

# The PAULA Language for Designing Multi-Dimensional Dataflow-Intensive Applications

Frank Hannig, Holger Ruckdeschel, and Jürgen Teich

Hardware/Software Co-Design, Department of Computer Science  
University of Erlangen-Nuremberg, Germany

**Abstract.** In this paper, we present the PAULA language which is designed for modeling dataflow-intensive applications. The language is intended for designing highly parallel algorithms at instruction, data, and loop level parallelism. The PAULA language allows very compact and efficient behavioral descriptions and serves as design entry when generating dedicated hardware accelerators, or might be used as high-level programming language for tightly coupled multi-processor architectures. The language covers a broad range of applications from the areas of digital image, video and other signal processing, linear algebra, cryptography, and many other scientific computing domains where efficient parallelization techniques and hardware accelerators are indispensable. Key features of the PAULA language are: Its functional programming paradigm that makes parallelism explicit, an intuitive and compact syntax for describing iteration spaces, the deployment of big operators, and the advantage to describe also architectural parts of a target architecture. The usage of PAULA as design entry for hardware synthesis in our PARO design tool is demonstrated for several selected algorithms.

## 1 Introduction and Related Work

The desire for more mobility and the enthusiasm for ubiquitous electronic gadgets on the one side and the steady advances in semiconductor industry on the other hand are driving forces in the market of embedded digital systems. The growing number of stream-based applications is eager for computational power. Examples for such systems include handhelds for digital audio and video broadcasting, next generation game and entertainment consoles with high-definition television (HDTV) support and high-capacity storage media like HD DVD or Blu-ray Disc, or sophisticated applications in medical image processing and radar technology with real-time requirements. The rising complexity of the before mentioned applications typically implemented as hardware and software systems, and the growing importance of time-to-market, require powerful modeling methods and tools to automate the design and implementation process. While software compilers are used in every day life by engineers, there still exist only a few fully functional tools for the synthesis of hardware implementations from high-level algorithm descriptions. Examples of such systems are Catapult-C from Mentor Graphics [17], Forte Synthesizer [4], Agility Compiler from Celoxica [1], or PICO Express from Synfora [19, 20]. All these design tools start from a subset of sequential C, C++, or SystemC code. However, starting with sequential languages has the disadvantage that their semantics force a lot of restrictions on the execution order of the program. Most of the parallelism contained in the original mathematical model of the algorithm is lost during the transformation to sequential code. For instance, a simple summation  $s = \sum_{i=0}^7 a[i]$  is in C often written as a for loop in the following manner:

```
int s = 0;    for (i=0; i<=7; i++) { s += a[i]; }
```

By this, a sequential order is already predefined and a later parallelization can become a crucial task. In Fig. 1, the difference between a sequential and a parallel implementation is shown. Granted that enough resources (adders) are available, the latency could be reduced from linear to logarithmic runtime. The mapping of such algorithms to massively parallel architectures requires data dependency analysis in order to make the inherent parallelism explicit. However, this process is very complex since in sequential languages variables that are once defined can be overwritten arbitrarily. Another disadvantage of C-based hardware design is that most design tools support only a limited subset of the language. Porting existing, highly optimized C code to such an environment is a time consuming task and often ends in completely rewriting the code from scratch. In order to avoid this, in modern software compilers like gcc [6] as of version 4 or LLVM [13] as intermediate representation a so-called *static single assignment (SSA)* form is used where each variable is written exactly once. SSA allows to apply manifold compiler optimizations and transformations in a very efficient way. But

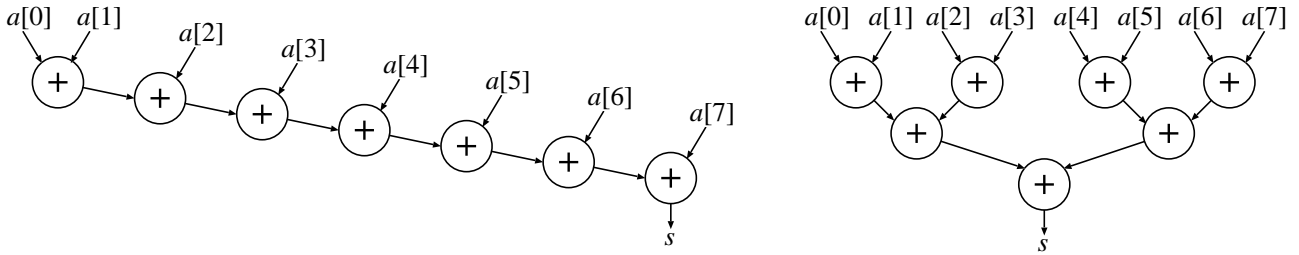


Fig. 1. Sequential and parallel implementation of a sum consisting of eight summands.

since the SSA form is used only in the intermediate representation (basic block level), these compilers cannot solve the data dependence analysis problem for multi-dimensional arrays.

