Formalized methodology for data reuse exploration for low-power hierarchical memory mappings

S. Wuytack, J.Ph. Diguet, F. Catthoor, and H. De Man

Abstract— **Efficient use of an optimized custom memory hierarchy to exploit temporal locality in the data accesses can have a very large impact on the power consumption in data dominated applications. In the past experiments have demonstrated that this task is crucial in a complete low-power memory management methodology. But effective formalized techniques to deal with this specific task have not been addressed yet. In this paper, the surprisingly large design freedom available for the basic problem is explored in-depth and the outline of a systematic solution methodology is proposed. The efficiency of the methodology is illustrated on a real-life motion estimation application. The results obtained for this application show power reductions of about 85% for the memory sub-system compared to the case without a custom memory hierarchy. These large gains justify that data reuse and memory hierarchy decisions should be taken early in the design flow.**

Keywords— **Special-issue-lowpower97, system-level, low-power-design, tradeoffs, memory, video-processing**

I. Introduction

A large part of the power dissipation in data dominated applications is due to data transfers and data storage. This power component can often be reduced by introducing an optimized custom memory hierarchy that exploits the temporal locality in the data accesses. The impact of this can be very large, as has been demonstrated by us on an H.263 video decoder [1] and a motion estimation application [2].

The idea of using a custom memory hierarchy to minimize the power consumption is based on the fact that memory power consumption depends primarily on the access frequency and the size of the memory. For on-chip memories, which are not very much partitioned, memory power increases with the memory size. In practice, the relation is between linear and logarithmic depending on the memory library. For off-chip memories, the power is much less dependent on the size because they are internally heavily partitioned. Still they consume more energy per access than the smaller on-chip memories. Hence, power savings can be obtained by accessing heavily used data from smaller memories instead of from large background memories. Such an optimization requires architectural transformations that consist of adding layers of smaller and smaller memories to which frequently used data can be copied [3]. Memory hierarchy optimization introduces copies of data from larger to smaller memories in the data flow graph. This means that there is a trade-off involved here: on the one hand, power consumption is decreased because data is now read mostly from smaller memories, while on the other hand, power consumption is increased because extra memory transfers are introduced. The memory hierarchy design task has to find the best solution for this trade-off.

Memory hierarchy design for power optimization is basically different from caching for performance optimization [4]. The latter determines how to fill the cache such that data has been loaded from main memory before it is needed. Instead of minimizing the number of transfers, the number of transfers is often increased to maximize the chance of a cache hit, leading to wasted power by prefetching data that may never be needed.

Some custom memory hierarchy experiments on real-life applications can be found in literature. What is not solved, however, is how to decide on the optimal memory hierarchy. In this paper, we present a formalized methodology for this decision and give an indication of how large the search space really is. The latter is much larger than conventionally exploited in state-of-the-art designs. For more information on the context of this work we refer to [5].

The rest of the paper is organized as follows. Section II discusses the related work. Section III defines the global memory hierarchy problem. Section IV defines the data reuse decision problem and points out important issues towards a methodology. Section V presents our methodology for solving the data reuse exploration and decision problem. Section VI discusses the results of the data reuse exploration experiment for a motion estimation application. Section VII concludes the paper.

II. Related Work

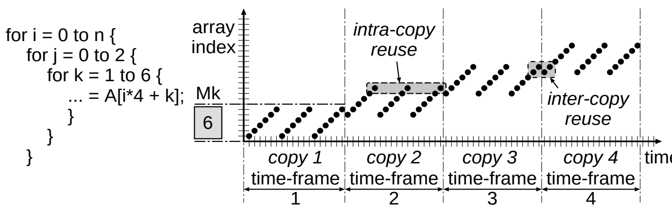
The main work related to data reuse exploration lies in the parallel compiler area, especially related to the cache hierarchy. Here, several papers have analyzed memory organization issues in processors [6]. This, however, has not resulted yet in any formalizable method to guide the memory organization issues. In most work on parallel MIMD processors, the emphasis in terms of storage hierarchy has been on hardware mechanisms based on cache coherence protocols [7]. Partitioning or blocking strategies for loops to optimize the use of caches have been studied in several contexts [4]. The main focus is on performance improvement though and not on memory cost. Recently, also in a system synthesis context, applying transformations to improve the cache usage has been addressed [8], [9]. None of these approaches determine the best memory hierarchy organization for a given (set of) applications and only few address the power cost. Only an analysis of memory hierarchy choices based on statistical information to reach a given throughput expectation has been discussed recently [10]. In the hardware realization context, much less work has been performed, mainly oriented to memory allocation [11], [12], [13]. This paper discusses how to decide on the optimal use of the memory in a systematic way.

III. Memory Hierarchy Design

This section defines the memory hierarchy design task. First, it shows that memory hierarchy design is about exploiting temporal locality. Then, it shows that memory hierarchy design consists of two steps: *data reuse decision*, which is the topic of this paper, and *memory layer assignment*, which is left for a future paper.

A. Exploiting temporal locality

Memory hierarchy design exploits data reuse local in time to save power by copying data that is reused often in a short time period to a smaller memory, from which the data can then be accessed. Fig. 1 illustrates this for all read operations to a given array.



```
for i = 0 to n {
    for j = 0 to 2 {
        for k = 1 to 6 {
            ... = A[i*4 + k];
        }
    }
}
```

Fig. 1. Exploiting data reuse local in time to save power.

