Object-Oriented Domain Specific Compilers for Programming FPGAs

Oskar Mencer*, Marco Platzner**, Martin Morf, Michael J. Flynn
Computer Systems Laboratory,
Department of Electrical Engineering,
Stanford, CA 94305, USA.
*Computing Sciences Research Center
Lucent, Bell Labs
Murray Hill, NJ 07974, USA.
** Computer Engineering and Networks Laboratory,
Swiss Federal Institute of Technology,
Zurich, Switzerland.
email: mencer@research.bell-labs.com
platzner@tik.ee.ethz.ch
{morf,flynn}@arith.stanford.edu
Special Issue (Section) on Reconfigurable Computing
Original Control Number: 9900428

Abstract

Simplifying the programming models is paramount to the success of reconfigurable computing with FPGAs. This paper presents a methodology to combine true object-oriented design of the compiler/CAD tool with an object-oriented hardware design methodology in C++. The resulting system provides all the benefits of object-oriented design to the compiler/CAD tool designer and to the hardware designer/programmer. The two examples for domain-specific compilers presented are **BSAT** and **StReAm**. Each domain-specific compiler is targeted at a very specific application domain, such as applications that accelerate boolean satisfiability problems with **BSAT**, and applications which lend themselves for implementation as a stream architecture with **StReAm**.

The key benefit of the presented domain specific compilers is a reduction of design time by orders of magnitude while keeping the optimal performance of hand-designed circuits.

Keywords

Adaptive computing, Configurable computing, Computer arithmetic, Boolean satisfiability

I. Introduction

In this paper we present an object-oriented methodology for domain specific compilers for reconfigurable computing with Field-Programmable Gate Arrays (FPGAs). Our methodology combines a true object-oriented design of the compiler/CAD tool, with an object-oriented hardware design methodology in C++. The resulting system provides all the benefits of object-oriented design to the compiler/CAD tool designer and to the hardware designer/programmer while keeping the optimal performance of hand-designed circuits.

An overview of the general structure of the system is captured in the "city model" shown in Figure I. The infrastructure consists of PAM-Blox[7], an object-oriented module-generation environment. On top of PAM-Blox, we build domain specific compilers. The two examples for domain specific compilers presented in this paper are **BSAT** and **StReAm**. Each domain specific compiler is targeted at a very specific application domain, such as in our case applications that accelerate Boolean satisfiability problems, and applications which lend themselves for implementation as stream architectures.

A. Reconfigurable Computing with FPGAs

FPGAs offer reconfigurability/programmability on the bit-level at the cost of larger VLSI area and slower maximal clock frequency compared to custom VLSI. However, FPGAs are programmable devices that could compete with, or complement microprocessors. Since their introduction, FPGAs have shown the potential for high performance and low power computation[16] resulting from high degrees of parallelism and pipelining:

- **Parallelism**: FPGAs exploit parallelism on the bit-level, arithmetic level, instruction level (ILP for microprocessors), and application level. FPGAs follow a long history of architectures that enable parallelism, such as massively parallel computing, superscalar and VLIW processors. For example, boolean satisfiability architectures for FPGAs[18][22] extract massive bit-level parallelism to achieve orders of magnitude speedups over software.
- **Pipelining**: FPGAs have programmable registers in every cell making them natural candidates for highly pipelined architectures. For example, vector processors utilize pipelining of the data stream to achieve high throughput. Systolic arrays[2][3] offer a regular structure that can be pipelined for high throughput applications. As a current example, stream architectures[9][12][13] use a pipelined dataflow graph, mapped directly into hardware to improve performance and power consumption[16] by an order of magnitude over microprocessors.

