Journal of Circuits, Systems, and Computers
Vol. 13, No. 1 (2004) 151–180
© World Scientific Publishing Company

World Scientific
www.worldscientific.com

EVALUATING POWER EFFICIENT DATA-REUSE DECISIONS FOR EMBEDDED MULTIMEDIA APPLICATIONS: AN ANALYTICAL APPROACH

STAMATIKI KOUGIA
Intracom S.A., Department of Defense Systems Development, 19.5 Markopoulou Ave., 19002, Paiania, Athens, Greece

ALEXANDER CHATZIGEORGIOU
Department of Applied Informatics, University of Macedonia, Thessaloniki, 54006, Greece

SPIRIDON NIKOLAIDIS
Department of Physics, Aristotle University of Thessaloniki, Thessaloniki, 54006, Greece

Received 29 January 2002
Revised 16 April 2003

Power consumption of multimedia applications executing on embedded cores is heavily dependent on data transfers between system memory and processing units. The purpose of this paper is to extend an existing power optimizing methodology based on data-reuse decisions, in order to determine the optimal solution in a rapid and reliable way. An analytical approach is proposed by extracting expressions for the number of accesses to each memory layer. Moreover, the design space is further reduced since these analytical expressions are calculated only for a subset of all transformations. The results concerning the power efficiency of data-reuse transformations are in agreement to those in previous studies. However, the exploration time of the design space is significantly reduced. The proposed methodology is also applied to the case of multiple parallel processing cores, proving that the relative effect of each transformation is independent on the number of processors and the applied memory architecture.

Keywords: Power consumption; embedded processors; data-reuse; code transformations; multimedia.

1. Introduction

Most embedded applications in the multimedia and the telecommunication domain turn out to be data-dominated with the data-related power consumption affecting heavily the total power budget.[1–4] At the same time an increasing number of real time applications such as image and video processing is being available on portable devices. Low-power consumption is of primary importance for such systems because of the requirements for long battery life and large integration scales, and the related cooling and reliability issues.[5] Therefore, design for low power, especially at the system level where the most significant savings can be obtained, has become a major concern.[3]

For multimedia applications two general implementation choices exist: The first is to use dedicated hardware (nonprogrammable or partially-programmable platforms), which offers maximum performance since such systems are tailored to each targeted application. However, this solution comes at high cost while it completely lacks flexibility. The alternative is to use embedded instruction set processor cores (fully-programmable platforms), which offer increased flexibility, smaller time-to-market and opportunities for reuse at the cost of lower performance than the previous solution. In both cases the intense requirement for high performance imposes the partitioning of the initial algorithm into parallel threads and its assignment to multiple processing elements executing concurrently.

Catthoor *et al.*[1] suggested that a number of code transformations could be applied to any algorithm aiming at a memory hierarchy where copies of data from memories of larger size that exhibit high reuse are stored to additional memory layers of smaller size. In this way, exploiting the temporal locality of data memory references,[6] the greater part of the accesses is moved to small-sized on-chip memories. Since smaller memory size means less energy dissipation, significant power savings can be obtained.[1,7]

Data storage and transfers[1] dominate the total power consumption in the case of application-specific hardware since no instruction memory exists. A formalized methodology for data-reuse exploration in order to reduce the power consumption of data-intensive applications has been proposed in Ref. 6, where a systematic way on how to decide on the optimal memory hierarchy is developed. However, instruction related to power consumption is not taken into account since only the power component due to data accesses and transfers is considered. In Ref. 8 it was illustrated that when multimedia applications are implemented on embedded programmable processors, the power component due to instruction memory accesses becomes dominant and cannot be neglected during design exploration. In this work, however, only uni-processor systems are examined, without handling the case of partitioned algorithms. The combination of partitioning and techniques for reducing the power consumed on the memory hierarchy has been proposed in Refs. 1, 8–10. However, the efficacy of the applied methodology has only been proven for custom hardware architectures[1] and for commercially available processors[9] without dealing with multiple embedded processor cores. A partitioning approach for improving the memory utilization of algorithms coded in Weak Single Assignment Code has been presented in Ref. 10. Experimental results on power, area and performance from the application of a data-reuse methodology and the development of a custom memory hierarchy for the case of multiprocessor embedded architectures have been given in Ref. 11, while the effect of cache on the instruction memory power consumption has been discussed in Ref. 12.

All previous works adopt a simulative approach in order to determine the optimal solution from a pool of possible implementations, which is the end result of the data-reuse exploration and decision methodology.[6] Each alternative has to be implemented, compiled and executed on a processor simulator in order to evaluate its power efficiency. Unfortunately, since the design space can be extremely large[6] and this problem is further complicated in the case of varying algorithmic parameters, the previous approaches for this task can be very time and effort consuming. Moreover, this process can hardly be automated prohibiting the development of appropriate EDA tools that could be used early in the design process.

In this paper, an analytical approach is proposed for the fast and reliable determination of the optimal solution in terms of power, performance and area. During data-reuse exploration, power efficient code transformations are examined using as demonstrator application the three-step logarithmic search motion estimation algorithm. This algorithm is widely used in all major video encoding standards such as MPEG-X and H.26X. A specific memory hierarchy is developed according to Ref. 6 in order to exploit the presence of highly reused data sets in each transformation. Analytical expressions for the number of accesses to each memory layer are extracted, enabling the fast estimation of the optimal solution. In the same way, the number of executed assembly instructions is also analytically expressed as function of the algorithmic parameters. For the case of multi-processor systems the exploration methodology is applied on three memory architectures: distributed, shared and shared–distributed memory hierarchies. The proposed methodology offers the possibility to explore a pool of possible design decisions and to determine in short times the optimum solution, considering the trade-off between power, performance and area.

The rest of the paper is organized as follows: In Sec. 2 the data-reuse methodology and its application to motion estimation algorithms is described. The proposed definition of key transformations and analytical techniques for the calculation of memory related power consumption is discussed in Sec. 3, distinguishing between data and instruction memory accesses. The methodology is extended for the case of parallel processing units in Sec. 4. Finally we conclude in Sec. 5.

2. Data-Reuse Transformations: Theory

The target architecture (Fig. 1) is based on embedded processing units, each one communicating with one or more data memory layers and optionally with its own instruction memory, depending on whether the system is programmable or not. Instruction memories are considered to be on-chip single-port ROMs. The size of this memory is determined by the code size, which in turn depends on the applied transformation to the original code.[9] The data memory hierarchy may consist of several memory blocks communicating with the processor over a global bus. Memory blocks are considered to reside on chip except for the initial memory layer that holds the previous and the current frame, which is an off-chip memory.

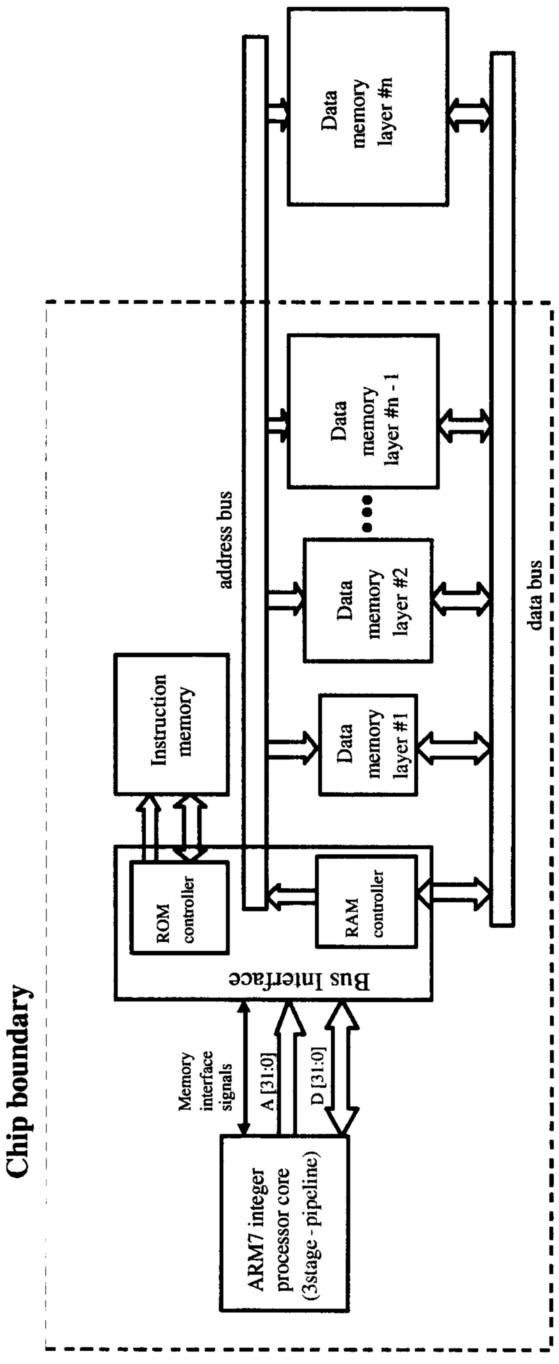


Fig. 1. Target architecture.