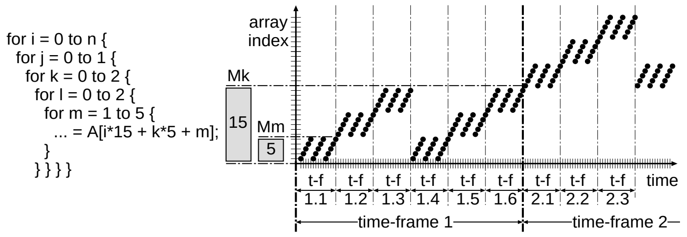
The horizontal axis is the time axis. It shows how the data accesses are ordered relatively to each other in time. The vertical axis shows the index of the array elements. Every dot represents a memory read operation, scheduled at a certain time and accessing a given array element. In this example most values are read multiple times. Assuming that the data is still needed later on, all of the array elements have to be stored in a large background memory. However, when we look at smaller time-frames (indicated by the vertical dashed lines), we see that only part of the data is needed in each time-frame, so this part of the data would fit in a smaller, less power consuming memory. If there is sufficient reuse of the data in that time-frame, it can be advantageous to copy the data that is used frequently in this time-frame to a smaller memory. Consequently, the second time an array element has to be read, it can be read from the smaller memory instead of the larger memory. This leads to the following definitions.

Definition: time-frame
The execution time of an application can be subdivided into a number of non-overlapping time-intervals, called level 1 time-frames. *Each level i time-frame can be subdivided further into non-overlapping level $i+1$ time-frames.*

Definition: copy-candidate
A copy-candidate corresponding to an array A and time-frame TF_i is a set of array elements of A that are read in TF_i and are considered for copying to a lower hierarchy level.

During the data reuse task it will be decided which copy-candidates are really worth to be copied in order to save power. Once this decision is made, the transfers for making the copies will be added to the application code and the copy-candidates become real (partial) copies of their corresponding arrays.

Data reuse is the result of **intra**-*copy reuse* and **inter**-*copy reuse* (cfr. Fig. 1). *Intra-copy* reuse means that each array element is read several times from its memory during one time-frame. *Inter-copy* reuse means that advantage is taken from the fact that part of the data needed in the next time-frame could already be available in the memory from the previous time-frame, and therefore does not have to be copied again.



```
for i = 0 to n {
  for j = 0 to 1 {
    for k = 0 to 2 {
      for l = 0 to 2 {
        for m = 1 to 5 {
          ... = A[i*15 + k*5 + m];
        }
}}}}
```

Fig. 2. Possibility for multi-level hierarchy.

Taking full advantage of temporal locality for power, usually requires architectural transformations that consist of adding several layers of memories (each corresponding to its own time-frame level) between the large background memories and the small foreground memories (registers in the data-path). Every layer in the memory hierarchy contains smaller memories than the ones used in the layer above it. An example of this is shown in Fig. 2. It shows time-frames that are subdivided into smaller time-frames. Each level of time-frames potentially corresponds to a memory layer in the memory hierarchy.

B. Steps in memory hierarchy design

Two steps can be identified in memory hierarchy design:
1. The *data reuse exploration and decision* step decides *which* intermediate copies have to made for accessing the data in a power efficient way.
2. The *memory layer assignment* step decides for each array and copy of an array on *which layer* in the common custom memory hierarchy it will be stored.

After memory layer assignment, an optimal memory architecture has to be derived for each layer. This is done by the subsequent memory allocation and array-to-memory assignment tasks in our Data Transfer and Storage Exploration (DTSE) methodology [5]. This paper focuses on the data reuse decision step only.

IV. Data Reuse Decision

This section discusses the data reuse exploration and decision step of memory hierarchy design. It points out important elements for a systematic methodology described in the next section.

A. Search space

The following assumptions allow to focus the data reuse task, without really restricting the search space in practice:

• *Only read operations have to be considered*
The reason for this is that repeated reading of the same data *value* makes sense (i.e., repeatedly reading the same memory location without intermediate writes to that memory location), whereas writing *the same* data *value* usually does not make sense. So there is no need for creating a memory hierarchy for repeatedly written data. The only thing that has to be decided for write operations is in which layer a certain (temporary) array will be written. This is decided in the memory layer assignment step *after* data reuse decisions are made.

• *Only one array has to be considered at a time*
Data reuse *exploration* can be tackled for each array separately. The main reason for this is that each copy in the memory hierarchy has its own root array, i.e., the copies form a tree where the *root* is the original array that is being accessed. A copy cannot be obtained as the mix of different arrays. We will also assume that the data reuse *decision* can be taken for each array separately. As explained later on, this limits the search space to some extent.

B. Data-reuse factor

The usefulness of memory hierarchy for saving power is strongly related to the array's data reusability, because this is what determines the ratio between the number of read operations from a copy of an array in a smaller memory, and the number of read operations from the array in the larger memory on the next hierarchical level.

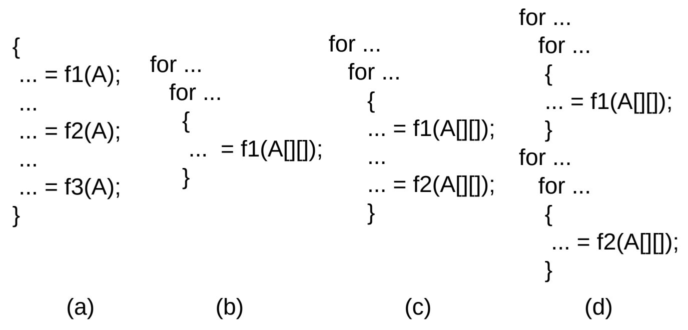
The reuse factor of a group of data D stored in a memory on layer i relative to layer $i-1$, where layer 1 is the furthest from the data paths, is defined as:

$$F_R(i, D) = \frac{N_R(M_i, D)}{N_R(M_{i-1}, D)} \qquad (1)$$

where $N_R(M_i, D)$ is the total number of times an element from D is read from a memory at layer i. A reuse factor larger than 1 is the result of *intra*-copy and *inter*-copy reuse (cfr. Fig. 1).

