Int J Parallel Prog (2012) 40:262–289
DOI 10.1007/s10766-011-0187-0

LALP: A Language to Program Custom FPGA-Based Acceleration Engines

Ricardo Menotti · João M. P. Cardoso · Marcio M. Fernandes · Eduardo Marques

Received: 2 February 2011 / Accepted: 20 August 2011 / Published online: 4 October 2011

Abstract Field-Programmable Gate Arrays (FPGAs) are becoming increasingly important in embedded and high-performance computing systems. They allow performance levels close to the ones obtained with Application-Specific Integrated Circuits, while still keeping design and implementation flexibility. However, to efficiently program FPGAs, one needs the expertise of hardware developers in order to master hardware description languages (HDLs) such as VHDL or Verilog. Attempts to furnish a high-level compilation flow (e.g., from C programs) still have to address open issues before broader efficient results can be obtained. Bearing in mind an FPGA available resources, it has been developed *LALP (Language for Aggressive Loop Pipelining)*, a novel language to program FPGA-based accelerators, and its compilation framework, including mapping capabilities. The main ideas behind LALP are to provide a higher abstraction level than HDLs, to exploit the intrinsic parallelism of hardware resources, and to allow the programmer to control execution stages whenever the compiler techniques are unable to generate efficient implementations. Those features are particularly useful to implement *loop pipelining*, a well regarded technique used to

R. Menotti (✉) · M. M. Fernandes
Departamento de Computação, Universidade Federal de São Carlos, São Carlos, Brazil
e-mail: ricardomenotti@acm.org

M. M. Fernandes
e-mail: marcio@dc.ufscar.br

J. M. P. Cardoso
Departamento de Engenharia Informática, Faculdade de Engenharia,
Universidade do Porto, Porto, Portugal
e-mail: jmpc@acm.org

E. Marques
Inst. Ciências Matemáticas e Computação, Universidade de São Paulo, São Carlos, Brazil
e-mail: emarques@icmc.usp.br

accelerate computations in several application domains. This paper describes LALP, and shows how it can be used to achieve high-performance computing solutions.

Keywords Loop pipelining · Compilers · Reconfigurable computing · FPGA

1 Introduction

In order to attend the growing demand for high performance of certain embedded systems, programmable devices such as FPGAs can be used as alternative solutions or extensions to von-Neumann based processors. FPGA-based systems are able to achieve performance levels close to ASICs, with the advantages of flexibility and programmability. However, the typical development process employed for reconfigurable devices is similar to the one used for ASICs, demanding hardware background.

Most computing problems of practical use have been implemented in software to be executed in General Purpose Processors (GPPs). In an attempt to use this large quantity of algorithms, researchers have developed high-level synthesis techniques and tools [1]. The goal of high-level synthesis is the generation of specialized hardware architectures from an algorithm described in a high level language, such as C, Java, or some variant of them. However, the generation of architectures from high-level descriptions rarely results in optimal systems, since imperative languages do not explicitly expose parallelism among operations. Other characteristics of such languages impose difficult barriers for synthesis tools, as described in many papers such as [2]. In addition, modern FPGAs have an increasing set of sophisticated resources that can be used to substitute the basic operations performed by GPPs. Digital Signal Processing (DSP) and reconfigurable memory blocks are examples of such resources.

Many efforts trying to achieve a direct mapping of algorithms into hardware concentrate on *loops* since they often represent the most computationally intensive regions of application codes. A particularly useful technique for this purpose is *loop pipelining*, which is usually adapted from software pipelining techniques [3]. The application of this technique is strongly related to instruction scheduling, which often prevents an optimized use of the resources present in modern FPGAs. Most of the high-level compilation tools, such as Garp C [4], MATCH [5] and the Snider's compiler [6], deal with loops by using software pipelining techniques based on Rau's *Iterative Modulo Scheduling* [7]. The technique rearranges the order of instructions taking into account machine resources and dependence constraints. The resulting schedule of instructions contains a prologue, a kernel, and an epilogue. Another approach, also based on software pipelining, is *kernel identification*, a technique relying on loop unrolling to find patterns upon which a kernel is built, and then executed in fewer cycles [8]. In addition, *pipelining vectorization* techniques [9] have also been used for this purpose, but they are usually restricted to regular loops.

State of the art research frameworks for high-level synthesis (*e.g.*, SPARK [10]) and tools used to build hardware acceleration modules (*e.g.*, C2H from Altera [11]) reveal that the obtained results concerning loops are not optimal, and that there is still room for improvements. Thus, in an attempt to alleviate the problem (*i.e.*, to achieve loop implementations with higher performance), this paper proposes a novel domain

specific language called LALP (Language for Aggressive Loop Pipelining), and its corresponding compilation framework. The first version of LALP and its compilation flow have been previously introduced in [12,13]. The language allows programmers to describe sequential code and loop computations using C-like syntax. The compilation framework is able to generate optimized hardware structures for the execution of loop computations, targeting a platform based on reconfigurable hardware.

It can be argued that this approach bridges the gap between lower and higher abstraction levels. However, it presents the advantages of a higher abstraction level than existing HDLs (e.g., VHDL, Verilog), and still allowing the programmer to control scheduling at the clock cycle level (but without the low level notion of clock signals). In addition, this approach allows textual description of important features (generated by front-end analysis tools), enabling the exploration of deep pipelining levels, with different loop control mechanisms (e.g., a single centralized counter per loop, or distributed counters and control).

The remainder of this paper is organized as follows. The next section presents the basic ideas of a scheme called *Aggressive Loop Pipelining*, the underlying technique targeted by LALP. This domain specific language was specially developed to support it, and is described in Sect. 3. A framework developed to map LALP semantics into FPGA structures is described in Sect. 4, followed by some experimental results in Sect. 5. The paper is concluded with a discussion about related work in Sect. 6, and final remarks in the last section.

2 Aggressive Loop Pipelining

In recent years new techniques for loop pipelining have been used to avoid the limitations imposed by the basic operation primitives [14,15]. Those data-driven approaches suggest that unconventional loop pipelining schemes may better exploit the available resources in reconfigurable architectures, and so increase their performance. The basis of the approach described in this paper, called *Aggressive Loop Pipeline (ALP)*, adapts some of the ideas proposed in [14] for execution in FPGA platforms. However, instead of a data-driven scheme, the ALP technique attempts to achieve the maximum throughput using counters to furnish the iteration space in loops, and shift-registers to synchronize operations [16]. By doing so, it avoids the need of a centralized control based on finite state machines, a *key difference* from traditional approaches to loop pipelining. By using the ALP scheme, operations in the loop body are executed at clock cycles according to the paths taken during execution, and optimal loop pipelining can be achieved. The major reason for this is the fact that traditional loop pipelining techniques statically assign operations to each stage of the FSM (finite state machine) that controls the loop. In order to do so, they either need to consider the critical path latency of the loop body (too conservative), or to take into account all possible paths that can be taken (too complex).

Consider the example in Fig. 1, which shows a loop body (a), and the corresponding diagram for the ALP execution (b). As depicted, the execution datapath has a counter component (CNT) mimicking the *for* statement of the source code. i.e., updating the i value, and testing for the stop condition. The parameter $step$ has two functions.

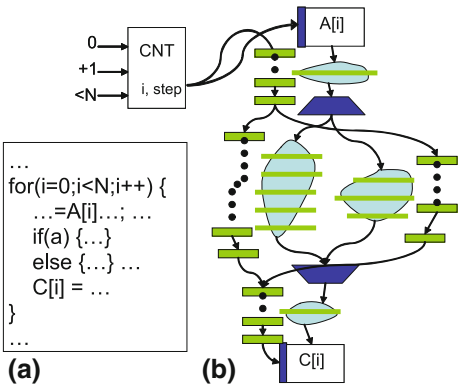

Fig. 1 Example of aggressive loop pipelining: **a** segment of code; **b** hardware structure

The straightforward one is to define the number of cycles between consecutive updates of the parameter i. The more critical one is to define the pipeline stage in which an instruction is to be executed, in some sense similar to the use of *predicate* registers, employed by some software pipelining techniques. By adding pipeline stages to the datapath structures executing the loop body it is possible to pipeline even loops with conditional constructs. In principle, cascade registers are used to propagate both the i and $step$ values. A possible optimization of the scheme would be the duplication of the counter component (enabled by the original $step$ signal), avoiding the cascading of the i value to deep levels of the execution datapath. It should be noticed, however, that in any case register cascading could be economically implemented by using the native support to shift-registers present in most FPGAs.

3 LALP Domain Specific Language

In order to facilitate the application of the ALP technique, we defined LALP, a domain specific language. The main idea of LALP is to allow the user to program highly efficient hardware engines without the required details of HDLs such as Verilog or VHDL. Being LALP a domain-specific language, it does not need the overall flexibility ensured by typical HDLs. On the other hand, LALP exposes to the user some of the intrinsic advantages of using contemporary FPGAs, such as concurrent execution, a large number of hardware units, and configurable memories. Statements in LALP are intrinsically concurrent, with sequencing being ensured by control and data dependences, or by using explicit synchronization points, if necessary.

