High Performance Image Processing using TTAs

Marnix Arnold Reinoud Lamberts Henk Corporaal

Delft University of Technology
Department of Electrical Engineering
Section Computer Architecture and Digital Systems
P.O. Box 5031, 2600 GA Delft, The Netherlands
e-mail: {marnix,reinoud,heco}@duteca.et.tudelft.nl

Abstract

In previous ASCI papers ([1], [2]), a processor development framework for Transport Triggered Architectures (TTAs) was presented. This paper discusses the application of this framework to the design of a processor aimed at different image processing algorithms. In particular, gray-scale neighborhood operations are considered. The applicability and advantages of special function units are discussed, and the resulting processor configuration is described.

Keywords: Gray-scale neighborhood operations, Transport triggered architectures, Cosynthesis

1 Introduction

In this paper, we discuss the automated design of an application specific instruction set processor (ASIP). Specifically, we concentrate on generating processors and code for image processing applications, trying to exploit the inherent instruction level type of parallelism in this special class of algorithms. Currently, Delft University of Technology cooperates with Océ Research (Venlo) on practical applications for image processing ASIPs. The goal of this cooperation is to research whether ASIPs for image processing are attractive compared to other, existing solutions.

The processor architecture that will be used to implement the ASIP is a Transport Triggered Architecture called MOVE. It was developed at the Computer Architecture Group of Delft University of Technology. TTAs are a lot like VLIW architectures in that they can perform multiple operations per cycle. The main difference is the way in which operations are executed: whereas in VLIWs instructions specify RISC type operations, in TTAs they specify data transports. Operations are triggered as a side effect of these data transports: the destination of a transport implicitly specifies the kind of operation that will be performed on the data. A MOVE configuration consisting of a transport network (with 9 buses) and function units (FUs) is shown in figure 4. Note that the FUs do not have to be fully connected to the transport network.

The most important advantages of TTAs (when compared to traditional architectures) are their inherent flexibility, scalability and simplicity [1] (resulting in a short design cycle). To exploit these advantages, an automated design framework, called the *MOVE framework* (figure 1), has been developed. It consists of a hardware and a software development subsystem, which are used by an optimizer program to explore the architecture design space. By varying several architecture parameters such as number of transport buses, number and type of function units, etc, it tries to find processor configurations with optimal cost/performance ratios for a given application.

The image processing algorithms that have been implemented are discussed in the next section. The third section describes the development process, and discusses the usefulness and applicability of *special function units* (SFUs). We also evaluate the found processor configuration. In the final section, we present some conclusions and recommendations.

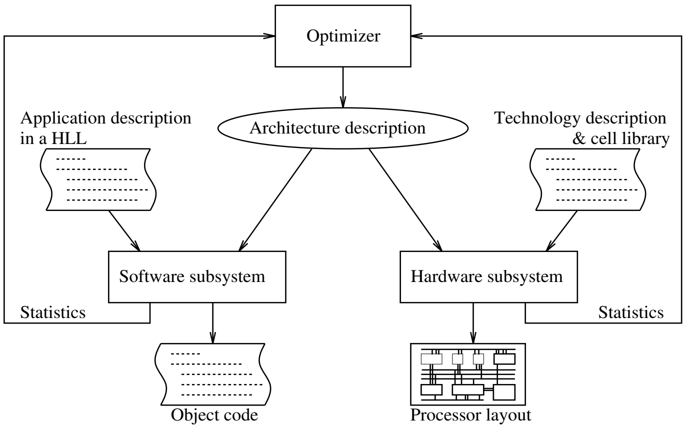

Figure 1: The MOVE framework.

2 Image processing algorithms

For our case study, we concentrated on implementing two examples of gray-scale neighborhood operations [3]: convolution and edge detection on a 3x3 area (figure 2). These operations are part of a larger image processing application. We will discuss both briefly and analyze their potential for parallelism.

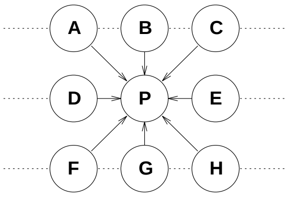

Figure 2: The 3x3 pixel neighborhood.

Convolution

The convolution operation is a *linear* gray-scale operation. For each pixel *P*, a new value P_{out} is calculated from its old value and the values of its neighbors. For the neighborhood shown in figure 2, the operation can be written as:

$$P_{out} = c_0.P + c_1.(A+C+F+H) + c_2.(B+G) + c_3.(D+E)$$

The values of the coefficients $c_{1..3}$ determine the kind of transformation that is performed (e.g. positive values smoothen the image, negative values sharpen it). In principle, all pixels can be processed in parallel since their calculations are independent of each other's new values. Control flow is simple, no branches need to be evaluated during the processing of a pixel. The actual level of parallelism that can be attained in a TTA processor implementation is determined by the maximum number of concurrent operation slots in the processor (upper bound on parallelism), as well as the ability of the operation scheduler (part of the compiler) to fill these slots with actual operations (or moves).

