Data- and task parallel image processing on a mixed SIMD-ILP platform using skeletons and asynchronous RPC

Citation for published version (APA):
Caarls, W., Jonker, P. P., & Corporaal, H. (2004). Data- and task parallel image processing on a mixed SIMD-ILP platform using skeletons and asynchronous RPC. In M. Schweizer (Ed.), *Proceedings 5th PROGRESS Symposium on Embedded Systems (Nieuwegein, The Netherlands, October 20, 2004)* (pp. 1-8). STW Technology Foundation.

Document status and date:
Published: 01/01/2004

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Data- and Task Parallel Image Processing on a Mixed SIMD-ILP Platform using Skeletons and Asynchronous RPC

W. Caarls, P.P. Jonker, and H. Corporaal
{W.Caarls, P.P.Jonker}@TNW.TUDelft.nl, h.corporaal@tue.nl
Quantitative Imaging Group, Department of Electrical Engineering
Delft University of Technology, Eindhoven University of Technology

Abstract— **The SmartCam project investigates new opportunities provided by the integration of sensing and processing in a single surveillance-camera sized device. More specifically, it will provide tooling to find an application-dependent mixture of single-instruction multiple-data (*SIMD*) and instruction-level parallel (*ILP*) processors using *design space exploration*. This will allow developers in fields such as robotics, surveillance, and industrial inspection to adapt the hardware to their application, instead of the other way around.**

Since a wide variety of hardware configurations are possible, and since it is undesirable to rewrite the program for every one, we have proposed the use of *algorithmic skeletons* [4] to express data parallelism. An efficient heterogeneous system requires the exploitation of task parallelism as well, though, and we have opted for a more conventional presentation to make the transition to our system easier.

This paper describes our asynchronous remote procedure call (*RPC*) system, optimized for low-memory and sparsely connected systems such as smart cameras. It uses a *futures*[13]-like model to present a normal imperative C-interface to the user in which the skeleton calls are implicitly parallelized and pipelined. Simulation provides the dependency graph and performance numbers for the mapping, which can be done at run time to facilitate dependent branching.

Keywords— **Design Space Exploration, Heterogeneous Architectures, Constrained Architectures, Algorithmic Skeletons, Remote Procedure Call, Futures, Run-time Scheduling**

I. Introduction

As processors are becoming faster, smaller, cheaper, and more efficient, new opportunities arise to integrate them into a wide range of devices. However, since there are so many different applications, there is no single processor that meets all the requirements of every one. The SmartCam [6] project investigates how an application-specific processor can be generated for the specific field of intelligent cameras, using design space exploration.

The processing done on an intelligent camera has very specific characteristics. On the 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 while typically consuming less bandwidth [2]. The architecture template on which the design space exploration is based therefore contains data-parallel (SIMD) as well as instruction-parallel (ILP) processors.

One of the main goals of the project is keeping the system easy to program. This means that one single program should map to a wide range of configurations of a wide range of processors. It also means that the application developer shouldn't have to learn a parallel programming language. The solution presented below is based on using algorithmic skeletons to exploit data parallelism within each operation, while a form of asynchronous RPC allows the operations to run concurrently.

The structure of this paper is as follows: section II reviews some related work. Section III presents our prototype architecture, while sections IV and V describe our programming environment and some optimizations. Section VI details our implementation, and section VII presents some results from our prototype. Finally, section VIII draws conclusions and points to future work.

II. Related work

Even restricting ourselves to systems that fit inside a camera, there exist many different image processing architectures:

- **DSPs**, VLIW processors optimized for signal processing, like the Texas Instruments TMS320C6x series [20] and the Philips TriMedia [19].
- **Vector architectures**, scalars or superscalars with

an SIMD coprocessing unit, like the Berkeley VIRAM [8] project, and to a lesser degree Intel's MMX/SSE [14] and Motorola's AltiVec.

- **SIMD arrays**, among many others, NEC IMAP [21] and Philips XeTaL [1].
- **FPGAs**, which can implement operations in hardware.

Recently, SIMD arrays of VLIW processors have also been developed, like NEC's IMAP-CE [9], and the Stanford Imagine [16] architecture.

