FPGA Computing in a Data Parallel C

Maya Gokhale and Ron Minnich
Supercomputing Research Center
17100 Science Dr.
Bowie, MD 20715

Abstract

We demonstrate a new technique for automatically synthesizing digital logic from a high level algorithmic description in a data parallel language. The methodology has been implemented using the Splash 2 reconfigurable logic array for programs written in Data-parallel Bit-serial C (dbC). The translator generates a VHDL description of a SIMD processor array with one or more processors per Xilinx 4010 FPGA. The instruction set of each processor is customized to the dbC program being processed. In addition to the usual arithmetic operations, nearest neighbor communication, host-to-processor communication, and global reductions are supported.

1 Introduction

Field Programmable Gate Arrays (FPGAs) have been successfully demonstrated by various research groups ([GHK+91], [Tuo92], [ABD92], [Ath92]) as effective general purpose computing engines. With the exception of [Ath92], however, the FPGAs must be programmed at the logic block or gate level in order to make reasonable use of logic and routing resources on the chip.

In this work, we present a methodology to program an array of FPGA chips in a high level parallel C using a SIMD programming model. Using this methodology and the compiler tools developed for this process, it is possible to regard the array of FPGAs as an array of processing elements, where each chip contains one or more PEs. The instruction set of the PEs for a particular application is derived from the parallel C program embodying that application. The instructions are customized to the application; for example, rather than generating a general-purpose load or store instruction, the compiler generates a set of load and store instructions each one of which has the source and destination encoded into the instruction. This customization results in a simpler data path, smaller instructions, and in most cases instructions with no parameters. The host processor issues these customized instructions to the PE array, and may receive data back from individual processors or from the entire array.

The logic to be synthesized for the various components of this system is simple enough that commercially available synthesis tools can efficiently generate logic for FPGAs. In addition, the data parallel C has extensions which allow the programmer to specify bit lengths of parallel operands, enabling the programmer to express bit level computation and to conserve resources such as routing lines and logic blocks on the FPGA chips.

These ideas have been tested in the context of the Data-parallel Bit-serial C language (dbC) [SG92], which also runs on Sparcs, Crays, and the Terasys workstation [GHI+92]. A prototype translation system has been constructed to target the Splash 2 Xilinx FPGA array [ABD92] from a given dbC program.

2 The Splash 2 Array

The Splash 2 configuration is shown in Figure 1 (from [ABD92]), which illustrates a 3-board system. Each board contains 16 Xilinx 4010 FPGAs, X1–X16, and a controller, X0, also a 4010. Each chip has a 256K $\times$ 16 bit local memory. The 16 chips are connected as a linear systolic array with a 36-bit wide path. Arbitrary connectivity is also possible through the crossbar, which provides a 36-bit path among the 17 chips. X0 controls the crossbar.

The Splash boards can communicate with the Sun SparcStation II host via two input and two output FIFOs, which are on an additional interface board. The input FIFOs supply 36-bit data to X1 and the output FIFOs receive 32-bit data from X16. As shown in the figure, the bus from the input FIFOs also serves as a "SIMD" bus, by feeding the contents of the input FIFOs to X0. X0 can then configure the crossbar to

0-8186-3890-7/93 $03.00 © 1993 IEEE

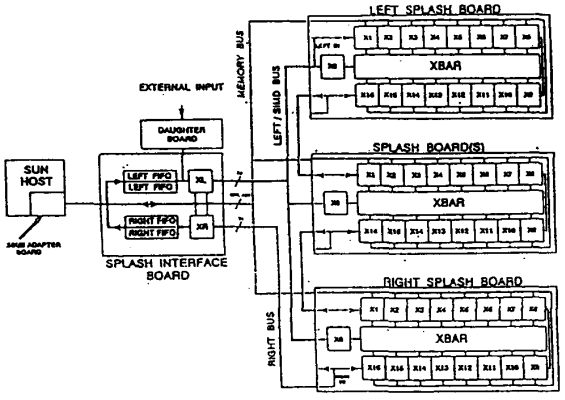

Figure 1: The Splash 2 System

broadcast instructions and data to X1–X16. There are several control lines for global action: there is a single interrupt line from each chip to the host. There are two signals GLOBAL AND/OR and GLOBAL VALID from each chip to X0 and from X0 back to the interface board.

