Data cache management on EPIC architecture : Optimizing memory access for image processing

K. Brifault, H-P. Charles
University of Versailles, PRiSM Laboratory, FRANCE
{kbrifa, hpc}@prism.uvsq.fr

Abstract

Nowadays, multimedia applications are more and more used, and take a larger place in the workloads of modern computing systems. It appears that the classical 1D spatial locality arrangement in the cache is not adapted to the management of this data type, because its structures exhibit an intrinsic 2D locality.
In this article, we study cache behavior and alternative strategies for multimedia, using a JPEG benchmark on an Itanium2™ cache system. We demonstrate, through systematic experiments, that performance can be very sensitive to data structure, revealing that cache organization has a major impact on performance. We then present the results of our experiments with suggestions to improve future media processing compiler design.

1 Introduction

Multimedia is a broad concept that covers audio, video, 2D and 3D graphics, animation, and image handling. Since MPEG-1/2 wide adoption and later with H.261/H.263 standards appearing, multimedia has become one of the main workloads for modern personal computing systems [3]. However, it represents a challenging design target for the industry, because of the complex processing, requiring higher and higher performance computers, which preserving low cost to meet consumer demand [10].
With the exponential growth in semiconductor technology these last two decades, and with the recent confluence of hardware and software technologies [2], the ability of computers has been rapidly arisen. The nature of most computer hardware has had to change in order to answer the needs of companies in new applications, such as image processing, medical imagery, realistic simulation, speech recognition and broadband communications [23]. A typical case of such evolution is the introduction on most architectures of specialized instructions to satisfy multimedia requirements (UltraSPARC™ VIS, Pentium® MMX™, HP MAX2...). These instruction sets have been created to take into account the common features of multimedia applications (data type, algorithm structure) and to speed up their processing. To illustrate, a register is not considered as an unique value of 32, 64 or 128-bit, but as a vector of 8, 16 or 32-bit components. In addition, graphical applications require a different arithmetic, the saturated arithmetic[1]. Finally, many standard scalar operations have been vectorized to reflect the fact that the same operation can be applied simultaneously on vectors. This last feature is extremely useful to exploit ILP[2] [8] present on modern microprocessors.
Unfortunately, all of these advanced mechanisms are not so easily exploited by software. In particular, current compilers have a very hard time to generate multimedia instructions. The most classical way of using multimedia hardware within a system is to rely on the use of media-processing libraries or rewrite key portions of the application in assembly language using the multimedia instructions or code, in a high level language, macros that make available the functionality of the media-processing primitives.
In addition, multimedia applications, due to the fairly large data sets they have to manipulate, are not escaping the classical problem of efficient use of memory hierarchies. De facto, it is perhaps one of the key performance limiting factor on many multimedia applications [9]. Therefore, the purpose of this work is to explore the cache behavior of multimedia applications such as JPEG. In this article, a typical multimedia/image processing ap-

[1] Value of each pixel component is limited in the 0 to 255 range, avoiding underflow/overflow.

[2] ILP: Instruction Level Parallelism is a family of processor and compiler design techniques that speed up execution by causing individual machine operations, such as memory loads and stores, integer additions and floating point multiplications, to execute in parallel.

plication (FDCT) has been selected and its behavior on an Itanium2™ architecture has been studied in depth. The goal of our study is not only to understand cache conflicts and compilation limitations, but to propose various techniques of optimization for image processing to implement in a dynamic code generator.
This paper is organized as follows. Section 2 gives a brief overview of our experimental setup, hardware platform as well as software platform. The next section describes our target codes. Section 4 details our methodology. In section 5, results on simplified kernels are presented allowing a clear isolation of performance problems. Finally, section 6 concludes the paper with a few remarks and presents some opportunities for future research.

