VLSI DESIGN
2001, Vol. 12, No. 2, pp. 101-111
Reprints available directly from the publisher
Photocopying permitted by license only

A Systematic Approach to Reduce the System Bus Load and Power in Multimedia Algorithms

KOEN DANCKAERT[a,*], CHIDAMBER KULKARNI[a,†], FRANCKY CATTHOOR[a,‡], HUGO DE MAN[a,¶] and VIVEK TIWARI[b,§]

[a]*IMEC, Kapeldreef 75, B-3001 Leuven, Belgium;*
[b]*Intel Corp., 3600 Juliette Ln., Santa Clara, CA 95052, USA*

(Received 20 June 2000; In final form 3 August 2000)

Multimedia algorithms deal with enormous amounts of data transfers and storage, resulting in huge bandwidth requirements at the off-chip memory and system bus level. As a result the related energy consumption becomes critical. Even for execution time the bottleneck can shift from the CPU to the external bus load. This paper demonstrates a systematic software approach to reduce this system bus load. It consists of source-to-source code transformations, that have to be applied before the conventional ILP compilation. To illustrate this we use a cavity detection algorithm for medical imaging, that is mapped on an Intel Pentium® II processor.

Keywords: Multimedia; Memory optimization; Low-power design; Bandwidth reduction; Bus load reduction; Medical imaging

1. INTRODUCTION

Multimedia systems such as medical image processing and video compression algorithms, typically use a very large amount of data storage and transfers. This causes huge bandwidth requirements at the off-chip memory and system bus level. The required memories and bus transfers will consume a lot of power [8, 14], which is especially a problem in embedded systems. It has been shown that a dominant part of the power in embedded multimedia systems is consumed by

*Corresponding author. Also Ph.D. student at Katholike Univ. Leuven. Tel.: +32-16281407, Fax: +32-16281515, e-mail: Koen.Danckaert@imec.be
†Also Ph.D. student at Katholike Univ. Leuven. Tel.: +32-16281407, Fax: +32-16281515, e-mail: Chidamber.Kulkarni@imec.be
‡Also prof. at Katholieke Univ. Leuven. Tel.: +32-16281202, Fax: +32-16281515, e-mail: Francky.Catthoor@imec.be
¶Also prof. at Katholieke Univ. Leuven. Tel.: +32-16281201, Fax: +32-16281515, e-mail: Hugo.DeMan@imec.be
§e-mail: Vivek.Tiwari@intel.com

data storage and transfers (as opposed to the computations which consume much less) [23], and even on the CPU chip itself it is a major contribution [10, 20].

Additionally, in many cases the performance bottleneck for this kind of applications lies not in the CPU, but in the board containing the external system bus and the off-chip memories. Even when the bottleneck is still with the CPU computations for a single application, the huge bandwidth requirements will cause performance problems when this application has to share the system bus with other applications or background processes running on the same board (containing one or more processors).

In this paper, we demonstrate our DTSE (Data Transfer and Storage Exploration) methodology, which is a software approach to alleviate the system bus load problem. In this approach, several source-to-source code transformations to optimize the global memory accesses of an application are applied in a systematic way.

In the past we have developed such a methodology for customizable hardware systems [4], where the memory hierarchy can be designed for a specific application. In this paper we apply the relevant steps of this methodology (with adaptations) to a programmable system with a fixed memory hierarchy, to show that it reduces the system bus load on programmable systems too.

Because most of the basic methodology steps are explained in [4] (for customizable systems), we will not focus in detail on the steps themselves. Instead we demonstrate the methodology by applying it on a representative application (a cavity detector for medical imaging), with a programmable processor (a 450 MHz Pentium® II with 512K L2-cache) as target architecture. In Section 2, we will give a short overview of our methodology. In Section 3, it is applied to the cavity detection application. Section 4 gives some other experimental results, and Section 5 discusses the current status of our work. Conclusions are drawn in Section 6.

2. OVERVIEW OF THE METHODOLOGY

In [4], we have thoroughly described our DTSE methodology for data transfer and storage exploration, for customizable systems. A number of data storage and transfer optimization steps are included in this methodology:

1. Memory oriented data-flow analysis and pruning of the initial system specification.
2. Global data-flow and loop transformations of the system's description for reduction of the required background memories and accesses but also to enable further optimization steps.
3. Data reuse transformations to exploit a distributed memory hierarchy more effectively in the algorithm. In this step, additional data copies are introduced in a judicious way.
4. Storage cycle budget distribution to determine the bandwidth requirements and the balancing of the available cycle budget over the different memory accesses in the algorithmic specification.
5. Memory allocation and signal to memory/port assignment, to determine the necessary number of memory units and their type (if freedom is left, *e.g.*, in the off-chip memory organization). The goal of this step is to produce a netlist of memory units from a memory library as well as an assignment of signals to the available memory units.
6. In-place mapping. Here the aim is to increase the cache hit rate and reduce the memory size by storing signals with non-overlapping lifetimes in the same physical memory location. Conventionally memory reuse is only applied for variables with different scopes. We apply aggressive in-place transformations also when arrays have partly overlapping lifetimes.
7. Main memory data layout organization. Here we rearrange the data in memory to reduce cache conflict misses as much as possible.