Another option is to directly start from a functional language as for instance Sisal [5], Id [18], or Haskell [21]. Sisal allows recursion and finite streams. It was derived from VAL [16] a function-based language designed for data flow computers. Sisal and Id can be classified as shared memory parallel programming languages. Because of many similarities between Id and Haskell it was a foundation for pH, a parallel dialect of Haskell. However, also Haskell has only restricted abilities to handle true multi-dimensional arrays (i.e., arrays in which every dimension is treated as equivalent).

Apart from commercial systems, there exist several academic research projects. Many of them try to avoid the restrictions of sequential languages by using different programming and execution models. Examples of such systems are the Compaan/Laura suite [11, 23] which uses Kahn process networks as intermediate representation, or the SPARK environment [8] which is also a C-based design tool and thus suffers from the same problems as described above. Furthermore, SPARK can handle only one dimensional arrays. Another project is the MMAAlpha system [7, 22], based on loop parallelization in the polytope model [3, 15] similar to our approach. However, MMAAlpha does not contain any architectural description possibilities such as the modeling of multi-cycle operations or module selection. Further, it seems that the development of MMAAlpha is not continued anymore. The major contributions of this paper are:

- A *functional programming language* dedicated for mapping computational intensive algorithms onto parallel tightly coupled processor architectures with local memory is presented. That is, full SSA – also for *multi-dimensional arrays* – is provided.
- Compared to the aforementioned functional languages, PAULA has powerful expressions to specify polyhedral and lattice iteration domains.
- PAULA allows the convenient usage of *big operators* like  $\Sigma$ .
- The language can handle *run-time dependent control flow* to a certain degree.
- Finally, besides *behavioral description* possibilities also *architectural modeling* can be considered.

As stated before, the PAULA language can be divided into two major parts: A part for the behavioral description and a part for the architectural description. Thus, at first, in Section 2 the syntax and semantics of the behavioral part are described. In Section 3, the architectural part of the language is discussed. Afterwards, the application of PAULA in our PARO design system and some results are presented in Section 4. Finally, we summarize our contributions and discuss future work.

## 2 Behavioral Description

The class of algorithms that can be expressed by a PAULA program is based on the mathematical model of *dynamic piecewise linear/regular algorithms (DPLA/DPRA)* [9, 10]. Basically, a DPLA consists of a set of recurrence equations. For instance, when modeling signal processing algorithms, a designer naturally considers mathematical equations. Hence, programming in PAULA is very intuitive. A program is thus a system of quantified equations that implicitly defines a function of output variables in dependence of input variables. Some other semantical properties are particular to the PAULA language. *Single assignment property*: Any instance of an indexed variable appears at most once on the left hand side of an equation. *Computability*: There exists a partial ordering of the equations such that any instance of a variable appearing on the right side of an equation earlier appears

---

**Algorithm 1** Example PAULA program (FIR filter)

---

```
program FIR
{ /* Type alias definitions */
  typealias coeff_t signed fixed<12,11>;
  typealias input_t signed fixed<16,15>;
  typealias product_t signed fixed<28,26>;
  typealias output_t signed fixed<36,26>;
  /* Variable declarations */
  variable A 1 in coeff_t;
  variable U 1 in input_t;
  variable Y 1 out output_t;
  variable a 2 coeff_t;
  variable u 2 input_t;
  variable x 2 product_t;
  /* Parameter declarations */
  parameter N;
  parameter T;
  /* Program blocks */
  par (i >= 0 and i <= T-1)
  { /* Nested program blocks */
    par (j >= 0 and j <= N-1)
    { /* Equations */
      a[i,j] = A[j];
      u[i,j] = U[i-j]    if (i-j >= 0);
      u[i,j] = 0        if (i-j <= -1);
      x[i,j] = a[i,j] * u[i,j];
    }
    /* More equations */
    Y[i] = SUM[j >= 0 and j <= N-1](cast<output_t>(x[i,j]));
  }
}
```

---

in the left hand side in the partial ordering. *Execution Model*: The execution model of programs is architecture independent. A program may be executed as follows: (1) All instances of equations are ordered respecting the above defined partial ordering. (2) The indexed variables are determined by successive evaluation of equations.

