PERFORMANCE AND ENERGY OPTIMIZATION OF MULTIMEDIA APPLICATIONS USING DMA COMBINED WITH PREFETCH

M. Dasygenis[1]*, E. Brockmeyer*[2]*, D. Soudris*[1]*,*
F. Catthoor[2,3]*, A. Thanailakis*[1]*, and G. Papakostas*[4]

[1]VLSI Design and Testing Center,
[4]Automatic Control Systems Laboratory,
Dept. of Electrical and Computer Eng.
Democritus University of Thrace, 67 100 Xanthi, Greece
{mdasyg,dsoudris,thanail,gpapak}@ee.duth.gr

[2]Desics, IMEC, Kapeldreef 75, Leuven, Belgium
[3]Also professor at the Katholieke Univ. Leuven, Belgium
{Francky.Catthoor, Erik.Brockmeyer}@imec.be

ABSTRACT

Memory access latency is becoming an increasing performance problem in modern embedded systems. While throughput can be addressed by simply widening data paths and by using several memory banks (at the cost of area and energy), no such simple solution strategy is available for reducing latency. To counter this bottleneck, numerous techniques, methodologies or performance enhancements, involving hardware and/or software means, have been proposed to hide memory access latency. This paper presents a novel systematic approach to hide this latency using the Direct Memory Access (DMA) mode, which is present in all modern memories, combined with a software prefetch mechanism. It is shown that in applications that make use of large block transfers, the off-chip memory accesses can be hidden from the processor resulting in efficient designs in terms of performance (while still reducing energy for the same task execution). Experimental results on six well known multimedia and imaging applications that were measured using the TI C6201 Device Simulator, illustrate that the memory latency can be masked efficiently, improving the performance by more than 45%.

1. INTRODUCTION

The design of embedded or integrated systems has become more and more complex, especially due to their particular characteristics and specific usage (for instance mobile computing), which require stringent energy and area constraints. On the other hand, in data dominated applications, like interactive multi-media, data storage and transfers are the most important factors in terms of meeting the real time constraints that are imposed. The main bottleneck of these applications is the huge amount of transfers and storage requirements from and to on-chip and off-chip memory that results in extreme power consumption and performance degradation.

This bottleneck has become more crucial, as the gap between processor and memory speeds continues to grow. The large bandwidth that is required to fulfill the time critical and memory intensive applications, poses a significant barrier in the realization of complex real time embedded systems. An obvious solution could be a system consisting of many multiport memories. However, such a memory organization increases prohibitively the total energy consumption, making it a non practical solution. Multi-port memories cause a large cost in area and power and high-speed memories are restricted to very small sizes. However, they may not be avoidable in stringent timing constraints. Hence, techniques that on one hand the real time constraints are met and one the other hand efficient embedded realizations of the memory architecture are achieved are required.

To counter this bottleneck, numerous techniques methodologies or performance enhancements, involving hardware and/or software means, have been proposed to hide memory access latency. One of the ways of overcoming this increasing disparity between processor and memory access speed is using prefetching techniques. While prefetching does not reduce the latency of memory access speed, it hides this latency by overlapping memory accesses and instruction execution. In the literature, a number of hardware-based data prefetching techniques exist for some time, many of which customize hardware for specific memory reference patterns. Chen and Baer have proposed stride prefetchers [1] that correlate non-unit data address striders with a memory instruction PC in a small table and prefetch based on stride. Jouppi has proposed stream buffers [2] to detect and prefetch the stride, while Joseph and Grunwald have proposed Markov prefetchers [3] associating multiple subsequent addresses with each correlation. A novel research work was presented by Lai et al [4], in which they have introduced a hardware prefetcher which performs dead block prediction and dead block correlation, in order to accurate, precise and timely prefetch the required blocks. Finally, a recent hardware scheme disclosed by Hu et al [5], has introduced new useful metrics regarding generational behavior in cache lines, and presented hardware structures that exploit these metrics to improve performance. Solely hardware prefetching can boost the performance of an applications, but it is only applicable to specific types of applications that exhibit good spatial locality with few algorithmic dependencies. Applications that consists of loops with small production/consumption windows like multimedia applications, cannot benefit from the hardware prefetching, because the large number of dependencies is a key hurdle, that can only be circumvented by algorithmic transformations.

Except the hardware prefetching techniques, authors have suggested the use of software prefetching. One of the first research works that addressed the prefetching techniques using compiler-based techniques was from Lipasty *et al* [6] and Mowry *et al* [7] who developed heuristics that consider pointers passed as arguments on procedure calls and inserted prefetchers at the call sites for the data references by the pointers. Horowitz *et al* [8] have approached this problem by proposing a new class of memory operations called informing memory operations, which essentially consist of a memory operation combined with a conditional branch-and-link operation that could be used to tailor software-controlled prefetching. Cooksey *et al* [9] on the other hand, proposed a hardware only data prefetching architecture that exploits the memory allocation used by operating systems to improve the performance of pointer intensive applications. A mixed hardware/software based solution was also proposed by Gschwind *et al* [10], in which they extended *(i)* the memory subsystem by intergrading a prefetch buffer mechanism, and *(ii)* the MIPS R3000 instruction set to include instructions to initiate the prefetching in an given application. A thorough cost/performance analysis of data prefetching was presented by Metcalf [11], who classified the different kinds of prefetching modeled and compared them. Finally, Grun *et al* [12] addressed the dominant bottleneck of memory access speed, by proposing recently the APEX framework, an approach that extracts, analyzes and clusters the most active access patterns in the applications, and aggressively customizes the memory architecture to match the needs of the application, using memory models. Again software prefetching alone, cannot yield significant performance improvement, due to its unavailability to handle data dependencies. As the experimental results show in Section 6, an efficient way of confronting the memory latency, is an approach of software prefetching, which is pipelined and performed using DMA