C. Classification of data reuse

Fig. 3 presents a classification of four cases in which data reuse can be exploited by means of memory hierarchy. These four cases are:



```
(a)
{
 ... = f1(A);
 ...
 ... = f2(A);
 ...
 ... = f3(A);
}
```

```
(b)
for ...
  for ...
  {
   ... = f1(A[][]);
  }
```

```
(c)
for ...
  for ...
  {
   ... = f1(A[][]);
   ...
   ... = f2(A[][]);
  }
```

```
(d)
for ...
  for ...
  {
   ... = f1(A[][]);
  }
for ...
  for ...
  {
   ... = f2(A[][]);
  }
```

Fig. 3. Classification of data reuse opportunities.

(a) *No loops*
In case there are no loops, there is no structured use of arrays. In fact, each array element is treated independently from the others similar to scalars. This case is *not* considered in our methodology. It is left for scalar methodologies which are more suited for this.

(b) *One read instruction in a (nested) loop*
This is the basic case on which our methodology is based. Both intra-copy and inter-copy reuse are possible here, when the loop nesting (i.e., the order of the different nested loops, and the direction in which the loops are traversed) is fixed. Indeed, in this case, the ordering of the different copies is known and every two consecutive copies can be examined for overlap (i.e., inter-copy reuse).

(c) *Multiple read instructions in a (nested) loop*
Here it is assumed that each read instruction has a different index expression, because otherwise they can be reduced to the previous case by reading once from background memory and storing the result in a foreground register. When the read instructions are accessing different parts of the array, a different memory hierarchy can be constructed for each of them. In practice, these memory hierarchies can contain partly the same data, and are then best combined. Determining which part of the memory hierarchy can be shared, can be done with a geometrical data flow analysis.

(d) *Read instructions in different loop nests*
In this case, a memory hierarchy can be derived for each of the loop nests separately. Because they are in different loop nests, these memory hierarchies can be very different from each other. Depending on the temporal locality (which is only known when the ordering of all loop nests is already fixed), it may be useful to copy the data that is common to these loops to an extra memory layer. Again a polyhedral analysis can be used to determine which part of the array is used in common. Remark that when more than two loops are involved, part of the array can be common to only a few of the loops involved, making things much more complex.

V. Proposed Methodology

In this section, we propose a methodology for data reuse exploration and decision based on a number of assumptions to make the solution feasible for real-life applications.

A. Assumptions

The following is assumed in our methodology:

• *Nesting order and direction of nested loops is fixed*
The fixed nesting order is required to determine the time-frames and copy-candidates corresponding to each read instruction in the loop nest. The fixed iteration direction is required to determine the overlap for estimating the *inter-copy* reuse.

• *Time-frames are determined by loop boundaries*
Finding an optimal time-frame hierarchy is a very complex problem. However, we believe that the optimal time-frame boundaries are likely to coincide with the loop boundaries of loop nests. Therefore, we use as a heuristic that the loop boundaries correspond to time-frame boundaries, instead of trying to find globally optimal time-frame boundaries.

• *A copy-candidate contains all data read in its time-frame*
It is assumed that *all* data being accessed by a certain read instruction in a certain time-frame will be copied to a copy-candidate, such that all data required by the read instruction can be found inside the copy-candidate. Ideally, only part of the data that is accessed more than once should be copied. The loss due to this restriction is small in practice.
• *Copy-candidates of a time-frame level are stored in-place*
It is assumed that at the end of a time-frame, the data copied into the intermediate memory is not needed anymore, and will be overwritten by the data needed in the next time-frame. Therefore the size of the copy-candidate corresponding to a certain time-frame level is determined by the time-frame leading to the largest copy-candidate.
• *Copy-candidates are stored in perfectly fitting memories*
It is assumed that a copy-candidate will be stored in a memory with word depth and bit width equal to those of the copy-candidate. In general, this assumption leads to an underestimation of the power cost, as usually several copy-candidates will have to share a memory. This real memory size is only known until after array-to-memory assignment and can therefore not be taken into account.
• *Inter-copy reuse is fully exploited*
It is assumed that inter-copy reuse will be fully exploited. This means that data that is already present in the smaller memory from the previous copy will not be copied again. Only data that is not yet present has to be copied over the data of the previous copy that is not needed anymore. This affects the number of accesses to each of the copy-candidates. The size of the copy-candidates is unaffected.

B. Data reuse exploration

Based on the assumptions listed in the previous subsection, we propose a systematic data reuse exploration methodology. Fig. 4 illustrates the different steps for a small example.