2 Experimental Setup

2.1 Hardware Setup

The machine used in our experiments, a 900 MHz uniprocessor Itanium2™ with 2 GB of memory [12, 13]. Itanium2™ is an "in order" processor, which offers a wide degree of parallelism, as well as the possibility of executing six multimedia instructions at the same clock cycle, (in blocks of three instructions named bundles). On this machine, the cache memory hierarchy is organized in three levels, all of them being on chip. L1 level, 4-way associative, split between a data cache (16 KB) and an instruction cache (16 KB), using 64 B cache lines. Besides, theses caches can only be used for storing integer data. L2 level, unified, is a 8-way associative of 256 KB with a 128 B cache lines. L3 level, unified, 12-way associative, 1.5 MB, uses 128 B cache lines. Latencies (for the integer load/store instructions) of the various levels are given in the table below:

Cache	L1D	L2U	L3U
Latency (cycles)	2	≤ 6	≤ 13

2.2 Software environment

The test machine was running Linux IA-64 Red Hat 7.1 based on the 2.4.18 SMP kernel. Several compilers were used, but all the results are given for the Intel® C++ Compiler Version 7.1, with the combination of `-O3` and `-restrict`, which sets up optimizations such as software pipelining, prefetch instructions, precation and rotating registers. The `-restrict` option was essential in getting top performance, as it allowed the compiler to assume that distinct arrays were pointing to disjoint memory regions (aliasing problem).

To validate our experiments, we used a driver that we have developed, which allows to build a set of micro-benchmarks named `kernel`, written in C or assembly programming language. The driver is a toolkit which measures performances via hardware counters of concerned architectures.

3 Target codes

To perform our study, we chose to take a traditional benchmark of image processing, namely JPEG compression [17]. A first series of measurements via `gprof` (a standard profiling tool) allowed us to identify the Fourier Discrete Cosine Transform (FDCT) as one of the most time consuming around 25% of total execution time and therefore our efforts were focused on studying memory and code behavior of the FDCT routine [20].

3.1 Data Structure and access

One of main difficulty in image processing resides in data handling. Images are stored (and manipulated) as 2 dimensional arrays. Each element (referring to a single pixel) is further subdivide into 3 subcomponents of 2B long each (these subcomponents are called Y-Cb-Cr). In general most of the arithmetic can be carried in parallel both at the pixel level and at the subcomponent level (the elementary operations dealing with 2B operands). One of the mismatch between data structure/access for images and cache structure lies in the "2D" spatial locality while cache structure can only exploit "1D" spatial locality [4]. Hence, this problem causes a high amount of cache misses which affects performance. A very active research field studies the evaluation of multilevel cache hierarchy and the hardware prefetching strategies [1, 14, 15, 16].
Furthermore, many image processing algorithms (in particular the FDCT routine) operates on the image by first decomposing the whole image into disjoint square blocks of 8 by 8 pixels and then perform a FDCT on each $8 * 8$ block. This access pattern is not very efficient on standard cache systems because if the whole image is stored row wise (cf. standard C), access along rows will exploit spatial locality but access along columns will offer very poor spatial locality unless the cache is able to hold simultaneously a few rows of the whole image.

3.2 Algorithm/Implementation

Nowadays, the FDCT is coded accordingly to the proposal made in the `libjpeg` library from IJG[3], which implements the algorithm using Fourier transforms. The