Edge detection

The edge detection algorithm based on the min/max operation is *non-linear*. Each output pixel P_{out} is assigned the difference between the maximum and the minimum value in a neighborhood (3x3 or 5x5) around input pixel *P*, including *P* itself. For the neighborhood shown in figure 2, the operation can be written as:

$$P_{out} = Max\{A...H, P\} - Min\{A...H, P\}$$

While the potential for parallelism is the same as for the convolution operation, the minimum/maximum calculations (requiring lots of branches) make it more difficult to parallelize by a compiler. It will be shown in subsection 3.2 that adding special functionality to the TTA processor template increases the compiler-detected parallelism significantly.

3 The TTA Image Processor design process

In the MOVE framework, two main design criteria are hardware cost and performance. The *solution space* is defined by all possible design points in the 2-dimensional cost-performance space. The *explorer* (or *optimizer*, figure 1) within the framework [2] finds its way through this solution space by iteratively scheduling the application for different architecture configurations. The hardware [1] and software subsystems produce relevant information about these configurations, such as cycle time, costs and number of cycles needed to run the applications. Based on this information, the explorer tries to find a configuration with a better cost/performance[1] ratio, by iteratively reducing the number of available buses, FUs and registers (hardware resource reduction). The resulting points in the solution space lie on a so-called *Pareto-curve* [4] (figure 3, discussed further on in this paper, contains several Pareto-curves). From this curve, the designer chooses a configuration, which is then used by the framework to do connectivity optimization. The explorer removes connections between FUs and the transport network (connectivity reduction), and re-evaluates performance after each subsequent removal. The results are again plotted in a graph, from which the designer chooses the final architecture configuration.

Subsection 3.1 describes the design process and results for the two categories of image processing algorithm as listed in section 2, using only the standard, RISC-like functionality. In subsection 3.2 we describe how specialized function units can improve the quality of the solutions found by the framework. The architecture configuration that resulted from the automatic design process is presented in subsection 3.3. Two special function units that are currently being considered for inclusion in the framework are described in subsection 3.4.

3.1 Implementation with traditional operations

The first step in mapping an application onto a MOVE processor is to write a C or C++ version of the algorithm. This implementation is compiled to traditional MOVE-operations, comparable to those found in most general purpose processors. Critical procedures are identified using profiling tools. The explorer will concentrate on these procedures while searching the design space.

In our case, the critical procedure is the part that calculates the output value for each pixel from its own input value and those of its neighbors. The operation count of the critical procedures of the convolution and edge detections algorithms, using only RISC-like operations, is given in table 1, colums two and three.

Using the RISC-like, default instruction set, the MOVE framework is used to find the optimal TTA configuration for both types of gray-scale neighborhood operations (convolution and min/max edge detection). It turns out that the framework is able to find a much more efficient implementation for the convolution operation than for the edge detection algorithm[2]. This is mainly caused by the large number of branches needed when calculating the greatest or smallest of two numbers. In VLIW architectures like MOVE, such branches can usually be eliminated by means of a technique called *if-conversion* ([4], pp. 94). This is also the case for this application. However, our current compiler is unable to detect in advance register delay-line problems that occur when an attempt is made to software-pipeline the if-converted code. The exact nature of these problems falls outside the scope of this report but is discussed in some depth in [4].

The problems' effect is a rather large steady state of the software pipeline: 8 cycles for edge detection[3], as opposed to 3 for the convolution algorithm, given a very large hardware configuration (e.g. one with cost 400, in figure 3). It can be seen from the graph, however, that the cost/performance curve "Edge detection,

[1] For the sake of exploration speed, a mathematical approximation is used to calculate the hardware cost and cycle time of the architecture, rather than gate- or layout-level circuit information. "Costs are expressed relative to the cost of a [...] adder; cycle time is expressed in nanoseconds." ([4], pp. 140)

[2] When scheduling on an 'ideal' (i.e. very large) processor configuration. This is done to find an upper bound of the compiler-detected parallelism, without being constrained by hardware resources.

[3] Figures are obtained using software-pipelining combined with if-conversion, but without loop unrolling.

Operation	#ops/pixel convolution (no SFU)	#ops/pixel edge-detect (no SFU)	#ops/pixel convolution (Addercmp)	#ops/pixel edge-detect (Addercmp)
add/sub	12	6	12	7
greatest	n/a	n/a	1	4
smallest	n/a	n/a	1	4
mul	4	-	4	-
gt	3	8	1	1
ld	3	3	3	3
st	1	1	1	1
shr	1	-	1	-
total	24	18	24	20

Table 1: Operation counts without and with the Addercmp SFU.

no SFUs" already flattens out at 8 cycles per pixel at a cost of around 200. Any hardware resources that are added beyond this point can not be used to increase performance.

