Architecture Study for Smart Cameras

Harry Broers[1], Wouter Caarls[2], Pieter Jonker[2], Richard Kleihorst[3]

[1] Philips Applied Technologies, Vision, Optics & Sensors, Eindhoven, 5600 MD, The Netherlands
[2] Delft University of Technology, Quantitative Imaging, Delft, 2628 CJ, The Netherlands
[3] Philips Research, Digital Design and Test, Eindhoven, 5656 AA, The Netherlands
email: harry.broers@philips.com

Summary

Embedded real-time image processing is widely used in many applications, including industrial inspection, robot vision, photo-copying, traffic control, automotive control, surveillance, security systems and medical imaging. In these applications, the size of the image can be very large, the processing time should often be small and real-time constraints should be met. Starting in the 1980's, many parallel hardware architectures for low-level image processing have been developed. They range from frame-grabbers with attached Digital Signal Processors (DSPs), to systolic pipelines, square and linear single-instruction multiple-data stream (SIMD) systems, SIMD pyramids, PC-clusters, and since a number of years, smart cameras. As processors are becoming faster, smaller, cheaper, and more efficient, new opportunities arise to integrate them into a wide range of CMOS processor devices. Since there are so many different applications, there is no single processor that meets all the requirements all applications. The processing done on a smart camera has very specific characteristics. On one hand, low-level image processing operations such as interpolation, segmentation and edge enhancement are local, regular, and require vast amounts of bandwidth. On the other hand, high-level operations like classification, path planning, and control may be irregular, consuming less bandwidth. In this paper we will focus on the range of smart cameras of the Philips laboratories and its software support for easy programming that were partly developed in close co-operation with the SmartCam [1] project.

1 Introduction

The SmartCam project investigates how a set of application-specific processors can be generated for intelligent cameras using design space exploration (the hardware framework), and how we are able to schedule the inherent data and task parallelism in an application in such a way, that a balance is found for both data and task parallel parts of the application software (the software framework). The found schedule is optimal for a certain architecture description. For the selection of the best architecture in combination with the best schedule, one can cycle through design space exploration and scheduling.
Developing embedded parallel image processing applications is usually a very hardware-dependent process, requiring deep knowledge of the processors used. It is possible to explore parallelism along three axes: data-level parallelism (DLP), instruction-level parallelism (ILP) and task-level parallelism (TLP). Consequently one

Proc. EOS Conference on Industrial Imaging and Machine Vision (Munich, Germany, June 13-15), European Optical Society, 2005, 39-49.

can encounter one or more of these processors in a smart camera. For the software this means that if the chosen hardware does not meet the requirements, the application must be rewritten for a new platform. These problems can be avoided by encapsulating the parallelism. In the SmartCam project, algorithmic skeletons [2] are used to express the data parallelism inherent to low-level image processing. But, since different operations (low, intermediate and high level image processing) run best on different kinds of processors, we need to exploit task parallelism as well. For this an asynchronous remote procedure call (RPC) system is used, which was optimized for low-memory and sparsely connected systems such as the Philips laboratories line of smart cameras. The software framework eventually uses a normal C-interface to the user in which the skeleton calls are implicitly parallelized and pipelined.

The structure of this paper is as follows: section 2 reviews some related work. Section 3 discusses and profiles smart camera applications. Section 4 describes our smart camera architecture and characteristics. Section 5 describes our programming environment and section 6 discusses some optimizations. Section 7 presents some results from our system and finally section 8 draws conclusions and points to future work.

2 Related Work

Because of the increased power and area efficiency, SIMD arrays, and in particular linear processor arrays (LPAs), are still frequently used in embedded applications. Vision accelerator boards are employed in real-time control systems where there is enough room to have a workstation. They contain LPAs (IMAP-CE [3]), DSPs (FUGA [4]), or GP processors (Genesis*Plus* [5]). The IMAP-CE uses a data-parallel C extension called 1DC [6] to program the LPA, while the FUGA and Genesis*Plus* are programmable in standard C++. All boards provide optimized library routines for common image processing operations. In addition, the Genesis*Plus* uses the library routines to interface with a separate neighborhood processor as well. The use of an explicitly data parallel language makes the IMAP-CE slightly more difficult to program, but also potentially faster. It occupies a place between assembly language, which is always fastest but not realistically used by image processing researchers, and a library-only based approach, which may shield the programmer too much to make any optimizations. It seems that a library-based system in which the user can descend to a (parallel) programming level, if necessary, is the best approach.