In the proposed approach only the power due to accesses to foreground and background memories is taken into account since the power due to accesses to register files is significantly smaller. According to the power model that has been used, the power consumed due to accesses in the ith memory layer, is directly proportional to the number of accesses, f_i, and depends on the size, S_i, and the number of ports, Nr_ports_i, of the memory, the power supply and the technology. For a given technology and power supply voltage the consumed energy can be expressed as:

$$E_i = f_i \cdot E_{p/a}\left(S_i, Nr\_ports_i\right), \tag{1}$$

where $E_{p/a}$ is the energy cost per access and is a product of the capacitance (C) that is being switched in a memory module and the supply voltage, V_{DD} ($E_{p/a} = C \cdot V_{DD}^2$). According to Ref. 13, the aggregate capacitance is a polynomial of the number of bits N and the number of words W in the memory array, of the form $C = c_0 + c_1 \cdot W + c_2 \cdot N + c_3 \cdot WN$, where the coefficients account for the technology and number of ports. For on chip memories, the relation between memory power and memory size is between linear and logarithmic.[6] The model that has been used in this study employs a modified version of the Landman model.[13]

In data-dominated applications such as multimedia algorithms, significant power savings can be achieved by developing a custom memory organization that exploits the temporal locality in memory accesses.[1,2,7,14] According to the proposed methodology, data sets that are often being accessed in a short period of time are identified and placed into levels of the memory hierarchy implemented by memory chips of smaller size. Since memories of smaller size have a lower energy cost per access, the total power consumption tends to be reduced. On the other hand the total number of accesses to memory elements is increased since additional accesses are required in order to move data from the background to foreground memories, introducing a trade-off between additional memory transfers and savings due to lower memory sizes. The power effectiveness of the proposed approach depends on the ratio of the number of read operations from a copy of a data set in a memory of small size over the number of read operations from this data set in the memory of larger size on the next hierarchical level.[6] In general, the impact of Data Transfer and Storage Exploration is larger for applications that exhibit high data-reuse, while such a methodology would be less useful in less regular algorithms.

This data-reuse exploration is performed by applying a number of code transformations to the original code, which are determined by the group of data sets that are being used in the algorithm and is different from the conventional approach where hardware cache control determines the size and timing of data copies.[3,14] In the case of motion estimation algorithms, the possible data-reuse transformations together with the introduced levels in the memory hierarchy, are shown in Fig. 2. These transformations are extracted according to the methodology described in Ref. 6. The parameters for these algorithms are: the size of the current and previous frame ($N \times M$), block size ($B \times B$) and p which determines the search region $[-p, p]$ around

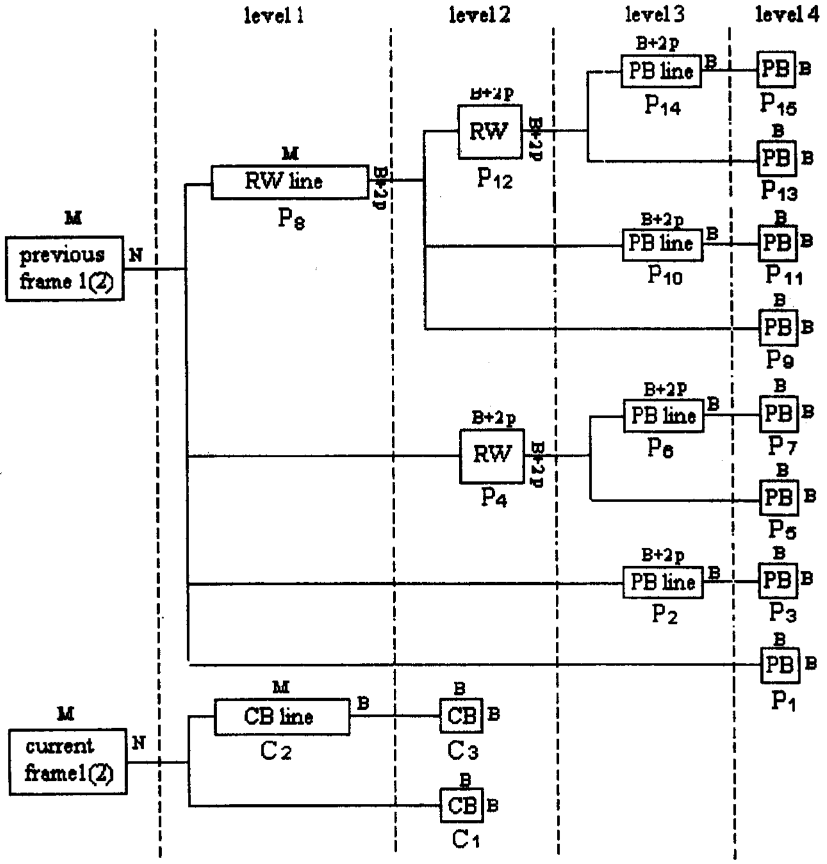


Fig. 2. Copy tree for a motion estimation kernel.

the location of a specific block in the current frame. These transformations involve memories for a line of reference windows (RW line) of size $(B+2p) \times M$, a reference window (RW) of size $(B+2p) \times (B+2p)$, a line of candidate blocks (PB line) of size $(B+2p) \times B$, a candidate block (PB) of size $B \times B$, a line of current blocks (CB line) of size $B \times M$ and a current block (CB) of size $B \times B$. Each rectangle in the figure is annotated by the number of the corresponding transformation and the size of the introduced memory, given parametrically. Capital letters C, P indicate current and previous frame respectively, in which the transformation takes place.

To illustrate the advantage that can be gained by the application of the proposed transformations, a typical motion estimation algorithm will be used as test vehicle, namely the two-dimensional logarithmic search which aims at reducing the computational complexity of the typical full-search algorithm by employing a heuristic search strategy for motion estimation similar to binary search. Both algorithmic kernels are shown in Fig. 3. For the calculation of the motion vector we use the mean absolute error (MAE) as a matching criterion.[15] For each code, three basic

Full Search

```
for(x=0;x<N/B;x++) /*For all blocks in the current frame*/
 for(y=0;y<M/B;y++)

   for(i=-p;i<p+1;i++)   /* For all candidate blocks */
    for(j=-p;j<p+1;j++)

     for(k=0;k<B;k++) /* For all pixels in the block */
      for(l=0;l<B;l++)
      {
        read pixel in current frame;
       if (current pixel displaced by i, j) lies outside frame
         previous pixel = 0;
       else
         read pixel from previous frame;
      }
```

Log Search

```
for(x=0;x<N/B;x++) /*For all blocks in the current frame*/
 for(y=0;y<M/B;y++)
 {
  d=4;
  while(d>0)
  {
    for(i=-d;i<d+1;i+=d)     /* For all candidate blocks */
     for(j=-d;j<d+1;j+=d)
     {
      for(k=0;k<B;k++)       /*For all pixels in the block */
       for(l=0;l<B;l++)
       {
        read pixel in current frame;
        if(current pixel displaced by i, j) lies outside frame
          previous pixel = 0;
        else
           read pixel from previous frame;
       }
     }
     d=d/2;
  } }
```

Fig. 3. Full search and three step logarithmic search algorithm kernels.

double nested loops implement the main part of the algorithm: The outer double loop selects all blocks in the current frame, the intermediate loop corresponds to the displacement in both dimensions according to which a reference window is selected and the most inner loop is used for the selection of all pixels in the block under study.

To illustrate how the proposed transformations are applied, the introduction of a line of reference windows, which corresponds to transformation P_8, is explicitly shown as pseudocode in Fig. 4, for both algorithms (for clarity, a nonoptimized code is shown, since all data in the introduced array are updated without exploiting the fact that some data already exist. However, in the rest of the paper codes are optimized since all assumptions proposed in Ref. 6 are adopted).

The introduced double loop between the loop of x and y indicated by bold fonts, performs a transfer of all pixels that belong to a line of reference windows from the previous frame array to the introduced array previous_line, except for pixels that lie outside the frame and thus obtain the value zero. The addition of an independent entity such as the array previous_line suggests the introduction of an additional memory layer holding the array data. In a similar way, all code transformations determine the additional memory layers that have to be used, according to data transfers between one data array and another.

3. Data-Reuse Transformations: Analysis

The application of the existing methodology[1] for the Data Transfer and Storage Exploration (DTSE) step, including its extension for the case of embedded processor cores as proposed by Refs. 8 and 11 requires the evaluation of each code transformation by executing the corresponding code on a simulator, such as the ARMulator.[16] This process can be extremely time consuming, leading to prohibitive requirements in terms of time and effort, especially for the cases when design parameters vary. As an example, the power exploration of the full search algorithm requires the trace of executed assembly instructions in order to calculate the number of accesses to the instruction memory. The process of generating the trace file takes approximately four hours on a Pentium III 500 MHz processor.

In contrast to previous approaches, an optimized methodology for the analytical calculation of the total number of accesses to each memory layer and the number of executed instructions is introduced. In this way it is possible, without having the transformed code simulated, to evaluate the power consumption for each transformation at very short times. One of the main contributions of the proposed approach is that it will be proved that it is not necessary to examine all possible transformations in order to evaluate their power efficiency. Rather, only a few key transformations have to be studied in order to extract the required information. The main advantage of the proposed methodology over previous techniques is the speed of exploration: the evaluation of analytical expressions takes time in the order of a few milliseconds.

Full Search | **Log Search**

Introduction of a line buffer of reference windows for the previous frame (indicated bold)

Full Search

```
for(x=0;x<N/B;x++) /* For all blocks in a line of blocks */

 for(i=0;i<B+2p;i++)/*For a line of reference window*/
  for(j=0;j<M;j++);
  {
   if (current pixel displaced by i) lies outside frame
     previous_line[i][j] = 0;
   else
     read previous_line from previous frame;
  }
 for(y=0;y<M/B;y++)

   for(i=-p;i<p+1;i++)        /* For all candidate blocks */
    for(j=-p;j<p+1;j++)

     for(k=0;k<B;k++)       /*For all pixels in the block */
      for(l=0;l<B;l++)
      {
       read pixel in current frame;
       if (current pixel displaced by j) lies outside frame
         previous pixel = 0;
       else
         read pixel from previous_line;
      }
```

Log Search

```
for(x=0;x<N/B;x++) /*For all blocks in a line of blocks*/

 for(i=0;i<B+2p;i++)/*For a line of reference window/
    for(j=0;j<M;j++);
    {
     if (current pixel displaced by i) lies outside frame
        previous_line[i][j] = 0;
     else
        read previous_line from previous frame;
    }
 for(y=0;y<M/B;y++)
 {
  d=4;
  while(d>0)
  {
   for(i=-d;i<d+1;i+=d)     /*For all candidate blocks*/
    for(j=-d;j<d+1;j+=d)
    {
     for(k=0;k<B;k++)  /*For all pixels in the block*/
      for(l=0;l<B;l++)
      {
      read pixel in current frame;
      if (current pixel displaced by j) lies outside frame
          previous pixel = 0;
      else
          read pixel from previous_line;
      }
    }
    d=d/2;
```

Fig. 4. Transformed codes (transformation P_8) for the full search and three step logarithmic search.

