An Instruction to Accelerate Software Caches

Arnaldo Azevedo[1] and Ben Juurlink[2]

[1] Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics,
and Computer Science
Delft University of Technology
Delft, the Netherlands
a.p.pereiradeazevedofilho@tudelft.nl
[2] Embedded Systems Architectures
Faculty of Electrical Engineering and Computer Science
Technische Universität Berlin
Berlin, Germany
b.juurlink@tu-berlin.de

Abstract. In this paper we propose an instruction to accelerate software caches. While DMAs are very efficient for predictable data sets that can be fetched before they are needed, they introduce a large latency overhead for computations with unpredictable access behavior. Software caches are advantageous when the data set is not predictable but exhibits locality. However, software caches also incur a large overhead. Because the main overhead is in the access function, we propose an instruction that replaces the look-up function of the software cache. This instruction is evaluated using the Multidimensional Software Cache and two multimedia kernels, GLCM and H.264 Motion Compensation. The results show that the proposed instruction accelerates the software cache access time by a factor of 2.6. This improvement translates to a 2.1 speedup for GLCM and 1.28 for MC, when compared with the IBM software cache.

1 Introduction

The shift towards multicore architectures to increase performance while maintaining within the power envelop has brought new challenges. One of these challenges is the design of a memory hierarchy tuned for power efficiency. Scratchpad memories reappeared as a solution for the memory hierarchy because they are very efficient in terms of power and performance [1]. DSPs, GPUs, and more notably the Cell processor [2] are examples of high end multicore processors that use scratchpad memories instead of regular caches.

Multimedia applications feature mostly predictable data sets and access patterns, making it possible to transfer the necessary data before the computation. These data transfers usually need to be explicitly exposed by the programmer. This feature also makes possible to overlap computation with data transfers by means of double buffering techniques. Scratchpad memories also have predictable

latencies. These characteristics make scratchpad memories a common choice for embedded multimedia processors.

However, some multimedia applications do not feature predictable data sets and access patterns. This is the case, for example, in Motion Compensation (MC), Grey Level Co-occurrence Matrix (GLCM), Texture Mapping, and Computed Tomography Processing. These applications present two significant problems for 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. The second problem is that data locality cannot easily 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 access. An alternative is the implementation of a software cache (SC). SCs are software structures that offer the same functionality as hardware caches, i.e., they store data that can be retrieved later upon requests [3]. Differently from hardware caches, SCs can be easily customized to the characteristics of the application. Implementing a software cache for the Cell processor is a current topic of research in the community [4][5][6]. However, a software cache still has high overhead, representing up to approximately 50% of the application execution time [7]. Such overhead can harm the application performance when compared with hand programmed DMA transfers. Such evaluations are not presented in the cited works. In fact in [8] it was shown that for H.264 Motion Compensation the software cache access time is higher than the actual memory transfer time.

In this paper, we propose to accelerate software caches using limited hardware support. While traditional hardware caches are area and power consuming, software caches have high processing overhead. Our aim is to close the performance gap between software caches and hardware caches while keeping the hardware overhead to a minimum. For this purpose we propose a cache look up instruction called Lookup_SC. The Lookup_SC instruction is designed to enhance the performed of the MultiDimensional Software Cache proposed in [8]. The instruction calculates the tag and the set, searches for the tag in the tag array, and returns a hit or miss flag together with a pointer to the address of the requested data. In case of a miss, it is handled by software functions. For the experimental evaluation we use the Cell SPE [9] processor core.

This paper is organized as follows. The targeted software cache is presented in Section 2. The proposed instruction is detailed in Section 3 and the simulation methodology is described in Section 4. Results are presented and analyzed in Section 5 and conclusions are drawn in Section 6.