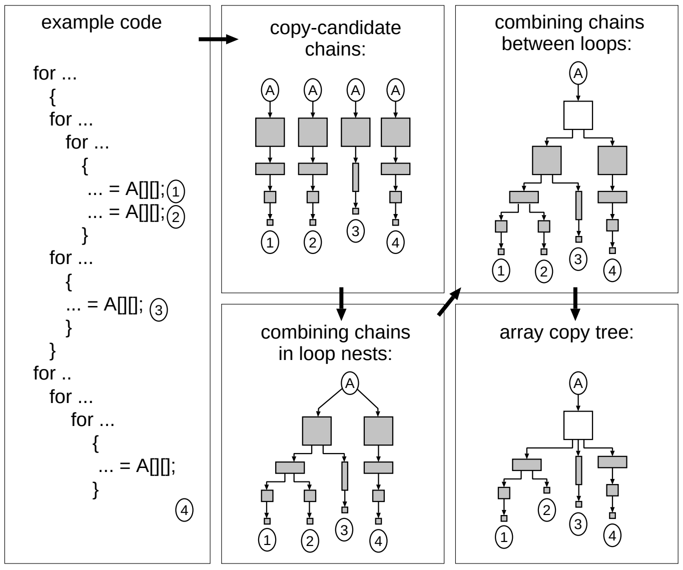


Fig. 4. Data reuse exploration methodology: example.

Copy-candidate chains

For each read instruction inside a loop nest, a *copy-candidate chain* can be determined in the following way. The first copy-candidate in the chain contains *all* array elements accessed by the given read instruction during the execution of the loop nest. The second copy-candidate in the chain is associated with the iterations of the outer loop. Each iteration has a corresponding time frame. All array elements that are accessed by the read instruction during a given time frame are stored in the corresponding copy-candidate. The storage space of the copy-candidate is shared among the different iterations of the loop. In case the number of elements accessed by the read instruction varies between iterations, the size of the copy-candidate is determined by the iteration that accesses the most array elements. This can be repeated for each of the remaining loop nest levels. Together these copy-candidates form the *copy-candidate chain* for the considered read instruction. Such a chain represents the maximal exploitation of data reuse of type (b) in Fig. 3.

Copy-candidate trees

Copy-candidate chains of different read instructions accessing the same array can be combined into a *copy-candidate tree* for that array:
1. *Read instructions belonging to the same loop nest*
Often, the copy-candidates of read instructions belonging to the same loop nest contain the same data. If this is the case for all iterations, the copy-candidate can be shared between the read instructions. Moreover, if a copy-candidate can be shared, also the copy-candidates before it in the chain can be shared. Because the first copy-candidate, which is the array itself, can always be shared, the copy-candidate chains can always be combined into a *copy-candidate tree*. Such a tree represents the maximal exploitation of data reuse of type (b) and (c) in Fig. 3.
2. *Read instructions belonging to different loop nests*
Copy-candidates that cannot be combined in the previous way because they belong to different loop nests (cfr. Fig. 3(d)), can still share the same data. In this case, an extra intermediate copy-candidate can be inserted in the copy-candidate tree to exploit this data reuse opportunity. When more than two copy-candidates share data in this way, the search space grows quickly: for every possible subset of them an extra copy-candidate could be introduced. This freedom will be explored in future research and will not be considered further in this paper. So we will treat the read instructions as operating on independent arrays.
Two simple rules can be applied to prune copy-candidates from copy-candidate trees because they will never occur in an optimal memory hierarchy: the size of the copy-candidates must decrease from one layer to the next, and the reuse ratio (Eq. 1) of each layer must be larger than 1.

Copy-candidate graphs

From the previous step we can conclude that for every array, a copy-candidate tree can be determined. Each node in the tree represents a copy-candidate. A copy-candidate can be characterized by its required memory size, number of write operations to copy data into it, and the total number of read operations to copies on lower layers. Also its data reuse-factor can be calculated. From this tree other valid copy trees can be derived because it is allowed to copy data from any ancestor node in the tree, not only the parent node. Therefore, we extend the copy-candidate tree to a *copy-candidate graph* in the following way: for every node in the tree, we add edges starting from all its ancestor nodes towards the node itself. All possible trees that can be derived by selecting a single path from the root node to every leaf node, represents a valid copy tree.

Array copy trees

The *array copy tree* is the lowest cost tree obtained from the copy-candidate graphs in the way described above. Selecting the array copy tree from all possible copy trees is called data reuse *decision* (cfr. subsection V-D). The cost function for selecting the array copy tree is defined next.

C. Cost function

The cost function for selecting the optimal copy tree is a weighted sum of a power and area estimate for the copy tree CT. The cost function is given by:

$$\begin{aligned} cost(CT) = \quad & \alpha \cdot \sum_{c\,\in\,CT} [P_r(N_{bits}(c), N_{words}(c), f_{read}(c)) \\ & \quad + P_w(N_{bits}(c), N_{words}(c), f_{write}(c))] \\ + \quad & \beta \cdot \sum_{c\,\in\,CT} A(N_{bits}(c), N_{word}(c)) \end{aligned} \tag{2}$$

where

- c is a copy-candidate of the considered copy tree CT,
- $P_{r/w}(N_{bits}, N_{words}, f_{access})$ is the power estimate for read/write operations of a memory with bit width N_{bits}, word depth N_{words}, and that is accessed with a *real* access frequency f_{access},
- $A(N_{bits}, N_{words})$ is the area estimate for a memory with specified parameters, *and*
- α and β are weighting factors for area/power trade-offs.

The *real* access frequency f_{access} of a memory is obtained by multiplying the number of memory accesses per frame with the frame rate (**not** the clock frequency).

