PAM Programming Environments: Practice and Experience

Patrice Bertin
Hervé Touati
Digital Equipment Corporation
Paris Research Laboratory
85, avenue Victor Hugo
92500 Rueil-Malmaison, France

Abstract

Digital Equipment's Paris Research Laboratory started to investigate the use of FPGA-based reconfigurable hardware accelerators in 1988. We call them PAMs *for Programmable Active Memories. Over the past six years, we have designed and implemented four generations of PAM hardware and four generations of PAM programming environments. Several dozen people, ranging from inexperienced students to senior hardware designers, have used our systems [3]. A wide range of applications belonging to several important application domains have demonstrated the interest of these novel computing devices. In this paper we present the software lessons we draw from this collective experience.*

1 Introduction

In 1988, under the direction of Jean Vuillemin, Digital Equipment Corporation's Paris Research Laboratory (PRL) started to investigate the use of SRAM-based FPGAs for building reconfigurable hardware coprocessors. We call these coprocessors Programmable Active Memories or PAMs [2, 1]. PRL developed four generations of PAMs and four generations of PAM programming environments. We gathered the most experience with DECPeRLe-1, our third generation PAM board, and Perle1DC, its programming environment.

24 DECPeRLe-1 boards were built, 7 of which have been installed in collaborating research sites around the world. The following data summarizes the most significant applications that were developed in 1993 for DECPeRLe-1 using the Perle1DC programming environment[1]. Other work under way includes a large Viterbi decoder for deep-space probe data reception (Macquarie University, Sidney, Australia) and the second-level trigger of a deep-underwater neutrino telescope in the Aegean Sea (CERN, Geneva).

- A stereo vision system, used to compute *range maps* from simultaneous images obtained by two cameras. A software implementation of this system computes the correlation of a pair of images in 59 seconds on a SPARCstation II. A specialized board with four DSPs performs the same task in 9.6 seconds. The same task on DECPeRLe-1 runs in 0.28 seconds, over thirty times faster.
- A 3-D Heat Equation solver. Running at 25 MHz, it computes 40 billion operations (16 bit adds and shifts) per second.
- A digital audio 256-channel professional quality sound synthesizer, using additive synthesis and frequency modulation. An optimized DSP implementation of the same algorithm, running on a Motorola 56001 at 40 MHz, can only achieve 25 channels without the frequency modulation feature.
- For the second-level triggers of its planned Large Hadron Collider, CERN published several benchmarks algorithms. We studied and implemented on DECPeRLe-1 one of these benchmarks, the *hadron calorimeter* [4]. Besides being the most cost-effective, the solution provided by PAMs is today the *only* one that CERN could identify as meeting the bandwidth and latency requirements.

In this paper we present the lessons we draw from the development and use of Perle1DC, our current generation PAM programming environment. Some of the ideas and algorithms presented here are well known. Our main goal is to show how a few simple ideas put together can lead to a design environment which is both low-cost and adequately productive.

The rest of the paper is organized as follows. In sections 2 and 3, we give a short presentation of DECPeRLe-1 and its requirements in terms of programming environments. In section 4, we discuss the main design decisions we made when developing Perle1DC. In section 5, we present what worked well. In section 6, we present what did not work so well and what we would do differently today. In section 7 we present our future plans. Section 8 summarizes the main points of the paper.

[1] For performance comparisons, one should keep in mind that DECPeRLe-1 was built using 1991 state-of-the-art digital electronics technology.

2 Background

2.1 PAMs

A PAM (Programmable Active Memory) is a reconfigurable hardware coprocessor tightly coupled to a host workstation. PAMs can have different shapes and sizes; our third generation board, DECPeRLe-1, is made out of 24 Xilinx XC3090 chips, 4 MB of static RAM, and a fast FIFO-based interface to the I/O bus of a workstation. For this we use Digital Equipment Corp. TURBOchannel bus protocol, which provides up to 100MB/s of bandwidth between an I/O device and the host's memory.