2 Multidimensional Software Cache

In this section, we briefly describe the Multidimensional Software Cache (MDSC), introduced in [8]. The MDSC has the ability to mimic in the SC the logical organization of the accessed data structure. Specifically, cache blocks are 1- to 4-dimensional. Fig. 1 illustrates 1D, 2D, and 3D cache blocks storing a video sequence composed of the *Lena* image. An 1D MDSC is similar to a conven-

(a) Eight 1D MDSC blocks.

(b) Two 2D MDSC blocks, four lines high.

(c) Two 3D MDSC blocks, four lines high and 2 images deep.

Fig. 1. Examples of 1D, 2D, and 3D MDSC blocks.

tional hardware cache as its stores a number of consecutive bytes (from external memory) in each block, as depicted in Fig. 1(a). A 2D cache can be used to store rectangular image areas, as depicted in Fig. 1(b). In this example, four vertically consecutive segments of the image lines are allocated per MDSC block. 3D MDSC blocks are a set of consecutive (in the third dimension) co-located (i.e., with the same vertical and horizontal coordinates) 2D blocks. A 3D MDSC can be used to store areas of a sequence of video frames, as depicted in Fig. 1(c). Similarly, a 4D MDSC, not depicted, can be used in Multiview video processing or animated 3D representations as its blocks are sets of 3D blocks. Besides multidimensional cache blocks, another characteristic of the MDSC is the use of data structures indices to index the cache. So, instead of consulting the cache as $access\_SC(\&datastructure[i][j])$ we propose to access it using $access\_MDSC(\&datastructure, i, j)$. Although similar, the second access method makes explicit more information about the data structure and the access.

This approach makes the MDSC differ from a regular cache in two ways. First, it differs in the tag and set calculation. For the MDSC, the tag is a concatenation of each index and the set is a mask operation based on the indices. The second difference is the format and the load of the multidimensional block. The multidimensional block is formed by a set 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 the DMA requests for each line are grouped in a so called DMA list.

The MDSC has two advantages over regular software caches, such as the IBM Cell software cache (IBMSC) [3]. First, it reduces the memory latency by grouping several memory requests, since a single DMA list operation has a lower latency than several sequential DMAs [8]. The second advantage is that the MDSC can be used to reduce the number of accesses to the SC [10]. As the shape of the cache block and its boundaries are known, a single cache lockup is

necessary for accessing an entire block. After the first access, the remaining data can be accessed using simple pointer arithmetic.

The MDSC is associated with a single data structure, i.e., for each data structure there is a separated MDSC. However, there are two ways to cache several data structures. If they have the same dimensions, it is possible to have one SC that stores the different data structures. This is done by using an extra dimension to represent the data structures, where each index value would point to a different data structure. This works even if the data structures do not have a contiguous address space as the base address of the data structure is a parameter of the software cache access function. Otherwise, an MDSC for each data structure is necessary.

3 MDSC Accelerator

One of the main drawbacks of a software cache is the runtime overhead. In [8], it is shown that the MDSC access time is larger than the data transfer time for H.264 Motion Compensation. To reduce the access time overhead for the MDSC we propose an instruction to accelerate the MDSC access. In this section, we describe the instruction and its functionality.

The proposed hardware module is implemented in the processor pipeline and accessed through a new instruction. This new instruction, called *Lookup_SC*, receives the access parameters and returns if the data is currently present or not and a pointer to where the data is stored in the SPE Local Store (LS). It performs the following steps:

1. calculate the set based on the input indices, shifts and masks;
2. calculate the tag based on the input indices, shifts and masks;
3. check if the tag is present in the tag array;
4. calculate the offset of the data inside the cache block (can be done partially in parallel with the previous steps);
5. if the tag is present, uses its position to calculate the address of the cache block;
6. return the hit/miss flag, the position of the data, the tag, and the set and the tag position.

In software these steps take about 45 cycles on the SPE. Although it is a small number of cycles, since the lookup needs to be performed to access each data value even this small cycle count leads to a significant performance penalty. The actual reading and writing of data from/to main memory were excluded from the hardware module to keep the hardware complexity and costs at a minimum level. This is feasible because the MDSC yields high hit rate compared with traditional caches.