In this work the problem of hiding the memory latency is addressed by employing a prefetch mechanism combined with the DMA mode that most current embedded systems have. To the best of our knowledge until know, nobody has systematically considered the beneficial opportunities of prefetching combined with DMA. Designers are using this today manually in an ad hoc way in the critical parts of their code, but this is very tedious and error-prone. We attempt to bridge this gap, and illustrate that performance (and power) can be improved by the combination of prefetching and DMA. Five motion estimation (ME) kernels, belonging to the multimedia domain, and one image processing application, are used as test vehicles. Multimedia and imaging applications have a uniform and known a priori memory access pattern. In this paper we remedy the identified problem with only a small penalty in terms of design time in making the prefetch copies and initiating explicitly the DMA transfer. As the experimental measurements show, prefetching combined with DMA gives a boost in performance (e.g. ~45% performance improvement on the Quadtree Structured Difference Pulse Code Modulation (QSDPCM) [13]) because all the off-chip memory accesses are successfully hidden from the CPU. Also the related energy waste is reduced.

The rest of the paper is structured as follows: Section 2 introduces the target memory architecture we have considered while Section 3 briefly describes the DMA access mode. Then, Section 4 presents the prefetching technique, followed by Section 5 which analyzes how DMA is combined with prefetching. The subsequent Section discusses the demonstrator applications and experimental results, concluding with Section 7.

2. TARGET MEMORY ARCHITECTURE

The memory architecture that is used in this research work is depicted in Figure 1, which is a simplified block diagram of the TMS230C6000 CPU architecture [14]. This architecture consists of a ‘C6000 CPU core with two independent 32-bit data paths for accessing on-chip or off-chip memory, two blocks of on-chip memory, a program/data bus, a peripheral off chip DMA (Direct Memory Access) controller and an external memory. The amount and location of internal memory depends on the particular device selected. In our case (6201 CPU core), exist two 32Kbytes on-chip memory blocks. On the data bus, a data memory controller is introduced, which services requests to internal memory by either the CPU or the DMA. Off-chip and on-chip memory are used in a continuous memory map, which means that some memory addresses are located in the on-chip space, some are reserved, and some are located off-chip.

3. DMA OVERVIEW

In most contemporary embedded systems, memory transfers from (to) off chip memory can be performed in many ways. The easiest and most common (but very inefficient) way is when the CPU controls and operates the transfer. When the transfer is CPU controlled the processor stalls because it does not perform any computation at all, but waits for the memory transfers, which require multiple cycles, to be completed. An alternative way to perform the transfer is the Direct Memory Access (DMA) mode. If the transfer occurs in this way, the CPU is not responsible for operating the memory transfer, but that task is deferred to another controller, which is called a DMA controller. Utilizing DMA transfers means that the CPU can be used in performing other tasks. In order to use this mode, the processor has to specifically instruct the DMA to start copying from (or to) off chip memory, to (or from) on chip memory. This is accomplished by a group of assembly commands that are specific to every embedded architecture. During a DMA transfer, the CPU can request data from the on-chip memory which have not been fetched yet. In order to avoid this problem, a register can be set when the DMA transfer has finished, which will be checked by the CPU before it access the on-chip data. If the DMA transfer has not finished, the CPU will have to wait (denoted as DMA_WAIT) until the flag is set, otherwise the on-chip data that the processor requires, may be invalid. It is evident that the time spent in the DMA_WAIT has to be limited. Usually DMA_WAIT statements are placed as late as possible, but not more lately than the place that the specific on chip data is required by the processor. Ideally the transfer would have been completed by the time the processor reaches the DMA_WAIT and thus no cycles will be spent stalling. However putting the DMA transfer later, requires more on-chip memory space. This side effect arises from the increased life time of on-chip memory data, making inplace mapping techniques for memory compaction less effective [15]. In realistic cases a complete removal of the stalls is not always possible, which means that some cycles are spent in DMA_WAIT statements. Alternative, another thread could be initiated. Of course, interleaving of processing threads could result in much overhead.

Usually in current embedded systems, more than one DMA channel is available to the developer. The DMA channel with the lowest number (i.e. DMA0), is the channel with the highest priority and vice versa. DMA transfers cannot be performed in parallel. For this reason, if multiple DMA transfers are initiated, the DMA transfer of the highest priority will be completed first, and then the DMA of the next priority will continue. Finally the DMA transfer should be as large as possible, because starting a DMA imposes an overhead to the system.

