Code Transformations for Data Transfer and Storage Exploration Preprocessing in Multimedia Processors

Francky Catthoor

Koen Danckaert

Sven Wuytack
IMEC

Nikil D. Dutt
University of California, Irvine

Platform-independent source code transformations can greatly help alleviate the data-transfer and storage bottleneck. This article covers global data-flow, loop, and data-reuse-related transformations, and discusses their effect on data transfer and storage, processor partitioning, and parallelization.

The data-transfer and storage bottleneck in modern processor architectures is a very important and timely problem, as discussed in another article in this issue ("Random-Access Data Storage Components in Customized Architectures," by L. Nachtergaele, F. Catthoor, and C. Kulkarni). This problem is especially relevant in embedded applications, where cost issues such as memory footprint and power consumption are vital. In this article, we show how source-to-source code transformations play a crucial role in the solution of this problem for multimedia and telecommunications applications.

Many of these code transformations can be platform-independent if they are defined carefully, in a well-chosen order.[1,2] This very useful property lets us apply them to a given application's code before we have any knowledge of platform architecture parameters such as memory size, communication scheme, and even processor type. Although the resulting optimized code behaves better on any of the modern platforms, passing it through a platform-dependent stage produces further cost-performance improvements. These optimizations are especially useful when the target is an at least partly customizable memory organization based on embedded DRAM and SRAM. Benini and De Micheli[3] provide a good overview of general related work on system-level transformations, focusing especially on reducing power consumption.

Code-rewriting techniques for access locality and regularity

Code-rewriting techniques, consisting of loop and dataflow transformations, are essential to modern optimizing and parallelizing compilers. They function mainly to enhance temporal and spatial locality for cache performance, and to expose the algorithm's inherent parallelism to the outer loop nests (for asynchronous parallelism) or inner loop nests (for synchronous parallelism).[4-6] These techniques also find uses in communication-free data allocation techniques[7] and in communications optimizations in general.[8]

Thus, it is no surprise that these code-rewriting techniques are also very important in data transfer and storage (DTS), especially for embedded applications that eventually allow more customized memory organizations. The first optimization step of our code-rewriting script, for example, acts as an enabling step for more platform-dependent optimization steps, significantly reducing the required amount of storage and transfers and improving access behavior. Basically, global loop transformations increase the locality and regularity of the code's accesses. In an embedded context, this is clearly good for memory size (area) and memory accesses (power),[9,10] but of course also for performance.[11]

The main distinction between our work here and the vast amount of earlier related work in the compiler literature is that we focus particularly on these transformations across all loop nests in the entire program.[9] When the scope is limited to one procedure or even one loop nest, code transformations can enhance locality (and parallelization possibilities) within that loop nest, but they do not improve the global data flow and associated buffer space present between the loop nests or procedures. So the first step of our process is really a precompilation phase that should occur before the more detailed, but also quite locally applied, conventional compiler loop transformations. This preprocessing also enables later steps in our global script that further reduce storage and transfers—steps addressing data reuse, memory hierarchy assignment, memory organization, and in-place mapping.

In addition, our script includes an initial global dataflow transformation step, by which we modify the algorithmic data flow to remove redundant data transfers (reads and writes to data that are only partially needed), which typically occur in the practical code. Dataflow transformations of a second class, however, also serve as enabling transformations for other steps in our global methodology because they break dataflow bottlenecks.[12]

Now let's turn to a very simple example to show how loop transformations can significantly reduce an algorithm's data storage and transfer requirements.

Example

This example consists of two loops: the first produces array $A[]$; the second reads $A[]$ to produce array $B[]$. Only the $B[]$ values must be kept afterward.

```
for (i=1; i<=N; ++i) {
  A[i] = …;
}
for (i=1; i<=N; ++i) {
  B[i] = f(A[i]);
}
```

If this algorithm were implemented directly, it would significantly increase storage and bandwidth requirements (assuming N is large), because all $A[]$ signals must be written to an off-chip background memory in the first loop and then read back in the second loop. Rewriting the code using a loop-merging transformation gives

```
for (i=1; i<=N; ++i) {
  A[i] = …;
  B[i] = A[i];
}
```

In this transformed version, the $A[]$ signals can be stored in registers because they are immediately consumed after they have been produced, and are no longer needed. In the overall algorithm, this significantly reduces storage and bandwidth requirements.

Cavity detection application

Next, we apply our more global, combined dataflow and loop transformation approach to a cavity detection application, which we use throughout this article as our main test vehicle. The cavity detection algorithm is a medical image-processing application that extracts contours from images to help physicians detect brain tumors. The initial algorithm consists of several functions, each with one image frame

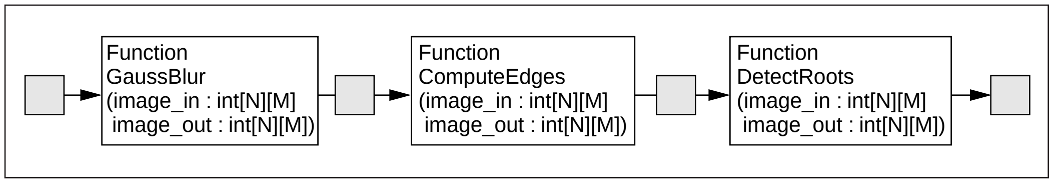


Figure 1. Initial cavity detection algorithm.

as input and one as output, as shown in Figure 1. "The cavity detection test vehicle" sidebar includes more information about how the application works.

