Laura: Leiden Architecture Research and Exploration Tool

Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis, Ed Deprettere

Leiden Embbeded Research Center,
Leiden Institute of Advanced Computer Science (LIACS),
Leiden University, The Netherlands
{claus,stefanov,kienhuis,edd} @liacs.nl

Abstract. At Leiden Embedded Research Center (LERC), we are building a tool chain called *Compaan/Laura* that allows us to map fast and efficiently applications written in Matlab onto reconfigurable platforms. In this chain, first the Matlab code is converted automatically to executable Kahn Process Network (KPN) specification. Then a tool called *Laura* accepts this specification and transforms the specification into design implementations described as synthesizable VHDL. In this paper, we present our methodology implemented in the *Laura* tool, to automatically convert KPNs to synthesizable VHDL code targeted for mapping onto FPGA-based platforms. With the help of *Laura*, a designer is able to either fast prototype signal processing and multimedia applications directly in hardware or to extract very fast valuable low-level quantitative implementation data such as performance in terms of clock cycles, time delays and silicon area.

1 Introduction

The potential of achieving high-performance implementations onto FPGA-based systems (platforms) has been demonstrated by the FPGA research community for applications in the domain of signal processing, multimedia, and imaging. These performance improvements depend very much on the expertise of the hardware designer, who has to possess an accurate knowledge of the underlying FPGA platform and the application. Moreover, the mapping of applications onto this type of platforms is in most cases done manually, which leads to a slow, difficult, and error prone design process. Therefore, we have developed a methodology that allows fast and efficient mapping of a class of multimedia and signal processing applications onto FPGA-based platforms. Part of this methodology is captured in the *Laura* tool that we present in this paper. Central to our methodology is the use of the Kahn Process Network (KPN) [3] model of computation to specify applications. The Laura tool accepts applications written in this KPN model and produces synthesizable VHDL code that implements the application for a specific FPGA platform.

Our methodology uses the KPN model of computation as it is a convenient model to specify imaging applications like Stereo Vision, multimedia applications like MJPEG, and classical signal processing applications like Digital Beam-forming. The model reveals the inherent parallelism of an application that is exploited when mapping the application onto FPGA platforms that are inherently fine-grained parallel platforms.

The KPN specification represents an application in terms of distributed control and distributed memory, which in our case is derived from a sequential code written in Matlab using a tool called *Compaan*. The distributed control and distributed memory are key to obtain efficient implementations on FPGAs for stream oriented applications. This is in great contrast to the original Matlab code that is using a single thread of control and shared memory. Other work describing the mapping of Matlab code (or C for that matter) onto FPGA uses other computational models like CDFG [2] or CSP [5]. These models are well suited for control dominated applications, but less for stream oriented applications.

We present our methodology to map an application written in Matlab onto an FPGA platform in Section 2. In Section 3, we look in more detail at the Laura tool that we have developed. In Section 4, we explain in more detail, using a running example, how Laura constructs an architecture in VHDL. In Section 5, we present experiments that have been obtained by using Laura for three applications. We conclude this paper in Section 6.

2 Integrating *Laura* in an FPGA-based design flow

The *Laura* tool takes as an input a KPN specification of a given application and generates synthesizable VHDL code that targets a specific FPGA platform. In general, specifying an application as a KPN is a difficult task. Therefore, we use our compiler called *Compaan* [4] that fully automates the transformation of Matlab code into Kahn Process Networks (KPNs). The applications Compaan can handle, have to be specified as parameterized static nested loop programs, which is a subset of the Matlab language. We have designed the Laura tool to operate as a back-end of the Compaan compiler, realizing a fully automated design flow that maps sequential algorithms written in Matlab onto reconfigurable platforms. This design flow is shown in Figure 1.