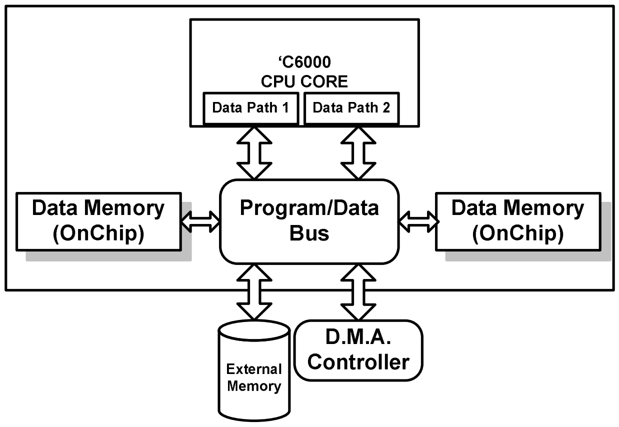


Figure 1. The Memory Architecture of the TI C6201

The combination of the previous facts, enables the designer to initiate a very large DMA with low priority, which will have to be completed in a very large time window. When small off-chip accesses are required, these can happen with the highest priority, suspending temporarily the previous ongoing DMA. After the DMA with the highest priority is concluded, the first DMA will resume (without the extra overhead of re-initiating the DMA transfer).

4. PREFETCHING

The memory subsystem is a key bottleneck in embedded applications. To reduce memory stall time, many current processors support, software-controlled prefetching. With this technique, the compiler or programmer schedules an explicit prefetch instruction for a memory location that will be accessed by the processor at a later time, with the goal of bringing the off-chip data elements into the on-chip internal memory, before the CPU issues a demand memory access [16].

A large portion of many applications, and especially applications that make use of large multidimensional arrays, are heavily dominated by read and write accesses to arrays. These arrays are too big to fit in on-chip memories, and for this reason they are placed in large off-chip memories. These operations cannot be adequately speeded up by caches, as they exhibit poor locality and a large working sets, incurring many capacity misses. However, accesses to this type of data are very predictably. After accessing an element, the probability is high that the next element will be accessed. Multimedia applications, like motion estimation kernels, exhibit a high predictability when the current and previous frames are accessed. For this reason prefetching is especially useful in this kind of applications. This prefetching occurs concurrently with the execution of the actual program. Thus, the penalty incurred by accessing off chip memory elements is hidden under normal program execution. This translates into a high effectiveness of the prefetching. But the latency is still a problem in many applications that require strict timing constraints.

5. PREFETCHING COMBINED WITH DMA

To alleviate the bottleneck of the memory latency we have combined the prefetching technique with the DMA mode that modern memories provide. Two key points are to be decided in this context. What are the prefetching candidates, and where to place the DMA wait statements!

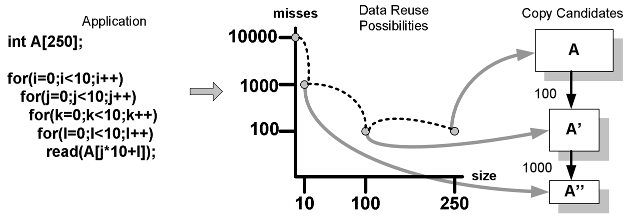
5.1 PREFETCHING CANDIDATES

In order to specify the prefetch candidates the prototype tool MHLA [17] is used. This tool takes advantage of temporal locality and limited lifetime of the memory architecture constraints while taking into account the copy overhead, and explores all the possible different layer assignments. One of the outputs of this tool is a set of copy candidates (CC, i.e. prefetch copies) for a specific on-chip memory size. The tool has performed a design space exploration on the six applications, and the prefetch copies are available. Any selected prefetch copies can be implemented in the code in such a way that the CPU controls the transfer (using for-loop or the ‘memcopy’ function).

Generally speaking, a relation exists between the size of a CC and the number of transfers from the higher layer, typically called misses (see Fig. 2). This figure shows a simple loop nest with one reference to an array *A* with size 250. The array has 10000 accesses. Several CCs for array *A* are possible. For example we could add a copy *A*" of size 10 which is made in front of the k-loop by adding the statement “for (*z*=0;*z*<10;*z*++) *A*”[*z*]=*A*[*j**10+*z*];". This statement is executed 100 times, resulting in 1000 misses to the A array. This CC point is shown in Fig 2b. Note that the good spatial locality in this example does not influence the amount of misses to the next level. In theory any CC size ranging from one element to the full array size is a potential candidate. However, in practice only a limited set of CCs leads to efficient solutions. Obviously, a larger CC can retain the data longer and can therefore avoid more misses. All possible CCs for this very simple case are shown in Fig. 2 b.