This proposed hardware accelerator differs from a traditional hardware cache in a number of ways. First, there is no need to perform the cache access in a single or few cycles. The high performance cache lookup implementation present in regular hardware caches can be replaced by a pipelined, more latency tolerant

structure integrated in the existing core datapath. Second, there are no hardware handling of misses and no expensive replacement policies. These tradeoffs, while not giving the best cache configuration, aim at keeping the hardware and power consumption overheads at a minimum level while increasing the performance compared to the complete software implementation.

The instruction has 3 operands: 2 input registers and one output register. The first register contains a maximum of four 32-bit indices of the multidimensional cache. 32-bit wide integers were chosen as this is the most common type for loop indices, likely to be used to access the cache. The second input register is used to pass the cache parameters. These parameters are the base address of the software cache tag array in LS, the size of the cache block, the number of sets, shifts and masks for each index, and the size in bytes of each dimension. A 128-bit register is capable of storing this information because of the small address space of the LS (256KB), the small range of the parameter (usually 8 bits), and the wide register in the SPE. All these parameters only need to be packed together once, as they do not change throughout the execution.

There are three possible places to store the tag array. First, a new tag array structure could be placed on the SPE. This, however, would require a large area overhead as registers consume a relatively large amount of resources. Second, the tag array could be placed in the register file. This option would require to modify the compiler so that it does not allocate a specific number of registers. This option would also require an indexed register access that would complicate the pipeline. The third option is to use the LS. This option has the advantage of allocating the tag array in the largest memory storage in the processor. This avoids increasing the pressure on the register file and the extra resources of an register file for the tag array. This option has a higher latency than the previous ones, however, as a Load has a 6-cycle latency. However, this larger latency is a good tradeoff as it offers an easier pipeline integration, as presented next.

The resultant hardware behavior is depicted in Fig. 2. W, Z, Y, and X are the 32-bit indices for each of the dimensions. $Tag\_Array\_Address$ is the 16-bit address of the tag array in the LS divided by 16 to save bits. Each dimension of the MDSC is associated with a *shift*, a *mask*, and a *dimensionsize* parameter. Each of these parameters is 8bits wide. *shift* is the base-2 logarithm of the size of the corresponding cache block dimension. *mask* is the base-2 logarithm of the dimension range (maximum number of bits of the index) minus the corresponding *shift* added to the lower indices *masks*. *dimensionsize* is the base-2 logarithm size of the cache block dimension added to the lower cache block dimensions sizes. The *address* is actually the index in the memory array that acts as the cache memory. The *n_sets* parameter carries the number of sets in the MDSC. In practice, *n_sets* is passed already decremented by one.

Despite the number of operations in the LookUp_SC instruction, it remains relatively simple due to the reduced size of its operands, its simple operations, and the parallelism of the operations. The LookUp_SC instruction hardware is composed by the set calculation, the tag formation logic, the comparator, and the address calculation logic. The tag formation and the set calculation

```
set = ((((W >> shift[w]) ^ (W >> (shift[w] + 1))) +
        ((Z >> shift[z]) ^ (Z >> (shift[z] + 1))) +
        ((Y >> shift[y]) ^ (Y >> (shift[y] + 1))) +
        ((X >> shift[x]) ^ (X >> (shift[x] + 1)))) & (n_sets-1));

tags = Load((Tag_Array_Address + set) << 4);

tag = 1 + ((((W >> shift[w]) << mask[w]) +
            ((Z >> shift[z]) << mask[z]) +
            ((Y >> shift[y]) << mask[y]) +
             (X >> shift[x])) << 2);

address = (W & ~(0xFFFFFFFF << shift[w])) << dimensionsize[w] +
          (Z & ~(0xFFFFFFFF << shift[z])) << dimensionsize[z] +
          (Y & ~(0xFFFFFFFF << shift[y])) << dimensionsize[y] +
          (X & ~(0xFFFFFFFF << shift[x])) << dimensionsize[x];

quad_result = (0, ((block_size*(set<<2)) << 4) + address, tag, set<<2);

if (tags[0] == tag)      return quad_result + (1,               0, 0, 1);
else if (tags[1] == tag) return quad_result + (1,   block_size<<4, 0, 1);
else if (tags[2] == tag) return quad_result + (1, 2*block_size<<4, 0, 2);
else if (tags[3] == tag) return quad_result + (1, 3*block_size<<4, 0, 3);
else return quad_result;
```