The language allows the use of directives (in the form of specific constructs) to guide synchronization and the pipeline generation. The LALP compiler uses these directives to determine, whenever possible, the number of clock cycles and other parameters that will ensure the correctness of the code being executed. When directives are omitted, the compiler tries to infer those parameters. Note, however, that whenever the LALP compiler is unable to determine the clock cycles that would ensure correctness, it warns the user about that.

The language has a C-like syntax, which was chosen due to its familiar grammar and widespread use. The current implementation of LALP supports basic constructs such as declaration of types, scalar and array variables, arithmetic expressions, logic expressions, *for* loops, and *sync* qualifiers. Support for function calls are still absent, but this is not a significant issue to evaluate the potential of the techniques being developed.

3.1 Basic Features

The main component of LALP is a *module*. Each module will result in a hardware engine to be implemented in FPGAs. LALP modules have parameters that can be of types *in* (input) or *out* (output). These parameters specify the interfaces to the environment or system where the hardware engine resultant from the module will be integrated. The body of a module specifies the required computations. Data types, constants and declarations can also be expressed before a module. An example of a module with two outputs (*sum* and *done*) and one input (*init*) in shown bellow:

```
1 // constants and data types can be expressed here
2 dotprod_alp(out int sum, out bit done, in bit init) {
3 // declarations can be expressed here
4 // computations are expressed here
5 }
```

LALP currently allows the programmer to define scalar and uni-dimensional array variables. All data types are specializations of the *fixed* (fixed-point) data type. Integers can be defined with a variable number of bits, signed or unsigned. Although real numbers can be defined (represented as fixed-point types), the current version of the compiler does not as yet using floating-point data types. Future extensions should address this issue, and also include support for floating point data types. The example below illustrates the declaration of *int*, *uint*, *uint8*, and *bit* data types. The *typedef* keyword is used for this purpose, although a straightforward declaration using just *fixed(...)* could also be employed.

```
1 typedef fixed(32, 1) int;
2 typedef fixed(32, 0) uint;
3 typedef fixed(8, 0) uint8;
4 typedef fixed(1, 0) bit;
5 ...
6 int a; // similar to: fixed(32, 1) a;
```

In LALP, constants can be declared using the keyword *const* as in *const N = 32*. When declaring scalar and array variables one can also initialize them as illustrated in the example below. In this example, f is an array of N *int* values, with the first two elements being equal to 0 and 1, respectively.

```
1 int f[N] = {0, 1};
2 int a = 0;
```

LALP does not support *if-then* and *if-then-else* constructs. There is, however, support for assignments whose input comes from one of two sources, as illustrated in the example below.

```
1 max = a < b ? b : a;
```

LALP allows the use of predicated-based constructors in order to specify conditional execution of operations, or the dependences on a triggering condition. The code segment below shows an example of a predicate triggering one of two possible assignments. This example produces a hardware implementation equivalent to the one obtained in the previous example.

```
1 p1 = a < b;
2 max = a when !p1;
3 max = b when p1;
```

Another example showing the use of predicates based on triggering conditions is shown below. In this case, the assignment to the *diff* variable is done 3 clock cycles after the generation of the *len.step* event.

```
1 diff = val − valpred when len.step@3;
```

3.2 Implementing Loops in LALP

The key features of LALP appear in the support for loop execution, in the form of *for* loops, and *sync* qualifiers. Loops are implemented using a *counter* statement, which includes attributes that can be used in other LALP statements. Those attributes correspond to ports of the correspondent counter hardware component, and are presented in Table 1.

Sync qualifiers are expressed as @n, being n the number of clock cycles needed to trigger an action, *e.g.*, to perform an operation. As an example to show the use of counters and *sync* qualifiers, consider the following C code statements extracted from an ADPCM decoder implementation:

```
1 for (len=0; len<DATASIZE; len++) {
2   ........
3   diff = val − valpred;
4   if (diff < 0)
5     sign = 8;
6   else
7     sign = 0;
8   ........
```

The code below shows a possible LALP implementation for the C code just presented.

Table 1 Attributes of the counter component

	Name	Function
Generics	`bits`	number of bits of `input`, `output` and `termination`
	`steps`	number of clock cycles per `output` update (default: 1)
	`increment`	increment/decrement value (default: 1)
	`down`	counting direction (default: 0, increment)
	`condition`	stop condition (default: 1, <=)
Ports	`input`	used to load the initial value
	`termination`	end value of counting
	`clk`	clock signal
	`clk_en`	enable execution
	`reset`	re-initiate counting
	`load`	load initial value
	`step`	used to synchronize operations
	`done`	signalizes the end of counting
	`output`	current value of counting

```
1  counter (len=0; len<DATASIZE; len++@10);
2   ........
3   diff = val - valpred when len.step@3;
4   sign = diff < 0 ? 8 : 0 when len.step@4;
5   ........
```

In the first statement (line 1), *counter* implements a *for* loop that will update the value of *len* every ten cycles. The second statement (line 3) specifies a subtraction, with the result being registered in the *third* cycle of every loop iteration. Similarly, the conditional assignment in the last statement (line 4) occurs in the *fourth* cycle of every loop iteration, exemplifying the handling of data dependences by means of the *sync* qualifier: *diff* is defined in *step@3*, and used in the next clock cycle (@ values taken from a broader context).

Currently, *while* and *do-while* loops do not have specific LALP constructs. Future studies will address possible extensions. However, for some of these loop types a counter-based scheme could also be also used.

When implementing an algorithm in LALP, it is assumed in principle that all instructions in the loop body can be executed in parallel, i.e., one new iteration starts and completes in one clock cycle (the default behavior of the *counter* statement). Thus, the programmer is responsible for determining intra- and inter-dependences, i.e., dependences between instructions from the same and from distinct iterations, respectively. In both cases, the *sync* qualifier is used to ensure the correct order of values being produced and consumed. Note, however, that the mapping tool is able to determine the location and values of some of the *sync* qualifiers. Although this task is currently as much as possible automated, it can be said that in some cases it is still useful to allow programmers to make changes by means of *sync* qualifiers. As an aid to the process

```
1   i1 = 0;
2   for (i = 0; i < num_fdcts; i++) {
3     for (j = 0; j < N; j++) {
4       f0 = dct_io_ptr[ 0+i1];
5       f1 = dct_io_ptr[ 8+i1];
6       ...
7       i1++;
8     }
9     i1 += 56;
10  }
11  i1 = 0;
12  for (k = 0; k < N*num_fdcts; k++) {
13    ...
14  }
```

Fig. 2 C code example with nested loops

of assigning *sync* qualifiers to dependent statements, the Graphviz [17] visualization tool is currently being used. It shows graph structures representing computations in a given LALP module, with possible scheduling information to help, whenever needed.

LALP also allows *nested loops*, which can be implemented by using distinct *counter* statements, and corresponding step signals for each loop. A simple example of a *for* loop nesting is presented bellow:

```
1  for(i=0;i<N;i++) {
2   for(j=0;j<M;j++) {
3   ...
4       aux = A[i*M+j];
5   ...
6   }
7  }
```

The corresponding LALP implementation of the nested loops presented above is shown below:

```
1  counter (i=0; i<N; i++)@j.done);
2  j.init = i.step
3  counter (j=0; j<M; j++);
4   ...
5   index = i*M+j;
6   A.address = index;
7   aux = A;
8   ...
```

Some loops taken from a DCT (Discrete Cosine Transform) implementation (Fig. 2) can be programmed as in the LALP code fragment shown in Fig. 3. Note that counters related to the inner loops start executing based on the value of the indexing variable of the counter related to the outermost loop. In this case, the first counter increments its indexing variable, `i`, by 64, every 72 clock cycles. This in turn will enable the start of a new iteration of the second loop, which is directly dependent on the new values of `i` being produced. A similar behaviour applies to the third loop, which is directly

```
counter (i=0; i<num_fdcts; i+=64@72);
i.clk_en = init;
i_plus_8 = i + 8;
  counter (j=i; j<i_plus_8; j++@9);
  j.clk_en = init;
  j.load = i.step;
  j_plus_64 = j + 64;
    counter (i1=j; i1<j_plus_64; i1+=8);
    i1.clk_en = init@2;
    i1.load = j.step;
    dct_io_ptr.address = i1;
    f0 = dct_io_ptr.data_out when (j.step@3);
    f1 = dct_io_ptr.data_out when (j.step@4);
    ...
counter (k=0; k<num_fdcts; k++);
k.clk_en = i.done@17;
  ...
```

Fig. 3 LALP example with nested loops

Fig. 4 C code for *Dotprod*

```
#define N 2048
int x[N], y[N];

int dotprod() {
  int i, sum = 0;
  for (i=0; i<N; i++)
    sum += x[i] * y[i];
  return sum;
}
```

dependent on the value `j`, the indexing variable of the second counter. This example shows the possibility of using more LALP counters than the actual number of loops in the original C code, based on the existence of other counter-based functionalities in the code (see variable `i1` in Fig. 2 and its correspondent counter in Fig. 3).