5.2 USING DMA TRANSFERS

The second key point is to replace the memcopy commands with the DMA commands. A DMA transfer consists of a DMA_START and a DMA_WAIT command. The DMA_START command instructs the DMA controller to start the DMA transfer for a given block size, from a specific off-chip (or on-chip) memory address to a specific on-chip (or off-chip) memory address. The DMA_WAIT command is a crucial point in the DMA implementation. In the DMA_WAIT command, the processor stalls, waiting for the DMA transfer to be completed. When the DMA transfer is completed, then the processor can resume the execution of the application. The DMA_WAIT statement usually is placed after a computational intensive nested loop. Thus while the CPU is performing the computations, the DMA performs the memory transfer. In case a dependency between elements exists that are involved in the DMA transfer, loop transformations to break the dependencies are employed, like loop pipelining or loop unrolling [15].



```
int A[250];

for(i=0;i<10;i++)
  for(j=0;j<10;j++)
    for(k=0;k<10;k++)
      for(l=0;l<10;l++)
        read(A[j*10+l]);
```

Figure 2. Copy Candidates Example

The core of the proposed approach is to hide the prefetch techniques from the CPU using DMA. This means that the DMA transfer must take place and happen in parallel with a CPU computational task. Thus, when CPU finishes the computation and continues the execution of the program, the off-chip data that will be required, will have been transferred to on-chip and will be available. This results in hiding the off-chip memory access latency. Figure 3 and 4 illustrate two effective transformations, using DMA priorities and DMA pipelining, respectively.

5.2.1 DMA PRIORITIES

Figure 3 illustrates the approach when multiple DMA priorities are used. In Figure 3 (i) the block_copy commands transfer arrays from the off-chip to the on-chip space for prefetching, which is done with the CPU. The next step is to transform these block_copy commands to DMA transfer commands. For this reason in Figure 3(ii) the DMA0_START and DMA0_WAIT commands are added, denoting that the DMA priority 0 has to be initiated. The second DMA transfer has no dependencies from the previous loop. For this reason, this DMA transfer (from *off-chipA* to *on-chipB*) can be initiated in advance, before the computation of the previous loop. However, in the previous loop another DMA transfer exists which prohibits us to use the same priority. For this reason the second DMA transfer is initiated using a lower priority (denoted as DMA1_START). Figure 3 (iii) shows the final version of the algorithm, in which the transfer *off-chipA* to *on-chipB* is initiated in advance with priority 1. Thus, when there is the need to transfer from *off-chipC* to *on-chipD*, the DMA1 is suspended, because a DMA transfer with a higher priority is initiated. When DMA0 is finished, the DMA1 can resume its operation. In this way the DMA transfer from *off-chipA* to *on-chipB* has been masked from the CPU. The efficiency of this masking is reflected on the CPU stalled cycles on the DMA1_wait statement. If there are few cycles (or none) spent waiting in this wait statement, then the masking is successful. With other words, if the DMA1 transfer requires a number of cycles similar to the number of cycles that are required by the CPU to perform the computation of the nested loop, then the masking is efficient. Selecting the loop that will mask the DMA transfer is a very crucial point, and for this reason further investigation to automate this is needed. Finally, the DMA0 wait statement cannot be masked due to existing dependencies.

5.2.2 DMA PIPELINE

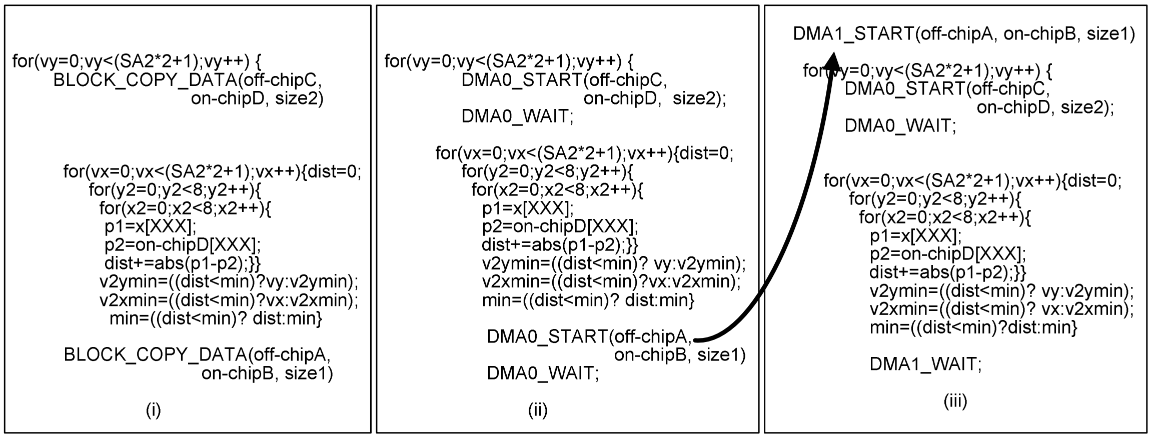
Figure 4 illustrates another approach that is followed in hiding the DMAs using pipelining. In Figure 4 (i) a DMA transfer goes from an off-chip memory (*off-chipA*) to an on-chip memory (*on-chipB* of *'size'* bytes). In the subsequent loop, a dependency prohibits the masking of this DMA transfer by moving the DMA_WAIT statement after the loop. In order to break this dependency, loop pipelining is employed, and *on-chipB* is doubled in size (Figure 4 (ii)). In the same figure also it is illustrated that the first iteration of the DMA transfer is unrolled in order to create the DMA pipeline. This results in the situation that when the CPU computes the motion vectors of the current iteration, the DMA transfers elements of the next iteration. The dependency is now broken, and in Figure 4 (iii), the DMA_WAIT statement has been moved to the end of this loop nest. Again this DMA0_wait statement has to be evaluated and measured in terms of CPU cycles spent stalling there. Finally, the first DMA0_wait statement, which is the pre-pipeline context, cannot be masked due to dependencies.

