International Journal of Embedded and Real-Time Communication Systems, 1(4), 1-20, October-December 2010 1

A Multidimensional Software Cache for Scratchpad-Based Systems

Arnaldo Azevedo, Delft University of Technology, The Netherlands

Ben Juurlink, Technische Universität Berlin, Germany

ABSTRACT

In many kernels of multimedia applications, the working set is predictable, making it possible to schedule the data transfers before the computation. Many other kernels, however, process data that is known just before it is needed or have working sets that do not fit in the scratchpad memory. Furthermore, multimedia kernels often access two or higher dimensional data structures and conventional software caches have difficulties to exploit the data locality exhibited by these kernels. For such kernels, the authors present a Multidimensional Software Cache (MDSC), which stores 1- 4 dimensional blocks to mimic in cache the organization of the data structure. Furthermore, it indexes the cache using the matrix indices rather than linear memory addresses. MDSC also makes use of the lower overhead of Direct Memory Access (DMA) list transfers and allows exploiting known data access patterns to reduce the number of accesses to the cache. The MDSC is evaluated using GLCM, providing an 8% performance improvement compared to the IBM software cache. For MC, several optimizations are presented that reduce the number of accesses to the MDSC.

Keywords: *Cell Processor, Computer Organization, H.264 Motion Compensation, Memory Hierarchy, Multidimensional Cache, Scratchpad Memory, Software Cache*

INTRODUCTION

Most processors use a cache to overcome the memory latency. Some processors, however, employ software-controlled high-speed internal memories or *scratchpad* memories to exploit locality. Processors based on scratchpad memories are very efficient in terms of power and performance (Banakar et al., 2002). The power efficiency is due to the simple structure of the memory compared to caches. Scratchpad memories also have predictable latencies. These characteristics make scratchpad memories a common choice for embedded processors.

Many kernels (e.g., multimedia kernels) have a working set that is predictable, which makes it possible to transfer data before the computation. It is often also possible to overlap computation with data transfers by means of a double buffering technique, where the data in one buffer is processed while the data for the next processing stage is fetched in another buffer. In scratchpad-based systems these data transfers usually need to be explicitly programmed using

DOI: 10.4018/jertcs.2010100101

Direct Memory Access (DMA) requests. There are also many multimedia kernels, however, that process data that is known just before it is needed. This is the case, for instance, in the Motion Compensation (MC) kernel of H.264 video decoding. Only after Motion Vector Prediction it is possible to fetch the data necessary to reconstruct the frame. Other kernels have working sets that exceed the capacity of the scratchpad memory. This is the case in the Gray Level Co-occurrence Matrix (GLCM) kernel. It features a relatively random access that renders DMA requests for each individual access impractical. MC is an interesting kernel as its memory access pattern is similar to other important multimedia kernels such as texture mapping. GLCM features a fine-grain random access pattern that is representative of other tabulation algorithms, such as histogram.

Both kernels exhibit data locality that could be exploited by a cache. In MC the motion vectors are often closely related so that data that is (logically) adjacent to the reference area is needed to decode the next macroblock (MB). In GLCM the difference of adjacent pixels is often small so that the kernel accesses small parts of the GLCM matrix. In a scratchpad memory a cache can be emulated. This is often referred to as a software cache. Software caches, however, incur high overhead, representing up to approximately 50% (Gonzalez et al., 2008) of the total application execution time. Such high overheads could harm performance compared to hand-programmed, just-in-time DMA transfers. It is therefore necessary to reduce the number of cache accesses as much as possible. An additional feature of these as well as many other multimedia kernels is that they access 2- or higher-dimensional data structures and adjacent sub-rows are not consecutive in memory.

For such kernels we propose a Multidimensional Software Cache (MDSC). The MDSC stores 1- to 4-dimensional blocks (sub-matrices) and the cache is indexed by the matrix indices rather than a linear memory address. This approach both minimizes the memory transfer time and the number of cache accesses. The first is achieved by grouping memory requests, thereby reducing the overhead associated with memory requests. The latter is achieved by exploiting the multidimensional access behavior of the application.

Our experimental platform is the Cell processor. Implementing a software cache for the Cell processor is an active research topic (Balart et al., 2007; Lee et al., 2008; Chen et al., 2008). Balart et al. (2007) propose a compile time software cache with support for asynchronous transfers. The compiler uses asynchronous transfers to overlap memory transfers with computation. They report a speedup of 1.26 to 1.66 over synchronous transfers. Chen et al. (2008) propose a similar approach with support for runtime prefetching based on the access patterns. These works are complementary to our work since the MDSC can be used as the software cache implementation for the compiler.

The current version of the MDSC does not feature cache coherency. Currently it is not needed because either only read-only data is cached or efficient multicore kernel implementation avoids the need for coherency. However, cache coherency is an important feature for caches in a multicore environment. Lee et al. (2008) and Seo et al. (2009) propose a coherent shared memory interface for the Cell BE using software caches. It employs a software cache in the local store for page-level caching. It guarantees coherence at the page level and uses centralized lazy release coherency.

A static analysis tool for finding the best parameters for a software cache given an application is proposed in (Senthil et al., 2008). The tool uses traces with annotated memory accesses and bases its analysis on the frequency of cache accesses to a given cache line and the number of accesses between two accesses to the same cache line. A similar tool would be desirable for the MDSC as in this work we rely on exhaustive search to find the optimal parameters. The development of such a tool is future work.

Zatt et al. (2007) show that caching the MC reference area can save up to 60% bandwidth and

more than 75% of the memory cycles compared to issuing a new request for each reference area. The presented solution is hardware specific, however, and therefore not sufficiently general/flexible to be implemented in a programmable embedded multimedia system.

This article makes the following contributions:

- An evaluation of the overhead incurred by a generic software cache for MC and GLCM.
- We propose the Multidimensional Software Cache (MDSC) that caches 1- to 4-dimensional blocks of data that are logically adjacent, thereby reducing the number of cache accesses and the DMA startup overhead.
- We determine the optimal parameters of the MDSC for MC as well as GLCM.
- For MC several optimizations are presented that reduce the number of accesses to the MDSC.
- We compare the performance of the MDSC to the performance achieved by hand-programmed DMAs as well as the heavily optimized IBM software cache. The experimental results show that for GLCM the MDCS provides an 8% performance improvement compared to the IBM software cache. For MC, the MDSC provides an average 65% improvement over just-in-time DMAs and 43% over the IBM software cache.

This article is organized as follows. First section describes the architecture of the Cell processor and evaluates the latency and throughput of DMA operations. The MDSC implementation, properties, and its application programming interface (API) are presented in the next section. It is followed by the description of the employed benchmarks and the MDSC optimizations for MC, and the presentation of the methodology used to evaluate the proposed software cache. Afterwards, the experimental results are presented and discussed. Finally, conclusions are drawn.

CELL PROCESSOR ARCHITECTURE

This section briefly describes the Cell processor. The main characteristics of the Cell processor are presented with the focus on the memory system. It also reports the memory latency for transferring data from main memory as a function of the size of the request and the number of requesting processing elements.

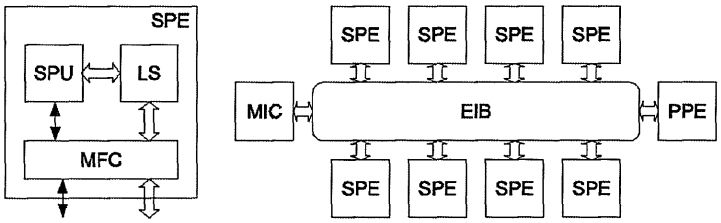
Cell Processor

The Cell Broadband Engine (Kahle et al., 2005; Gschwind et al., 2006) is a heterogeneous multi-core processor designed for multimedia and game processing. It consists of one Power Processor Element (PPE) and eight Synergistic Processing Elements (SPEs) connected by the Element Interconnect Bus (EIB) that contains four 16B-wide data rings. A block diagram of the processor is depicted in Figure 1.