The proposed methodology for speeding up the design exploration can be applied once the set of possible data-reuse transformations have been determined according to Ref. 6. The applied methodology can be viewed as a two-phase process. The first phase corresponds to the definition of a minimum set of key transformations from which the required information for the number of data and instruction memory accesses for all transformations can be extracted. In the second phase the number of data and instruction memory accesses is analytically calculated. These numbers are fed to the power model that is used for the calculation of the memory-related power consumption.

3.1. *Definition of key transformations*

Data Memory

The number of data accesses to each memory layer is the sum of the accesses, which are made in order to update this memory from its previous memory layer, and the accesses, which are made in order to update the next memory layer. This can be summarized in the following equation:

$$DA_{i,n} = a_{i,p} + a_{i,n} \, , \tag{2}$$

where $(p < i \leq n)$:

- $DA_{i,n}$: the total number of accesses to data memory layer i when followed by memory layer n.
- $a_{i,p}$: the number of accesses to memory layer i to update its contents from a previous layer p.
- $a_{i,n}$: the number of accesses to memory layer i to update the contents of a next layer n.

In case $i = n$, the accesses correspond to the final read in order to process the data of memory layer i.

However, for each $p < i$, $a_{i,p} = a_i$. This means that the number of accesses, which are made in order to update a memory layer from a previous one, is independent of the previous layer from which data are read.

Consequently, $DA_{i,n} = a_i + a_{i,n}$. Therefore, the number of accesses to a memory layer depends only on the following memory layer. As an example, transformations P_{14} and P_6 have the same number of accesses to memory RW (Fig. 2).

According to the above, in order to calculate the number of accesses for each data transfer between memory layers, a table like the one in Table 1, has to be built. The dimension of the table is $(d+1) \times (d+1)$ where d are the memory levels of the applied hierarchy. The columns correspond to the memory layer, which is being accessed while the rows correspond to the memory layer that follows. The contents of the cells provide the total number of accesses for the memory layer, which is indicated by the column, when it is followed by the memory layer that is indicated in the corresponding line of the table.

Table 1. Number of accesses to each memory layer according to the layer which follows ($a = \lceil \log_2 p \rceil$).

Next Layer \ Accessed Layer	**0 (Previous)**	**1 (RW_line)**	**2 (RW)**	**3 (PB_line)**	**4 (PB)**
0 (Previous)	$N \cdot M \cdot a^3$				
1 (RW_line)	$M(B+2p)+(N/B-1)M(B+1)-2pM$	$(N/B)M(B+2p)+NMa^3+(N/B-1)M(2p+1)$			
2 (RW)	$(N/B)(B+2p)^2-(N/B)(B+2p)p-2p(B+p)+(M/B)(N/B-1)4Bp$	$(N/B)M(B+2p)+(N/B-1)M(2p+1)+(N/B)(B+2p)(B+1)+N(M/B-1)(B+2p)$	$(N/B)(B+2p)^2+(N/B)(M/B-1)(B+2p)(B+4p)+NMa^3$		
3 (PB_line)	$(N/B)(M/B)(B+2p)(3B+2p)$	$M(B+2p)+(N/B-1)M(B+1+4p)+(N/B)(M/B)(B+2p)3B+(N/B)(M/B)(B+2p)2p$	$(N/B)(B+2p)^2+(N/B)(M/B-1)(B+2p)(B+4p)+(N/B)(M/B)(B+2p)(3B+2p)$	$(N/B)(M/B)(B+2p)[3B+2((p-1)B-p)]+NMa^3$	
4 (PB)	$(N/B)(M/B)(9B^2+6Bp-41)-28N$	$M(B+2p)+(N/B-1)M(B+1+4p)+(N/B)(M/B)(9B^2+6Bp-2B-18)$	$(N/B)(B+2p)^2+(N/B)(M/B-1)(B+2p)(B+4p)+(N/B)(M/B)(9B^2+6Bp)$	$(N/B)(M/B)(24B^2+34Bp-4p^2)$	$(N/B)(M/B)3B(15B-2p)+NMa^3$

To calculate the total number of data accesses for each transformation, all involved memory layers have to be defined and for each memory layer its subsequent one has to be determined, to find the entry in the table that contains the corresponding number of accesses. For example, the accesses to data memory layers for transformation P_{15} are given in the shaded cells of Table 1.

In order to extract the expressions for the number of data accesses for all transformations, the designer does not have to go through all possible code transformations. Rather, only the memory accesses associated with each data transfer between memory layers have to be calculated. As an example, for the case of a complete memory exploration space (an exploration space for which all possible memory configurations are considered) of depth d, the number of all possible data transfers is:

$$\#poss\_data\_transfers = \binom{d+1}{2} + (d+1) = \frac{(d+1)!}{2\cdot(d-1)!} + (d+1)\,. \tag{3}$$

The first term corresponds to the number of possible data transfers between memories residing in different levels and is equal to the number of memory pair combinations. The second term corresponds to the number of possible data transfers between the processor and each one of the memory layers (including the initial one). The total indicates how many expressions have to be filled in the table containing the number of accesses.

Considering that the number of possible transformations for a complete memory hierarchy are:

$$\#poss\_transformations = \sum_{i=0}^{d-1} 2^i\,, \tag{4}$$

we define as "key" transformations the minimum set of transformations, which are required in order to fill the table with the necessary information. For a memory hierarchy of depth 4, all possible transformations are 15 while the key transformations are only 8.

Instruction Memory

Considering two memory layers $A(i)$ and $B(j)$ placed in memory hierarchy levels i and j respectively ($i < j$), the code that performs the update of memory $B(j)$ from its previous one $A(i)$, is identical in any transformation that includes this memory sequence, independently on other memory layers that might exist. For example, the code that describes the update of PB from PB line is identical in transformations P_{15}, P_{11}, P_7 and P_3 (Fig. 2). Consequently, the number of accesses to the instruction memory is also the same and is a function of the memory sizes and the position of memory layer j in the memory hierarchy. In the proposed memory hierarchy (Fig. 2) for a motion estimation kernel (e.g., full search) each memory level corresponds to a loop of the program. The update of a memory layer in level 1 takes place between the two outer loops of the kernel (loops 1 and 2). In the same way, the update of a memory in level 2, takes place between loops 2 and 3. Consequently the layer in level j is updated within loop j (before the scope of loop $j+1$). It follows that the number of accesses to the instruction memory, according to the execution of the code that describes the update of a memory layer j, is equal to the corresponding assembly instructions multiplied by the number of iterations of all loops in which it takes place. This can be summarized in the following expression:

$$IA_{ij} = \varepsilon_{ij} \cdot \prod_{k=1}^{k<j} n_k \,, \tag{5}$$

where n_k is the number of iterations of kth loop, ε_{ij} the number of assembly instructions in order to update the memory layer j from its previous layer i and IA_{ij} the number of accesses to the instruction memory.

Consequently, the number of accesses to the instruction memory in order to update the contents of a data memory from its previous one, is independent on other memory layers in the hierarchy. To extract the necessary information for the calculation of instruction memory accesses, a set of key transformations can also be found, which are the same as in the case of data memory. In this way, a reference table similar to that constructed for data accesses is built (Table 2), to calculate the number of executed instructions for each data transfer between memory layers. The contents of the cells provide the total number of accesses to the instruction memory required for updating the memory layer that is indicated by the row, from the memory layer that is indicated by the corresponding column.

3.2. *Analytical calculation of memory accesses*

Data Memory

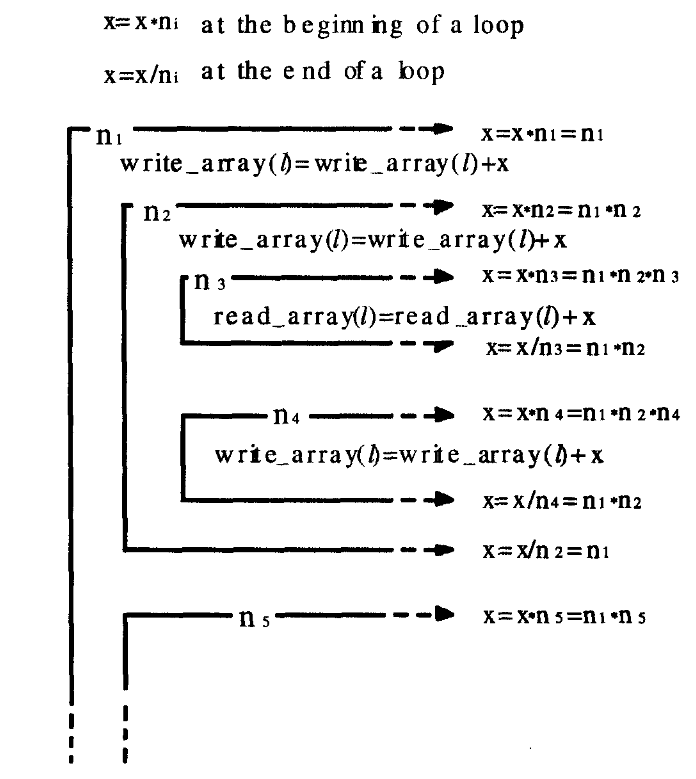
Usually motion estimation algorithms are constructed upon loops (nested and independent). According to the proposed methodology, analytical expressions for the number of accesses to each memory layer are extracted using the loop hierarchy and without the need to compile or simulate the corresponding code.