In four of the six applications that we have analyzed, namely Full Search (FS), Hierarchical Search (HS), Parallel Hierarchical One Dimensional Search (PHOD), and 3 Step Logarithmic Search (3SLOG) [18] only one prefetch candidate is present and one computational loop, and thus the DMA_WAIT statement is placed after the nested loop. In the QSDPCM and CAVITY applications multiple prefetch candidates and multiple loops exist. For this reason, more than one DMA channel with different priorities is used. Again the goal is to mask the DMA transfer, for this reason the DMA_WAIT statement is placed after a subsequent nested loop. A point that has to be stressed is that the memory hierarchy consists of one on-chip memory and one off-chip memory, both addressed by a common memory map. For this reason, the arrays are specifically placed in memory addresses establishing this way that the prefetch copies are on-chip and the current/previous frames are off-chip.

6. DEMONSTRATORS AND EXPERIMENTAL RESULTS

Our demonstrator applications are selected to be several well known real-life motion estimation and imaging algorithms: *(i)* Quadtree Structured Difference Pulse Code Modulation (QSDPCM), (*ii*) Full Search (FS), *(iii)* Hierarchical Search, (*iv)* Parallel Hierarchical One Dimensional Search, (*v)* 3-Step Logarithmic Search (3SLOG), (*vi*) Cavity Detection. To evaluate the performance benefit derived from the prefetch mechanism combined with DMA, we measure the execution time of the real life multimedia kernels executed on the embedded development suite, TI Code Composer Simulator [19]. All the measurements are performed using typical parameters of these applications. The experiments are carried out with the luminance component of QCIF frame (144x176) format, while blocks of 16x16 pixels are selected. All these algorithms calculate the motion vectors of two successive frames, but they differ in granularity, the precision and the complexity. Specifically, the QSDPCM algorithm is an interframe compression technique for video images. It involves a motion estimation step, and a quadtree based encoding of the motion compensated frame-to-frame difference signal. Finally, The Cavity detection algorithm is an image processing application used mainly in the medical field for detecting tumor cavities in computer tomography pictures and is based on edge detection.

Most video coding concepts contain a motion compensated temporal DPCM (differential pulse coding modulation) module in which the motion compensated prediction (MCP) error image is transform coded, employing the widely introduced discrete cosine transform (DCT). In the QSDPCM method, the MCP error image is adaptively decomposed in blocks of variable size, where in each block the MCP error image is represented by the local sample means. The variable block size structure is described by a quadtree. The local sample means are independently Huffman coded. QSDPCM has the highest computational complexity of all the other three algorithms, but it yields the best signal-to-noise (SNR) ratio. FS algorithm has a high computational complexity also and is a very expensive (in terms of operations per frame), but guarantees finding the optimal motion vectors due to the thoroughly frame motion scan.

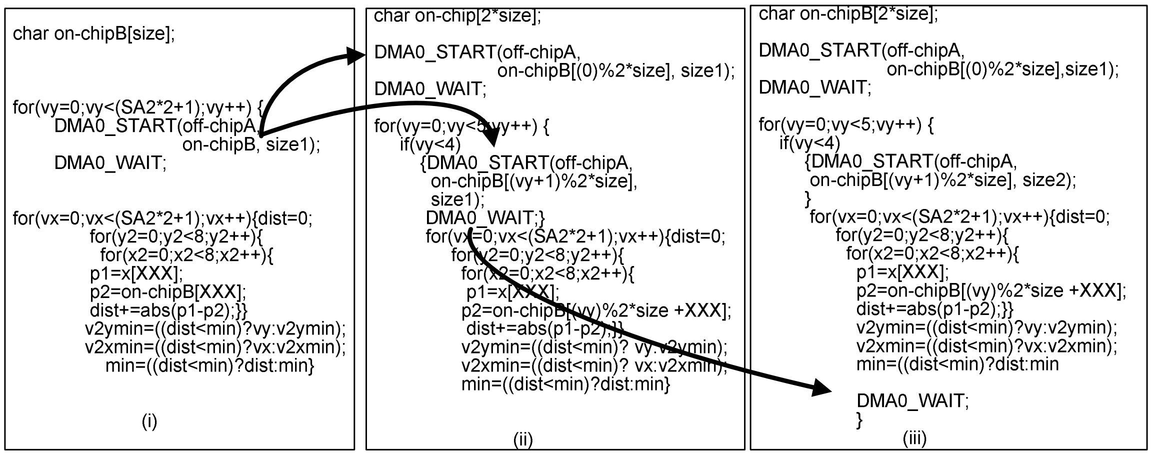


```
for(vy=0;vy<(SA2*2+1);vy++) {
        BLOCK_COPY_DATA(off-chipC,
                           on-chipD, size2)

          for(vx=0;vx<(SA2*2+1);vx++){dist=0;
               for(y2=0;y2<8;y2++){
                 for(x2=0;x2<8;x2++){
                  p1=x[XXX];
                  p2=on-chipD[XXX];
                  dist+=abs(p1-p2);}}
                 v2ymin=((dist<min)?vy:v2ymin);
                 v2xmin=((dist<min)?vx:v2xmin);
                   min=((dist<min)? dist:min}

          BLOCK_COPY_DATA(off-chipA,
                              on-chipB, size1)

                         (i)
```

```
for(vy=0;vy<(SA2*2+1);vy++) {
              DMA0_START(off-chipC,
                              on-chipD,  size2);
              DMA0_WAIT;

         for(vx=0;vx<(SA2*2+1);vx++){dist=0;
              for(y2=0;y2<8;y2++){
                for(x2=0;x2<8;x2++){
                  p1=x[XXX];
                  p2=on-chipD[XXX];
                  dist+=abs(p1-p2);}}
                  v2ymin=((dist<min)? vy:v2ymin);
                  v2xmin=((dist<min)?vx:v2xmin);
                  min=((dist<min)? dist:min}

                   DMA0_START(off-chipA,
                                   on-chipB, size1)
                   DMA0_WAIT;

                         (ii)
```

```
DMA1_START(off-chipA, on-chipB, size1)

  for(vy=0;vy<(SA2*2+1);vy++) {
        DMA0_START(off-chipC,
                        on-chipD, size2);
        DMA0_WAIT;

     for(vx=0;vx<(SA2*2+1);vx++){dist=0;
          for(y2=0;y2<8;y2++){
            for(x2=0;x2<8;x2++){
              p1=x[XXX];
              p2=on-chipD[XXX];
              dist+=abs(p1-p2);}}
              v2ymin=((dist<min)? vy:v2ymin);
              v2xmin=((dist<min)? vx:v2xmin);
              min=((dist<min)?dist:min}

              DMA1_WAIT;

                    (iii)
```

