Available online at www.sciencedirect.com

Real-Time Imaging 9 (2003) 371–386

www.elsevier.com/locate/rti

Data memory power optimization and performance exploration of embedded systems for implementing motion estimation algorithms[☆]

K. Tatas[a,*], M. Dasygenis[a], N. Kroupis[a], A. Argyriou[b], D. Soudris[a], A. Thanailakis[a]

[a] Department of Electrical and Computer Engineering, VLSI Design and Testing Center, Democritus University of Thrace, Xanthi 67100, Greece
[b] School of Electrical & Computer Engineering, 325716 Georgia Institute of Technology, Atlanta, GA, 30332-1070, USA

Abstract

A memory power optimization and performance exploration methodology based on high-level (C language) code transformations that allows the system designer to explore various data memory power, data memory area and performance trade-offs early in the design process of embedded multimedia systems is introduced. This exploration strategy is introduced for both single and multiprocessor environments. The latter requires partitioning of the application. After employing software transformations, the experimental results, obtained using four well-known motion estimation kernels provide an insight on the performance and energy consumption trade-offs, comparing memory hierarchies for the ARM programmable core and prove the validity of the proposed approach.

1. Introduction

In an age when wireless communications flourish and portable multimedia devices increase in complexity and functionality while decreasing in size and cost, designers face the conflict of improving performance while reducing power consumption and supporting rapidly evolving standards, since they constantly have to add new functionality and at the same time ensure the real-time operation of the system, with advances in process technology being their only ally. Out of this necessity to achieve the greatest possible power and performance gains, design goals have shifted to the higher design levels. In order to cope with the changing specifications and meet the tight performance and power constraints, architectures comprising one or more embedded programmable processors and dedicated hardware accelerators began to emerge. Still, even if these architectures meet the programmability and performance requirements, power consumption is the decisive factor in making a portable device feasible and commercially viable.

It has been proven that the main contributor to the power budget of such an embedded system is the data memory [1]. This power consumption is related to the number of memory fetches, which also lead to a respective number of transfers on the memory data bus. However, the power consumption of the bus is not significant [2]. Various memory exploration and optimization techniques have been proposed in the literature [1,3,4]. In [5] a data memory optimization methodology based on an on-chip memory hierarchy was introduced for programmable processors. The memory hierarchy is reflected in the high-level specification language (C) as code transformations, known as data-reuse transformations. In other words, the source code of the application is rewritten with no change in functionality in order to reflect the hardware implementation in general and the data memory hierarchy specifically. The optimization methodology is based on the fact that a large off-chip memory consumes greater power per memory transfer than a small on-chip one, therefore the methodology attempts to move the greatest number of transfers from the off-chip memory to on-chip ones, by locating data that are used often. This methodology is extended here and specially tuned for multimedia applications implemented in embedded systems. Since performance is

[☆] This work was partially supported by the project ED 501 PENED'99 funded by G.S.R.T. of the Greek Ministry of Development and the project PRENED'99 funded by the Research Committee of the Democritus University of Thrace.

*Corresponding author. Tel.: +30-25410-79961.

E-mail addresses: ktatas@ee.duth.gr (K. Tatas), mdasyg@ee.duth.gr (M. Dasygenis), nkroup@ee.duth.gr (N. Kroupis), anargyr@ece.gatech.edu (A. Argyriou), dsoudris@ee.duth.gr (D. Soudris), thanail@ee.duth.gr (A. Thanailakis).

1077-2014/$ - see front matter © 2003 Elsevier Ltd. All rights reserved.
doi:10.1016/j.rti.2003.09.006

also critical, performance evaluation is included in order to allow the designer to assess the methodology's impact on performance. Additionally, performance optimizing transformations [6] are employed in order to increase performance and lessen the impact of some data-reuse transformations that do not affect performance in a positive way. These are software techniques of modifying the code for superior performance without affecting functionality.

More specifically, in this paper, all the above software techniques are combined in a cohesive, high-level design methodology for exploring embedded system architecture trade-offs in terms of data memory power consumption, performance and data memory area and applied to four popular motion estimation algorithms. The exploration is undertaken for both a single-processor and a multiprocessor environment. In the case of multiple processors, partitioning of the application is required in order to implement it utilizing all available processors. Three different possible target embedded architecture models are explored in combination with 21 different data memory hierarchies reflected in the high-level descriptions as data-reuse transformations, yielding a plethora of implementations. The exploration that will be presented determines the optimal memory hierarchy/target architecture combination according to the design specifications. The results clearly indicate a direct relation between the complexity of the algorithm, i.e. number of operations, and the power savings obtained by this strategy.

This paper is organized as follows: Section 2 provides a description of the assumed target architecture and the experimentation methodology used. Section 3 describes our target applications. The proposed methodology including the employed transformations and partitioning scheme is explained in Section 4. Experimental results with detailed analysis are presented in Section 5. The paper concludes with Section 6.

2. Target architecture and experimentation methodology

2.1. Target architecture models

Custom processors possess the advantage of high performance at the expense of flexibility and architectures for multimedia applications have been proposed [7,8]. On the other hand, programmable processors achieve great flexibility at the expense of performance. A programmable processor environment is the selected platform in this proposed work. We considered both a single and a multiprocessor environment of n processors. In the case of multiple processors, three distinct architectures are derived according to the data memory organization. Each processor in the three considered multiple processor architectures has its own application specific data memory hierarchy (ASDMH) [5,9]. ASDMH is explored in combination with three well-established data memory architecture models, namely: (1) distributed data memory hierarchy (DMH), (2) shared data memory hierarchy (SMH) and (3) shared-distributed data memory hierarchy (SDMH) as they are shown in Figs. 1–3, respectively. For all these data memory architectures, a shared background memory module (off-chip) is assumed. In the case of DMH a separate data memory hierarchy exists for each processor (Fig. 1). Therefore, all memory modules are single ported, but in the case of a lot of common data shared by the processors, a significant area penalty is probable. SMH implies a common memory hierarchy shared by all processors (Fig. 2). Since in the data-dominated processing domain it is very difficult and performance-inefficient to schedule all memory accesses sequentially, we assume that the number of ports for each memory block equals the maximum number of parallel accesses to it. Finally, in the case of SDMH, being a combination of the above two models, the common data to the N processors are placed in a shared memory hierarchy, while a separate data memory hierarchy exists for the lowest levels of the hierarchy (Fig. 3). For demonstration purposes, we have examined the case of $N = 1$ and 2 without any restrictions about memory hierarchy levels.

2.2. Performance, power and area metrics

In order to demonstrate the efficiency of the proposed methodology, high-level estimation models are required, so that the various assumed implementations (optimized and non-optimized) can be compared in terms of performance, memory area and memory power consumption. Taking the target architecture into account, and using Landman's memory energy consumption model [10], measurements of the efficiency of the algorithm, in terms of data memory energy and performance, related to the ARM core can be obtained. In particular, Landman presented a black box capacitance model for architectural energy analysis. Given its capacitance, the energy consumption is calculated by

$$E = \frac{1}{2} V_{dd}^2 (C_{read} \cdot \#read\_accesses + C_{write} \cdot \#write\_accesses), \qquad (1)$$

where C_{read} and C_{write} denote the capacitances for read and write operation and *#read_accesses* and *#write_accesses* are the number of read and write operations, respectively. The values of C_{read} and C_{write} are specified by the model itself, while the number of accesses depends on the chosen application, and they are calculated by the designer.

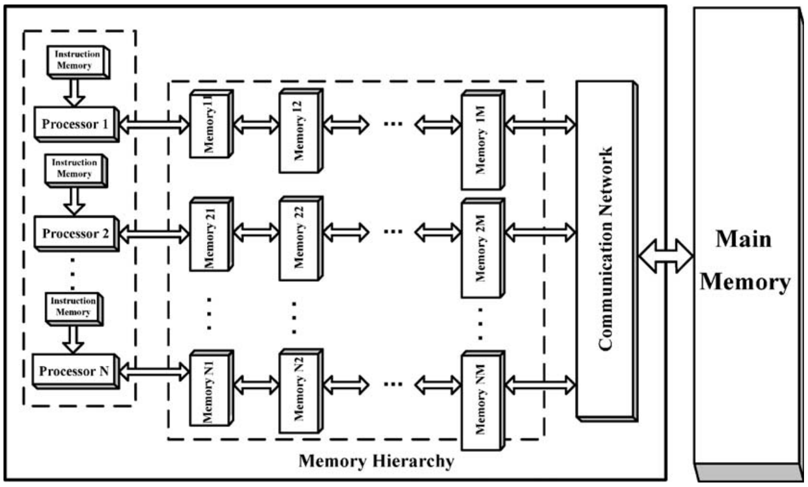

Fig. 1. The distributed data memory hierarchy (DMH) architecture model.