In order to enable the synthesis of dedicated hardware accelerators or the mapping onto tightly coupled multi-processor architectures, the mathematical model of DPLAs have been extended by several constructs. Algorithm 1 shows an example program that describes a well known FIR filter. The individual language elements are described in the subsequent sections.

## 2.1 Programs

The key element of the PAULA language is the *program*. Every program starts with a program header that contains the program name. The header is followed by a declarative part that includes type alias definitions and declarations of variables and parameters in an arbitrary order<sup>1</sup>. After the declarations, there can be one or more so called *program blocks*, which are described in detail in Section 2.3.

## 2.2 Declarations

PAULA supports many different, parameterized data types, for example `signed integer<32>` which is a signed integer with a width of 32 bits. Several other integer, Boolean, fixed, and float data types exist. Moreover, the language could be easily extended by other data types. In order to keep a program simply readable and to reduce the effort when types of several related variables need to be changed, one may define aliases for existing data types. For example, the above mentioned FIR filter program uses type aliases for the data types of the filter coefficients, input, and output samples. This obviously

---

<sup>1</sup> In contrast to C, there are different name spaces for each kind of identifiers. So it is no problem to have a variable and a function that both have the name "foo". It is always clear from the context whether an identifier denotes an indexed variable, an iteration variable, a function or a data type. However, in order to improve readability, it is strongly recommended to use distinct identifiers for variables, iteration variables, and so on.

increases readability and also helps to avoid mistakes when changing data types. If for example the width of the output data should be changed, only the `typealias` statements have to be modified, and there is no danger of forgetting to update also the type cast in the last equation of the program.

Like in many other programming languages, the PAULA language requires variables to be declared before they can be used<sup>2</sup>. The syntax of such a declaration is the following:

```
variable name dimension [in/out] datatype;
```

Variables can either be declared as input variables using the `in` keyword, output variables when the `out` keyword is present, or normal (internal) variables when neither `in` nor `out` is given. The *name* of a variable is an arbitrary identifier. The *dimension* of a variable must be a positive integer, *datatype* can either be one of the built-in types or a type alias.

Further declarations are parameter and constant declarations. In a wide range of applications, it is desired to have parameterized iteration spaces. For example, consider the afore given FIR filter which has a certain number of taps  $N$ . This number appears several times in the program. A good way to describe this is to declare  $N$  as a parameter<sup>3</sup>. As most other languages, PAULA allows to define constants. These *named* constants can be used everywhere in an expression.

### 2.3 Program Blocks

A *program block* contains a set of equations (see Section 2.4) and/or recursively nested program blocks. Furthermore, each program block must somehow define an iteration space, that is, a set of iterations at which all equations and nested program blocks are defined.

There are two kinds of program blocks which differ only in the way their iteration space is defined: *par statements* and *for loops*. The *par* statement is the basic kind of program block. The syntax of the *par* statement is the following:

```
[label:] par (iteration space)
{ /* ... equations and nested program blocks ... */ }
```

The optional label is a name that might be used to identify and reference a program block. Labels are useful if program transformations (e.g., loop unrolling) should be applied only to a specific part of a program. If *par* statements are nested, the iteration variables of the parent program block(s) become parameters for the iteration space of the inner statement and can be used in the indexing functions of indexed variables in the inner program block. Example:

```
par (i >= 0 and i <= 10) {
  y[i] = ... /* Here, only i is visible */
  pb1: par (j >= i and j <= 2*i) {
    x[i,j] = ... /* Here, i and j are visible */
  }
}
```

If the iteration space of a program block is a simple, 1-dimensional range like  $1 \dots 10$ , *for loops* can be used as a more compact alternative to *par* statements. The syntax is the following:

```
[label:] for (iteration_variable = lower_bound to upper_bound [step stepsize])
{ /* ... equations and nested program blocks ... */ }
```

If the step size is omitted, it is assumed as 1. This loop is similar to the “for”-loop that can be found in many classic programming languages. Note, the “for”-loop is just used to generate a set of iteration points. It implies no execution order of the loop body, that is, if there are no restrictions by data dependencies, all iterations might be executed in parallel. For example, the *par* statement

```
par (i >= 0 and i <= 10) { ... }
```

is equivalent to the following *for* loop:

```
for (i = 0 to 10) { ... }
```

Similarly, the *par* statement

```
par (i=2*x: x >= 3 and x <= 8) { ... }
```

<sup>2</sup> In this section, only indexed variables are discussed. For iteration variables, different rules apply, which are described later.