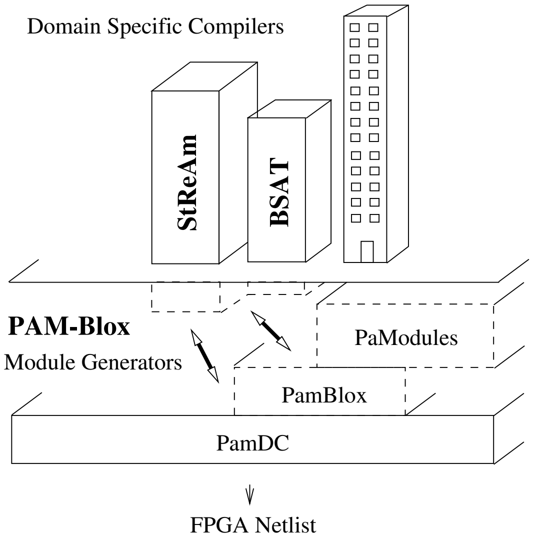

Fig. I. The "city-model" for programming FPGAs. Vertical domain specific compilers such as **StReAm** and **BSAT** sit on top of a horizontal infrastructure for module-generation, PAM-Blox.

B. Design Environments for Reconfigurable Computing

Simplifying the programming models is paramount to the success of reconfigurable computing. Traditional CAD tools for reconfigurable computing are FPGA extensions to general-purpose VLSI CAD tools or derivations such as Synopsys FPGA Express. General-purpose VLSI CAD assumes design times of months to years. Reconfigurable computing, on the other hand, requires a hardware design process with a user interface similar to high-level programming languages. As a response to this gap researchers developed general purpose programming languages for FPGAs such as HandelC[4], JHDL[8], and Pebble[6]. To achieve high-performance circuits, a program for FPGAs needs to express a functional view just like microprocessor programs, but also a physical view corresponding to the architecture of the FPGA circuit. Therefore, general purpose compilation tools for FPGAs try to include the physical view, or architecture of the design, into the semantics of the programming language leading to added programming complexity.

The system presented in this paper takes a different approach and simplifies FPGA programming by creating domain specific compilers. This means that first a compiler is developed for a specific application domain with a specific optimal architecture. Then, application circuits are developed within a lean hardware description environment that uses the semantics of a domain specific compiler. Both the compiler and the hardware design methodology are object-oriented. The advantages of this approach are:

- **Simplified Application Design:** Domain specific compilers implicitly assume a particular architecture, thus keeping the semantics of the description purely functional.
- **Improved Productivity:** The object-oriented design methodology enables code-reuse and thus improves productivity. Code-reuse is also supported by the VLSI CAD environments SystemC[14] and OCAPI[15]. While SystemC focuses on simulation of VLSI designs, OCAPI is a general purpose VLSI CAD tool. These systems provide general-purpose solutions while the examples in this paper show compilers that are optimized for a particular application domain.
- **High Performance:** By building the domain specific compilers on a hierarchy of module generators, the performance of the generated FPGA circuits is comparable to hand-designed circuits.

II. PAM-Blox: Object-oriented Module Generation

Traditional VLSI design for high-performance ASICs consists of complete hand-layout of the data-path and high-level compilation of the control circuit. FPGAs do not support this highly flexible use of silicon area. For data-paths it is therefore sufficient to specify the logic, map the logic to lookup-tables and specify their location on the FPGA device.

Experience with PamDC[1], a gate-level design environment from the PAM project, has shown that a low level, structural representation of FPGA circuits in C++ is very well suited for high-performance FPGA design. The major drawback of PamDC is the enormous design effort required at the gate-level. In order to simplify the design process, we introduce additional levels of abstraction on top of PamDC. Figure I shows an overview of the PAM-Blox system.

PamBlox is a template class library for hardware objects of low complexity, such as adders, counters, etc. *PaModules* are complex, fixed circuits implemented as C++ objects. PaModules consist of multiple PamBlox and are optimized for a specific data-width. Examples are constant(k) coefficient multipliers (KCMs), Booth multipliers, dividers, and special purpose arithmetic units such as a constant multiply modulo $(2^{16} + 1)$ operation for IDEA encryption[16].