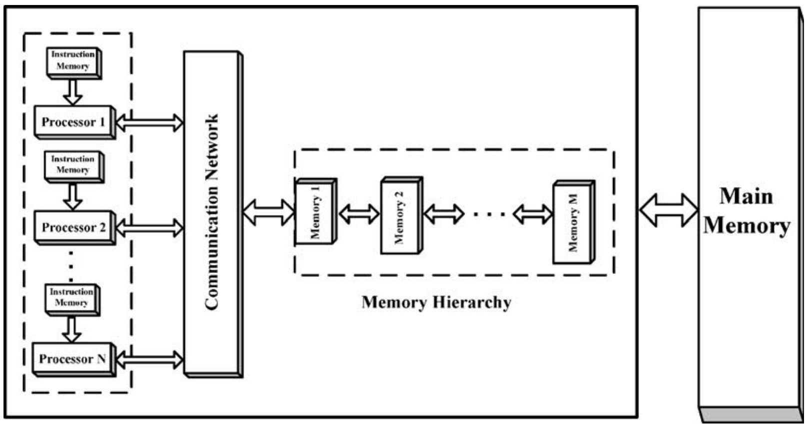

Fig. 2. The shared data memory hierarchy (SMH) architecture model.

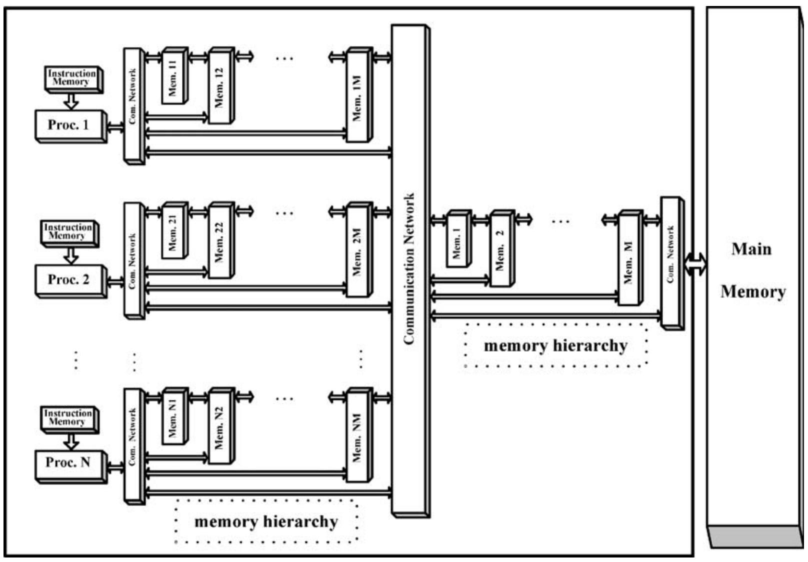

Fig. 3. The shared-distributed data memory (SDMH) architecture model.

In order to obtain performance measurements, we assume that on-chip data memory accesses need one processor cycle per access. In the case of multiple processors, the latency of the system is assumed to be equal to the latency of the slowest processor (in the case of heterogeneous structure). In other words, we use the number of executed cycles resulting from the considered processor core simulation environment. Therefore,

$$Delay = \max_{i=1,\dots,N} \{\#cycles\_processor_i\},$$

where $\#cycles\_processor_i$ denotes the number of cycles required for the ith processor to execute its respective part of the application and it is provided by the processor simulation environment. Here, for experimental reasons we use the ARMulator [11].

For the area occupied by the memories, Mulder's model is used [12]. The total area cost is

$$Area\_cost = \sum_{j=1}^{N} Area\{word\_length(j)\ \#words(j), \#ports(j)\},$$

where j is a memory element.

3. Demonstrator application

As mentioned above, our target applications are widely used motion estimation multimedia kernels. It has been noted that the motion estimation complexity stands for 60–80% of the total computational complexity of video encoding [13,8]. On the other hand, the nested loop structure of these algorithms leads to heavy data reuse, which makes these applications suitable for candidates to demonstrate the proposed methodology. The following subsections provide a brief overview of the main search area strategies and matching criteria, which are used for the function of motion estimation.

We consider the following Motion Estimation Algorithms: (i) Full Search (FS) [13], (ii) 3-Step Logarithmic Search (3SLOG) [9], (iii) Parallel Hierarchical One-Dimensional Search (PHODS) [13] and (iv) Hierarchical Search (HS) [14]. All these algorithms are block based, which means that in order to compute the motion vectors between two successive frames, the previous frame and the current frame are divided into a number of blocks, each of which have a fixed number of pixels. The algorithm compares every block from the current frame with the block from previous frames that are placed near the original point. Considering a cost function, the best match between a previous and a current block frame is the one exhibiting the minimum value in the cost function. This results to the motion vectors $MV(i,x)$, $MV(i,y)$, of the ith block and x- and y-axis, respectively.

Since the search is being performed among rectangular regions, the cost function is referred to as a block matching criterion (BMC), where as the search techniques to find the motion vectors, which yield the smallest cost, are referred to as block matching algorithms (BMA).

3.1. Full Search

The simplest method to compute the motion vectors of successive frames is to compute the cost function at each location in the search space. This is referred to as the full-search algorithm. Thus, for every block of the previous frame, an exhaustive search in all the nearby blocks of the current frame within a radius p, called candidate blocks, is being made, in order to find the current block that yields the smallest cost (or minimizes the cost function of the distance criterion). A pseudo-code for the FS is illustrated in Fig. 4. The simplicity of this code can be seen in its C language description, where three sets of regular loops exist: (i) one set allows to compute the motion vectors for every block of the previous frame, (ii) one set allows searching within the search space in the current frame and (iii) one set allows performing calculations for the distance criterion for every pixel of the two frames.

Although FS is a computationally expensive algorithm, it guarantees finding the blocks that yield the minimum cost value. Due to its high computational complexity, alternative search methods are desirable. Algorithms like 3SLOG, PHODS, and HS achieve suboptimal performance at significantly reduced complexity compared to the FS method. By the term suboptimal, we imply that the heuristic search strategies do not guarantee that blocks with the minimum cost function will be found. The complexity is significantly reduced either by decreasing the number of search locations (3SLOG, PHODS) or by computing using fewer pixels per frame (HS).

3.2. 3-Step Logarithmic Search

Another popular scheme for motion estimation using fewer search locations is the Logarithmic Search. In this algorithm, the search rectangle $[-p, p]$ is divided into two areas: one inside a $[-p/2, p/2]$ rectangle and one outside it. Instead of searching the whole area of

```
for(y=0;y<N/B;y++)
 for(x=0;x<M/B;x++)
 {
  for(i=-p;i<p+1;i++)
   for(j=-p;j<p+1;j++)
   {
    for(m=0;m<B;m++)
    for(n=0;n<B;n++)
    {
      read_from_current_frame;
     Check_Bound_Condition;
     read_from_previous_frame;
    Distance_Criterion();
    }
   }
 }
```

Fig. 4. Full Search pseudo-code.

```
for(y=0;y<N/B;y++)
 for(x=0;x<M/B;x++)
 {
  while(d>0)
  {
   for(i=-d;i<d+1;i+=d)
    for(j=-d;j<d+1;j+=d)
    {
     for(m=0;m<B;m++)
      for(n=0;n<B;n++)
      {
       read_from_current_frame;
       Check_Bound_Condition;
       read_from_previous_frame;
       distance_criterion_check();
      }
     d=d/2; /*Change the step */
    }
  }
 }
```

Fig. 5. 3-Step Logarithmic Search pseudo-code.

$[-p/2, p/2]$ rectangle, the distance criterion is computed at the origin and at the eight major points in the perimeter of the $[-p/2, p/2]$ area. This means that if the distance between these points is d, we compute the cost function at the points (0,0), (0,d), (0,$-d$), ($-d$,0), (d,0), (d,d), (d,$-d$), ($-d$,d) and ($-d$,$-d$). The distance d is given by

$$d = 2^{k-1},$$

where $k = \lceil \log_2 p \rceil$. The most popular form is when $k = 3$ and $p = 7$ and is called "3-Step Logarithmic". Using the best match among the eight points as a new origin, we perform a new similar search with a new step $d \leftarrow d/2$ in the major eight points in the perimeter of the new rectangle. Finally, a third search is performed in a similar way. Pseudo-code for this algorithm is shown in Fig. 5.