3.3 LALP Examples

For illustration purposes, the implementation of three hardware blocks using LALP is shown in this section: *Dotprod*, *Max*, and *Sobel*. *Dotprod* is a simple example showing the basic implementation of a loop accessing vectors of integers. *Max* was chosen to illustrate the use of conditional code in LALP, while *Sobel* exemplifies a more complex piece of code, often used as an image operator.

The function *Dotprod* performs the dot product of two integer vectors, as shown in the C source code in Fig. 4.

A possible implementation of the *Dotprod* function using LALP is shown in Fig. 5. As can be seen, integer and bit data types are type defined to the `fixed` LALP type, taking as parameters the number of storage bits, and the sign option, respectively. In the module code itself, a few LALP specific constructs are noticeable. The first one is the `counter` statement, which defines a *for* loop dependent on its indexing variable `i`. In

```
1  const DATA_WIDTH = 32;
2  const N = 2048;
3
4  typedef fixed(DATA_WIDTH, 1) int; // 1 means signed
5  typedef fixed(1, 0) bit;
6
7  dotprod_alp(out int sum, out bit done, in bit init) {
8    {
9      int x[N], y[N];
10     int acc;
11     fixed(16, 0) i;
12   }
13   counter (i=0; i<N; i++@1);
14   x.address = i;
15   y.address = i;
16   acc += x * y when i.step@1;
17   sum = acc;
18   done = i.done@2;
19 }
```

Fig. 5 *Dotprod* description in LALP

principle, all instructions directly or indirectly dependent on `i` can be simultaneously executed. Obviously this is not always possible, so synchronization delays can be inserted. That is the case of the `acc += x * y` operation, where the assignment to `acc` must wait 1 cycle after `i` has been redefined, and so allowing time for the multiplication completion. This delay is specified by means of the `when step@1` qualifier. As pointed out before, the actual value of the @ qualifier may be compiler inferred (Sect. 4.2).

A possible optimization can be done with little modifications in the LALP code for *dotprod*, in this case specifying a multiplier with 6 pipeline stages. This can be done by changing the LALP statement that computes the value of *acc*, as shown bellow.

```
1  acc += x *@6 y when i.step@7;
```

In case the input data from local memories (identified as `x` an `y`) is registered, a possible LALP solution would include two additional `@ sync` qualifiers, as shown below:

```
1  acc += (x@1) *@6 (y@1) when i.step@8;
```

As part of the compilation flow, a control/data flow graph (CDFG) is generated and scheduled, as shown in Fig. 6. In that graph, double-edged nodes represent registered operations. It can be seen that instructions are grouped in either of three execution cycles, for parallel execution within the respective cycle.

The second example, *Max*, determines the maximum value in a vector of integers. The LALP code is similar to the one presented for *dotprod*, especially in terms of reading a vector of integers. The main difference worth being illustrated is the conditional processing inside de loop (counter statement), as shown bellow.

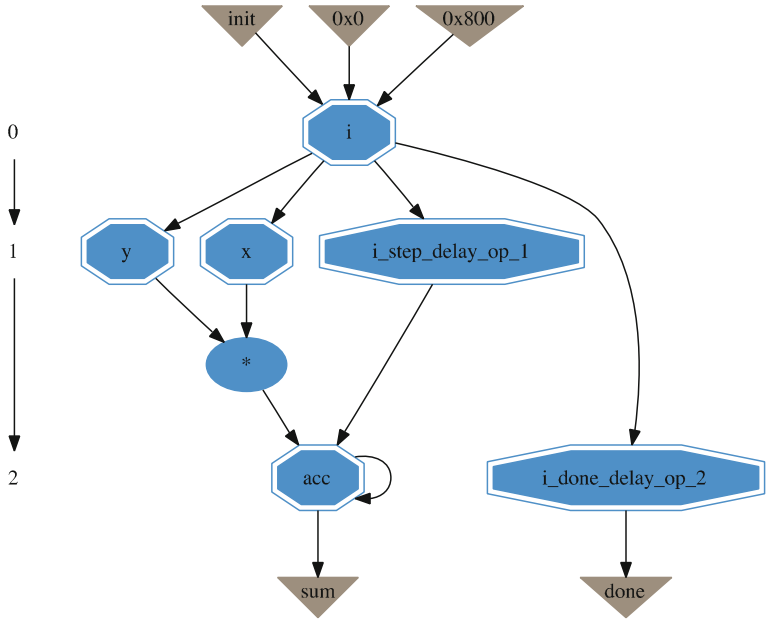

Fig. 6 Scheduled CDFG for *Dotprod*

```
1  ........
2  counter (i=0; i<N; i+=1);
3  v.address = i;
4  a = v; // the same as a = v.data_out;
5  b = a when (a > b) & (i.step@2);
6  maxval = b;
7  ........
8  }
```

Now the implementation of a *Sobel* image operator is presented, which is based on the original C source code illustrated in Fig. 7. The operator traverses an input image, here represented as a single-dimension array, using windows of 9 pixels to calculate resultant pixels. A possible LALP version of the *Sobel* image operator is presented in Fig. 8.

The original *for* `loop` is represented in the LALP version with the `counter` statement in line 20. The indexes related to the 8 load operations from array `in` (see Fig. 7) are represented by the 8 assignments to variable `addr` (see Fig. 8). Those assignments are scheduled to different clock cycles by means of the `@ sync` qualifiers. Variable `addr` is then assigned to the memory address port identified as `input` (line 29). The values related to those array references are then read and assigned to 8 different variables at consecutive clock cycles (lines 30–37), which are also scheduled using `@ sync` qualifiers. Lines 38 and 39 implement the calculations of `H` and `V` using the previously loaded values, in this implementation using multipliers with 6 pipeline stages (identified by `*@6`). Following the code related to the if constructs in

```
1  #define cols 10
2  #define rows 10
3  #define N cols*rows
4
5  char in[N];
6  char out[N];
7
8  void sobel() {
9    int H, O, V, i;
10   int i00, i01, i02;
11   int i10,      i12;
12   int i20, i21, i22;
13   for (i = 0; i < cols*(rows-2)-2; i++) {
14     i00=in[i       ]; i01=in[i       +1]; i02=in[i       +2];
15     i10=in[i+ cols];                      i12=in[i+ cols+2];
16     i20=in[i+2*cols]; i21=in[i+2*cols+1]; i22=in[i+2*cols+2];
17     H = -  i00 - 2*i01 -  i02 +
18          +  i20 + 2*i21 +  i22;
19     V = -  i00          +  i02
20          - 2*i10          + 2*i12
21          -  i20          +  i22;
22     if (H<0)
23       H = -H;
24     if (V<0)
25       V = -V;
26     O = H + V;
27     if (O > 255)
28       O = 255;
29     out[i + 1] = (char)O;
30   }
31 }
```

Fig. 7 Sobel C source code

the original C code, there is the LALP code responsible for saving the resulting computations (`Otrunk`) in the array stored in the local memory identified as `output`. At this stage, the `i` values output from the counter component would have passed through a 19-level shift-register (`@ sync` qualifier included in line 45), before reaching the memory address port. Once again, the actual values of the @ qualifiers may be compiler inferred, as discussed in Sect. 5.4.

4 Mapping LALP Code into Hardware

Transforming LALP source code into a synthesizable hardware description is accomplished by means of a compilation framework, whose main tasks are represented in Fig. 9. The *front-end* takes as input a program implemented in LALP, and after parsing it, generates the corresponding CDFG (Control-Data Flow Graph), as described in Sect. 4.1. The next stages are responsible for the core *mapping* tasks, and include *scheduling*, *balancing*, and *synchronization* of operations. In these stages, a number of analysis and modifications are performed over the CDFG, using the algorithms described in Sects. 4.2 and 4.3. During the final steps of the compilation flow, the *back-end* generator (Sect. 4.4) selects components from an existing VHDL library, and wires them according to the constraints specified in the scheduled CDFG. The resulting VHDL description is then ready for RTL (Register Transfer Level) synthesis. Auxiliary input (e.g., C code) and output (DOT Graphviz) files are also shown in

```
const DATA_WIDTH = 16;
const ROWS = 10;
const COLS = 10;
const SIZE = ROWS*COLS;

typedef fixed(DATA_WIDTH, 1) int; // signed 16 bits
typedef fixed(1, 0) bit;
typedef fixed(8, 0) byte; // unsigned 8 bits

sobel(in bit init, out bit done) {
  {
    int H, O, V, Hpos, Vpos, Otrunk;
    int i, addr;
    int i00, i01, i02;
    int i10,      i12;
    int i20, i21, i22;
    byte input[SIZE];
    byte output[SIZE];
  }
  counter (i=0; i<78; i+=1@8);
  addr = i;
  addr = (i) + 1 when i.step@1;
  addr = (i) + 2 when i.step@2;
  addr = (i) + COLS when i.step@3;
  addr = ((i) + COLS) + 2 when i.step@4;
  addr = ((i) + COLS) + COLS when i.step@5;
  addr = (((i) + COLS) + COLS) + 1 when i.step@6;
  addr = (((i) + COLS) + COLS) + 2 when i.step@7;
  input.address = addr;
  i00 = input when i.step@2;
  i01 = input when i.step@3;
  i02 = input when i.step@4;
  i10 = input when i.step@5;
  i12 = input when i.step@6;
  i20 = input when i.step@7;
  i21 = input when i.step@8;
  i22 = input when i.step@9;
  H = ((-i00)+(-2 *@6 i01))+(((-i02)+i20)+(2 *@6 i21+i22));
  V = ((-i00)+i02)+(((-2 *@6 i10)+2 *@6 i12)+((-i20)+i22));
  Hpos = H < 0 ? -H : H;
  Vpos = V < 0 ? -V : V;
  O = Hpos + Vpos;
  Otrunk = O;
  Otrunk = 255 when O > 255;
  output.address = i@19;
  output.data_in = Otrunk;
  done = i.done@19;
}
```

Fig. 8 Sobel LALP implementation

Fig. 9, but those are mainly used to help the programmer in the process of creating and optimizing LALP code.