Figure 3: Hiding DMAs using priorities



```
char on-chipB[size];


for(vy=0;vy<(SA2*2+1);vy++) {
       DMA0_START(off-chipA,
                       on-chipB, size1);
       DMA0_WAIT;

for(vx=0;vx<(SA2*2+1);vx++){dist=0;
            for(y2=0;y2<8;y2++){
              for(x2=0;x2<8;x2++){
            p1=x[XXX];
            p2=on-chipB[XXX];
            dist+=abs(p1-p2);}}
           v2ymin=((dist<min)?vy:v2ymin);
           v2xmin=((dist<min)?vx:v2xmin);
              min=((dist<min)?dist:min}

                    (i)
```

```
char on-chip[2*size];

DMA0_START(off-chipA,
                 on-chipB[(0)%2*size], size1);
DMA0_WAIT;

for(vy=0;vy<5;vy++) {
    if(vy<4)
        {DMA0_START(off-chipA,
         on-chipB[(vy+1)%2*size],
         size1);
        DMA0_WAIT;}
        for(vx=0;vx<(SA2*2+1);vx++){dist=0;
            for(y2=0;y2<8;y2++){
              for(x2=0;x2<8;x2++){
               p1=x[XXX];
              p2=on-chipB[(vy)%2*size +XXX];
               dist+=abs(p1-p2);}}
              v2ymin=((dist<min)? vy:v2ymin);
              v2xmin=((dist<min)? vx:v2xmin);
              min=((dist<min)?dist:min}

                    (ii)
```

```
char on-chipB[2*size];

DMA0_START(off-chipA,
                  on-chipB[(0)%2*size],size1);
DMA0_WAIT;

for(vy=0;vy<5;vy++) {
    if(vy<4)
        {DMA0_START(off-chipA,
         on-chipB[(vy+1)%2*size], size2);
        }
         for(vx=0;vx<(SA2*2+1);vx++){dist=0;
             for(y2=0;y2<8;y2++){
               for(x2=0;x2<8;x2++){
                 p1=x[XXX];
                 p2=on-chipB[(vy)%2*size +XXX];
                 dist+=abs(p1-p2);}}
                 v2ymin=((dist<min)?vy:v2ymin);
                 v2xmin=((dist<min)?vx:v2xmin);
                 min=((dist<min)?dist:min

                 DMA0_WAIT;
                 }
                        (iii)
```

Figure 4: Hiding DMAs using pipelining

HS is a fast ME scheme that use a combination of search strategies that use both fewer search locations and fewer pixels in computing the motion vectors, while PHODS and 3SLOG belong to the class of very fast algorithms that reduce the motion–estimation complexity by reducing the number of search locations that are used in determining the motion vectors. Cavity on the other hand, consists of 10 loop nests and its complexity is a little higher than FS.