For the more embedded market, with a need to be very small and power efficient, cameras that integrate sensing and processing are emerging (figure 1). Again, DSP (Vision Components [7], iMVS [8]) and GP (Legend [9], Inca 311 [10], mvBlueLYNX [11]) solutions are often used, but single chip LPAs (Xetal [12], added to Inca 311) and even integrated sensor/LPA chips (MAPP2500 [13]) exist as well. Again, all systems are programmable using an image acquisition and processing library, but the single chip LPAs, because of the simplicity of their processing elements, cannot easily be programmed in C. Xetal tries to remedy this by providing a C++-like language called XTC, while the MAPP2500 avoids the problem altogether by only providing a few

algorithms specific to the expected application domain (range imaging). Two of the smart cameras, Inca311 and Legend, are also programmable using graphical programming languages [14]. Both are targeted at industrial inspection, and allow novices in the field of image processing to graphically connect algorithms like sub-pixel edge detection, angle measurements and template matching. Efforts have been made to put such a user interface above a library-based approach, providing another level of abstraction in a single framework.

Figure 1: Philips smart camera product line

3 Application Profiling

The optimal smart camera architecture mainly depends on its application. Each application dictates the type, order and intensity of image processing operations. Consequently, the architecture needs to be scaled up or down. To a varying degree, all application segments look for better performance at lower cost and lower power consumption.

3.1 Mobile-multimedia Processing

This class of applications is characterized by low-cost, moderate computational complexity and low power. The application can usually live with reduced quality of services, e.g., lower-frame rate and compressed video streams. The objective from the point of view of product manufacturers is cost reduction, which translates to reduction in silicon area and power-efficient computation. The latter objective can often be compromised for the former since mobile devices are active for a short period of time compared to standby duration and the battery energy is wasted mainly in the standby phase.
The computational complexity for this class of applications varies from 300 MOPs for basic camera pre-processing for image resolutions of 640*480 at 30 frames per second (fps) to 1.5 GOPs for more complex pre-processing including auto white-balance, exposure time control (about 150 operations per pixel). In future, the computational complexity will increase, because the tendency in mobile video is to increase resolutions and frame rates, combined with more complex applications.

3.2 Intelligent Interfaces and Home Robotics

This class of applications corresponds to emerging house robots with vision features. Typical examples include intelligent devices with gesture and face recognition [15], autonomous visual guidance for robots, and smart home surveillance cameras. These devices cover the medium-cost range. From a user point of view, the response times and accuracies of the intelligent devices are of high importance and imply faster burst

performance. They need to operate in uncontrolled environments (i.e. varying light conditions), and smarter (complex) algorithms are needed to achieve the desired performance. The power aspect remains an issue especially in standalone modules such as battery powered surveillance robots.
Because of the extra intelligence needed in this class of applications, the number of operations per pixel is in the order of 300 or more. This translates to more than 3 GOPs for a 30 fps VGA size video stream. Even though the devices can exhibit long idle times until they are excited by an event, e.g., an intruder in a scene, in their active duration the same degree of performance is required to guarantee real-time behavior.

3.3 Industrial Vision

Unlike the previous two cases, applications in this segment are cost-tolerant and more emphasis is given in achieving top-performance sometimes at a given power budget. The emergence of smart cameras has made it possible for various industrial applications to replace large expensive PC based vision systems with compact and light modules having different vision functionalities.
In this class of applications a number of basic pixel-level operations such as edge detection, enhancement, morphology, etc. need to be performed. Because of the high video rates often in excess of 100 fps, the computational complexity is easily more than 4 GOPs.

4 Smart Camera Architecture

In the SmartCam [16] environment, an application designer will be able to generate an optimal smart camera hardware configuration for his specific domain, based on his application code and various constraints such as size, cost and power consumption. However, for this approach to be feasible it is necessary to restrict the search space by imposing an architecture template. Based on the previous expertise and projects our architecture template will consist of a sensor, reconfigurable logic (i.e. FPGA) to interconnect the sensor and processors and to perform some fixed applications such as lens correction, one or more LPA(s), instruction level parallel processor(s), memories, and communications peripherals. The general template can be parameterized with regard to resolution, number of processing elements (PEs) and PE functionality, data width, the amount and type of functional units, etc.