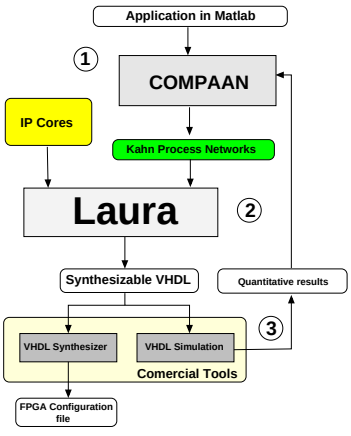

Fig. 1. The Compaan/Laura tool chain

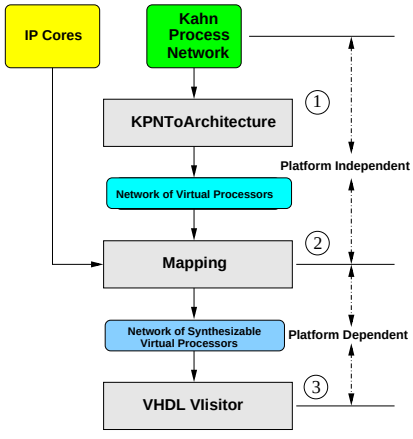

Fig. 2. Steps in Laura

In the first part of the design flow, an application specification is given in Matlab. This is because Compaan only accepts Matlab code. Nevertheless, the design flow is equally applicable to C code or Java code, as their model of computation is equal to the imperative model of computation of Matlab. The Compaan compiler itself is composed of a number of tools. One tool in Compaan performs an aggressive array-dataflow analysis by exploring all data-dependencies in the original program. The result of this tool is a data structure representing the dependence graph of the program. Another tool in Compaan converts this data structure into a KPN specification.

In the second part of the design flow, Laura transforms a KPN specification together with predefined IP cores into synthesizable VHDL code. The IP cores are needed as they implement the functionality of the functions used in the original Matlab program. They are provided to Laura by the *IP cores* box in Figure 1.

In the third part of the design flow, the generated VHDL code is processed by *Commercial Tools* to obtain quantitative results. These results can be interpreted by designers, leading to new design decisions. These decisions are reflected by writing a new Matlab program that exposes, for example, more or less parallelism. For that purpose, we have developed a tool called *MatTransform* that manipulates the Matlab input specification in a Source-to-Source fashion to generate more instances of the application, in which each instance exposes a different level of concurrency without altering the algorithm's behavior [8]. The concurrency is altered by performing high-level transformations like loop unrolling (unfolding), retiming (skewing), and code merging. By rewriting Matlab code, we can explore different mappings of a Matlab algorithm in an efficient way. When an obtained algorithm instance meets the requirements of the designer, the corresponding VHDL output is synthesized by a commercial tool and mapped onto an FPGA platform.

3 The Laura tool

The KPN model of computation [3] assumes concurrent autonomous processes that communicate in a point to point fashion over unbounded FIFO channels, using a *blocking-read* synchronization primitive. Each process in the network is specified as a sequential program that executes an internal function. At each execution (also referred to as an iteration) this function reads/writes data from/to different FIFO channels. Because of the unboundedness of the FIFO channels, the KPN cannot be translated directly into a VHDL representation and mapped onto a hardware platform. Instead, a *blocking-write* primitive is needed next to the *blocking-read*. Also, the FIFO channel sizes now need to be fixed such that no deadlock occurs. Using the method presented in [6], we find a bound on the size of the FIFOs such that the network will not deadlock.

To convert a KPN specification into hardware, we have implemented in Laura a strategy that divides the conversion process into two parts: a platform independent part and a platform dependent part. In the platform independent part, we define an abstract model of the architecture on which we map a KPN application. The model of architecture defines the key components of the architecture and their attributes. It also defines the semantic model, i.e., how the various components interact with each other. Hence,

the architecture also implements autonomous processes, that communicate over channels using blocking read and blocking write semantics.

The abstract architecture model is captured in Laura in terms of a class-hierarchy. This class hierarchy describes a network of virtual processors. Each of them is composed of four units: a *Read unit*, a *Write unit*, an *Execute unit* and a *Controller unit*. The first three units are synchronized by the Controller unit of the processor. Each FIFO channel in the KPN specification is represented by a *Hardware Channel* unit.

In the platform dependent part we start to add information to the abstract architecture model that is specific for the target platform. At this stage, we include IP cores in the Execute units that implement the functions of the original application. Also, we set attributes of the components like bit-width and size of the Hardware Channels.