3.3. Parallel Hierarchical One-Dimensional Search

Another motion estimation scheme that is widely used in embedded multimedia systems is the PHODS. This algorithm is different from the remaining motion estimation kernels, because the search for the block that minimizes the cost function is performed in one dimension (vertically or horizontally) every time unit, independently from the other. The combination of the motion vectors of the abscissa and ordinate are the motion vectors of the two successive frames. Specifically, as illustrated in the pseudo-code of Fig. 6, the search occurs on the two dimensions of the search space independently, and the motion vector is described by the merged coordinates (dx, dy) of the best matching block on x and y dimension, respectively. Thus, first a search on the x dimension is performed, comparing the three blocks located at the points by $-D$, 0 and D of the x-axis. The cost function is used to find the block in this

```
for(y=0;y<N/B;y++)
 for(x=0;x<M/B;x++)
 {
  while(S>0)
  {
   /* For all candidate blocks in X dimension */
   for(i=-S;i<S+1;i+=S)
    {
     for(m=0;m<B;m++)
      for(n=0;n<B;n++)
      {
       read_from_current_frame();
       Check_Bound_Condition;
       read_from_previous_frame();
       distance_criterion_check();
      }
    }
    /* For all candidate blocks in Y dimension */
   for(i=-S;i<S+1;i+=S)
   {
    for(m=0;m<B;m++)
     for(n=0;n<B;n++)
      {
       read_from_current_frame();
       Check_Bound_Condition;
       read_from_previous_frame;
       distance_criterion_check();
      }
      S=S/2;
   }
  }
```

Fig. 6. Parallel Hierarchical One-Dimensional Search (PHODS) search pseudo-code.

axis that is the best match, producing the motion vector dx. After computing the vector dx, an independent search on the y dimension is performed, which computes the motion vector dy, comparing the three blocks located at the points $-D$, 0 and D. The search begins with step $D = \lfloor \log_2 p \rfloor$, where p is the search space parameter. In our analysis, we used as search parameter $p = 7$ pixels, which means $D = 4$. After the completion of the motion estimation on both axis by $-D$, 0, and D, the search step D is divided by two, with $D \leftarrow D/2$ and the new origin for this block is $(x + dx, y + dy)$. This origin is the best match of the previous cycle. The algorithm is repeated with the new search step around the new starting point. The computation of the motion vectors of the block is terminated when D becomes zero.

3.4. Hierarchical Search

A motion estimation scheme that uses a search strategy of both fewer search locations and fewer pixels in computing the cost function, is HS. In this algorithm, two low-resolution versions of the current frame and the reference frame are formed by subsampling both of them by a factor of two and four. The motion vector computation begins at the lowest resolution frame (level 2), where the search space is $\frac{1}{4}$ of the original one. Thus, if we assume that the block of the current frame is located at (x, y), the corresponding block of levels 1 and

```
create_current_frame_subsampled_by_two();
create_current_frame_subsampled_by_four();
create_previous_frame_subsampled_by_two();
create_previous_frame_subsampled_by_four();
/* ------------- Level 2 ------------- */
for(i=-p/4;.....) /* ME at fr. subsamp. by 4 */
  for(j=-p/4;.....)
  {
    for(m=0;.....)
     for(n=0;.....)
     {
      read_from_current_frame_4();
      Check_Bound_Condition;
      read_from_previous_frame_4();
      distance_criterion_check();
     }
  }
/* ------------- Level 1 ------------- */
for(i=-1;.....) /* ME at fr. subsamp. by 2 */
  for(j=-1;.....)
  {
   for(m=0;......)
    for(n=0;......)
    {
     read_from_current_frame_2();
     Check_Bound_Condition;
     read_from_previous_frame_2();
     distance_criterion_check();
    }
  }
/* ------------- Level 0 ------------- */
for(i=-1;.....) /* ME at original frame */
  for(j=-1;.....)
  {
   for(m=0;.....)
    for(n=0;.....)
    {
     read_from_current_frame_0();
     Check_Bound_Condition;
     read_from_previous_frame_0();
     distance_criterion_check();
    }
  }
```

Fig. 7. Hierarchical Search motion estimation pseudo-code.

2 is located at position ($x/2$, $y/2$) and ($x/4$, $y/4$), respectively. Having found the motion vectors in level 2, the algorithm performs a FS with search space $p = \pm 1$ block in level 1, using as a starting point the coordinates of the motion vector of the level 2. This means that the search is performed with the origin being the block that corresponds to the level 2 block with the closest match. If necessary, the motion vectors are updated with the feedback, or left unchanged. Similarly, the algorithm performs a FS scheme with search space $p = \pm 1$ at level 0, with origin the new motion vectors of the level 1. The location that yields the smallest value in the cost function corresponds to the final motion vector output. Because the FS method is used for motion estimation at each level of the hierarchy, the algorithm structure is very similar with FS, as it can be seen in the pseudo-code presented in Fig. 7. The first four functions of the algorithm create two low-resolution frames for the previous frame, subsampling the frame by a factor of two and four. Thus, if the original frame's dimensions are $M \times N$ pixels, the dimensions of the subsampled versions of it by 2 and 4 are going to be $M/2 \times N/2$ and $M/4 \times N/4$, respectively.

Block-matching motion estimation algorithms compute the motion vectors by minimizing a cost function, which is usually called distance criterion. Various distance criteria have been proposed and analyzed in literature [8]. In our case, the sum of absolute differences (SAD) [13], has been selected. SAD is one of the most useful criteria and its function is to compare the sum of the absolute difference of the luminance values of pair of pixels, which belong to previous and the current frame. Its mathematical operations are the addition of the pixel's luminance and the absolute difference of the values. The addition and the absolute difference operations are fundamental arithmetic operations for the processor, which is translated to lower computational complexity for the ME algorithm. The associated formulas are:

$$SAD(dx, dy) = \sum_{m=x}^{x+B-1} \sum_{n=y}^{y+B-1} |I_f(m,n) - I_{f-1}(m+dx, n+dy)|,$$

$$(MV_x, MV_y) = \min_{(dx,dy)\in R^2} SAD(dx, dy).$$

4. Proposed methodology

The proposed methodology is illustrated in Fig. 8. Since both performance and power estimation are critical, steps for the improvement of both are included. A high-level (C language) description of the application is the input to the first stage, known as the performance optimizing transformation stage. Performance optimizing transformations [6] such as common subexpression elimination are established techniques that increase performance. These performance optimizing transformations can actually be applied at any stage, but it is less time consuming to apply them first, and certainly before parititioning the application. Therefore, first the code is modified for improved performance. In [1] the emphasis is on the data memory power consumption, and therefore a performance optimization stage is not included in that approach. The second stage is the global loop transformation stage. Global loop transformations [1] such as loop merging and tiling are used to increase the regularity of the loop structure of the application. This is most often necessary in multi-kernel algorithms such as HS in our case, but not essential in the case of full-search which has a very regular nested loop structure as seen in the previous section. The output of this stage is a transformed code with greater regularity.