Because Steps 4 and 5 are not relevant when the target architecture is a programmable processor with a fixed memory hierarchy, they will not be applied in this paper.

3. APPLICATION TO A REPRESENTATIVE EXAMPLE

3.1. The Initial Cavity Detection Algorithm

The cavity detection algorithm extracts edges from medical images, such that they can more easily be interpreted by physicians [2]. The initial algorithm consists of a number of functions, each of which has an image frame as input and one as output (see Fig. 1). The initial code is given in Figure 2.

We have made two types of measurements to compare the data transfer and storage requirements of different versions of the algorithm. First of all, to obtain precise feedback on the number of data accesses, we have developed an access counting tool (ATOMIUM), which counts all accesses for each array in an algorithm, and this in each function (or even on a finer granularity). This allows to locate the bottlenecks in terms of data storage and transfers. In Figure 6, we have summarized these results. Note that the distinction between main memory and local memory in this figure is made manually, by assigning each array to a level of the memory hierarchy. On a customizable system, this can also be done physically, but on a cache-based programmable system, these results do not directly indicate the real data transfers. To obtain these, we have used the Pentium® II event counters to count the number of L1 cache misses and the number of system bus memory transfers, as shown in Figure 7.

For the initial algorithm, we can see in these figures that the system bus bandwidth requirements are very high. The main reason is that each of the functions reads an image from off-chip memory, and writes the result back to this memory. After applying our DTSE methodology, these off-chip memories and transfers will be heavily reduced, resulting in much less off-chip data storage and transfers. Note that all these steps are performed in an application-independent systematic way.

3.2. Global Data-flow and Loop Transformations

During this step, the code is transformed to expose maximal locality and potential data reuse. This is done by applying data-flow and loop transformations. Because loop transformations are relatively well known for increasing locality and introducing parallelism [1, 9, 12, 22], we will not go into much detail, such that we can concentrate more on the other steps of our methodology. Note however that we we apply these transformations on a more global scale than conventionally done, to optimize the global data transfers which are most crucial for the system bus load [6].

First some redundant initializations are removed by a data-flow transformation. Then another data-flow transformation is applied to remove the computation of the maximum of the whole image (in the function *Reverse*()). Although this computation is negligible in terms of CPU time, it is very important in terms of data transfer and storage, because the whole image has to read from memory, and stored again, before the next function (*DetectRoots*()) can proceed. In this case however, this maximum computation can be eliminated by a global data-flow transformation.

Next, a loop folding and merging transformation is applied to the y-loops in the algorithm. As a

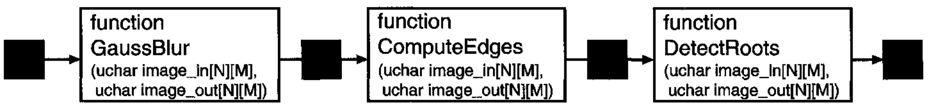


FIGURE 1 Initial cavity detection algorithm.

```
void GaussBlur (uchar image_in[M][N], uchar gauss_xy[M][N]) {
  uchar gauss_x[M][N]; // uchar stands for unsigned char
  for (y=0; y<M; ++y)
    for (x=0; x<N; ++x)
      gauss_x[y][x] = ... // Apply horizontal gauss blurring
  for (y=0; y<M; ++y)
    for (x=0; x<N; ++x)
      gauss_xy[y][x] = ... // Apply vertical gauss blurring
}

void ComputeEdges (uchar gauss_xy[M][N], uchar comp_edge[M][N]) {
  for (y=0; y<M; ++y)
    for (x=0; x<N; ++x)
      comp_edge[y][x] =... // Replace pix. with max. diff. with its neighbors
}

void Reverse (uchar comp_edge[M][N], uchar ce_rev[M][N]) {
  for (y=0; y<M; ++y)  // Search for the maximum value that occurs : maxval
    for (x=0; x<N; ++x)
      maxval = ...
  for (y=0; y<M; ++y)  // Subtract every pixelvalue from this maximum value
    for (x=0; x<N; ++x)
      ce_rev[y][x] = maxval - comp_edge[y][x];
}

void DetectRoots (uchar comp_edge[M][N], uchar image_out[M][N]) {
  uchar ce_rev[M][N];
  Reverse (comp_edge, ce_rev);  // Reverse image
  for (y=0; y<M; ++y)
    for (x=0; x<N; ++x)
      image_out[y][x] =... // is true if no neighbors are > than ce_rev[y][x]
}

void main () {
  uchar image_in[M][N], gauss_xy[M][N], comp_edge[M][N], image_out[M][N];
  GaussBlur(image_in, gauss_xy);
  ComputeEdges(gauss_xy, comp_edge);
  DetectRoots (comp_edge, image_out);
}
```