When an architecture model is established for a given KPN specification, we convert the architecture model into VHDL code using a Visitor Design Structure. For each component in the abstract architecture, we have a small piece of VHDL code that expresses how to represent that component on the target architecture. The visitor structure gives Laura a lot of flexibility. If needed, the output can easily be convert to other formats like Verilog or SystemC.

The steps that make up the Laura tool are shown in Figure 2. In the first step, the *KPNToArchitecture* method converts the given KPN specification into an equivalent network of virtual processors (*Network of Virtual Processors*). This is a platform independent step as no information on the target platform is taken into account. In the second step, platform specific information is mapped onto the abstract architecture model leading to a network of Synthesizable Processors (*Network of Synthesizable Processors*). In the third step, the architecture model is visited by a VHDL visitor to generate the VHDL code.

4 Laura in action

To make clear how a Matlab program is converted into a VHDL code, we explain the steps done in Laura using the very simple Matlab program given in Figure 4. This program consists of three loops. In the first loop, variable `a(j)` is initialized using function `Init`, which represents a *Source*. In the second loop, the function `Compute` performs an operation on `a(j-1)`, introducing a self-loop. Finally, the last loop takes the result of `a(6)` using function `Pass`, representing a *Sink*. The Matlab program is given to the Compaan compiler that converts it into a KPN representation consisting of three different processes. A graphical representation of this KPN is given in the top-part of Figure 3. One process (P1) is the Source, one process implements the `Compute` function (P2), and one process is the Sink (P3). The picture clearly shows the self-loop of function `Compute`. As said before, each process contains a sequential program. In Figure 5, the sequential program for process P2 is given in C++ using the YAPI [1] format.

The sequential program produced by Compaan always follows a particular sequence of events. These events are highlighted by the three different boxes in Figure 5. The first box, contains the code that reads data from input ports. The actual computation takes place in the second box (i.e., performing the function `Compute` from the Matlab program of Figure 4). In the third box, we show the code that writes out data produced

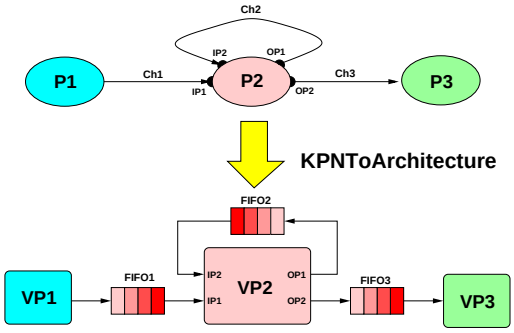

Fig. 3. An example of *KPNToArchitecture* step in Laura

by the computation. The three boxes are enclosed by a for-loop, indicating that the sequence of events needs to be repeated for a given number of times. As a consequence, this process operates in a stream based fashion, an operation model which is very applicable to multi-media and digital signal processing applications.

4.1 KPNToArchitecture

The KPN shown in the upper part of Figure 3 is mapped by the KPNToArchitecture step in Laura onto an abstract architecture model. This model is composed of *Virtual Processors* and bounded hardware communication channels. The lower part of the Figure 3 represents the network of virtual processors that has the same topology as the input KPN. This is because Laura currently performs a *one-to-one* mapping. The three processes *P1*, *P2*, and *P3* are mapped onto the virtual processors *VP1*, *VP2*, and *VP3*, respectively. The KPN unbounded FIFO channels *Ch1*, *Ch2*, and *Ch3* are mapped onto the bounded hardware FIFOs *FIFO1*, *FIFO2*, and *FIFO3*, respectively.

Every virtual processor is composed of four units: a *Read* unit, a *Write* unit, an *Execute* unit, and a *Controller unit*, as shown in Figure 6. The Execute unit is the computational part of a virtual processor. It has *Input Arguments* that provide to the unit the necessary data for execution and *Output Arguments* that are the result of the computation process. In our model, the Execute unit fires when all the input arguments have data and always produces data to all the output arguments. The Read unit is responsible for assigning all the input arguments of the Execute unit with valid data. Since there are more input ports than arguments, the Read unit has to select from which port to read data. This information is stored in the *Control Table* of the Read unit. The *Input Port* is the input interface that connects the virtual processor with a communication channel. The *Output Port* is the output interface that connects the virtual processor with a communication channel. The Write unit is responsible for distributing the results of the Execute unit to the relevant processors in the network. A write operation can be executed only when all the output arguments of the execute unit are available for the write unit. A *Control Table* is used to select the proper Output Port according to the current iteration of the virtual processor.

```
for j = 1 : 1 : 1,
  [ a(j) ] = Init;
end
for j = 2 : 1: 6,
  [a(j)] = Compute (a(j-1));
end
for j =6 : 1 : 6,
  [] = Pass(a(j));
end
```