Now let's demonstrate our global loop and dataflow transformation technique on the cavity detection code. In addition to the initial transformations, we'll apply data reuse and in-place mapping transformations later in this article; then we'll demonstrate the eventual effect of the loop transformations, which are in fact only enabling transformations.

For the initial cavity detection algorithm, as given in Figure 1, the data-transfer and storage requirements are considerable. The main reason is that each function reads an image from off-chip memory and writes the result back to this memory. Applying our global DTS exploration methodology heavily reduces these off-chip memories and transfers, resulting in far less off-chip data storage and transfers. We perform all steps of the process in an application-independent, systematic way.

Preprocessing and pruning. First, we rewrite the code in a three-level hierarchy. The top level, level 1, contains system-level functions between which no optimizations are possible. In level 2, we combine all relevant data-dominated computations into a single procedure, which is more easily analyzable than a set of procedures or functions. Level 3 contains all low-level (for example, mathematical) functions, which are not relevant for the data flow. Thus, we apply all further optimizations to the level 2 function. This is a key feature of our approach, because it exposes the available freedom for the actual exploration steps. The code shown later in this article is always extracted from this level 2 description.

Next, we analyze the data flow and replace all pointers with indexed arrays and then transform the code into single-assignment code that makes the flow dependencies fully explicit. This allows more aggressive dataflow and loop transformations. Furthermore, it leads to more freedom for our later data-reuse and in-place mapping stages. These stages will further compact the data in memory, more globally and efficiently than in the initial algorithm code.

Global dataflow transformations. In its initial version, the algorithm always performs initializations for the entire image frame. However, this is unnecessary; initializing only the borders saves us from a lot of costly memory accesses. In principle, designers are aware of this, but in practice original code usually contains many redundant accesses. By systematically analyzing the code for such redundancies (which our preprocessing phase makes practicable), we can identify all access redundancy in a controlled way.[12]

This step of our script also achieves our bottleneck removal objective. The initial algorithm contains a Reverse( ) function, computing the maximum value of the whole image; this computation creates a real bottleneck for DTS. From the perspective of computations, this bottleneck is almost negligible; but from the perspective of transfers, it is crucial, as the entire image must be written to off-chip memory before this computation and then read back afterward. However, in this application, a global dataflow transformation lets us actually remove this computation.

Indeed, the Reverse( ) function is a direct translation from an original system-level description of the algorithm, which reuses specific functions. It can be avoided by adapting the algorithm's next step, DetectRoots( ), through a dataflow transformation. Instead of "image_out[x] [y] = if (p > q).." where *p* and *q* are pixel elements produced by Reverse( ), we can write "image_out = if (−p < −q).." or

The cavity detection test vehicle

Figure A presents the code for the cavity detection application we used to test our code transformation procedure. The complete cavity detection algorithm contains additional functions, but for simplicity, we have left them out of this figure.

The application's first function is a horizontal and vertical Gauss-blurring step, which replaces each pixel by a weighted average of itself and its neighbors.

The second function, ComputeEdges(), takes each pixel and computes the difference between it and each of its eight neighbors; it then replaces the pixel with the maximum of these differences.

The last function, DetectRoots(), reverses the image. To do this, it computes the maximum value of the image and replaces each pixel with the difference between this maximum value and itself. Then, for each pixel, we look at whether any neighboring pixel is larger. If this is the case, the output pixel is false; otherwise it is true.

```
void GaussBlur (unsigned char image_in[N][M], unsigned char gxy[N][M]) {
     // Perform horizontal horizontal and vertical gauss blurring on each pixel
     unsigned char gx[N][M];
     for (y=0; y<M; ++y)
       for (x=0; x<N; ++x)
         gx[x][y] =...        // Apply horizontal gauss blurring
     for (y=0; y<M; ++y)
       for (x=0; x<N; ++x)
         gxy[x][y] = ...      // Apply vertical gauss blurring

void ComputeEdges (unsigned char gxy[N][M], unsigned char ce[N][M]) {
     // Replace every pixel with the maximum difference with its neighbors
     for (y=0; y<M; ++y)
       for (x=0; x<N; ++x)
         ce[x][y] = ...       // Replace pixel with the maximum
                              // difference with its neighbors

void Reverse (unsigned char ce[N][M], unsigned ce_rev[N][M]) {
     // Search for the maximum value that occurs : maxval
     for (y=0; y<M; ++y)
       for (x=0; x<N; ++x)
         maxval = ...         // Compute maximum value
     // Subtract every pixel value from this maximum value
     for (y=0; y<M; ++y)
       for (x=0; x<N; ++x)
         ce_rev[x][y] = maxval ce[x][y];

void DetectRoots (unsigned char ce[N][M], unsigned char image_out[N][M]) {
     unsigned char ce_rev[N][M];
     // Reverse image
     Reverse (ce, ce_rev);
     // image_out[x][y] is true if no neighbors are bigger than ce_rev[x][y]
     for (y=0; y<M; ++y)
       for (x=0; x<N; ++x)
         image_out[x][y] = ... // Is true if no neighbors are
                              // bigger than current pixel

void main(){
     unsigned char image_in[N][M], gxy[N][M], ce[N][M], image_out[N][M];
     // ... (read image)
     GaussBlur(image_in, gxy);
     ComputeEdges(gxy, ce);
     DetectRoots (ce, image_out);
```

Figure A. Pseudocode for cavity detection application.