[3] IJG: Independent Jpeg Group

arithmetic expression for computing each element of the transform can be easily coded using standard scalar instructions [17, 18].
However, as mentioned above, in this encoding process, the input component's samples are grouped into $8 * 8$ blocks and each block is transformed by the FDCT from the spatial domain to the frequency domain. The implementation uses a two-pass scheme which carries out an one-dimensional transform repeated 8 times on both rows and columns.
Following that algorithm structure and to exploit at best ILP (and minimize branch penalties), all of our source (named `kernel_u8`) were unrolled 8 times at the innermost loop level. In fact the total unrolling degree was 24 since each pixel contains 3 subcomponents.
Additionally, since we were focusing on memory behavior and since most of the operands for an $8 * 8$ FDCT can be stored in registers, we produced a stripped down variant of the FDCT, for which all of the arithmetic operations (except address computations) have been suppressed. This variant finally amounts to a simple Copy by blocks of $8*8$ pixels but has exactly the same memory access pattern as the original FDCT and it allows to exactly focus on memory behavior. Although this Copy is fairly simple, a few different implementations (named `kernels`) were produced and studied.
The first one, named `kernel_u8_gen_v7.1`, written in C code is given below (it should be noted that the innermost loop corresponds in fact to an unrolling degree of 24: 8 pixels each having three subcomponents):

```
void kernel(INT16 *restrict x,INT16 * restrict y,...){
  ...

  _3height = height*3;
  _3hDCT = _3height*DCTSIZE;
  _3width = width*3;
  l1=_3width / DCTSIZE;
  l2=height / DCTSIZE;


  k = 0;
  for(k2=0 ; k2<l2 ; k2++){
    nv=k2*_3hDCT;

    for(k1=0 ; k1<l1 ; k1+=3){
      ny= k1*DCTSIZE;

      for(j=0 ; j<DCTSIZE ; j++){
        vh=j*_3width;
        vz = vh + ny + nv ;

#ifdef PREFETCH
        __lfetch(__lfhint_none,x+vz+48);
        __lfetch(__lfhint_none,y+k+48);
#endif

        y[k+0] =  x[0 + vz];  // Y
        y[k+1] =  x[3 + vz];
        ...

        y[k+8] =  x[1 + vz];  // Cb
        y[k+9] =  x[4 + vz];
        ...

        y[k+16] =  x[2 + vz]; // Cr
        y[k+17] =  x[5 + vz];
        ...
        k+=24;
      }
    }
  }
}
```

This first kernel is the "reference" kernel. It was compiled using the Intel® C++ Compiler and the options described in the previous section. The main purpose of this kernel is to explore the compiler and cache behaviors with classic codes. The second kernel (`kernel_u8_intrinsic_prefetch`) differs from the first by only two lines which have been added in the source code to enforce prefetch instruction generation.
The third and the fourth kernels respectively called `kernel_u8_ptr_C` and `kernel_u8_ptr_asm` are reordered in C and assembly of the intrinsic prefetch version (i.e. they both include explicit prefetch instructions). The main purpose of these two variants is to evaluate the impact of address computation simplification.
Finally, the last one, named `kernel_u8_rotating_register`, has been directly coded in assembly language and requires a fine knowledge of architecture. It relies on software pipelining of the innermost loop and uses the specific rotating registers of the Itanium™ architecture.
Last but not least, the full FDCT code was tested in two different versions: the first "one" is plain, i.e. relying solely on compiler for optimizations, the second one contains prefetch instructions added at the source code level via intrinsics.

4 Methodology

Besides the different kernels described in the previous section, other parameters have been explored. Image size has been varied extensively (from $16 * 16$ pixels up to $768 * 768$ pixels) to study the impact of the various cache levels. The smaller images would fit entirely in the L1D cache while the largest exceeds L3 cache size. In this article, we did not go further because nothing justified it: neither interaction between X and Y arrays nor peaks of misses appear. It is obvious that images used in the experiments could have had a size of $1024*1024$ pixels or even more.
All of the images were square and two arrays (X and Y) were used to store them: the first one for the original image and the second one for the transformed one.
Each pixel, coded on 48 bits, is described by three shorts representing respectively the brightness (Y) and the chrominance (Cb and Cr). Hence an image named `img16` of $16 * 16$ pixels (256 pixels) has a size of 1.5KB. Images

are read line by line, and must imperatively be multiple of 16 pixels (this limitation is due to the used algorithm of FDCT).
In the table below, the table summarizes image sizes which have been tested together with their cache "level" (i.e. cache level which is big enough to store source and destination images).