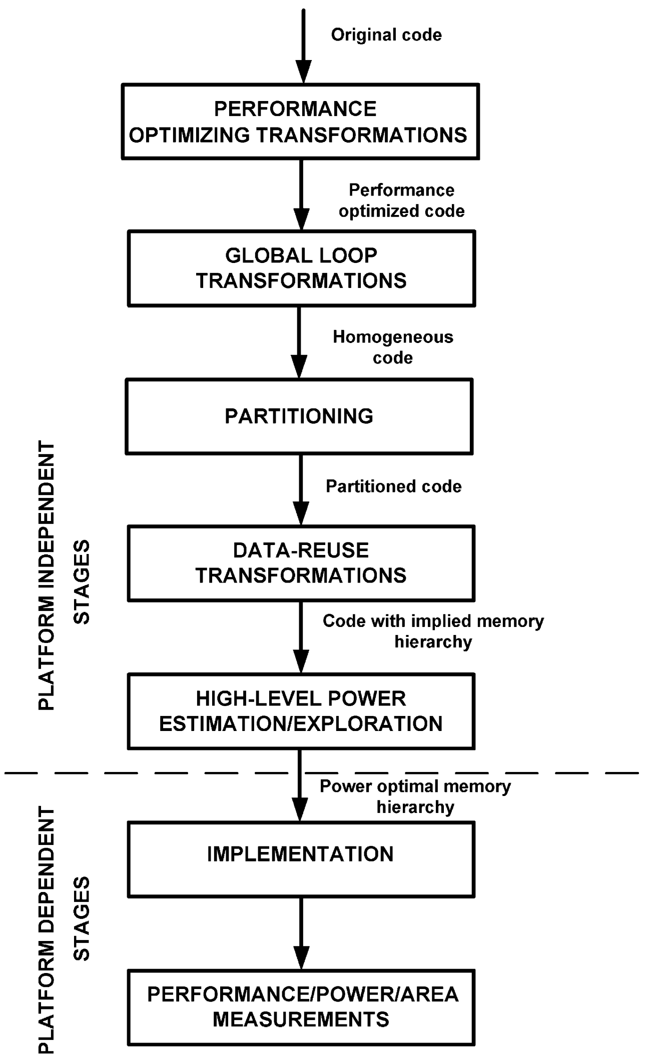
Original code
PERFORMANCE OPTIMIZING TRANSFORMATIONS
Performance optimized code
GLOBAL LOOP TRANSFORMATIONS
Homogeneous code
PARTITIONING
Partitioned code
DATA-REUSE TRANSFORMATIONS
Code with implied memory hierarchy
HIGH-LEVEL POWER ESTIMATION/EXPLORATION
Power optimal memory hierarchy
IMPLEMENTATION
PERFORMANCE/POWER/AREA MEASUREMENTS
PLATFORM INDEPENDENT STAGES
PLATFORM DEPENDENT STAGES

Fig. 8. The proposed design optimization methodology.

The homogenized code then undergoes data-reuse transformations that imply a specific on-chip memory hierarchy. In other words the code is modified once more in order to reflect the data memory hierarchy of each possible implementation. The basic concept behind data-reuse transformations is that to place data that are used often in smaller memories, use them as many times as necessary, and then load the memories with new data, effectively increasing the total number of data transfers but moving them to smaller memories.

Then, high-level data memory power exploration is performed, by using high-level power estimation models. Comparisons among the various transformations yield the most efficient memory hierarchy in terms of data memory power and data memory area. Since the number of memory transfers required by the application is independent of the specific implementation, the first three steps are platform independent.

The next step is where the platform dependent part of the methodology begins. The performance measures are taken from the ARMulator simulation environment, in other words an ARM-based programmable multiprocessor environment is assumed. The methodology and exploration strategy are explained in greater detail in the next subsections.

4.1. Performance optimizing transformations

A category of performance optimizing transformations known as common subexpression elimination was applied to the source code of the application. Specifically, mathematical expressions featuring constants that appeared often in the source code were also declared as constants, and therefore evaluated once at the beginning of the code's execution instead of every time they appear during the program's execution. This is particularly effective in the case of expressions which include multiplication and division, since these operations require a significant number of processor cycles for their execution.

4.2. Global loop transformations

Global loop transformations such as loop merging and tiling can lead to a more homogeneous source code. They do not change the functionality, nor do they directly affect performance or power consumption. They are meant as an intermediate step that enables the application of the following steps of the methodology. In the case of HS application, direct application of data-reuse transformations would be difficult and it would not lead to optimal power savings, due to the nature of the algorithm which contains more than one kernel. Whereas, the application of global loop transformations not only makes the subsequent application of data-reuse transformations significantly easier, but it also enables the partitioning scheme employed here, known as locally serial, globally parallel (LSGP) [15], which will be explained in the following section, allowing us to implement our application in a multiprocessor environment. In the case of HS, the subsampling by 2 and 4 kernels were merged with the main motion estimation kernel, forming one global loop.

4.3. Partitioning scheme

In order to implement our applications in a multiprocessor environment, partitioning is essential. It should be performed in such a way that the total processing load is distributed among all processors as evenly as possible, in order to achieve optimal resource allocation, and therefore performance. For that reason, partitioning was done using the LSGP technique. In the case of p partitions the form of the partitioned motion estimation algorithms is the one shown in Fig. 9. Our experiments were carried out assuming $p = 2$, meaning

```
Do in parallel:
Begin
for(x=0;x<⌈N/(pB)⌉;x++){sub-algorithm}
for(x=⌈N/(pB)⌉;x<⌈2N/(pB)⌉;x++){sub-algorithm}
:
for(x=⌈((p-1)N)/pB⌉;x<⌈N/B⌉;x++){sub-algorithm}
end;
```

Fig. 9. The partitioned motion estimation algorithms.

2 partitions. Therefore, the first processor executes the algorithm for loop index x range 0 to $\lceil N/2B \rceil - 1$ and the second one for $\lceil N/2B \rceil$ to $N/B-1$ in a parallel fashion.

4.4. Data-reuse transformations

The basic concept behind employing data-reuse transformations is that a smaller memory consumes less power than a larger one, and actually two or more memories consume less power than a single one with a total capacity which equals the multiple memories' combined capacity. Also, an on-chip data transfer is much less power-consuming than an off-chip one. Therefore, a well-known technique consists of moving data that are used many times to small local memories, so that the greatest number of memory transfers is from and to small, local memories. Even if the actual number of data transfers is increased that way, the total power consumption decreases. The use of such memory data transfer manipulation techniques has been explored and applied to motion estimation algorithms before [5,16]. The key is to decide which data sets are appropriate to be placed in a (small) separate memory. Otherwise, a number of different memories will be required for each data set resulting to significant area penalty.

Let us consider the various data sets that appear in our multimedia applications. The entire previous and current frames are divided into blocks. Each block of the current frame is compared with a number of blocks from the previous frame which are inside a reference window. This is performed for all blocks in the x dimension and then the y dimension index is incremented by one and this procedure is repeated. Therefore the data sets that can be reused are a single block or a line of blocks from the current frame, a single candidate block, a line of candidate blocks, a single reference window and a line of reference windows from the previous frame. All combinations of the above data sets lead to 21 different memory hierarchies that can be expressed in the high-level description (for example C code) as 21 different data-reuse transformations. We examined and evaluated the effect of these 21 data-reuse transformations on the target architectures described in the previous section in terms of power, performance, and area. The transformations were applied after partitioning. They involved the insertion of memories for a line of current blocks (CB line), a current block (CB), a line of candidate blocks (PB line), a candidate block (PB), a line of reference windows (RW line), and a reference window (RW). The data memory hierarchies that correspond to all 21 transformations (copy tree) can be seen in Fig. 10. It is identical for all processors and target architectures and the dashed lines indicate the memory hierarchy levels. Each rectangle contains three labels, where the number implies the applied data-reuse transformations associated with the memory hierarchy level.

The memory hierarchy for the first transformation is shown in an ellipse and the second in a polygon in order to clarify this. In the case of the hierarchical motion estimation algorithm, there should be additionally four rectangles on the left side corresponding to the subsampled frames, but we applied no data-reuse transformations to the subsampled frames. The reason is that these memories are too small themselves to significantly benefit from data-reuse transformations. Therefore, this strategy was only applied to the original frames and the copy tree of the HS algorithm degenerates to the one in Fig. 10.

A short description of how each transformation was applied in the case of our applications follows. The on-chip memory size for every transformation is also given. Each term in the right part of the equation corresponds to the memory size of a specific on-chip data memory hierarchy level (Fig. 10). We consider two frames of sizes: $image\_length \times image\_width$ (pixels), divided in square blocks of $block\_width \times block\_width$ (pixels). The search space in pixels is denoted by $search\_space$. An example is given for the first two transformations

4.4.1. 1st Transformation description

The data set that was reused in the first transformation is a line of candidate blocks from the current frame. In other words a line of blocks from the current frame (CB line) which is stored in the background (external) memory is copied to a local memory. The data is used as many times as they are required and then a new line of blocks is loaded to the local memory.

Memory hierarchy: This transformation introduces one additional level of data memory hierarchy. The size of the on-chip memory is

$$Mem_{on-chip} = \overbrace{[block\_width \times image\_width]}^{1st\_hierarchy\_level} \times pixel\_bits \text{ (bits)},$$

where $pixel\_bits$ is the number of bits per pixel, i.e. 8 for grayscale frames and 24 for color frames.