A PAM *program* comprises two interacting parts: *a sequential circuit description part*, which is compiled into FPGA configuration bitstreams; and *a software description part*, which is compiled into a piece of software to be run on the host workstation.

2.2 When to Use a PAM?

The main reason to go into the trouble of a PAM implementation of a given algorithm is to get performance unattainable in software. If the same performance can be obtained by executing a straightforward C program on a workstation, it makes little sense to use a PAM. To date, most of our applications have had a combination of the following characteristics:

- high computing power requirements (Heat equation solver, Viterbi decoder, sound synthesizer);
- high I/O bandwidth requirements (CERN calorimeter);
- real-time constraints (CERN calorimeter; stereoscopic vision system; sound synthesizer);
- data formats for which standard CPU hardware is not optimized (none of these applications use IEEE 64-bit floating-point computations);
- hardware interface capability (digital camera interface).

In all of these DECPeRLe-1 applications but the last, performance was critical, and heavy pipelining combined with careful low-level design optimization was the key to the solution.

3 Requirements

To obtain adequate performance with the available PAM hardware, the implementation has to be done with some minimum level of care. In a PAM program sloppiness bites twice: once in circuit density and device utilization (space) and once in circuit speed (time). As a consequence, one can easily get a factor of four or more slow down by a careless implementation of a good algorithm. Since performance is so critical to the economical viability of PAMs, high-level programming from behavioral descriptions is not going to be a practical approach; at least not until we reach the point where we can afford to throw away a factor of four in silicon resources (one has to keep in mind that we already pay a performance penalty by using SRAM-based FPGAs instead of raw silicon). Programmable interconnects are also slower and less dense than their hardwired counterparts. As a consequence, complex designs cannot be placed and routed automatically unless they do not attempt to reach high device utilization; even then, the resulting circuitry is significantly slower than what can be achieved with good hand-designed geometry.

Careful low-level circuit implementation has always been possible through a painful and laborious process: schematic capture. For PAM programming, schematic capture is not a viable alternative. Though it can provide the best performance, it is too labor intensive for large PAM designs.

Given these constraints, we have but one choice: a middle-ground approach where designs are described at the structural level and the structure can be annotated with geometric information to guide the physical design. More precisely, here is a set of properties that we think a PAM programming environmnent should have:

- The hardware description should be algorithmic. The size of the applications we develop precludes the use of schematic capture.
- It should be possible to compile the hardware description without any manual intervention at any intermediate stage of the compilation process. This requirement implies that the source program should contain mechanisms to annotate the hardware description with whatever placement or resource allocation directives needed to obtain the desired implementation quality.
- The level of abstraction provided by the programming environment should be the highest abstraction level that can be shared by most PAM applications, but not any higher. Our goal is to explore many different application domains; specialized high-level graphic entry systems are thus premature. General-purpose high-level graphic entry systems do not work.
- Mixed-mode (behavioral/gate-level) block level to system level simulation should be supported. Even though SRAM-based FPGA technology allows for development by trial and error, simulation remains a crucial development tool: mixed-mode simulation cannot be performed on FPGAs; gate-level simulation runs several orders of magnitude faster on FPGAs than in software, but compilation times are several orders of magnitude worse than for software. Thus both simulation techniques are needed: short simulation runs are better performed in software, long execution traces are better obtained from hardware.

4 Design Decisions

4.1 Choice of a Programming Language

Three choices are possible: a general-purpose programming language such as C++, a hardware description language such as Verilog or VHDL, or our own language. We have discarded so far the latter alternative, though we are still divided on the issue of whether the world has still room for a new hardware description language targeted at synchronous logic in general and FPGAs in particular. We are nevertheless continuing research in this direction.

The use of a complex hardware description language such as VHDL has several advantages: we can build on commercially available software packages; we can build upon the accumulated experience of hardware designers trained with the language. However there are several drawbacks with this approach for PAMs: we give away low-level control on circuit geometry, which is required for speed and density by most of our applications; we lose any chance to convert software engineers to PAM programming; there are much fewer VHDL programmers than C++ programmers.