Name of dataset	Image size	Cache level
img016	1.5 KB	L1D
...	...	...
img032	6 KB	L1D
img048	13.5 KB	L2U
...	...	...
img144	121.5 KB	L2U
img160	150 KB	L3U
...	...	...
img352	736 KB	L3U
img368	793.5 KB	Mem
...	...	...
img544	1.69 MB	Mem

The table above is given as an indication, as overlapping between cache levels can occur. For instance, the image named `img288` can be part in the L3U and part in memory. In the experimental section, figures reporting L2 and L3 cache misses (Fig. 4 and 5) are given to visualize where are located images in the cache hierarchy.
For these various images, the impact of intrinsic prefetch instructions and of their parameters (prefetch type - `none, nta` - and distance) has been studied. However, only performance corresponding to the best options are reported here.
In our experiments, the array layout in the virtual memory space is tightly controlled. In particular, the impact of the starting address of each array (X,Y) is studied in depth. To achieve this goal, the parameters Offset X (resp. Offset Y) are introduced according to the following relations:

Element	Virtual address (Bytes)
X[i]	512 KB + Offset X + 8*i
Y[i]	512 KB + DIST + Offset Y + 8*i

`DIST` is an additional parameter mainly used to avoid array overlap, i.e., making sure that arrays X and Y are located in disjoint portions of memory space. In most of our experiments `DIST` remains constant.
It could be argued that the starting address of arrays does not have a major impact on performance, however, this is far from being true [16]. Besides, the offsets are varying in the 0 to 512 range with increments of 8. Finally, measurements are taken by the `Perfmon` library (via our toolkit) which accepts up to four readings of events at a time. The selected events are the cycle counter of the processor and the number of cache misses in L2U and in L3U.

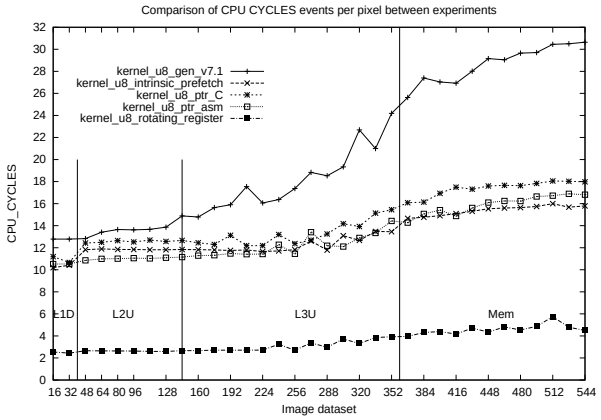

Figure 1: CPU CYCLES per pixel

Dataset (cycles/pixel)	L1D	L2U	L3U
kernel_u8_gen_v71	12.70	13.50	22.69
kernel_u8_intrinsic_prefetch	10.43	11.82	12.67

Table 1: The impact of prefetch instruction.

5 Results

All of the performance numbers presented are normalized with respect to one pixel, i.e., the measurements correspond to the average number of cycles, L2 misses or L3 misses to compute one pixel.

5.1 Prefetch impact

Images can be regarded as two-dimensional matrices where each value named pixel, is referenced by two indices. When a pixel is used in multimedia processing, there is a high probability that a given application needs to access adjacent data in both vertical and horizontal direction in the near future. But, if horizontal values $M[i][j]$ and $M[i+1][j]$ of a matrix are stored in allocating blocks of adjacent bytes, vertical values $M[i][j]$ and $M[i][j+1]$ can be a lot of bytes apart. This storage causes many cache misses and affects performance.
Our strategy for managing prefetchs is based on the knowledge of benchmark characteristics and the structure of image data. We have selected specific prefetch instruction type to manage the different levels of cache hierarchy and chosen prefetch distance to provide a data line of a given block.
In `Figure 1`, the results obtained for the reference variant (`kernel_u8_gen_v7.1`) generated automatically (not by hand), proved to be quite bad when compared with the three other variants coded in C (the assembly coded variant being left on the side for time being). Furthermore