We assume a general structure of loops like the one in Fig. 5 where n_i is the number of iterations of i loop and $array(l)$ is the array (memory) for which the data accesses have to be obtained. By serially parsing the code description, it is detected whether a loop is started or finished and one variable keeps track of the number of times data are being read or written from/to the array within a loop. Variables *read_array*(l) and *write_array*(l) are updated each time a read or write operation is encountered taking into account the depth of the corresponding loop. Two separate variables for reads and writes are used since in the general case a power model assumes different values for energy per access for each case.

As an example, let us consider the introduction of a line buffer of reference windows (*previous_line*) to the Full Search algorithm, indicated by bold fonts in

Table 2. Number of assembly instructions for accesses between memory layers.

Mem.Layer / Next.Layer	(Previous)	(RW_line)	2 (RW)	3 (PB_line)	4 (PB)
0 (Previous)					
1 (RW_line)	$f[(2p+B),M,\varepsilon]+\left(\frac{N}{B}-1\right)\cdot\begin{bmatrix}f[2p,M,\varepsilon]+\\ f[(B+1),M,\varepsilon]\end{bmatrix}$	$f\left[\frac{N}{B},\frac{M}{B},f[(2p+1),(2p+1),f[B,B,\varepsilon]]\right]$			
2 (RW)	$\frac{N}{B}\cdot\begin{pmatrix}f[(2p+B),(2p+B),\varepsilon]+\\ \left(\frac{M}{B}-1\right)\cdot\begin{bmatrix}f[(2p+B),2p,\varepsilon]+\\ f[(2p+B),B,\varepsilon]\end{bmatrix}\end{pmatrix}$	$\frac{N}{B}\cdot\begin{pmatrix}f[(2p+B),(2p+B),\varepsilon]+\\ \left(\frac{M}{B}-1\right)\cdot\begin{bmatrix}f[(2p+B),2p,\varepsilon]+\\ f[(2p+B),B,\varepsilon]\end{bmatrix}\end{pmatrix}$	$f\left[\frac{N}{B},\frac{M}{B},f[(2p+1),(2p+1),f[B,B,\varepsilon]]\right]$		
3 (PB_line)	$\frac{N}{B}\cdot\frac{M}{B}\cdot\begin{pmatrix}f[B,(B+2p),\varepsilon]+\\ 2p\cdot\begin{bmatrix}f[(B-1),(B+2p),\varepsilon]+\\ f[1,(B+2p),\varepsilon]\end{bmatrix}\end{pmatrix}$	$\frac{N}{B}\cdot\frac{M}{B}\cdot\begin{pmatrix}f[B,(B+2p),\varepsilon]+\\ 2p\cdot\begin{bmatrix}f[(B-1),(B+2p),\varepsilon]+\\ f[1,(B+2p),\varepsilon]\end{bmatrix}\end{pmatrix}$	$\frac{N}{B}\cdot\frac{M}{B}\cdot\begin{pmatrix}f[B,(B+2p),\varepsilon]+\\ 2p\cdot\begin{bmatrix}f[(B-1),(B+2p),\varepsilon]+\\ f[1,(B+2p),\varepsilon]\end{bmatrix}\end{pmatrix}$	$f\left[\frac{N}{B},\frac{M}{B},f[(2p+1),(2p+1),f[B,B,\varepsilon]]\right]$	
4 (PB)	$\frac{N}{B}\cdot\frac{M}{B}\cdot(2p+1)\begin{pmatrix}f[B,B,\varepsilon]+\\ 2p\cdot\begin{bmatrix}f[B,(B-1),\varepsilon]+\\ f[B,1,\varepsilon]\end{bmatrix}\end{pmatrix}$	$\frac{N}{B}\cdot\frac{M}{B}\cdot(2p+1)\begin{pmatrix}f[B,B,\varepsilon]+\\ 2p\cdot\begin{bmatrix}f[B,(B-1),\varepsilon]+\\ f[B,1,\varepsilon]\end{bmatrix}\end{pmatrix}$	$\frac{N}{B}\cdot\frac{M}{B}\cdot(2p-1)\begin{pmatrix}f[B,B,\varepsilon]+\\ 2p\cdot\begin{bmatrix}f[B,(B-1),\varepsilon]+\\ f[B,1,\varepsilon]\end{bmatrix}\end{pmatrix}$	$\frac{N}{B}\cdot\frac{M}{B}\cdot(2p+1)\begin{pmatrix}f[B,B,\varepsilon]+\\ 2p\cdot\begin{bmatrix}f[B,(B-1),\varepsilon]+\\ f[B,1,\varepsilon]\end{bmatrix}\end{pmatrix}$	$f\left[\frac{N}{B},\frac{M}{B},f[(2p+1),(2p+1),f[B,B,\varepsilon]]\right]$


x=x*ni at the beginning of a loop
x=x/ni at the end of a loop
n1
x=x*n1=n1
write_array(l)=write_array(l)+x
n2
x=x*n2=n1*n2
write_array(l)=write_array(l)+x
n3
x=x*n3=n1*n2*n3
read_array(l)=read_array(l)+x
x=x/n3=n1*n2
n4
x=x*n4=n1*n2*n4
write_array(l)=write_array(l)+x
x=x/n4=n1*n2
x=x/n2=n1
n5
x=x*n5=n1*n5

total number of write accesses in array(l) = n1+n1*n2+n1*n2*n4
total number of read accesses in array(l) = n1*n2*n3

Fig. 5. General loop structure.

Fig. 4. The aim is to find the expression for the number of data memory accesses to the *previous_line* array. By parsing the code from the beginning, the variable that keeps track of the number of iterations is multiplied by N/B, when encountering the first loop (N/B is the total number of its iterations). By multiplying the same variable with the number of iterations of all loops that are being initiated, the two statements at which data is written on to the *previous_line* array, are performed for $N/B \cdot (B+2p) \cdot M$ times (which becomes the value of the corresponding *write_array* variable). This expression corresponds to one part of the total number of accesses to this particular memory, since accesses are not only made to update its contents from a previous memory, but also or to process its contents by the processor, altering the *read_array* variable.

According to the above, it is possible to extract analytical expressions for the number of data accesses to memory layers and to fill in the required information in the table of memory layers as shown in Table 1. It should be mentioned that in the case of conditional accesses to a memory layer, the number of accesses to an array should be equal to the number of times the corresponding conditions are fulfilled.

The proposed analytical expressions have been validated by comparisons to simulation results, using counters for the calculation of the total number of accesses to the introduced data memory layers. The analytical calculations are error free leading to an accuracy of 100% in all cases. This is because the expressions are not subject to the image content or any statistical parameters but depend only on the number of loop iterations and the corresponding accesses to pixels, whose number can be accurately evaluated.

In this way it is possible to feed the total number of accesses on each memory to the power model in order to evaluate the total power consumption. Consequently, the most power efficient solution from a pool of possible alternatives can be determined very fast without having to execute each code on a simulator in order to count the number of accesses.

In Fig. 6 the total energy consumption due to accesses to data memory layers is presented for all transformations and compared to that corresponding to the original code, for the three step logarithmic search algorithm. Since transformations on the previous and the current frame can be concurrently applied, two combinations of code transformations (P_{14} and C_1, P_4 and C_1) have also been examined. As expected, the power reduction becomes even larger when transformations on both frames are applied.

In the case of the logarithmic search, the most power efficient transformation for the presented case ($M \times N = 144 \times 176, B = 16, p = 7$) is transformation P_4 for the previous frame, while C_1 is the best transformation for the current frame. Transformation C_1 always yields better results than the other two, since current blocks have no overlap and thus no advantage of a line of current blocks can be made.

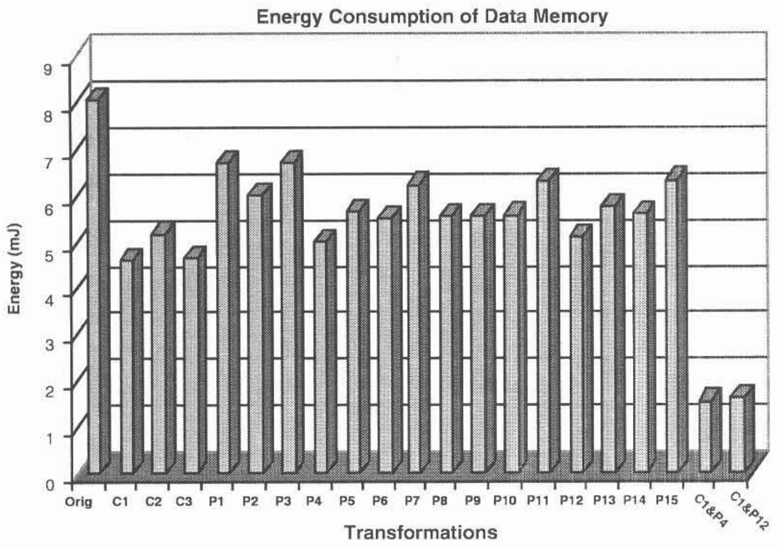


Fig. 6. Data memory energy consumption.

Except for the fast calculation of the power consumption, the analytical expressions of Table 1 allow for the exploration of the whole design space by varying parameters such as the frame size (N, M), the size of the search space (p) and the block size (B). In Figs. 7 and 8 the energy consumption for three code transformations of the logarithmic search algorithm is presented for varying frame and block sizes. The possibility to evaluate the effectiveness of each transformation for varying algorithmic parameters is one of the key points in a complete design exploration and the determination of the best possible solution.