The PPE is a simplified version of the PowerPC processor family. It is based on IBM's 64-bit Power Architecture ("Power Architecture," 2010) with 128-bit vector media extensions. It is fully compliant with the 64-bit Power Architecture specification and can run 32-bit and 64-bit operating systems and applications. The PPE is dual-threaded and has a two-way in-order execution pipeline unit with 23 stages. The PPE supports a conventional two-level cache hierarchy with 32KB L1 instruction and data caches and a 512KB unified L2 cache.

As depicted in Figure 1, each SPE contains a Synergistic Processing Unit (SPU), a Local Store, and a Memory Flow Controller (MFC). The Local Store is a 256KB scratchpad memory and the MFC is composed of a DMA engine, a memory management unit, and a bus interface. The SPUs are tailored for multimedia processing and are single-threaded, non-preemptive, two-way in-order processors. The register file consists of 128 128-bit wide registers. All instructions are Single-Instruction-Multiple-Data (SIMD) and they operate on 128-bit vectors with varying element width, i.e., 2 × 64-bit, 4

Figure 1. Cell Broadband Engine block diagram



× 32-bit, 8 × 16-bit, 16 × 8-bit, or 128 × 1-bit. Data should be 128-bit aligned and there is no hardware support for scalar operations. The design decision of not supporting scalar and unaligned operations was taken to reduce the control complexity and to eliminate several stages from the critical memory access path (Gschwind et al., 2006).

An SPE can only access data and code stored in its 256KB Local Store. To access the external memory the SPU issue a DMA request to the MFC. There are four types of DMA requests: *put*, *get*, *putlist*, and *getlist*. A *put* request writes data from the Local Store to the external memory. A *get* copies data in the external memory to the Local Store. Requests can be grouped in a list with up to 1024 requests, and are issued by *putlist* and *getlist* requests. The DMA unit requests the data and sets a flag when the request is performed. Data and instructions are transferred in packets of at most 16KB and both the source and the target address must be 16B aligned. The DMA unit can handle up to 16 requests concurrently and data communication can be performed in parallel with computation. Double buffering can be employed to hide the DMA transfer latency. The Local Stores are mapped in the global memory address space to allow Local Store-to-Local Store communication, but this memory (if cached) is not coherent in the system.

DMA Latency

Figure 2 depicts the DMA latency as function of the DMA size. It can be seen that memory request latencies are approximately the same up to 1024 bytes for the Cell processor. It also depicts the DMA latency when several SPUs are communicating simultaneously. There is no difference in delay when 1 or 2 SPUs are fetching data because the Cell processor features a dual channel memory controller. However, a single SPU cannot make use of both channels simultaneously. Full bandwidth is achieved only when several SPUs are accessing the external memory simultaneously.

Several DMA operations can be grouped in a single DMA list operation in order to reduce the DMA startup cost. Figure 3 depicts the latency of DMA list operations for several numbers of requests (Y) and request sizes (X). For example, the label Y2X256 means that 2 DMA operations are grouped in a single DMA list operation, and that each operation fetches 256 bytes. We refer to the size of each individual DMA operation as the *line size*. For clarity, results for 64- and 128-byte lines have been omitted, as the results are very similar to the results for 32- and 256-byte lines. The latency for requesting the same block using several individual DMA requests is depicted for comparison.

The results show that requesting multiple lines reduces the request overhead. The average

Figure 2. DMA latency as function size of the transfer size, for several SPEs communicating simultaneously

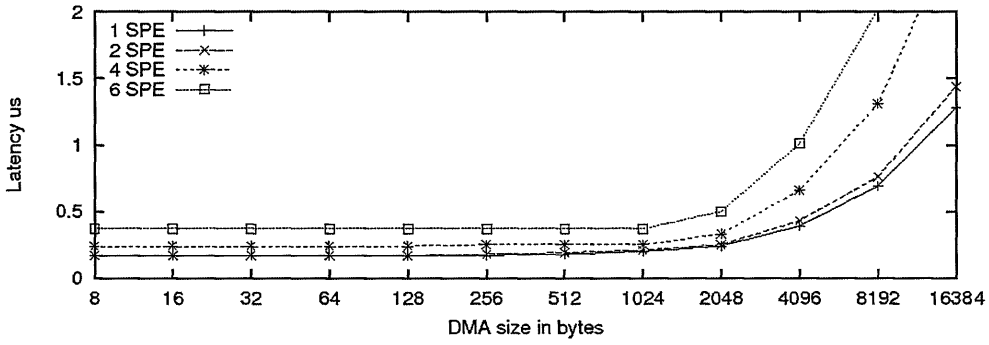
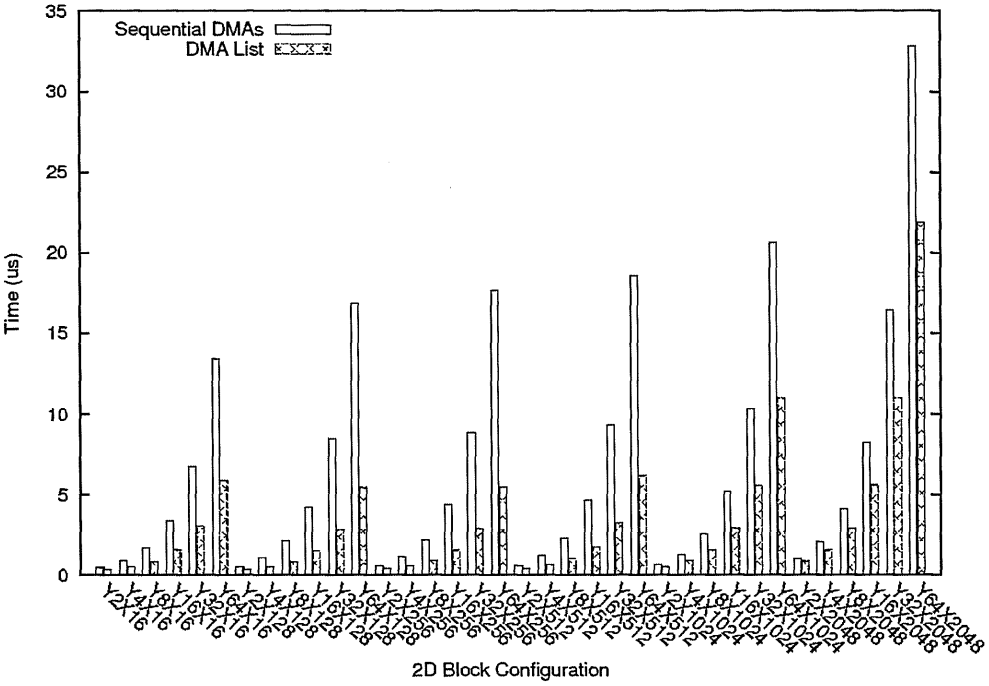


Figure 3. Latency of DMA list operation compared with a sequence of individual DMAs requests for the same 2D block configuration



request time is reduced by 50% using DMA list, when compared with using sequential DMA requests. It ranges from 27% when fetching 2 lines to 69% for 64 lines.

MULTIDIMENSIONAL SOFTWARE CACHE

In this section we propose a Multidimensional Software Cache (MDSC). We start by highlight-

ing the differences between hardware caches and software caches. Afterwards the reasons to shift from address indexed caches to data structure indexed caches are presented. Next, the arguments for a multidimensional software cache are given. With the underlining motives addressed, the proposed software cache (SC) structure is presented.

Scratchpad memories are more area and power efficient than hardware caches. However, they require additional programming effort as they need explicit commands to fetch data from the main memory. These commands can be automatically handled by compilers, but are usually handled by the programmer for better performance or because of lack of tools.

One option to increasing the efficacy of scratchpad based systems is to use a software cache. They increase the programmability and with software cache the limited size of the Local Store is less of a concern. Software caches, however, incur additional overhead, which can be prohibitive. This overhead is further increased if the cache does not match the application's data access pattern. This is the case when using a generic cache for MC and for other image processing applications, such as texture mapping.

Software caches provide the abstraction of a large, fast local memory to the programmer. This abstraction captures all data and instructions used in the program. Because of its generality, indexing by the memory address is a natural choice. However, because of this generality and because every access has to be done through the caches, it is not possible to exploit application-specific data locality.