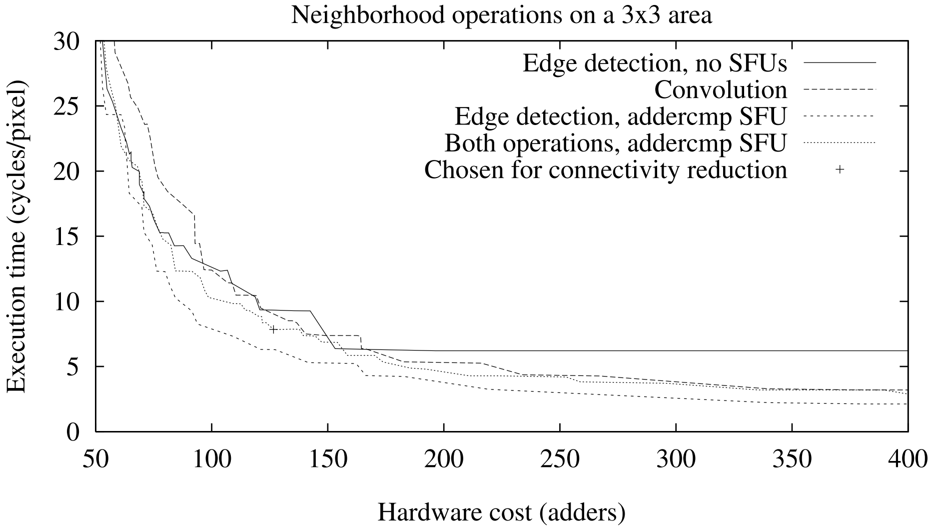

Figure 3: The TTA design space for the 3x3 operations, with and without the special FU.

3.2 Implementation with Special Function Units

An important part of our research is to see if and how the use of *special function units* (SFUs) can improve the quality of solutions produced by the MOVE framework. In this subsection, we describe an SFU that was designed specifically to solve the aforementioned problem with the edge-detection algorithm.

The performance of the edge detect implementation can be increased dramatically by adding a special function unit, the *addercmp* (adder-comparator) FU. It is an extension of an adder which can do conditional assignments, i.e. return the greatest or smallest of two operands as its result. Since this eliminates the branches, it is possible to efficiently schedule (software-pipeline) the critical loop.

This is reflected in figure 3, which shows a significant improvement of the cost-performance ratio. Table 1 shows the operation count of the critical loop when the addercmp FU is used. It turns out that while, for edge detection, the operation count is actually higher than in the initial implementation (20 vs. 18 operations, see table 1, columns four and five), the MOVE compiler schedules the new code much more efficiently, i.e. it exploits the parallelism better.

In the convolution algorithm, the addercmp FU is applicable only twice (for clipping). This does not yield any scheduling gains because these branches could easily be eliminated with if-conversion.

The special functions “greatest” and “smallest” are a cheap extension of functionality, since they are implemented using mostly existing hardware (the adder). The unit’s latency increases with the delay of one selector, but this is outweighed by the scheduling advantage that the added functionality affords. The addercmp unit’s usefulness is actually higher than that of a normal adder, since it can still perform normal additions and subtractions in addition to the “greatest” and “smallest” operations. This is especially noticeable when we combine the convolution and edge detection operations on the same MOVE configuration. The convolution operation needs many additions (adder units), whereas the edge-detection operation needs mostly compares (comparator units). When we replace the adders needed by the convolution operation with addercmp units, the comparators are no longer needed.

3.3 The resulting MOVE processor configuration

Because we want to develop a processor that is equally suited to the convolution and the edge-detection operation, we let the explorer search the design space for both applications simultaneously. After resource optimization, a hardware configuration is chosen from the graph in figure 3. Based on the cost/performance ratios and what we deemed hardware-feasible, a reasonable configuration might be the one indicated with a ’+’. It contains 9 buses and 8 FUs. This configuration is used as the starting point for connectivity reduction, i.e. the explorer attempts to remove unnecessary connections between the FUs and the buses. The resulting configuration is shown in figure 4.

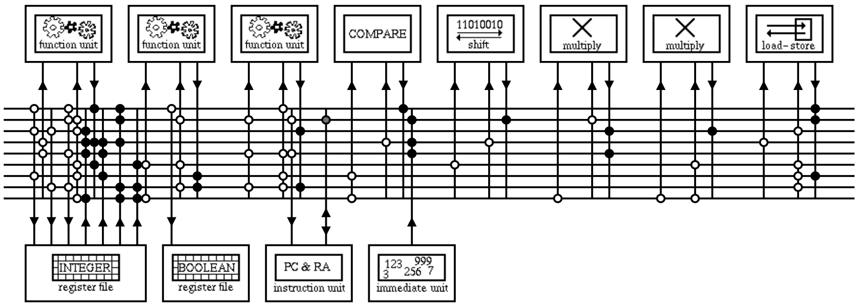

Figure 4: The resulting MOVE processor configuration.