The memory hierarchy of this particular transformation is indicated by an ellipse in Fig. 10.

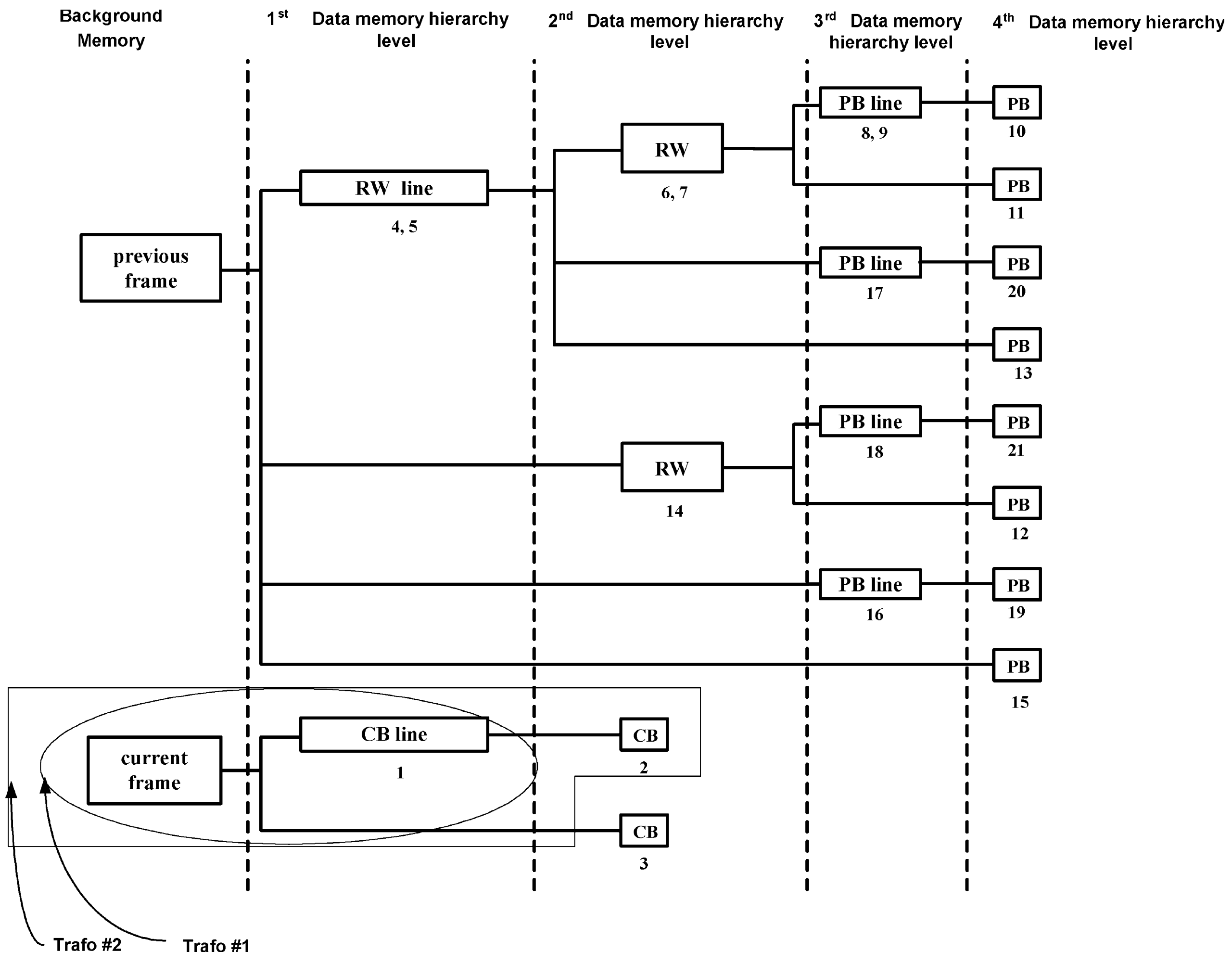

Fig. 10. The copy tree structure of the motion estimation algorithms.

4.4.2. 2nd Transformation description

The second transformation is an extension of the first one that introduces an additional level of hierarchy. After copying a line of blocks from the current frame (CB line) to a local memory, a single block (CB) is copied to an even smaller, also local, memory. After one block is used entirely, the next one is loaded from the line of blocks, until the entire line of blocks is exhausted. Afterwards, a new line of blocks is loaded to the local memory and this process begins anew. The memory hierarchy of this transformation is also shown in Fig. 10.

Memory hierarchy: This transformation introduces two levels of data memory hierarchy. The size of the on-chip memory is

$$Mem_{on-chip} = [\overbrace{block\_width \times image\_width}^{1st\_hierarchy\_level} + \overbrace{block\_width \times block\_width}^{2nd\_hierarchy\_level}] \times pixel\_bits \text{ (bits).}$$

4.4.3. 3rd Transformation description

In the case of the third data-reuse transformation that was employed, a block of the current frame (CB) is copied directly from the external memory to a local one, without fetching an entire line of blocks first.

Memory hierarchy: Only the second level of hierarchy is present, therefore the size of the required on-chip memory is

$$Mem_{on-chip} = [block\_width \times block\_width] \times pixel\_bits \text{ (bits).}$$

4.4.4. 4th Transformation description

The fourth transformation consists of copying and reusing a reference window line (RW line) from the previous frame. It introduces one level of hierarchy like the first transformation, but this time on the previous frame data and not the current frame.

Memory hierarchy: This transformation introduces one level of data memory hierarchy. The size of the on-chip memory is

$$Mem_{on-chip} = [(block\_width + 2 \times search\_space) \times image\_width] \times pixel\_bits \text{ (bits)}.$$

4.4.5. 5th Transformation description

The fifth transformation consists of also copying a line of reference windows (RW line) to a local memory, but this time, overlapping data between reference windows (RW), are copied from the reference window line itself instead from the external memory which stores the entire previous frame.

Memory hierarchy: The memory hierarchy and the size of the required on-chip memory is exactly the same as in the previous transformation.

4.4.6. 6th Transformation description

The sixth transformation is an extension of the fourth one, introducing a second level of data memory hierarchy. After a reference window line has been copied, a single reference window is copied to a smaller internal memory. After the reference window has been used as many times as necessary, the second one is fetched from the memory containing the reference window line, until all reference windows of the line have been exhausted. Then, a new reference window line is loaded to the second hierarchy level memory, and the entire procedure begins once more.

Memory hierarchy: This transformation introduces two levels of data memory hierarchy. The size of the on-chip memory is

$$Mem_{on-chip} = [(block\_width + 2 \times search\_space) \times image\_width + (block\_width + 2 \times search\_space) \times (block\_width + 2 \times search\_space)] \times pixel\_bits \text{ (bits)}.$$

4.4.7. 7th Transformation description

Transformation number 7 is a similar extension of the fifth one. It introduces a second level of on-chip data memory hierarchy by copying a reference window from a line of reference windows that is copied from the external memory. The difference between this transformation and the previous one is that the reference window this time is loaded in a similar manner to that in which the line of reference windows is, in other words overlapping data between reference windows are reused.

Memory hierarchy: This transformation implies the same data memory hierarchy and on-chip memory size as the previous one.

4.4.8. 8th Transformation description

This transformation introduces a third level of memory hierarchy. A line of candidate blocks (PB line) of the previous frame is loaded from the reference window, which in turn is loaded from a line of reference windows.

Memory hierarchy: This transformation implies three on-chip data memory hierarchy levels with a total on-chip memory size of:

$$Mem_{on-chip} = [(block\_width + 2 \times search\_space) \times image\_width + (block\_width + 2 \times search\_space) \times (block\_width + 2 \times search\_space) + block\_width \times (block\_width + 2 \times search\_space)] \times pixel\_bits \text{ (bits)}.$$

4.4.9. 9th Transformation description

This transformation is very similar to the previous one, only this time overlapping data are reused also in the case of the line of candidate blocks.

Memory hierarchy: The memory hierarchy and the size of the required on-chip memory is exactly the same as in the previous transformation

4.4.10. 10th Transformation description

In this transformation a fourth and final level of data memory hierarchy is introduced. From the line of candidate blocks that were loaded, a single candidate block (PB) is loaded and reused.