3 dbC

dbC is an ANSI C superset similar to MPL and C*. The programming model is the familiar SIMD one in which a Front-End (FE) processor controls instruction sequencing of many Processing Elements (PEs). In dbC PE data, of which there are as many instances as there are PEs, is designated with a *poly* attribute in the declaration. FE data, of which there is only one copy, is declared in the normal C way. PE and FE data can be intermixed freely in the program.

dbC compiles to C code containing Generic SIMD instructions. Generic SIMD defines the instruction set of an abstract memory-to-memory SIMD machine; it look similar to the Paris instruction set [TMC89]. This SIMD code can be targeted to the CM-2, Terasys [GHI+92], the PASSWORK simulation system (which runs on Sun workstations and Crays), or Splash.

There are one, two, and three-address functions, some with a common bit-length for all variables, and some in which the bit length can be different for every variable.

Other functions support Processor Element (PE) to front-end (FE) communication. These include communication from the FE to an individual PE or to all PEs, and global reduction operations. Nearest neighbor communication and data routing (scatter/gather) are also supported.

In Generic SIMD, the data types are scalar types with a bit-length attribute; the operators are assign, arithmetic, reduction and communication operators.

4 Example

We will demonstrate the system and in particular the steps required to emit code for the Splash 2 array with a small example[1].

Starting from a dbC program, the steps are:

- Generation of intermediate code ("Generic SIMD")
- Enumeration of data types and operators on the types. One novel feature of this phase is that the operators are specific to the program being analyzed: for example, a 3-bit add of x and y is different than a 3-bit add of y and z or a 3-bit move of x to y. The architecture and instruction set are being customized on a per-program basis.
- Determination of variables, and data movement between variables. The data path, rather then being the generalized data path found in general-purpose computers, is customized on a per-program basis.
- Determination of the control structure, i.e. what decoders for instructions are needed and what those decoders must control. The decoders are also customized for the program.
- Establishment of inter-PE (and inter-chip) data paths for nearest neighbor communication.
- Establishment of inter-PE (and inter-chip) data paths for global reduction operations.
- Generation of:
  1. VHDL types for the data types
  2. VHDL "signals" for the variables
  3. VHDL control statements for the instruction decode

[1]This example was written in dbC by Neil Coletti of SRC and has been run on the CM-2 and the PASSWORK simulation system [IG92] as well as the Splash 2 simulator [Bue92] [Arn92].

4. appropriate VHDL assignment statements for each of the operators
5. Port declarations and interconnection to support nearest neighbor communication and global reduction

- Synthesis of VHDL to EDIF format
- Conversion of EDIF to XNF
- Running the PPR tool on the XNF to produce LCA-format
- Conversion of LCA format to the final bitstream for programming the chip

Of course, in the final version, the only part of this process visible to the user is that the user must use the dbC compiler with a switch to indicate that Splash is the target, not one of the other dbC target systems (e.g. CM-2). The hardware path to Splash is signified by a switch to the same dbC compiler used for all the other targets. At the present time, each tool must be invoked separately by the user.

4.1 dbC Program

We show in Figure 2 a dbC program to compute the cross correlation of two bit streams. The program compares two bitstreams and accumulates a count of the number of times an individual bit in the bitstreams had the same value. The bitstreams are compared with a delay of zero bits, then with successively larger delays, usually one bit longer for each delay. For each of the delays a counter records the number of matches (see Figure 3). The delay is sometimes called a "lag".

In a typical implementation, there are individual cells which perform the correlation between two streams of data for one value of the delay, i.e. there is a cell for delay 0, delay 1, and so on. Each cell includes a comparator and a counter. The comparator compares the data in the bitstreams and if they are the same the counter is incremented. Comparison of the two bits can be performed with exclusive-or or exclusive-nor. At least one of the bitstream data streams moves from cell to cell, but the counter data is stationary, as shown in figure 3.

