Multiprocessor Mapping of Process Networks: A JPEG Decoding Case Study

E. A. de Kock
Philips Research, Prof. Holstlaan 4, 5656 AA, Eindhoven, The Netherlands
Erwin.de.Kock@philips.com

ABSTRACT

We present a system-level design and programming method for embedded multiprocessor systems. The aim of the method is to improve the design time and design quality by providing a structured approach for implementing process networks. We use process networks as re-usable and architecture-independent functional specifications. The method facilitates the cost-driven and constraint-driven source code transformation of process networks into architecture-specific implementations in the form of communicating tasks. We apply the method to implement a JPEG decoding process network in software on a set of MIPS processors. We apply three transformations to optimize synchronization rates and data transfers and to exploit data parallelism for this target architecture. We evaluate the impact of the source code transformations and the performance of the resulting implementations in terms of design time, execution time, and code size. The results show that process networks can be implemented quickly and efficiently on embedded multiprocessor systems.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures. C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems, signal processing systems.* C.4 [**Performance of Systems**]: *Design studies.* D.1.3 [**Programming Techniques**]: Concurrent Programming – *parallel programming.* D.2.13 [**Software Engineering**]: Reusable Software. D.3.4 [**Programming Languages**]: Processors – *optimization.* J.7 [**Computers in Other Systems**]: *Consumer products.*

General Terms

Algorithms, performance, design, standardization, languages.

Keywords

Multiprocessor mapping, process network, code transformation, system design method, data parallelism, task-level parallelism.

1. INTRODUCTION

The complexity of designing and programming embedded multimedia systems is growing rapidly. In order to reduce the complexity and, hence, manage the design effort and the time-to-market, the system's function and architecture need to be separated [9] in the design process. The separation allows the re-use of functions for implementation on different architectures as well as the re-use of architectures for implementation of different functions. We address the re-use of functions for implementation on different architectures.

A key issue of our design method is that application designers model system functions in terms of process networks [8] and not in terms of a platform interface. Consequently, they are not bothered with implementation decisions concerning the architecture costs and architecture constraints. Subsequently, system designers can transform the functions for a specific architecture taking into account its costs and constraints, such that the functions can be implemented efficiently in hardware and software. Since the functions are free of implementation decisions, system designers do not have to perform reverse engineering to obtain a suitable functional specification that can be implemented on a specific architecture. This saves valuable design effort and design time. After transformation the networks consist of tasks which inter-operate through a platform interface using concepts such as described in [2], [7], [12], and [14]. We apply these transformations on source code such that the resulting code can be used as input for existing silicon and software compilers.

The transformations encode system-level design decisions in the source code. Typical design decisions concern task-level and data parallelism such as the minimization of the number of run-time tasks [5][13], task concurrency management [11], and data transfer and storage [3][4]. A survey of data and memory optimization techniques for embedded systems is given in [10]. Our method differs from those mentioned above by the fact that we start from an inherently parallel functional model in the form of a process network and by the fact that we target multiprocessor implementations. The advantage of using process networks is that task-level parallelism is already made available in the function. Unlike procedural programs, process networks do not contain global data structures that have to be broken down to obtain a parallel program, which usually is a hard problem. The fact that we target multiprocessor implementations implies that minimizing the number of run-time tasks is not necessarily the best solution because we need to consider the binding of tasks to processors and the resulting communication between the processors.

In this paper we apply the proposed method in order to implement JPEG decoding on a multiprocessor architecture. In Section 2 we model JPEG decoding as a process network. In Section 3 we present the multiprocessor architecture. In Section 4 we discuss the design decisions and present the corresponding code transformations. In Section 5 we show the results of these

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSS'02, October 2–4, 2002, Kyoto, Japan.
Copyright 2002 ACM 1-58113-562-9/02/0010…$5.00.

transformations. In Section 6 we conclude with some remarks and discuss future work.

2. JPEG DECODING NETWORK

We have modeled JPEG decoding functionality as described in [15] as a process network using the Y-chart Applications Programmers Interface (YAPI) [6]. The process network is shown in Figure 1. It consists of 26 processes that communicate using FIFO channels.

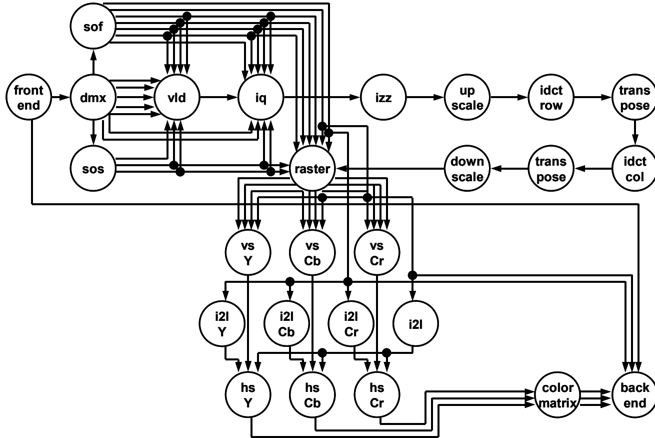

Figure 1. JPEG decoding process network.

The frontend process reads a file in JPEG File Interchange Format (JFIF) from disk and writes it as a byte stream to the DMX process. The DMX process de-multiplexes the byte stream into the DHT, DQT, SOF and SOS segments of JFIF. The SOF process parses the frame headers that are contained in the SOF segment and writes the frame header data to its outputs. The SOS process parses the scan headers that are contained in the SOS segment and writes the scan header data to its outputs. The variable length decoding (VLD) process decodes the run length and Huffman encoded minimum coding units using the Huffman tables that are contained in the DHT segment. It writes the decoded pixel blocks (8x8 matrices) as a pixel stream to the inverse quantization (IQ) process. The IQ process multiplies the pixels by the quantization factors that are contained in the DQT segment. Subsequently, the pixel blocks undergo inverse zigzag (IZZ), and two-dimensional inverse discrete cosine transformation (IDCT). The two-dimensional IDCT operation is modeled using two one-dimensional IDCT processes and two matrix transposition (transpose) processes. The IDCT operation contains two bit-scaling processes (upscale and downscale) to increase and decrease the bit precision from 8 to 11 bits and back. Next, the raster process reorders the pixels from their order in pixel blocks, minimum coding units, and stripes into scan order in pixels, lines, and frames. The raster process de-multiplexes the pixel stream with the interleaved luminance (Y) and chrominance (Cb and Cr) pixels into three separate streams for Y, Cb, and Cr pixels. Subsequently, the vertical (VS) and horizontal scaling (HS) processes duplicate pixels such that each stream contains the number of pixels that has been specified in the frame header data. Hence, the resulting streams are in 4:4:4 format. The image-to-line (i2l) processes glue the frame-based output interfaces of the vertical scaling processes to the line-based input interfaces of the horizontal scaling processes. The color matrix process converts the Y, Cb, and Cr streams to R, G, and B streams. Finally, the backend process reads the R, G, and B streams and writes them in sun raster file format to disk.

Each process has an input/output relation such that its latency, i.e., the number of pixels that is read before the first pixel is written, is minimal. The minimum latency minimizes the memory requirements of the process. Furthermore, the minimum latency of each process minimizes the communication constraints with its environment and, therefore, maximizes the re-usability of the process. For example, the horizontal scaling processes have a latency of one pixel. If we re-use them in another process network in which we only need to scale the luminance pixels, then one pixel delay suffices to delay the chrominance pixels accordingly. If the horizontal scaling on luminance pixels would have a latency of one frame of pixels, then this would impose significant frame memory requirements in order to match the delay in the chrominance pixels. Similar arguments apply for feedback loops.

3. MULTIPROCESSOR ARCHITECTURE

We have implemented the JPEG decoding process network on a single tile of the CAKE multiprocessor architecture [14]. A tile consists of a heterogeneous set of processors and memories that communicate through a snooping interconnection network. Each processor has its own cache. The snooping protocol ensures that the caches have a coherent view on the single uniform shared memory space. In our tile configuration we have used a homogeneous set of MIPS processors and four memory banks to implement the memory space. All processors in the tile operate on a single queue of runable tasks. A small operating system, called tile run-time system, dynamically assigns tasks to processors. If one processor suspends a task, then another processor can resume this task. The YAPI process network library has been implemented in software on top of the tile run-time system. In this library, each process is implemented as a separate task.

4. MAPPING

In order to efficiently implement the JPEG decoding process network on the CAKE multiprocessor architecture we have exploited three observations. The first observation is that pixel-based communications, i.e., read and write calls that are used to communicate single pixels, are expensive in software due to the large number of synchronization actions. The second observation is that fine-grain tasks, i.e., tasks that perform little computation compared to communication, are expensive in software due to the relatively large overhead of data transfer between the tasks. The third observation is that RISC processors such as MIPS processors cannot exploit data parallelism unlike, for instance, VLIW processors. To exploit data parallelism, we use multiple RISC processors and we transform data parallelism into task-level parallelism. Based on these observations we apply three types of transformations that we call data packaging, task fusion, and task unrolling.

We do not focus on context switching. Context switching can only occur during execution of read and write calls because the tile run-time system schedules its tasks non-preemptively. We do not exploit the tradeoff between the buffer capacity of the FIFO channels between the tasks and the amount of context switching as discussed in [1]. To minimize the amount of context switching, we allocate a large buffer of 8K per FIFO channel. Hence, context switching only occurs during execution of read calls on empty FIFO channels.

4.1 Data Packaging

In the first transformation, called data packaging, we decrease the number of read and write calls of a process by increasing the number of communicated pixels per call. As a consequence we have to increase both the internal memory requirements of the process and the latency of the process, which decreases its re-usability. An example of data packaging is shown in Figure 2 and Figure 3, where we have increased the maximum packet size in the IQ process from 1 pixel before transformation to 64 pixels after transformation. We furthermore applied data packaging in the variable length decoding, upscale, downscale, horizontal scaling, color matrix, and backend processes.

```
void IQ::main() {
  ...
  for(int i=0; i<nrOfScanComponents; i++) {
    int hi = H[scanComponentId[i]];
    int vi = V[scanComponentId[i]];
    int ti = tid[scanComponentId[i]];
    for(int j=0; j<vi; j++) {
      for(int k=0; k<hi; k++) {
        for(int l=0; l<64; l++) {
          VYApixel Cin;
          VYApixel Cout;
          read(CinP, Cin);
          Cout = QTable[ti][l]*Cin;
          write(CoutP, Cout);
        }
      }
    }
  }
  ...
}
```

Figure 2. Pixel-based inverse quantization process.

```
void IQ::main() {
  ...
  for(int i=0; i<nrOfScanComponents; i++) {
    int hi = H[scanComponentId[i]];
    int vi = V[scanComponentId[i]];
    int ti = tid[scanComponentId[i]];
    for(int j=0; j<vi; j++) {
      for(int k=0; k<hi; k++) {
        VYApixel Cin[64];
        VYApixel Cout[64];
        read(CinP, Cin, 64);
        for(int l=0; l<64; l++) {
          Cout[l] = QTable[ti][l]*Cin[l];
        }
        write(CoutP, Cout, 64);
      }
    }
  }
  ...
}
```

Figure 3. Block-based inverse quantization process.

The advantage of data packaging is that the system can transfer the data in larger packets if the FIFO channels between the processes have sufficient buffer capacity. This reduces the number of synchronization actions.

4.2 Task Fusion

In the second transformation, called task fusion, we combine two or more processes in one process in order to avoid the data transfer overhead of fine-grain tasks. As an example we show the inverse zigzag process in Figure 4 that we have combined with the block-based IQ process shown in Figure 3. The resulting process is shown in Figure 5.

```
void IZZ::main() {
  while (true) {
    VYApixel Cin[64];
    VYApixel Cout[64];
    read(CinP, Cin, 64);
    for (unsigned int i=0; i<64; i++) {
      Cout[zigzag[i]] = Cin[i];
    }
    write(CoutP, Cout, 64);
  }
}
```

Figure 4. Inverse zigzag process.

```
void IQ_IZZ::main() {
  ...
  for(int i=0; i<nrOfScanComponents; i++) {
    int hi = H[scanComponentId[i]];
    int vi = V[scanComponentId[i]];
    int ti = tid[scanComponentId[i]];
    for(int j=0; j<vi; j++) {
      for(int k=0; k<hi; k++) {
        VYApixel Cin[64];
        VYApixel Cout[64];
        read(CinP, Cin, 64);
        for(int l=0; l<64; l++) {
          Cout[zigzag[l]] = QTable[ti][l]*Cin[l];
        }
        write(CoutP, Cout, 64);
      }
    }
  }
  ...
}
```

Figure 5. Inverse quantization and zigzag process.

The advantage of task fusion is that the system has to transfer less data if the data can be kept locally in registers and caches. This can improve the performance since the processors sequentially perform the computation and the communication of a process. If computation and communication can be done in parallel, then task fusion can be used to balance the computation and communication load. Furthermore, task fusion reduces the amount of context switching.

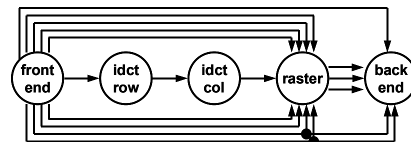

Figure 6. Coarse-grain process network.

We repeatedly apply task fusion until we obtain the coarse-grain process network shown in Figure 6. In Figure 6, the frontend process contains the original frontend, DMX, SOF, SOS, VLD, IQ, IZZ, and upscale processes. Each IDCT process is a combination of the original IDCT process and its subsequent transpose process. The raster process contains the original downscale, raster, vertical scaling, image-to-line, and horizontal scaling processes. The backend process contains the original backend and color matrix processes.

4.3 Task Unrolling

Next we have optimized the coarse-grain process network shown in Figure 6 both for execution on a single processor and for execution on multiple processors. In order to optimize for execution on a single processor, we apply task fusion to combine the five coarse-grain processes into a single process. This removes all multi-tasking overhead, since the resulting process is executed as a single task by the tile run-time system. To optimize the execution on multiple processors, we have made the available data parallelism explicit as task-level parallelism by unrolling the IDCT and raster processes; see Figure 7. Each of the resulting IDCT and raster processes handles only one of the color components Y, Cb, and Cr, whereas the original processes handled all color components.

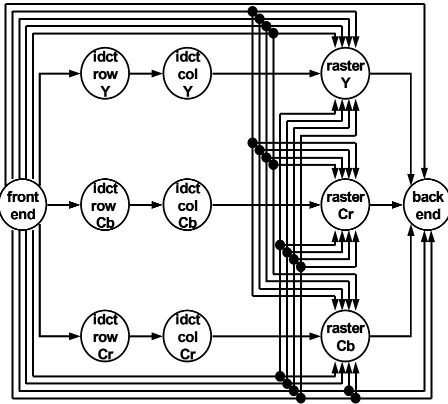

Figure 7. Data parallel process network.

The advantage of task unrolling is that the system can process data in parallel without the overhead of additional data transfers associated with pipelining.

5. RESULTS

We compare the implementations of the JPEG decoding process network with an implementation of JPEG decoding in the form of a C program that has been developed in a procedural programming style. We have labeled the different JPEG programs in Table I such that we can easily refer to them.

Table I. The characteristics of the JPEG decoding programs.

Program	Style	#T	DP	TF	TU
JPEG 1	C	1			
JPEG 2	PN	26	No	No	No
JPEG 3	PN	26	Yes	No	No
JPEG 4	PN	5	Yes	Yes	No
JPEG 5	PN	11	Yes	Yes	Yes
JPEG 6	PN	1	Yes	Yes	No

The procedural program has label JPEG 1. The functional process network shown in Figure 1 has label JPEG 2. The process network that results from applying data packaging has label JPEG 3. The coarse-grain process network that results from task fusion and that is shown in Figure 6 has label JPEG 4. The data parallel process network that is shown in Figure 7 has label JPEG 5. Finally, the single process that is the result of combining the five processes of Figure 6 has label JPEG 6. Table I provides an overview of the programs and their characteristics in terms of design style (procedural C or process networks), number of run-time tasks (#T), and applied transformations (data packaging (DP), task fusion (TF), and task unrolling (TU)). We have compared these six JPEG programs in terms of design time, execution time, and code size.

Both the design of the procedural program (JPEG 1) and the design of the functional process network (JPEG 2) have taken three person weeks. Two different persons have developed the programs using the same data structures and implementations for the basic functionality such as VLD and IDCT. They were not familiar with JPEG. The transformations of JPEG 2 into JPEG 3 up to JPEG 6 have taken two weeks and have been done by the designer of JPEG 2.

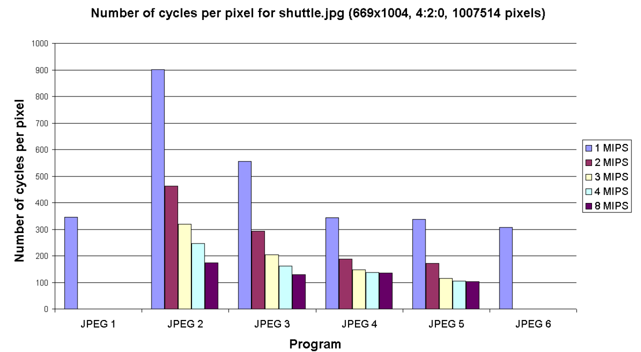

Figure 8. The execution times of the JPEG programs.

Figure 8 shows the execution times of the different programs in number of cycles per pixel. We have executed JPEG 1 and JPEG 6 on only one processor because they consist of only one process. We have executed the other programs also on 2, 3, 4, and 8 processors. We express the execution times in the number of cycles per pixel in the input file. The JPEG picture “shuttle.jpg” has a dimension of 669x1004 pixels in 4:2:0 format, which results in a resolution of 1,007,514 pixels. To obtain the number of cycles per pixel we divide the execution time in cycles by the resolution in pixels.

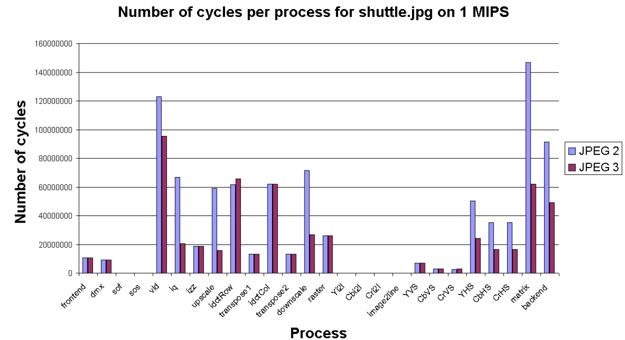

Figure 9. The impact of synchronization rate reduction.

Figure 9 shows the execution time in cycles per process for JPEG 2 and JPEG 3 on one processor. The transformation to reduce the synchronization rate for the variable length decoding, upscale, downscale, horizontal scaling, color matrix, and backend processes accelerates the execution of these processes.

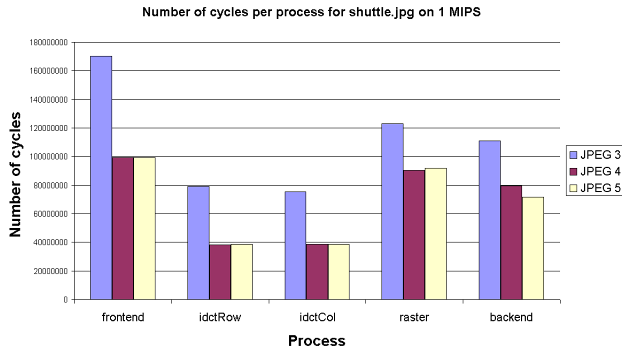

Figure 10. The impact of data transfer reduction.

Figure 10 shows the execution time in cycles per process for JPEG 4 on one processor. We have compared the execution time of each process with the sum of the execution times of the corresponding processes of JPEG 3 and JPEG 5. For instance, the backend process in JPEG 4 is a combination of the color matrix process and the backend process in JPEG 3. Similarly, the idctRow process in JPEG 4 is a combination of the idctRowY, idctRowCb, and idctRowCr processes in JPEG 5. The comparison between JPEG 3 and JPEG 4 shows that the removal of fine-grain tasks accelerates the execution. The acceleration is due to less data transfer overhead since more data can be kept locally in registers and caches of the processors. The comparison between JPEG 4 and JPEG 5 shows that the acceleration is not due to less context switching since they have similar execution times while JPEG 4 contains only half of the number of processes of JPEG 5. Note that we have allocated a large buffer of 8K for each FIFO channel to minimize the amount of context switching due to full FIFO channels.

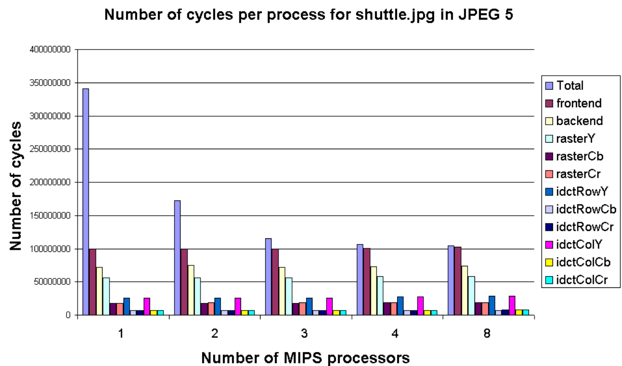

Figure 11. The impact of data parallelism.

Figure 11 shows the execution time in cycles per process for JPEG 5 on tiles with multiple processors. Furthermore, it shows the total execution time of JPEG 5 on these tiles. We note that the execution times of the processes are largely independent of the number of processors. We also note that we can fully exploit the data parallelism of JPEG 5 with three or more processors because the execution of the frontend process determines the total execution time. In the frontend process, the variable length decoding takes most of the execution time as shown in Figure 9. The reading of the JPEG file from disk, which was done in the frontend process of JPEG 2, is not a bottleneck.

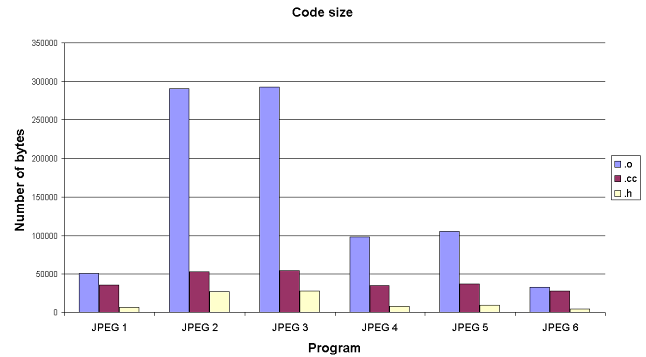

Figure 12. Code sizes of the JPEG programs.

Figure 12 shows the size in number of bytes of the object code and source code of the JPEG programs. We have compiled the programs with a gcc 2.8.1 based cross-compiler for MIPS using the -O3 option. The size of the object code of the process network programs is large compared to the size of the object code of the single process programs JPEG 1 and JPEG 6. This is due to the fact that all code in the YAPI process network software library is inline code in order to improve the execution times of the programs. The disadvantage is that there is a lot of duplicate code in the object code. In order to execute the programs, we link them to the tile run-time system and other system libraries such as the C++ library. The size of the combined system libraries is 530K.

6. CONCLUSION

We have presented a system design method to implement process networks on multiprocessor architectures. There are two advantages of using process networks as functional specifications. Firstly, it shields application designers from implementation details and, hence, it improves the re-usability of functional specifications. Secondly, it reduces the design complexity because system designers start from clean functional specifications that do not contain architecture-specific implementation decisions.

The JPEG decoding case study shows that process networks can be implemented efficiently on multiprocessor architectures in a systematic way by reasoning on costs and constraints. The transformation of the functional process network accelerates the execution with almost a factor of three both on a single processor as well as on a tile of three processors. Furthermore, we have shown that the transformations on process networks are relatively simple. Our experience is that transformations on procedural programs that achieve similar results are much harder to perform.