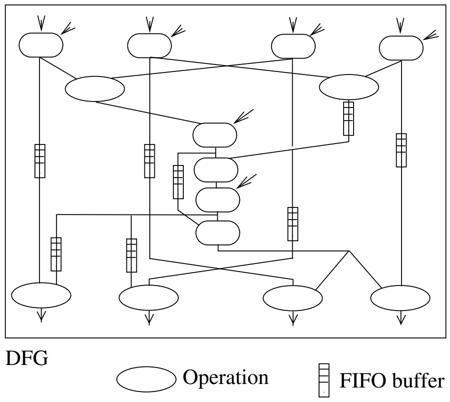

Fig. II. Acyclic dataflow graph (DFG) with operations and distributed FIFO buffers.

With PAM-Blox, hardware designers can benefit from all the advantages of object-oriented system design such as:

- **Inheritance:** Code-reuse is implemented by a C++ class hierarchy. Child objects inherit all public methods (function) and variables (state). For example, all objects with a carry-chain, such as adders, counters, and shifters, inherit the absolute and relative placement functions from their common parent.
- **Virtual Functions:** Part of the parent of a hardware object can be redefined by overloading of inherited (virtual) methods. For example, a two's complement subtract unit can be derived from an adder by forcing a carry-in of one, and inverting one of the inputs.
- **Template Class:** The template class feature of C++ enables us to efficiently combine C++ objects and module-generation. In case of an adder, the template parameter is the bit-width of the adder. The instantiation of a particular object based on the template class creates an adder of the appropriate size.
- **Operator overloading**, function overloading and template functions are used by **StReAm** described below.

Currently, PAM-Blox supports all reconfigurable boards with Xilinx XC4000 series FPGAs. The methodology described in this paper, however, is not limited to this particular FPGA family and may be adapted to any SRAM-based FPGA.

III. Domain Specific Compiler Example 1:**StReAm**

A. *Programming with **StReAm***

The application domain for **StReAm** includes all compute-intensive applications with a performance-critical part that can be implemented as data streaming through a reasonably sized dataflow graph (Figure II).

StReAm uses operator overloading, function overloading and template functions in C++ to create dataflow graphs which are consecutively scheduled to obtain stream architectures. **StReAm** enables high-level programming of any Xilinx XC4000 FPGA on the expression level. **StReAm** includes automatic scheduling of stream architectures, hierarchical wire naming and block placement. **StReAm** simplifies the design of complex stream architectures to just a few lines of code resulting in a reduction of design/program time from weeks to less than a day.

A hardware integer (`HWint`) data type supports the common operators for addition, subtraction, multiplication, division, modulo, etc. The programmer can define other operators and functions by utilizing operator overloading and template functions in C++. Extending the set of operators and functions requires manual design of optimized PamBlox or PaModules. Thus, the designer can adapt the arithmetic units to the specific needs of the application. **StReAm** currently supports arrays of the hardware integer type `HWint`, expressions with `HWint`'s and C++ integers resulting in hardware constants, and static '`for`' loops.

B. *Families of Arithmetic Operators*

One of the advantages of using FPGAs for computing is the flexibility on the arithmetic level. We define families of arithmetic units that are compatible with each other. Currently, **StReAm** supports the following arithmetic families: bit-serial, 4-bit (nibble) serial, parallel pipelined, parallel combinational.

Future work includes extending the hardware types to other number representations such as logarithmic numbers (`HWlog`), fixed point numbers (`HWfix`), floating point numbers (`HWfloat`), the residue number system (`HWresidue`), redundant number representations, and rational number systems[23].

Each arithmetic unit includes a precision value as part of the state of the hardware object. The precision value inside the hardware object enables the evaluations of error propagation through the dataflow graph at compile time. The stream architecture also includes an overflow bit as part of the `HWint` type. The overflow bit of the output of an