D. Data reuse decision

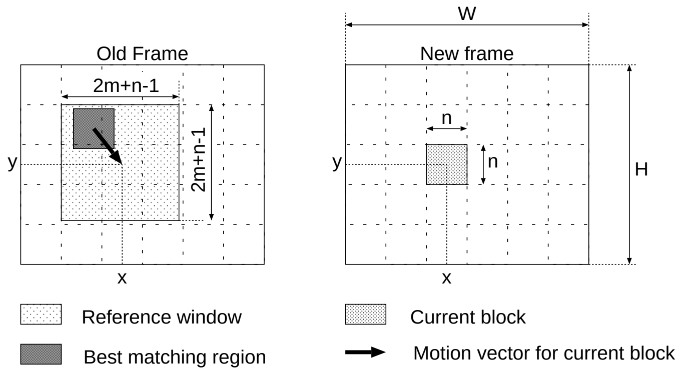
The end result of the data reuse decision task is an optimal *array copy tree* for each array in the application. Here, it is assumed that the optimal copy tree can be derived for each array independently from the other arrays. This is not completely true because the optimal copy trees depend partly on how well the different copies can share memory space (inter-array in-place [15]). However, because the data reuse decision is best taken early in the design flow, not enough data about inter-array in-place is available to include it in the decision process.

VI. Test Vehicle: Motion Estimation Algorithm

The motion estimation (ME) algorithm is used as a test-vehicle to illustrate the proposed methodology.

A. Algorithm and Cost Functions

The motion estimation algorithm is used in moving image compression algorithms. It allows to estimate the motion vector of small blocks of successive image frames. The version we consider here is the kernel of what is commonly referred to as the "full-search full-pixel" implementation [16]. The algorithm and its parameters are shown in Fig. 5.



```
for (g=0; g<H/n; g++)              /* vertical CB counter */
  for (h=0; h<W/n; h++) {          /* horizontal CB counter */
    $\Delta_{opt}[g][h] = +\infty$;
    for (i=-m; i<m; i++)           /* vertical searching of RW */
      for (j=-m; j<m; j++) {       /* horizontal searching of RW */
        $\Delta = 0$;
        for (k=0; k<n; k++)        /* vertical traversal of CB */
          for (l=0; l<n; l++) {    /* horizontal traversal of CB */
            $\Delta$ += abs($New[g.n+k][h.n+l]$
                            $-Old[g.n+i+k][h.n+j+l]$;)
          }
        $\Delta_{opt}[g][h]$ = min($\Delta$, $\Delta_{opt}[g][h]$);
      }
  }
```

Fig. 5. The motion estimation algorithm and its parameters

For the experiments we have used the parameters of the QCIF format (W=176, H=144, m=n=8) with a frame rate of 30 frames/s. We are using an accurate but proprietary model for estimating the power and area of the memory modules from a specific library for which we are not allowed to publish absolute values on area and power. Therefore only relative values are provided.

B. Data reuse exploration for ME application

Copy-candidate chains. For the motion estimation application only the frame arrays *Old (O)* and *New (N)* will be considered here, because the other arrays can easily be stored in a foreground register. Fig. 6 shows the copy-candidate chains for the two read instructions accessing array O and N respectively. Copy-candidates N_4 and N_5 can be pruned because they are not smaller than N_3.

Copy-candidate trees. Since there is only one read instruction from the *Old* frame and one from the *New* frame, there is nothing to combine in this example: the copy-candidate

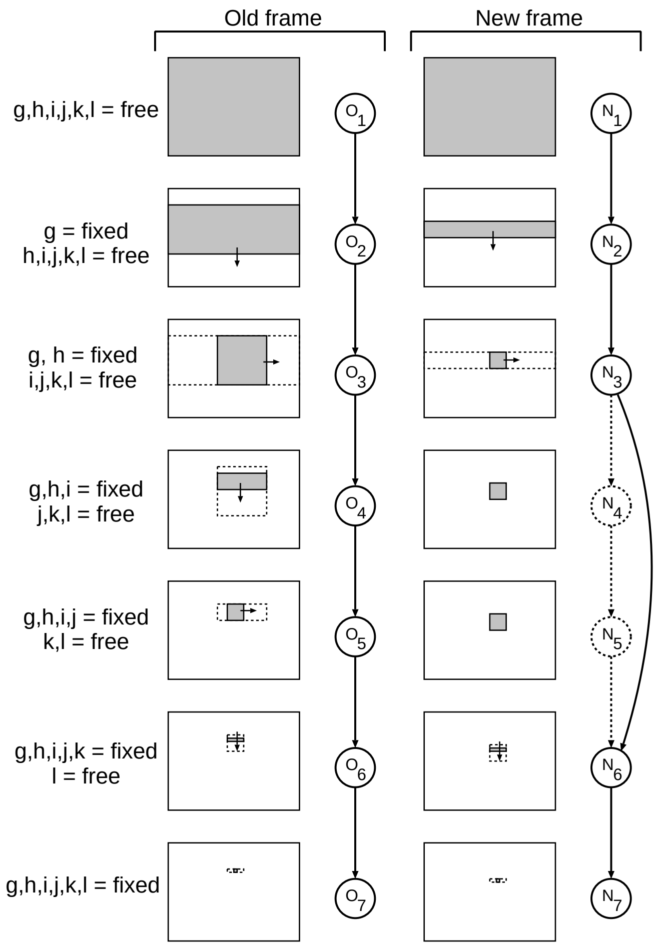


Fig. 6. Copy-candidate chains for ME application.

chains for the frame arrays *are* the copy-candidate trees. The nodes O_6 and N_6 can now be pruned, though, because they have a reuse ratio of one (i.e., no reuse).