"image_out = if(c – p < c – q).." where c = maxval is a constant. So instead of performing the Reverse( ) function and implementing the original DetectRoots( ), we omit the Reverse( ) function and instead implement the code shown in Figure 2 (next page).

```
void cav_detect (unsigned char image_in[N][M], unsigned char image_out[N][M]) {
   for (y=0; y<M; ++y)
      for (x=0; x<N; ++x)
         gx[x][y] = …        // Apply horizontal gauss blurring
   for (y=0; y<M; ++y)
      for (x=0; x<N; ++x)
         gxy[x][y] =         // Apply vertical gauss blurring
   for (y=0; y<M; ++y)
      for (x=0; x<N; ++x)
         ce[x][y] = …        // Replace pixel with the maximum
                             // difference with its neighbors
   for (y=0; y<M; ++y)
      for (x=0; x<N; ++x)
         image_out[x][y] =  // Is true if no neighbors are
                             // smaller than current pixel
}
```

Figure 2. Code for cavity detection algorithm after preprocessing and global dataflow transformations.

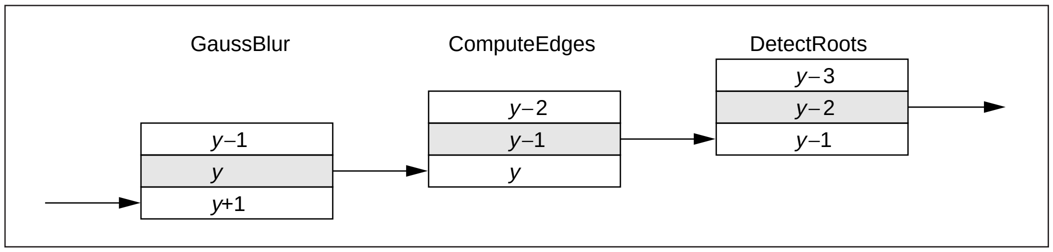


Figure 3. Computational flow for cavity detection algorithm after *y*-loop transformation.

Global loop transformations. The loop transformations of our methodology are relatively conventional, but we apply them far more globally than is conventional—over all relevant loop nests. Our more far-reaching approach is crucial to optimizing the global data transfers. Thus, the steering guidelines for our procedure clearly differ from the traditional compiler approach.

In our example, we first apply a global *y*-loop folding and merging transformation. Figure 3 shows the resulting computational flow, a line-based pipelining scheme. This is possible because, after the dataflow transformations, all the algorithm's computations are neighborhood computations. The code, from which we've omitted conditions on *y* to reduce the example's complexity, is now as shown in Figure 4.

We can apply a similar global loop-folding and merging transformation to the *x* loops too. Indeed, to perform the ComputeEdges( ) function on pixel (*x*, *y*), we don't have to wait until GaussBlur( ) has been performed on line *y* + 1 as a whole. In fact, ComputeEdges(x,y) can be executed right after GaussBlur(x+1,y+1) has been executed. This further increases the code's locality, and thus the possibilities for compile-time data reuse (not determined by the hardware cache controller). As a result, the computations are now performed according to a fine-grained (pixel-based) pipelining scheme (see Figure 5). The code is as shown in Figure 6, again omitting conditions on *x* and *y*.

The result of these transformations is a greatly improved locality, which the data reuse and in-place mapping steps exploit to reduce the application's storage and bandwidth requirements. We need only three line buffers per function for the intermediate memory between the algorithm's different steps; the initial version needed frame memories between the functions. In-place mapping further reduces this requirement to two line buffers per function. Figure 7 (page 76) shows the final results for the cavity detection application. The transfer results are largely platform independent, but

```
void cav_detect (unsigned char in_image[N][M], unsigned char out_image[N][M])
{
   for (y=0; y<M+3; ++y)
      for (x=0; x<N; ++x)
         gx[x][y] = …          // Apply horizontal gauss blurring
   for (x=0; x<N; ++x)
         gxy[x][y-1] = …       // Apply vertical gauss blurring
   for (x=0; x<N; ++x)
         ce[x][y-2] = …        // Replace pixel with the maximun
                               // difference with its neighbors
   for (x=0; x<N; ++x)
         image_out[x][y-3] = … // Is true if no neighbors are smaller
                               // than this pixel
}
```

Figure 4. Cavity detection algorithm following *y*-loop transformation.

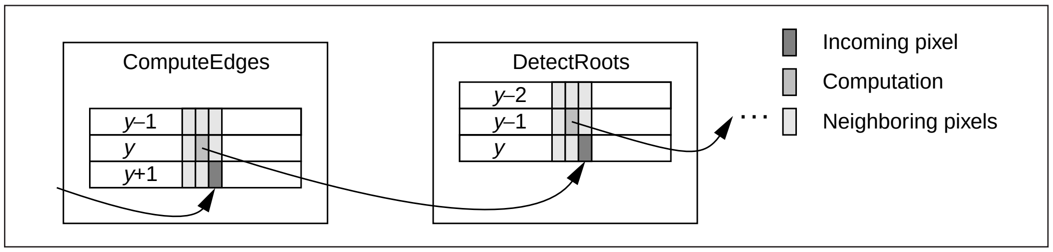


Figure 5. Flow for cavity detection algorithm after *x*-loop transformation.

```
void cav_detect (unsigned char in_image[N][M], unsigned char out_image[N][M]) {
   for (y=0; y<M+3; ++y)
      for (x=0; x<N+2; ++x) {
         gx[x][y] = …           // Apply horizontal gauss blurring
         gxy[x][y-1] = …        // Apply vertical gauss blurring
         ce[x-1][y-2] = …       // Replace pixel with the maximum
                                // difference with its neighbors
         image_out[x-2][y-3] =  // Is true if no neighbors
                                // are smaller than this pixel
      }
}
```