In this example stream a is sent from the host, while stream b is duplicated on each PE. Stream b could be sent from the host as well, or could be stored in the local memories. The counter R accumulates the result of the correlation.

The user specifies the size and shape of the processor array by initializing the pre-defined variables

```
#include <interproc.hd>
typedef poly unsigned Boolean:1;
Boolean a, temp_b ;
#define N 32
#define NPROC 16
unsigned DBC_net = 1;
unsigned DBC_net_shape [1] = {NPROC};
int right[1] = {1};
poly unsigned int R:16 = 0;
void main()
{
  all{
    int i;
    a = 0;
    for (i=0; i < N; i++) {
      temp_b = i;
      DBC_write_to_proc(&a, 1, 0);
      R += a ^ temp_b;
      DBC_net_send(&a, a, right);
    }
    printf ("%d\n",
    DBC_read_from_proc(R, 0));
  }
}
```

Figure 2: dbC Cross Correlation Program

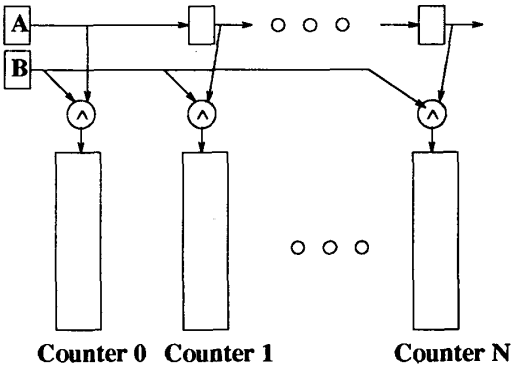

Figure 3: Correlation of two bit streams

DBC_net, which gives the number of dimensions, and DBC_net_shape, which gives the rank of each dimension.

The keyword **poly** indicates the declaration of parallel variables or data types. Integers and logicals in the parallel domain may have arbitrary user-defined bit length. In the example program, the variables a and temp_b are one bit parallel variables, and the counter R, which will hold the result, is a 16-bit unsigned integer.

The all keyword indicates that all processors are to participate in the body of the compound statement. The body of the all-block contains an initialization of the parallel variable a, a sequential for statement, and a final print statement. Within the for-loop, there are four statements. The first assigns to each processor's temp_b the current value of i. The second statement causes host-to-processor communication: the host writes a "1" to processor 0's a. The third statement updates the counter R. On each processor, the result of a "xor'ed" with temp_b is added into R. Finally, each processor in the linear array shifts its value of a to the right. The loop repeats for 32 iterations. The final statement of the program simply reads and prints the value of the counter R from processor 0. Note that the value of register R could be read from any PE, even as the program is running. This type of register examination has traditionally been very difficult for programmers to design into Splash programs.

4.2 Generic SIMD Code

A fragment of the Generic SIMD code for the program is shown in Figure 4. Each "function" call prefaced by opPar is a generic SIMD instruction.

The MoveZero instruction clears a single bit of a. Movec moves the "constant" i to b for 1 bit. Write_to_proc writes a 1 into processor 0's a for 1 bit. The Bxor3 instruction performs a boolean xor of a and temp_b into a compiler-generated temp t2 for 1 bit. Next the 1-bit temp is moved to a 16-bit temp t3, which is then added into R (Add2 instruction). The net send shifts one bit of a right. Finally the Read_from_proc reads R from processor 0.