Copy-candidate graphs. The copy-candidate graphs for the two frame arrays are shown in Fig. 7.

Array copy trees. In this case the copy-candidate trees are in fact chains.[1] Because of the chains, the search process for finding the array copy tree can be represented as a simple search tree. Fig. 8 shows these search trees for our motion estimation example. The (optimal) array copy trees are indicated in grey. The dashed lines divide the search trees in a number of layers. Each layer corresponds to a copy-candidate (or time-frame level). The solutions can either include this copy-candidate (copy is shown in search tree) or skip it (copy is not shown). Every node in the tree represents a solution: the copy-candidate on that node is the lowest layer in the hierarchy, and all copy-candidates on the path between the root node and the node itself are intermediate layers in the hierarchy. For each solution, the area (A) and power (P) of the complete array copy tree relative to the solution without hierarchy are indicated.

[1]In general this is not true as demonstrated by us before for the H.263 video conferencing decoder test vehicle in [1].

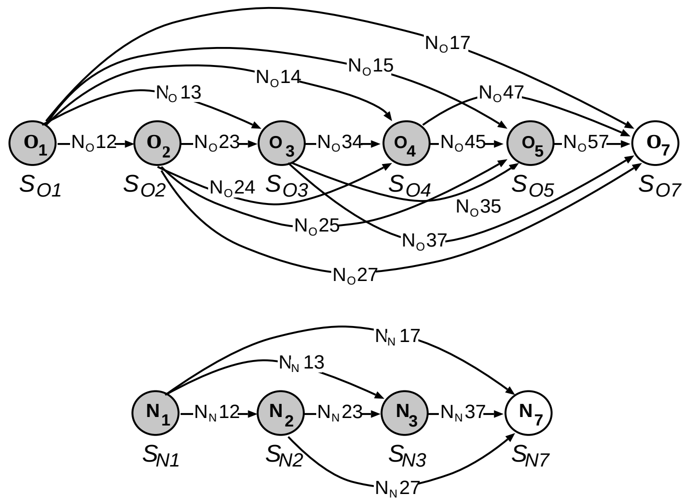


Fig. 7. Copy-candidate graphs for ME application.

C. Discussion

A surprising result is that the total memory area can *decrease* by adding extra memory layers (cfr. Fig. 8). The reason for this is that the maximum access frequency of the memories is taken into account in our estimations. If a certain memory would be accessed above its maximum access frequency, this memory will be split into two memories of half the size to increase the memory bandwidth. This splitting introduces overhead. By adding extra memory layers with small memories, the bandwidth requirements of the large background memories can be reduced, and therefore splitting can be avoided for the large memories. This area gain can be larger than the area lost by adding a few small memories.

The optimal memory hierarchy for power is:

- for the *Old* frame, a 3-level hierarchy that leads to a power saving of 83% compared to the solution without memory hierarchy;
- for the *New* frame, a 2-level hierarchy that leads to a power saving of 87% compared to the solution without memory hierarchy.

These figures do not include the power dissipation in the interconnect. Taking this into account will result in even larger gains, as off-chip communication dissipates much more power than on-chip communication. Without memory hierarchy all data transfers are off-chip. With memory hierarchy, most of these are replaced by less power consuming on-chip transfers.

If we compare the result with the one we proposed in an ad hoc way in an earlier paper [2], we note that a different memory hierarchy with only two levels was selected which results in a higher power consumption. This clearly shows that by using a more systematic design space exploration methodology which exploits the full search space available, as proposed in this paper, better results can be obtained.

VII. Conclusion

Exploiting temporal locality in the memory accesses by means of an optimized memory hierarchy can effectively reduce the power dissipation of data-dominated applications. The memory hierarchy design task can be split into two steps: data reuse decision and memory layer assignment. The first step is the topic of this paper, the second is left for a future paper.

A systematic methodology for the data reuse decision step has been proposed based on a number of realistic assumptions. The feasibility and the large impact of the proposed techniques have been shown on a real-life video application. The results obtained for the motion estimation application show power reductions of about 85% for the memory sub-system compared to the case without memory hierarchy. Similar results have been obtained for other applications not presented in this paper. These figures do not include the power dissipation in the interconnect. Taking this into account will result in even larger gains. These large power gains justify that the memory hierarchy should be decided early in the design script.

Currently we are extending the methodology to work also for instruction set processors with a (partially) fixed memory hierarchy.

References

[1] L.Nachtergaele, F.Catthoor, B.Kapoor, D.Moolenaar, S.Janssen, "Low power storage exploration for H.263 video decoder", *IEEE workshop on VLSI signal processing*, Monterey CA, Oct. 1996. Also in *VLSI Signal Processing IX*, W.Burleson, K.Konstantinides, T.Meng, (eds.), IEEE Press, New York, pp.116-125, 1996.

[2] S.Wuytack, F.Catthoor, L.Nachtergaele, H.De Man, "Power Exploration for Data Dominated Video Applications", *Proc. IEEE Intnl. Symp. on Low Power Design*, Monterey, pp.359-364, Aug. 1996.

[3] S.Wuytack, F.Catthoor, F.Franssen, L.Nachtergaele, H.De Man, "Global communication and memory optimizing transformations for low power systems", *IEEE Intnl. Workshop on Low Power Design*, Napa CA, pp.203-208, April 1994.

[4] D.Kulkarni, M.Stumm, "Linear loop transformations in optimizing compilers for parallel machines", Technical report, Comp. Systems Res. Inst. Univ. of Toronto, Canada, Oct. 1994.