Figure 6. Cavity detection algorithm after *x*-loop transformation.

the execution time is instantiated for a Pentium II platform. However, we have obtained similar relative improvement factors on many other processors as well.[2]

Code-rewriting techniques to improve data reuse

An efficiently used memory hierarchy is of primary importance in optimizing data transfer and storage. To exploit such a memory hierarchy, the code to be mapped should expose maximal data reuse possibilities. To achieve this, the loop and dataflow transformations of our preprocessing step are essential, because they significantly improve the code's overall regularity and access locality. This is beneficial on its own, but achieving this regularity and locality also enables our script's next step, the data reuse step, which even further reduces transfers to the large background memories. During this step, we add hierarchical data reuse copies to the code, exposing the different levels of reuse that are inherently present (but not directly visible) in the transformed code.

This step of our procedure is clearly distinct from the conventional approach, where the hardware cache control determines the size and timing of these copies based on the avail-

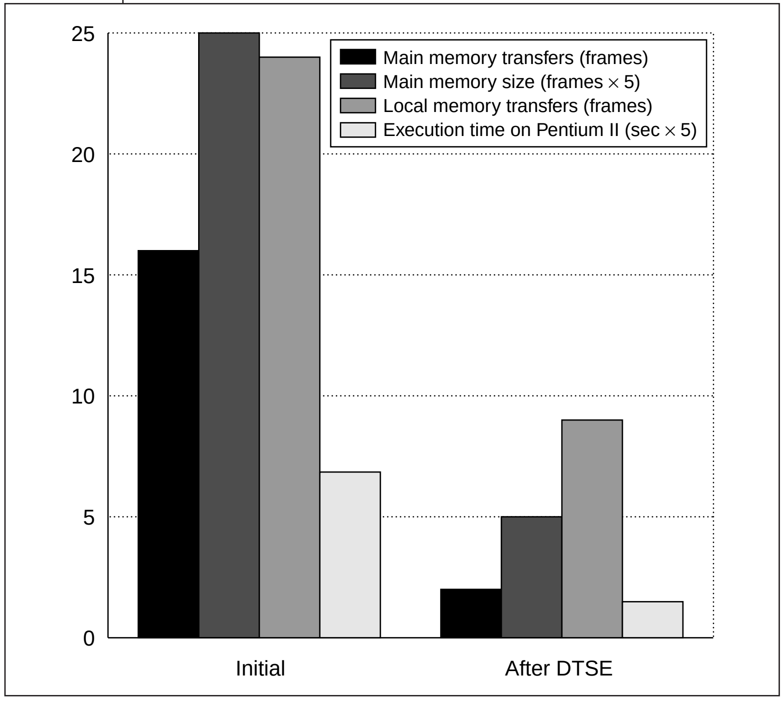


Figure 7. Results for cavity detection application.

able locality of access. In our approach, a global exploration of the data reuse copies is performed to globally optimize the size and timing of the copies in the code. A custom memory hierarchy can then be designed on which these copies can be mapped very efficiently.[13] However, even for a predefined memory hierarchy, as typically present in programmable processors, the newly derived code from this step implicitly steers the data reuse decisions and still greatly benefits system bus load, system power budget, and cache miss behavior.[14]

Adding hierarchical data reuse copies

Let's take a simple example first, one that is already the result of loop transformations:

```
for (i=0; i<N; ++i)
  for (j=0; j<=N-L; ++j) {
    b[i][j] = 0;
    for (k=0; k<L; ++k)
      b[i][j] += a[i][j+k];
}
```

When executed on a processor with a small cache, this code would perform far better than the initial code. To map the code on a custom memory hierarchy, however, we must know the optimal size of the hierarchy's different levels. To this end, we add signal copies (buffers) to the code to make the data reuse explicit. This results in the following code, omitting the initialization for simplicity:

```
int a_buf[L];
int b_buf;
for (i=0; i<N; ++i)
  {initialize a_buf}
  for (j=0; j<=N-L; ++j) {
    b_buf = 0;
    a_buf[(j+L-1)%L] =
      a[i][j+L-1];
    for (k=0; k<L; ++k)
      b_buf += a_buf[(j+k)%L];
    b[i][j] = b_buf;
  }
```

This code contains two data reuse buffers:

- a_buf [] (*L* words), for the *a* [] [] signals and
- b_buf (1 word), for the *b* [] [] signals

In general, more than one level of data reuse buffers is possible for each signal. We have developed a formal methodology[13] that arranges all possible buffers in a tree. For each signal, such a tree is generated, and an optimal alternative is selected.

Cavity detection illustration

Let's return to the cavity detector, taking the code from the stage when the dataflow and loop transformations we described earlier have introduced more locality and possibilities for data reuse.