Fig. 4. A very simple Matlab Program

```
1 void P2 ::main() {
2 for (int i = 2 ; i <= 6 ; i += 1 ) {
3     if (i-2 == 0) {                    READ
4       //reads a token from a channel
5       in_0 = read(IP1);
6     }
7     if (i-3 >= 0) {
8       //reads a token from a channel
9       in_0 = read(IP2);
10    }
11    out_0 = Compute(in_0) ;     EXECUTE
12    if (-i+5 >= 0) {                   WRITE
13      //writes a token to a channel
14      write( OP1, out_0);
15    }
16    if (i-6 == 0) {
17      //writes a token to a channel
18      write( OP2, out_0);
19    }
20  } // for i
21 }
```

Fig. 5. Process P2

The virtual processor's *Controller* synchronizes all the processor's units and keeps track of how many times the processor has already fired. The Read unit and the Write unit can block the next firing when a blocking-read or a blocking-write situation occurs, thereby stalling the complete processor. A blocking-read situation occurs when data is not available at a given input port. A blocking-write situation occurs when data cannot be written to a particular output port.

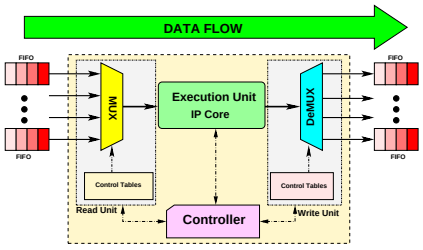

Fig. 6. The Virtual Processor model

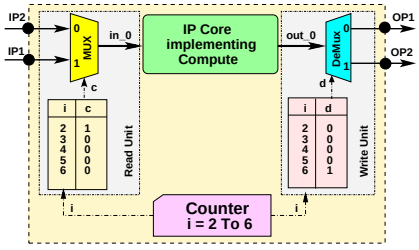

Fig. 7. VP2 in the example shown in Fig. 3

Let us consider the P2 process as it is specified by the sequential code given in Figure 5. This code is analyzed in the KPNToArchitecture step to instantiate the cor-

responding components of the virtual processor. The Read unit is generated based on the information contained between lines 3 and 10. Two input ports, **IP1** and **IP2**, are required to read the input argument **in_0** of the Execute unit. Because a 2-to-1 relationship exists between the input ports and the input argument, a Control Table is needed to select the proper input port for reading the input argument at a particular iteration of the processor. For the example, the Control Table $c = [1, 0, 0, 0, 0]$ is derived based on the number of firings (line 2) and the **if** statements from lines 3 and 7. The Write unit is instantiated according to the lines 12 to 19. It requires two output ports **OP1** and **OP2** to write the output argument **out_0** of the Execute unit. Again a Control Table is derived based on the number of firings (line 2) and the **if** statements from lines 12 and 16. The Control table is equal to $d = [0, 0, 0, 0, 1]$. The Control unit of the processor is instantiated as a counter that iterates i from 2 to 6. For the Execute unit an interface is defined, based on the information contained in line 11. This interface is used again in the Mapping step (Figure 2) when an IP core is connected to the Execute unit. The complete virtual processor that corresponds to process P2, is shown in Figure 7.

4.2 Mapping

The Mapping step is used to include additional information to the abstract architecture model. This is information about the IP cores used by the virtual processors and the bit-width of data. At this step, the width and size of a hardware channel is provided. Furthermore, the notion of a clock event is taken into consideration.

We use IP cores in designing new hardware applications to reduce the design time. This means that we add in the Mapping step the functionality of the Execute unit in terms of an IP core. In order to select the appropriate IP core, the Mapping step searches through a library of predefined cores until it matches the required functionality. The found IP core is subsequently associated to the Execute unit of the virtual processor. For the IP cores which are pipelined, additional information needs to be provided to the Control unit to accommodate the control for the pipelining.