Future work includes transformations to efficiently implement process network communication on multiprocessor systems with shared memory architectures. These architectures typically have platform instances such as described in [7] which have separate synchronization and data transfer primitives. This separation allows the reduction of synchronization rates without additional memory requirements. Furthermore, this separation can reduce the amount of data transfer between communicating tasks if these tasks have access to the same memory space. Future work furthermore includes the development of tools to automate the presented code transformations.

7. ACKNOWLEDGMENTS

We would like to thank Paul Stravers and Jan Hoogerbrugge for providing the CAKE multiprocessor architecture simulator to evaluate the performance of the JPEG programs.

8. REFERENCES

[1] Basten, T., and J. Hoogerbrugge, "Efficient Execution of Process Networks," In A. Chalmers, M. Mirmehdi and H. Muller, editors, Proceedings Communicating Process Architectures, pp. 1-14, 2001.

[2] Brunel, J.-Y., W.M. Kruijtzer, H.J.H.N. Kenter, F. Petrot, L. Pasquier, E.A. de Kock, and W.J.M. Smits, "COSY Communication IP's," Proceedings 37th Design Automation Conference, pp. 406-409, 2000.

[3] Catthoor, F., K. Danckaert, S. Wuytack, N.D. Dutt, "Code transformations for data transfer and storage exploration preprocessing in multimedia processors," IEEE Design & Test of Computers, vol. 8, issue 3, pp. 70-82. 2001.

[4] Chung, E-Y., L. Benini, G. De Micheli, "Source code transformation based on software cost analysis," Proceedings Int. Symposium on System Synthesis (ISSS), pp. 153-158, 2001.

[5] Cortadella, J., A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, A. Sangiovanni-Vincentelli, "Task generation and compile-time scheduling for mixed data-control embedded software," Proceedings 37th Design Automation Conference, pp. 489-494, 2000.

[6] De Kock, E.A., G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, K.A. Vissers, "YAPI: Application Modeling for Signal Processing Systems," Proceedings 37th Design Automation Conference, pp. 402-405, 2000.

[7] Gangwal, O.P., A.K. Nieuwland, P.E.R. Lippens, "A scalable and flexible data synchronization scheme for embedded HW-SW shared-memory systems," Int. symp. on system synthesis (ISSS), pp. 1-6, 2001.

[8] Kahn, G., "The semantics of a simple language for parallel programming," Information Processing, J.L. Rosenfeld (ed.), North-Holland Publishing Co. 1974.

[9] Keutzer, K., S. Malik, A.R. Newton, J.M. Rabaey, and A Sangiovanni-Vincentelli, "System-Level Design: Orthogonalization of Concerns and Platform-Based Design," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 19, no. 12, pp. 1523-1543, 2000.

[10] Panda, P.R., F. Catthoor, N.D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P.G. Kjeldsberg, "Data and Memory Optimization Techniques for Embedded Systems," ACM Transactions on Design Automation of Electronic Systems, vol. 6, no. 2, pp. 149-206, 2001.

[11] Prayati, A., Chun Wong, P. Marchal, N. Cossement, F. Catthoor, R. Lauwereins, D. Verkest, H. De Man, A. Birbas, "Task concurrency management experiment for power-efficient speed-up of embedded MPEG4 IM1 player," Proceedings International Workshops on Parallel Processing, pp. 453-460, 2000.

[12] Rutten, M.J., J.T.J. van Eijndhoven, E.J.D. Pol, "Design of Multi-Tasking Coprocessor Control for Eclipse," Proceedings 10th Int. symposium on Hardware/Software Codesign (CODES), 2002.

[13] Sgroi, M., L. Lavagno, Y. Watanabe, A. Sangiovanni-Vincentelli, "Synthesis of embedded software using free-choice Petri nets," Proceedings 36th Design Automation Conference, pp. 805-810, 1999.

[14] Stravers, P., and J. Hoogerbrugge, "Homogeneous multiprocessing and the future of silicon design paradigms," Proceedings International symposium on VLSI Technology, Systems, and Applications (VLSI-TAS), pp. 184-187, 2001.

[15] Wallace, G.K., "The JPEG still picture compression standard," IEEE Transactions on Consumer Electronics, vol. 38, issue 1, pp. xviii-xxxiv, 1992.