 (5.40)	955 (5.41)
Neighbor	914 (5.65)	679 (7.61)	678 (7.62)

Table 3: MPEG-2 decoder results. MADT (in 10^3 clock cycles) on image and non-image data with a cache size of 64 KB and different prefetching techniques for image and non-image data (values in brackets are the MADT speedups)

Without prefetching, the MADT for MPEG-2 is of $5166 \cdot 10^3$ clock cycles. Neighbor prefetching on non-image data has not been considered, since meaningless. Along the diagonal (starting from upper left corner) of Table 3 the performance achieved by using the same prefetching technique for all data are reported. SPT performs slightly better than the OBL technique. Moreover, when no prefetching or just standard prefetching methods are used on image data (first three rows on Table 3), adopting prefetching also on non-image data does not improve performance considerably.

On the contrary, adopting neighbor prefetching on image data does not only abate MADT, but achieves a better exploitation of OBL and SPT on the other data. By combining neighbor prefetching on image data with OBL (or SPT) on non-image data, MADT decreases from $5166 \cdot 10^3$ down to 679 (or 678) $\cdot 10^3$ clock cycles with a final speedup of 7.61 (or 7.62). This simulation is reasonably implementable in standard cache architecture, just providing a data-type dependent prefetching selector.

As final test, we simulated the presence of two separated caches, one for non-image data and one for image data, allowing parallel access. Table 4 compares unified versus split cache, reporting MADT values with different prefetching on image data and no prefetching or OBL prefetching for non-image data. Results with SPT on non-image data are omitted since are very similar to those shown in Table 4.

	No pref. on non-image		**OBL on non-image**	
	Unified	**Split**	**Unified**	**Split**
No pref.	5166	5552	5098	5506
OBL	1097	1245	1002	1199
SPT	1083	1428	956	1382
Neighbor	914	856	679	811

Table 4: MPEG-2 decoder results. MADT (in 10^3 clock cycles) on image and non-image data with a unified cache of 64 KB vs. split cache of 32KB+32KB. The prefetching technique used for non-image data is reported in the first row.

Table 4 shows that, without prefetching on non-image data, the split cache achieves the best performance if neighbor prefetching is adopted for image data. Nevertheless, on average, a unified cache outperforms split cache. This demonstrates that interference between different data types is less important than the possible advantages of a unified allocation space (note that we compared a 32KB + 32KB split cache with a 64 KB unified cache).

4.2 Performance results on the MPEG-4 decoder

MPEG-4 decoder is a more severe workload since simulations show that the MADT on MPEG-4 decoder is much heavier than on MPEG-2 decoder. In Tables 5 and 6, we report MADT for an MPEG-4 test sequence of 4 frames only versus a MPEG-2 test sequence of 61 frames. This is basically due to the huge amount of references that MPEG-4 decoder generates, producing large traces and a number of references comparable to the MPEG-2 decoder just with 4 frames only.

The initial miss rate and MADT for MPEG-4 is very high and it could benefit more of potential improvement offered by prefetching. The speedup with OBL and SPT on image data is less than in the case of MPEG-2 decoder, since the algorithm is more complex and the access for visual object manipulation is not regular. Instead, neighbor prefetching, in addition to being the best performing technique, achieves a very high speedup (38.81), abating MADT.

	No pref.	**OBL**	**SPT**	**Neighbor**
MADT	4842	1830	1169	125
Speedup	1	2.65	4.14	38.81

Table 5: MPEG-4 decoder results. MADT (in 10^3 clock cycles) on image data only with a cache size of 32 KB (non-image data are assumed always hitting the cache)

Non-image Image	**No pref.**	**OBL**	**SPT**
No pref.	10256 (1)	7151 (1.43)	6390 (1.60)
OBL	7324 (1.40)	4047 (2.53)	3352 (3.06)
SPT	6619 (1.55)	3388 (3.03)	2695 (3.81)
Neighbor	5763 (1.78)	2410 (4.26)	1716 (5.98)

Table 6: MPEG-4 decoder results. MADT (in 10^3 clock cycles) on image and non-image data with a cache size of 64 KB and different prefetching techniques for image and non-image data (values in brackets are the MADT speedups)

In Table 6, a unified cache of 64 KB is used to simulate the performance on all data of the MPEG-4 decoder. In this case, the SPT technique achieves better performance than OBL; in fact, a stride-based approach such as SPT has about half the MADT clock cycles of the OBL technique (referring to Table 6, SPT on both non-image and image data achieves a MADT of $2695 \cdot 10^3$, whereas the OBL's MADT is $4047 \cdot 10^3$).

In accordance with MPEG-2 results, adopting neighbor prefetching on image data with any other prefetching technique on non-image data achieves the best performance on the MPEG-4 program, too (neighbor with SPT is the best combination, achieving a MADT of $1716 \cdot 10^3$ clock cycles, with a speedup of 5.98). This proves that mixed techniques are the best solution for programs that have a variable behaviour with respect to the spatial locality on different data types.

Eventually, we report the comparison between the unified cache and two separated caches, one for each type of data. Table 7 shows the MADT results using no prefetching or OBL prefetching for non-image data.

	No pref. on non-image		**OBL on non-image**	
	Unified	**Split**	**Unified**	**Split**
No pref.	10256	10271	7151	7162
OBL	7324	7259	4047	4149
SPT	6619	6598	3388	3489
Neighbor	5763	5554	2410	2445

Table 7: MPEG-4 decoder results. MADT (in 10^3 clock cycles) on image and non-image data with a unified cache of 64 KB vs. split cache of 32KB+32KB. The prefetching technique used for non-image data is reported in the first row.

Unlike the comparison reported in Table 4 for the MPEG-2 decoder, in the case of MPEG-4, the split cache achieves better performance than the unified cache on average, if no prefetching is used on the non-image data (Table 7). This can be justified by the fact that the complexity and intensive memory traffic of the MPEG-4 algorithm results in a more frequent prefetching issuing. In this case replacements and pollution have a prime role in the overall cache performance and having the two caches separated reduces the interference. However, performance is only slightly affected by this behavior.

The same reasoning should be valid in the case of OBL prefetching for the non-image data (Table 7) but two considerations must be made. First, the number of non-image references in the MPEG-4 algorithm is very high, larger than the number of image references, thus influencing more the overall performance. Second, in the case of the split cache, the reduction of the number of misses given by OBL on non-image data implies a reduction in the time allowed to issue prefetches in the image cache. In fact, in our temporal assumptions, the prefetches and the misses in one cache add time available for the prefetching completion in the other cache. As a consequence, the prefetches in the image cache are less overlapped with execution due to

the decrease in the number of misses on the non-image data. Therefore, MADT for the split cache becomes greater than for the unified cache.

5 Conclusions

In this work, we reported several measures from simulations of data-type dependent prefetching techniques on a multimedia workload. Many tests have been performed, confirming that a speedup improvement can be achieved only by selecting an adequate prefetching technique with respect to the data access type.

In this work, we proved that advantages are maximized with a selective prefetching, adopting neighbor prefetching for images (i.e. 2D) and a different technique for non-image data. In conclusion, we can summarize that: (i) the proposed neighbor prefetching outperforms the other techniques on image data with a speedup up to 38.81 for the MPEG-4 decoder; (ii) considering the whole set of memory references (image and non-image) the best choice for image data is still the neighbor prefetching, using OBL or SPT for non-image data; (iii) the best performance is achieved with the combination neighbor-SPT.

From a hardware implementation point of view, the combination neighbor-SPT adds up the requirements of both techniques; instead, in the combination neighbor-OBL, the OBL prefetching could be implemented as a subset of the neighbor prefetching, thus making this implementation more viable.