Software caches for scratchpad based processors such as the IBM software cache ("Example Library," 2010) capture accesses to specific structures. Only the data structures that do not fit in the Local Store are likely to be accessed through the SC. Just like hardware caches, the IBM software cache uses the memory addresses of the data to index the cache. Once again it loses the opportunity to use the SC parameters information to exploit data locality. In this case it is critical, as accessing the cache implies runtime overhead. It is possible to exploit data locality to reduce SC access overhead, due to its characteristics. These characteristics include the parameters that are known at compile time and that SC captures only few specific data structures access.

The MDSC uses the indices of the accessed data structure to index the cache. So, instead of consulting the cache as *access_SC(&datastructure[i][j])*, we propose *access_MDSC(&datastructure, i, j)*. Although similar, the second makes explicit more information about the data structure and the access. Another characteristic of the MDSC is the ability to mimic in the SC the logical organization of the accessed data structure in the main memory. The cache blocks are 1- to 4-dimensional. A 2D cache can be used to store rectangular areas of images while a 3D cache can be used to store areas of a sequence of video frames. We cannot give an example where 4D cache blocks would be useful. However, because of the SIMD instructions set of the SPE, 4D cache blocks are just as efficient as 2D and 3D cache blocks.

This approach makes the MDSC differ from a regular cache in two ways. First, it differs in the tag calculation and the set allocation. In a regular cache the tag is the address shifted by the base-2 logarithm of the size of the cache line. For MDSC the tag is the concatenation of each index shifted by the base-2 logarithm of the size of its block dimension. The set is also calculated based on the indices. An exclusive or operation is performed between each index value and its successor and the results are added together. The set is the sum modulo the number of sets in the MDSC. The second difference is the format and the load of the multidimensional block. The multidimensional block is formed by a group of cache lines gathered from memory according to the number and size of the dimensions of the MDSC. For a 2D MDSC, n cache lines represent a block. A strided access to the main memory is performed to load the consecutive lines. In our implementation a DMA list is created with a DMA request for each list entry. It is similar for 3 and 4-dimensional blocks

where a 3D block is a collection of 2D blocks and a 4D blocks is a collection of 3D blocks.

This approach presents two advantages over regular caches. First, it reduces the memory latency by grouping several memory requests. Since, as was shown in the previous section, a single DMA list operation has a lower latency than several sequential DMAs. For regular caches, accessing a new image area would result in a new DMA request for each line of the new area being accessed. The second advantage is that the MDSC can be used to reduce the number of accesses to the SC. As the shape of the cache block is known, it can be used to access the data of a cache set without actually checking if the data is present in the cache. This can be done by simple pointer arithmetic. In other words, a single cache lockup is necessary for accessing an entire block.

Like a regular cache, the MDSC performs the following steps to access a block:

1. Check if the data is already present in Local Store. Each block in the cache is represented by a tag. The tag is formed by concatenating the indices of the data that is being accessed after dividing each index by the size of its respective block dimension. In the case of a set associative cache, a hash function is used to define in which set the block referred to by the indices is stored. The new formed tag is then compared with the other tags in the specific set in the tag array (in case of a set associative cache) or with all tags in the tag array (for a fully associative cache).
2. In case the tag is not present, decide where to place the new block. If the block is not currently stored in the cache it is first necessary to determine where it should be placed. The MDSC uses the next position of the last allocated block in a FIFO fashion, both for the set associative and fully associative configurations.
3. If out of space, decide which block to eject from Local Store. If the chosen place was already been used by another block that was modified, it needs to be written back to the main memory. The tag of the block being written back is separated according to the size of each dimensions of the block to recover the block address in the main memory.
4. If necessary, perform DMA operations. Issue DMA requests to, if necessary, copy the modified data back to main memory and to copy the new block from main memory to the Local Store. A MDA list is prepared one entry each line of the multidimensional block. With the list ready a *putlist* and/or *getlist* is issued. The process blocks until the DMA unit informs the SPU that the new block is present in the Local Store.
5. Perform memory access - A modulo operation is performed between each index of the request position with it's respectively dimension size. The result is used to calculate the position of the requested data in the cached block.

The associativity of the MDSC can be configured. It allows a fully or a set associative configuration and for static or dynamic implementation. A fully associative cache is possible when the number of cache blocks is small. For the fully associative configuration, the MDSC uses First-In-First-Out (FIFO) policy to replace blocks when the cache is fully utilized. The FIFO was selected due its low implementation complexity. For set associative a round-robin mechanism is used to select the block to be replaced.

In the static implementation the MDSC parameters are constants and thus are known at compile time while the dynamic implementation allows the MDSC parameters to be modified at runtime. The former is more efficient than the latter as it allows optimization of the code for several parameter options. However, a dynamic configuration is necessary when the data to be cached can have different characteristics, such as the resolution of the video being decoded, and the MDSC needs to adapt to these characteristics.

The Application Programming Interface (API) consists of two functions: *ac-*

cess_sc(&datastructure, i, j) and *accessp_sc(&datastructure, i, j)*. The function *access_sc* returns the data stored in the (i,j) position while the function *accessp_sc* returns its address (memory pointer). These functions checks if the 2D block which contains *datastructure[i][i]* is present in the software cache. If it is, this function returns immediately. If not, this function blocks until the 2D block which contains *datastructure[i][i]* is fetched from main memory to the software cache and then returns to the caller.

Because the MDSC uses matrix indices to index the cache, the boundaries of the data structure need to be specified. This can be done via macros in the static configuration, thus increasing the performance, or dynamically at runtime when using the dynamic configuration. Figure 4 depicts how the MDSC can be configured.

STUDIED APPLICATIONS AND MDSC ENHANCEMENTS

In this section, the applications used for the case study are presented and qualitative reasons are given why the studied applications could profit from a MDSC. First, the GLCM algorithm is presented. It is followed by a description of MC. For the MC kernel it is possible to exploit the access behavior. We will describe several enhancements that exploit this fact.

GLCM

The Gray-Level Co-occurrence Matrices (GLCM) is a tabulation of how often different combinations of pixel brightness values (gray levels) occur in an image. The second order GLCM considers the relationship between groups of two (usually neighboring) pixels in the original image. It considers the relation between two pixels at a time, called the reference and the neighbor pixel. The GLCM is useful to extract statistical characteristics of the image and is used in medical imaging and content based image retrieval (Shahbahrami et al., 2008). In this study, all 9 neighboring pixels are examined, as depicted by the pseudo code in Figure 5.

In this application, the source image being processed can be easily accessed through DMAs. The temporal locality of the image is very low and the spatial locality can be captured with DMAs. Also the DMA latency can be hidden using double buffering. However, the GLCM matrix is indirectly indexed and its size

Figure 4. MDSC interface

```
#define CACHE_NAME gsc_y
#define CACHED_TYPE unsigned char
#define CACHE_TYPE          0     /* 0=read-only, 1=read-write              */
#define CACHE_STATS               /* Activates statistic collection         */
#define CACHE_FULL_ASSOC    0     /* 1=cache is fully associative           */
#define CACHE_LOG2NWAY      2     /* Log 2 number of ways                   */
#define CACHE_LOG2NSETS     5     /* Log 2 number of sets                   */
#define CACHE_DIM           3     /* Number of block dimensions             */

#define CACHE_X_LOG2SIZE    9     /* Log 2 first dimension (line size)      */
/* When accessing the MDSC, the first index must be between 0 and CACHE_X_RANGE */
#define CACHE_X_RANGE       1920
#define CACHE_LOG2_X_RANGE 11   /* Log2(CACHE_X_RANGE)

#define CACHE_Y_LOG2SIZE 4      /* Log 2 second dimension (number of lines)*/
#define CACHE_Y_RANGE 1088
#define CACHE_LOG2_Y_RANGE 11   /* Log2(CACHE_Y_RANGE)

#define CACHE_Z_LOG2SIZE 1      /* Log 2 third dimension                  */
#define CACHE_Z_RANGE 256
#define CACHE_LOG2_Z_RANGE 8    /* Log2(CACHE_Z_RANGE)                    */

#include <mdsc-api.h>
```