Fig. 2. Pseudo-C code of the LookUp_SC instruction.

are series of shifts and masks operations to select the significant bits from the indices. Similarly, the address calculation uses shifts, masks, and adds to find the position of the data inside the cache block. No multiplication is necessary as the cache multidimensional blocks dimensions are powers of 2. The *quad_result* is the quadword to be returned in case of a hit. It consists of four 32-bit integers, the hit flag, the index of the requested data in the data storage array, the tag, and the position of the tag in the tag array, respectively. *quad_result* is finally updated depending of the position of the *tags* quadword in which *tag* is found. The LS is accessed only once to retrieve the tag array (*tags*) vector pointed by *set*.

Most of the hardware components of the accelerator can execute in parallel. Set calculation, the tag formation, and address calculation can be performed in parallel. With the result of the set calculation, *quad_result* can also start being calculated. The set calculation followed by the indices load and the tag formation followed by the tag comparison need to be performed sequentially.

The SPE pipeline is divided into a front-end and a back-end part. The front end pipeline is the same for all instructions and it does the instruction fetch-

```
CACHED_TYPE AMDSC_read( int y, unsigned int x, unsigned int ea){
    vec_uint4 index =  {x, y ,x ,y};
    quadresult = spu_LookUp_SC(index, parameters);

    int address = spu_extract(quadresult, 1);
    int tag     = spu_extract(quadresult, 2);
    int set     = spu_extract(quadresult, 3);
    CACHED_TYPE ret = __cache_mem[address];

    if (unlikely (spu_extract(tagpos, 0) != 0))
      ret = __cache_mem[__cache_miss(y, x, tag, set, ea)];

    return ret;
}
```

Fig. 3. Resulting C code for the MDSC read function integrated with the LookUp_SC instruction.

ing, instruction decoding and access of the register file. There are five back-end pipelines for branch, permute, load/store, fixed point, and floating point instructions. The load/store back-end pipeline would be modified to perform the mentioned computations.

The Lookup_SC instruction can be easily integrated in the existing MDSC code with only a few modifications required. The resulting code is depicted in Fig. 3 where x and y are the indices of structure to be accessed, *ea* is the address of the accessed data structure in the external memory, *__cache_mem* is the MDSC data storing array, and *parameters* are the parameters of the MDSC instance. The first modification is to merge all the indices in a single register, as performed by $vec\_uint4\ index = x, y, x, y;$. Second, the lookup function is replaced by the *spu_Lookup_SC* instrinsic that interfaces the our proposed instruction. The result of the Lookup_SC instruction is then separated into regular integers by *spu_extract*. The *spu_extract* shifts a specified word position in the quadword to the first word of the quadword, where it can be processed as a regular integer. The last step is to replace the calculation of the data address with the result of the LookUp_SC instruction. The miss handling function is not changed.

4 Experimental Methodology

Our evaluation methodology for the proposed accelerator uses two separated and complementary approaches. The first approach uses a Cell SPE core and the accelerator is emulated using a load instruction. The second approach uses the CellSim Simulator [11]. In the simulator the new instruction is added to the instruction set of the SPE.