[5] F.Catthoor, S.Wuytack, E.De Greef, F.Franssen, L.Nachtergaele, H.De Man, "System-level transformations for low power data transfer and storage", *chapter in "Low power design" (eds. B.Brodersen, A.Chandrakasan)*, IEEE Press, 1998.

[6] A.Faruque, D.Fong, "Performance analysis through memory of a proposed parallel architecture for the efficient use of memory in image processing applications", *Proc. SPIE'91, Visual communications and image processing*, Boston MA, pp.865-877, Oct. 1991.

[7] L.Liu, "Issues in multi-level cache design", *Proc. IEEE Int. Conf. on Computer Design*, Cambridge MA, pp.46-52, Oct. 1994.

[8] D.Kolson, A.Nicolau, N.Dutt, "Elimination of redundant memory traffic in high-level synthesis", *IEEE Trans. on Comp.-aided Design*, Vol.15, No.11, pp.1354-1363, Nov. 1996.

[9] P.Panda, N.Dutt, A.Nicolau, "Memory Organization for Improved Data Cache Performance in Embedded Processors", *1996 International Symposium on System Synthesis*, La Jolla CA, pp.90-95, Nov. 1996.

[10] B.Jacob, P.chen, S.Silverman, T.Mudge, "An analytical model for designing memory hierarchies", *IEEE Trans. on Computers*, Vol.C-45, No.10, pp.1180-1193, Oct. 1996.

[11] P.Lippens, J.van Meerbergen, W.Verhaegh, A.van der Werf, "Allocation of Multiport Memories for Hierarchical Data Streams", *Proc. IEEE Int. Conf. Comp.-Aided Design*, pp. 728-735, Santa Clara, Nov. 1993.

[12] L.Ramachandran, D.Gajski, V.Chaiyakul, "An algorithm for array variable clustering", *Proc. European Design and Test Conf.*, pp. 262-266, Paris, Mar. 1994.

[13] F.Balasa, F.Catthoor, H.De Man, "Dataflow-driven memory allocation for multi-dimensional processing systems", *Proc. IEEE Int. Conf. Comp. Aided Design*, San Jose CA, Nov. 1994.

[14] L.Stok, "Data path synthesis", *Integration, the VLSI journal*, Vol 18, pp. 1-71, June 1994.

[15] E.De Greef, F.Catthoor, H.De Man, "Array placement for storage size reduction in embedded multimedia systems", *Proc. of 11th Int. Conf. on Application-specific Systems, Architectures and Processors*, pp.66-75, Zurich, Switzerland, July 1997.

[16] T. Komarek, P. Pirsch, "Array Architectures for Block Matching Algorithms", IEEE Transactions on Circuits and Systems, vol 36, Oct. 1989.

Sven Wuytack was born in Leuven, Belgium, in 1970. He received the Electrical Engineering degree from the Katholieke Universiteit Leuven, Belgium, in 1993. He has been a Ph.D. degree candidate at the Inter-university Micro-Electronics Center (IMEC), Heverlee, Belgium since 1993. His research interests include system and architecture-level power optimization, mainly oriented towards memory organizations, and memory management in general. The major target application domains where this research is relevant are data structure dominated modules in telecom networks and real-time signal and data processing algorithms in image, video, and end-user telecom applications.

Jean-Philippe Diguet was born in France in 1969. He received the Engineering degree in Electronics from ESEO, France, in 1992 and the Ph.D. degree in Signal Processing from University of Rennes in 1996. He was working toward his thesis with the Lasti lab, Lannion, France. Then he joined IMEC in 1997 for a postdoctoral year supported by the French government fellowship *LAVOISIER*. He is now an assistant professor of electrical engineering at UBS university, Lorient, France and a member of the LESTER lab. His current research interests are in high level estimations for system design methodologies applied to signal-processing and telecommunication domains.

Francky Catthoor received the engineering degree and a Ph.D. in electrical engineering from the Katholieke Universiteit Leuven, Belgium in 1982 and 1987 respectively. From September 1983 till June 1987 he has been a researcher in the area of VLSI design methodologies for Digital Signal Processing, with Prof. Hugo De Man and Prof. Joos Vandewalle as Ph.D. thesis advisors. Since 1987, he has headed several research domains in the area of high-level and system synthesis techniques and architectural methodologies, all within the VLSI System Design Methodology (VSDM) division at the Inter-university Micro-Electronics Center (IMEC), Heverlee, Belgium. He is assistant professor at the EE department of the K.U.Leuven since 1989.

His current research activities belong to the field of architecture design methods and system-level exploration for area and power, mainly oriented towards memory management and global data transfer optimization. The major target application domains are real-time signal and data processing algorithms in image, video and end-user telecom applications, and data structure dominated modules in telecom networks. In 1986 he received the Young Scientist Award from the Marconi International Fellowship. In 1995 he has become an associate editor for the IEEE Trans. on VLSI Systems.