From Figures 3 and 5, which illustrate the effect of global loop transformations, we can identify two levels of data reuse: line buffers and pixel buffers. As explained earlier, we now insert explicit buffers into the code to optimally exploit the potential data reuse.

Line buffers. For each function, we can implement a buffer of three lines, such that the line being processed is stored with the previous line and the next line:

```
void cav_detect   unsigned char image_in[M][N],
                  unsigned char image_out[M][N])
{
  unsigned char gauss_x_lines[3][N];
  unsigned char gauss_xy_lines[3][N];
  unsigned char comp_edge_lines[3][N];
  for (y=0; y<M+3; ++y) {
     for (x=0; x<N+2; ++x) {
        gauss_x_lines[y % 3][x] = …         // Apply horizontal gauss blurring
        gauss_xy_lines[(y-1) % 3][x] = …    // Apply vertical gauss blurring
        comp_edge_lines[(y-2) % 3][x-1] = … // Replace with the maximum
                                            // difference with neighbors
        image_out[y-3][x-2] = …             // Is true if no neighbors
                                            // are smaller than
                                            // comp_edge_lines[(y-3) % 3][x-2];
     }
  }
}
```

Figure 8. Cavity detection algorithm after implementation of line buffers.

- The horizontal gauss blurring is done on an incoming pixel, and the result is stored in buffer gauss_x_lines[][].
- Next, the vertical gauss blurring is performed on one pixel in this buffer, and the result is stored in gauss_xy_lines[][].
- Then ComputeEdges( ) can be executed in gauss_xy_lines[][], the result of which is stored in comp_edge_lines[][].
- Finally, DetectRoots( ) is executed in comp_edge_lines[][], and the resulting pixel is stored in the output image.

The resulting code is quite complex after the insertion of all necessary preambles and postambles or conditions; therefore, Figure 8 gives only the code for the heart of the loop nest.

Pixel buffers. In Figure 5, we can also identify a second level of data reuse—the pixels in the neighborhood of the pixel being processed:

- For horizontal gauss blurring, the algorithm can implement a buffer of three pixels called in_pixels[], storing the last used values of the incoming image.
- For vertical gauss blurring, no such buffer is possible. However, the output of this step is stored into three-by-three pixel buffer gauss_xy_pixels[][].
- This buffer is used in ComputeEdges( ), and the result of that step is again stored in a three-by-three buffer, comp_edge_pixels[][].
- Finally, DetectRoots( ) is performed on this buffer, and the result is stored in the output image.

Omitting initializations, preambles, and postambles, the final code is as shown in Figure 9 (next page).

Results. We processed the different versions of the cavity detection code presented so far, with our Atomium access counting tool; Table 1 presents our results. The final version is far better suited for mapping onto a custom memory hierarchy. Vandecappelle and colleagues[15] discuss tools that can help produce such a customized memory organization based on given specifications.

Parallelism: choosing between task and data levels

Parallelization is a standard way to improve the performance of a system when a single processor cannot do the job. However, the best way to parallelize a system is not obvious, because many possibilities exist, and they can differ greatly in performance. The same is true for the impact of parallelization on data storage and transfers. Most of the research effort in this area addresses the problem of parallelization and processor partitioning.[4,16] However, these approaches do not take into account the background storage-related cost when applied to data-dominated applications; they optimize

```
void cav_detect       (unsigned char image_in[M][N],
                       unsigned char image_out[M][N])
   {
   unsigned char gauss_x_lines[3][N];
   unsigned char gauss_xy_lines[3][N];
   unsigned char comp_edge_lines[3][N];
   unsigned char in_pixels[3];
   unsigned char gauss_xy_pixels[3][3];
   unsigned char comp_edge_pixels[3][3];
   for (y=0; y<M+3; ++y) {
      for (x=0; x<N+3; ++x) {
         in_pixels[x % 3] = image_in[y][x];
         gauss_x_lines[y % 3][x-1] = … // Apply horizontal gauss blurring
         gauss_xy_pixels[(y-1) % 3][(x-1) % 3]
            = gauss_xy_lines[(y-1) % 3][x-1] = …
                                // Apply vertical gauss blurring
         comp_edge_pixels[(y-2) % 3][(x-2) % 3]
            = comp_edge_lines[(y-2) % 3][x-2] = …
                                // Replace with the maximum difference with
                                   neighbors
         image_out[y-3][x-3] = …// Is true if no neighbors are smaller than
                                // comp_edge_pixels[(y-3) % 3][(x-3) % 3];
      }
   }
}
```

Figure 9. Cavity detection algorithm after implementation of pixel buffers.

Table 1. Results of loop and data-reuse transformations for cavity detection application. Trafo 1 is loop transformations + line buffer reuse; Trafo 2 is loop transformations + line pixel buffer reuse.

Code version	Access to images	Line buffers	Pixel buffers
Initial	84,954,834	0	0
Trafo 1	5,229,068	29,346,281	0
Trafo 2	2,618,880	11,787,272	37,208,041

only speed, not power or memory size. More recent methods[17] usually consider data communication between processors, but they use an abstract model—a virtual processor grid, which has no relation with the final number of processors and memories. Our group described a first approach to more global memory optimization in the context of parallel processors;[18,19] we showed that extensive loop reorganization must be applied before parallelization.