FIGURE 2 Initial code of the cavity detection algorithm.

result, the functions will not work on whole image at once anymore, but on a line-per-line pipelining base. This is shown in Figure 3. Finally, a similar loop folding and merging transformation is applied to the x-loops too. The result is that the algorithm will now work on a fine-grain (pixel per pixel) pipelining base. This is shown in Figure 4. The code for the heart of the loop nest (without pre- and post-ambles) after these transformations is given in Figure 5.

Figure 6 shows that the main memory transfers have been reduced with a factor of 2 due to the data flow transformations. No further reduction in array accesses has been achieved, since the loop transformations do not change this; they only increase the locality. As a result of that however,

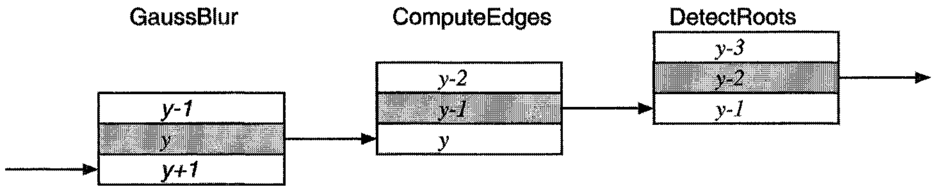


FIGURE 3 Cavity detection algorithm after y-loop transformation.

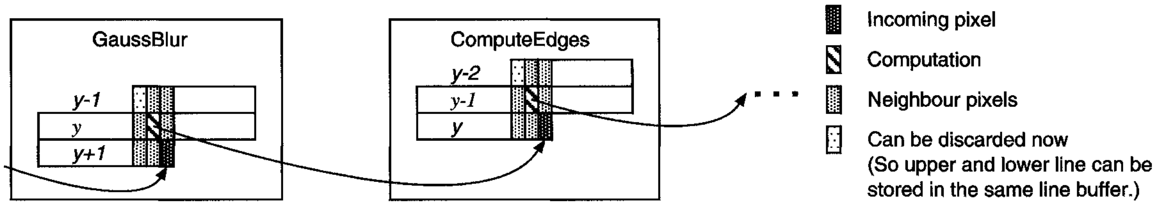


FIGURE 4 Cavity detection algorithm after x-loop transformation.

```
void cav_detect(uchar image_in[M][N], uchar image_out[M][N])
{
  for (y=0; y<M; ++y)
    for (x=0; x<N; ++x) {
      gauss_x[y][x] =... // Apply horizontal gauss-blurring
      gauss_xy[y-1][x] =... // Apply vertical gauss-blurring
      comp_edge[y-2][x-1] =... // Replace pix. with max. diff. with neighbors
      image_out[y-3][x-2] =... // true if no neighbors are < than curr. pix.
    }
}
```

FIGURE 5 Cavity detection code after application of data-flow and loop transformations.

the cache misses and system bus load are significantly reduced, as shown in Figure 7.

3.3. Adding Hierarchical Data Reuse Copies

While the loop transformations increase the locality of the code, we have found that in many cases this locality is not optimally exploited by the cache hierarchy. To make the data reuse more explicit, we introduce additional variables (usually small arrays) into the code in a judicious way. In these variables, copies of parts of other variables (usually large arrays) are stored. The explicit addition of these variables is necessary to enforce the data reuse decisions on the lower level stages of compilation. It is an alternative for relying on the HW cache controller which cannot make any global trade-offs at run-time.

Many alternatives for these hierarchical copies are available. We have a systematic search strategy and a model which represents all promising alternatives [24]. That allows us to identify the best solution. For the cavity detection example, two levels of data reuse can be identified from the above figures which illustrate the loop transformations: line buffers and pixel buffers.