The applications are measured in the Code Composer Simulator, using copy candidates where applicable, but without using DMA for the memory transfers (denoted as original). Then they are measured using DMA overlaid with CPU computations (denoted as prefetch+dma). Figure 5, presents the performance measurements of the six real-life applications. The y-axis is in logarithmic scale, in order to have all the applications depicted on the same plot. The y-axis, presents the cycles spent performing the computations, excluding the I/O functions. The lower the value, the better performance the application has, because it is executed in fewer cycles. Figure 6 illustrates in a different way the performance gain of every application. The values depicted in this graph show the percentage of the performance improvement, compared with the original (1- *performance_optimized /performance_original*). QSDPCM application has the highest performance improvement (47%) because it is an application that has a large number of computational intensive loops. On the other hand, the motion estimation kernels exhibit smaller performance gains since they have less computational complexity than QSDPCM. Specifically the performance gain of FS, HS, 3SLOG and PHODS is 10%, 12%, 8% and 23% respectively, while CAVITY has a performance gain of 33%.

A measure of the efficiency of the DMA mode is the cycles of the CPU that are spent in stalling during the DMA_WAIT statement. An efficient DMA implementation depends on the minimization of the cycles spent in the DMA_WAIT. Figure 7 presents the cycles spend in the ME applications. It can be easily seen that the cycles spent are indeed minimal; for example QSDPCM application has only 14954 cycles in CPU stalling during the DMA_WAIT, which is a very small percentage of the total cycles in the optimized application (< 1%). Figure 8 illustrates different Prefetch+DMA possibilities for the QSDPCM application. In order to achieve optimum efficiency, the prefetch candidates have to be carefully selected together with their on-chip memory size and the proper scheduling of the DMA transfer. Different choices, lead to different performance. Thus, the designer can study the global trade-off curve, and decide the implementation that best fits his constraints.

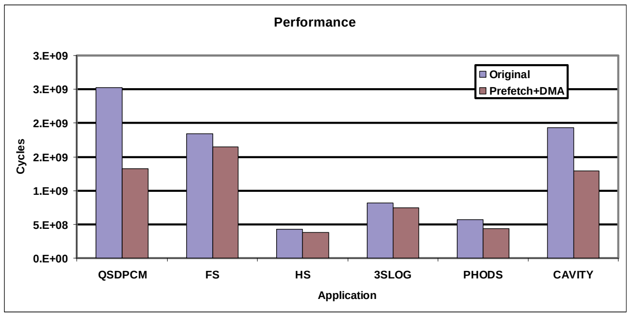


Figure 5. Performance evaluation of original and optimized ME applications (y-axis: logarithmic scale)

.

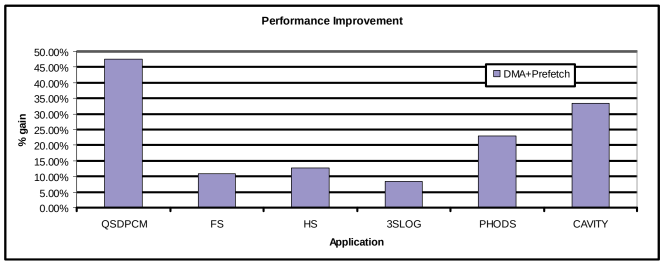


Figure 6. Performance improvement of the optimized ME applications

The performance improvements also have a direct positive impact on the energy consumption for a given task (The instantaneous power is probably the same). TI has published an application report [20] that analyzes the power consumption on the TI devices and peripherals. In this report it is shown that the C6201 core consumes ~49% of total power consumption of the system for high or low activity models, while peripherals (like the DMA controller) consumes 1% of the total power consumption and hence have nearly no overhead when activated. This means that decreasing the execution cycles on the core of a given application utilizing DMA mode has a strong direct impact on the overall power consumption. It is known that fewer cycles spent on an application execution, for the same energy cost/cycle, translates to fewer mW dissipated by the system. We are not able to provide accurate numbers of the power reduction at this point due to the lack of information on the TI core breakdown), but it can be safely assumed that a performance increase of 45% has an energy decrease of nearly the same value to implement the same task.

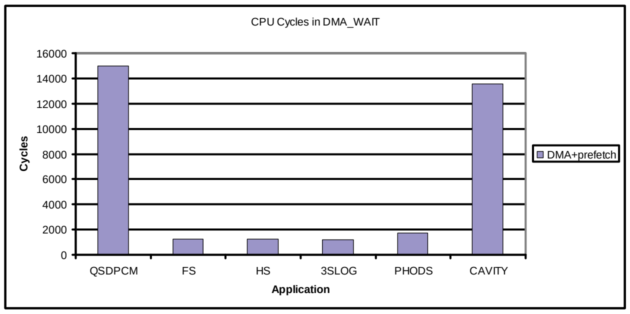


Figure 7. CPU stall cycles in DMA_WAIT statement

In the case that hardware prefetchers as [11] or [12] are used in the specific applications, it is expected that the efficiency will be reduced, because the hardware schemes perform only prefetching and not any algorithmic modifications. These applications have a lot of dependencies which they can only be broken using specific algorithmic transformations (i.e. loop pipeline), diluting the effectiveness of pure hardware prefetching. Without performing e.g. software pipelining, the elaborated hardware like DMA controllers or data movers or hardware prefetchers would have abysmal efficiency. The key contribution of this paper is that software prefetching which is pipelined and performed using DMA exhibits a significant performance and power improvement.