Memory hierarchy: Four levels of on-chip data memory hierarchy exist in this transformation. The size of the total on-chip memory is

$$Mem_{on-chip} = [(block\_width + 2 \times search\_space) \times image\_width + (block\_width + 2 \times search\_space) \times (block\_width + 2 \times search\_space) + block\_width \times (block\_width + 2 \times search\_space) + block\_width \times block\_width] \times pixel\_bits \text{ (bits)}.$$

4.4.11. 11th Transformation description

In transformation number 11, the candidate block is loaded directly from the reference window, without an intermediate line of candidate blocks. In other words, the third level of data memory hierarchy is missing.

Memory hierarchy: Three levels of memory hierarchy exist, namely the first, second and fourth. The total memory size is given by the formula:

$$Mem_{on-chip} = [(block\_width + 2 \times search\_space) \times image\_width + (block\_width + 2 \times search\_space) \times (block\_width + 2 \times search\_space) + block\_width \times block\_width] \times pixel\_bits \text{ (bits)}.$$

4.4.12. 12th Transformation description

This transformation includes only three of the five levels of data memory hierarchy. Specifically, a single reference window is loaded from the external memory, and then a single candidate block is loaded from the reference window.

Memory hierarchy: Data memory hierarchy levels 1 and 3 are absent. The total memory size is

$$Mem_{on-chip} = [(block\_width + 2 \times search\_space) \times (block\_width + 2 \times search\_space) + block\_width \times block\_width] \times pixel\_bits \text{ (bits).}$$

4.4.13. 13th Transformation description

The 13th transformation consists of copying a single candidate block directly from the reference window line, with no other intermediate copies (hierarchy levels).

Memory hierarchy: Only data memory hierarchy levels 1 and 4 are present. The total memory size is

$$Mem_{on-chip} = [(block\_width + 2 \times search\_space) \times image\_width + block\_width \times block\_width] \times pixel\_bits \text{ (bits).}$$

4.4.14. 14th Transformation description

This transformation contains only a reference window that is copied directly from the external memory which stores the whole previous frame.

Memory hierarchy: Therefore, only two hierarchy levels are present, namely the first and third one, with a combined memory size of:

$$Mem_{on-chip} = [(block\_width + 2 \times search\_space) \times image\_width + block\_width \times (block\_width + 2 \times search\_space)] \times pixel\_bits \text{ (bits).}$$

4.4.15. 15th Transformation description

In transformation number 15, a single candidate block is copied directly from the external memory that contains the entire previous frame image.

Memory hierarchy: Only the last level of data memory hierarchy is present. The on-chip memory size is

$$Mem_{on-chip} = [block\_width \times block\_width] \times pixel\_bits \text{ (bits).}$$

4.4.16. 16th Transformation description

This transformation also contains only one level of hierarchy. This time a line of candidate blocks of the previous frame is copied from the external memory.

Memory hierarchy: Only hierarchy level 3 is present. The on-chip data memory size is given by

$$Mem_{on-chip} = [block\_width \times (block\_width + 2 \times search\_space)] \times pixel\_bits \text{ (bits).}$$

4.4.17. 17th Transformation description

A line of reference windows is loaded from the background memory. Then, a line of candidate blocks is loaded from this line of reference windows. Finally, a single candidate block is loaded from the line of candidate blocks.

Memory hierarchy: Three data memory hierarchy levels are present in this transformation. The memory size is

$$Mem_{on-chip} = [(block\_width + 2 \times search\_space) \times image\_width + block\_width \times (block\_width + 2 \times search\_space) + block\_width \times block\_width] \times pixel\_bits \text{ (bits).}$$

4.4.18. 18th Transformation description

A line of candidate blocks is loaded from a reference window which is in turn copied from the previous frame which is stored in the external memory.

Memory hierarchy: Hierarchy levels 2 and 3 are present in this transformation. The memory size is

$$Mem_{on-chip} = [(block\_width + 2 \times search\_space) \times (block\_width + 2 \times search\_space) + block\_width \times (block\_width + 2 \times search\_space)] \times pixel\_bits \text{ (bits).}$$

4.4.19. 19th Transformation description

In this transformation, a line of candidate blocks is loaded from the previous image and then a single candidate block is copied from the line of candidate blocks.

Memory hierarchy: This transformation includes the last two levels of the data memory hierarchy. In terms of on-chip memory size, we have

$$Mem_{on-chip} = [block\_width \times (block\_width + 2 \times search\_space) + block\_width \times block\_width] \times pixel\_bits \text{ (bits).}$$

4.4.20. 20th Transformation description

In this transformation, the second hierarchy level is missing, namely the reference window.

Memory hierarchy: The on-chip memory size is

$$Mem_{on-chip} = [(block\_width + 2 \times search\_space) \times image\_width + block\_width \times (block\_width + 2 \times search\_space) + block\_width \times block\_width] \times pixel\_bits \text{ (bits)}.$$

4.4.21. 21st Transformation description

The first level of data memory hierarchy is absent in this transformation. In other words, a reference window is copied from the entire previous frame. Then, a line of candidate blocks is copied from the reference window, and finally a candidate block is copied from the line of candidate blocks.

Memory hierarchy: The second, third and fourth levels of hierarchy are present and the total on-chip memory size is

$$Mem_{on-chip} = [(block\_width + 2 \times search\_space) \times (block\_width + 2 \times search\_space) + block\_width \times (block\_width + 2 \times search\_space) + block\_width \times block\_width] \times pixel\_bits \text{ (bits)}.$$

A comparative study of this partitioning scheme in combination with 21 data reuse transformations on power, performance and area was undertaken. In the case of the HS motion estimation scheme, the above-mentioned memory hierarchies were applied to level 0 (original images, not subsampled ones), where the maximum power savings can be obtained since the memories are big enough. At levels 1 and 2 (subsampled frames) the memories are too small to justify the insertion of buffers. Our exploration showed that even applying data reuse transformations to all levels, does not yield significantly better results than the ones obtained by applying the transformations only to the highest level (original images, not subsampled ones).

5. Experimental results

The objective of the presented work is to use software techniques (i.e. transformations) in order to explore various architecture trade-offs in performance, data memory power and data memory area in programmable platform implementations. The software transformations described in the previous sections lead to a plethora of possible alternative implementations. Comparisons among the three target architectures for all four motion estimation algorithms mentioned above for one and two processors in terms of data memory power consumption are illustrated in Figs. 11–14. The first conclusion that is evident from these graphs is the similar fashion in which the transformations affect the data memory power consumption of all four algorithms and all three memory hierarchies. The transformations reduce the data memory power consumption by approximately 40%, while the most efficient of them by over 50% for all four target applications. Still, not the same transformation is the optimum one for all models and algorithms.

Of the three memory hierarchies in the case of two processors, the SMH architecture is the most power-consuming of the three. The memory hierarchy generally consumes more energy in the case of one processor than two, with the exception of the SMH. This is explained by the fact that a single memory consumes more power than two memories with the same combined size. Therefore the SMH hierarchy, which includes a single memory that stores the data for both the processors consumes about the same as the single processor data memory hierarchy.

Furthermore, there is a direct connection between the computational complexity of the algorithm and the power dissipated during its execution. FS is by far the most computationally complex algorithm of the four and it consumes almost ten times more power than the other three. The least power-consuming of the four is HS with the remaining two (3-Step Logarithmic and PHODS) consuming about the same amount of power.

The effect of transformations on the performance of each memory hierarchy model and algorithm is shown in Figs. 15–18. Full-search, not surprisingly, requires the greatest number of cycles since it is the algorithm with the greatest number of computations. As expected, performance is benefited from partitioning, since it can be seen from the graphs that all three two-processor architectures perform better than the single processor one, for all four applications. The SMH and DMH architectures seem to be affected in the least by the transformations, while the SDMH architecture seems to be affected significantly and not always in a positive way. The optimal transformations in terms of performance are different for each model and each application kernel. In fact, PHODS and HS do not seem to be positively effected by the transformations, while the greatest benefit in performance seems to occur in the case of 3SLOG.