The choice of an LPA is simple, because it is perfectly suited for the data parallelism inherent to low-level image processing operations. ILP processors, such as very long instruction word (*VLIW*) and *superscalar* processors can execute multiple independent instructions per cycle, exploiting a finer-grained level of parallelism than LPAs. This is necessary because higher-level vision processing tasks are too irregular to execute on LPAs. Figure 2 depicts the architecture of a Philips smart camera based on the architecture template discussed before. The architecture uses a Xetal as LPA processor. The Xetal, a SIMD processor, includes 320 processing elements, each with one ALU. It is suitable for many low-level operations to exploit DLP. Trimedia [17] is a

VLIW example; it can execute five operations per cycle. It is suitable for high-level operations to exploit ILP.

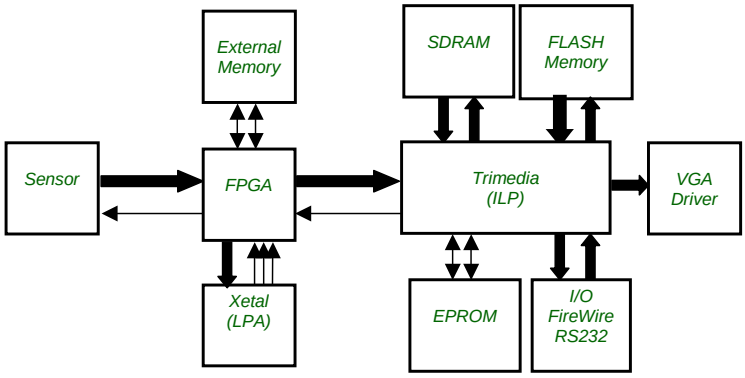


Figure 2: Philips smart camera architecture

Since the architecture comprises reconfigurable logic, it is possible to change the architecture depending on the application. The Philips RoboCup team (autonomous soccer robots) [18] uses a color vision system for localization of the ball, goals and players in the field (see figure 3a).

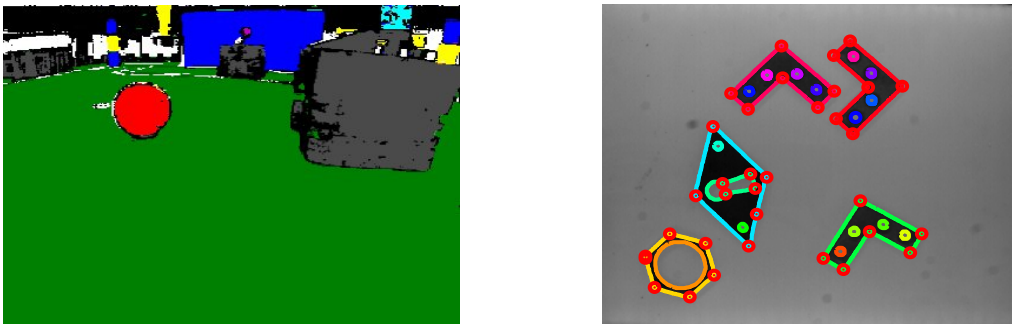
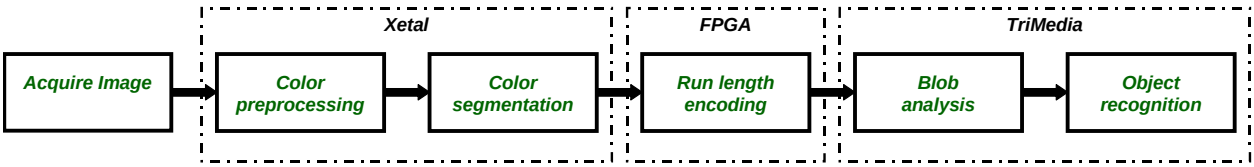
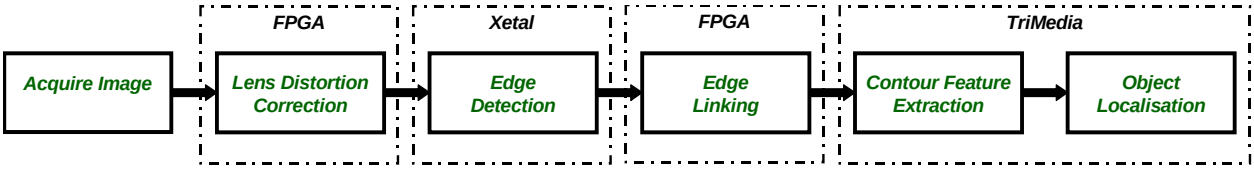


Figure 3: a) Color segmented image of Philips RoboCup vision system, b) Contour segmented image of industrial application

The application can be split up in different parts like color segmentation and object detection. The partitioning of the application onto the architecture is shown in figure 4a. In industrial applications, contour information is often used to inspect or localize objects (see figure 3b). The partitioning of the application is illustrated in figure 4b. Notice that in both cases the FPGA is also used to order (run length encoding and edge linking) the data from Xetal in such a way that the TriMedia can process it further with minimal effort.