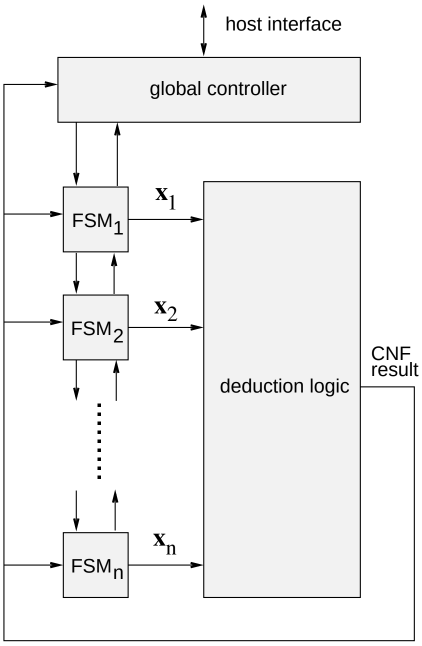

Fig. III. **BSAT** architecture consisting of n FSMs, deduction logic, and a global controller.

arithmetic unit is set if the previous arithmetic operation overflows, or if any overflow bit of the inputs to the previous operation is set.

IV. Domain Specific Compiler Example 2: **BSAT**

A. *Architectures for Boolean Satisfiability*

The Boolean satisfiability problem (BSAT) is to find an assignment of truth values to the variables $x_1, \ldots, x_n$ which makes a Boolean expression of these variables in conjunctive normal form (CNF) true. Recently, several reconfigurable accelerators have been presented for BSAT [19] [18] [22]. These accelerators make use of the great amount of fine-grained parallelism in BSAT instances which matches well the computing structures of FPGAs.

The block diagram of our basic **BSAT** architecture is shown in Figure III and consists of three parts: i) the array of FSMs, ii) the deduction logic, and ii) the global controller. Each variable of the CNF corresponds to one FSM and is modeled in 3-valued logic. Thus it can take on the values $\{0, 1, X\}$, where X denotes an unassigned variable. Given a partial assignment, the deduction logic computes the result of the Boolean expression. The result is fed back to all FSMs which implement chronological backtracking search [22]. The global controller starts the computation and handles I/O. The deduction logic and the number of FSMs are instance-specific.

B. ***BSAT*** *Design Tool Flow*

The **BSAT** design tool flow includes basically three steps. The first step is to generate instance-specific logic for a given satisfiability problem. The second step maps, places, and routes this logic for a specific target FPGA and results in a configuration bitstream. The third step configures the reconfigurable resource, starts the computation, waits for completion, and extracts the results.

The two major issues in the **BSAT** design tool flow are fast circuit generation and the use of optimized FSMs. Depending on the complexity of the problem instance, circuit generation can take by order of magnitude longer than the execution of the hardware algorithm itself. FSM optimization is crucial because as simulations have shown, for most problem instances the FSMs are the limiting factor in terms of hardware complexity.

In order to keep a unified specification of the **BSAT** circuit in C++ and still get maximal optimization of the state machines, we integrate the PAM-Blox design flow with Synopsys FPGA Express II (Figure IV). The application circuit is described in C++, using the libraries PamBlox and PaModules and the domain specific library PamFSM for state machine definition and placement. Running the design executable creates behavioral Verilog for the state machines. Synopsys FPGA Express II is called for synthesis, optimization, and technology mapping. The structural elements of the state machines and the PamBlox-based design are merged on the Xilinx netlist level, augmented with placement directives. State machines can be instantiated multiple times and placed anywhere on the FPGA. Domain specific placement significantly improves the performance of FPGA designs. Placement of state machines is a key feature in **BSAT**, as it is not supported by conventional CAD tools such as Synopsys FPGA Express.