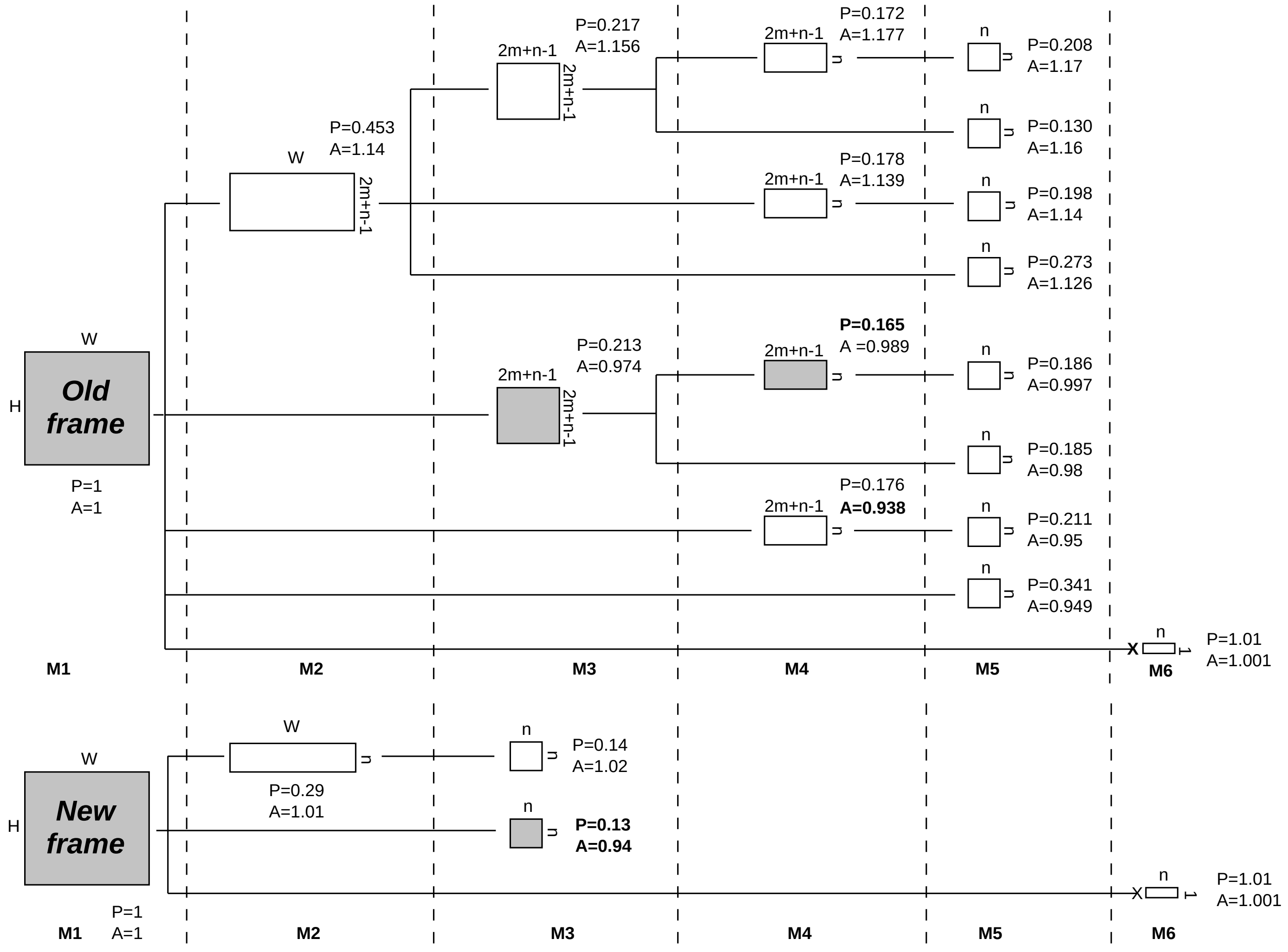


Fig. 8. Search trees for data reuse decision.

Hugo J. De Man was born in Boom, Belgium on September 19, 1940. He received the electrical engineering degree and the Ph.D. degree in Applied Sciences from the Katholieke Universiteit Leuven, Heverlee, Belgium, in 1964 and 1968, respectively. In 1968 he became a member of the staff of the Laboratory for Physics and Electronics of Semiconductors at the University of Leuven, working on device physics and integrated circuit technology. From 1969 to 1971 he was at the Electronic Research Laboratory, University of California, Berkeley, as an ESRO-NASA Postdoctoral Research Fellow, working on Computer-Aided Device and Circuit Design. In 1971 he returned to the University of Leuven as a Research Associate of the NFWO (Belgian National Science Foundation).

In 1974 he became a Professor at the University of Leuven. During the winter quarter of 1974-1975 he was a Visiting Associate Professor at the University of California, Berkeley. He was an Associate Editor for the IEEE Journal of Solid-State Circuits from 1975-1980 and was European Associate Editor for the IEEE Transactions on CAD from 1982 to 1985. He received a Best Paper Award at the ISSCC of 1973 on Bipolar Device Simulation and at the 1981 ESSCIRC conference for work on an integrated CAD system. In 1986 he received together with L. Claesen the Best paper Award in CAD from the ICCD-86 Conference and in 1987 for best publication in 1987 in the International Journal of Circuit Theory and Applications.In 1986 he also became fellow of the IEEE. In 1989 he received the Best Paper Award at the Design Automation Conference.

From 1984 to 1995 he was Vice-President of the VLSI systems design group of IMEC (Leuven, Belgium), where the actual field of this research division is design methodologies for Integrated Systems for telecommunication. Examples are spread spectrum and ATM components design. Research of this group has been at the basis of the EDC-Mentor Graphics DSP-Station.

Since 1995 he became a Senior Research Fellow of IMEC responsible for research in system design technologies. Prof. De Man is a fellow of IEEE, a corresponding member of the Royal Academy of Sciences, Belgium and a member of the Royal Flemish Engineering Society (KVIV).