(a)



(b)

Figure 4: Partitioning examples of the vision application on processing elements of the architecture: a) Blob-based RoboCup color application, b) Contour-based industrial application

5 Programming

Our programming environment [16] is based on C, to provide an easy migration path. In principle, it is possible (although slow) to write a plain C program and run it on our system. In order to exploit concurrency, though, it is necessary to divide the program into a number of image processing operations, and to apply these using function calls. Parts of the program, which cannot easily be converted, can be left alone unless the speedup is absolutely necessary. The main program, which calls the operations and includes the unconverted code, is run on a *control processor*, while the image processing operations themselves are run on the *coprocessors* that are available in the system (the control processor itself may also act as a coprocessor). Only this main program can make use of global variables; because of the distributed nature of the coprocessor memory, all data to and from the image processing operations must be passed using parameters.

5.1 Within-operation parallelism

The main source of parallelism in image processing is the locality of pixel-based operations. These low-level operations reference only a small neighborhood, and as such can be computed mostly in parallel. Another example is object-based parallelism, where a certain number of possible objects or regions-of-interest must be processed. Both cases refer to *data parallelism*, where the same operation is executed on different data (all pixels in one case, object pixels or objects in the other).
Data parallel image processing operations map particularly well on linear SIMD arrays. However, since we don not want the application developer to write a parallel program, we need another way to allow him to specify the amount of parallelism present in his operations. For this purpose, we use *algorithmic skeletons*. These are *templates* of a certain computational flow that do not specify the actual operation, and can be thought of as higher-order functions, repeatedly calling an *instantiation function* for every computation. Take a very simple binarization:

```
for (y=0; y<HEIGHT; y++)
        for (x=0; x<WIDTH; x++)
                out[y][x] = (in[y][x]>128);
```

Using a higher-order function, **PixelToPixelOp**, we can separate the structure from the computation. **PixelToPixelOp** will implement the loops, calling **binarize** every iteration:

```
int binarize(int value)
        return (value>128);

void PixelToPixelOp(int (*op)(int),
        int in[HEIGHT][WIDTH], int out[HEIGHT][WIDTH])
        for (y=0; y<HEIGHT; y++)
                for (x=0; x<WIDTH; x++)
                        out[y][x] = op(in[y][x]);

PixelToPixelOp(binarize, in, out);
```

Note that implementing **PixelToPixelOp** column-wise instead of row-wise - by interchanging the loops - does not change the result, because there is no way for **op** to reference earlier results (side effects are not allowed). It can be said that by specifying the inputs and outputs of the instantiation function, the skeleton characterizes the available parallelism. So, by choosing a skeleton, the programmer makes a statement about the parallelism in his operation, while not specifying how this should be exploited. This freedom will allow us to optimally map the operation to different architectures. Another benefit is that the image processing library normally shipped with DSPs and other image processors is replaced by a skeleton library, which is more general and thus less in need of constant updates.

5.2 Between-operation parallelism

An image processing application consists of a number of operations described above, surrounded by control flow constructs. In order to provide an easy migration path, these operations will be called as higher-order functions, although they are implemented using *source-to-source transformations*. Furthermore, because our hardware platform is heterogeneous, it is important that multiple of these operations are run concurrently, as not all processors can be working on the same computation. We are therefore using asynchronous RPC calls as a method to exploit this task-level parallelism.
In RPC, the *client* program calls *stubs* that signal a *server* to perform the actual computation. In our case, the application is the client program running on the control processor, while the skeleton instantiations are run on the coprocessors. This alone does not imply parallelism, because the stub waits for the results of the server before returning. In asynchronous RPC, therefore, the stub returns immediately, and the client has to *block* on a certain operation before accessing the result. This allows the client program to run concurrent to the server program, as well as multiple server programs to run in parallel.
However, this still has the disadvantage of requiring the client program to wait on the completion of an operation before passing its result to another one, even though it never uses the results itself. To address this problem, we are using MultiLisp's [19] concept of *futures*, placeholder objects that are only blocked upon when the value is needed for a computation. Since simple assignment is not a computation, passing the value to a stub doesn't require blocking; once the called function needs the information, it will block itself until the data is available, without blocking the client program:

```
while(1)
        Read(in);
        PixelToPixelOp(op1, in, inter1);
        PixelToPixelOp(op2, in, inter2);
        PixelReductionOp(op3, inter1, inter2, out);
        /* ... Concurrent client code ... */
        block(out);
        /* ... Dependent client code ... */
```