Line Buffers For each of the functions, a buffer of three lines can be implemented, in which the line

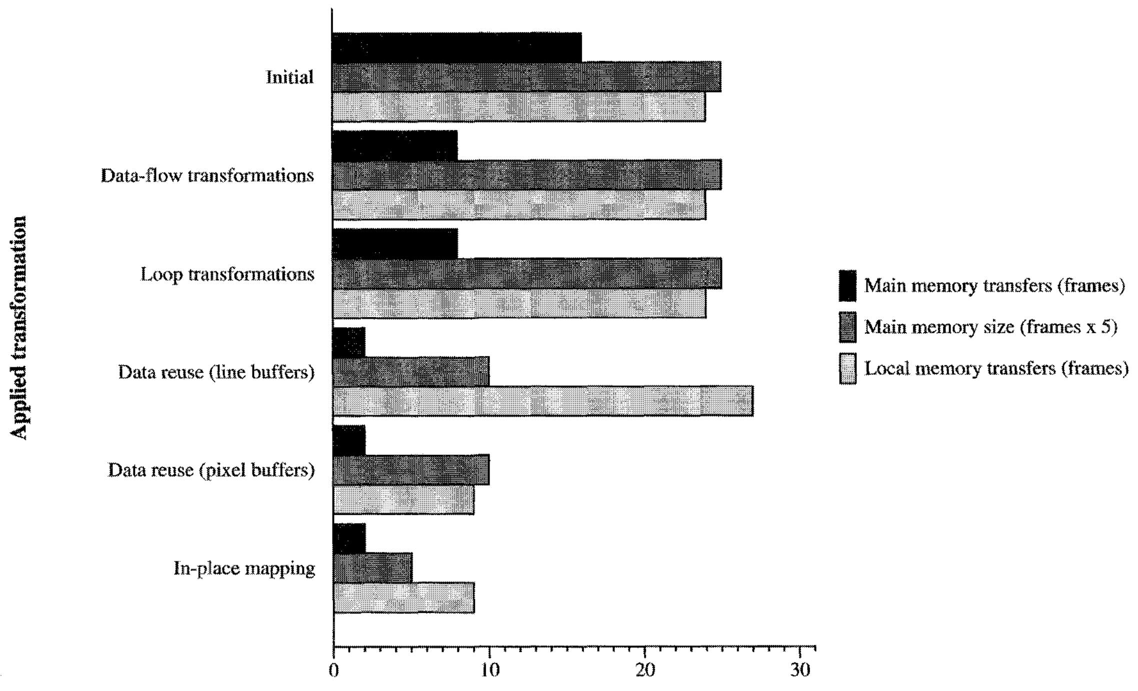


FIGURE 6 Array accesses and sizes, for image size of 1280 × 1000.

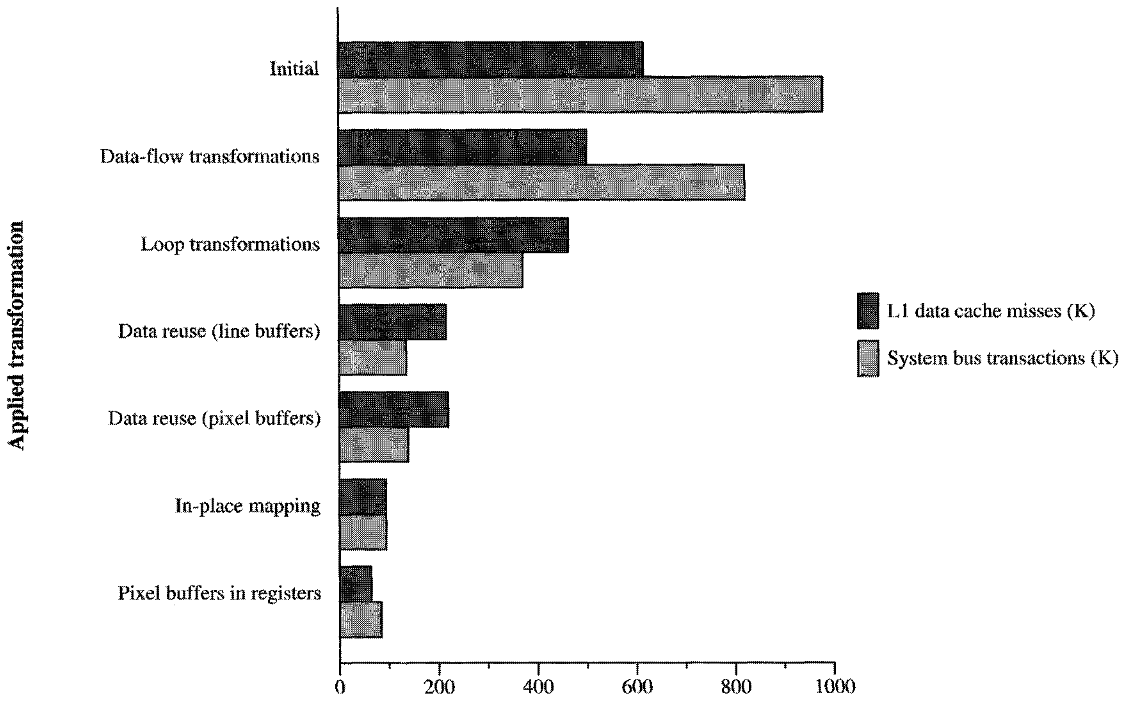


FIGURE 7 Cache and system bus results. Note that system bus transactions correspond ± to L2 cache misses and write-backs.

being processed is stored, together with the previous and the next line:

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

The resulting code for the heart of the loop nest is given in Figure 8.