The use of a general-purpose programming language does not suffer from these problems. First the solution is cheaper: we are tapping a much wider market in terms of training, documentation and programming tools. Though we need to develop a generic software library to handle netlists and simulation, the amount of work is limited (6 man-months). Moreover we keep full control on the generated result, and we can include circuit geometry information as desired. The training investment is limited, as most engineers, hardware and software alike, have had some exposure to C or C++.

4.2 Synchronous Circuit Entry

We decided on the use of C++ mainly because of its widespread use and its malleability. To describe synchronous circuits with C++ is straightforward: we just need to introduce a new type `Net`, overload the C++ Boolean operators to describe combinational logic, and add a primitive for the register initialized to zero. Using these simple constructs, a C++ program can be used to *generate* a netlist representing any synchronous circuit. This type of low-level description is made convenient by the use of basic programming constructs such as arrays and records, `for` loops and procedures, for which no special training is necessary.

The execution of such a program builds a netlist in memory; this netlist can then be analyzed and translated into an appropriate format (XNF or EDIF), or used for simulation. Linking a netlist description program with behavioral code (also written in C++) yields mixed-mode simulation with no special effort. To give an example of the ease of use of this description, here is the code representing a generic n-bit ripple carry adder:

```
template<int N>
RippleAdder: public Block {
  public:
    RippleAdder(): Block("RippleAdder"){}
    void logic(Net<N>& a, Net<N>& b, Net<N>& c,
               Net<N>& sum, Net& carry) {
      input(a);
      input(b);
      input(c);
      output(sum);
      output(carry);
      for (int i = 0; i < N; i++){
        sum[i] = a[i] ^ b[i] ^ c[i];
        carry[i] = (a[i] & b[i])
                 | (b[i] & c[i])
                 | (c[i] & a[i]);
      }
    }
};
```

4.3 Description of Geometry

At this level of description, we can easily annotate individual nets with placement or routing directives. For example to specify that our ripple-carry adder should be aligned vertically, with the corresponding carry and sum bits generated by the same logic block, we would simply add the following lines to the description of the adder:

```
void placement(Net<N>& sum, Net<N>& carry) {
  for (int i = 0; i < N; i++) {
    carry[i] <= sum[i];
    sum[i+1] <= sum[i] + OFFSET(0,1);
  }
}
```

Contrary to the silicon compilers of a decade ago [5], these placement annotations have no influence on the logic behavior of the generated netlist. They do not specify contacts; they only specify layout directives. More precisely, the placement annotations shown here play two roles:

- they specify the partitioning of logic into physical blocks;
- they specify absolute or relative placement of physical blocks on a discrete two-dimensional grid.

No assumption is made at this stage on the size of the physical blocks nor the size of the two-dimensional grid. PERLE1DC, the C++ software library that generates the annotated netlist, does not contain any information related to any particular FPGA family, and does not need to. Its only job is to perform consistency checks and solve placement constraints by propagation. Obviously, the programmer has to know some basic characteristics of the FPGA family being used. PERLE1DC is device-independent, the logic is device-independent, but the placement and logic partitioning annotations are not.

4.4 Mapping to FPGAs

The mapping of the logic and geometry is performed by a back-end tool which emulates the interface of a schematic capture software in order to guarantee that the placement and logic partitioning information is preserved by the FPGA vendor software. Logic that is not explicitly partitioned or placed by the user is passed unchanged to the FPGA vendor software for automatic partitioning and placement. In any case, the routing is left to the commercial tool.

4.5 When the Abstraction Breaks

In an ideal world, a PAM would be viewed as a large FPGA, and PAM programming would only involve describing the logic and annotating it with abstract geometric constraints, providing logic partitioning and placement both at the chip level and the logic block level.

Unfortunately, chip-to-chip interconnect is a scarce resource, and logically contiguous logic blocks on a PAM do not have the same interconnection delays when connected through a direct on-chip route or an inter-chip PCB trace, which sometimes makes it necessary to have precise control on which net gets connected to which I/O pin. Another concession to hardware reality was to provide direct support for bidirectional (tri-state) buses. These design decisions compromise the elegance and simplicity of the 2-D synchronous circuit programming model, but are absolute practical requirements.