To emulate the MDSC accelerator in the Cell we proceed as follows. First, the kernel is profiled and broken down into the following parts: *Processing*, consisting of the basic operations of the kernel without the access to the SC; *Access*, consisting of accesses to the SC without considering miss handling or data transfer, only SC hits; *Miss Handling*, consisting of the selection of the cache block to be replaced, the calculation of the memory address, and the update of the cache status; and *DMA*, corresponding to the actual transferring of the data from the external memory to the local store.

The SPE hardware counter is used to profile the kernels. As there is only one hardware counter an incremental approach is used to break down the kernel. First the *Processing* time is extracted by commenting out the MDSC access. To isolate the *Access* time a run of the application is performed with the access to the SC requesting the same indices, thus with only one miss. This is possible because the evaluated kernel's processing times are insensitive to the data being processed. For isolating *Miss Handling* the actual data is used to access the MDSC and the DMA request commands are commented out. The *DMA* time is extracted by running the full kernel and subtracting the time taken by the previous stages.

Second, the Lockup_SC instruction is emulated using existing instruction. Our proposed instruction is comparable with an indexed load instruction and thus the LQX (the SPE indexed load instruction) is used. The LQX instruction performs a 32-bit add to calculate the address to be fetched from the LS. In our proposed instruction, the address calculation depends on the *set* calculation, as depicted in Fig. 2. Because the *set* is 8-bit wide and most of its calculation can be performed in parallel, we assume that using the same latency as the LQX instruction is feasible. The LQX instruction is accessed by an SPU intrinsic is placed in the *C* code. The *Access* time of the target application is then profiled again using the emulated instruction. The accelerated *Access* time is used instead of the *Access* time of the original kernel. This new kernel time is used to estimate the performance gain of the proposed instruction. This is a valid approach because the Lockup_SC instruction only changes the MDSC access time with all the remaining stages intact.

We also evaluate the proposed hardware extension using the CellSim Simulator. It is not possible to use the IBM SystemSim [12] because there is no access to the source code to make the necessary modifications. The CellSim Simulator was used to validate the instruction behavior and to qualitatively analyze the required hardware complexity. CellSim is not cycle accurate as it does not distinguish between the odd and even pipelines of the SPE. To handle this problem, the instruction fetch rate parameter is available. This parameter defines a constant number of instructions that the execution unit fetches per cycle. To tune this parameter, the results of the kernels profiling are used. In our experiments the instruction fetch was set according to each kernel. The difference in number of cycles between the simulators and our measurements is around 7% less for the MDSC accelerated with LookUp_SC. Because of this difference we opted to report the results acquired following our first approach.

In order to evaluate the LookUp_SC instruction, two kernels were selected. The selection was based on the presence of unpredictable access patterns and the suitability for the MDSC. Below we briefly describe each kernel.

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 [13]. In this study, all 9 neighboring pixels are examined.

In this application, the source image can be easily accessed through DMAs. However, the GLCM matrix is too large to be stored in the LS. Because the matrix is indirectly indexed, it is not possible to determine in advance which position of the matrix will be accessed to make efficient use of DMAs. Photos, however, usually exhibit large amounts of spatial redundancy that can be exploited here by caches.

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. Using DMAs to retrieve the reference area leads to 75% of the execution time of the MC kernel being spent on waiting for the DMAs to complete [8]. These numbers show the importance of improving the performance of MC.

The work presented in [10] shows that MC has sufficient locality to be exploited by a cache. It also shows that SC access time dominates the execution time of the kernel. The paper then presented software mechanisms to exploit know access behavior to reduce the number of accesses to the MDSC. E.g. integrating the different color components (Y, Cb, and Cr) into one request.

In this work we focus only on the acceleration given by our proposed instruction. So, in this MC implementation we use independent caches for each color component, as with the IBMSC. This was chosen to isolate the benefits of the proposed hardware.