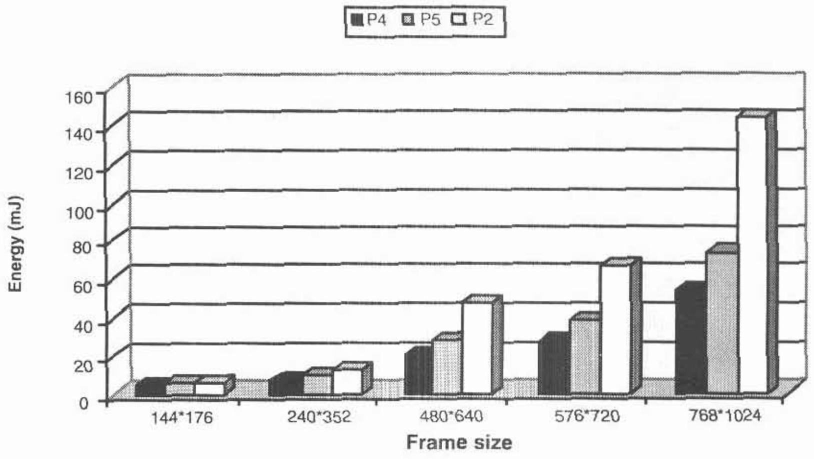


Fig. 7. Data memory energy consumption for three transformations/several frame sizes $(B = 16)$.

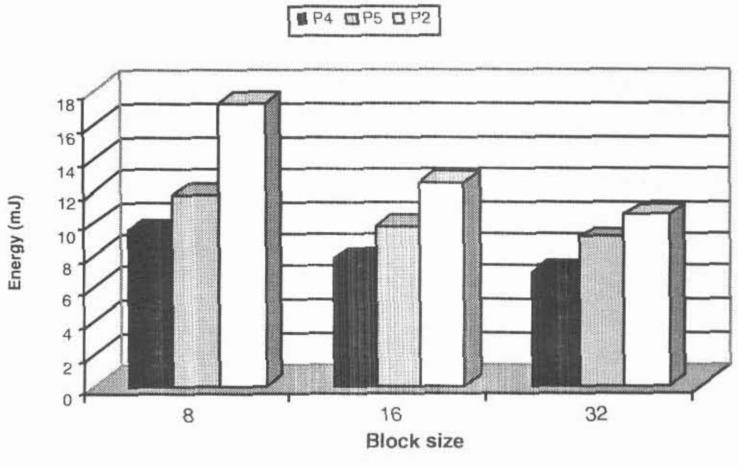


Fig. 8. Data memory energy consumption for three transformations/several block sizes $(M, N) = (240, 352)$.

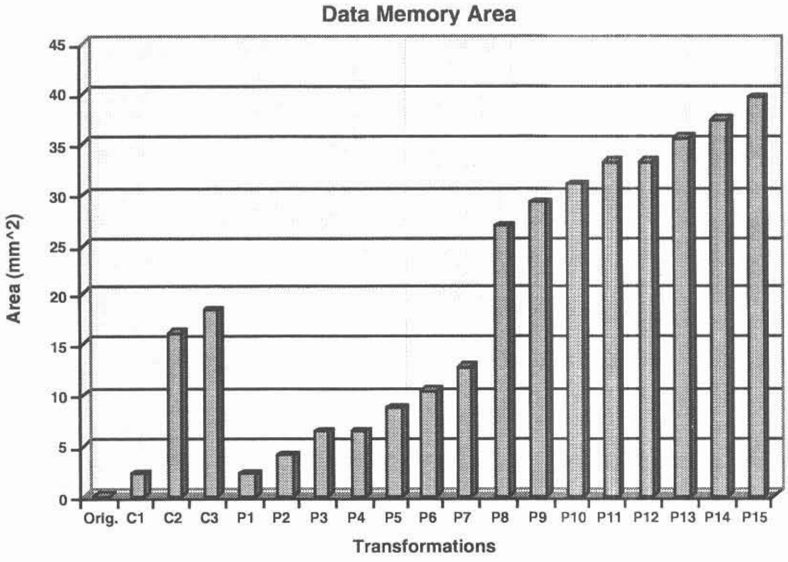


Fig. 9. Area occupied by data memory.

Since the introduction of additional memory layers comes with an area penalty, this parameter has also to be taken into account using appropriate area models.[17] In Fig. 9 the effect of the proposed code transformations on area for the three-step logarithmic search is illustrated.

Instruction Memory

The total number of executed instructions is also calculated parametrically, similar to data accesses. This is in contrast to previous works, which adopt a simulative approach requiring the compilation and execution of each code transformation on a processor simulator in order to obtain the total number of executed instructions. The number of assembly instructions is obtained from the number of iterations of the nested loops that implement each of the applied motion estimation algorithms. In its general form, each double nested loop containing ε instructions of the form:

```
for(i = 0; i < $n_0$; i++)
    for(j = 0; j < $n_1$; j++)
    {
        #$\varepsilon$ instructions
    }
```

corresponds to:

$$\#instr. = c_1 + c_2 \cdot n_0 + n_0 \left[c_1 + (c_2 + \varepsilon) \cdot n_1\right] = f(n_0, n_1, \varepsilon) \tag{6}$$

executed assembly instructions. The reason for selecting a double loop for the definition of function f is the two-dimensional nature of motion estimation algorithms.

Constants c_1 and c_2 are fixed for each loop, independent on the number of iterations and for the ARM processor $c_1 = 4$ and $c_2 = 5$, assuming a step of one. These constants correspond to the assembly instructions that implement the for loop. For large numbers of ε, function f diminishes to $f(n_0, n_1, \varepsilon) = n_0 \cdot n_1 \cdot \varepsilon$.

The number of ε instructions within the loop, depends on the branch conditions imposed by the *if* statements for deciding whether a pixel in the reference area lies outside the previous frame or not. However, the number of times each of the logical criteria is fulfilled, is explicitly known from the previous analysis on data and consequently the exact number of assembly instructions can be obtained. For the sake of simplicity the case of the full search algorithm is considered while the complete expressions are not shown in Table 2.

Starting from the most inner loop, the number of executed assembly instructions is calculated and the result is added to the number of instructions between nested loops (which in turn can be loops for introducing additional memory layers or single instructions). The final number of instructions is fed to the next outer loop until the total number of executed assembly instructions is obtained, resulting in a limited number of algebraic expressions. Since the indices of each loop are determined by the algorithmic parameters M, N, B and p, the total number of instructions is obtained as a polynomial function of these parameters. Consequently, the total number of accesses to the instruction memory, which is equal to the number of executed assembly instructions, can be efficiently evaluated (Table 2) leading to a very fast calculation of the instruction memory energy consumption.

As it can be observed, the expressions for the cases that correspond to the final read from the closer to the processor memory layer are the same, since the final read operation lies in the most inner loop of the algorithmic kernel. All other expressions have similar form since they consist of transfers from one memory layer to another for all columns/rows (except for the case when a pixel outside the previous frame is accessed, where no memory updating is performed and the previous pixel is read as zero). Each expression is multiplied with a parameter corresponding to the "depth" in which the additional memory layer is introduced (i.e., the loop in which it is placed). Considering the expressions in a row, it is observed that expressions are of the same form, since the same code is used for updating the memory layer that is indicated by the row from its previous one, independently on the previous memory layer. The position of the previous memory layer in the memory hierarchy affects only the addressing equations, i.e., the number of assembly instructions within the loops. The proposed expressions calculate accurately the absolute number of executed assembly instructions except for a small deviation which is due to high-level statements that are being compiled to different number of assembly instructions according to the processor state (i.e., number of available registers) and due to program parts whose execution depends on the image content (i.e., instructions related to the calculation of the minimum distance between blocks). The accuracy of the proposed approach has been validated by comparisons with simulation results obtained using the ARMulator.[16]

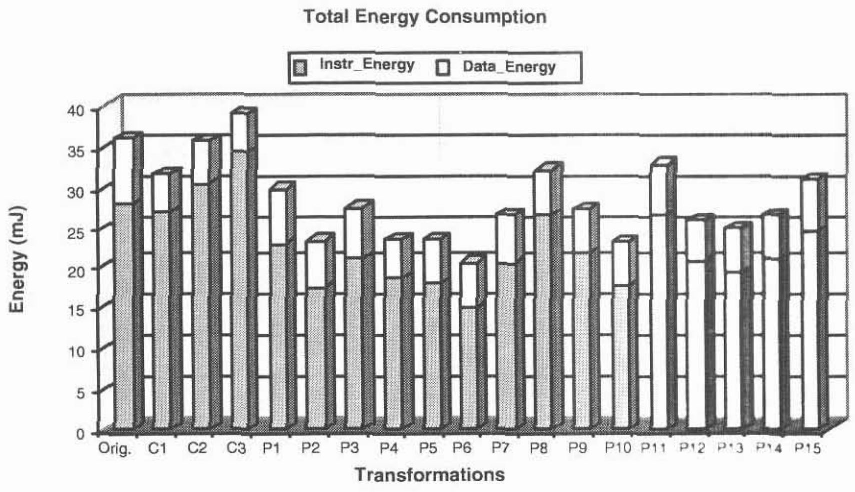


Fig. 10. Instruction memory energy consumption over total energy consumption.

In Fig. 10 the power consumption due to instruction memory accesses is shown as part of the total power consumption for the original and the transformed codes. As it can be observed, transformation P_4 *is no longer the most power efficient* transformation (minimum power consumption is achieved by transformation P_6). It becomes clear that in the presence of an instruction memory the number of accesses to the instruction memory as well as the instruction memory size should be efficiently evaluated in order to determine the best possible code transformation.