Figure 5. Pseudo-code for GLCM

```
int GLCM[256][256];

for(i=1; i < img_height-1; i++)
  for (j=1; j < img_width-1; j++)
  {
      GLCM[ img[i][j] ][ img[i-1][j-1] ]++;
      GLCM[ img[i][j] ][ img[i-1][j ]  ]++;
      GLCM[ img[i][j] ][ img[i-1][j+1] ]++;

      GLCM[ img[i][j] ][ img[i][ j-1]  ]++;
      GLCM[ img[i][j] ][ img[i][ j+1]  ]++;

      GLCM[ img[i][j] ][ img[i+1][j-1] ]++;
      GLCM[ img[i][j] ][ img[i+1][j ]  ]++;
      GLCM[ img[i][j] ][ img[i+1][j+1] ]++;
  }
```

is 256KB, 256×256×4 bytes. This is the entire size of the Local Store and this is only for one color component of the image. Differently from the source image, it is not possible to determine in advance which position of the matrix will be accessed to make use of the DMAs. Using DMA requests to access each position of the matrix would lead to huge number of DMA requests that would slowdown the performance by 2 orders of magnitude, making this solution unfeasible.

Photos, however, usually exhibit large amounts of spatial redundancy, which is exploited by image compression algorithms. The same type of redundancy can be exploited here by caches. Because the change of color is usually smooth, two-dimensional portions of the GLCM matrix are likely to be accessed close in time, in other words, the spatial redundancy of photos is translated as temporal redundancy when updating the matrix.

H.264 Motion Compensation

Motion Compensation (MC) is the process of copying an area of the reference frame to reconstruct the current frame. For advanced video codecs such as H.264, both the reference frame and the Motion Vectors (MV) need to be calculated. In H.264, this process is known as Motion Vector Prediction (MVP) and is part of the MC. Only after the MVP it is possible to request the necessary data to reconstruct the frame. In H.264, MVs can span half of the vertical frame size and it is possible to have up to 16 frames as candidates for reference frame. This makes it impossible to speculatively load all possible areas in advance. In our Cell implementation of macroblock (MB) decoding, the MC kernel is the most time consuming, representing 62% of the total execution time. It requests the reference area through DMA transfers and waits until data is present in the Local Store. The rest of the execution time is spent in DMA data in and out (excluding the reference area) 14%, deblocking filter 17%, and Inverse Discrete Cosine Transform 7%. The memory requests represent 75% of the execution time of the MC kernel. These numbers show the importance of improving the performance of MC.

The unpredictability of the data accesses in MC causes two significant problems on scratchpad memory-based processors. The first problem is that the data transfer cannot be overlapped with the computation. The process has to wait for the data to be transferred to the scratchpad memory. Because H.264 allows very fine grained area to be copied, up to 4x4 pixels, the waiting time for the data can be significant. The second problem is that the data locality cannot be exploited. It is difficult to keep track of the memory area present in the scratchpad memory and new data must be requested for each macroblock (MB) partition. Because the MVs are usually small and not randomly distributed, the same area can be copied several times. Zatt

et al. (2007) show that caching the reference area can save up to 60% bandwidth and more than 75% of the memory cycles.

First, the data locality exhibit by H.264 MC is investigated. H.264 sequences from HD-VideoBench (Alvarez et al., 2007) are used as input for the experiments. Each video sequence consists of 100 frames in standard (SD), high-definition (HD), and full high-definition (FHD) resolutions at 25 frames per second.

To evaluate the data locality, the number of bytes requested from memory is measured. For the measurement, the motion vectors and reference indices are extracted from the encoded sequences for each MB partition. Because of the MC quarter-pixel precision, adjacent additional areas need to be fetched from the memory. For vertical filtering, five extra pixels are required for each line, while for horizontal filtering, five extra lines are required. Details of the MC implementation can be found in (Azevedo et al., 2007).

A tool was developed to translate the extracted MVs to memory requests in the DineroIV (Edler & Hill, 2010) cache simulator input format. DineroIV was used to report the requested number of bytes for each sequence. Three simulations were performed and the results are reported in Figure 6. The first simulates a 1-byte cache to depict the temporal data reuse of MC. The second simulation reports the data traffic for a 16-byte cache with a 16-byte line size. The third simulation reports the data traffic for a 64KB direct mapped cache with 64-byte cache lines.The size of the original uncompressed sequence is presented as file size. The difference between the first and second simulations with the original file size shows the amount of data being reused.

The results show that the sequences exhibit data locality. In case of the 1-byte cache, the data locality is temporal, while in the second case, of the 16-byte cache, the reported data locality is both spatial and temporal. The Riverbed sequences do not exhibit a lot of data reuse. These sequences use mostly intra prediction MBs that uses neighboring pixels to predict the area to be reconstructed, thus not making use of MC. MC references about twice the volume of data of the original sequence (1-byte cache). But, because of memory alignment constraints, the actual volume of transferred data is about 3.5 times the volume of the original sequence (as shown by the 16-byte cache result). The 64KB cache reduces the volume of data transferred by 34% compared to the 16-byte cache. It reduces the data volume to 2.3 times of the original sequence. This indicates that the cache is capturing part of the data locality of the MC. It can be improved as it is an unified cache capturing the three color component access, thus increasing conflicts.

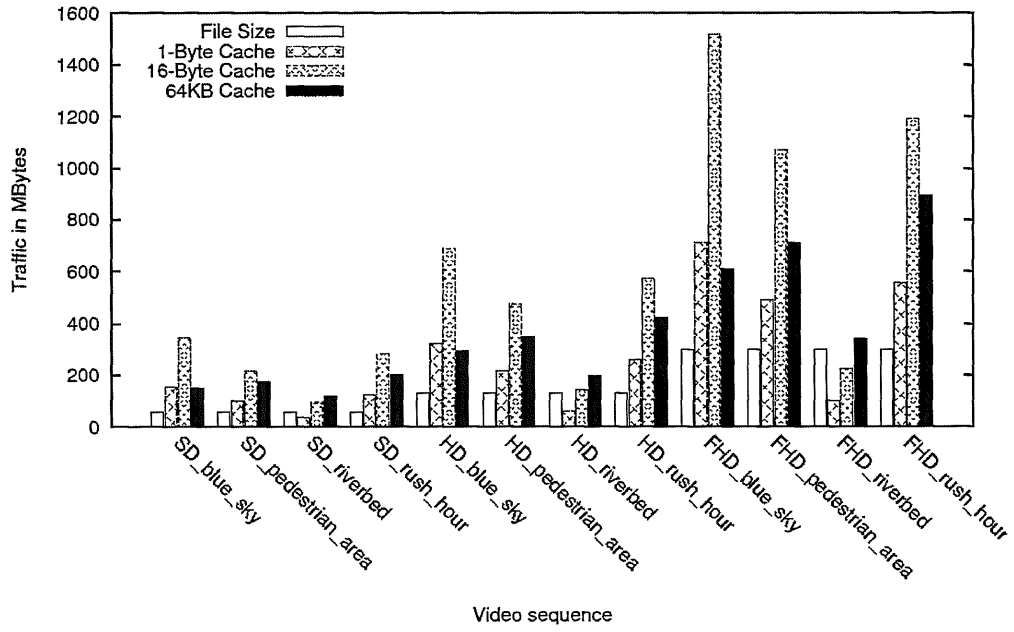
The MDSC reference frame number and the vertical and horizontal coordinates of the MV.This access method enables to exploit the access pattern as it exposes pattern specific information. Each block of the MDSC is an x×y rectangular area of a frame. The x and y values and their ranges are configurable at runtime.

Because of the data locality characteristics of MC, a fully associative configuration was selected, as it reduces the number of conflicts. A fully associative cache is possible because of the small number of blocks present in the implementation, as will be shown in the next section. To support different video resolutions, the MC has to use a dynamic configuration of the MDSC.