4.3 Generating VHDL

In the second phase of compilation to Splash, the generic SIMD code is extracted from the generated C program, mechanically re-formatted, and then input to a Lisp-based translator which emits VHDL code. This behavioral code consists of an entity for a single

```
opParMoveZero_1L(a.address, 1);
for (i = 0; i < 32; i ++ ) {
  opParMovec_1L(temp_b.address, i, 1);
  DBC_write_to_proc(a.address, 1, 0, 1);
  opParBxor3_1L(opParAddOffset(
                _DBC_poly_frame_t, 1) /* t2:1:1 */,
                a.address, temp_b.address, 1);
  opParMove_2L(opParAddOffset(
               _DBC_poly_frame_t, 2) /* t3:16:2 */,
               opParAddOffset(
                 _DBC_poly_frame_t, 1) /* t2:1:1 */,
               16, 1);
  opParAdd2_1L(R.address,
               opParAddOffset(
                 DBC_poly_frame_t, 2) /* t3:16:2 */,
               16);
  DBC_net_send(a.address, a, right, 1); } /* end for */
printf(
"%d\n", DBC_read_from_proc(R, 0));
}
```

Figure 4: Generated C code for Correlation

FPGA chip (which is used for Splash array chips 1–16) as well as a VHDL entity for the Splash controller chip X0. The array chips are used as PEs, while X0 is used for FE to PE communications and control.

The instruction set for this SIMD engine consists of the instructions extracted from the generated C program and customized to the specific instruction instance. For example, the boolean xor instruction synthesized expects the operands to be locations a and temp_b and the result to go into t2. Thus there is no need for run-time computation of source and destination; a data path to compute and gain access to arbitrary source and destination; or much of the other complexity that comes with a general-purpose instruction set. Figure 5 shows a fragment of the decoder generated for the correlation program.

The program has 13 instructions (of which 2 are shown in the figure) and a "noop". The local controller generated for this program is instantiated on each chip. It takes the top four bits 36–32 from the crossbar as the instruction number, and produces a bit vector of signals to the processing elements.

The program for each PE is shown in Figure 6. As the port map illustrates, the translator must generate a path for the value of a to be shifted out of each PE and into the next one. "Constants" broadcast from the host come into the Broadcast_Value port, and output from a PE goes out the Output_Value port.

In terms of VHDL the program is an event-driven

```
entity SIMD'PE'CONTROL is
  Port(
    clk                  : in  Bit;
    Instruction          : in Natural range 0 to 13;
    Decoded'Instruction :
                  out Bit'Vector (13 downto 0);
    Enable               :out Bit'Vector (4 downto 0) );
end SIMD'PE'CONTROL;

architecture Behavior of SIMD'PE'CONTROL is


begin

Control'Process: process
  begin
    wait until clk'event and clk = '1';

    Enable <= "00000";

- #<opPar :OPPARNOOP"-always" NIL NIL NIL NIL>
    if Instruction = 0 then
      Decoded'Instruction(0) <= '1';
      else Decoded'Instruction(0) <= '0';
    end if;

- #<opPar :OPPARWRITETOPROC #<Poly-Address "a" 0> 1 1 1 0>
    if Instruction = 1 then
      Decoded'Instruction(1) <= '1';
      else Decoded'Instruction(1) <= '0';
    end if;

- #<opPar :OPPARMOVE-1L"-always" #<Poly-Address "t1" 0> 1
      #<Poly-Address "Context" 0> 1>
    if Instruction = 2 then
      Decoded'Instruction(2) <= '1';
      else Decoded'Instruction(2) <= '0';
    end if;

 ...

  end process;
end Behavior;
```

Figure 5: SIMD Decoder

loop. It waits for a rising clock edge, then decodes the instruction (the *if* statements) and takes the action indicated by the opcode. Note that the increment of the counter is indicated by the statement

```
R <= R + t3
```

. The variable t3 is the result of the comparison in a previous clock cycle.

The program for an entire chip consists simply of a single controller and a programmer-controlled number of PE's. The top 4 bits of input from the crossbar go to the decoder, and the bottom 32 bits go to each PE. Output from a PE (in response to a "read from proc" opcode, for example) is sent out on the crossbar. In the chip level entity description, the nearest neighbor data path carrying a between PEs is instantiated. The leftmost PE receives its a from the left linear path, and the rightmost PE sends its a out the right linear path.