4.6 System Level Tools

At the system level, the programming environment provides two main functions: a device driver interface, and full simulation support of that interface. The latter lets the designer simulate together the hardware and software parts of a PAM program. The former provides a few essential controls to the application program:

- the usual UNIX I/O interface, with open, close, read and write. Reads and writes can be either synchronous or asynchronous, and use programmed I/O or DMA transfers;
- download of the configuration bitstreams for the PAM FPGAs, and readback of their state (the values of all the flip-flops of all the chips);
- initialization and readback of the PAM memory banks;
- software control of the PAM board clock. Not only the speed of the clock can be set very precisely in software, but also its mode of operation: the clock can operate at full speed, in single step or in double step mode. In single or double step modes, readback can be activated to get an execution trace of the system. In double step mode, the clock runs at full speed every other cycle: this mode provides a very powerful way of isolating critical paths under real execution conditions.

4.7 Visualization Tools

Our emphasis on textual entry created the need for visualization tools. Fortunately, it is easier to display graphics than to make them editable, so we were able to develop several useful visualization tools with a moderate amount of effort. The simplest of these tools can read a netlist generated by PERLE1DC and display it in a hierarchical and simplified view of a PAM board. Net names and logic equations are displayed as they appear in the source program. This simple tool, developed using a public domain library for structured graphics, has been very helpful for finding errors in partitioning and placement directives.

Two visualisation tools can be used to browse through simulation or execution traces. The first of these tools, KRONO, developed at Digital Western Research Laboratory by Louis Monier and Joel Bartlett, is a simulation trace browser that operates at the level of the source hardware description, and understands the design hierarchy. The second of these tools, SHOWRB, developed by Mark Shand at PRL, operates on readback traces. It can display the state of every flip-flop in every FPGA of DECPERLE-1 at the rate of tens of frames per second. In conjunction with the double-step mode, it can be used to detect critical paths along execution traces.

4.8 Related Work

The idea of using a general-purpose programming language to describe circuits is not new [6]. Our placement constraints are strongly reminiscent of work done in module generators for custom VLSI [5]. The main difference here is that the behavior is entirely described by logic equations; placement directives have no influence on the logic.

5 What Worked

In this section, we present the features of our programming environment that we feel were successful. These features are sorted in decreasing order of importance.

- *Let users write module generators:* though difficult to write for custom VLSI, module generators are straightforward to write for the purely synchronous world of FPGAs, given the layout directives that our environment provides. Non-trivial module generators can be written by undergraduate students in a matter of a few days. This capability is the main reason why we could see untrained engineers or students develop complex applications spanning dozens of chips in a matter of a few months.
- *The attributed netlist abstraction:* it is the right way to handle the complexity of the underlying hardware and the need for user directives in a clean, uniform and extensible fashion. The logic behavior of the design is entirely contained in the netlist. Mapping the netlist to other technologies (e.g. ASIC) is therefore straightforward. Attributes need only be understood by certain back-end tools and can safely be ignored by all the others.

- *Simulation is useful:* mixed-mode simulation allows designers to certify datapath components before using them in complex designs. Full-system simulation eliminate the need for generating special input patterns to test the hardware part of the program. Full-system simulation also allows the simultaneous debugging of the application driver and the hardware description part.

- *Deterministic output is useful:* we went into extremes to make sure that the netlist generated by the design library was essentially insensitive to the ordering of equations in the source. That was beneficial in three ways: it helped making the compilation process more incremental, by helping detecting unnecessary runs of the place and route software; it made it possible to use simpler regression tests; it finally helped the routing process as it tended to lay out regular structures regularly.

The use of C++ as base programming language was also very handy to ensure upward compatibility, each time we had to change the application programmer's interface to the design library.

6 What Did Not Work

In this section we discuss design decisions that we feel now were inappropriate.