<sup>3</sup> Parameters only apply to iteration spaces. If parameterized data types are needed, type aliases can be used.

which uses a lattice iteration space, could also be described in the following way:

```
for (i = 6 to 16 step 2) { ... }
```

The next example describes a 2-dimensional parameterized iteration space.

$$\mathcal{I} = \{i = x, j = 2 \cdot y + 1 \wedge -N \leq x \leq 2 \cdot N \wedge 0 \leq y \leq 5\}$$

where  $N$  is a parameter. The PAULA syntax for this iteration space is very straight forward:

```
par (i=x, j=2*y+1: x >= -N and x <= 2*N and y >= 0 and y <= 5) { ... }
```

The lattice definition declares the *iteration variables*  $i$  and  $j$ , and the so-called *internal iteration variables*  $x$  and  $y$ . The latter are used only to generate the iteration space but cannot be referenced anywhere else.

## 2.4 Equations and Indexed Variables

*Equations* are that kind of statements that do the actual calculations of a program. The basic syntax of an equation is:

```
[label:] indexed_variable = expression;
```

An example equation in a PAULA program could be:

```
x[i,j] = x[i-1,j] + y[2*i,j-1];
```

Similar as program blocks, equations may have a label which gives them a unique name and allows referencing. The data type of the expression on the right hand side of the equation must be equal to, or automatically convertible to the data type of the indexed variable on the left hand side.

In a DPLA, data is represented by so-called *indexed variables*. As stated earlier, these variables have to be declared at the beginning of a program, have a data type, and a certain dimension. Accessing the contents of an indexed variable is done by an indexing function. For instance, the indexing function of variable  $y$  of the above example can also be written as  $\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix}$ . The general form of an indexing function is given by  $QI + d$  with  $Q \in \mathbb{Z}^{k \times n}$  and  $d \in \mathbb{Z}^k$ , where  $k$  is the dimension of the indexed variable and  $I \in \mathcal{I}$  is an iteration point inside the iteration space  $\mathcal{I} \subset \mathbb{Z}^n$ .

**Equations with Iteration Dependent Conditional** To allow irregularities in a program, an equation may have an iteration dependent conditional. That is, a given equation is valid only for a subset of iterations inside the current program block. The syntax is

```
indexed_variable = expression if (iteration_space);
```

Iteration dependent conditionals can only use iteration variables that are defined by iteration spaces of enclosing program blocks. Example (matrix multiplication,  $C = A \times B$ ):

```
par (i >= 1 and i <= N and j >= 1 and j <= N and k >= 1 and k <= N) {
  /* Equations with iteration dependent conditional... */
  c[i,j,k] = 0                               if (k == 1);
  c[i,j,k] = a[i,j,k] * b[i,j,k] + c[i,j,k-1] if (k > 1);
  C[i,j]   = c[i,j,k]                         if (k == N); /* Output */
  /* More equations (without iteration dependent conditional)... */
  a[i,j,k] = A[i,k]; /* Input matrix A */
  b[i,j,k] = B[k,j]; /* Input matrix B */
}
```

Again, note that the order of equations does not matter and thus the equations reading the input matrices can appear after the computation and output equations of the program.

**Equations with Run-time Dependent Conditional** Besides iteration dependent conditionals, a PAULA equation can also have a run-time dependent conditional. The syntax is

```
indexed_variable = ifrt(cond_expression, then_expression, else_expression);
```

| (a)        |                            |                           |                               | (b)      |          |
|------------|----------------------------|---------------------------|-------------------------------|----------|----------|
| Precedence | Operator                   | Description               | Associativity                 | Operator | Function |
| 1          | ()                         | grouping operator         | —                             | +        | add      |
|            | $\mathcal{A}$              | constant                  |                               | -        | sub      |
|            | $a []$                     | indexed variable access   |                               | *        | mul      |
|            | $f ()$                     | function call             |                               | /        | div      |
|            | $SUM [] ()$<br>$cast<> ()$ | big operator<br>type cast |                               | %        | mod      |
| 2          | +                          | unary plus                | right to left                 | ==       | eq       |
|            | -                          | unary minus               |                               | !=       | neq      |
|            | !                          | logical negation          |                               | >        | gt       |
|            | ~                          | bitwise complement        |                               | <        | lt       |
| 3          | *                          | multiplication            | left to right                 | >=       | geq      |
|            | /                          | division                  |                               | <=       | leq      |
|            | %                          | modulus                   |                               | &        | band     |
| 4          | +                          | addition                  | left to right                 |          | bor      |
|            | -                          | subtraction               |                               | ^        | bxor     |
| 5          | <<                         | bitwise shift operators   | left to right                 | ~        | bnot     |
|            | >>                         |                           |                               | <<       | shl      |
| 6          | ==, !=, <                  | relational operators      | only binary relations allowed | >>       | shr      |
|            | >, <=, >=                  |                           |                               | &&       | land     |
| 7          | &                          | bitwise and               | left to right                 |          | lor      |
| 8          | ^                          | bitwise xor               | left to right                 | !        | lnot     |
| 9          |                            | bitwise or                | left to right                 |          |          |
| 10         | &&                         | logical and               | left to right                 |          |          |
| 11         |                            | logical or                | left to right                 |          |          |