4.1 CDFG Generation

The CDFG is built based on operations and operands present in the LALP source code. After the CDFG generation, the compiler performs the core stages of the mapping

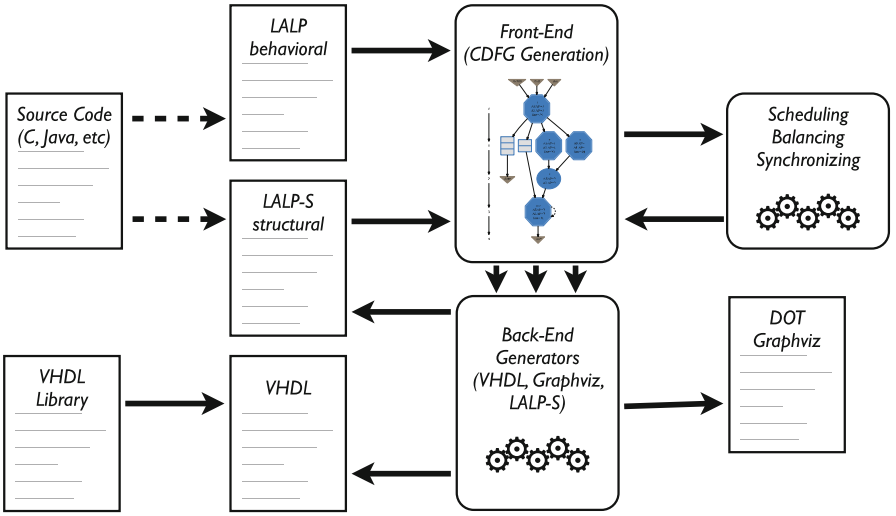

Fig. 9 The complete compilation flow

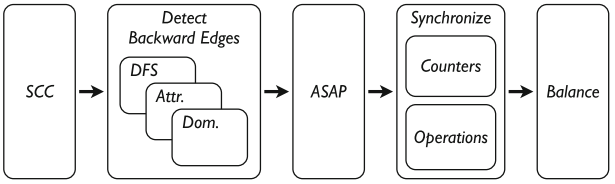

Fig. 10 Core stages of the mapping process

process, as represented in Fig. 10. For scheduling purposes, it is necessary to identify all back edges, which are responsible for recurrences, i.e. dependences between instructions from distinct iterations. The presence of recurrences may impose limitations to the rate of iterations execution. While conservative assumptions may unnecessarily degrade performance, poor analysis can result in wrong results being produced.

In order to identify back-edges and circuits in the CFDG, the strongly connected components (SCC) in the graph are first detected [18], helping to reduce the complexity of the problem by considering a smaller number of nodes at a time. For reducible graphs, the back edges can be identified by using a *tree dominance* algorithm, or *depth-first search* [19]. Back edges are the ones having a given component as source, and one of its dominators as destination.

Although only natural loops are used, (`goto`, `break`, and `continue` statements are not allowed), the resulting CDFGs can be irreducible, which prevents the applicability of this technique. In this case, the scheme is not able to differentiate dependences involving computations from the same or from distinct iterations. To deal with such cases, several different approaches have been proposed in the literature [20]. The LALP compilation framework identify back edges in irreducible graphs based on the topological ordering of instruction attributions (Attr) existing in the input code. As part of this process, and in order to guide the scheduling process, the user may need

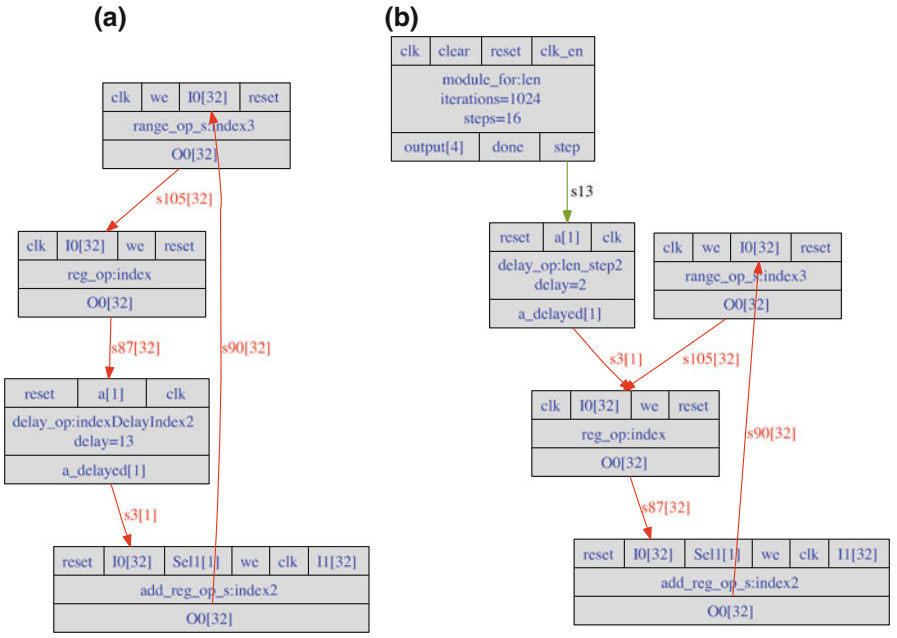

Fig. 11 Synchronized schedule: **a** using delays; **b** using *step* signal

to furnish to the compiler recurrences and back edges in the CDFG in the form of annotations.

4.2 Scheduling and Balancing

The scheduling process determines the execution cycle of each operation in the CDFG. The relative ordering of operations is determined based on their respective predecessors and successors. As opposed to software pipelining techniques targeting microprocessors, the scheduling approach used when compiling LALP does not assume sharing of components, apart from memory ports. Furthermore, the relative ordering between loads and stores to the same memory - multiple loads can be performed concurrently if using a multi-port memory—is kept as defined in the source code. These two observations relax the need for a list scheduling based technique.

Before the scheduling actually starts, all back edges are removed, as the scheduler has access to the CFDG recurrence annotations. The minimal number of clock cycles necessary to execute one loop iteration is currently being calculated based on the longest recurrence edge. The highest value obtained determines the minimum execution rate. All components (instructions) should ideally be accommodated within the calculated number of cycles, with possible delays employed to order and synchronize the execution of instructions (see Fig. 11a).

Scheduling is carried out using an ASAP (as soon as possible) strategy, without resource constraints. Starting from the first node in the graph, consumers are scheduled

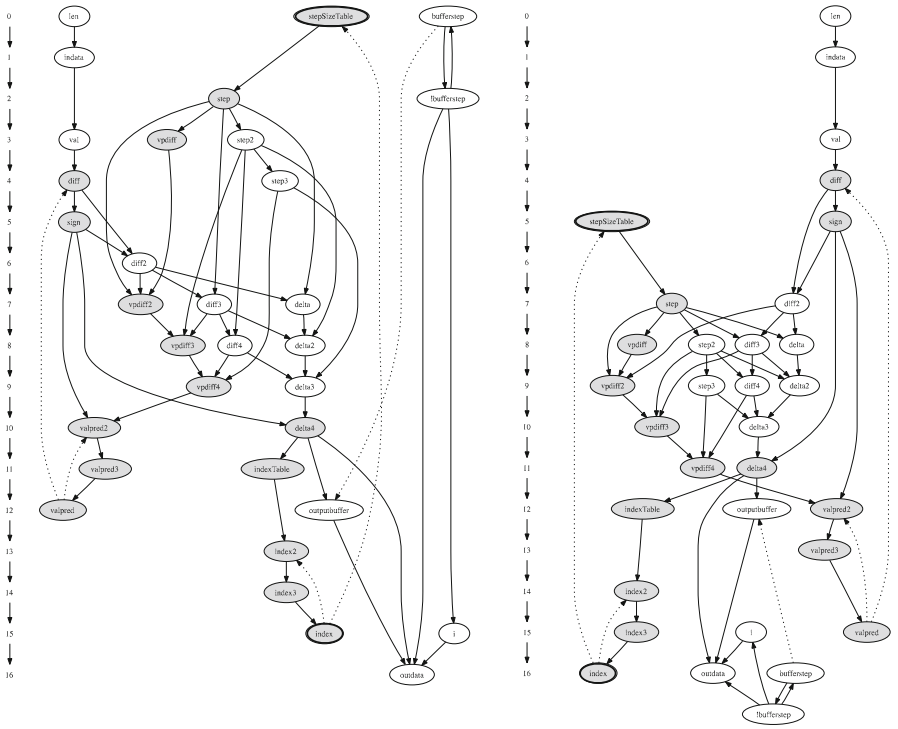

Fig. 12 Schedules for *ADPCM Coder*: **a** Original ASAP. **b** Enhanced algorithm