7. CONCLUSIONS

The memory hierarchy plays an important role and should be used efficiently in embedded systems. One of the key obstacles in achieving high memory performance utilization is memory access latency. As a result, various techniques have been devised to hide memory access latency. This paper illustrates that the performance (and power consumption) of multimedia and image processing applications, which are characterized by a uniform access pattern, can be significantly improved, by using the DMA mode that most contemporary systems have, combined with the prefetch mechanism. Six real life applications are used as test vehicles for evaluation. The measurements show that it is possible to mask the prefetch latency, using the DMA mode, with a cost overhead of increased usage of on-chip memory space. This results in applications that perform faster and consume less energy to execute the same task. These results motivate further research in developing an interactive design support tool that will analyze the code and present all the possible tradeoffs between performance and on chip memory size.

ACKNOWLEDGEMENTS

This work was partially sponsored by a scholarship from Public Benefit Foundation of Alexander S. Onassis and from Marie Curie Host Fellowship project HPMT-CT-2000-00031.

REFERENCES

[1] Tien-Fu and Jean-Loup Baer, "Reducing memory latency via non-blocking and prefetching caches", *Proc. ASPLOS V*, pp. 51-61, October 1992.

[2] Norman P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers", *Proc. 17th Annual International Symposium on Computer Architecture*, pp. 364-373, May 1990.

[3] Doug Joseph and Dirk Grunwald, "Prefetching using Markov Predictors", *IEEE Transactions on Computers*, 48(2): pp. 121-133, February 1999.

[4] An-Chow Lai, Cem Fide and Babak Falsafi, "Dead-Block prediction & Dead-Block Correlating Prefetchers", *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pp. 144-154, July 2001.

[5] Zhigang Hu, Stefanos Kaxiras and Margaret Martonosi, "Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior", *Proc. 29th Int. Symposium on Computer Architecture*, pp. 209- 220, May 2002.

[6] M. Lipasti, W. Schmidt, S. Kunkel, and R. Roediger, "SPAID: Software prefetching in pointer and call intensive environments", *Proc. of 28th Annual Int. Symp. on Micro Architecture*, Ann Arbor, Michigan, pp 231-236, Nov. 1995.

[7] T. Mowry, M. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching", *Proc. of 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ACM, pages 62-73, Boston, Oct. 1992.

[8] M. Horowitz, M. Martonosi, T. Mowry, M. Smith, "Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors", ISCA 1996, pp. 260-270.

[9] R. Cooksey, S. Jourdan, and D. Grunwald, "*A Stateless, Content-Directed Data Prefetching Mechanism*", ACM, San Jose, CA, USA, pp. 279-290

[10] M. Gschwind, and T. Pietcsh, "Vector Prefetching", ACM SIGARCH Computer Architect News, Volume 23, Issue 5, December 1995, pp. 1-7.

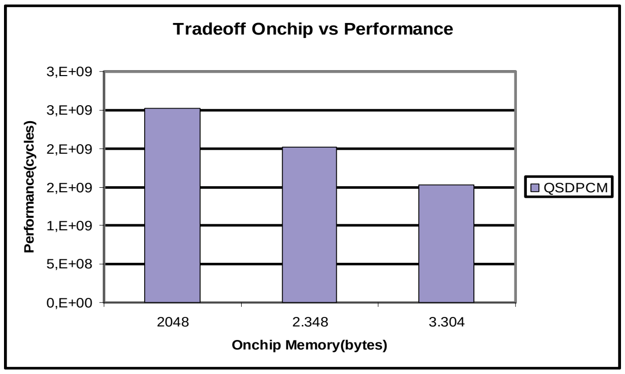


Figure 8. Tradeoff in QSDPCM application between on-chip memory size and performance

[11] C. Metcalf, "Data Prefetching: A Cost/Performance Analysis", http://www.incert.com/metcalf/papers/prefetch, 1993.

[12] P. Grun, N. Dutt, and A. Nicolau, "Access Pattern-Based Memory Connectivity Architecture Exploration", *ACM Trans. on Embedded Computing Systems*, Vol. 2, pp. 33-73, Feb. 2003.

[13] P. Strobach, "QSDPCM - A New Technique in Scene Adaptive Coding", *Proc. 4th Eur. Signal Processing Conf.*, Grenoble, France, Amsterdam, pp.1141-1144, Sep. 1988.

[14] Texas Instrument Code Composer Studio Manuals, TMS320C6000 Technical Brief, SPRU197D, Feb. 1999.

[15] F. Catthoor *et* al, "Custom Memory Management Methodology- Exploration of Memory Organization for Embedded Multimedia System Design", Kluwer Academic Publishers, Boston, 1998.

[16] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching", *Proc. of 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1991.

[17] E. Brockmeyer, M. Miranda, H. Corproraal, and F. Catthoor, "Layer assignment techniques for low energy in multi-layered memory organizations", Proc. of DATE'03, pp. 1070-1075.

[18] V. Bhaskaran and K. Konstantides, "Image and Video Compression Standards", Kluwer Academic Publishers, 1998.

[19] Texas Instrument Code Composer Studio IDE C6000 for the TMS320C6000, http://www.ti.com.

[20] Texas Instrument Code Composer Studio Manuals, TMS320C62x/C67x Power Consumption Summary, SPRA486C, July 2002.