**Table 1.** In (a), the operators in PAULA and their precedence is shown. The mapping of operators to functions is given in (b).

where *cond\_expression* is the actual conditional, an expression which must be of data type Boolean. Furthermore, the expressions *then\_expression* and *else\_expression* must each be of the same type as the indexed variable on the left hand side of the equation or must be convertible to that type. The value of the right hand side of the equation, that is, the value “returned” by the `ifrt` statement, is the value of *then\_expression* if *cond\_expression* evaluates to “true”, or the value of *else\_expression* otherwise. In order to satisfy the single assignment property both branches must be defined. An example equation with a run-time dependent conditional that catches a division by zero could be:

```
x[i,j] = ifrt(b[i,j] != 0, a[i,j]/b[i,j], 65535);
```

The next example might be part of an image processing algorithm. It ensures that the value of a pixel does not exceed a certain limit. Of course, if the condition is somewhat more complex, it can be separated in an own equation.

```
c[i,j] = a[i,j] > 255;
v[i,j] = ifrt(c[i,j], a[i,j], 255);
```

## 2.5 Expressions

The syntax and semantics of expressions in the PAULA language are not much different than those found in C and similar languages. Table 1(a) lists all available operators and their precedence. During the design flow, operators are replaced by functions. This is done because binding possibilities (see Section 3.3) are only defined for functions, not operators. Table 1(b) lists the mapping of operators to functions. Of course, the user may define (i.e., give binding possibilities for) arbitrary additional functions. Please note that the semantics of only the operators are defined by the language. The design system may transform expressions that contain operators by applying common mathematical rules. For instance,  $a*b+a*c$  can be transformed to  $a*(b+c)$  and thus saving one multiplication. Such transformations may not be performed on functions.

For the sake of brevity, we describe only the operators *function* and *reduction* detailed in the following because their semantics differ or do not exist in conventional languages such as C.

**Functions** In the PAULA language, functions are real mathematical functions, that means their

return value solely depends on the arguments. Functions cannot have a state and there is also no global state. For every function, a binding possibility has to be defined (see Section 3.3) which provides a hardware implementation and/or a simulation model, and thus brings the semantics of the function. Examples of functions are  $\text{add}(x[i], x[i-1])$  or  $\text{sin}(y[i, j])$ .

**Big Operators** Big operators implement mathematical operators such as  $\sum$  or  $\prod$ . Often these operators are also called reductions. The syntax is the following

`operator [ iteration_space ] ( expression )`

where *operator* is one of SUM, PRODUCT, MIN, MAX. For instance,  $\sum_{j=0}^{10} b[i, j]$  can be written in PAULA as

`SUM[j >= 0 and j <= 10] (b[i, j])`

However, in contrast to the common mathematical notation the iteration space is not required to be 1-dimensional. For example, one can write

`SUM[j >= 0 and j <= 10 and k >= 0 and k <= j] (c[i, j, k])`

to describe the following sum:  $\sum_{(j,k) \in \mathcal{I}} c[i, j, k]$  with  $\mathcal{I} = \{(j, k) : 0 \leq j \leq 10 \wedge 0 \leq k \leq j\}$ .