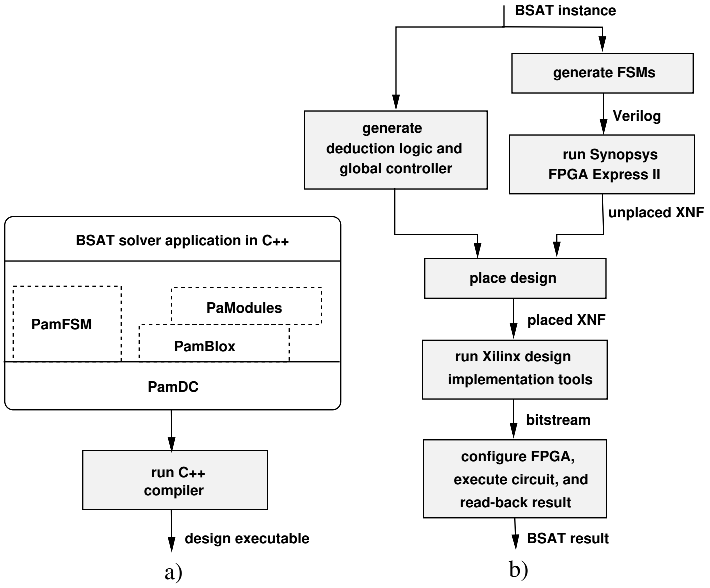

Fig. IV. (a) **BSAT** solvers in C++ use the PAM-Blox and PamFSM libraries. (b) Running the executable creates FSMs in behavioral Verilog which are optimized and merged with the PAM-Blox design on the Xilinx netlist (XNF) level.

V. Benchmarks and Results

A. ***StReAm*** *Results*

The results below show the performance of the final circuits for Xilinx XC4000 FPGAs after Xilinx place and route.

FIR Filter

The following code creates FIR filters with constant coefficients. Operators for addition and multiplication are overloaded to create the appropriate arithmetic units. Multiplying by a constant integer instantiates efficient constant-coefficient multipliers. Data width and datapath width are specified separately to enable digit-serial arithmetic. In the case below we implement a 16-bit FIR filter with 4-bit digit serial arithmetic units. The `delay` operator inserts the FIR filter delays (deltas) similar to the way delays are specified in the Silage language[5]. Array variables `in[]`, and `out[]` are the inputs and outputs of the stream architecture.

```
const int NUM_BLOCK_INPUTS=1;
const int NUM_BLOCK_OUTPUTS=1;
const int BITS = 16, COMP_MODE = DIGIT_SERIAL;
const int STAGES=4, coef[STAGES]={23,45,67,89};
HWint<BITS> delayOut, adderOut;

void Filter::build(){
  delayOut=in[0];
  adderOut=delayOut*coef[0];
  for (i=1;i<STAGES;i++){
    delayOut=delay(delayOut,1);
    adderOut=adderOut+delayOut*coef[i];
  }
  out[0]=adderOut;
}
```

The results (see Table I) show a 4-stage FIR filter implemented with combinational arithmetic units, and three pipelined versions. As expected the bit-serial design takes the smallest area with the longest latency. The parallel, pipelined version has higher throughput but requires most area. The lower part of the table shows the maximal number of stages that **StReAm** can fit on a Xilinx XC4020 FPGA with 800 CLBs. All designs are created with the same few lines of code shown above by simply setting the compiler parameter `COMP_MODE`.

IDEA Encryption

IDEA[10] is a strong encryption algorithm encrypting 64-bit data blocks, using symmetric 128-bit keys. The 128-bit keys are expanded further to 52 sub-keys, 16 bits each. The kernel loop (or round) is generally executed 8 times for either encryption or decryption. Hand crafted results for a stream architecture implementation of IDEA are presented in [16]. **StReAm** produces the same, optimal, IDEA implementation as the hand design at a fraction of the programming

TABLE I

StReAm: FIR Filter Results

	combi-national	pipelined parallel	pipelined digit-serial	pipelined bit-serial
4 stage FIR				
Area[CLB]	246	293	210	184
Cycle Time(CT)	70.1ns	20.8ns	21.2ns	24.2ns
Latency	3	9	17	52
Throughput	16bits/CT	16bits/CT	4bits/CT	1bit/CT
FIR Stages	6	6	14	17
Area[CLB]	332	432	678	635
Latency	5	11	57	260
Cycle Time(CT)	88.7ns	25.1ns	27.3ns	28.0 ns
Throughput	16bits/CT	16bits/CT	4bits/CT	1bit/CT