The final result of the mapping step is an annotated architectural model called *Network of Syntesizable Virtual Processors* (NSVP) that is targeted to a particular FPGA platform.

4.3 Visitor

The last step of Laura generates the correct VHDL code for the NSVP structure. First, the communication network is generated, followed by the various processors, and finally a test bench. Within the Visitor there is a well-defined relationship between the components of the abstract architecture model and its representation in VHDL. This means that we have a VHDL template for each component. For example, there is a template for the various units in a processor as well as for the processor itself. The relationship is often one-to-one, but for example in the case of the hardware communications channels, a one-to-many relationship exists. A hardware communication channel operates as a data buffer that can be realized using flip-flops, a look-up table, or internal BRAM memory. This gives the Visitor a lot of flexibility to derive alternative VHDL code taking advantage of specific elements of the target platform.

5 The experiments

Our experimental results are obtained by evaluating the synthesizable VHDL code generated by Compaan/Laura for three computational intensive algorithms. The first one is the *Sequence Alignment* algorithm [7] from the field of bio-informatics. Using the unfolding transformation provided by the MatTransform [8] tool box of Compaan, we generate three different networks. The application specific processor uses an IP core called *Match* that is composed of two adders and a comparator. The second algorithm is the implementation of the 2D-DCT function that is used in data compression algorithms such as MJPEG. In this case, we used the freely available 2D-DCT IP core from the Xilinx web site. The third one implements the QR factorization algorithm used in signal processing applications. It has two IP cores, *Vectorize* and *Rotate*, provided by QinetiQ, Ltd [10]. Table 1 shows the complexity of the input KPNs given by the number of processors and communication channels that has to be handled by the Laura tool. The complexity of the IP cores used to implement the application specific processor is given by the number of hardware multipliers and the pipeline depth used to implement the core.

Experiment	*No. of Processors*	*No. of Channels*	*Pipeline Stages*	*Multipliers*
Sequence Alignment	7	13	0	0
Seq. Alignment Unfold 2x2	10	40	0	0
Seq. Alignment Unfold 3x3	15	83	0	0
2D-DCT	4	4	92	6
QR(Rotate, Vectorize)	5	18	55, 42	8, 8

Table 1. The Process Network complexity

For each benchmark algorithm, a description of the algorithm in Matlab was written and passed through Compaan and Laura. We verified the hardware in two ways. The first way is by simulating the generated hardware using a VHDL simulator and comparing the results to the output of the algorithm executed in the Matlab interpreter. The second way is by implementing the generated hardware onto our reconfigurable platform and comparing the results to the Matlab output. The VHDL simulator provided the total number of cycles needed to execute a given algorithm, as shown in the **Cycles** column of

Experiment	*Cycles*	*Clock delay (ns)*	*Used Slices*	*Used Area Virtex II-6000*
Sequence Alignment	865	16.030	1321	3%
Seq. Alignment Unfold 2x2	466	15.751	3127	9%
Seq. Alignment Unfold 3x3	293	18.511	5874	17%
2D-DCT	364	19.733	1610	4 %
QR(N=7,T=21)	19181	24.390	11270	33 %

Table 2. Experimental Results

Table 2. We use the XST synthesizer and the Xilinx Foundation 5.1i tool to synthesize, place, and route the output of Laura. The clock delay and the total amount of slices needed to implement the networks onto a Virtex II-6000 are also provided in Table 2.