This is equivalent to the double sum:  $\sum_{j=0}^{10} \sum_{k=0}^j c[i, j, k]$

Similarly to the nesting concept of program blocks, iteration variables from the program block that contains the expression are available as parameters in the iteration space of the big operator:

```
par (i >= 0 and i <= 15) {
  a[i] = SUM[j >= 0 and j <= i] (b[i, j]);
}
```

The usage of big operators allows a compact and intuitive description style as the following image processing code fragment of an approximated 2-D Gauss window filter demonstrates

```
w[0,0] = 1; w[0,1] = 2; w[0,2] = 1;
w[1,0] = 2; w[1,1] = 4; w[1,2] = 2;
w[2,0] = 1; w[2,1] = 2; w[2,2] = 1;
h[x,y] = SUM[i>=0 and i<=2 and j>=0 and j<=2] (pic_in[x+i,y+j] * w[i,j]);
pic_out[x,y] = h[x,y] >> 4; // divided by 16
```

### 3 Architectural Description

The main purposes of the PARO design system are to generate hardware descriptions (e.g., VHDL) for parallel algorithms or to generate code for tightly coupled processor arrays. The design flow involves, among others, the steps allocation, binding, scheduling, and synthesis and code generation, respectively. Obviously, during hardware or code generation, detailed knowledge about the target hardware architecture is required. The definition of this architecture model can be also done in the PAULA language. This section describes the syntax and semantics of such architectural data definitions. An architecture model can be subdivided into three parts: (1) *resource type definitions*, (2) *resource allocation*, (3) *binding possibility definitions*. Since the architectural definitions are completely independent of the algorithm, it is recommended to put them into separate source files and include them in the main file, for example:

```
include("alu.arch.paro")
include("multiplier.arch.paro")

program example { ... }
```

Two files are included in this example which contain architectural data for an ALU and a dedicated multiplier.

### 3.1 Resource Type Definitions

A *resource type definition* gives information about which resource types are available in an architecture model. The following algorithm shows an example definition for a resource type that is called 'alu16'.

```
resourcetype alu16
{ ops 16;
  input a integer<16>;
  input b integer<16>;
  output c integer<16>;
  output d boolean;
  component alu;
  parameter WORDSIZE = 16;
}
```

This resource type offers up to 16 different operations ('ops 16'). For resource types that support only one operation, the 'ops' statement may be omitted. Next, the data I/O ports of the corresponding hardware component are defined. Our example ALU has two inputs 'a' and 'b', and one output 'c', all of type 16 bit integer, and an additional Boolean output 'd'. Depending on which operation the ALU currently performs, the result of the computation is available on either port 'c' (for operations that have a result of 16 bit integer, e.g., addition, subtraction, shift, and so forth), or on output 'd' (for operations that have a Boolean result, e.g., comparisons). Afterwards, the resource type is mapped to a hardware component 'alu', which may be a VHDL entity name. This mapping together with the following 'parameter' statement allows to use an existing, generic HDL implementation (e.g., a generic VHDL entity) as basis for many different resource types. It should be mentioned only that there exist further statements as for instance one to annotate different cost metrics to a resource type in order to allow early cost analysis and to enable design space exploration.

### 3.2 Resource Allocation

The *resource allocation* selects how many instances of a given resource type shall be available within one processor element<sup>4</sup>. The syntax is the following:

```
allocation resourcetype_name integer;
```

For example, "allocation alu16 3;" would allocate up to three instances of the resource type 'alu16'. It is also possible to allocate an infinite number of instances per resource type, using the keyword 'infinite' instead of a number.

### 3.3 Definition of Binding Possibilities

By defining *binding possibilities*, information about which operations are supported by each resource type is added to the architecture model. In PAULA, only functions can be mapped to hardware. Operators (like '+' or '-') are replaced by functions as a first step of hardware synthesis. The syntax of a binding possibility definition is the following:

```
bindingpossibility function name (types_of_params) ret_type on resource { body }
```

Where *types\_of\_params* is a comma-separated list of data types which implicitly defines the number of parameters of the function. For example:

```
function mul(integer<32>,integer<32>) integer<64>
```

This declares a function called "mul", which gets two 32-bit integers and returns a 64-bit integer. One exemplarily binding possibility for resource type 'alu16' (cf. Section 3.1) may look as follows.

```
bindingpossibility function mul(integer<16>, integer<16>) integer<16> on alu16
{ op 0;
  input a,b;
  output c;
  cycles 2;
  pipelinerate 1;
  simulatorplugin "sim_alu16.so", "mul_int16";
}
```

<sup>4</sup> The number of processors is not specified in the architectural part, it is defined during the tiling and mapping phases of the synthesis.

The body of a binding possibility consists of three parts, the operation and port mapping, and the scheduling parameters. The *operation mapping* ('op' statement) defines which operation number (opcode) must be selected by the generated hardware control unit so that the resource performs the desired operation. The *port mapping* ('input' and 'output' statements) assigns function parameters and the result to input/output ports of the hardware component that implements the resource type to which the function is bound. The port names that appear here must be declared in the respective resource type definition, and their data types must match. The order of the input ports corresponds to the order of the function parameters. The *scheduling parameters* ('cycles' and 'pipelinerate' statements) define the execution time and the pipeline rate of the operation on the given resource type. The statement `simulatorplugin` defines where the simulator can get a simulation model for the considered function. The two arguments are the plugin file (e.g., a dynamically loadable C library) and the name of the (C-)function in the plugin.