Of these architectures, pure SIMD arrays are the most suited for low-level image processing, because they have the largest degree of parallelism, while DSPs work best with irregular problems, allowing different instructions to run within one cycle. The other architectures make a compromise to perform well on both domains, but since we are trying to separate these, it makes sense to choose domain-specific architectures.

Programming environments for image and signal processing applications are also widely ranged. Tightly coupled systems usually have parallel extensions to a sequential language, like Celoxica's Handel-C [3] for FPGA programming, or 1DC [10] for the IMAP cards. More loosely coupled systems usually work with the concept of a *task* or *kernel*, and differ in how these tasks are programmed and composed.

Process networks such as Eclipse [17] allow much freedom in specifying the tasks, but require a static connection network between them. StreamC/KernelC [11], developed for Imagine, reduces the allowed syntax within a kernel, but makes the interconnections dynamic by using *streams*. Their current implementation doesn't support task parallelism, however. EASY-PIPE [12] does, but requires a batch of tasks to be explicitly compiled and dispatched by the user. Their main contribution is the use of algorithmic skeletons to make programming the tasks easier. Finally, Seinstra [18] allows no user specification of the tasks, instead relying on an existing image processing library. It is also limited to data parallelism, but these restrictions allow it to be more transparent to the user, presenting a purely sequential model.

Futures were introduced in the MultiLisp [15] language for shared-memory multiprocessors. Requesting a future spawns a thread to calculate the value, while immediately returning to the caller, which only blocks when it tries to access it. Once the calculation is complete, the future is overwritten by the calculated value. Batched futures [13] apply this concept to RPC, but with the intent to reduce the amount of RPC calls by sending them in batches that may reference each other's results.

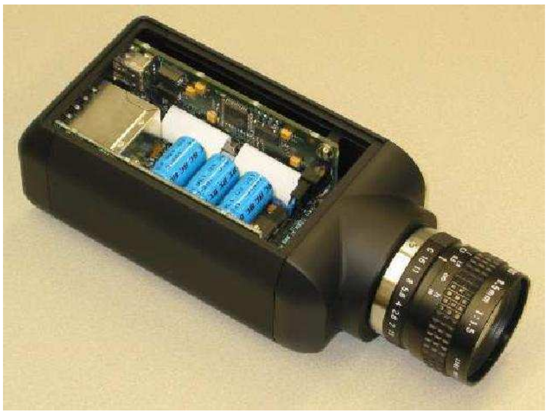


Fig. 1. Inca+ Prototype Smart Camera

III. Architecture

Our prototype architecture is the Philips CFT Inca+ prototype (see figure 1). This is a minimal implementation of our architecture template, consisting of one XeTaL [1] SIMD processor and one TriMedia VLIW processor. There is one video speed channel from the sensor to the XeTaL and one video speed channel from the XeTaL to the TriMedia . The TriMedia can program the XeTaL via I2C. The architecture is described in more detail in [7], and is schematically summarized in figure 2.

The XeTaL chip consists of 320 PEs and a control processor, running at pixel clock. At VGA resolution with a pixel clock of 16MHz and 30fps, it can process over 1000 instructions per pixel, and has enough memory to store 16 image lines. The TriMedia is a 5-way VLIW processor running at 180MHz. At the same video speed that means around 100 operations per pixel, but the pixel accesses may be irregular. An external 32MB SDRAM provides enough storage for most applications at this resolution. The TriMedia runs the pSOS multithreaded real-time operating system. This architecture is suited for image processing because it takes advantage of the fact that image processing applications progress from low-level, high-bandwidth operations to high-level, low-bandwidth operations. One drawback is that because there is no channel from TriMedia to XeTaL , the TriMedia cannot be used as a temporary frame store. This will be remedied in a new prototype platform that is under development.