The HDVideoBench ([14]) is used as input for the benchmarks. Each video sequence consists of 100 frames in standard (SD), high-definition (HD), and full high-definition (FHD) resolutions at 25 frames per second. GLCM input is the first frame of each HDVideoBench sequence that was transformed into an RGB image. The MC input are the motion vectors and reference indices extracted from the encoded sequences for each macroblock partition.

5 Experimental Results

In this section experimental results for the LookUp_SC instruction accelerating the MDSC are presented. First, the impact of the LookUp_SC instruction on the access time of the MDSC for 1 to 4 dimensions is analyzed. Then results for the GLCM and MC kernels are presented and analyzed. Three baseline versions are presented for each kernel. The results of an implementation without a SC are given to state the need/impact of a SC for the given kernel. The second baseline is an implementation using a publicly available SC, the IBMSC for the Cell [3]. The third baseline uses the MDSC implementation of the kernel without acceleration. The MDSC implementation is highly optimized, including manual instruction scheduling, to ensure comparison fairness.

5.1 LookUp_SC

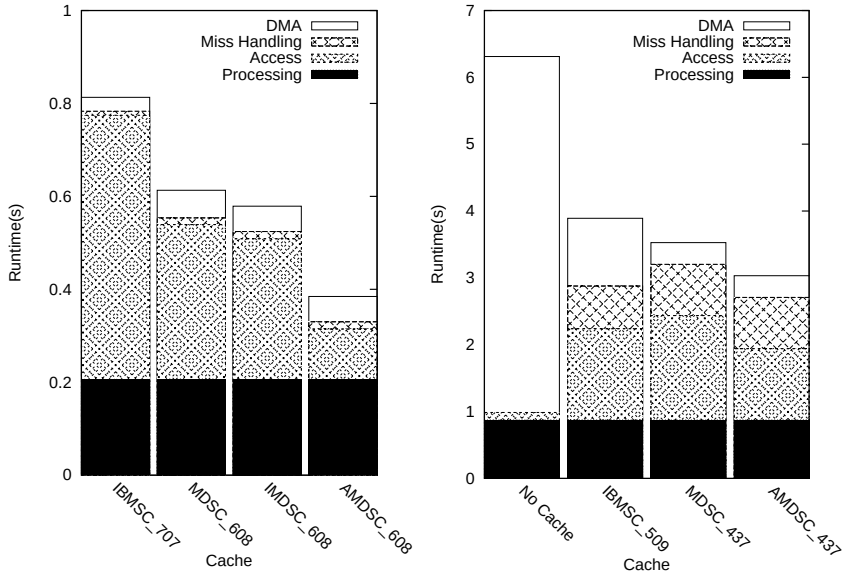
To measure the impact of the proposed instruction we first isolated the cache access functions for IBMSC, MDSC and Accelerated MDSC (AMDSC). It was performed by several runs of a 200K iterations loop containing 10 accesses to the cache. The request was such that only the first cache access would result in a miss. The access function is then modified to a dummy one containing only a return with a valid data that is dependent on the inputs. The cache access time is calculated by subtracting the total runtime of the regular functions by the dummy runtime and then divided by the number of iterations.

In this experiment, the IBMSC has an access time of 57 cycles. The MDSC access time varies from 44 to 48 cycles for 1 to 4-dimensional MDSC instances. The AMDSC with LookUp_SC has a constant access time of about 17 cycles. This is a 2.6 speedup compared with the MDSC and a 3.3 when compared with the IBMSC.

5.2 GLCM Results

In Fig. 4(a) the average execution time for the Full High Definition (FHD) inputs are depicted. The DMA time to fetch the picture data are not taken into consideration as they can be overlapped with the computation. The *No Cache* baseline is not presented because 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. The *Processing* time also depicts the execution time for the GLCM if the matrix would fit in the LS. Each software cache has a suffix SXY, where S corresponds to the number of sets, X to the number of rows in the 2-dimensional cache block (which is always 0 for the 1-dimensional IBMSC), and Y to the number of columns, all in base-2 logarithm. Furthermore, each SC is 4-way set associative.