## 4 Usage of the PAULA Language

PAULA serves as entry language for our PARO high-level synthesis tool. Based on a given algorithm, various source-to-source compiler transformations and optimizations can be applied within the design system. Among others, these transformations include: Constant and variable propagations, common sub-expression elimination, loop perfectization, dead-code elimination, affine transformations of the iteration space, strength reduction of operators (usage of shift and add instead of multiply or divide), and loop unrolling. Besides these transformations and optimizations, key features of PARO are sophisticated partitioning and resource constrained scheduling techniques in order to balance performance with cost in terms of area, I/O bandwidths, and several levels of memory. In addition, by generation of a distributed control flow, costly *housekeeping*<sup>5</sup> is reduced [2]. Finally, our design system generates highly parallel and deeply pipelined RTL models which are converted subsequently into synthesizable VHDL code.

As a case study, we have synthesized several algorithms from various application domains, some of which are taken from the well-known MediaBench suite [14]. We profiled the JPEG and MPEG2 algorithms and identified some of the most computational intensive loop kernels. The results and the results of some other algorithms are shown in Table 2. All algorithms were implemented using 16-bit integer or fixed point arithmetics and synthesized using the Xilinx ISE 6.3i toolchain targeting a Virtex-II 8000 FPGA. For each example, the table shows the cost in terms of FPGA primitives, the maximum clock frequency, the total execution time for the algorithm in clock cycles, and the average number of clock cycles between the availability of two successive output instances (e.g., samples, pixels). The latter is the inverse of the throughput, that is, a smaller number means a higher throughput. An initial latency does not affect the throughput. Note that an output interval less than 1 means that more than one output instance are available per clock cycle. A closer look at two results of Table 2 impressively emphasize the efficiency of our approach. For instance, Gaussian filtering can be applied with 42 frames per second for a  $2000 \times 2000$  image, this is twice the size of actual full HDTV resolution. As other example, the 64-tap FIR filter achieves an throughput of almost 320 MByte/s.

## 5 Summary and Outlook

We presented a functional programming language for the design of multi-dimensional dataflow-intensive applications. In comparison to sequential languages such as C, C++, or SystemC, the main strength of PAULA is that it imposes no restrictions on the execution order of the program. Furthermore, the PAULA language and PARO design system enable a compact description and the efficient generation of highly parallel hardware accelerators.

Currently, we are working on the automated target code generation for programmable tightly coupled multi-processor architectures [12]. In the future, we would like to study communicating loops and to analyze the performance when integrating such accelerators into a system-on-a-chip.

---

<sup>5</sup> Housekeeping denotes the cost within the control flow such as loop counters, address generators, or comparators.

**Table 2.** Experimental results.

| Algorithm                              | No. of LUTs | No. of FFs | No. of MULTs | No. of BRAMs | max. Clock (MHz) | Exec. Time (cycles) | avg. Output Interval (cycles) |
|--|-------------|------------|--------------|--------------|------------------|---------------------|-------------------------------|
| Edge Detection, 1000×1000 image        | 2997        | 1913       | -            | 44           | 120              | $7.5 \cdot 10^5$    | 3                             |
| Gaussian Filtering                     |             |            |              |              |                  |                     |                               |
| – 1000×1000 image, 3x3 mask            | 683         | 1463       | 9            | 2            | 171              | $1.0 \cdot 10^6$    | 1                             |
| – 2000×2000 image, 3x3 mask            | 696         | 1472       | 9            | 4            | 169              | $4.0 \cdot 10^6$    | 1                             |
| – 2000×2000 image, 5x5 mask            | 1555        | 3922       | 25           | 8            | 169              | $4.0 \cdot 10^6$    | 1                             |
| FIR Filter, 64 Taps                    | 5782        | 9089       | 64           | -            | 167              | 68                  | 1                             |
| Matrix Multiplication, 6×6 matrix size | 1888        | 4067       | 36           | -            | 166              | 20                  | 0.28                          |
| Discrete Cosine Transformation         | 1754        | 1152       | 8            | 1            | 130              | 94                  | 0.65                          |
| Elliptical Wave Digital Filter         | 1169        | 624        | 1            | -            | 94               | $2.5 \cdot 10^5$    | 1                             |
| MPEG2 Quantisizer                      | 637         | 1190       | 1            | -            | 141              | 222                 | 2.95                          |
| JPEG Loop 1                            | 82          | 79         | -            | -            | 224              | 63                  | 1                             |
| JPEG Loop 2                            | 570         | 1139       | -            | -            | 158              | 126                 | 1                             |

## Acknowledgment

This work has been supported in part by the German Science Foundation (DFG) in project under contract TE 163/13-1 and TE 163/13-2. Also, we would like to thank the anonymous reviewers for their effort and for their valuable comments.