MC Enhancements

The video frames are stored in YCbCr format instead of the RGB format, and each component is stored in a separated data structure. To increase the compaction, the color components (Cb and Cr) are subsampled 1:4 as they are less perceptive to the human eye. The MVs are the same for all components, but, because of the subsampling, they need to be adjusted for the Cb and Cr components. The MDSC configuration exploits this feature and checks and requests at once all components. This reduces the number of accesses to the MDSC by a factor of 3 and overlaps the memory requests, thus reducing the memory latency.

Figure 6. Data locality in MC



Four additional enhancement strategies that are employed to reduce the number of accesses to the MDSC are described below. Each strategy builds upon the strategies presented above it.

Extended_X: To reduce the number of accesses to the software cache, an extended line technique was implemented based on the technique described in (Azevedo et al. (2007)). The maximum line size to be accessed is 21 pixels. This consists of the 16 pixels of the maximum MB partition plus 5 extra pixels for quarter-pixel filtering. These pixels can be spread over three 128-bit words. In this technique not only the pixels corresponding to 2D cache block are fetched, but also the 32 pixel columns to the right of the block. Adding these as extra columns for each cache line guarantees that all the data that need to be filtered are present in the cache. This reduces the number of cache accesses per line from 2, or occasionally 3, to 1. Note that this implies that some pixels can be present twice in the cache, once as part of the macroblock it pertains to and once stored with the macroblock to the left of its macroblock. When accessing the cache, however, the first one is returned. Because only read-only data is cached, this does not cause inconsistency problems.

Extended_XY: This technique is an extension of the Extended_X technique and can be applied only when the vertical span of the block is equal or larger than 32 lines, as the number of line needs to be a power of 2. Because the maximum MB partition plus the additional area are 21 lines long in the vertical direction, just two accesses to the MDSC are sufficient to guarantee that the data are present in the cache. Only the first and last lines of the partition need to be accessed. The border between the two cached areas is found by masking the y coordinate of the MV with the height of the block.

SIMD: Since the Cell SPE is a SIMD architecture, a natural step to improve perfor-

mance is to vectorize the tag search. Each tag is a 32-bit integer and the SPE allows vector operations with four 32-bit words. In this optimization, four positions of the tag array are compared simultaneously with the searched tag. Once the tag is found, each of the four positions of the tag array is compared individually to find the block index.

Fixed: As previously stated, the parameters for the MDSC are configurable at runtime. In this version, the cache parameters were fixed, meaning that loop boundaries are known at compile time. This allows for certain loop optimizations to be performed, including the elimination of branches and loop unrolling.

EXPERIMENTAL METHODOLOGY

This research focuses on the performance of the cache access functions. Because of that, the kernels that access the cache will be measured. For the MC, the access to the reference area is evaluated. For the GLCM, the whole function is measured. This is because the GLCM only performs a load, an add, and a store for each position. The images are loaded through explicit DMA transfers and do not make part to the GLCM calculation.

The measurements were performed on a Sony Playstation 3 (PS3). The Cell in the PS3 has 6 out of the 8 SPEs available. One SPE is disabled for redundancy purposes and another one is used by the system for resources access management. Another important characteristic is that the PS3 has only 256MB of RAM. This small amount of memory causes memory swaps with the disk. For this reason, the MC kernel was modified to access only 5 frames, which corresponds to the number of frames in the decoder frame buffer. Otherwise the DMA transfer time is doubled due to memory (de) allocation routines by the OS.

To measure the performance of the kernels, SPU hardware decrementers were used. The decrementer runs at a smaller frequency than the processor itself. In our case, it runs at 78.8 MHz, which is 40 times slower. This approach is not suitable for fine grain profiling, but is sufficiently accurate to measure the performance of functions. The *spu_read_decrementer* and *spu_write_decrementer* intrinsics are used to access the decrementer.

EXPERIMENTAL RESULTS

The HDVideoBench (Alvarez et al., 2007) is used as benchmark for the experiments. Each video sequence is composed by 100 frames in standard (SD), high-definition (HD), and full high-definition (FHD) resolutions at 25 frames per second.

All results were obtained using a single SPE. The experiments were not performed using several cores because the parallelization strategy would influence the results and would change the focus of the work. Both kernels can run in a multicore environment without cache coherency. The GLCM kernel could have a separated matrix for each core and process a slice of the frame. After finishing the processing, the matrices would need to be added together for the final result. For MC the cache is a read-only, thus is does not need cache coherency to work in a multicore environment.

GLCM Results

To generate input for GLCM, the first frame of each HDVideoBench sequence was transformed into an RGB image and each component was processed. Several configurations of a 4-way set associative 64KB MDSC were tested to determine the optimal configuration. The number of sets ranged from 4 to 64, the number of lines from 1 to 32, and the line size from 128 to 1024 bytes. Figure 7 depicts the results of all possible configurations that respect the 64KB cache size. In this figure each bar is labeled as S×L×B, where S is the base-2 logarithm of the number of sets, L is the base-2 logarithm of the number of lines in a 2D block, and B is the base-2 logarithm of the line size in bytes. For instance, 4×3×7 denotes the configuration

with 16 sets, 8 lines, and 128 bytes per line. In other words, the block size of this MDSC configuration is 8×128. As shown in the figure, the GLCM performs better with a higher number of sets with a shorter line size. Because of these characteristics a fully associative MDSC was not evaluated as it requires the opposite characteristics to perform well.

The best performing MDSC configuration consists of 64 sets and uses a block size of 1×256 bytes (6×0×8). Surprisingly, 1-dimensional blocks yield the highest performance. The reason for this is the latency for strided access. The reduction in miss rate, when increasing the block height, is not larger than the increase in the latency for longer DMA list request. However, the performance of the configurations 6×1×7 and 4×3×7 that uses blocks of 2×128 bytes and 8×128 are less than 1% lower than the best configuration.

Figure 8 compares the performance of the optimal MDSC configuration to the optimal configuration of the IBM SC. Experimentally we determined that the optimal 64KB IBM SC configuration for GLCM is 4-way set-associative, consists of 128 sets, and uses a line size of 128 bytes. For comparison, the time taken by the GLCM kernel when the GLCM matrix fits in the Local Store is also depicted. To achieve this, the image color resolution had to be quantitized to 6 bits.

Compared to the IBM SC, the MDSC provides an 8% improvement on average. This performance improvement is due to the lower miss rate achieved by the MDSC. For example, for the FHD BlueSky sequence (denoted FHD/BS in Figure 8, the MDSC incurs a miss rate of 2.4%, while the IBM SC incurs a miss rate of 2.6%. This 0.2% difference in miss rate translates to an 8% increase in memory requests by the IBM SC compared to the MDSC. It also increases the number of times the miss handling code of the software cache is executed. The miss handling code is much more time demanding than the hit branch as it has to choose a block to replace, calculate the block memory address based on the old tag, and issue a request for the new block.