For each software cache we determined the optimal configuration but the cache size is fixed at 64KB. The optimal configuration of the IBMSC has 128 sets and a line size of 128 bytes. Surprisingly, the optimal configuration of the MDSC uses one-dimensional blocks. Furthermore, it uses 64 sets and a block size



(a) GLCM (b) MC

Fig. 4. Average runtime for FHD sequences for No_Cache, IBMSC, MDSC, and AMDSC.

of 256 bytes. Compared with the IBMSC, the MDSC has higher DMA transfer and miss handling times. This is caused by the multidimensional handling of each MDSC block, even if the block is 1-dimensional, as in this particular case (a DMA list transfer is slightly slower than a conventional DMA transfer). On the other hand, the MDSC has a smaller access time that compensates for the other time components. Also, the MDSC has a higher hit rate. For a more thorough analysis of the GLCM results with MDSC please refer to [8].

The SC access time dominates the execution time for the IBMSC and MDSC with 70% and 54%, respectively. The reason is that GLCM has a very small amount of computation per access. The AMDSC speeds up the access time by a factor of 3, compared with MDSC. The acceleration in access time translates to a total speedup of 2.1 and 1.6 for the total kernel, when compared with IBMSC and with the regular MDSC, respectively.

In this particular kernel, the improvement comes from two factors. First, there is the reduction in the number of instructions. The LookUp_SC instruction replaces several instructions, what leads to performance gain. A second factor is that it also reduces the size of the cache access function. With the reduced size, the access function can be inlined in the GLCM calculation function that further increases the speedup. With the inlined MDSC (IMDSC) access function, the speedups of the LookUp_SC instruction for the access and the whole kernel are 2.8 and 1.5, respectively.

5.3 MC Results

The IBMSC has a hit rate of 94%, while the MDSC has a hit rate of 98%. This explains the lower *DMA* time for the MDSC. On the other hand, the MDSC complex *Miss Handling* leads to a higher miss handling time than the IBMSC, even with fewer misses. The Accelerated MDSC results for MC are depicted in Fig. 4(b). The configuration notation is the same as in Section 5.2.

The LookUp_SC instruction yields a performance improvement of 47% for the AMDSC access time when compared to the MDSC access time. This relative improvement is lower than in the previous experiments. The reason for this is that the MC processing function is an accelerated code that operates on a temporary array. The cache access copies the reference area to this temporary array. This is performed inside a nested loop, which introduces a significant overhead. On the whole MC kernel processing time the AMDSC performance improvements are 108%, 28%, and 16% when compared with the *No Cache*, IBMSC, and MDSC implementations, respectively.

6 Conclusions

In this paper, we presented a new instruction to accelerate software caches. Software caches are an efficient way to improve performance of applications that do not have a predictable access behavior to make efficient use of DMAs but do exhibit locality. On the other hand, software caches incur high overhead.

The LookUp_SC instruction performs several critical computations and is composed by a number of hardware operations. Because several operations can be performed in parallel, the overall latency of the instruction is estimated to be the same as the latency of an indexed load instruction.

As a side effect of the reduction of the instructions in the AMDSC access functions, it can be inlined without a severe impact on the code size. This particularly improves the performance of applications with a high ratio of number of memory access to computation.

Our experimental results showed that the LookUp_SC instruction significantly accelerates the MDSC access function. The overall acceleration depends on the application that is being accelerated. For MC, the access function acceleration is 47%, while for GLCM access function a speedup of 3 is achieved, when compared to the MDSC access function. For the whole kernels the acceleration is 1.28× and 2.1×, for MC and GLCM, respectively.

As future work we plan to adapt our instruction to conventional software caches. As regular software caches use the memory position as index, the address calculation would not be accelerated by our proposed instruction. Because of this we expect a lower improvement than the one achieved in this paper.