Finally, the C program generated from the user's dbC, which runs on the Sparc host (see Figure 4), must be modified: the "opPar" calls must be replaced by new Splash macros which write the new opcode (for example, opcode 13 for the accumulation into R) along with data, if required for that opcode, to the SIMD bus. The X0 program broadcasts input from the SIMD bus to chips 1–16 over the crossbar, and

```
entity SIMD'PE is
  Port(
    PORT'IN'a            : IN UNSIGNED(0 downto 0);
    PORT'OUT'a           : OUT UNSIGNED(0 downto 0);
    clk                  : in  Bit;
    Broadcast'Value      : in UNSIGNED (31 downto 0);
    Output'Value         : out RBit3'Vector (15 downto 0);
    Decoded'Instruction : in Bit'Vector (13 downto 0)
  );
end SIMD'PE;

architecture Behavior of SIMD'PE is

    signal Context       : Bit'Vector(0 downto 0);
    signal Local'Broadcast'Value: UNSIGNED(31 downto 0);
    signal t1            : UNSIGNED(0 downto 0);
    signal t2            : UNSIGNED(0 downto 0);
    signal t3            : UNSIGNED(15 downto 0);
    signal temp'b        : UNSIGNED(0 downto 0);
    signal a             : UNSIGNED(0 downto 0);
    signal R             : UNSIGNED(15 downto 0);
    signal DBC'iproc     : UNSIGNED(31 downto 0);

begin

Simd'Process: process
  begin
    Local'Broadcast'value <= CONV'UNSIGNED(Broadcast'value, 32);
    wait until clk'event and clk = '1';
    Output'Value <= "ZZZZZZZZZZZZZZZZ";
    if Decoded'Instruction(0) = '1'then
      - Noop;
    end if;
    if Decoded'Instruction(1) = '1'then
      if DBC'iproc(31 downto 0) = CONV'UNSIGNED(0, 32) then
        a(0 downto 0) <= CONV'UNSIGNED(1, 1); end if;
    end if;
    if Decoded'Instruction(2) = '1'then
      t1(0 downto 0) <= Context(0 downto 0);
    end if;
    if Decoded'Instruction(3) = '1'then
      Context(0 downto 0) <= CONV'UNSIGNED(1, 1);
    end if;
    if Decoded'Instruction(4) = '1'then
      Context(0 downto 0) <= t1(0 downto 0);
    end if;
    if Decoded'Instruction(5) = '1'then
      if DBC'iproc(31 downto 0) = Local'broadcast'value(31 downto 16) then
        Output'Value <= BVtoRB3V(R(15 downto 0)); end if;
    end if;
    if Decoded'Instruction(6) = '1'and Context(0) = '1' then
      a(0 downto 0)<=PORT'IN'a(0 downto 0);
    end if;
    if Decoded'Instruction(7) = '1'and Context(0) = '1' then
      PORT'OUT'a(0 downto 0)<=a(0 downto 0);
    end if;
    if Decoded'Instruction(8) = '1'and Context(0) = '1' then
      temp'b(0 downto 0) <= CONV'UNSIGNED(CONV'INTEGER
          (Local'Broadcast'Value(15 downto 0)), 1);
    end if;
    if Decoded'Instruction(9) = '1'and Context(0) = '1' then
      R(15 downto 0) <= CONV'UNSIGNED(0, 16);
    end if;
    if Decoded'Instruction(10) = '1'and Context(0) = '1' then
      t3(15 downto 0) <= CONV'UNSIGNED(0, 15) & t2(0 downto 0);
    end if;
    if Decoded'Instruction(11) = '1'then
      a(0 downto 0) <= CONV'UNSIGNED(0, 1);
    end if;
    if Decoded'Instruction(12) = '1'and Context(0) = '1' then
      t2(0 downto 0) <= a(0 downto 0) xor temp'b(0 downto 0);
    end if;
    if Decoded'Instruction(13) = '1'and Context(0) = '1' then
      R(15 downto 0) <= R(15 downto 0)+t3(15 downto 0);
    end if;
  end process;
end Behavior;
```

Figure 6: A Single PE to do Correlation

handles any reconfiguring of the crossbar which might be needed for certain opcodes.

5 Synthesis

To evaluate the efficiency of the generated VHDL code, we have synthesized Xilinx-specific chip configurations from the generated behavioral chip programs.

For this example it should be possible to hand-pack 20 PEs per chip. The dbC-derived program can pack 13 PEs per chip. Thus the hand-packed version is 50% more efficient. An examination of the costs for the dbC-based version show that the additional space is taken up by the SIMD control logic and the logic needed to handle the read and write to individual variables in the PEs. Also, as discussed below there are also some problems with VHDL synthesis which require unneeded connections to be installed so that the synthesis tools will not delete the logic. This makes our process seem less efficient that it really is; once the synthesis tools are fixed we expect to be more efficient.

In the hand-coded version, the PEs do not have the ability to examine the registers while the program is running; the chip would require re-design for this capability, and the number of PEs per chip would be fewer. Thus we are trading off a degree of efficiency for a capability which did not exist before. In earlier Splash designs users tried to add the capability to read and write variables that could be accessed while the chip was running; such designs proved to be very difficult for individuals to manage, and changing which variables were accessed involved a major redesign. In the dbC-based program, changing which variables are accessed is as simple as adding a statement. The variables may be accessed via dbC functions such as printf or via examination from the debugger.

For a different cross-correlation design the hand-packed numbers are 16 PEs, while the dbC numbers are 11 PEs per chip. The differences in the number of PEs could again be accounted for by the SIMD control logic and the variable read and write logic, as well as covering for problems with VHDL synthesis.

In an early version of the second cross-correlation design a register was added between PE stages to ensure that the setup times for a destination register were met. It should prove to be much simpler to issue the instruction to set up the source register, issue a NOP to allow for the propagation time to the destination register, and then issue the instruction to load the destination register. The cost in time is the same, and the change reduces the cost in space of the design. This addition of a wait state is common to hardware designs, although usually accomplished by an additional state in a hardware state machine.

5.1 "Training" the user

It has been said that automatic vectorizers have succeeded not so much at vectorizing dusty deck codes as at training users to write vectorizable code. Similarly, to get good performance from the complex tools being used in the dbC-to-Splash translation process, it is desirable to express the computation so that the tools can generate efficient code. The following example demonstrates this point.

We first coded the accumulation step of the correlation as

```
R += a ^ temp_b;
```

as shown in figure 2. This generated a 16 bit temp t3 which was added to R (Figure 4). The synthesis tool built a full 16-bit register for t3, and 15 full-adders and one half-adder simply to add t3 to R. Since t3 was guaranteed to only ever be one bit, this was very inefficient. We re-coded the dbC as follows:

```
temp_b = a ^ temp_b;
R = R + temp_b;
```

This eliminated the 16-bit temp, and the Add2_1L was replaced with an Add3_3L. The 16-bit adder was no longer generated, and we were able to increase from 10 PEs per chip to 13 PEs per chip, 30% more efficient packing.

5.2 Over-Optimization

As originally written, the algorithm did not require the counter values to be returned to the host. On XILINX chips it is possible to dump the entire state of the chip at any time. Thus, the counter values can be recovered even though outputs of the registers holding R are not connected; the value of the data in those registers is still completely accessible via the chip state dump. We depend on VHDL optimization to efficiently map from the dbC-generated VHDL to a chip. However, VHDL compilers attempt as part of the optimization to eliminate those registers that have outputs not connected to any other inputs. Since this is the case with R, that register is (erroneously) omitted by the VHDL synthesizer. Unfortunately, the effect cascades: since R is gone, there is no need to accumulate into R, ... In fact, the VHDL synthesis tool optimized the entire chip out of existence because

this one variable R had no inputs connected to its outputs through a path that the tool understood.

There is a DONT_TOUCH_NETWORK attribute that can be defined for nets, which indicates to the VHDL compiler that regardless of the optimizations it may be doing, it should not eliminate nets having the attribute. However, this did not work in the version of the synthesis tools we were using.

To circumvent this problem, we changed the algorithm to read back the counter value from PE 0. This is costly: it results in an additional 16-bit register with tri-state outputs for each PE, and therefore a lower than necessary number of PEs per chip.

6 Future Work

The tools described in this paper are still being refined. Although enough has been implemented to demonstrate that the approach is viable, much work remains.

6.1 Read from Proc

One question is how to properly design the read-from-proc capability. In the VHDL program shown, Output_Value is a tri-state register. We have not found a way to get our VHDL synthesis tools to use the Xilinx on-chip tri-state hardware without adding an extraneous register. Also, it may be more effective to have a selector external to the PE, but still on the chip, and have the PE drive the output value all the time. The actual selection would be done by a tree of multiplexers.

Another problem with read-from-proc is that there are not enough crossbar configurations to route the output of any chip to chip 16. There are enough for chips 1–14. PEs on Chip 15 must send via the systolic data path. PEs on Chip 16 can send directly to the FIFO. This complicates generation of the VHDL program: if read-from-proc is used in a dbC program, we need three different chip programs, one for Chips 1–14, one for Chip 15, and one for Chip 16, rather than the single program as shown in section 4.3.

6.2 Local Memory

Our translator does not currently allocate variables in the local memory associated with each chip. As implementation progresses, we will use the local memory for parallel arrays. There are several degrees of freedom in how the memory is to be organized. If 16 PEs can be placed on a chip, it is advantageous to allocate each bit of a 16-bit word to a different PE, and to assemble the operands bit serially. If there are few (4) PEs per chip, then a word-oriented organization is preferable.

6.3 Loops on X0

In the current model, instructions are issued from the Sparc host onto the SIMD bus. X0 reads the SIMD bus and broadcasts to X1–X16. It would be desirable to migrate the control of inner loops to X0, returning control to the host program only when the loop finishes.

7 Summary

We have demonstrated a methodology and a translator which can be used to synthesize custom FPGA programs. dbC, a data-parallel C similar to MPL and C*, has been used to program a variety of SIMD, vector-parallel, and general-purpose computers. We have demonstrated that dbC can also target hardware on the Splash 2 system. A number of issues remain, such as the interaction of dbC and VHDL optimization, optimizing communication, and the use of the Splash 2 local memory.

References

[ABD92] J. Arnold, D. Buell, and E. Davis. Splash 2. *Proceedings, 4th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '92)*, pages 316–322, 1992.

[Arn92] J. Arnold. Splash 2 Programmer's Manual. Technical report, Supercomputing Research Center, December 1992.

[Ath92] P. Athanas. An adaptive machine architecture and compiler for dynamic processor reconfiguration. Technical Report LEMS-101, Brown University, 1992.

[Bue92] D. Buell. A Splash 2 Tutorial, version 1.1. Technical Report 92-087, Supercomputing Research Center, December 1992.

[GHI+92] Maya Gokhale, Bill Holmes, Ken Iobst, Alan Murray, and Tom Turnbull. A massively parallel processor-in-memory array

and its programming environment. Technical Report TR-92-076, Supercomputing Research Center, 1992.

[GHK+91] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building and using a highly parallel programmable logic array. *IEEE Computer*, pages 81–89, January 1991.

[IT92] K. Iobst and T. Turnbull. PASSWORK User's Guide. Technical Report 90-014, Supercomputing Research Center, February 1992.

[SG92] J. Schlesinger and M. Gokhale. Dbc reference manual. Technical Report 92-068, Supercomputing Research Center, October 1992.

[TMC89] Paris reference manual. Technical Report Version 5.0, Thinking Machines Corporation, February 1989.

[Tuo92] H. Tuoati. Perle1dc: a C++ library for the simulation and generation of DecPeRLe-1 designs. Technical Report 4, Digital Paris Research Laboratory, November 1992.