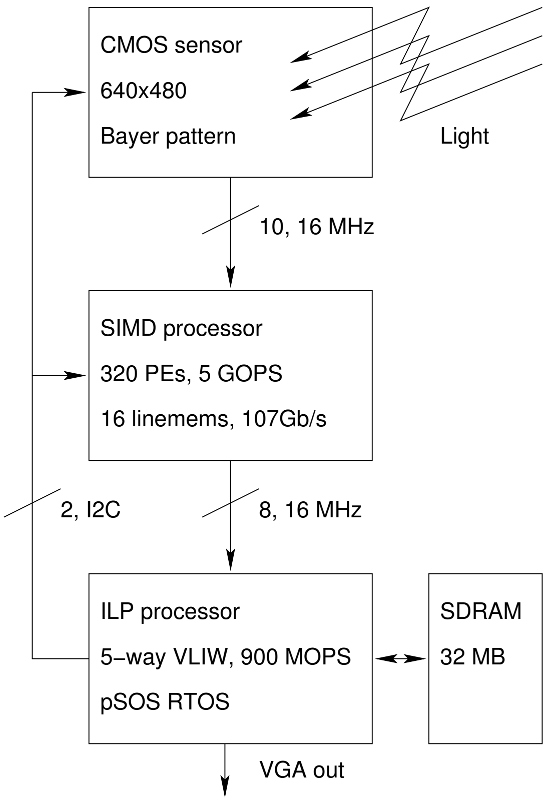


Fig. 2. Inca+ prototype architecture

IV. Programming

Our programming environment 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 sequence of image processing operations, and to string these together using function calls. Parts of the program which cannot easily be converted can be left alone unless the speedup is absolutely necessary.

A. 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 (LPAs, [5]). However, since we don't 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 reversing 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.

B. 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 the instantiation function will be inlined at compile-time to ensure efficiency. 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* which 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:

```
Read(in);
_block(in);
PixelToPixelOp(op1, in, out1);
PixelToPixelOp(op2, in, out2);
/* ...  Concurrent client code ...        */
_block(op1);
_block(op2);
```

However, this still has the disadvantage of requiring the client program to wait on the completion of *in* before proceeding, even though it never uses the results except to pass them on to other RPC calls. To address this problem, MultiLisp introduced the concept of *futures*, placeholder objects which are only blocked upon when the value is needed for a computation. Since simple assignment is not a computation, passing the value to a function 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 ...       */
```

In this piece of code, **PixelReductionOp** still cannot run in parallel with both **PixelToPixelOp**s, though, because it has to wait for the data to become available, even though the client program can continue. Indeed, the RAW dependency makes it impossible to run them concurrently on the same image, but it *is* possible to run them concurrently on *different* images. We therefore introduce the concept of a *composite operation*, which behaves the same as a normal operation, except that it consists of more than one sub-operation. As such, it may wait for data independently of the calling program:

```
ProcessImage(in)
  PixelToPixelOp(op1, in, inter1);
  PixelToPixelOp(op2, in, inter2);
  PixelReductionOp(op3, inter1, inter2, out);
  /* ...  Concurrent client code ...      */
  _block(out);
  /* ...  Dependent client code ...       */