Pixel Buffers In Figure 4, we can see that a second level of data reuse can be identified, *i.e.*, the pixels in the neighborhood of the pixel being processed:

- For the horizontal gauss-blurring, a buffer of three pixels *in_pixels* can be implemented, storing the last used values of the incoming image.
- For the vertical gauss-blurring, no such buffer is possible. However, the output of this step is stored into a three by three pixels 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, *i.e.*, *comp_edge_pixels*.
- Finally, *DetectRoots*() is performed on this buffer, and the result stored in the output image.

The resulting code is given in Figure 9 (again without initializations and pre- and post-ambles).

Results The results are given in Figure 6 again. This shows that the line buffers have a huge impact on the main memory size and transfers, while the pixel buffers reduce mainly the local memory transfers. In Figure 7, we can see that also the cache misses and system bus load are reduced by these steps. Note that the pixel buffers can actually be stored in registers; but they have to be implemented as scalar variables instead of arrays to allow the compiler to do this. However, this is only done at the end of our DTSE script (see Section 3.6), so the influence on cache misses and bus transactions is only visible at the end of Figure 7.

3.4. Aggressive In-place Mapping

The aim of this step is to reduce the memory size by storing signals with different lifetimes in the same physical memory location. Conventionally memory reuse is only applied for variables with

```
void cav_detect (uchar image_in[M][N], uchar image_out[M][N])
{
  uchar gauss_x_lines[3][N], gauss_xy_lines[3][N], comp_edge_lines[3][N];

  for (y=0; y<=M-1+3; ++y) {
    for (x=0; x<=N-1+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 max. diff.
      image_out[y-3][x-2] = ... // true if no neighbors are < than curr. pix.
    }
  }
}
```

FIGURE 8 Cavity detection code after introducing line buffers.

```
void cav_detect (uchar image_in[M][N], uchar image_out[M][N])
{
  uchar gauss_x_lines[3][N], gauss_xy_lines[3][N], comp_edge_lines[3][N];
  uchar in_pixels[3], gauss_xy_pixels[3][3], comp_edge_pixels[3][3];

  for (y=0; y<=M-1+3; ++y) {
    for (x=0; x<=N-1+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 vert. gauss-blurring
      comp_edge_pixels[(y-2) % 3][(x-2) % 3]
        = comp_edge_lines[(y-2) % 3][x-2] = ... // Replace with max. diff.
      image_out[y-3][x-3] = ... // true if no neighbors are < curr. pix.
    }
  }
}
```

FIGURE 9 Cavity detection code after introducing pixel buffers.

different scopes. We apply aggressive in-place transformations also when arrays have partly overlapping lifetimes [7, 19, 21]. The goal is to make the data fit better in the caches, such that capacity misses can be reduced.

1. Each set of 3 line buffers (*e.g.*, *gauss_x_lines*[3][*N*]) can be reduced to 2 line buffers (*e.g.*, *gauss_x_lines*[2][*N*]) and a few scalars (as illustrated in Fig. 4), because the array variables in the first and the last one have largely different life-times. The extra scalars are the few pixels in the current neighborhood computations, but these are in the pixel buffers anyway.
2. The input and output image buffer can be combined. Again these buffers as such do not have non-overlapping life-times, but most of the pixels in them do. The pixels for which this is not the case, are in the line buffers anyway.

The results of this step are given in Figures 6 and 7 again.

3.5. Main Memory Data Layout Optimization

In the next step of our DTSE script, the layout of the data in main memory has to be chosen to reduce the cache conflict misses. In this step, the techniques from [3, 5, 11, 15, 17] can be applied, but we also go further as described in [13]. There the reduction of the conflict misses is accomplished by storing arrays in memory in an interleaved way, based on cache parameters such as line size, associativity and cache size. Arrays with potential conflict misses are assigned to memory regions that will be mapped to different lines of the cache. Note that not only the starting address or the internal layout of an array is changed: the arrays are really cut in pieces and interleaved with each other.

However, this technique is especially suited for embedded systems with small caches. After the previous optimizations of the cavity detection algorithm, the number of conflict misses is already minimal on the 512K 4-way L2-cache of the Pentium® II. Therefore, we have not included this step in the cumulative results of Figure 7. However, when applied on the initial version, the data layout step reduces both L1 misses and system bus transactions with about 20%.