6 Optimizations

While our futures-like implementation is much less elaborate than MultiLisp's (requiring, for example, explicit blocks on results, although these could be inserted by the compiler), it does tackle two other problems: data distribution and memory usage. Both originate from our architecture template, which features distributed-memory processors with a relatively low amount of on-chip memory.

6.1 Data distribution

The data generated by most image processing operations is not accessed by the client program, but only by other operations. This data should therefore not be transported to the control processor. In order to achieve this, we make a distinction between images (which are streams of values) and other variables.
Images are never sent to the control processor unless the user explicitly asks for them, and as such no memory is allocated and no bandwidth is wasted. Rather, they are transported between coprocessors directly, thus avoiding the scatter-gather bottleneck present in some earlier work [20]. All other variables (thresholds, reduction results, etc.) are gathered to the control processor and distributed as necessary. The programmer can use them without an explicit request.
The knowledge about which data to send where, simply comes from the inputs and outputs to the skeleton operations, which are derived from the skeleton specification and are available at run time. Coprocessors are instructed to send the output of an operation to all peers that use it as an input.

6.2 Memory usage

Our concern about memory usage stems from the fact that especially SIMD LPAs for low-level image processing may not have enough memory to hold an entire frame, let alone multiple frames if independent tasks are mapped to it. These processors are usually programmed in a pipelined way, where each line of an image is successively led through a number of operations. We would like our system to conserve memory in the same way, and have therefore specified all our skeletons to read from and write to *FIFO buffers*. The distribution mechanism allocates these buffers, and sets up transports as described above. The operations themselves read the needed information from the buffer, process it, and write the results to another buffer. A method is provided to signal that no more data will be forthcoming. This conserves memory, because even a series of buffers is generally much smaller than a frame. Simultaneously, it hides the origin of

the data, making the operations independent of the producers of their input and the consumers of their output.
The price of all this is that operations must consume data in a certain order, and if the source operation doesn't generate it in the correct sequence, a *reordering operation* must be inserted, typically requiring a frame memory. Fortunately, many low-level operations can tolerate different orderings, while more irregular operations are generally run on processors with enough memory.

7 Results

We have implemented a double thresholding edge detection algorithm on our architecture. In this algorithm, the Bayer pattern sensor output is first interpolated, then the Sobel X and Sobel Y edge detection filters are run and combined, the output is binarized at two levels, and finally the high threshold is propagated using the low threshold as a mask image. This final propagation cannot be run on the Xetal, because it requires a frame memory.

Table 1. Timing results of the double thresholding edge detection algorithm

Trial	**Processing time**
Single operation (TriMedia)	115 ms
Split operations (TriMedia)	124 ms
Parallel (Xetal + TriMedia)	67 ms

Three situations were compared: one in which the entire algorithm was implemented in a single operation on the TriMedia, as a baseline for how a sequential application would be written. Next, the operation was split into tasks as described above, and all tasks were mapped to the TriMedia; this shows the overhead caused by the task switching and buffer interaction. Finally, all low-level operations were mapped to the Xetal, while the propagation and display were mapped to TriMedia; this resembles the final situation as it would run on our system (see table 1). Because Xetal has only 16 line memories, the buffers between the filters were one line. On the TriMedia, they were 16 lines, to avoid too much context switching. An allocate-and-release scheme was used on the TriMedia, so that no extra state memory was needed in the filters, and no unnecessary copies were made.
As can be seen, the overhead of running the RPC system is around 8% (with 16-line buffers; the overhead approaches zero if full-frame buffers are used, but that is unrealistic). This seems quite a reasonable tradeoff if we consider that it can now run transparently on the parallel platform, achieving a 42% processing time decrease. Actually, because the filtering and propagation are done concurrently in the parallel case, the processing time is bounded by the slowest operation, which is the propagation.