closer to producers. The algorithm runs interactively, until no change in schedule the length occurs between two consecutive runs. A particular optimization employed in the ASAP algorithm accounts for operations having predecessors only from previous iterations. Because back-edges are removed, those operations would be scheduled right at the beginning of the schedule, according to the basic ASAP logic. Instead of doing that, the enhanced algorithm schedules those operations close to their successors. In order to show the impact of this strategy, two schedules of an ADPCM coder have been produced. The first schedule is shown in Fig. 12a, which was produced using the basic ASAP algorithm. The highlighted instructions show a recurrence that in practice determines the rate of execution of a single iteration, i.e., 16 cycles (0 to 15). The second schedule, shown in Fig. 12b, was obtained with the enhanced algorithm. The highlighted instructions show a recurrence of 12 cycles (5 to 16), an improvement also observed in other experiments.

The schedule of operations is supported by a *dedicated counter*, used to synchronize the operations within each loop body found in the code. The counter component generates a *step* signal that stays high one cycle per iteration, and is propagated through the loop pipelining stages. By doing so, the same execution rate is enforced (see Fig. 11b).

Besides the ASAP scheduling algorithm, a second one is also employed, ALAP (as late as possible). It starts from the last scheduled node, and tries to move producers closer to consumers. The main reason for doing so is to gather information in

```
const c14 = 2048;
const c13 = 0;

dotprod_alp(out fixed(1,0) done, out fixed(32,1) sum, in fixed(1,0)
    init) {
  {
    i: counter(16, 1, 1, 0, 0);
    x: block_ram(11, 32);
    y: block_ram(11, 32);
    x_data_out_mult_op_s_y_data_out: mult_op_s(32);
    acc: add_reg_op_s(32);
    i_done_delay_op_2: delay_op(1, 2);
    i_step_delay_op_1: delay_op(1, 1);
  }
  i.input <-(s0) c13;
  i.termination <-(s1) c14;
  i.clk_en <-(s2) init;
  x.address <-(s3) i.output;
  y.address <-(s4) i.output;
  x_data_out_mult_op_s_y_data_out.I0 <-(s5) x.data_out;
  x_data_out_mult_op_s_y_data_out.I1 <-(s6) y.data_out;
  i_step_delay_op_1.a <-(s7) i.step;
  acc.I0 <-(s8) acc.O0;
  acc.I1 <-(s9) x_data_out_mult_op_s_y_data_out.O0;
  acc.we <-(s10) i_step_delay_op_1.a_delayed;
  sum <-(s11) acc.O0;
  i_done_delay_op_2.a <-(s12) i.done;
  done <-(s13) i_done_delay_op_2.a_delayed;
}
```

Fig. 13 CDFG representation of the dotprod example using LALP-S

order to *balance* the final schedule. After running both schedulers, the difference in cycles between both schedules is computed for each operation. That is called *operation mobility*. It is important that the execution time required by alternative paths (e.g., following branch statements) are equivalent. To ensure that, registers may be inserted within the shortest path(s), in order to balance the execution times leading to a common destination.

4.3 CDFG Manipulation

Besides the generation of the CDFG in a visual format, the user is also allowed to have access to a textual description of it. This textual description is described in a structural language, briefly called LALP-S (a structural view of LALP). The graph in Fig. 13 illustrates the *Dotprod* example in LALP-S. The generation of the CDFG in LALP-S allows the user to make changes at this stage, if required. By doing so, a VHDL-RTL description can also be generated from LALP-S, (i.e., the compilation flow is able to process as input LALP-S descriptions besides the standard LALP programs). LALP-S can also be used to directly describe the target hardware engine.

As can be seen in the example in Fig. 13, LALP-S descriptions represent components in the CDFG, and interconnections among them. Thus, this representation can be directly translated into VHDL-RTL, as long as the VHDL library includes a representation for each component used, and the interconnections respect the interfaces between components.

4.4 Back-End: VHDL Generation

As with most hardware compiler approaches, the resulting hardware description is produced based on a library of VHDL components. They represent basic operations and hardware resources usually present in FPGAs. Those resources can be basic arithmetic functional units, counters, accumulators, configurable memory blocks, and even digital signal processing blocks. Based on that library, a VHDL representation of the CDFG, ready for RTL synthesis, is produced. The library been used has a total of 56 components, and its main limitation is the lack of support for floating point operations. The VHDL-RTL descriptions are then submitted to a standard tool (e.g Xilinx ISE) capable of RTL synthesis, mapping, placement and routing, and finally generating the FPGA bitstreams.

5 Experimental Results

This section presents the results achieved using LALP to map a number of kernels, of varying complexity, to FPGAs. In addition to the examples previously presented, other examples existing in public benchmark repositories were considered. Whenever possible, implementations based on LALP were compared with implementations obtained using selected C to hardware compilers. For LALP descriptions, the same algorithms previously presented in C source code were used. This is important as it may reflect a possible (semi-)automatic conversion from C to LALP. On-chip memories (BRAMs) were used to store input/output data for all the experiments. A maximum of one load/store per clock cycle was allowed. So, it can be inferred that more optimized results might have been achieved if multi-port memories were employed.

For the resulting designs, the latencies presented were measured by counting the number of clock cycles needed for their respective executions. The measurements were done using a VHDL simulator. For microprocessor-based solutions, the latencies presented were measured with hardware timers, executing the benchmarks using a given target processor, and without operating system support. The latencies correspond to the best system conditions as all instructions and data were available from local on-chip FPGA memories.

5.1 Benchmarks

The selected benchmarks include some typical kernels of *embedded applications*, like signal and image processing, encoding/decoding, etc. In addition, some simple kernels that exhibited some characteristics of interest during development and testing were also used. Those benchmarks are: *ADPCM Coder, ADPCM Decoder, Autcor, Bubble Sort, Dotprod, FDCT, Fibonacci, Max, Pop Count, Sobel*, and *Vector Sum*. The main characteristics of the original C code of each benchmark are shown in Table 2. The code length of each benchmark ranges from 10 to 145 lines of code, and contains between 1 and 3 loops. The table also shows the number of declared arrays, and if constructs in each benchmark. For comparison purposes, the number of lines of code in LALP and VHDL, respectively, are also presented.

Table 2 Source code characteristics for each benchmark

Benchmark	Lines of code			Loops	Arrays	Ifs
	C	LALP	VHDL			
ADPCM Coder	83	71	1,718	1	4	10
ADPCM Decoder	70	60	1,352	1	4	9
Autocorrelation	16	29	470	2	2	0
Bubble Sort	15	31	418	2	1	1
Dotprod	10	18	225	1	1	0
FDCT	145	175	5,290	3	3	0
Fibonacci	10	19	202	1	1	0
Max	10	18	225	1	1	1
Pop Count	11	118	2,294	2	2	0
Sobel	36	52	1,298	1	2	3
Vector Sum	12	20	234	1	1	0

5.2 Speedup Achieved and Hardware Resources

Each of those benchmarks was programmed in LALP, taking the original C code as a reference. It should be noticed that the computing strategy for each benchmark has been maintained in the manual translation from C to LALP, unless for those identified by the symbol ◇, corresponding to versions with "data reuse". Data reuse aims to reduce memory accesses, and can be achieved by using shift-registers. Also note that in these LALP-based implementations, no manual changes to the respective CDFGs (using LALP-S descriptions) were made.

Results for hardware resources, maximum operating frequency, and execution time are shown, considering three implementations: (1) using the LALP language to program the FPGA; (2) using a C to hardware tool (referred herein as C2Verilog [21]) with public web access; (3) using the ROCCC compiler [22]. The target hardware is a Xilinx Virtex 6 FPGA (XC6VLX75-T3FF484), using ISE 11.4 as the back-end tool. The maximum clock frequencies shown are estimations reported by the ISE tool.

The high clock frequencies obtained for the LALP implementations presented in Table 3 are due to the use of the aggressive loop pipelining approach (see Sect. 2). Two marked examples, *Sobel*◇ and *Fibonacci*◇, are versions using data reuse. This feature allowed to speedup *Sobel* by 9.3×, using however 1.26× more slices. As for *Fibonacci*, the achieved speedup was 3.6×, using in this case 0.74× of the slices used by the version without data reuse. The throughput results are based on the theoretical maximum achievable, i.e., processing *one* single data item per clock cycle, considering the memory access limitations already mentioned. A throughput greater than 0.9 was achieved for 6 benchmarks, with an overall average of 0.6. To further increase the throughputs, the original algorithms would have to be substantially changed.