3.6. Address and Loop Flow Related Optimizations (ADOPT)

Up to now, we have not given performance results for the above versions, because some DTSE stages make the array index expressions much more

TABLE I Performance results for cavity detection algorithm

	Execution time
Initial	1.38 s
DTSE only	2.64 s
DTSE + ADOPT	0.58 s

TABLE II Results for two other applications

Application	L1 misses	System bus trans.	Execution time
QSDPCM (initial)	332 K	370 K	5.15 s
QSDPCM (DTSE + ADOPT)	72 K	65 K	2.74 s
Voice coder (initial)	96 K	70 K	0.013 s
Voice coder (DTSE + ADOPT)	40 K	13 K	0.006 s

complex, and these have to be optimized again (*e.g.*, by replacing modulo operations with counters). Also some conditions and more complex loop bounds were added, and these can be removed from the heart of the loop nest by a loop splitting operation. These optimizations are not really part of the DTSE methodology, but should be performed additionally to increase the performance. We have a systematic methodology for these optimizations too: ADOPT (ADdress OPTimizations) [16]. Note that, as part of these address optimizations, also the pixel buffers introduced in Section 3.3 are replaced by individual scalars (instead of arrays), such that the compiler can map them on registers.

3.6.1. *Performance Results*

Table I gives the performance results for the Pentium® II target processor. The table shows that the complex addressing introduced by DTSE reduces the performance at first, but after removing this addressing overhead again, a clear performance gain w.r.t. the initial version is obtained.

4. OTHER EXPERIMENTAL RESULTS

The cavity detection is a very regular application, which lends itself very well to the DTSE optimizations. That is why we have used it in this paper to illustrate the methodology. In Table II, we give some results for two other applications: a Voice Coder, and a video compression algorithm based on motion estimation (QSDPCM) [18]. The table shows that we obtain good results for these applications too.

5. CURRENT STATUS

It is not our goal to automate the DTSE methodology as a whole (this is not what designers want either). Rather, we want to develop a systematic methodology, with tools that automate some substeps, or help to select the best alternative. Currently we have prototype tools which apply the in-place mapping and data layout steps. For the data reuse step, we have a tool to represent all alternatives as a tree. Such a tree can be constructed for each alternative which results from the data-flow and loop transformation step. This makes it convenient for a designer to select the best alternative. Throughout the design process, our ATOMIUM access counting tool can be used to identify the (remaining) bottlenecks.

6. CONCLUSION

In embedded multimedia systems, the energy consumption and performance bottlenecks are mostly in the system bus load and off-chip memory accesses (instead of the CPU) nowadays. In this paper, we have demonstrated a systematic code rewriting methodology, partly supported by tools, to optimally exploit the memory hierarchy and reduce the system bus load for these applications. The approach is valid for both programmable and custom hardware realizations. Here, we have illustrated the methodology on a cavity detection application for medical imaging, with a programmable processor (Pentium® II) as target architecture. The result is a large saving in the system bus load. The CPU performance improvement is also very significant, but it is smaller than the system bus load reduction.

References

[1] Amarasinghe, S., Anderson, J., Lam, M. and Tseng, C., "The SUIF compiler for scalable parallel machines", In: *Proc. of the* 7th *SIAM Conf. on Parallel Proc. for Scientific Computing*, 1995.

[2] Bister, M., Taeymans, Y. and Cornelis, J. (1989). "Automatic segmentation of cardiac MR images", *Computers in Cardiology*, *IEEE Computer Society Press*, pp. 215–218.

[3] Calder, B., Krintz, C., John, S. and Austin, T., "Cache-conscious data placement", *Proc. ASPLOS-VIII*, pp. 139–149, San Jose CA, Oct., 1998.

[4] Catthoor, F., Wuytack, S., De Greef, E., Franssen, F., Nachtergaele, L. and De Man, H. (1998). "System level transformations for low power data transfer and storage", In: *Paper collection on "Low power CMOS design"* (Eds. Chandrakasan, A. and Brodersen, R.), *IEEE Press*, pp. 609–618.

[5] Chatterjee, S., Jain, V., Lebeck, A., Mundhra, S. and Thottethodi, M., "Nonlinear array layouts for hierarchical memory systems", *Proc. ACM Int. Conf. on Supercomputing* (*ICS*), Rhodes, Greece, June, 1999.

[6] Danckaert, K., Catthoor, F. and De Man, H., "System level memory optimization for hardware-software co-design", *Proc. IEEE Intnl. Workshop on Hardware/Software Co-design*, pp. 55–59, Braunschweig, Germany, March, 1997.