Here, we focus on the parallelization itself; in another article, we discuss the effect of code transformations on processor partitioning in a hardware-software codesign context.[20] The two major types of parallelism are task and data. In task parallelism, we assign an application's different subsystems to different processors. In data parallelism, each processor executes the entire algorithm, but only on a part of the data. In addition, hybrid task-data parallel alternatives are possible. When data transfer and storage optimization is an issue, even more attention must be paid to the way the algorithm is parallelized. We present two examples to illustrate this point:

- The cavity detection algorithm, the medical image-processing application we discussed earlier in this article; and
- Quadtree Structured Difference Pulse Code Modulation (QSDPCM), an algorithm for video compression.

Cavity detection

Now let's look at some ways to parallelize the cavity detection algorithm, assuming we need a speedup of about 3. In practice, most image-processing algorithms do not require large speedups, so this is more realistic than massive speedup. We parallelize two versions of the algorithm (initial and globally optimized) in two ways: first using task parallelization, and then using data parallelization.

Initial algorithm. Applying algorithmic parallelization to the initial algorithm leads to a coarse-grained pipelining solution (at the level of the image frames). This works well for load balancing on a three-processor system, but it is clearly unacceptable if we have efficient memory management in mind. Data parallelism is a better choice. Neglecting some border effects, the cavity detection algorithm lends itself very well to this kind of parallelism, as there are only neighbor-to-neighbor dependencies. Thus, each processor can work more or less independently from the others except at the boundaries, where some idle synchronization and transfer cycles occur. Each processor still needs two frame buffers, but now these buffers are only one-third of a frame. So the data-parallel solution requires the same amount of buffers as the monoprocessor solution.

Globally optimized algorithm. In this case, applying algorithmic parallelization involves assigning each step of the algorithm to a different processor, but now we arrive at a fine-grained pipelining solution. Processor 1 has a buffer of two lines: $y - 1$ and y. Line $y + 1$ enters the processor as a scalar stream; the GaussBlur-step can be performed synchronously on line y, the result of which can be sent to the second processor as a scalar stream. Processor 2 can concurrently (and synchronously) apply the ComputeEdges step to line $y - 1$, and so on. Hence, we need only a buffer of two lines per processor, or six in total. This is the same number of buffers that we needed for the monoprocessor case. Thus, we have improved performance without sacrificing storage and transfer overhead (which translates to area and power overhead).

When we use data parallelism with the globally optimized version, we need six FIFO line buffers per processor, or 18 total. Buffer length depends on the way we partition the image. If we use row-wise partitioning (the first processor processes the upper third of the image and so on), the 18 buffers are the same size as in the monoprocessor case. Column-wise partitioning yields better results: We still need 18 buffers, but their length is only a third of a line. Because the access is not FIFO compatible, the buffers must be organized as SRAMs, which are far more expensive than FIFO buffers.

Table 2. Parallelization results, cavity detection application.

Algorithm version	Parallelism type	Frame memories	Frame transfers	Line buffers
Initial	Data	2	30	0
Initial	Task	6	30	0
Transformed	Data	0	0	6 SRAM
Transformed	Task	0	0	6 FIFO

Table 3. Parallelization results, QSDPCM.

Algorithm version	Partitioning	Area	Power
Initial	Pure data	1	1
	Pure task	0.92	1.33
	Modified task	0.53	0.64
	Hybrid 1	0.45	0.51
	Hybrid 2	0.52	0.63
Reorganized	Pure task	0.0041	0.0080
	Modified task	0.0022	0.0040
	Hybrid 1	0.0030	0.0050
	Hybrid 2	0.0024	0.0045

Table 2 summarizes the results. The algorithmically parallelized version was the optimal solution here. Its load balancing is less ideal than that of the data-parallel version, but it is important to trade off performance and DTS exploration. For example, avoiding a 32-Kbit buffer by using an extra processor would be advantageous even if this processor were idle 90% of the time (a very bad load balance), because the cost of this extra processor in terms of area and power is less than the cost of a 32-Kbit on-chip memory.

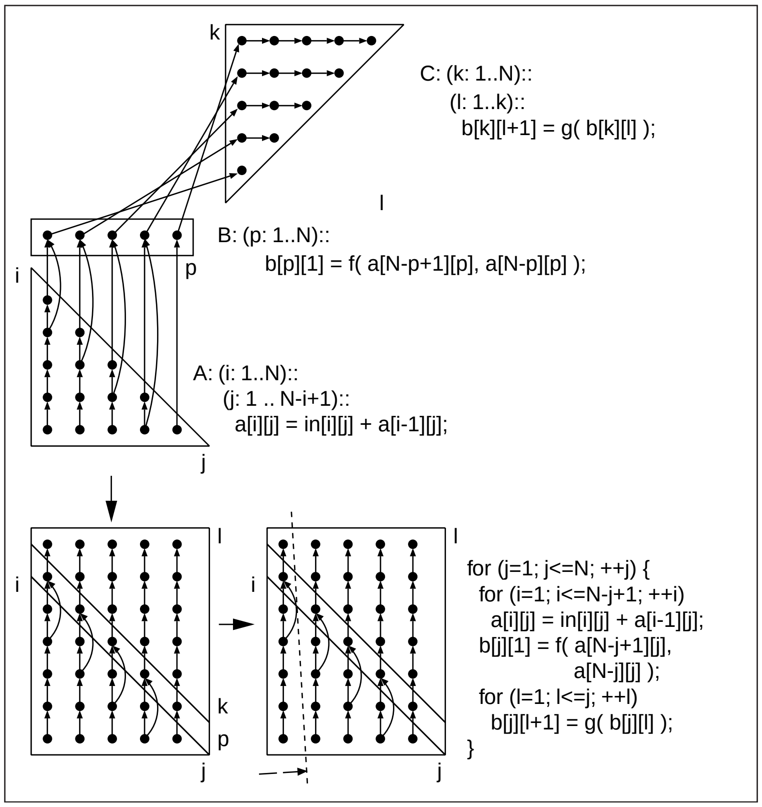
Elsewhere, we include a more elaborate demonstration of parallelization, using QSDPCM for interframe video compression.[18,19] Table 3 shows an overview of the results we achieved. We obtained the estimated area and power figures using a Motorola model. (Because this model is proprietary, we give only relative values in this article.) The rankings for the different alternatives (initial and reorganized) are clearly distinct. For the reorganized description, the task-level-oriented hybrids are better because this kind of parti-