It should be mentioned that in the results shown in Fig. 10, the power consumption due to instruction memory accesses is overestimated. This is because no instruction caching was taken into account, which for data dominated applications (where cache misses do not occur frequently) would result in a smaller number of accesses to the instruction memory. The reason for not considering a cache is that the ARM 7 TDMI processor core does not have a cache memory.

It can be observed that for *all transformations the instruction-related* power consumption is reduced. Normally, since additional copies of data are introduced, the original code size and consequently the corresponding memory size should be increased leading to an increase of the instruction-related power. However, the number of executed assembly instructions is reduced due to simpler addressing and control logic in the most inner loop as a result of the transformations. In other words, since data accesses are moved to the introduced outer loops, the addressing equations in the most inner loop (which is executed for most iterations) and the corresponding conditional statements are significantly simplified resulting in smaller number of executed instructions. Moreover, even if the code size is increased, the memory size might not change since only fixed memory sizes are used. For example, for code sizes of 2.5 KB and 3 KB, the same memory size (4 KB) is used.

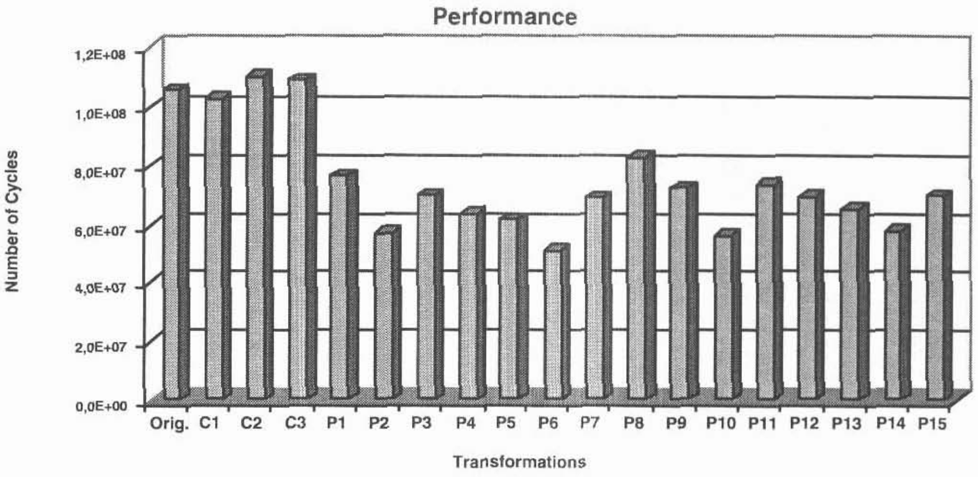


Fig. 11. Code performance for different transformations.

Obviously, code transformations affect the processor performance, i.e., the number of cycles required for the execution of the code. In Fig. 11 the effect of the proposed code transformations on performance is illustrated for the case of the three-step logarithmic search.

An overview of the proposed methodology for the evaluation of the best possible solution in terms of power (given the timing constraints) is shown in Fig. 12. In this flowchart it is emphasized that only key code-transformations are evaluated in terms of data and instruction-related power in order to find the most power efficient implementation of all possible solutions.

4. Results for Parallel Processors

The proposed exploration methodology can be equally applied to parallel processing units executing a motion estimation algorithm concurrently. To illustrate this case, a generalized architecture consisting of parallel processor cores is considered. Concerning the data memory organization an application-specific data memory architecture (ASDMA) is assumed.[1] The memory architecture consists of one or more levels which are determined by the applied transformation and which communicate with the processor by a common bus. Since the main focus is on parallel processing systems, the flexibility of using distributed or shared memory layers imposes the mapping of the transformed algorithm onto three different memory architectures (Fig. 13)[7,11]:

- distributed memory architecture (DMA),
- shared memory architecture (SMA),
- shared–distributed memory architecture (SDMA).

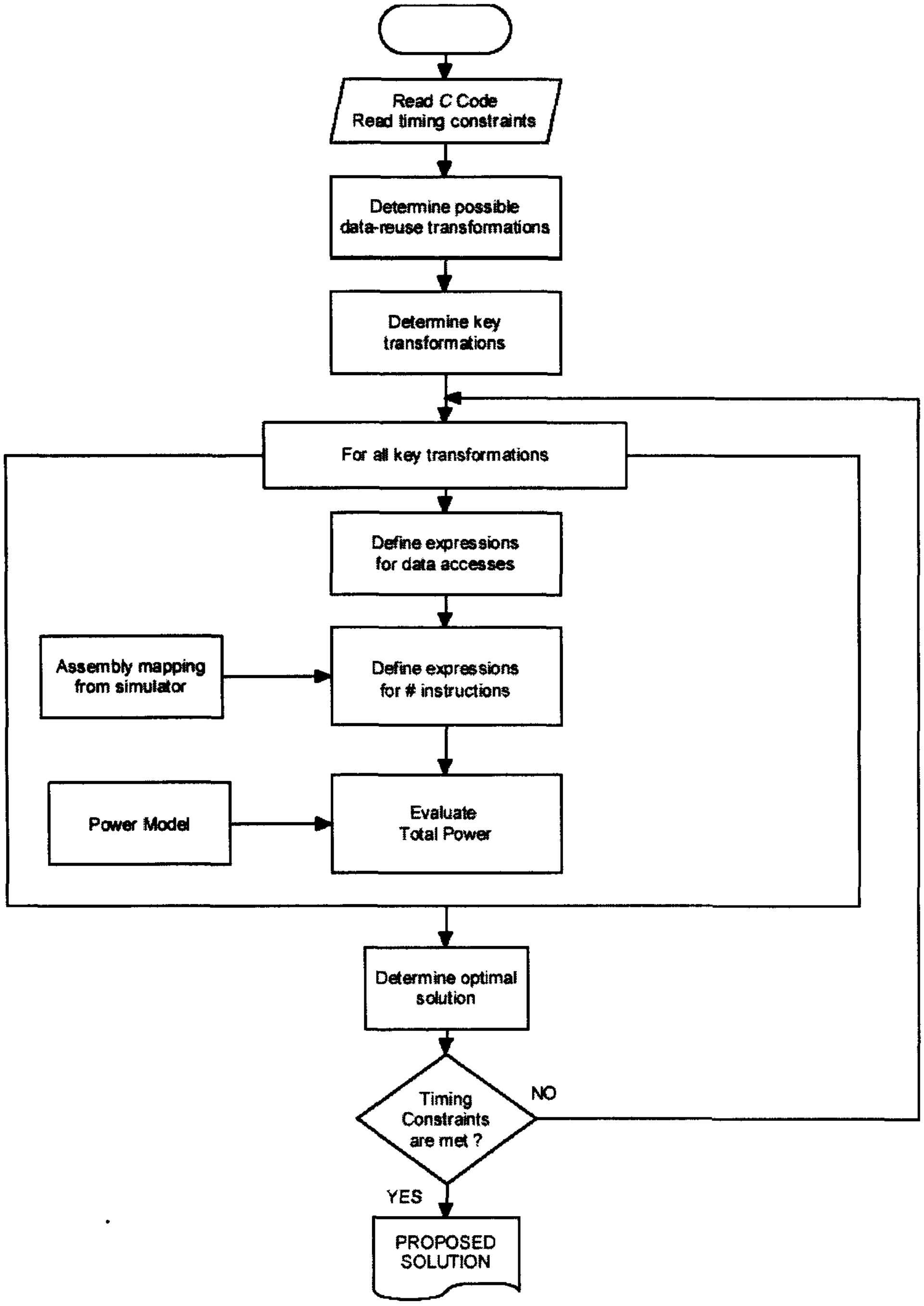

Read C Code
Read timing constraints
Determine possible data-reuse transformations
Determine key transformations
For all key transformations
Define expressions for data accesses
Assembly mapping from simulator
Define expressions for # instructions
Power Model
Evaluate Total Power
Determine optimal solution
Timing Constraints are met ?
NO
YES
PROPOSED SOLUTION

Fig. 12. Overview of the proposed methodology for evaluating the optimal solution.

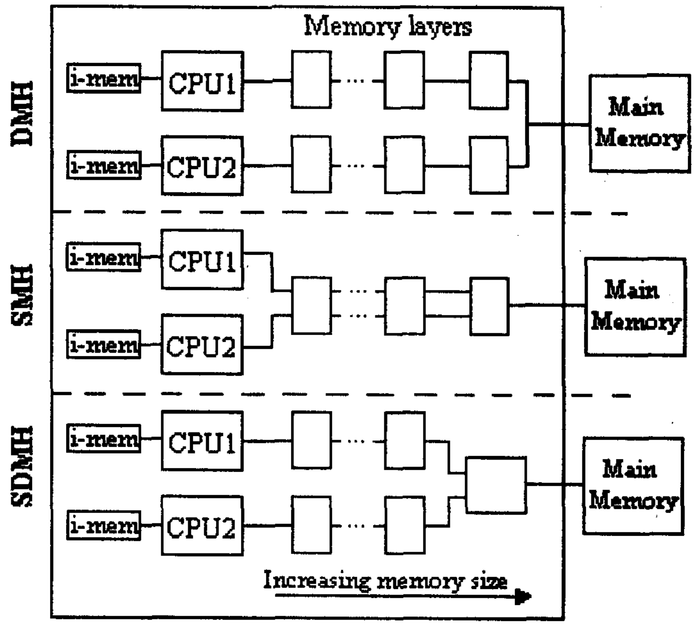


Fig. 13. Memory architectures for multi-processor systems.

For all data memory architectures a shared single port off-chip DRAM background memory module is considered, which in the case of motion estimation algorithms, usually holds the previous and/or the current frame. Every memory layer in these three architectures is of the same size as the corresponding layer of the single processor architecture. The distributed memory modules are considered to be single-port SRAMs, while the shared ones are dual-port SRAMs.