while(1)
  Read(in);
  CompositeOp(ProcessImage, in)
  /* ...  More client code ...            */
```

In this instance, that allows us to run different stages of different images in parallel, because the next **Read** (and **ProcessImage**, once **Read** finishes) can start right after the previous one finished, instead of having to wait for the processing.

V. 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.

A. 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 [12]. All other variables (thresholds, reduction results, etc.) are gathered to the control processor and distributed as necessary. These can be used by the programmer 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.

B. 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 operations 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.

VI. Implementation

Because work on the skeleton instantiation is still in the early stages, we will discuss only our RPC system. This system has been implemented on both a network of workstations (*NOW*) and the Inca+ prototype. The traces in this section were generated on the NOW, while the performance figures in section VII are gathered from the prototype implementation.

The library consists of the following components: a *front-end* that enqueues operations, a *mapper* that maps operations to processors, and a *dispatcher* that dispatches operations, variables, and sets up buffers and transports. If administrative data is not destroyed, a *trace generator* can write a trace once the program finishes. This trace can be used to benchmark individual operations to assist the mapping.

A. Enqueueing

Each call to an RPC stub enqueues that operation in a list. All arguments are passed *by reference*, and are tracked by their address. As such, an operation that uses a variable that hasn't been produced yet can be marked as a consumer of that variable, such that it will be sent as soon as it is available. Note that this means our concept of futures is limited to their use in stub calls, and using an output variable outside of one requires an explicit **_block** (although these may be inserted by a compiler).

As a consequence of the use of buffers, it is necessary to know how many consumers a stream will eventually have, because the data in the (cyclic) buffer may not be overwritten unless it has been read by all consumers. One possibility is to state that every output can be read by only one operation, requiring an explicit *duplication* operation if there is more than one reader. Rather, we have chosen to allow an arbitrary amount of readers, and to require a *finalization* once all consumers have been specified. This makes it easier to conditionally add readers.

Composite operations are a special case. These are implemented using *threads*, but we want them to behave as much as function calls as possible. This means that their arguments should be passed *by value*, and thus we need to make explicit copies of their input arguments. If such an input is a future, a new *instance* is created, which allows the composite operation to independently **_block** on it. The same goes for stream inputs and outputs, which must be independently **_finalize**d (although there is an implicit finalization at the end of the operation). Composite operations may be arbitrarily nested.

We use a *single assignment* semantic, such that each buffer only has one writer. This means that if an output variable is reused, it points to a *different* buffer. As a somewhat counterintuitive consequence, The *first* assignment of an output by a composite operation is used, instead of the last.

B. Mapping

Mapping means selecting the best processor for an operation to run on, optimizing throughput and latency. A *static* mapping can be generated if the course of the program is known. This is the case in parts of the program between data-dependent branches, called *basic blocks*. A problem is that we cannot finish one basic block before starting the next, as this requires a frame memory for every image that is passed between the two, and would effectively reduce task parallelism to zero at each branch.