Figure 7. Time taken by the GLCM kernel for several MDSC configurations

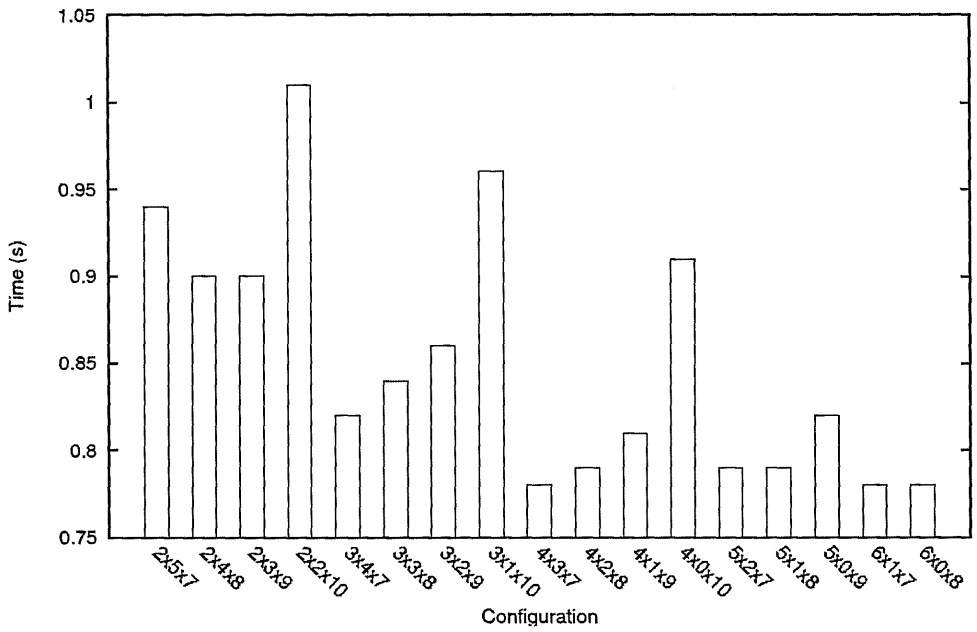
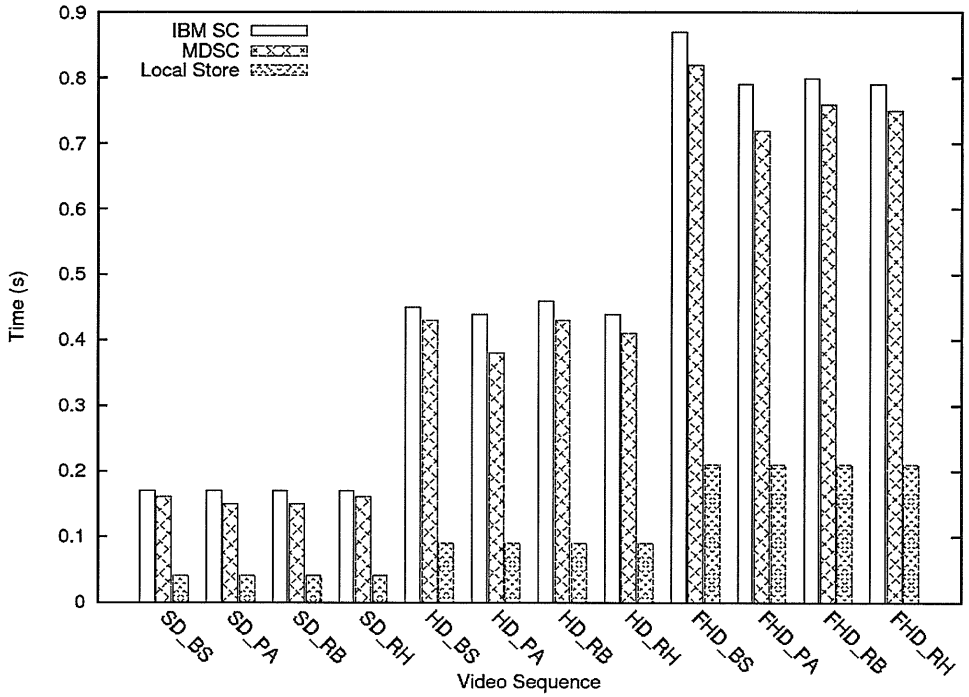


Figure 8. Time taken by the GLCM kernel for several video sequences when the optimal IBM SC configuration is employed (IBM SC), when the optimal MDSC configuration is employed (MDSC), and when the GLCM matrix would fit in the Local Store



The MDSC set hash function, based on indices instead of linear addresses, reduces the number of conflict misses compared to the IBM SC. The MDSC hash function more equally distributes the number of accesses over the sets. For example, for FHD BlueSky the average deviation of the number of accesses to each set of the MDSC is reduced by 18% when compared with the IBM SC with the same configuration. With a better distribution of accesses the number of replacements is lower, which reduces the miss rate.

Compared to when the GLCM matrix would fit in the Local Store, which is included only for comparison, the MDSC incurs a slowdown of 3.75 on average. Considering that the GLCM kernel consists of simple processing and the fact that an access to the MDSC takes around 20 cycles, the MDSC (as well as the IBM SC) quite efficiently bridge the memory gap.

MC Results

In order to determine the optimal 2D block size of the MDSC for the MC kernel, the design space was explored. The MVs from the HDVideoBench sequences were used as input. MVs and reference indices are extracted from the encoded sequences for each MB partition. Blocks of size $2^n \times 2^m$ were tested, for n between 1 and 6, and for m between 5 and 8. For each block size, the miss rate was calculated. The size of the MDSC was fixed at 96KB, 64KB for the Y components and 16KB for the Cb and Cr components each. As noted in Studied Applications section, the MDSC for MC is fully associative. None of the enhancement discussed in MC Enhancements were considered in this exploration.

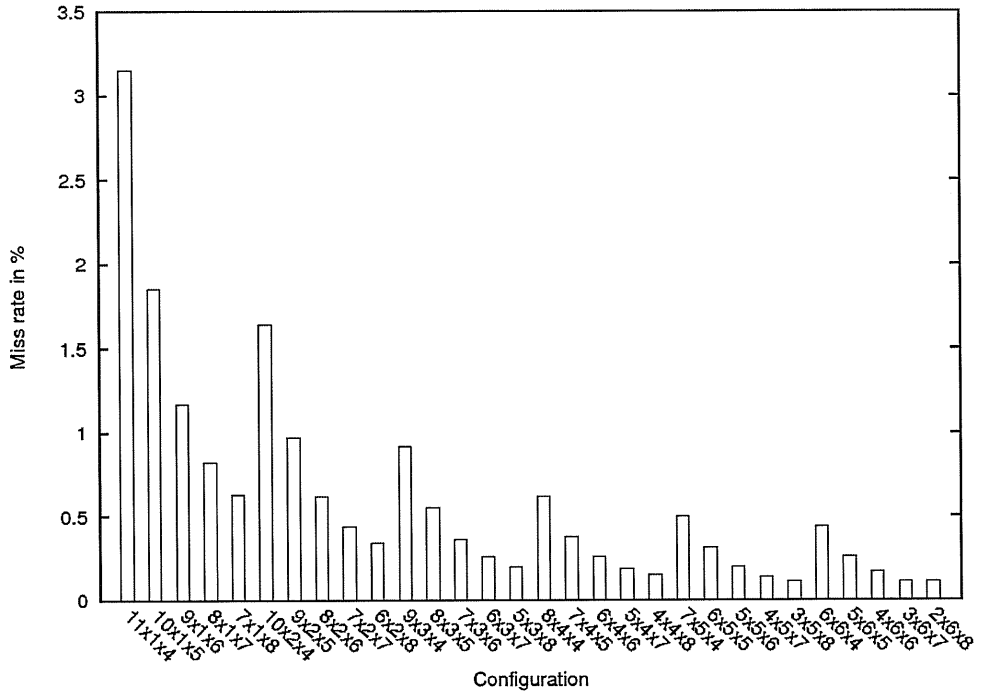
Figure 9 depicts the miss rate for each design point. It uses the same labeling style

as previous figures with base-2 logarithm of the number of blocks in the cache, number of lines per block, and line size in bytes. Not surprisingly, the miss rate decreases when the 2D block size increases. The results show that the 8×64×128 blocks exhibit a miss rate of 0.11%, and that the 8×32×256 and 4×64×256 exhibit the same miss rate. The 32×256 block was selected because, as depicted in Figure 2, fetching 256 bytes is as efficient as fetching fewer bytes. 32 rows also allow the use of the Extended_XY enhancement methodology. The 0.11% miss rate reduces the total number of DMA transfers to 32% of the implementation using hand programmed DMA requests.

A similar exploration was performed for the IBM SC. The IBM SC was configured to be 4-way set associative and the line size was varied from 16 to 256 bytes. As for the MDSC, a 64KB cache was used for the Y components and two 16KB caches for both the Cb and Cr components. It should be noted that unlike the MDSC the IBM SC uses three separated caches, one for each component. The IBM SC uses the Round-Robin replacement policy. The best performing SC configuration uses 256-byte lines and has a miss rate of 8.6% for Y component. Because the IBM SC can use only 1-dimensional blocks while the application processes 2-dimensional blocks, the IBM SC miss rate is much higher that the miss rate of the MDSC.

Figure 10 breaks down the time taken by the MC kernel when the baseline MDSC (without enhancements) is employed into time needed to access the MDSC and time required for the DMA transfers. For comparison, the time taken by a version that does not use a software cache but fetches the reference areas from main memory using explicit, hand-programmed DMA transfers is also included and labeled *DMA*. The results include the time for frame border detection, the time to fetch the additional quarter-pixel area, the time to fetch additional Cb

Figure 9. Miss rate incurred by the MC kernel for different configurations of a 96KB MDSC



and Cr components, and the handling of 128-bit alignment constraints. The border detection and the alignment calculation are included because in the DMA time because they are overlapped with memory transfers and account for less than 1% of the total DMA time. The baseline MDSC implementation performs an MDSC access for every 16-byte quadword. It is clear from the figure that the majority of the time is spent accessing the cache rather than transferring data. This is because for every MDSC access the index has to be calculated and the tag has to be compared to the tags stored in the software cache. This overhead is relatively time consuming compared to the time taken by the DMA transfers. Furthermore, for all but one sequence, the version that uses hand-programmed DMAs is faster than the version that employs the baseline MDSC. The used sequences are composed of 100 frames and are 4 seconds long (25 fps). The proposed enhancements exploit the known memory access behavior to reduce this overhead.

Figure 11 compares the performance of the direct DMA version of MC, the IBM SC, the baseline MDSC, and the MDSC extended with the enhancements described in Section MC Enhancements. It depicts the time in seconds to fetch the reference area from main memory to the SPE scratchpad. Our baseline for comparison is the DMA version of MC. The line labeled *Real Time* depicts the performance required for real time processing. As in the previous experiment, the DMA include border and alignment handling while the other versions depict the time required for MC only.

When the number of MDSC accesses is reduced with the Extended_X technique, an average 25% improvement over the DMA version is achieved. The Extended_X technique reduces the number of MDSC accesses by a factor of 2, because it ensures that when the first pixel in a line is present in the MDSC, the entire line is present. Checking only the presence of the first and last line of the MB partition, as is done in the Extended_XY technique, results in only two MDSC accesses per MB partition. This substantially increases the efficiency of the MDSC implementation and achieves an average 60% execution time reduction compared to the direct DMA version of the code.

The SIMD version of the MDSC does not provide an additional performance improvement. Its overhead cancels the benefits because of the small number of 2-dimensional blocks (eight) in the implemented MDSC. Fixing the

Figure 10. Breakdown of the time taken by the MC kernel for different input sequences when the baseline MDSC is employed and the time taken when explicit, hand-programmed DMA transfers are used

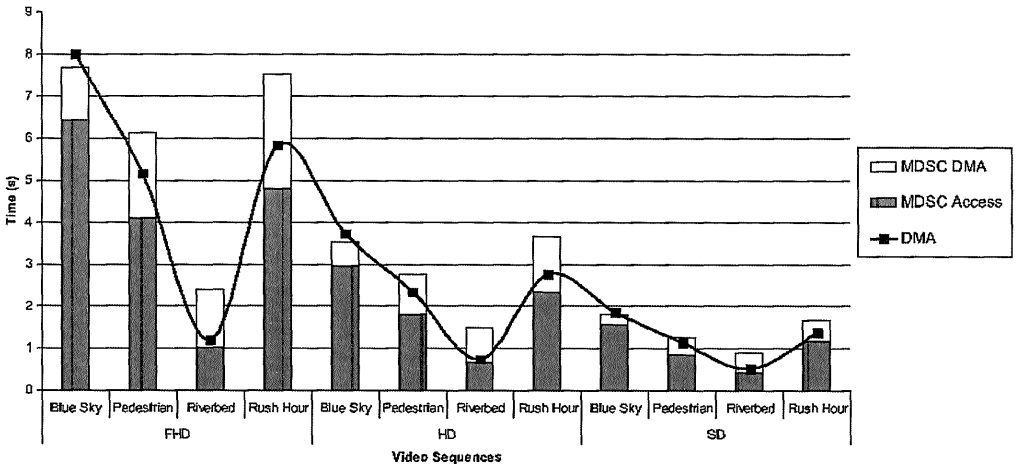
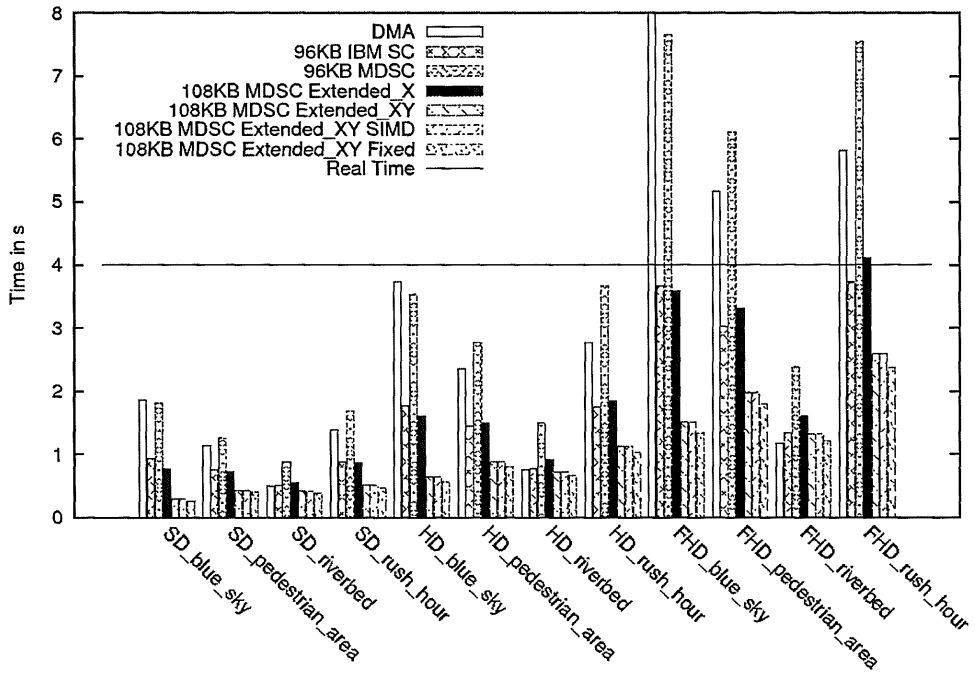


Figure 11. Time taken by MC for the direct DMA version, the IBM SC, the MDSC, and the various MDSC enhancements. The input sequences are 100 frames and 4 seconds long



parameters of the MDSC so that certain compiler optimizations can be performed yields an additional 5% execution time reduction, leading to a total average performance improvement of 65%. Fixing the parameters of the MDSC is similar as using the MDSC static configuration.

Compared to the IBM SC, the MDSC with the Extended_XY enhancement is 37% more performance efficient. The performance improvement increases to 43% when comparing the IBM SC with the Fixed implementation of the MDSC. This improvement is mainly due to the fact that because the MDSC uses 2-dimensional blocks and because the Extended_XY technique ensures that the entire reference area is included in at most two MDSC blocks, two MDSC accesses are sufficient to determine if the reference area is in cache, whereas the IBM SC requires at least one access for every line in the reference area. Also, the MDSC exploits the relationship between the Y, Cr and Cr components to reduce the number of access. If an area is present in the Y cache it then it is present in the Cb and Cr caches, thus only the address calculation is required to access Cr and Cr data.

Overall, the results show that software caches can exploit the data locality exhibited by MC. To obtain actual performance improvements, however, the number of access needs to be minimized. Furthermore, the MDSC allows reducing the number of accesses more than 1-dimensional cache organizations such as the IBM SC, thereby yielding higher overall performance.

CONCLUSION

In this article a Multidimensional Software Cache has been proposed for systems based on scratchpad memories such as the Cell processor. The objectives of the MDSC, next to increasing the programmability, are to exploit the data locality that cannot easily be exploited

by hand-programmed DMAs, to reduce the DMA startup overhead by employing list DMAs instead of several sequential DMAs, and to minimize the number of cache accesses by using large, multidimensional blocks. Furthermore, the cache is indexed by the indices of the base element of the block rather than the memory address, which allows reducing the number of conflict misses. The proposed software cache organization has been evaluated for GLCM and H.264 MC, which is representative of many other multimedia kernels.

The GLCM uses indirect addressing, but because the difference between adjacent pixels is usually small, it exhibits data locality. Somewhat surprisingly, the MDSC configuration that yields the highest performance uses 1-dimensional blocks. However, the performance of two configurations was less than 1% lower. This indicates that in an organization where the two channels could be used simultaneously, the benefits of a 2-dimensional block would be more pronounced. Compared to the heavily optimized IBM software cache, the MDSC improves performance by 8%. The indexing function of the MDSC reduces the number of conflicts and accounts for the performance improvement.

For MC, first the data locality that it exhibits has been analyzed. This analysis shows that MC has a significant amount of data locality that could be exploited by a (software) cache. Then the data access pattern of MC has been evaluated to design a software cache that exploits it. The proposed software cache stores frame areas instead of blocks of consecutive memory locations. In other words, it uses 2-dimensional cache blocks. Enhancements have been proposed to reduce the number of accesses to the MDSC and its associated overhead.

For MC, the experimental results show that without tuning the software cache to the application, the performance degrades compared to an implementation that uses hand-programmed DMAs and does not attempt to exploit the data locality. This performance degradation is the result of the access overhead to the software cache to check for the presence of the desired data. The enhancements proposed in order to reduce the number of accesses to the MDSC achieve an average 65% performance improvement over the hand-programmed DMA implementation. For only one sequence (Riverbed), the MDSC did not attain a performance improvement over the DMA version. The reason for this is the lack of data locality in the Riverbed sequence. The software cache overhead can be reduced by using information of the application's access characteristics to reduce the number of cache accesses. Compared to the IBM Cell software cache, the MDCS provides an improvement of 43%, averaged over all video sequences.

The performance of the MDSC reflects the machine organization used for this study. The Cell processor has two memory channels, but a single SPU cannot use both channels. The access latency does not change when one or two SPUs are accessing data simultaneously, as depicted in Figure 2. As future work, the impact of the MDSC on a processor that can communicate via multiple channels simultaneously will be studied. It is expected that this will increase the performance improvements achieved by the MDSC. Also, a lightweight hardware accelerator for software caches will be investigated that reduces the overhead of the MDSC without significantly increasing the area or power consumption.

REFERENCES

Alvarez, M., Salami, E., Ramirez, A., & Valero, M. (2007). HD-VideoBench: A benchmark for evaluating high definition digital video applications. In *Workload Characterization* (pp. 120-125). Washington, DC: IEEE Computer Society. DOI:10.1109/IISWC.2007.4362188

Azevedo, A., Zatt, B., Agostini, L., & Bampi, S. (2007). MoCHA: A bi-predictive motion compensation hardware for H.264/AVC decoder targeting HDTV. In *Circuits and Systems* (pp. 1617-1620). DOI: 10.1109/ISCAS.2007.378828

Balart, J., Gonzalez, M., Martorell, X., Ayguade, E., Sura, Z., & Chen, T. (2007). A novel asynchronous software cache implementation for the Cell-BE processor. In Proceedings of *Languages and Compilers for Parallel Computing . Urbana (Caracas, Venezuela), IL*, 125–140. doi:.doi:10.1007/978-3-540-85261-2_9

Banakar, R., Steinke, S., Lee, B., Balakrishnan, M., & Marwedel, P. (2002). Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of Hardware/Software Codesign*, Estes Park (pp. 73-78). New York: ACM. DOI:10.1145/774789.774805

Chen, T., Zhang, T., Sura, Z., & Tallada, M. G. (2008). Prefetching irregular references for software cache on Cell. In *Code Generation and Optimization* (pp. 155-164). New York: ACM. DOI: 10.1145/1356058.1356079

Edler, J., & Hill, M. D. (2010). *Dinero IV trace-driven uniprocessor cache simulator*. Retrieved January 28, 2010, from http://pages.cs.wisc.edu/~markhill/DineroIV/

Example Library API Reference. (2010). Retrieved January 28, 2010, from https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3B6ED257EE6235D900257353006E0F6A/$file/SDK_Example_Library_API_v3.0.pdf

Gonzalez, M., Vujic, N., Martorell, X., Ayguade, E., Eichenberger, A. E., Chen, T., et al. (2008). Hybrid access-specific software cache techniques for the Cell BE architecture. In *Parallel Architectures and Compilation Techniques* (pp. 292-302). New York: ACM. DOI: 10.1145/1454115.1454156

Gschwind, M., Hofstee, H., Flachs, B., Hopkins, M., Watanabe, Y., & Yamazaki, T. (2006). Synergistic processing in Cell's multicore architecture. *IEEE Micro, 26*(2), 10–24. .doi:10.1109/MM.2006.41

Kahle, J., Day, M., Hofstee, H., Johns, C., Maeurer, T., & Shippy, D. (2005). Introduction to the Cell multiprocessor. [Riverton, NJ: IBM Corp.]. *IBM Journal of Research and Development, 49*(4), 589–604. doi:10.1147/rd.494.0589

Lee, J., Seo, S., Kim, C., Kim, J., Chun, P., Sura, Z., et al. (2008). COMIC: A coherent shared memory interface for Cell-BE. In *Parallel Architectures and Compilation Techniques* (pp. 303-314). New York: ACM. DOI: 10.1145/1454115.1454157

Power Architecture Version 2.02. (2010). Retrieved January 28, 2010, from http://www-106.ibm.com/developerworks/eserver/library/es-archguide-v2.html

Senthil, G., Gudla, S., & Baruah, P. K. (2008). Exploring software cache on the Cell BE processor. In *High Performance Computing* (p. 5).

Seo, S., Lee, J., & Sura, Z. (2009). Design and implementation of software-managed caches for multicores with local memory. In *High Performance Computer Architecture* (pp. 55-66). DOI:10.1109/HPCA.2009.4798237

Shahbahrami, A., Borodin, D., & Juurlink, B. (2008). *Comparison between color and texture features for image retrieval*. Circuits, Systems and Signal Processing.

Zatt, B., Azevedo, A., Agostini, L., Susin, A., & Bampi, S. (2007). Memory hierarchy targeting bi-predictive motion compensation for H.264/AVC decoder. In *VLSI* (pp. 445-446). Washington, DC: IEEE Computer Society. DOI:10.1109/ISVLSI.2007.64

Arnaldo Azevedo received the BSc degree in computer science from the UFRN University, Natal, RN, Brazil, in 2004 and the MSc degree in computer science from UFRGS University, Porto Alegre, RS, Brazil, in 2006. Since 2006, he is a doctoral candidate in the Computer Engineering Laboratory of the Faculty of Electrical Engineering, Mathematics and Computer Science of Delft University of Technology, the Netherlands. He is currently investigating multimedia accelerators architecture for multi-core processors.

Ben Juurlink is professor for Embedded Systems Architectures in the Faculty of Electrical Engineering and Computer Science of the Technische Universität Berlin (DE). He has an MSc degree from Utrecht University (NL) and a PhD degree from Leiden University (NL). In 1997-1998 he worked as a post-doctoral research fellow at the Heinz Nixdorf Institute in Paderborn (DE). From 1998 to 2009 he was a faculty member of the Computer Engineering Laboratory of the EEMCS faculty of Delft University of Technology (NL). His research interests include multi- and many-core processors, embedded processors, low-power techniques, and memory systems. He has (co-)authored more than 80 papers in international conferences and journals and received a best paper award at the IASTED PDCS conference in 2002. He is the leader of the NWO project Archivire and a work package leader in the EU FP6 project SARC. He is a senior member of the IEEE, a member of the ACM, and a member of the HiPEAC NoE. He served in many program committees and is the subject area editor for the field of Processor Architectures of the Elsevier journal on Microprocessors and Microsystems (MICPRO) – Embedded Hardware Design. He has supervised four PhD students and is currently supervising five PhD students.