In the distributed memory architecture (DMA) separate memory blocks exist for each processor. The initial frame is partitioned into n slices (not necessarily equal) and each of the n processors executes the algorithm on the assigned block. Concerning the code of the full search and the three-step logarithmic search kernel (Fig. 3), when DMA is employed, only the outer double-loop is partitioned among the processors. In this way each processor handles on its own some of the blocks of the current frame. With shared memory architecture (SMA) all memory levels are common for the n processors. Since it is extremely difficult and performance-inefficient for data-dominated applications to schedule all memory accesses sequentially, it is assumed that the number of ports per memory block equals the number of processors that access this block. In this case, only the double inner loop is partitioned and as a consequence all processors are handling the same block and the workload is split on a per pixel basis. In the shared–distributed scheme (SDMA) the higher levels of the memory hierarchy are common while the lower levels are separate for each processor. It should be mentioned that with SDMA at least two memory levels are shared, since if only the off-chip memory is common, the whole scheme degenerates to the distributed memory architecture. Concerning the code in Fig. 3, partitioning is performed between the inner and outer double-loops, according to the introduced loops for reading data from each memory layer.

To illustrate the effect of data-reuse transformations, presented in Fig. 2, on the power consumption of the implementations of the three-step logarithmic search algorithm, a two-processor platform has been simulated using ARMulator in addition to the single processor. Typical values for the algorithmic parameters have been used[15]: $N \times M = 144 \times 176$, $B = 16$, $p = 7$. To avoid restricting our results by the timing characteristics of the ARM processor (e.g., clock period), instead of the power consumption we give results for the energy consumed for processing of a frame by the proposed architectures.

4.1. *Power*

Results of the energy consumption for the data memory accesses are given in Fig. 14. The total data-related energy consumption for a given memory architecture is the sum of the energy consumption of every memory layer included in that architecture:

$$E_{d\_total} = \sum_i f_i E_{p/a} \left(S_i, Nr\_ports_i\right) . \tag{7}$$

For the distributed architecture the energy consumption is

$$\begin{aligned} E_{d\_DMA} &= \sum_i \left[f_{1i} E_{p/a} \left(S_i, 1\right) + f_{2i} E_{p/a} \left(S_i, 1\right)\right] \\ &= \sum_i \left(f_{1i} + f_{2i}\right) \cdot E_{p/a} \left(S_i, 1\right) , \end{aligned} \tag{8}$$

where indices 1 and 2 denote the processors. According to Eq. (8) $(f_{1i} + f_{2i})$ is the number of total accesses of the two processors in ith memory layer. However, since memory sizes are equal for both processors and the sum of the accesses to

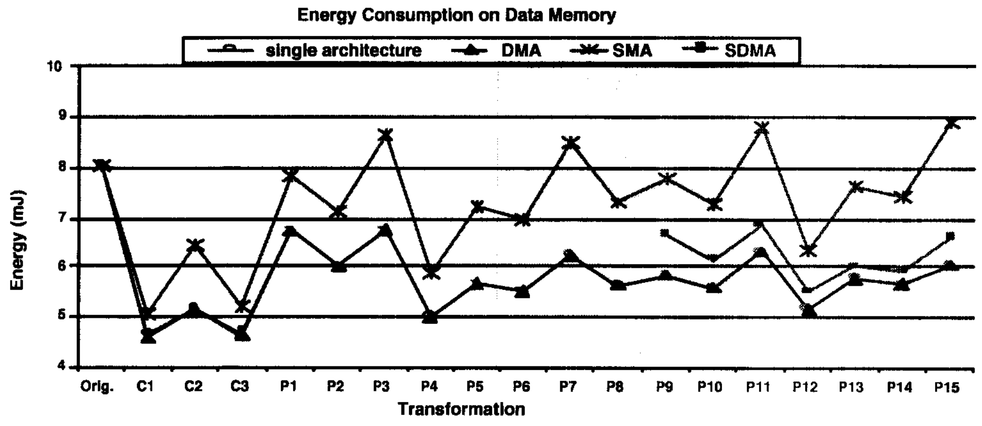


Fig. 14. Energy consumption on data memory.

both memories is equal to the number of accesses of the single processor to the corresponding memory, it holds

$$E_{d\_DMA} = \sum_i (f_{1i} + f_{2i}) E_{p/a}(S_i, 1) = \sum_i f_i E_{p/a}(S_i, 1) = E_{d\_single} \,. \tag{9}$$

This conclusion is validated in Fig. 14. It is clear that the energy consumption for the single processor is equal to that for the distributed one.

In the case of shared memory architecture, the sum of the accesses of the two processors to each memory is equal to the number of accesses of the single processor to that memory. Dual-port memories of the same size, which are more power consuming than a single-port memory, are used. Consequently, the data-related energy consumption is larger than that in the distributed scheme.

The energy consumption for SMA is given below:

$$\begin{aligned} E_{d\_SMA} &= \sum_i \left[f_{1i} E_{p/a}(S_i, 2) + f_{2i} E_{p/a}(S_i, 2)\right] = \sum_i (f_{1i} + f_{2i}) E_{p/a}(S_i, 2) \\ &= \sum_i f_i E_{p/a}(S_i, 2) \overset{[E_{p/a}(S_i,2) > E_{p/a}(S_i,1)]}{\Longrightarrow} E_{d\_SMA} > E_{d\_DMA} \,. \end{aligned} \tag{10}$$

In case of SDMA, the same as before holds for the accesses, while the energy consumption lies between the two other cases.

$$E_{d\_single} = E_{d\_DMA} < E_{d\_SDMA} < E_{d\_SMA} \,, \tag{11}$$

which can be clearly observed from the results in Fig. 14.

When only data memory power consumption is considered, transformation C_1 and P_4 provide the optimal solution for the current and previous frames, respectively. This is valid both for the single- and dual-processor and for all three memory architectures. In addition, it is observed that each transformation affects in a similar way the energy consumption of the different implementations, independently of the number of processors and the used memory architecture. For example, for the single processor architecture, transformation P_2 is more power efficient than P_1. This is also valid for the case of dual-processor architectures.

However, since any programmable platform fetches instructions from its instruction memory, the power consumption due to these accesses has to be taken into account.[8,11] The instruction memory energy consumption depends not only on the number of accesses for instruction fetching, but also on the code size. The code size and thus the instruction memory size for each of the two processors is almost similar to that of the single processor. The relation between the number of executed instructions in each architecture is

$$\varepsilon_{total}^{SMA} > \varepsilon_{total}^{SDMA} > \varepsilon_{total}^{DMA} > \frac{1}{2}\varepsilon_{total}^{single} \,. \tag{12}$$

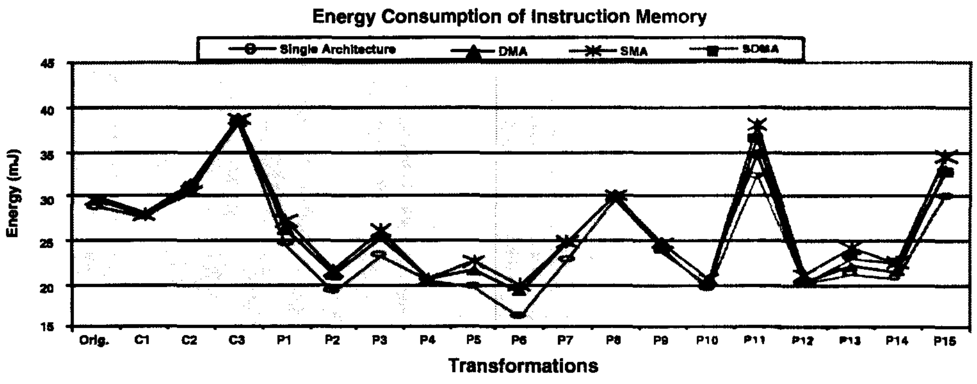


Fig. 15. Energy consumption on instruction memory.

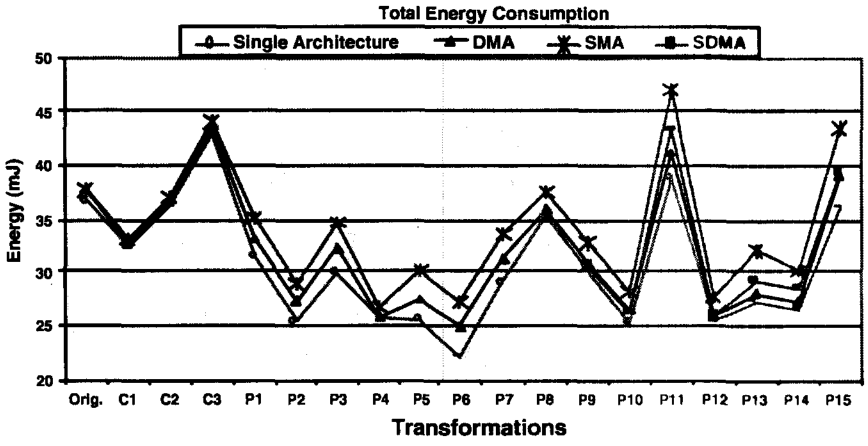


Fig. 16. Total energy consumption.

The differences between the three memory architectures are due to a number of different control operations, necessary in each code. The results, for the energy consumed on the instruction memory, are given in Fig. 15. As it is observed, this energy component is significantly greater than that of the data memories.

In Fig. 16 the total power consumption for all three memory architectures including the case of a single processor is shown. The distributed memory architecture seams to be the most energy efficient architecture of the parallel ones according to Refs. 6 and 7, and the experimental results. The shared memory architecture is the most energy costly because it consists of dual-port memories resulting in higher energy cost per access. Finally, from Fig. 16 it is observed that the relative effect of each transformation on the total energy remains unaffected by the number of processors and the memory architectures.