Another future work is the evaluation of the power consumption of the proposed instruction. A power consumption comparison between a hardware cache and our AMDSC with scratchpad memory would give interesting quantitative results and could be used for processor designers as another evaluation point for future power efficient processors.

References

1. Banakar, R., Steinke, S., sik Lee, B., Balakrishnan, M., Marwedel, P.: Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. In: Tenth Int. Symp. on Hardware/Software Codesign (CODES), Estes Park, ACM (2002) 73–78
2. Kahle, J., Day, M., Hofstee, H., Johns, C., Maeurer, T., Shippy, D.: Introduction to the Cell Multiprocessor. IBM Journal of Research and Development **49**(4) (2005) 589–604
3. Edler, J., Hill, M.D.: Example Library API Reference https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/3B6ED257EE6235D900257353006E0F6A/$file/SDK_Example_Library_API_v3.0.pdf.
4. Balart, J., Gonzalez, M., Martorell, X., Ayguade, E., Sura, Z., Chen, T., Zhang, T., Obrien, K., Obrien, K.: A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor. In: Languages and Compilers for Parallel Computing: 20th Int. Workshop, Lcpc 2007, Urbana, Il, USA, October 11-13, 2007, Revised Selected Papers, Springer-Verlag New York Inc (2007) 125–140
5. Lee, J., Seo, S., Kim, C., Kim, J., Chun, P., Sura, Z., Kim, J., Han, S.: COMIC: A Coherent Shared Memory Interface for Cell BE. In: PACT '08: Proc. of the 17th Int. Conf. on Parallel Architectures and Compilation Techniques, New York, NY, USA, ACM (2008) 303–314
6. Chen, T., Zhang, T., Sura, Z., Tallada, M.G.: Prefetching Irregular References for Software Cache on Cell. In: CGO '08: Proc. of the Sixth Annual IEEE/ACM Int. Symp. on Code Generation and Optimization, New York, NY, USA, ACM (2008) 155–164
7. Gonzàlez, M., Vujic, N., Martorell, X., Ayguadé, E., Eichenberger, A.E., Chen, T., Sura, Z., Zhang, T., O'Brien, K., O'Brien, K.: Hybrid Access-Specific Software Cache Techniques for the Cell BE Architecture. In: PACT '08: Proc. of the 17th Int. Conf. on Parallel Architectures and Compilation Techniques, New York, NY, USA, ACM (2008) 292–302
8. Azevedo, A., Juurlink, B.: A Multidimensional Software Cache for Scratchpad-Based Systems. Int. Journal of Embedded and Real-Time Communication Systems **1**(4) (2010) 1–20
9. Gschwind, M., Hofstee, H., Flachs, B., Hopkins, M., Watanabe, Y., Yamazaki, T.: Synergistic Processing in Cell's Multicore Architecture. IEEE Micro **26**(2) (2006) 10–24
10. Azevedo, A., Juurlink, B.: An Efficient Software Cache for H.264 Motion Compensation. In: Proc. of IEEE Int. Symp. on System-on-Chip. (October 2009) 147–150
11. Cabarcas, F., Rico, A., Rodenas, D., Martorell, X., Ramirez, A., Ayguade, E.: CellSim: A Cell Processor Simulation Infrastructure. HiPEAC ACACES-2007 279–282
12. : IBM Full-System Simulator for the Cell Broadband Engine Processor. http://www.alphaworks.ibm.com/tech/cellsystemsim.
13. Shahbahrami, A., Juurlink, B.: Optimization of Content-Based Image Retrieval Functions. In: 10th IEEE Int. Symp. on Multimedia. (December 2008) 607–612
14. Alvarez, M., Salami, E., Ramirez, A., Valero, M.: HD-VideoBench: A Benchmark for Evaluating High Definition Digital Video Applications. In: Proc. IEEE Int. Symp. on Workload Characterization. (2007) http://personals.ac.upc.edu/alvarez/hdvideobench/index.html.