Fortunately, the granularity of our operations is quite large (operating on images instead of single values), so we can spend a bit of time determining a mapping *dynamically* at run time. Currently, we employ a greedy strategy that will map an operation to the processor that most quickly produces its outputs, according to a simple network flow model. This model assumes that a processor's cycles or a connection's bandwidth are equally distributed over all operations or streams that are mapped to it. If an operation or stream can't use its share, the excess is distributed over the other contenders, etc.

We never map an operation if the source of its inputs isn't known, because that might result in impossibilities if there is no connection between the processors. This occurs when using composite operations, because their outputs are undefined until set by the thread running the operation.

C. Dispatching

After an operation has been mapped to a processor, it can be dispatched. During skeleton instantiation, each operation is assigned an identifier, and this is first sent to the coprocessor. Next, stream arguments are checked; if the stream doesn't have a buffer on the processor, a new buffer is created, and a transport is set up between the source buffer and the newly created one. Then, the buffer id is sent. Finally, non-stream input arguments are marshalled and sent as well.

The operation starts as soon as all its arguments have been received. It will run until it blocks because of reading an empty buffer, or writing a full buffer. The coprocessor then selects a new task that isn't blocked, and so on. In this way, the buffer sizes define the task switching granularity.

Once the operation has finished, it signals the control processor, and transmits its results, if any. If these results are needed by another operation, they are sent back by the control processor. Any futures referencing the results are resolved by copying the data to their addresses, and threads blocking on them are unblocked.

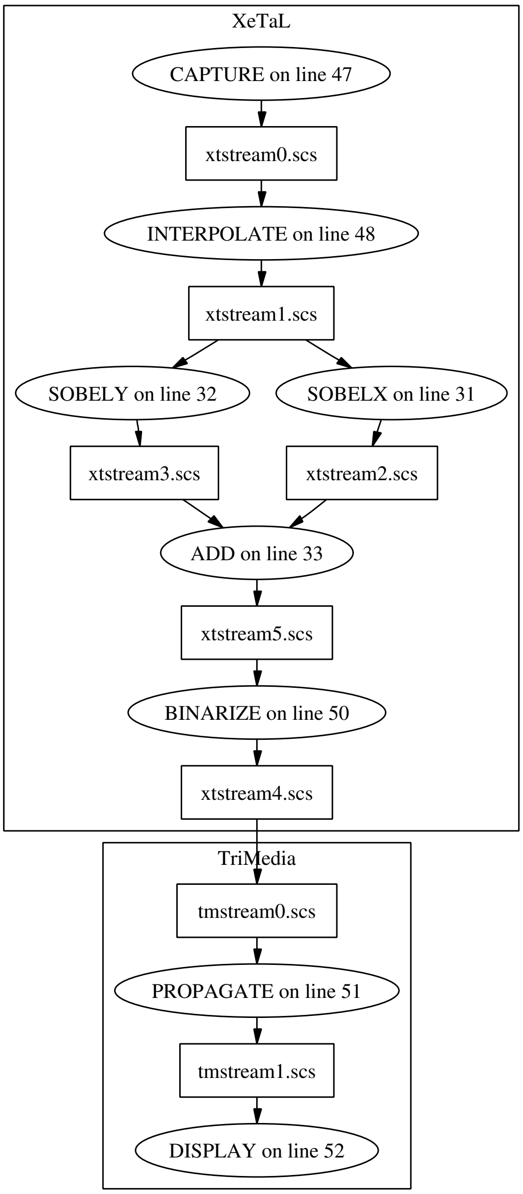
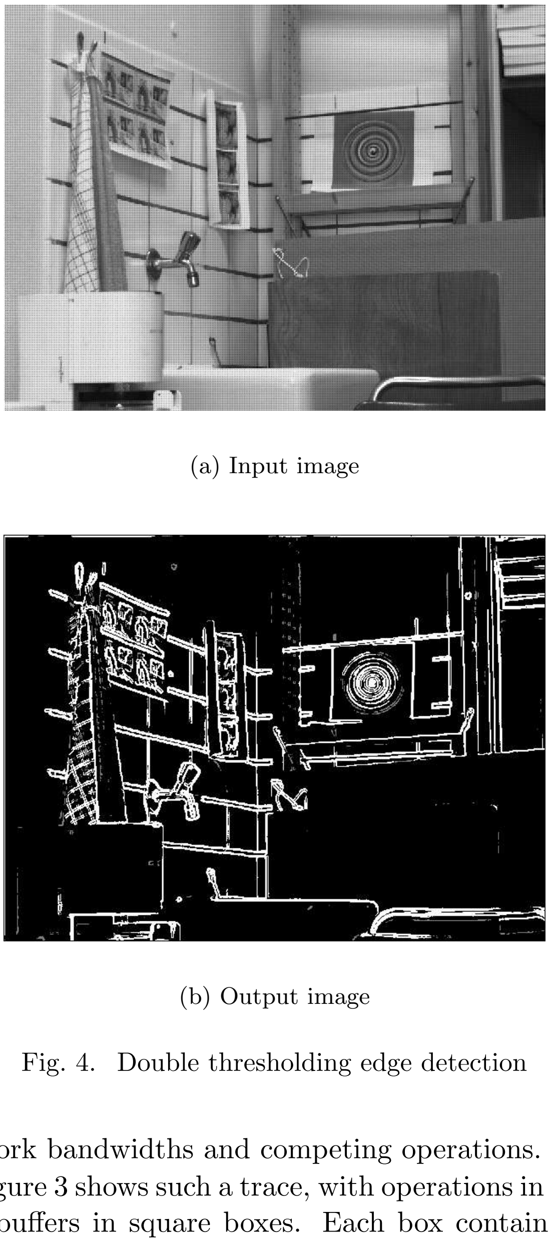


Fig. 3. Trace of a double thresholding edge detection algorithm

D. Trace generation and benchmarking

The mapping process needs information about the performance of each operation on each processor. This requires benchmarking all operations independently. By keeping a trace and storing all streams during a simulation run, we can generate the required information for such an independent benchmark: if we preload the input streams and preallocate output streams, we can get an ideal performance figure in the absence of



(a) Input image

(b) Output image

Fig. 4. Double thresholding edge detection

network bandwidths and competing operations.

Figure 3 shows such a trace, with operations in ovals and buffers in square boxes. Each box contains the name of the file which stores the trace's information.

VII. Results

We have implemented the double thresholding edge detection algorithm of figure 3 on the architecture described in section III. In this algorithm, the Bayer pattern sensor output (figure 4(a)) 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 (figure 4(b)). This final propagation cannot be run on the XeTaL , because it requires a frame memory.

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 shown in the trace, 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.

TABLE I
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

Because XeTaL only has 16 line memories, the buffers between the filters were 1 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. See table I.

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 trade-off 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.

VIII. 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 prototype architecture have shown that the system works, and can achieve a significant speedup by using an SIMD processor for low-level vision processing.