Finally, the data memory area for each memory hierarchy model and transformation for all algorithms is illustrated in Figs. 19–22. It can be inferred that each transformation influences area in almost identical manner for all data memory architectural models. It is also clear that all transformations increase area, since they impose the addition of extra data memory hierarchy levels. Moreover, for both DMH and

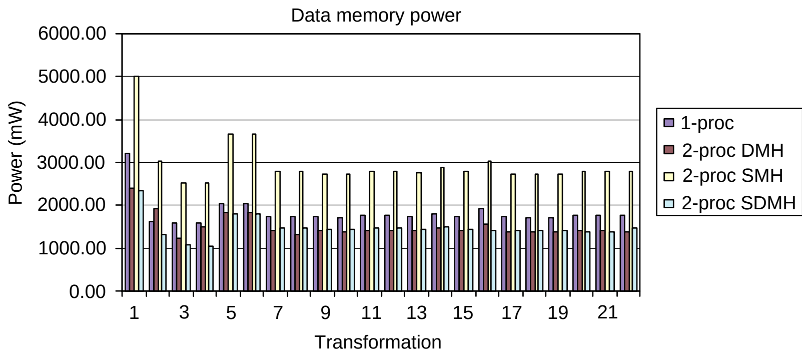

Fig. 11. Data memory power consumption for FS for 1 and 2 processors.

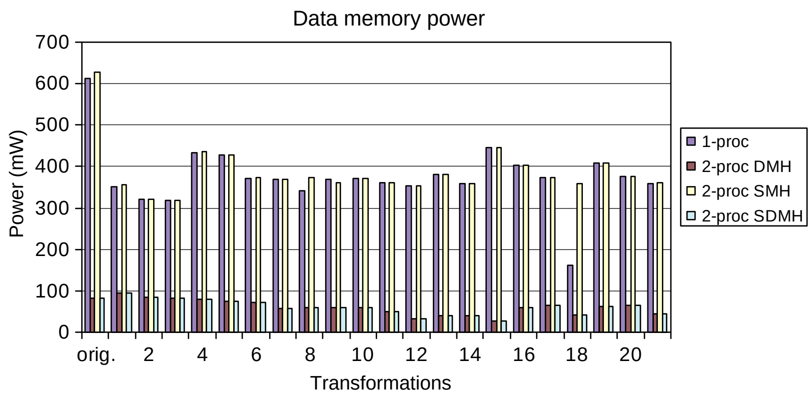

Fig. 12. Data memory power consumption of 3-Step Logarithmic motion estimation for 1 and 2 processors.

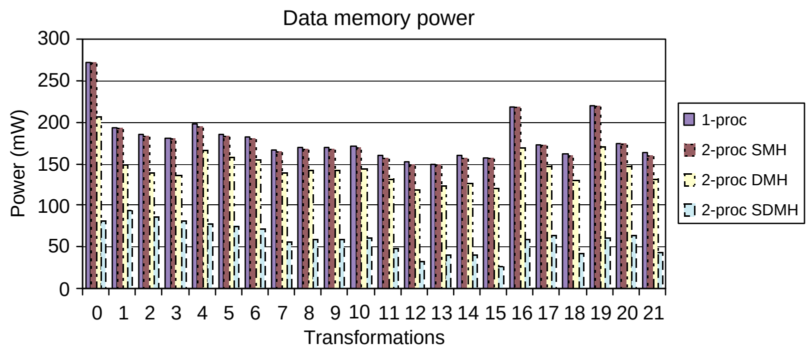

Fig. 13. Data memory power consumption of PHODS for 1 and 2 processors.

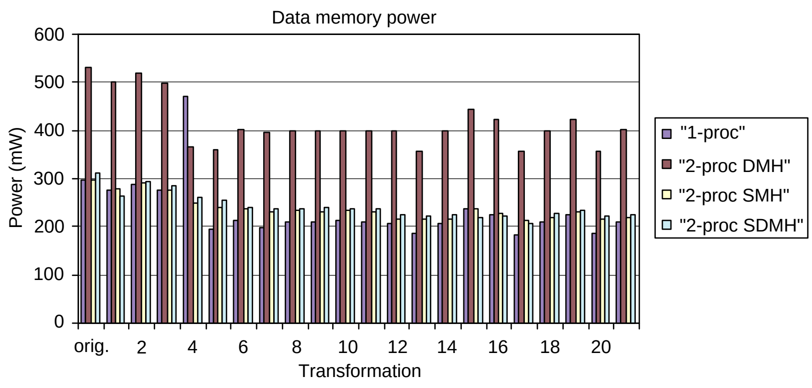

Fig. 14. Data memory power consumption of Hierarchical Search motion estimation for 1 and 2 processors.

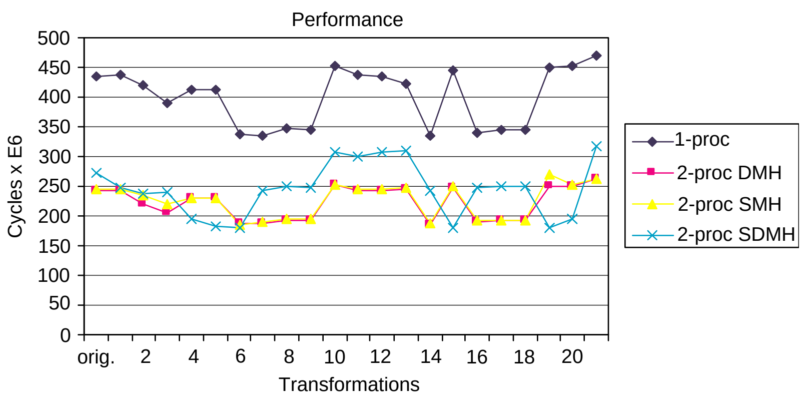

Fig. 15. Performance of FS.

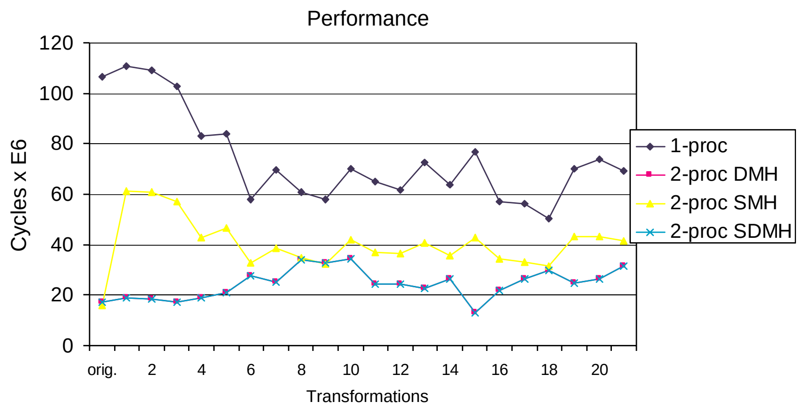

Fig. 16. Performance of 3-Step Logarithmic Search.

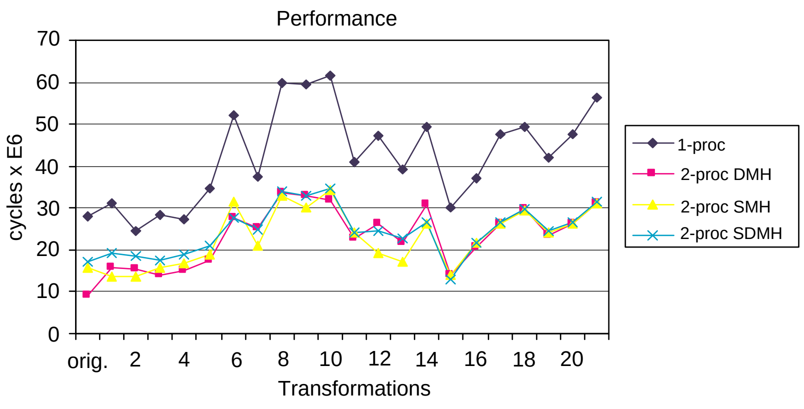

Fig. 17. Performance of PHODS.

SDMH the area cost is similar for each data-reuse transformation. In the case of SMH the area occupation is larger in all cases. This is due to the fact that several memory modules are dual ported, to be accessed in parallel by the processing elements. On the contrary, most memory modules are single ported and thus, they occupy less area. As it can be seen, the SDMH is the most area efficient, since in this data memory architecture model there are no memories in the hierarchy with duplicate data. Of the four algorithms, HS exhibits the greatest memory area (Fig. 22), since memories are required to store the subsampled frames, unlike the remaining three algorithms.