8 Conclusions and Future Work

We have presented a system in which an application developer can construct a parallel image processing application with minimal effort. Data parallelism is captured by specifying the way to process a single pixel or object, with the system handling distribution, border exchange, etc. Task parallelism of these data parallel operations is achieved through an RPC system, preserving the semantics of normal function calls as much as possible. Results from an actual architecture have shown that the system works, and can achieve a significant speedup by using an SIMD processor for low-level vision processing.
The automatic skeleton instantiation is currently limited to ILP processors, and we wish to include Xetal and FPGA skeletons as well. Furthermore, we want to investigate dynamic image sizes and data types. Finally, an automatic mapping step should combine CPU-, memory-, and bandwidth usages to best determine buffer sizes and assign operations to processors.

References

[1] W. Caarls, *SmartCam: Devices for Embedded Intelligent Cameras*, website, http://www.ph.tn.tudelft.nl/~wcaarls/smartcam, Delft, 2003
[2] M. Cole, Algorithmic Skeletons: Structured Management of Parallel Computation, in *Research Monographs in Parallel and Distributed Computing*, The MIT Press, 1989
[3] S. Kyo, T. Koga, S. Okazaki, and I. Kuroda, A 51.2 gops scalable video recognition processor for intelligent cruise control base on a linear array of 128 four-way vliw processing elements, in *IEEE Journal of Solid State Circuits*, November 2003
[4] Philips Applied Technologies, *Parallel Input Vision Processor Fuga*, website, http://www.apptech.philips.com/industrialvision/pdf_files/fuga.pdf
[5] Matrox Imaging, *GenesisPlus Vision processor board based on Motorola's PowerPC™ with AltiVec™ technology*, website, http://www.matrox.com/imaging/support/old_products/genesisplus/b_genesisplus.pdf
[6] S. Kyo, S. Okazaki, and I. Kuroda. An extended c language and compiler for efficient implementation of image filters on media extended micro-processors, in *Proceedings of ACIVS 2003 (Advanced Concepts for Intelligent Vision Systems*), Gent University, 2003
[7] Vision Components, *VC series smart cameras*, website, http://www.vision-components.de/products.html
[8] Fastcom Technology, *iMVS series intelligent machine vision systems*, website, http://www.fastcom-technology.com/pages/products/process/userguides/iMV%S_Overview_eng.pdf.
[9] DVT Sensors, *Legend series SmartImage sensors*, website, http://www.dvtsensors.com/products/LegendManager.php
[10] Philips Applied Technologies, *Inca 311: smart Firewire™ camera with rolling shutter sensor*, website, http://www.apptech.philips.com/industrialvision/pdf_files/inca311.pdf
[11] Matrix Vision, *mvBlueLYNX*, website, http://www.matrix-vision.com/pdf/mvbluelynx_e.pdf.

[12] A.A. Abbo, R.P. Kleihorst, L.Sevat, P. Wielage, R. van Veen, M.J.R. op de Beeck, and A. van der Avoird, A low-power parallel processor IC for digital video cameras, in *Proceedings of the 27th European Solid-State Circuits Conference,* Villach, 2001
[13] IVP, *MAPP2500 smart vision sensor*, website, http://www.ivp.se/documentation/technology/TheSmartVisionSensor010518.p%df.
[14] Philips Applied Technologies, Clicks Vision Application Builder, website, http://www.apptech.philips.com/industrialvision/pdf_files/clicks.pdf
[15] R.P. Kleihorst, H. Fatemi, and H. Corporaal, Real-time face recognition on a smart camera, in *Proceedings of ACIVS 2003 (Advanced Concepts for Intelligent Vision Systems*), Ghent University, 2003
[16] W. Caarls, P.P. Jonker, and H. Corporaal, SmartCam: Devices for embedded intelligent cameras, in *Proceedings of the 3rd PROGRESS workshop on Embedded Systems,* Mariel Schweizer, 2002
[17] Philips Semiconductors, *Nexperia PNX1300 Series,* website, http://www.semiconductors.philips.com/acrobat_download/literature/9397/75009542.pdf
[18] Philips Applied Technologies, *Philips RoboCup Team,* website, http://www.apptech.philips.com/robocup
[19] R. Halstead, Multilisp: A language for concurrent symbolic computation, in *ACM Transactions on Programming Languages and Systems*, October 1985.
[20] C. Nicolescu and P. Jonker, EASY PIPE - an “EASY to use” Parallel Image Processing Environment based on algorithmic skeletons, in *Proceedings of the PDIVM Workshop*, 2001.