The skeleton instantiation is currently done by hand,

and we want to automate this in the future. Furthermore, we want to augment the system by allowing different data types and data orderings, with on-the-fly insertion of conversion operators. Finally, the mapping should be expanded to include memory usage and buffer sizing as well as CPU and bandwidth usage.

This work is supported by the Dutch government in their PROGRESS research program under project EES.5411.

References

[1] 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, Austria*. Carinthia Tech Institute, September 18–20 2001.

[2] W. Caarls and P.P. Jonker. Benchmarks for smartcam development. In *Proceedings of Acivs 2003 (Advanced Concepts for Intelligent Vision Systems)*. Ghent University, September 2-5 2003.

[3] Celoxica Limited. *Handel-C Language Reference Manual*, 2003. `http://www.celoxica.com/techlib/files/CEL-W0308111132Q-60.pdf`.

[4] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. The MIT Press, 1989. ISBN 0-273-08807-6.

[5] P.P. Jonker. Why linear arrays are better image processors. In *Proceedings of the 12th IAPR International Conference on Pattern Recognition, Los Alamitos, CA*, volume III, pages 334–338. IEEE Computer Society Press, October 1994.

[6] P.P. Jonker and W. Caarls. Application driven design of embedded real-time image processors. In *Proceedings of Acivs 2003 (Advanced Concepts for Intelligent Vision Systems)*. Ghent University, September 2-5 2003.

[7] R. Kleihorst, H. Broers, A. Abbo, H. Embrahimmalek, H. Fatemi, H. Corporaal, and P. Jonker. An SIMD-VLIW smart camera architecture for real-time face recognition. In *Proceedings of ProRISC 2003*, pages 1–7. Technology Foundation STW, November 26-27 2003.

[8] C. Kozyrakis. *Scalable Vector Media Processors for Embedded Systems*. PhD thesis, University of California at Berkeley, May 2002.

[9] 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. *IEEE Journal of Solid State Circuits*, 38(11):1992–2000, November 2003.

[10] 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*, pages 234–241. Ghent University, September 2-5 2003.

[11] P. Mattson. *A Programming System for the Imagine Media Processor*. PhD thesis, Dept. of Electrical Engineering, Stanford University, 2001.

[12] C. Nicolescu and P.P. Jonker. EASY PIPE - an "EASY to use" Parallel Image Processing Environment based on algorithmic skeletons. In *Proceedings of the Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (held in conjunction with IPDPS)*, 2001.

[13] P.Bogle and B. Liskov. Reducing cross domain call overhead using batched futures. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 341–354. ACM Press, 1994. ISBN 0-89791-688-3.

[14] A. Peleg and U. Weiser. MMX technology extension to the intel architecture. *IEEE Micro*, 16(4):51–59, August 1996.

[15] jr. R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[16] S. Rixner. *Stream Processor Architecture*. PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, 2000.

[17] M.J. Rutten, J.T.J. van Eijndhoven, E.G.T. Jaspers, P. van der Wolf, O.P. Gangwal, and A. Timmer. Eclipse: A heterogeneous multiprocessor architecture for flexible media processing. *IEEE Design and Test of Computers: Embedded Systems*, pages 39–50, July/Aug 2002.

[18] F.J. Seinstra and D. Koelma. Lazy parallelization: A finite state machine based optimization approach for data parallel image processing applications. In *Proceedings of the Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (PDIVM 2003)*, 2003. Held in conjunction with IPDPS 2003.

[19] G.A. Slavenburg. *TM1000 Databook*. TriMedia Division, Philips Semiconductors, 1997.

[20] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, September 2000.

[21] Y.Fujita, N.Yamashita, and S.Okazaki. IMAP-Vision: An SIMD processor with high-speed on-chip memory and large capacity external memory. In M. Takagi, editor, *Proceedings of the 1996 IAPR Workshop on Machine Vision Applications*. International Association for Pattern Recognition, 1996.