Final performance figures are then obtained by scheduling the applications for the final processor configuration. The convolution operation is executed in 8 cycles per pixel, the edge-detection operation in 7 cycles per pixel (using addercmp FUs).

It is also interesting to see how the chosen configuration performs on the edge detection algorithm for a 5x5 pixel area. While essentially the same as the 3x3 version, the workload increases significantly, since now 25 pixels have to be considered each time, instead of 9. Scheduling the application code for the processor configuration of figure 4 yields a performance of 13 cycles per pixel; scheduling this application on a very large processor yields 4 cycles per pixel. The performance loss due to hardware constraints is comparable to that of the 3x3 edge detection operation: about three times as many cycles are needed (13 vs. 4 and 7 vs. 2, respectively).

3.4 Linebuffer and I/O stream FUs

Given the usefulness of special function units as a way to increase the quality of results produced with the MOVE design framework, it is interesting to see whether there are possibilities for other SFUs. Ideally, an SFU provides a ’short-cut’ for often-repeated tasks that would cost more general-purpose FUs a lot of effort. At the same time it is desirable that the SFU can be used for a wide range of applications, otherwise it would not be very useful to include it in the MOVE framework.

Two other SFUs are currently under development in order to:

- make the image processing applications run efficiently in a more realistic hardware environment.

- move often-used (and reusable) functionality into specialized hardware to keep the code size down (and hence the general-purpose hardware requirements, notably the number of buses).

In the current implementation, it is assumed that the neighboring pixels for any pixel in an image line are always randomly accessible. In a more realistic hardware environment, new pixels are fed into the processor one by one, and only a limited number of pixels can be accessed in any given cycle. Only a limited part of the image can be kept in local memory.

Due to the nature of the neighborhood operations, it is necessary to buffer two (for a 3x3 environment) to four (for a 5x5 environment) lines of the image. New pixels have to be read from an input and stored in linebuffers. In the initial implementation, this was done by a software wrapper around the critical loop of the application, that was not taken into account by the MOVE design framework. As a consequence, this implementation was incomplete in that it could not be viewed as an actual, working program for a MOVE processor. It did suffice to analyze MOVE performance on the critical loop, though.

In order to make the programs map onto real MOVE hardware, it is necessary to move the software wrapper's functionality into the algorithm code. It is desirable to add as few statements to the critical loop as possible, because these will almost certainly cause performance degradation.[4]

A linebuffer FU is being designed to move the part of the software wrapper that takes care of the buffering into hardware. It replaces three loads and a store, as well as the memory address calculations (add operations) involved with these. Pixel values can be read from the buffers through separate ports in parallel, and new pixels are stored through a separate write port. The FU itself keeps track of the position within the linebuffer.

Eliminating the address calculations from the MOVE-code also frees the registers that would be needed to keep track of the memory addresses for the input load, output store and linebuffer loads and store. This might make a smaller register file possible.

The input and output stream FUs are being implemented to meet the requirement that the MOVE image processors must be chainable. They replace the load and store instructions that needed to be executed for each new pixel/result pixel.

4 Evaluation and conclusions

In this paper, we showed how the MOVE development framework can be applied to finding solutions for digital image processing applications. The explorer enables a search through a very large design space within reasonable time. Thus it is possible to compare many design alternatives with each other without having to invest a lot of manual design effort.

The framework can also be used to exploit the flexibility and reusability of the MOVE architecture. It is possible to find a processor that is optimized for one application, but it is equally possible to find one dedicated to a whole class of applications (in our case, different neighborhood operations with different neighborhood sizes).

A large part of designing the MOVE processor is done automatically. However, a lot of manual interaction is still needed when Special Function Units are considered. Currently, these need to be called explicitely from the application code if they are to be used. Thus the decision whether to use an SFU has to be made beforehand, by the designer; it is not included in the automatic design space exploration phase. Future research will concentrate on automating this decision.

References

[1] Henk Corporaal and Reinoud Lamberts. TTA processor synthesis. In *First Annual Conf. of ASCI*, May 1995.

[2] Jan Hoogerbrugge and Henk Corporaal. Automatic Synthesis of Transport Triggered Processors. In *First Annual Conf. of ASCI*, May 1995.

[3] Anil K. Jain. *Fundamentals of Image Processing*. Prentice Hall, 1989.

[4] Jan Hoogerbrugge. *Code Generation for Transport Triggered Architectures*. Delft University of Technology, 1996.

[4] It could be argued that some or even all of the extra statements could be scheduled in parallel with existing statements, but this may not be possible because of hardware resource constraints.