The results obtained using ROCCC are presented in Table 4, although the tool was not able to compile all the benchmarks considered. Best efforts were made in several attempts to compile some benchmarks, including source code modifications, but

Table 3 Hardware resources and performance obtained with LALP

Benchmark	Data	FFs	LUTs	Slices	DSPs	Max freq. (MHz)	Latency	Exec. time (us)	Throughput
ADPCM Coder	1,024	798	916	257	0	535.017	12,288	22.97	0.083
ADPCM Decoder	1,024	543	596	181	0	535.017	3,089	5.77	0.331
Autocorrelation	1,600	312	161	44	0	669.703	3,010	4.49	0.532
Bubble Sort	32	225	197	61	0	423.889	4224	9.96	0.008
Dotprod	2,048	93	79	23	3	530.786	2,050	3.86	0.999
FDCT	640	5,960	5,110	1,322	56	483.676	1,430	2.96	0.448
Fibonacci	32	113	101	31	0	514.033	96	0.19	0.333
Fibonacci◇	32	107	81	23	0	623.364	32	0.05	1.000
Max	2,048	88	67	18	0	464.165	2,050	4.42	0.999
Pop Count	1,024	97	65	23	0	534.674	1,029	1.92	0.995
Sobel	100	329	358	140	0	264.061	639	2.42	0.156
Sobel◇	100	777	627	177	0	423.711	109	0.26	0.917
Vector Sum	2,048	177	113	34	0	546.538	2,050	3.75	0.999

◇ versions with data reuse

Table 4 Hardware resources and performance obtained with ROCCC

Benchmark	Data	FFs	LUTs	Slices	DSPs	Max freq. (MHz)	Lat.	Exec. time (us)	Through.	Improvements LALP/ROCCC	
										Slices	Speedup
Fibonacci	32	130	208	54	0	364.79	33	0.09	0.970	0.57	0.48
Fibonacci◇	32	130	208	54	0	364.79	33	0.09	0.970	0.43	1.76
Sobel	100	1371	1308	351	0	313.59	114	0.36	0.877	0.40	0.15
Sobel◇	100	1371	1308	351	0	313.59	114	0.36	0.877	0.50	1.41
Vector Sum	2048	554	751	200	0	291.98	2052	7.03	0.998	0.17	1.87
									Average	0.41	1.14

◇ versions with data reuse

unfortunately no success was achieved for some of them. The results obtained using ROCCC reveal that the compiler was able to do data reusing for the *Sobel* and *Fibonacci* benchmarks. In these cases, the results were better than the ones obtained using LALP without data reuse. However, LALP results using data reuse outperformed those using ROCCC. For the three examples shown in Table 4, LALP obtained an average speedup over ROCCC equals to 1.14×. With respect to the required hardware resources, LALP implementations used on average 59% less slices than ROCCC.

The results obtained using the C2Verilog compiler are shown in Table 5. In this case it was possible to evaluate and compare almost all of the benchmarks considered. Like ROCCC, C2Verilog was also able to efficiently perform data reuse for the *Sobel* and *Fibonacci* benchmarks. The LALP implementations were able to use on average 46% less slices than the implementations obtained by C2Verilog. C2Verilog achieved an

Table 5 Hardware resources and performance obtained with C2Verilog

Benchmark	Data	FFs	LUTs	Slices	DSPs	Max freq. (MHz)	Lat.	Exec. time (us)	Through.	Improvements LALP/C2V	
										Slices	Speedup
ADPCM Coder	1,024	990	701	522	0	515.62	40,963	79.44	0.025	0.49	3.46
ADPCM Decoder	1,024	750	611	528	0	529.63	25,346	47.86	0.040	0.34	8.29
Autocorrelation	1,600	6,296	6,729	1,738	3	161.01	10,631	66.03	0.151	0.03	14.69
Bubble Sort	32	2,353	4,904	1,369	0	261.77	4,098	15.66	0.008	0.04	1.57
Dotprod	2,048	744	582	476	0	303.26	14,348	47.31	0.143	0.05	12.25
Fibonacci	32	45	38	11	0	688.21	34	0.05	0.941	2.82	0.26
Fibonacci◇	32	45	38	11	0	688.21	34	0.05	0.941	2.09	0.96
Max	2,048	490	474	127	0	521.84	6,151	11.79	0.333	0.14	2.67
Pop Count	1,024	1,023	1,036	328	0	630.46	4,107	6.51	0.249	0.07	3.38
Sobel	100	4,616	6,322	1,716	0	433.15	1,730	3.99	0.058	0.08	1.65
Sobel◇	100	4,616	6,322	1,716	0	433.15	1,730	3.99	0.058	0.10	15.53
Vector Sum	2,048	664	660	169	0	665.38	6,150	9.24	0.333	0.20	2.46
									Average	0.54	5.60

◇ versions with data reuse

average throughput of 0.27, with only the *Fibonacci* example achieving a throughput greater than 0.9. The LALP implementations achieved on average a speedup of 5.1× (5.6× considering also the LALP examples with data reuse) over C2Verilog. It should be noticed that in some cases the number of slices required by the C2Verilog compiler is much higher. One possible explanation is that the number of memory accesses present in the original C code may generate non-optimized hardware structures. That could be the case as the tool documentation advises users to minimize memory operations, and few modifications were made in the original source code for those experiments.

The maximum clock frequencies obtained by LALP implementations were higher, on average 1.53× and 1.04× for ROCCC and C2Verilog, respectively. This can be justified by the more aggressive loop pipelining approach used by LALP implementations. LALP implementations have also been compared with the ones obtained using SPARK [10] and C2H [11]. For both, similar speedups were also obtained, along with savings in hardware resources, as presented in [23]. The results show evidence that LALP can indeed achieve speedups using less hardware resources than some of the other available tools. As claimed, LALP can be used to implement loop accelerators if one needs faster loop accelerators than the ones obtained by some others C to HDL compilers.

The observed savings in hardware resources obtained by LALP implementations may be justified by the use of the aggressive loop pipelining technique. It implements loop pipelining without using hardware resources to deal with epilogue, kernel, and prologue stages of software inspired loop pipelining techniques. Although those stages are present when executing the loops, they are not present as hardware dedicated stages, as already identified and discussed in [14]. LALP does not need to create control states and assigning operations to them. Instead, it reuses a single operation implementation for any execution stage that requires it.

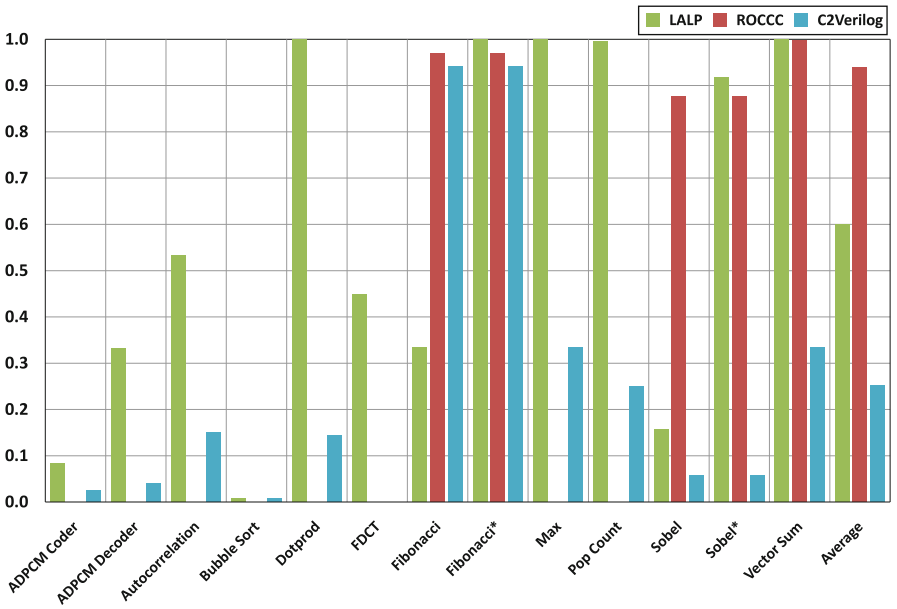

Fig. 14 Throughput comparison

Performance results are also shown in the form of *throughput* (Fig. 14) and *normalized execution time* (Fig. 15). Those figures reflect speedups ranging from 0.3× to 15.5×, when moving from the commercial tool to the LALP implementations (5.6× on average).

It can be inferred from those results that the *throughput* achieved by LALP, measured as the number of data words processed per clock cycle, is significantly higher. It should be noticed that the examples that could be compiled by ROCCC have all a high throughput, which is also reflected on the high average throughput obtained for this tool.

5.3 Comparing LALP Results with Microprocessor-Based Solutions

A further insight in the potential of using LALP can be obtained by comparing its performance results with those from the execution of the same benchmarks with microprocessors. As seen in Table 6, figures using the SimpleScalar/ARM simulator [24, 25], one of the two PowerPC440 hardcores included in the Xilinx Virtex-5 FPGA (XC5VFX130T-2FFG1738CES), and Altera Nios-II [26] were obtained. The LALP implementations achieved on average a speedup of 29× over SimpleScalar/ARM, 23× over PowerPC, and 305× over Nios-II. If clock frequencies are scaled up to 500 MHz, the respective LALP speedups would still be 12×, 19× and 52×, on average.

5.4 Programming Efforts: Automatic Versus Manual

The current capabilities of the LALP framework may require explicit programmer intervention, based on the comparison of the results obtained from the automatic

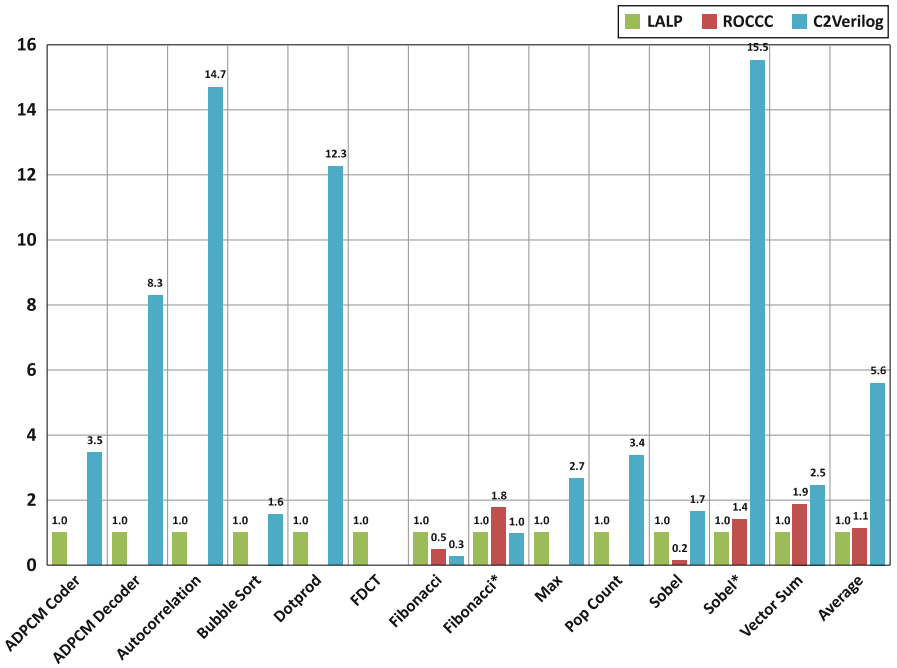

Fig. 15 Normalized execution time

Table 6 Performance obtained with microprocessors

Benchmark	*LALP*		ARM (200 MHz)		PPC (400 MHz)		Nios II (85 MHz)	
	Data	Exec. time (us)	Exec. time (us)	LALP speedup	Exec. time (us)	LALP speedup	Exec. time (us)	LALP speedup
ADPCM Coder	1,024	22.97	278.34	12.12	103.19	4.49	856.83	37.31
ADPCM Decoder	1,024	5.77	216.03	37.42	89.99	15.59	756.29	130.99
Autocorrelation	1,600	4.49	178.86	39.80	33.77	7.51	194.87	43.36
Bubble Sort	32	9.96	46.87	4.70	16.62	1.67	156.46	15.70
Dotprod	2,048	3.86	72.65	18.81	79.80	20.66	1,403.07	363.28
FDCT	640	2.96	222.30	75.19	26.71	9.04	296.67	100.34
Fibonacci	32	0.19	15.22	81.47	1.93	10.33	9.04	48.42
Max	2,048	4.42	72.60	16.44	49.11	11.12	234.62	53.12
Pop Count	1,024	1.21	11.49	9.49	291.01	151.21	3,815.09	1,982.34
Sobel	100	1.09	22.86	20.95	5.30	2.19	41.51	17.16
Vector Sum	2,048	3.75	83.19	22.18	77.27	20.60	2,130.21	567.92
Average				29.41		23.13		305.45

compilation, against those using the original C code. This section presents and discusses the number of *sync* qualifiers (@s) in LALP implementations that were resolved automatically, versus the ones required to be defined manually by the programmer.

Table 7 Synchronization directives—*Sync* qualifiers (@s)

Benchmark	Manual process	Semi-automatic process	
	Man. inserted	Aut. inserted	Man. inserted
ADPCM Coder	28	4	28
ADPCM Decoder	24	3	22
Autocorrelation	3	3	–
Bubble Sort	5	5	–
Dotprod	2	–	2
FDCT	119	39	118
Fibonacci	5	5	–
Fibonacci◇	1	–	1
Max	2	1	1
Pop Count	1	–	1
Sobel	26	16	13
Sobel◇	–	12	15
Vector Sum	3	–	3

◇ versions with data reuse

Table 7 shows two possible cases. The first one (Manual Process) represents the cases where a programmer defines manually all the *sync* qualifiers needed to obtain the correct LALP behaviour. The second one (Semi-Automatic Process) represents the case where the programmer delegates to the compiler the insertion of the *sync* qualifiers. The second process usually needs more *sync* qualifiers (22% more for these examples), and is able to determine only a subset of the total number required. For these examples, 65% of the total *sync* qualifiers needed to be inserted manually. Although this does not solve the need to manually insert some *sync* qualifiers, it can be an important help to the programmer. Future work should investigate and develop more advanced analysis algorithms able to increase the number of *sync* qualifiers that can be automatically inserted.

With respect to the experimental results obtained for the manual insertion and semi-automatic insertion of *sync* qualifiers, similar results were obtained for all benchmarks except for FDCT. For that benchmark the semi-automatic approach employed 40% more slices due to the use of a larger number of shift-registers propagating control signals.

5.5 Pipelining Exploration

One of the advantages of LALP is the possibility to evaluate different implementations by just changing the values in *sync qualifiers*. Figure 16 illustrates an example of the design space exploration conducted with the *Dotprod* example. In this case, the experiment evaluated the impact on hardware resources, maximum clock frequency, and latency, when increasing the number of pipelining stages of the multiplier used in the example. This exploration has been done by only modifying the values of *sync qualifiers* in the LALP code. Specifically, three @s have been changed for each design option.

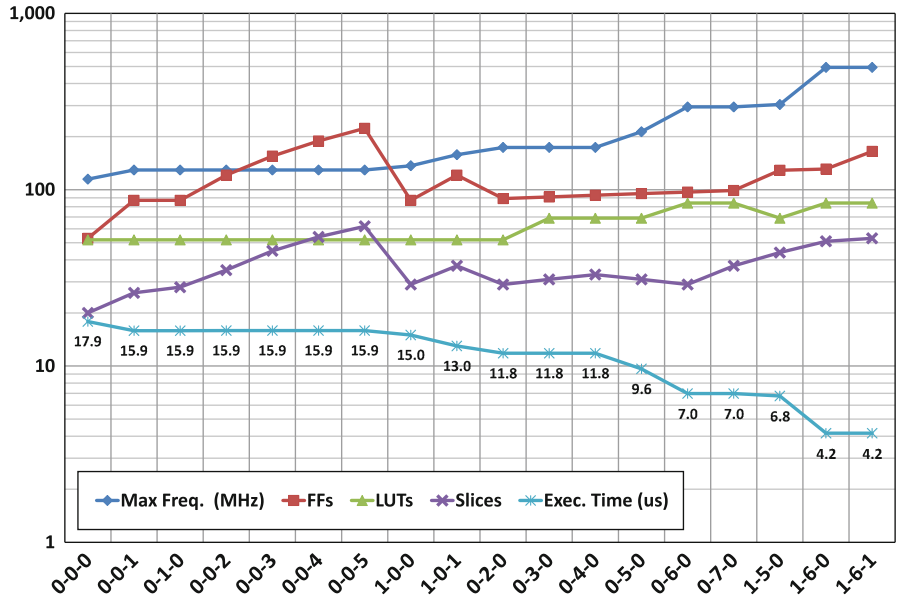

Fig. 16 Design space exploration for the *Dotprod* example

The results for this experiment are presented in Fig. 16. The 3-number points, *n-m-k*, in the X axis of Fig. 16, represent respectively: the existence (n=1) or absence (n=0) of a register in the data output of the local memory; the number of stages of the multiplier (m); and the number of registers in the output of the multiplier (k). As can be seen in those results, the more aggressive loop pipelining version uses a six-stage multiplier, achieving a speedup of 4.29× over the loop pipelining version using a single-stage multiplier. This speedup was obtained due to a similar increase in the maximum clock frequency, and required an implementation with 2.65× more Slices than the version with a single-stage multiplier. As an example of a trade-off analysis, an alternative implementation achieved a speedup of 2.56× using 1.45× more Slices.

6 Related Work

The work presented in this paper addresses an aggressive loop pipelining (ALP) scheme and a domain-specific language (LALP) to achieve efficient FPGA-based solutions integrating ALP. Due to its performance benefits, loop pipelining has been the focus of many research efforts [3] and may be considered a requirement to implement an efficient hardware compiler. In the context of configurable or custom architectures, most notably FPGA-based ones, many researchers have exploited loop pipelining. Most of the work focuses on innermost loop pipelining (see, e.g., [5,9,27] and [6]), and uses some loop transformations (e.g., unrolling, interchange, tiling) to enhance the applicability of loop pipelining [9]. Most approaches heavily rely on resource constraints (important when targeting processors and ASICs) and scheduling schemes. Those strategies may produce non-optimized results even though the only constraint

might be the port accesses to non-customized memories in the target architecture (the off-chip memories in FPGAs). One of such approaches is Modulo Scheduling, which can be efficiently performed by using Rau's Iterative Modulo Scheduling (IMS) algorithm [7]. It is believed that most hardware compilers use the IMS algorithm.

The approach to loop pipelining presented in this paper resembles more the one presented in [14], which proposes a dynamic loop pipelining scheme applied to a coarse-grained reconfigurable, data-driven architecture. Although not using a data-driven scheme, the approach presented in this paper also uses the notion of counters, with speed limited by the maximum possible rates for loading and storing data, along with inter-loop dependences.

In terms of programming languages, most efforts addressed the mapping of computations described in traditional software languages (mainly C) to FPGAs. The main reasons for that are the wide use of those languages, and the wide spread availability of legacy code. In the context of FPGAs, domain-specific languages have been more focused on specific reconfigurable architectures (such as DIL for PipeRench [28], and RapiD-C for the RaPID architecture [29]), than on general programming of FPGAs. The most well-known exceptions are Handel-C and the Haydn approach [30]. The later makes possible for the programmer to explicitly specify control stages using annotations and the computations for each stage. Regarding loop pipelining, the approach is able to specify certain aspects of Modulo Scheduling (e.g., the initiation interval, II). The LALP approach is different from the previous ones. It permits achieving optimized loop pipelining schemes by allowing the programmer to control relative clock cycles, registered assignments, and hardware counters, bearing in mind possible automatic optimizations and suggestions, along with a future path from C language to LALP.

7 Conclusions

This paper described a novel approach to program hardware accelerators implemented in FPGAs. The approach uses a domain-specific language (LALP), specially tailored to program application-specific architectures taking advantage of loop pipelining and other concurrency opportunities. LALP allows the user to specify loop computations using a behavioural description, with abstraction closer to the original software program.

LALP is able to express aggressive loop pipelining techniques targeting FPGAs, aiming to achieve higher performance levels than related techniques, especially those based on software pipelining. Experimental results have shown the usefulness of LALP to program hardware accelerators, achieving higher performance levels than those allowed by some academic and commercial C to hardware tools, in this paper employed for comparison purposes only. This is an evidence that the proposed approach may be considered a valid alternative if the design generated by other C to hardware tools do not meet certain performance or resources usage requirements.

Ongoing work on LALP and its supporting framework is concerned with improved algorithms to automate some mapping tasks, offering additional support to the programmer on data dependences and scheduling issues. In addition, compiler support for

automatic translation of C code to LALP is under development. This effort is justified not only due to the more familiar syntax of C language, but also as a way to incorporate in the LALP compilation flow other analyses and transformations available in existing optimizing compilers for the C programming language.

Acknowledgments This work has been partially funded by a CNPq/Grices bilateral Brazil/Portugal project. Ricardo Menotti, Marcio Fernandes, and Eduardo Marques are also members of INCT-SEC and acknowledge the support of CNPq and FAPESP, under grants 573963/2008-8 and 08/57870-9. João Cardoso also acknowledges the partial funding support of FCT, Portugal, under grant PTDC/EEA-ELC/70272/2006. We acknowledge Adriano Sanches for the support regarding the measurements with PowerPC.

References

1. Densmore, D., Passerone, R., Sangiovanni-Vincentelli, A.: A platform-based taxonomy for ESL design. IEEE Des. Test Comput. **23**(5), 359–374 (2006)
2. Edwards, S.A.: The challenges of synthesizing hardware from C-like languages. IEEE Des. Test Comput. **23**(5), 375–386 (2006)
3. Allan, V.H., Jones, R.B., Lee, R.M., Allan, S.J.: Software pipelining. ACM Comput. Surv. **27**(3), 367–432 (1995)
4. Callahan, T.J., Hauser, J.R., Wawrzynek, J.: The GARP architecture and C compiler. Computer **33**(4), 62–69 (2000)
5. Haldar, M., Nayak, A., Choudhary, A., Banerjee, P.: A system for synthesizing optimized FPGA hardware from matlab. In: ICCAD '01: Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design, pp. 314–319. IEEE Press, Piscataway, NJ, USA (2001)
6. Snider, G.: Performance-constrained pipelining of software loops onto reconfigurable hardware. In: FPGA '02: Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays, pp. 177–186. ACM Press, New York, NY, USA (2002)
7. Ramakrishna Rau, B.: Iterative modulo scheduling: an algorithm for software pipelining loops. In: MICRO 27: Proceedings of the 27th Annual International Symposium on Microarchitecture, pp. 63–74. ACM Press, New York, NY, USA (1994)
8. Aiken, A., Nicolaum, A.: Perfect pipelining: a new loop parallelization technique. In: ESOP '88: Proceedings of the 2nd European Symposium on Programming, pp. 221–235. Springer, London, UK (1988)
9. Weinhardt, M., Luk, W.: Pipeline vectorization. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **20**, 234–248 (2001)
10. Gupta, S., Gupta, R., Dutt, N., Nicolau, A.: SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits. Kluwer, Dordrecht (2004)
11. Altera Corporation: Nios II C2H compiler user guide (2009). http://www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf
12. Menotti, R., Cardoso, J.M.P., Fernandes, M.M., Eduardo, M.: Automatic generation of FPGA hardware accelerators using a domain specific language. In: International Conference on Field Programmable Logic and Applications (FPL), pp. 457–461. (2009)
13. Menotti, R., Manuel J.M.P., Fernandes, M.M., Eduardo, M.: LALP: a novel language to program custom FPGA-based architectures. In: Proceedings of the 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 3–10. IEEE Computer Society Press, Los Alamitos, CA, USA (2009)
14. Cardoso, J.M.P.: Dynamic loop pipelining in data-driven architectures. In: CF '05: Proceedings of the 2nd Conference on Computing Frontiers, pp. 106–115. ACM Press, New York, NY, USA (2005)
15. Rodrigues, R., Cardoso, JM.P., Diniz, P.C.: A data-driven approach for pipelining sequences of data-dependent loops. In: FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 219–228. IEEE Computer Society, Washington, DC, USA (2007)
16. Menotti, R., Marques, E., Cardoso, J.M.P.: Aggressive loop pipelining for reconfigurable architectures. In: International Conference on Field Programmable Logic and Applications (FPL), pp. 501–502 (2007)

17. AT&T Research: Graphviz: Graph visualization software (2011). http://www.graphviz.org/
18. Aspvall, B., Plass, M.F., Tarjan, R.E.: A linear-time algorithm for testing the truth of certain quantified boolean formulas* 1. Inf. Process. Lett. **8**(3), 121–123 (1979)
19. Muchnick, S.: Advanced Compiler Design Implementation. Morgan Kaufmann, Los Altos, CA (1997)
20. Ramalingam, G.: On loops, dominators, and dominance frontiers. ACM Trans. Program. Lang. Syst. **24**(5), 455–490 (2002)
21. C-to-Verilog.com: C-to-Verilog (2009). http://c-to-verilog.com/
22. Buyukkurt, B., Guo, Z., Najjar, W.A.: Impact of loop unrolling on area, throughput and clock frequency in ROCCC: C to VHDL compiler for FPGAs. In: Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC2006) (2006)
23. Menotti, R., Cardoso, J.M.P., Fernandes, M.M., Marques, E.: On using LALP to map an audio encoder/decoder on FPGAs. In: Proceedings of the 2010 IEEE International Symposium on Industrial Electronics (2010). To be published
24. Austin, T., Larson, E., Ernst, D.: SimpleScalar: an infrastructure for computer system modeling. Computer **35**(2), 59–67 (2002)
25. SimpleScalar LLC: SimpleScalar (2011). http://www.simplescalar.com/
26. Altera Corporation: Nios II processor reference handbook (2011). http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf
27. Gokhale, M.B., Stone, J.M., Gomersall, E.: Co-synthesis to a hybrid RISC/FPGA architecture. J. VLSI Signal Process. Syst. **24**, 165–180 (2000)
28. Budiu, M., Goldstein, S.C.: Fast compilation for pipelined reconfigurable fabrics. In: FPGA '99: Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays, pp. 195–205. ACM, New York, NY, USA (1999)
29. Cronquist, D.C., Franklin, P., Berg, S.G., Ebeling, C.: Specifying and compiling applications for RaPiD. In: FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, pp. 116. IEEE Computer Society, Washington, DC, USA (1998)
30. Coutinho, J.G.F., Jiang, J., Luk, W.: Interleaving behavioral and cycle-accurate descriptions for reconfigurable hardware compilation. In: FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 245–254. IEEE Computer Society, Washington, DC, USA (2005)