[7] De Greef, E., Catthoor, F. and De Man, H., "Memory size reduction through storage order optimization for embedded parallel multimedia applications", special issue on "*Parallel Processing and Multi-media*" (Ed. Krikelis, A.), In: *Parallel Computing* Elsevier, **23**(12), December, 1997.

[8] De Man, H., Catthoor, F., Goossens, G., Vanhoof, J., Van Meerbergen, J., Note, S. and Huisken, J., "Architecture-driven synthesis techniques for VLSI implementation of DSP algorithms", *Proc. of the IEEE*, special issue on "*The Future of Computer-aided Design*", **78**(2), 319–335, February, 1990.

[9] Feautrier, P. (1992). "Some efficient solutions to the affine scheduling problem", *Int. J. of Parallel Programming*, **21**(5), 389–420.

[10] Gonzales, R. and Horowitz, M., "Energy dissipation in general-purpose microprocessors", *IEEE J. of Solid-state Circ.*, **SC-31**(9), 1277–1283, September, 1996.

[11] Kandemir, M., Choudhary, A., Ramanujam, J., Shenoy, N. and Banerjee, P., "Enhancing spatial locality *via* data layout optimizations", *Proc. Euro-Par Conference*, Southampton, UK, Sep. 1998. "*Lecture Notes in Computer Science*" *series*, **1470**, 422–434.

[12] Kelly, W. and Pugh, W., "A framework for unifying reordering transformations", *Technical report CS-TR-3193*, Dept. of CS, Univ. of Maryland, College Park, April, 1993.

[13] Kulkarni, C., Catthoor, F. and De Man, H., "Advanced data layout organization for multi-media applications", *Int. Parallel and Distr. Proc. Symp.* (*IPDPS*), Workshop on "Parallel and Distrib. Computing in Image Proc., Video Proc., and Multimedia (PDIVM'2000)", Cancun, Mexico, May, 2000.

[14] Lippens, P., van Meerbergen, J., Verhaegh, W. and van der Werf, A., "Allocation of multiport memories for hierarchical data streams", *Proc. IEEE Int. Conf. Comp. Aided Design*, Santa Clara CA, Nov., 1993.

[15] Manjiakian, N. and Abdelrahman, T., "Array data layout for reduction of cache conflicts", *Int. Conf. on Parallel and Distributed Computing Systems*, 1995.

[16] Miranda, M., Catthoor, F., Janssen, M. and De Man, H., "High-level address optimization and synthesis techniques for data-transfer intensive applications", *IEEE Trans. on VLSI Systems*, **6**(4), 677–686, December, 1998.

[17] Rivera, G. and Tseng, C., "Data transformations for eliminating conflict misses", *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation* (*PLDI*), pp. 38–49, Montreal, Canada, June, 1998.

[18] Strobach, P. (1988). "QSDPCM–A New Technique in Scene Adaptive Coding," *Proc.* 4th *Eur. Signal Processing Conf.*, EUSIPCO-88, Grenoble, France, Elsevier Publ., Amsterdam, pp. 1141–1144.

[19] Strout, M., Carter, L., Ferrante, J. and Simon, B., "Schedule-independent storage mapping for loops", *Proc. ASPLOS-VIII*, pp. 24–33, San Jose CA, Oct., 1998.

[20] Tiwari, V., Malik, S. and Wolfe, A., "Power analysis of embedded software: a first step towards software power minimization", *IEEE Trans. on VLSI Systems*, **2**(4), 437–445, December, 1994.

[21] Wilde, D. and Rajopadhye, S., "Memory reuse analysis in the polyhedral model", *Proc. Euro-Par Conf.*, Lyon, France, Aug. 1996. *Lecture Notes in Computer Science*, **1123**, 389–397, Springer, 1996.

[22] Wolf, M., "Improving locality and parallelism in nested loops", *Ph.D. Dissertation*, Aug., 1992.

[23] Wuytack, S., Catthoor, F., Nachtergaele, L. and De Man, H., "Power exploration for data dominated video applications", *Proc. IEEE Intnl. Symp. on Low Power Design*, Monterey CA, pp. 359–364, August, 1996.

[24] Wuytack, S., Diguet, J. P., Catthoor, F. and De Man, H., "Formalized methodology for data reuse exploration for low-power hierarchical memory mappings", *IEEE Trans. on VLSI Systems*, **6**(4), 529–537, December, 1998.

Authors' Biographies

Koen Danckaert was born in 1972 in Ghent, Belgium. He received the engineering degree in computer science from the Katholieke Universiteit Leuven, Belgium in 1995. Since October 1995 he has been working towards a Ph.D. degree at the Inter-university Micro-Electronics Center (IMEC), Leuven, Belgium. He is supported by a fellowship of the Fund of Scientific Research – Flanders. His current research interests are global data transfer and storage exploration for (parallel) multi-media processors, mainly targeted to image and video processing applications.