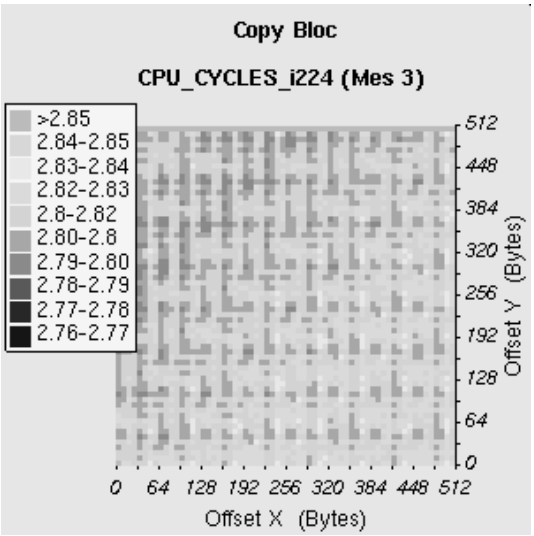

Figure 2: The impact of Offsets with isosurface

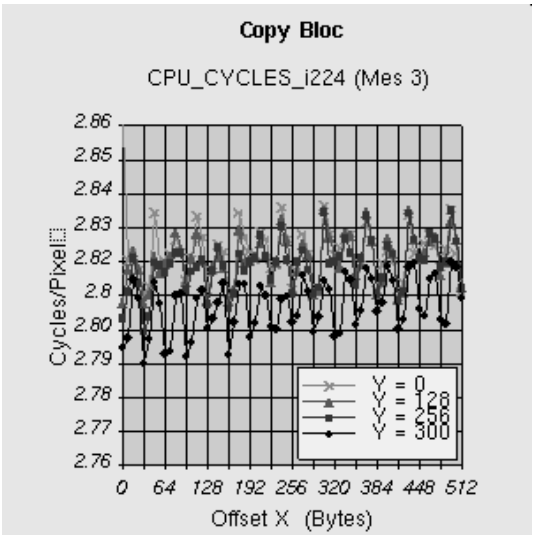

Figure 3: The impact of Offsets with graph

the performance gap increases when moving to the higher levels of the memory hierarchy. The key reason for this performance gap is the absence of prefetch instructions in the reference code: they were absent at the source level and the compiler did not generate any prefetch. This lack of optimization by the compiler is probably due to the triply nested loop structure.

In L1D region, for all versions, performance is disappointing, around 10-12 cycles per pixel (i.e. around 3.5 cycles per 2B load). This is far from the peak capabilities of Itanium2™, i.e. 2 loads and 2 stores per cycle. A close inspection of the assembly code revealed first that address computation was not performed very efficient and second that the innermost loop was not software pipelined.

The cache strategy, in this case, consists in considering that all the data is immediately accessible. But, image processing exhibits a 2D spatial locality and data can be not stored in the cache. A simple addition in C code of two prefetch instructions which is easy to set up, can reduce the memory access latency. However, even with the adding of prefetch instructions, our experiment result was not convincing. The value of CPU cycles is still too high in comparison with the machine ability.

Our toolkit allows us to visualize the impact of Offsets on performance, (cf. Figure 2 and 3) and we notice some patterns implying bank conflicts between loads and stores. For example, we notice that the loads exerted on a same cache line will conflict and will only be processed in two cycles instead of one. We can then imagine techniques to get closer to optimal performance, using an interleaved kernel.

5.2 Address computation

The IPF architecture provides the post-incrementation associated with some instructions such as `load` and `store`. Theses instructions allow to perform at the same time (cycle) address computation and memory transaction. For instance, we can suppose that we want to store array elements (defined as `shorts`) address of which are separated by 24 B. This operation can be performed by the code fragment below:

```
st2  [r40]=r42,48 ;; // post-increment of 24 (short).
```