The above conclusions imply that there is no optimum transformation for every design aspect (performance, power, area) or every memory hierarchy model, or even for every algorithm. Instead, there is a plethora of possible implementations, allowing the

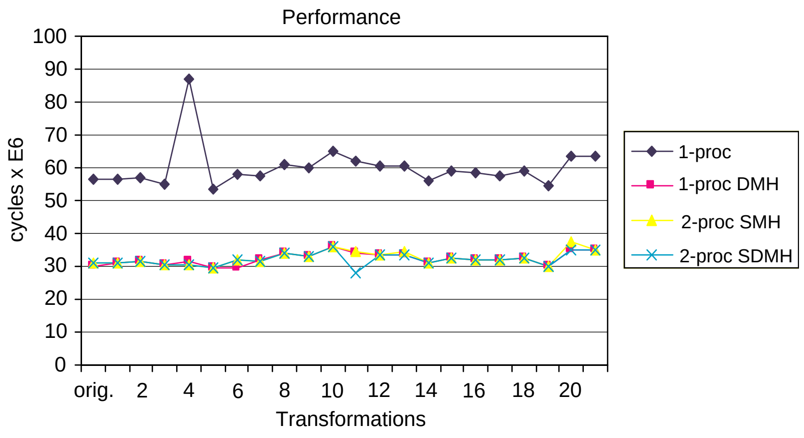

Fig. 18. Performance of HS.

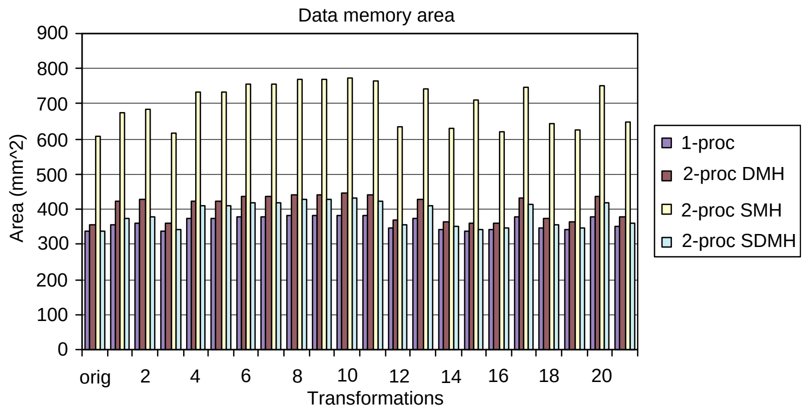

Fig. 19. Data memory area for FS.

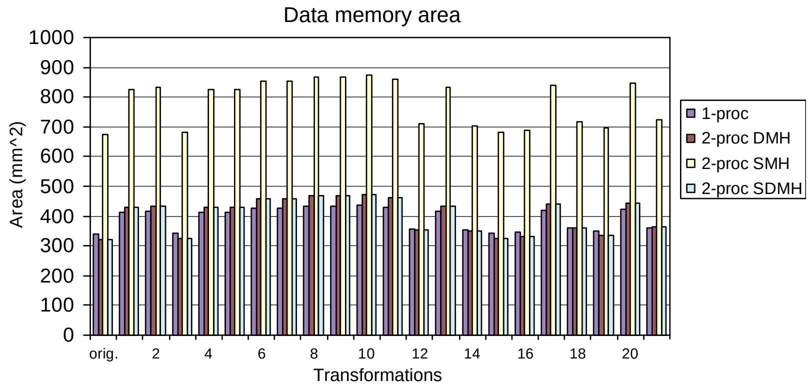

Fig. 20. Data memory area for 3-Step Logarithmic Search.

designer to select the appropriate implementation to fit the design constraints. Therein lays the greatest contribution of the methodology, allowing the designer to make the best choice before getting into more implementation details.

6. Conclusions—future work

A high-level design methodology for reducing the data memory power consumption and increasing performance for multimedia systems based on data-reuse

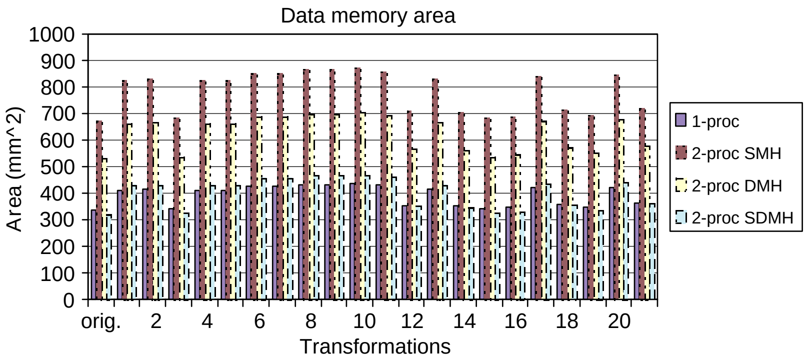

Fig. 21. Data memory area for PHODS.

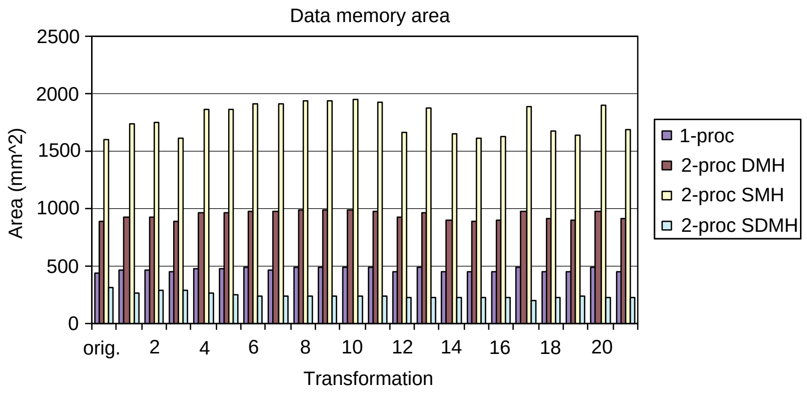

Fig. 22. Data memory area for HS.

transformations and LSGP partitioning was demonstrated on four popular multimedia kernels. The validity of such an exploration before the system design begins became evident from the experiments presented. No single "golden" transformation-memory architecture combination was found, but instead a great number of opportunities for performance, data memory power and data memory area trade-off was identified. This software transformation-based methodology is not limited to only the presented applications, but it is generally applicable in the data-dominated application domain.

References

[1] Catthoor F, et al. Data access and storage management on embedded programmable processors. Boston: Kluwer Academic Publishers; 2002.

[2] Chandrakasan A, Brodersen RW. Low power digital CMOS design. Boston: Kluwer Academic Publishers; 1998.

[3] Grun P, Dutt N, Nicolau A. Memory architecture exploration for programmable embedded systems. Boston: Kluwer Academic Publishers; 2003.

[4] Wolf W. Computers as components: principles of embedded computing system design. San Francisco: Morgan Kaufmann Publishers; 2001.

[5] Zervas ND, Masselos K, Goutis CE. Code transformations for embedded multimedia applications: impact on power and performance. In Proceedings of the Power-Driven Microarchitectures Workshop, ISCA 98, 1998.

[6] Aho AV, Sethi R, Ullman JD. Compilers: principles, techniques and tools. Reading: Addison-Wesley; 1986.

[7] Cheng SC, Hang HM. A comparison of block matching algorithms mapped to systolic-array implementation. IEEE Transactions on Circuits and Systems for Video Technology 1997;75:741–57.

[8] Kuhn P. Algorithms, complexity analysis and VLSI architectures for MPEG-4 motion estimation. Boston: Kluwer Academic Publishers; 1999.

[9] Jain J, Jain A. Displacement measurement and its applications in intraframe image coding. IEEE Journal of Transactions and Communication 1981;29:1799–808.

[10] Landman P. Low-power architectural design methodologies. Doctoral Dissertation, U.C., Berkeley; 1994.

[11] ARM software development toolkit, v2.11, Copyright 1996–97, Advanced RISC Machines.

[12] Mulder JM, Quach NT, Flynn MJ. An area model for on-chip memories and its application. IEEE Journal of Solid-State Circuits 1991;261:98–106.

[13] Bhaskaran V, Kostantinides K. Image and video compression standards. Boston: Kluwer Academic Publishers; 1998.

[14] Nam KM, Kim JS, Park RH, Shim YS. A fast hierarchical motion vector estimation algorithm using mean pyramid. IEEE Transactions on Circuits and Systems for Video Technology 1995;54:344–51.

[15] Kung SY. VLSI array processors. Englewood Cliffs: Prentice-Hall; 1988.

[16] Tatas K, Argyriou S, Dasygenis M, Soudris D, Zervas ND. Memory hierarchy optimization of multimedia applications on programmable embedded cores. San Jose: ISQED; 2001.