A methodology for automating loop transformations

To automate the proposed loop transformations, we use a polytope model.[1,2] In this model, an *n*-dimensional polytope geometrically represents each *n*-level loop nest. Figure B gives an example, in which the loop nest at the top is two-dimensional and has a triangular polytope representation, because the inner loop bound is dependent on the value of the outer loop index. The arrows in the figure represent data dependencies, and they indicate the direction of the data flow. The order in which the iterations are executed can be represented by an ordering vector, which traverses the polytope.

To perform global loop transformations, we have developed a two-phase approach. In the first phase, all polytopes are placed in one common iteration space. During this phase, the polytopes are merely considered geometrical objects without execution semantics. In the second phase, a global ordering vector is defined in this global iteration space. Figure B gives an example of this methodology. At the top, we see the initial specification of a simple algorithm; at the bottom left, the polytopes of this algorithm are placed in the common iteration space in an optimal way; at the bottom right, an optimal ordering vector is defined and the corresponding code is derived. We give more details in another work.[3]

Most existing loop transformation strategies work directly on the code. Moreover, they typically work on single loop nests, thereby omitting the global transformations that are crucial for storage and transfers. Many of these techniques also consider the body of each loop nest as one union;[4] we have a polytope for each statement, allowing more aggressive transformations. Affine-by-statement techniques constitute an exception to this distinction; these techniques transform each statement separately. However, our two-phase approach allows a still more global view vis-à-vis data transfer and storage issues.



```
C: (k: 1..N)::
    (l: 1..k)::
      b[k][l+1] = g( b[k][l] );

B: (p: 1..N)::
      b[p][1] = f( a[N-p+1][p], a[N-p][p] );

A: (i: 1..N)::
    (j: 1 .. N-i+1)::
      a[i][j] = in[i][j] + a[i-1][j];

for (j=1; j<=N; ++j) {
  for (i=1; i<=N-j+1; ++i)
    a[i][j] = in[i][j] + a[i-1][j];
  b[j][1] = f( a[N-j+1][j],
               a[N-j][j] );
  for (l=1; l<=j; ++l)
    b[j][l+1] = g( b[j][l] );
}
```

Figure B. Example of loop transformation methodology that can be automated.

References

1. F. Franssen et al., "Modeling Multi-Dimensional Data and Control flow," *IEEE Trans. VLSI Systems,* vol. 1, no. 3, Sept. 1993, pp. 319-327.
2. F. Catthoor et al., *Custom Memory Management Methodology—Exploration of Memory Organization for Embedded Multimedia System Design,* Kluwer Academic Publishers, Norwell, Mass., 1998.
3. K. Danckaert, F. Catthoor, and H. De Man, "A Preprocessing Step for Global Loop Transformations for Data Transfer and Storage Optimization," *Proc. Int'l Conf. Compilers, Architectures, and Synthesis for Embedded Systems* (CASES 2000), ACM Press, New York, 2000, pp. 34-40.
4. A. Darte, T. Risset, and Y. Robert, "Loop Nest Scheduling and Transformations," *Environments and Tools for Parallel Scientific Computing,* J.J. Dongarra et al., eds., Advances in Parallel Computing 6, North Holland, Amsterdam, 1993, pp. 309-332.

tioning maintains the balance between double buffers (in task-level partitioning) and replicates of array signals with the same functionality in different processors (in data-level partitioning). However, optimal partitioning highly depends on the number of the different submodules of the application and on the number of processors.

Regarding storage of the intermediate array signals, the results of partitioning based on the initial description reduce this memory size when partitioning becomes more data oriented. This is especially visible for the QSDPCM algorithm. This size is smaller for the first hybrid partitioning (245 Kbits), which is more data oriented than the second hybrid partitioning (282 Kbits) and the task-level partitioning (287 Kbits). In terms of the number of memory accesses to the intermediate signals, the situation is simpler. The number of accesses to these signals always decreases as the partitioning becomes more data oriented. ■