The resulting code (equivalent to a standard store followed by an address increment) is more compact and faster to issue. In our case, the Intel® compiler did not use such techniques and numerous address computation instructions were inserted in the code.
Hence, we first proposed an optimization in our C code, named `kernel_u8_ptr_C` to improve address computation. In this code, the compiler is able to set up the post-incrementation in the index of Y array (but unfortunately not for the X array). The corresponding kernel in C language is given below:

```
for(k2=0 ; k2<l2 ; k2++){
   for(k1=0 ; k1<l1 ; k1+=3){
     for(j=0 ; j<DCTSIZE ; j++){
       vz = vh + ny + nv ;

       ptr= x+vz;

       __lfetch(__lfhint_none,ptr+48);
       __lfetch(__lfhint_none,y+k+48);

       y[k+0] =  ptr[0];
       y[k+1] =  ptr[3];
       ...
       vh += _3width;
```

Dataset (cycles/pixel)	L1D	L2U	L3U
kernel_u8_gen_v71	12.70	13.50	22.69
kernel_u8_intrinsic_prefetch	10.43	11.82	12.67
kernel_u8_ptr_C	10.43	12.41	13.93
kernel_u8_ptr_asm	10.43	11.03	12.89

Table 2: The impact of post-incrementation.

```
      }
      ny+= DCT3;
    }
    nv+=_3hDCT;
}
```

Such optimizations did not really pay off (cf. Table 2). Modifying directly the assembly code for inserting post incrementation on the X array (`kernel_u8_ptr_asm`) was not more beneficial. The real performance problem lies elsewhere: first due to the lack of software pipelining at the innermost loop level and second the use of optimistic latencies for memory. The compiler assumes data is at least in L2U and schedules instructions accordingly: a load and an arithmetic instruction using the load result are scheduled 6 cycles apart. Now, if at execution time, load latency exceeds 6 cycles, this will often induce delays and performance loss. This is due to the Itanium EPIC architecture which offers much less opportunities (than superscalar architectures such as Power4) for instruction reordering at run time.
Below, you will find a table summarizing the experiments done ; all the result values have been normalized with respect to one pixel.

5.3 Software pipelining/Memory latency

In spite of our optimizations, the results remain around the same level as our previous ones and do not go below a significant threshold. Hence, they were very disappointing in regard to the expected possibilities, namely 0.6 cycles per load [16] (for floating-point loads). For instance, as a pixel is described by three components, in the best of cases, we get 3.67 cycles per integer load in L2U. Another factor, image size, plays a critical role in image processing. Most of images have a size around 300 KB. Therefore, they are usually allocating in the L3U cache. For an Intel® compiler, the instructions latency is the one given in L2U. A delay equal to 6 cycles between a load instruction and its associated store instruction is used in the ordinary course of events. But this delay is not adapted for image processing where it would be preferable to take instruction latency of the L3U cache memory.
To improve in a consequent way the performances impacted by the problem of the instruction latency, we used the rotating register. We started from scratch and developed by hand an assembly version (`kernel_u8_rotating_register`) using software pipelining of the innermost loop and assuming a load to use latency more conservative (around 13 cycles corresponding to data in L3U). To simplify register allocation, the rotating register mechanism was used. Although these optimizations were performed by hand, they can be easily automated and integrated into a compiler.

The results presented in Figure 1 lead to major performance improvements at all levels of the memory hierarchy: even in the L3U region, performance improved by 85.4%.

The price to be paid for such optimizations is a much higher register pressure. In fact although Itanium provides a large register set (128 integer registers), we could could not accommodate the whole loop in the Itanium register set. This was mainly due to the high unrolling degree used. Therefore a small prologue was introduced (`kernel_u8_rot_init`) before the innermost loop. The cost of this prologue (mainly used for address computations) is not negligible. And the effects show up in the obtained performance.

Dataset (cycles/pixel)	L1D	L2U	L3U
kernel_u8_rot_init	0.97	0.96	1.83
kernel_u8_rotating_register	2.44	2.6	3.31

5.4 Memory hierarchy behavior

Figure 4 (resp. 5) displays the average number of L2 (resp. L3) misses per pixel when varying image size.

Unfortunately, the hardware counters do not allow to discriminate in a simple manner loads and prefetch. Therefore all of the variants have a very similar behavior.

The L2 behavior matches pretty well our assumptions, i.e. in the L2U region, the L2 misses are close to zero. The L3 behavior is much less clear, the number of L3 misses becomes to increase in a non negligible manner while the images should still fit in L3U.

In a last part we will have the results of the FDCT with our basic techniques of optimization. We slightly modified this function in order to emphasize the instruction parallelism which will be useful to us in the continuation of our experiments. Indeed, we will be able via a dynamic code generator to then use the specific instructions to image processing.

We can note that with a simple addition of intrinsic prefetch, the improvement is of 8% in L2U and 15% in L3U (cf. `Table 3`).

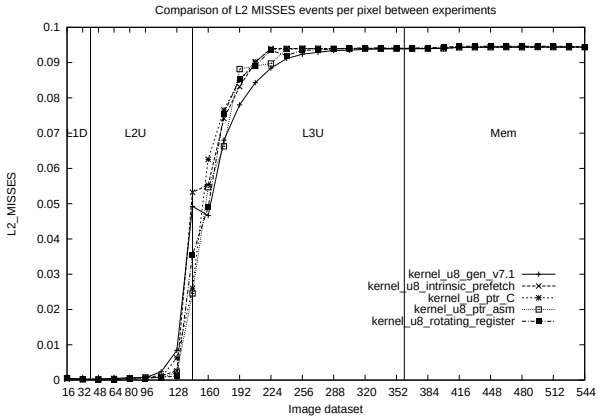

Figure 4: L2 MISSES per pixel

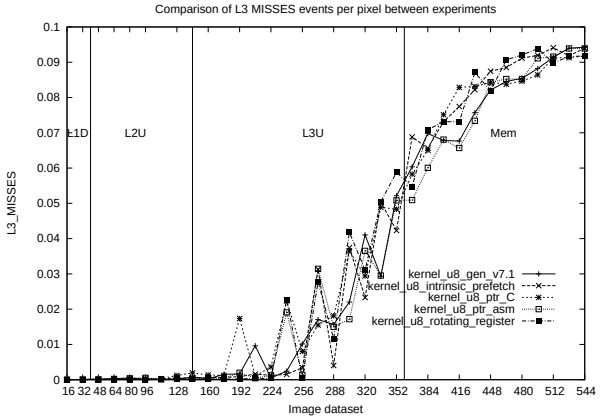

Figure 5: L3 MISSES per pixel

Dataset (cycles/pixel)	L1D	L2U	L3U
fdct_gen	18.86	21.38	23.65
fdct_prefetch	19.09	19.50	20.08

Table 3: The impact of prefetch instruction.

6 Conclusion

Modern microprocessors rely on complex cache systems to deliver top performance. Hence, this is difficult to get a good grasp of cache behavior and to exploit efficiently these capabilities. This is even truer for the multimedia instructions which have a 2D spatial locality, inadequate for the cache memory system [4]. In spite of that, and without using heavy artillery [21], performances can be considerably increased with prefetch techniques. With our technique of prefetch insertion, we improve the results in L3U by 47.76%. In doing a software pipeline, this improvement is of 85.4%.

The purpose of our approach was to understand the interactions which exist in the cache memory and to reduce their impacts in the future. Hence this work will be extended in a few directions such as implementing a dynamic code generator which will have taken into account these remarks. Besides, we are interested in the way of exploiting ILP before using the graphical instruction set and optimizing multimedia code at runtime. We want to give a possibility to define fine grain parallelism in multimedia processing to developers.

With this dynamic generator it will then be possible to define a new type, named `pixel`, adapted to image processing and using the graphical instructions of known architectures.

References

[1] J. L. Hennessy and D. A. Patterson, "*Computer Architecture, A Quantitative Approach - Second edition*", International Thomson Publishing, 1996.

[2] K. Diefendorff and R. Dubey, "*How Multimedia Workloads Will Change Processor Design.*" IEEE Computer, September 1997.