The final consideration is that prefetching should always be adopted for multimedia workloads, since they exhibit high miss rates, which can be drastically reduced by prefetching. Thus, an efficient and flexible prefetching technique is essential to achieve maximum performance improvement.

References

[1] H. Chang, L. Chen, M. Hsu, and Y. Chang. Performance analysis and architecture evaluation of mpeg-4 video codec system. In *Proceedings of IEEE Intl. Symp. on Circuits and Systems*, volume 2, pages 449–452, 2000.

[2] T. Chen and J. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. Comput.*, 44(5):609–623, 1995.

[3] R. Cucchiara, M. Piccardi, and A. Prati. Exploiting cache in multimedia. In *Proc. of IEEE Intl. Conf. on Multimedia Computing and Systems (ICMCS)*, volume 1, 1999.

[4] R. Cucchiara, M. Piccardi, and A. Prati. Temporal analysis of cache prefetching strategies for multimedia applications. In *Proc. of IEEE Intl. Performance, Computing and Communications Conf. (IPCCC)*, pages 311–318, 2001.

[5] J. Fu and J. Patel. Data prefetching in multiprocessor vector cache memories. In *Proc. of IEEE/ACM Intl. Symp. on Computer Architecture (ISCA)*, pages 54–63, May 1991.

[6] A. Gonzales, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proc. of ACM Intl. Conf. on Supercomputing*, pages 338–347, 1995.

[7] G. Irlam. *Spa*. Personal Communication, 1992.

[8] D. Joseph and D. Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers, Special Issue on Cache Memory and Related Problems*, 48(2):121–134, Feb. 1999.

[9] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. of 17th Intl. Symp. on Computer Architecture (ISCA)*, pages 364–373, May 1990.

[10] J. Kneip, B. Schmale, and H. Moller. Applying and implementing the mpeg-4 multimedia standard. *IEEE Micro*, 19(6):64–74, 1999.

[11] V. Milutinovic, B. Markovic, M. Tomasevic, and M. Tremblay. The split temporal/spatial cache: initial performance analysis. In *Proc. of the SCIzzL-5, Santa Clara, CA, USA*, 1996.

[12] A. Pimentel, L. Hertberger, P. Struik, and P. Van Der Wolf. Hardware versus hybrid data prefetching in multimedia processors. In *Proc. of IEEE Intl. Performance, Computing and Communications Conf. (IPCCC)*, pages 525–531, 1999.

[13] P. Ranganathan, S. Adve, and N. Jouppi. Performance of image and video processing with general-purpose processors and media isa extensions. In *Proc. of IEEE/ACM Intl. Symp. on Computer Architecture (ISCA)*, pages 124–135, 1999.

[14] G. Singh Manku, M. Prasad, and D. Patterson. A new voting based hardware data prefetch scheme. In *Proc. of IEEE Intl. Conf. on High-Performance Computing*, pages 100–105, 1997.

[15] A. Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[16] P. Soderquist and M. Leeser. Memory traffic and data cache behavior of an mpeg-2 software decoder. In *Proceedings of IEEE Int'l Conf. on Comptuer Design (ICCD)*, pages 417–422, 1997.

[17] J. Tse and A. Smith. Cpu cache prefetching: timing evaluation of hardware implementation. *IEEE Computer*, 47(5):509–526, 1995.

[18] S. P. Vanderwiel and D. Lilja. Data prefetch mechanisms. *ACM Comput. Surv.*, 32(2):174–199, June 2000.

[19] D. Zucker, M. Flynn, and R. Lee. A comparison of hardware prefetching techniques for multimedia benchmark. In *Proc. of IEEE Multimedia 96*, pages 236–244, 1996.

[20] D. Zucker, R. Lee, and M. Flynn. Hardware and software cache prefetching techniques for mpeg benchmarks. *IEEE Transactions on Circuits and Systems for Video Technology*, 10(5):782 –796, 2000.