For the shared–distributed case only some measurements are displayed since for some transformations the application of a shared–distributed scheme has no sense. These are the transformations that contain the RW line memory layer, since if this layer is split to two memories, one for each processor, the whole scheme degenerates to the distributed memory architecture.

4.2. *Area*

The area occupied by data memory elements shown in Fig. 17 is calculated using an appropriate model for area. In the figure only the on-chip memory elements, determined by the memory architecture for each transformation, are considered. It can be inferred that all transformations increase area, since they impose the addition of extra memory elements. It is also obvious that the distributed memory architecture is the most inefficient in terms of data memory area, since the on-chip memory modules occupy twice as much area than the single processor case. Moreover, it is less area efficient than the shared architecture since two single-port memory blocks occupy more area than a single double-port memory. Data memories in the shared architecture occupy more area than the single processor case, since on-chip memories are double-port. Shared–distributed architecture lies in between since it employs separate single-port memory blocks for the lower levels and double-port memory blocks for the higher levels.

4.3. *Performance*

In Fig. 18 performance is defined as the total number of required clock cycles for processing a frame. The observed deviations in performance between the three

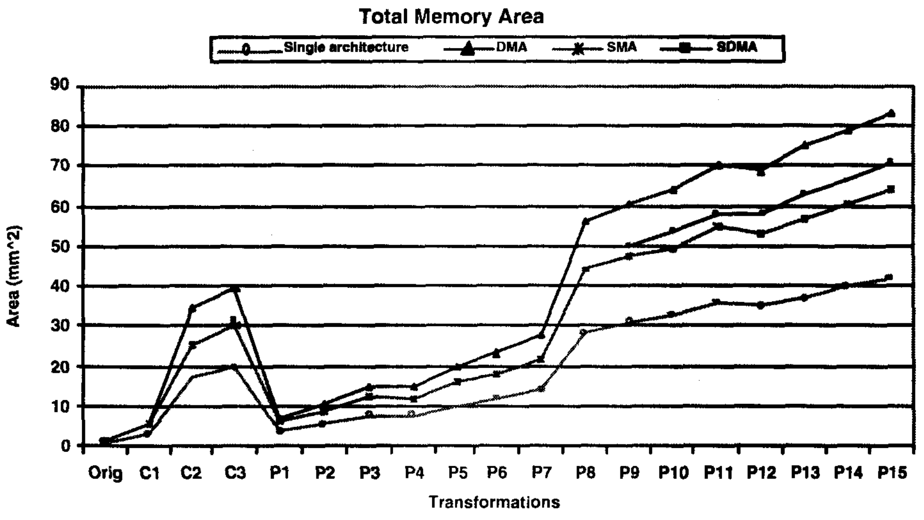


Fig. 17. Total data memory area.

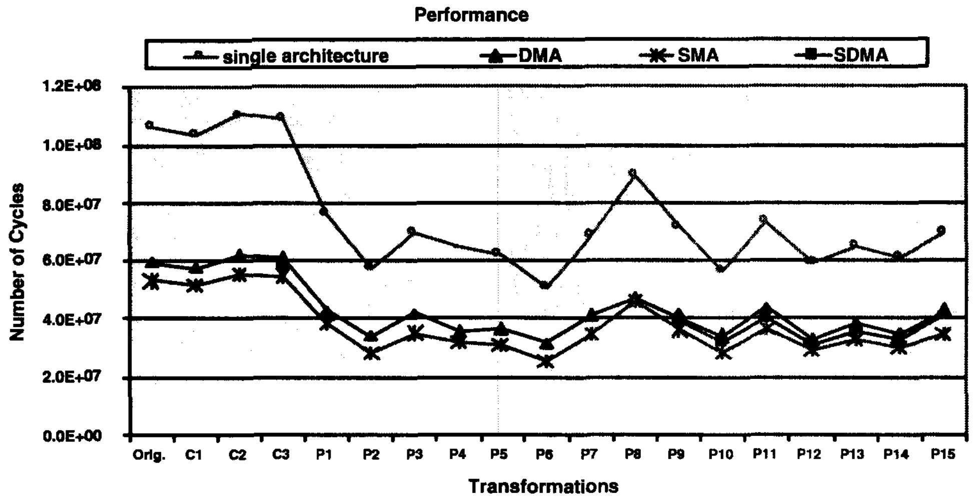


Fig. 18. Performance comparison.

memory architectures for the two-processor architecture are insignificant although a slightly better performance is observed for the shared memory architecture. Consequently, the selection of the most appropriate code transformation and memory architecture should be based mainly on energy and area criteria. Ideally, the use of two processors should double the achieved performance compared to the single processor case. However this is not feasible since the workload cannot be equally-partitioned between the two processors. Moreover, the performance of a parallel system is even more decreased due to control signals between the processors.

5. Conclusions

A novel methodology that extends the well established Data Transfer and Storage Exploration methodology for the evaluation of power efficient data-reuse transformations, has been presented. These transformations achieve power reduction by moving background memory accesses to foreground memories of smaller size. Analytical expressions for the number of accesses to each memory layer and the number of executed instructions are derived, allowing a fast exploration of the design space by varying all algorithmic parameters. These expressions are obtained only for the minimum set of data transfers between memory layers and then applied to all transformations, reducing significantly the required effort and time. Experimental results prove that for data-dominated applications, the optimal solution in terms of power, performance and area can result by the right combination of high-level decisions for the adaptation of a certain data memory architecture and the application of high-level data-reuse transformations.

The proposed methodology has also been applied to the case of parallel embedded multimedia processor cores. It is concluded that the relative effect of each transformation on energy and performance remains unaffected by the number of processors and the memory architecture. Consequently, full exploration of the effect of the transformations can be performed on single processor architectures, minimizing the required exploration space.

Acknowledgments

The authors would like to thank postgraduate students A. Zanikopoulos and Ch. Voliotidis for their contribution in simulating parallel processor systems. This work was supported by the ED 501 PENED'99 project funded by G.S.R.T. of the Greek Ministry of Development and European Union.

References

1. F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vande-Cappelle, *Custom Memory Management Methodology* (Kluwer Academic Publishers, Boston, 1998).
2. L. Benini and G. De Micheli, System-level power optimization: techniques and tools, *ACM Trans. Design Automation of Electronic Systems* **5** (2000) 115–192.
3. P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. V. Cappelle, and P. G. Kjeldsberg, Data and memory optimization techniques for embedded systems, *ACM Trans. Design Automation of Electronic Systems* **6** (2001) 149–206.
4. W.-T. Shiue, S. Udayanarayanan and C. Chakrabarti, Data memory design and exploration for low-power embedded systems, *ACM Trans. Design Automation of Electronic Systems* **6** (2001) 553–568.
5. A. Chandrakasan and R. Brodersen, *Low Power Digital CMOS Design* (Kluwer Academic Publishers, Boston, 1995).
6. S. Wuytack, J.-P. Diguet and F. Catthoor, Formalized methodology for data reuse exploration for low-power hierarchical memory mappings, *IEEE Trans. VLSI Syst.* **6** (1998) 529–537.
7. L. Nachtergaele, B. Vanhoof, F. Catthoor, D. Moolenaar, and H. De Man, System-level power optimizations of video codecs on embedded cores: A systematic approach, *J. VLSI Signal Processing Syst.* **18** (1998) 89–109.
8. N. D. Zervas, K. Masselos and C. E. Goutis, Data-reuse exploration for low-power realization of multimedia applications on embedded cores, in *Proc. 9th Int. Workshop on Power and Timing Modeling, Optimization and Simulation*, October 1999, Kos, Greece, pp. 71–80.
9. K. Masselos, F. Catthoor, C. E. Goutis and H. De Man, Code size effects of power optimizing code transformations for embedded mutlimedia applications, in *Proc. 9th Int. Workshop on Power and Timing Modeling, Optimization and Simulation*, October 1999, Kos, Greece, pp. 61–70.
10. U. Eckhardt and R. Merker, Hierarchical algorithm partitioning at system level for an improved utilization of memory structures, *IEEE Trans. CAD* **18** (1999) 14–24.
11. D. Soudris, N. D. Zervas, A. Argyriou, M. Dasygenis, K. Tatas, C. E. Goutis, and A. Thanailakis, Data-reuse and parallel embedded architectures for low-power, real-time multimedia applications, in *Proc. 10th Int. Workshop on Power and Timing Modeling, Optimization and Simulation*, September 2000, Gottingen, Germany, pp. 243–254.
12. M. Dasigenis, N. Kroupis, A. Argyriou, K. Tatas, D. Soudris, and N. Zervas, Data and

instruction memory exploration of embedded systems for multimedia applications, in *Proc. Int. Conf. Acoustics, Speech and Signal Processings*, May 2001, Salt Lake City, Utah, USA.

13. P. E. Landman and J. M. Rabaey, Architectural power analysis: The dual bit type method, *IEEE Trans. VLSI Syst.* **3** (1995) 173–187.
14. F. Catthoor, K. Danckaert, S. Wuytack, and N. D. Dutt, Code transformations for data transfer and storage exploration preprocessing in multimedia processors, *IEEE Design & Test of Computers* **18** (2001) 70–82.
15. V. Bhaskaran and K. Konstantinides, *Image and Video Compression Standards: Algorithms and Architectures* (Kluwer Academic Publishers, Boston, 1999).
16. ARM software development toolkit, v2.11, Copyright 1996-7, Advanced RISC Machines.
17. J. M. Mulder, N. T. Quach and M. J. Flynn, An area model for on-chip memories and its application, *IEEE J. Solid-State Circuits* **SC26** (1991) 98–105.