TABLE II

StReAm: Benchmark Results

	IDEA	IDCT	3D MOTION
Area[CLB]	460	463	320
Cycle Time(CT)	24.1ns	27.9ns	33.9ns
Throughput(bits/CT)	$4 \cdot (16/4) = 16$	12.4	36
Total Latency	17	15	27
Arithmetic	digit-serial 16-bit data	parallel* 14-bit	parallel 12-bit data

*sequential multiply

effort. In order to fit two loops onto one Xilinx XC4020E FPGA we use digit serial arithmetic with a datapath width of 4 bits. The following code shows one round of IDEA encryption:

```
const int NUM_BLOCK_INPUTS=4;
const int NUM_BLOCK_OUTPUTS=4;
const int BITS=16, COMP_MODE=DIGIT_SERIAL;
const int key[10]={9277,98,237,4,978,122,723,3654,24,1536};
HWint<BITS> t[9], temp;

void IDEA::build(){
  t[1] = ideaKCM16(in[0] , key[0]);
  t[2] = (in[1] + key[1]);
  t[3] = (in[2] + key[2]);
  t[4] = ideaKCM16(in[3] , key[3]);
  tmp = t[1] ^ t[3];
  tmp = ideaKCM16(tmp , key[4]);
  t[7] = (tmp + (t[2] ^ t[4]));
  t[8] = ideaKCM16(t[7] , key[5]);
  tmp = (t[8] + tmp);
  out[0] = t[1] ^ t[8];
  out[3] = t[4] ^ tmp;
  tmp = tmp ^ t[2];
  out[1] = t[3] ^ t[8];
  out[2] = tmp;
}
```

The resulting (see Table II) stream architecture with 14 arithmetic units and 8 automatically generated and scheduled FIFO buffers is shown in Figure II. In addition to operator overloading, IDEA requires a special `mod` $2^{16}+1$ multiplier implemented as a PaModule.

Inverse Discrete Cosine Transform (IDCT)

The IDCT is used in signal and image processing (e.g. MPEG, H.263 standards). We implement an 8x8 1-dimensional IDCT following [11]. The actual code for this example is beyond the space constraints of this paper. The resulting (see Table II) stream architecture consists of 98 arithmetic units and 4 FIFO buffers.

3D Motion: Real-time Translation and Rotation

In 3D graphics, a common problem is the translation and rotation of a large set of points in 3D. This stream of points is transformed by a translation vector and two 2D rotation angles obtained from one 3D rotation. The following implementation uses 2D CORDIC modules (`ROTATE()`)[17]. The rotate function demonstrates a multi-input, multi-output module instantiation by overloading the "," operator in `(x[0],y[0])`.

```
const int NUM_BLOCK_INPUTS=3;
const int NUM_BLOCK_OUTPUTS=3;
```

TABLE III
BSAT: BENCHMARK RESULTS

	benchmark			
	hole7	hole8	hole9	hole10
GRASP Runtime [s]	4.56	54.98	627.52	7616.40
Area [CLB]	314	412	522	658
Execution Time [s]	0.062	0.911	16.910	431.110
Compilation Time [s]	134	249	439	597
BSAT Runtime [s]	134.06	249.91	455.91	1028.11
Raw Speedup	73.548	60.351	37.109	17.667
Total Speedup	0.034	0.220	1.376	7.408

```
const int BITS=12, COMP_MODE=PARALLEL;
HWint<BITS> x_in,y_in,z_in;    //inputs
HWint<BITS> x0,y0,z0,phi1,phi2;//rotation
HWint<BITS> dx,dy,dz;          //translation
HWint<BITS> x[2],y[2],z[2];    //temp coords

MOTION3D::build(){
 x_in = in[0]; y_in = in[1];
 z_in = in[2];
 x0 = configReg[0]; y0 = configReg[1];
 z0 = configReg[2];
 phi1 = configReg[3]; phi2 = configReg[4];
 dx = configReg[5]; dy = configReg[6];
 dz = configReg[7];
(x[0],y[0])=ROTATE((x_in-x0),(y_in-y0),phi1);
(y[1],z[1])=ROTATE(y[0],(z_in-z0),phi2);
 out[0]=x[0] + x0 + dx; out[1]=y[1] + y0 + dy;
 out[2]=z[1] + z0 + dz;
}
```