## References

- CELOXICA, Agility. <http://www.celoxica.com>.
- H. Dutta, F. Hannig, H. Ruckdeschel, and J. Teich. Efficient Control Generation for Mapping Nested Loop Programs onto Processor Arrays. *Journal of Systems Architecture*, 53(5–6):300–309, May 2007.
- P. Feautrier. Automatic Parallelization in the Polytope Model. Technical Report 8, Laboratoire PRiSM, Université des Versailles St-Quentin en Yvelines, 45, avenue des États-Unis, F-78035 Versailles Cedex, June 1996.
- Forte Design Systems. <http://www.forteds.com>.
- J. Gaudiot, T. DeBoni, J. Feo, W. Böhm, W. Najjar, and P. Miller. The Sisal Project: Real World Functional Programming. In S. Pande and D. Agrawal, editors, *Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques, and Run Time Systems*, volume 1808 of *Lecture Notes in Computer Science (LNCS)*, pages 45–72. Springer, 2001.
- GCC, the GNU Compiler Collection. <http://gcc.gnu.org>.
- A. Guillou, P. Quinton, and T. Risset. Hardware Synthesis for Multi-Dimensional Time. In *Proc. IEEE 14th Int. Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, pages 40–50, The Hague, The Netherlands, June 2003.
- S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: A High-Level Synthesis Framework for Applying Parallelizing Compiler Transformations. In *Proc. of the 16th International Conference on VLSI Design*, pages 461–466, Jan. 2003.
- F. Hannig and J. Teich. Dynamic Piecewise Linear/Regular Algorithms. In *Proc. of the Fourth International Conference on Parallel Computing in Electrical Engineering (PARELEC)*, Dresden, Germany, Sept. 2004.
- F. Hannig and J. Teich. Resource Constrained and Speculative Scheduling of an Algorithm Class with Run-Time Dependent Conditionals. In *Proc. IEEE 15th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Galveston, TX, USA, Sept. 2004.
- B. Kienhuis, E. Rijpkema, and E. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proc. of the 8th Int. Workshop on Hardware/Software Co-Design*, pages 13–17, San Diego, CA, USA, 2000.
- D. Kissler, F. Hannig, A. Kupriyanov, and J. Teich. A Highly Parameterizable Parallel Processor Array Architecture. In *Proc. of the IEEE International Conference on Field Programmable Technology (FPT)*, pages 105–112, Bangkok, Thailand, Dec. 2006.
- C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, Palo Alto, CA, USA, Mar. 2004.
- C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.
- C. Lengauer. Loop Parallelization in the Polytope Model. In E. Best, editor, *CONCUR'93, Lecture Notes in Computer Science 715*, pages 398–416, 1993.
- J. McGraw. The VAL Language: Description and Analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(1):44–82, 1982.
- Mentor Graphics Corp. <http://www.mentor.com>.
- R. Nikhil. An Overview of the Parallel Language Id (a foundation for pH, a parallel dialect of Haskell). Technical report, Digital Equipment Corp., Cambridge Research Laboratory, MA 02139, USA, Sept. 1993.
- R. Schreiber, S. Aditya, B. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-Level Synthesis of Nonprogrammable Hardware Accelerators. Technical Report HPL-2000-31, Hewlett-Packard Laboratories, Palo Alto, CA, USA, May 2000.
- Synfora, Inc. <http://www.synfora.com>.
- S. Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, 1999.
- D. Wilde and O. Sié. Regular Array Synthesis using ALPHA. In *Proc. of the International Conference on Application Specific Array Processors (ASAP)*, pages 200–211, San Francisco, CA, USA, Aug. 1994.
- C. Zissulescu, B. Kienhuis, and E. Deprettere. Expression Synthesis in Process Networks generated by LAURA. In *Proc. IEEE 16th International Conference on Application-specific Systems, Architectures, and Processors (ASAP)*, pages 15–21, Island of Samos, Greece, July 2005.