- The actual low-level partitioning of logic into FPGA cells under user directives should be done by FPGA vendor software without requiring the cumbersome interface of schematic capture tools.
- Simulation is never fast enough.
- The idea of using a generic intermediate format based on attributed netlists should have been pushed one step further. Not only should the intermediate format contain information passed to layout tools, such as physical placement or timing constraints, but it should also contain information relative to the source program, such as the file name and line number of the logical equation defining a net. Back-end debugging tools can use this information to point interactive editors to the incorrect statements in the source program.
- I/O blocks were handled in a device specific, ad hoc manner. They should be handled just like the rest of the logic: as a netlist with attributes to indicate physical placement.
- Tristate buffers should be handled as basic netlist primitives, even if that complicates the semantics of the netlist by creating two kinds of nets.
- The design hierarchy and net structures should be kept, as well as all net name aliases. Choosing among aliases the name with the largest visibility (the highest in the hierarchy) is a simple but acceptable alternative.

7 Future Work

In this section, we present areas where we think the most software work is needed.

To get high performance out of our designs, we frequently use an heavily pipelined design style. In some designs, the length of the pipeline exceeds 100 clock cycles. Controlling long and complex pipelines is difficult, and we do not have any tool to automate this task. We tried unsuccessfully to perform geometry-preserving, automatic retiming on Xilinx XC3090 devices [7]. For retiming to work, one must able to remove flip-flops from some of the outputs of a gate, but not all. With Xilinx XC3000 devices, this requires the use of the two logic block outputs, with the effect of reducing by up to a factor of two the density of the logic. The Xilinx XC4000 family does not suffer from this limitation, so automatic retiming could be made to work there.

Automatic logic partitioning among chips would also be useful. We do not believe in fully automatic tools being immediately usable for large boards; but for small homogeneous boards, or for large boards under some programmer assistance, better automation should be possible. Due to the delays incurred by going through inter-chip interconnect, logic partitioning should be able to insert retiming registers when required. Finally, any FPGA placement program should be able to handle floorplanning tasks, i.e. should accept relative placement constraints as imput. This capability is crucial to allow the development of module libraries.

8 Conclusion

We presented a set of simple, low cost tools with which we have demonstrated complex high performance PAM designs in a matter of a few months. We hope to see some of these ideas incorporated into products.

Acknowledgments

We would like to thank for their help Philippe Boucard, Didier Martineau, Harry Printz, Didier Roncin, Mark Shand, Peter Stubley and Jean Vuillemin of Digital Equipment Corporation's Paris Research Laboratory.

References

[1] Patrice Bertin. *Mémoires actives programmables: conception, réalisation et programmation.* PhD thesis, Université Paris 7, UFR d'Informatique, June 1993.

[2] Patrice Bertin, Didier Roncin, and Jean Vuillemin. Introduction to programmable active memories. In J. McWhirter J. McCanny and E. Swartzlander Jr., editors, *Systolic Array Processors*, pages 301–309. Prentice-Hall, 1989. Also available as PRL research report number 3.

[3] Patrice Bertin, Didier Roncin, and Jean Vuillemin. Programmable active memories: a performance assessment. In *Symposium on Integrated Systems*, Seattle, WA, USA, March 1993.

MIT Press. Also available as PRL research report 24.

[4] P. Boucard, J. Vuillemin, and M. Shand. Calorimeter Collision Detector on DECPeRLe-1. Technical Report 31, Paris Research Laboratory, Digital Equipement Corp., 85, ave. Victor Hugo, 92563 Rueil-Malmaison Cedex, France, 1993.

[5] D. D. Gajski, editor. *Silicon Compilation.* Addison-Wesley, 1988.

[6] Louis Monier and Jean Vuillemin. Using silicon assemblers. In E. Hörbst, editor, *Proceedings of the IFIP International Conference on VLSI*, pages 315–324, Tokyo, Japan, August 1985. Elsevier Science Publishers.

[7] H. Touati, N. Shenoy, and A. Sangiovanni-Vincentelli. Retiming for Table-Lookup Field-Programmable Gate Arrays. In *First Berkeley Workshop on FPGAs*, February 1992.