The StreaModule above takes 3 input coordinates (`x_in, y_in, z_in`) representing a point in space. The result is a rotated and translated point (`out[0...2]`). The center of rotation (`x0, y0, z0`), angles (`phi1, phi2`) and translation vector (`dx, dy, dz`) are stored in configuration registers (`configReg`). The value of the configuration registers can be changed without reconfiguration of the FPGAs to perform a particular 3D motion. The code above results (see Table II) in 9 add/sub units, 2 CORDIC units and 1 FIFO buffer.

*B. **BSAT** Results*

We compare the performance of the state-of-the-art software solver GRASP [21] with the performance of our reconfigurable accelerator compiled by **BSAT**. Our prototype is implemented on a PC/NT4.0 platform and uses the Digital PCI Pamette board, equipped with FPGAs of type Xilinx XC4020, as reconfigurable resource.

Table III presents the experimental results for the benchmark class *hole* from the DIMACS benchmark suite [20]. With the **BSAT** design tool flow, the time for FPGA configuration and read-back can be neglected compared to the hardware compilation time, which itself is strongly dominated by the Xilinx tools.

The raw execution times for the reconfigurable accelerator increases more rapidly with the benchmark problem size than the hardware compilation time. This leads to a *cross-over* point in the total speedup around *hole9*. For this benchmark, **BSAT** and GRASP solvers have similar runtimes. For *hole10* we achieve a speedup of 7.408, which reduces the runtime from more than 2 hours in software to about 17 minutes in hardware.

VI. CONCLUSIONS AND FUTURE WORK

Domain specific compilers allow us to focus all optimizations and options on a particular microarchitecture such as Stream or BSAT Architectures. The chosen microarchitecture implicitly defines the set of applications, i.e. an application domain, which map well onto the particular microarchitecture. Once the corresponding domain specific compiler is constructed, the application programmer gets access to a specialized compilation tool that focuses on the application domain at hand. Combining domain specific compilers and the design language into one environment simplifies the effort to develop the compiler and at the same time reduces application design time significantly.

Object-oriented programming is the key to efficient and scalable design for both the compiler framework and the hardware. As a result, PAM-Blox is a convenient infrastructure for domain specific compilers without compromising the performance of the generated circuits. Example 1, **StReAm**, applies the object-oriented design methodology to high-level programming of data streaming applications. While conventional CAD/compiler systems for FPGAs make it very difficult to explore arithmetic optimizations, **StReAm** offers the flexibility to adapt the number representation, precision, and arithmetic algorithm to the particular needs of the application. Example 2, **BSAT**, enables us to quickly explore architectures and algorithms for solving Boolean satisfiability problems on FPGAs. By combining industry-strength state machine optimization with object-oriented module generation and placement, **BSAT** offers fast design time, high flexibility and high performance of the final designs.

Current limitations of our design environment are: In case the generated design does not fit on one FPGA, spatial and/or temporal partitioning must be done manually. Although our framework facilitates automatic spatial partitioning onto multiple FPGAs on the C++ level, temporal partitioning is left for future work.

VII. Acknowledgments

We would like to thank Compaq Systems Research Center for support of this work. Thanks to H. Hübert for helping with the implementation of **StReAm**, and L. Séméria for discussions on the draft of this paper.