[3] L. Chiariglione, "Impact of multimedia standards on multimedia industry", Proc. of IEEE, v. 86, n. 16, pp. 1222-1227, 1998.

[4] R. Cucchiara, M. Piccardi, A. Prati, "*Exploiting Cache in Multimedia*", Proc. of IEEE ICMCS'99.

[5] R. Cucchiara, M. Piccardi, A. Prati, "*Temporal Analysis of Cache Prefetching Strategies for Multimedia Applications*".

[6] A. Prati, "*Exploring multimedia applications locality to improve cache performance*", Proc. of the eighth ACM international conference on Multimedia October 2000.

[7] N. T. Slingerland and A. J. Smith, "*Cache Performance for Multimedia Applications*", Proc. of the 15th international conference on Supercomputing, June 2001.

[8] B. Ramakrishna Rau and J. A. Fisher, "*Instruction-Level Parallel Processing: History, Overview and Perspective*", Journal of Supercomputing, Vol. 7, No. 1, pages 9-50, January 1993.

[9] D. E. Culler, J-P. Singh, A. Gupta, "*Parallel Computer Architecture: A Hardware/Software Approach*", Morgan Kaufman, 1998.

[10] R. B. Lee, M. D. Smith, "*Media Processing: a new Design Target*", IEEE Micro, vol. 16, pp. 43-45, 1997.

[11] S. Sohoni, R. Min, Z. Xu, Y. Hu, "*A Study of Memory System Performance of Multimedia Applications*", Proc. of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, June 2001.

[12] "*Intel® Itanium™ Processor Reference Manual for Software Optimization*", Intel® Document Number: 245473-003, November 2001.

[13] "*Intel® Itanium2™ Processor Reference Manual for Software Development Optimization*", Intel® Document Number: 251110-001, June 2002.

[14] T. C. Mowry, M. S. Lam, and A. Gupta, "*Design and evaluation of a compiler algorithm for prefetching*", Proc. ASPLOS-V, pages 62-73, October 1992.

[15] P. Struik, P. van der Wolf, A. D. Pimentel, "*A Combined Hardware/Software Solution for Stream Prefetching in Multimedia Applications*", Proc. of SPIE Multimedia Hardware Architectures, pages 120–130, January 1998.

[16] W. Jalby, C. Lemuet "*Exploring and Optimizing itanium2 Cache(s) Performance for Scientific Computing*", 2nd Workshop on EPIC Architectures and Compiler Technology, November 2002. `http://www.epicea64.org/`.

[17] G.K. Wallace, "*The JPEG Still Picture Compression Standard*", Communications of the ACM 34, 1991.

[18] E. Feig, S. Winograg, "*Fast Algorithms for the Discrete Cosine Transform*", Research Report, IBM RC 16148, 1990.

[19] G. Aggarwal, D.D. Gajski, "*Exploring DCT implementations*", Technical Report, Department of Information and Computer Science, University of California, Irvine, March 1998.

[20] S. Roy, B. Shen, "*Implementation of an Algorithm for Fast Down-Scale Transcoding of Compressed Video on the Itanium™*", Technical Report, Hewlett-Packard Laboratories, Palo Alto, 2002.

[21] M. S. Schlansker, B. R. Rau, "*EPIC: Explicitly Parallel Instruction Computing*", Computing Practices, Hewlett-Packard Laboratories, February 2000.

[22] M. Smotherman, "*Historical background for HP/Intel® and IA-64*", `http://www.cs.clemson.edu/mark/epic.html`, May 2003.

[23] K. M. M. Rao, "*Image Processing for medical Applications*", 14th Conference WCNDT, `http://www.ndt.net/abstract/wcndt96/-wcndt96.htm`, December 1996.

[24] "*3D Military Products raise their scopes applications built for soldiers in Joe Public's Hands*", Geo Informatics, `http://www.geoinformatics.com/`, Issues On-line - Volume 5, June 2002.