To study the overhead introduced by our methodology in terms of cycle delays and area (i.e., used slices), we conducted a second experiment. In this experiment, we compare a single IP core with the same core embedded in a network. For a single IP core we determine its clock speed and area and compare this to the speed and area taken by the same IP core used in an application network. This gives an indication about the overhead introduced by our methodology. Table 3 shows the delays and the area used

Experiment	*Working Processor*	*Clock Delay*	*Slices*	*Delay Overhead*	*Area Overhead*
Sequence Alignment	Match	6.156	66	2×	20×
Seq. Alig. Unfold 2x2	4×Match	6.156	264	2×	11.8×
Seq. Alig. Unfold 3x3	9×Match	6.156	594	3×	10×
2D-DCT	2D-DCT	13.656	1365	1.4×	1.17×
QR	Vectorize, Rotate	15.862	3442	1.5×	3.27×

Table 3. Trade off between Computation and Communication

by the IP cores, the influence of communication on clock delay (**Delay Overhead**), and the used area (**Area Overhead**). We notice that for fine-grained core implementations the area needed to communicate data in a distributed way is dominant. For example, in case of Sequence Alignment, 20 times more area is needed then a stand-alone version of the *Match* IP core. The communication takes more that 2 times longer in terms of clock-delay than the stand-alone version, due to the routing of the hardware channels on the FPGA. The network of embedded coarse-grained cores, i.e., 2D-DCT, Vectorize and Rotate, introduce considerable less clock-delay than the network of embedded fine-grained cores, i.e., Match. The area overhead depends mainly on the network complexity in terms of channels used. See the difference in number of channels between 2D-DCT and QR in Table 1.

6 Conclusions and Limitations

In this paper, we have presented the Laura tool that implements our methodology to map KPNs generated by the Compaan tool onto a reconfigurable platform such as FPGAs. Although the tool generates only VHDL code, it can be reconfigured to generate other kinds of output, such as Verilog or SystemC. A number of experiments have been conducted for applications in the field of bio-informatics, image processing, and signal processing. The experiments show that we are able to derive fully automatically a hardware implementation from Matlab code. Because Laura implements Kahn Process Networks into hardware, it is well suited for stream oriented applications. Laura is not suited to map control dominated applications. To study the impact of the KPN model on the hardware realization, we investigated the trade off between a stand-alone IP core and an integrated IP core. We found that for more coarse-grained IP cores, the presented methodology gives the best results.

A number of limitation can still be found in Laura. The first issue is that Laura can handle only FIFO communication between processors. High-level code transformations, such as unfolding and skewing, can introduce out-of-order communication between processors [9]. In such case a FIFO can no longer be used in the communication between processes. Future work includes extending the communication components to include this out-of-order communication. The second issue is that Laura generates hardware implementations for non-parameterized KPN models, while Compaan is capable of deriving parameterized descriptions. Future work will focus on generating parameterized hardware networks. The third issue is that communication channels are not always used at their full capacity. We would like to collapse some of these channels onto one channel to share its hardware to reduce communication requirements.

7 Acknowledgments

We would like to acknowledge Alexandru Turjan of the LERC group, Leiden University, for his very valuable input and sharing his insights on the Laura work and his substantial effort to integrate the Laura work with Compaan. Also, we would like to thank to Steven Derrien for his insides toward the processor synchronization issues.

References

1. E. de Kock, G. Essink, W. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers. YAPI: Application modeling for signal processing systems. In *Proc. 37th Design Automation Conference (DAC'2000)*, pages 402–405, Los Angeles, CA, June 5-9 2000.
2. M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee. A system for synthesizing optimized fpga hardware from matlab. In *Proc. Int. Conf. on Computer Aided Design*, San Jose, CA, Nov. 2001.
3. G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.
4. B. Kienhuis, E. Rypkema, and E. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES)*, San Diego, USA, May 2000.
5. I. Page. Constructing hardware-software systems from a single description. In *Journal of VLSI Signal Processing, 12(1):87–107*, 1996.
6. T. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berkeley, 1995.
7. T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology, 147:195-197*, 1981.
8. T. Stefanov, B. Kienhuis, and E. Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *10th Int. Symposium on Hardware/Software Codesign (CODES'02)", pp. 7-12, Estes Park, Colorado, USA*, May 6-8, 2002.
9. A. Turjan, B. Kienhuis, and E. Deprettere. Realizations of the extended linearization model in the compaan tool chain. In *proceedings of the 2nd Samos workshop*, Samos, Greece, Aug. 2002.
10. R. Walke, R. Smith, and G. Lightbody. 20Gflops QR processor on a Xilinx Virtex-E FPGA. In *Proc. SPIE Advanced Signal Processing Algorithms, Architectures, and Implementations X*, pages 300 – 310, 2000.