References

[1] P. Bertin,D. Roncin,J. Vuillemin, *Programmable Active Memories: A Performance Assessment,* ACM FPGA Conference, Monterey, Feb. 1992.
[2] H.T. Kung, *Why Systolic Arrays,* IEEE Computer, vol. 15, no. 1, pp.37-46, Jan. '82.
[3] H.M. Ahmed, J.-M. Delosme, M. Morf, *Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing,* IEEE Computer, vol. 15, no. 1, pp.65-80, Jan. '82.
[4] Embedded Solutions, *Handel C,* http://www.embeddedsol.com/
[5] G. DeMicheli, *Synthesis and Optimization of Digital Circuits,* McGraw-Hill, '94.
[6] W. Luk, S. McKeever, *Pebble: A Language for Parametrised and Reconfigurable Hardware Design,* Proc. Field-Programmable Logic and Applications (FPL), pp.9-18, Aug.-Sep. '98, Springer '98.
[7] O. Mencer, M. Morf, M.J. Flynn, *PAM-Blox: High Performance FPGA Design for Adaptive Computing,* Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), pp.167-174, Apr. '98, IEEE CS '98.
[8] P. Bellows, B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, M. Rytting, *A CAD Suite for High-Performance FPGA Design,* Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), pp.12-24, Apr. '99, IEEE CS '99.
[9] R. Laufer, R.R. Taylor, H. Schmit, *PCI-PipeRench and SWORDAPI: A System for Stream-based Reconfigurable Computing,* Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), pp.200-208, Apr. '99, IEEE CS '99.
[10] X. Lai, J.L. Massey, S. Murphy, *Markov Ciphers and Differential Cryptanalysis,* Proc. EUROCRYPT '91, pp.17-38, Apr. '91, Springer '91.
[11] E. Linzer, E. Feig, *New Scaled DCT Algorithms for Fused Multiply/Add Architectures,* Proc. Int'l Conference on Acoustics, Speech, and Signal Processing (ICASSP), pp.2201-2204, May '91, IEEE '91.
[12] S. Rixner et al., *A Bandwidth-Efficient Architecture for Media Processing,* Proc. ACM/IEEE Int'l Symposium on Microarchitecture, pp.3-13, Nov.-Dec. '98, IEEE CS '98
[13] C. Ebeling, D.C. Cronquist, P. Franklin, J. Secosky, S.G. Berg, *Mapping Applications to the RaPiD Configurable Architecture,* Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), pp.106-115, Apr. '97, IEEE CS '97.
[14] *The SystemC Community,* http://www.systemc.org/
[15] P. Schaumont, R. Cmar, S. Vernalde, M. Engels, I. Bolsens, *Hardware Reuse at the Behavioral Level,* Proc. Design Automation Conference (DAC), pp.784-789, June '99, IEEE '99.
[16] O. Mencer, M. Morf, M. Flynn, *Hardware Software Tri-Design of Encryption for Mobile Communication Units,* Proc. Int'l Conference on Acoustics, Speech, and Signal Processing (ICASSP), pp.3045-3048, May '98, IEEE '98.
[17] O. Mencer, L. Séméria, M. Morf, J.M. Delosme, *Application of Reconfigurable CORDIC Architectures,* Journal of VLSI Signal Processing, March 2000, 24(2-3):211-221, Kluwer.
[18] P. Zhong, M. Martonosi, P. Ashar, S. Malik, *Accelerating Boolean Satisfiability with Configurable Hardware,* Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), pp.186-195, Apr. '98, IEEE CS '98.
[19] T. Suyama, M. Yokoo, H. Sawada, *Solving Satisfiability Problems on FPGAs,* Proc. Field-Programmable Logic and Applications (FPL), pp.136-145, Sep. '96, Springer '96.
[20] DIMACS satisfiability benchmarks, ftp://dimacs.rutgers.edu/ in directory: pub/challenge/sat/benchmarks/cnf/
[21] J. Silva, K. Sakallah, *GRASP – A New Search Algorithm for Satisfiability,* Proc. IEEE/ACM Int'l Conference on Computer-Aided Design (ICCAD), pp.220-227, Nov. '96, IEEE CS '96.
[22] M. Platzner, G. De Micheli, *Acceleration of Satisfiability Algorithms by Reconfigurable Hardware,* Proc. Field-Programmable Logic and Applications (FPL), pp.69-78, Aug.-Sep. '98, Springer '98.
[23] O. Mencer, *Rational Arithmetic Units in Computer Systems,* PhD Thesis (with M.J. Flynn), E.E. Dept., Stanford, Jan. 2000.