References

1. F. Catthoor et al., "Data Transfer and Storage Architecture Issues and Exploration in Multimedia Processors," *Programmable Digital Signal Processors: Architecture, Programming, and Applications*, Y.H. Yu, ed., Marcel Dekker Inc., New York, 2000.
2. K. Danckaert, F. Catthoor, and H. De Man, "A Preprocessing Step for Global Loop Transformations for Data Transfer and Storage Optimization," *Proc. Int'l Conf. Compilers, Architectures, and Synthesis for Embedded Systems* (CASES 2000), ACM Press, New York, 2000, pp. 34-40.
3. L. Benini and G. De Micheli, "System-Level Power Optimization Techniques and Tools," *ACM Trans. Design Automation for Embedded Systems (TODAES),* vol. 5, no. 2, Apr. 2000, pp.115-192.
4. S. Amarasinghe et al., "The SUIF Compiler for Scalable Parallel Machines," *Proc. 7th SIAM Conf. Parallel Processing for Scientific Computing,* SIAM, Philadelphia, 1995.
5. U. Banerjee et al., "Automatic Program Parallelisation," *Proc. IEEE,* vol. 81, no. 2, pp. 211-243, Feb. 1993.
6. M. Wolfe, *High-Performance Compilers for Parallel Computing,* Addison-Wesley, Reading, Mass., 1996.
7. T-S. Chen and J-P. Sheu, "Communication-Free Data Allocation Techniques for Parallelizing Compilers on Multicomputers," *IEEE Trans. Parallel and Distributed Systems,* vol. 5, no. 9, Sept. 1994, pp. 924-938.
8. M. Gupta, E. Schonberg, and H. Srinivasan, "A Unified Framework for Optimizing Communication in Data-Parallel Programs," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 7, July 1996, pp. 689-704.
9. F. Franssen et al., "Control Flow Optimization for Fast System Simulation and Storage Minimization," *Proc. 5th ACM/IEEE European Design and Test Conf.,* IEEE CS Press, Los Alamitos, Calif., 1994, pp. 20-24.
10. E. De Greef, F. Catthoor, and H. De Man, "Mapping Real-Time Motion Estimation Type Algorithms to Memory-Efficient, Programmable Multi-Processor Architectures," *Microprocessors and Microprogramming,* Oct. 1995, pp. 409-423.
11. K. Masselos et al., "A Performance Oriented Use Methodology of Power Optimizing Code Transformations for Multimedia Applications Realized on Programmable Multimedia Processors," *Proc. IEEE Workshop on Signal Processing Systems (SIPS),* IEEE Press, Piscataway, N.J., 1999, pp. 261-270.
12. F. Catthoor et al., "System-Level Dataflow Transformations for Power Reduction in Image and Video Processing," *Proc. Int'l Conf. Electronic Circuits and Systems,* IEEE Press, Piscataway, N.J., 1996, pp. 1025-1028.
13. S. Wuytack et al., "Formalized Methodology for Data Reuse Exploration for Low-Power Hierarchical Memory Mappings," *IEEE Trans. VLSI Systems,* vol. 6, no. 4, Dec. 1998, pp. 529-537.
14. C. Kulkarni, F. Catthoor, and H. De Man, "Cache Transformations for Low Power Caching in Embedded Multimedia Processors," *Proc. Int'l Parallel Processing Symp.* (IPPS 98), IEEE Press, Piscataway, N.J., 1998, pp. 292-297.
15. A. Vandecappelle et al., "Global Multimedia System Design Exploration Using Accurate Memory Organization Feedback," *Proc. 36th ACM/IEEE Design Automation Conf.,* ACM Press, New York, 1999, pp. 327-332.
16. C. Polychronopoulos, "Compiler Optimizations for Enhancing Parallelism and Their Impact on the Architecture Design," *IEEE Trans. Computers,* vol. 37, no. 8, Aug. 1988, pp. 991-1004.
17. A. Agarwal, D. Krantz, and V. Nataranjan, "Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems,* vol. 6, no. 9, Sept. 1995, pp. 943-962.
18. K. Danckaert, F. Catthoor, and H. De Man, "Sys-

tem-Level Memory Management for Weakly Parallel Image Processing," *Proc. EuroPar Conf.,* Lecture Notes in Computer Science Series, vol. 1124, Springer Verlag, New York, 1996, pp. 217-225.

19. K. Masselos et al., "A Methodology for Power Efficient Partitioning of Data-Dominated Algorithm Specifications within Performance Constraints," *Proc. IEEE Int'l Symp. Low Power Design,* IEEE CS Press, Los Alamitos., Calif., 1999, pp. 270-272.
20. F. Catthoor et al., *Custom Memory Management Methodology—Exploration of Memory Organization for Embedded Multimedia System Design,* Kluwer Academic Publishers, Norwell, Mass., 1998.

Koen Danckaert is pursuing a PhD at the Inter-University Micro-Electronics Center (IMEC), Leuven, Belgium. His research activities primarily target global loop transformations for data transfer and storage exploration for parallel multimedia applications. He received the Eng degree in computer science from Catholic University in Leuven, Belgium.

Sven Wuytack is a researcher at IMEC. His research interests include system and architecture-level power optimization, mainly oriented toward memory organization, and memory management in general. Wuytack has an Eng degree and a PhD, both in electrical engineering from Catholic University in Leuven, Belgium.

The biography of **Francky Catthoor** appears on page 4 of this issue. The biography of **Nikil D. Dutt** appears on page 68.

■ Direct questions and comments about this article to Francky Catthoor, IMEC, Kapeldreef 75, B3001 Leuven, Belgium; catthoor@imec.be.