Francky Catthoor received the Eng. degree and a Ph.D. in El. Eng. from the K.U. Leuven, Belgium in 1982 and 1987 respectively. Since 1987, he has headed research domains in the area of architectural and synthesis methodologies, within the DESICS division at IMEC. He is assistant professor at the EE department of the K.U. Leuven. Since 1995 he is an associate editor for the IEEE Trans. on VLSI Systems and since 1996 also for the Journal of VLSI Signal Processing. His research activities mainly belong to the field of application-specific architecture design methods and system-level exploration, with emphasis on memory and global data transfers.

Hugo J. De Man received the electrical engineering degree and the Ph.D. degree in Applied Sciences from the K.U. Leuven, Belgium, in 1964 and 1968, respectively. In 1974 he became a Professor at the University of Leuven. He received Best Paper Awards at the 1973 ISSCC, the 1981 ESSCIRC, and the 1989 DAC. 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. In 1999 he was awarded the Phil Kaufman Award by the EDA Consortium. Since 1995 he became a Senior Research Fellow of IMEC.

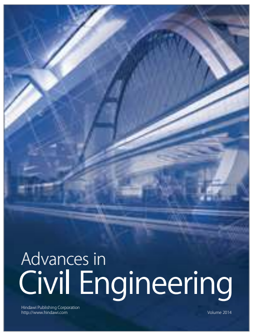
Vivek Tiwari received a B. Tech degree in Computer Science and Engineering from the Indian Institute of Technology in 1991, and a Ph.D. in Electrical Engineering from Princeton University in 1996. He is currently managing the Low Power Design Technology group at Intel Corp. The charter of the group is to reduce power in Intel's high-end microprocessors. The focus of his current research is on tools and micro-architectural techniques for power reduction. He received the IBM Fellowship in 1993, 1994 and 1995 and a Best Paper Award at ASP-DAC '95. He has served on the technical program committees of DAC, ISLPED and VLSI Design.

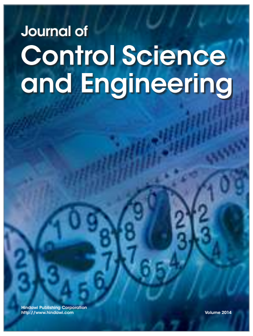
Journal of
Engineering

The Scientific World Journal

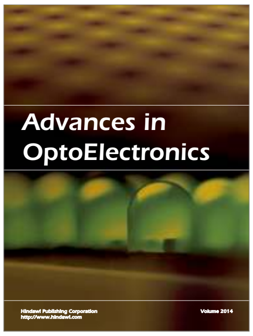
International Journal of
Rotating Machinery

Journal of
Sensors

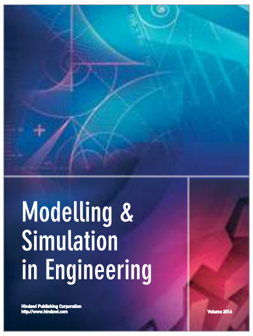
International Journal of
Distributed Sensor Networks

Advances in
Civil Engineering

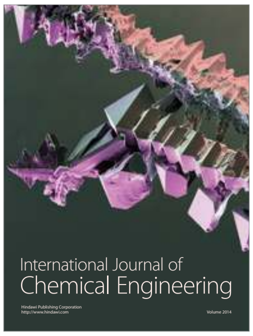
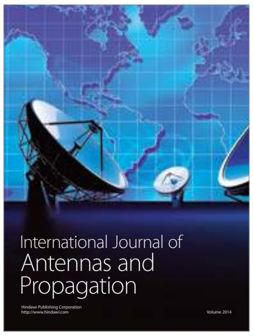

Journal of
Control Science and Engineering

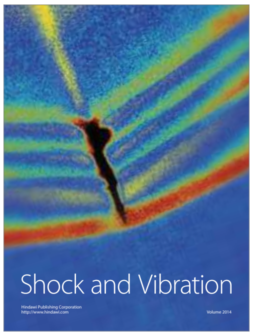
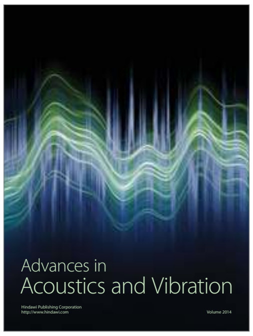
Journal of
Robotics

Journal of
Electrical and Computer Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and Observation

Modelling & Simulation in Engineering

International Journal of
Aerospace Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and Propagation

Active and Passive Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration