Object-Oriented Domain Specific Compilers for Programming FPGAs

Oskar Mencer, Marco Platzner*, Martin Morf, Michael J. Flynn

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 require the speedup of boolean satifiability with BSAT, and applications which lend themselfes 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, Computer arithmetic, Configurable-computing, Gate-array**

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.

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[9], object-oriented module-generation environment. On top of the infrastructure, 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 require the speedup of boolean satifiability, and applications which lend themselfes for implementation as a stream architecture described below.

A. Reconfigurable Computing with FPGAs

FPGAs offer reconfigurability on the bit-level at the cost of larger VLSI area and slower maximal clock frequency compared to custom VLSI. SRAM-based FPGAs feature fast reconfiguration of the entire chip. Thus, FPGAs are programmable devices that could compete with, or complement microprocessors.

Since their introduction, FPGAs have shown the potential for high performance (or throughput) and low power computation[25]. High performance and low power are a result of exploring high degrees of parallelism and pipelining:

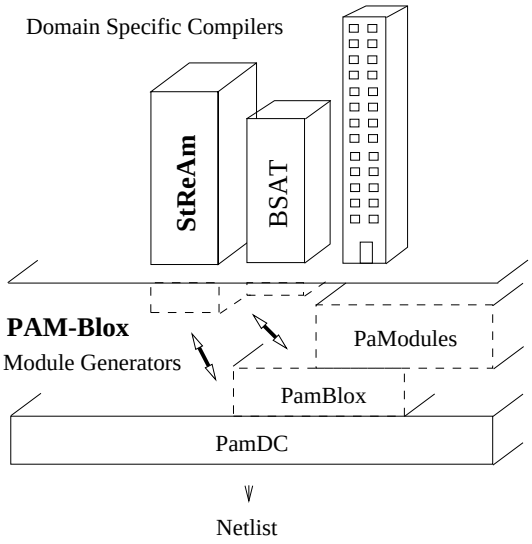

Fig. 1. The figure shows the "city-model" for programming FPGAs. Vertical domain specific compilers such as **StReAm** and **BSAT** sit on top of a horizontal foundation of PAM-Blox layers for module-generation.

- **Parallelism**: FPGAs enable us exploit parallelism on the bit-level, arithmetic level, instruction level (ILP for microprocessors), and application level. FPGAs follow a long tradition of architectures that enable parallelism, such as massively parallel computing, superscalar and VLIW processors. Transforming the sequential description of an algorithm in order to compute more operations in parallel is usually called "extracting parallelism"–a process that can be as painful as it sounds if the algorithm is not cooperative. For example, boolean satisfiability architectures for FPGAs[28][27] use parallelism to achieve orders of magnitude speedups on boolean satisfiability problems.
- **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 throughputs. Systolic arrays[2][3] offer a regular structure that can be pipelined for high throughput applications. As a current example, Stream architectures[15][**?**][20][21] use a pipelined dataflow graph, mapped directly into hardware, to improve performance and power consumption[25] by an order of magnitude over conventional microprocessors.

Simplifying the programming models is paramount to the success of reconfigurable computing. In order to simplify the programming of FPGAs it is necessary to hide a

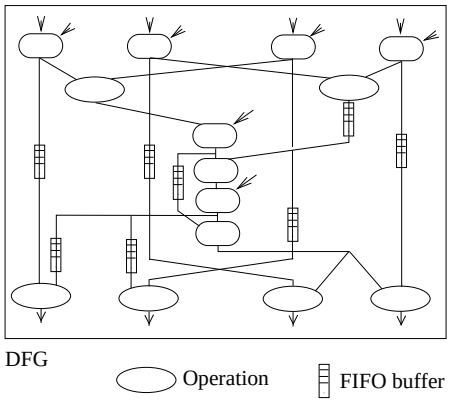

Fig. 2. The figure shows an acyclic dataflow graph (DFG) with operations and distributed FIFO buffers.

CAD tool within a compiler. In comparisson with microprocessors and VLSI, the complexity of programming environments for FPGAs lies between CAD tools and compilers. The main questions are: How should the programming language express the various levels of parallelism available in the FPGA? How to express the timing of the design? How to explore area-time tradeoffs? How to debug the program?

HandelC[5] is a hardware (FPGA) programming language based on communicating sequential processes[4]. Every expression implies a latency of one clock cycle. The designer explores area-time tradeoffs by rewriting expressions.

JHDL[10] and Pebble[8] are examples for structural languages for FPGAs on the PAM-Blox level. Neither JHDL nor Pebble offer operator overloading. Instead, expressions are constructed by nesting function calls.

Novel architectures combining a microprocessor with reconfigurable logic are natural targets for hardware compilation using software languages. An example for compiling C for the Garp processor including a reconfigurable datapath is [16]. The following sections show how our two examples of domain-specific compilers deal with the questions above.

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 offer the high flexibility of silicon area. For data-paths it is therefore sufficient to specify the logic, map the logic to lookup-tables and specify their location.

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[25].

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.

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.

StReAm uses operator overloading, function overloading and template functions in C++ to create dataflow graphs which are consecutively scheduled to obtain a stream architecture. **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 many 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.

In case the resulting circuit does not fit on one FPGA, the options are either spatial and/or temporal partitioning. Spatial partitioning results in multiple FPGAs working in parallel, while temporal partitioning utilizes reconfiguration of FPGAs. Further details on partitioning are beyond the scope of this paper.

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[32]. PaModules also include higher-level arithmetic modules such as CORDIC[26] (Coordinate Rotation Digital Integrated Computer) units.

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 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 block diagram of the basic architecture for solving SAT in hardware is shown in Figure IV. The circuit consists of three parts: i) an array of FSMs, ii) a datapath, and ii) a global controller.

Each variable of the CNF corresponds to one FSM. The FSMs are connected in a one-dimensional array; each FSM can activate its two neighboring FSMs at the top and at the bottom. The architecture of the FSM is algorithm-specific; i.e. for a specific SAT algorithm, all the FSMs are identical. The datapath is a combinational circuit that takes the variables as input and computes outputs that are fed back to the FSMs. The global controller starts the computation and handles I/O communication.

The variables of the CNF are modeled in 3-valued logic. A variable can take on the values $\{0, 1, X\}$, where X denotes an unassigned variable. The datapath computes the 3-valued result of the CNF expression. Initially, all variables are unassigned which also leads to CNF value X, and the global controller activates the top-most FSM. The state diagram for an FSM is shown in Figure **??**. An activated FSM assigns 0 to its variable and checks the resulting CNF value. If the CNF value is 1, the partial assignment already satisfied the CNF and the computation stops. If the CNF

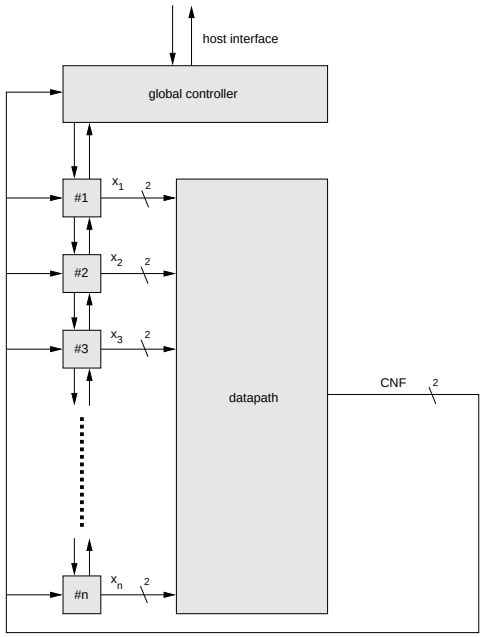

Fig. 3. Block diagram for the basic SAT architecture. It consists of an array of FSMs (#1, . . . ,#n), a datapath, and a global controller. The variables x_i and the CNF are modeled in 3-valued logic.

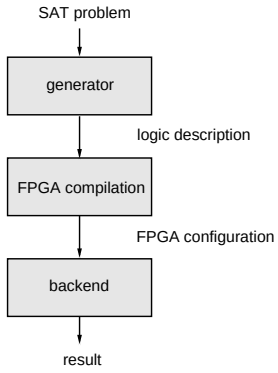

Fig. 4. Steps for solving SAT problems with instance-specific hardware accelerators.

is 0, the partial assignment made the CNF unsatisfiable. In this case, the FSM assigns the complementary value to its variable. If the CNF value is X, the partial assignment did neither satisfy the CNF nor did it make the CNF unsatisfiable. In this case, the FSM activates the next FSM at the bottom. If both value assignments have been tried, the FSM relaxes its variable by assigning X to it, and activates the previous FSM at the top. When the first FSM relaxes its variable and activates the global controller, the SAT problem is proven to be unsatisfiable. By this procedure, the array of interconnected FSMs implements chronological backtracking.

The BSAT design tool flow for instance-specific computation of SAT problems includes basically three steps, as shown in Figure 4. The first step is a generator program that takes a SAT problem as input and generates the instance-specific logic description of this problem. The next step, the FPGA compilation, maps, places, and routes

this description for a specific target FPGA family. The result of this step is a configuration bitstream. The third step, the backend, configures the reconfigurable resource, starts the computation, waits for completion, and extracts the results.

The two major issues in the design tool flow for reconfigurable SAT solvers are: fast circuit generation and the use of predesigned and optimized FSMs. Depending on the complexity of the SAT problem, 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 SAT problems the FSMs are the limiting factor in terms of hardware complexity.

Applications such as Boolean satisfiability require optimized state machines. In order to keep a unified specification of the circuit in C++ and still get maximal optimization of the state machine, we integrate the PAM-Blox design flow with Synopsys FPGA Express II. The tool flow is shown in Figure **??**. The application circuit is described in C++, using the libraries PamBlox, PaModules, and PamFSM for specifying state machines. 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 FSMs and the PAM-Blox design are merged on the Xilinx netlist level, possibly augmented with placement directives.

State machines can be instantiated multiple times and placed anywhere on the FPGA. Hand placement or clever automatic placement can significantly improve the performance of FPGA designs. In addition, placing the state machines is a simple and convenient way to determine the FPGA read-back positions of the state variables. Placement of state machines is a key feature in our environment, as it is not supported by conventional CAD tools such as Synopsys FPGA Express.

V. Benchmarks and Results

A. *StReAm Results*

The following three benchmarks demonstrate the advantages of designing stream architectures with **StReAm**. Results show the performance of the final circuits for the Xilinx XC4000 family after Xilinx place and route tools.

FIR Filter

The following code creates FIR filters with constant coefficients. Operators '+' and '*' 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[6]. The variables `in[]`, `out[]` are the inputs and outputs of the stream architecture.

TABLE I
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

```
const int NUM_BLOCK_INPUTS=1;
const int NUM_BLOCK_OUTPUTS=1;
const int BITS = 16;
const int COMP_MODE = DIGIT_SERIAL;
const int STAGES=4;
const int coef[STAGES]={23,45,67,89};
HWint<BITS> delayOut;
HWint<BITS> 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[17] 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 [25]. **StReAm** produces the same, optimal, IDEA implementation as the hand design at a fraction of the programming 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;
const int COMP_MODE=DIGIT_SERIAL;
const int key[10]={9277,98,237,4,978,122,723,3654,24,1536};
HWint<BITS> t[9];
HWint<BITS> 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 2. In addition to operator overloading, IDEA requires a special `mod` $2^{16}+1$ constant multiplier implemented as a PaModule with fixed bitwidth.

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. 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()`)[26]. 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;
const int BITS = 12;
const int 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;
}
```

TABLE II
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

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 software implementation with the performance on our reconfigurable computing system. Our hardware prototype is implemented on the PC platform, this time running Windows NT 4.0. As reconfigurable resource we use a Digital PCI Pamette board, equipped with 4 FPGAs of the type Xilinx XC4020.

We define the raw speed-up S_{raw} of the reconfigurable SAT solver as t_{sw}/t_{hw}, the run-time ratio of software and hardware SAT solvers. The overall run-time for computing a SAT problem in reconfigurable hardware consists of the hardware compilation time, t_{comp}, the time for configuring the FPGA, t_{config}, the actual hardware execution time, t_{hw}, and the time for reading back and extracting the result, t_{read}.

$$t_{overall} = t_{comp} + t_{config} + t_{hw} + t_{read} \quad (1)$$

The overall speed-up $S_{overall}$ is then given by $t_{sw}/t_{overall}$.

Table **??** presents the experimental results for the *hole* benchmarks. With our 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 design implementation tools.

The examples *hole6* to *hole9* were mapped onto one Xilinx XC4020. For *hole10*, an FPGA of type XC4025/XC4028 is necessary. As we know the number of clock cycles for *hole 10* from a simulation of the SAT solver and the maximum clock frequency from running the FPGA compilation tools, we were able to determine exactly the speed-ups for this benchmark. The hardware cost in Table **??** suggests that *hole10* can be mapped in one XC4020. However, our placement strategy tries to minimize the distances between the FSMs and the datapath logic blocks. This prevents us from placing too many FSMs in an FPGA.

benchmark	t_{sw} $[s]$	*hardware cost* $[CLBs]$	τ_{min} $[ns]$	t_{hw} $[s]$	t_{comp} $[s]$	$t_{overall}$ $[s]$	S_{raw}	$S_{overall}$
hole6	0.31	230	15.4	0.005	103	103.01	62.000	0.003
hole7	4.56	314	16.2	0.062	134	134.06	73.548	0.034
hole8	54.98	412	19.0	0.911	249	249.91	60.351	0.220
hole9	627.52	522	23.4	16.910	439	455.91	37.109	1.376
hole10	7616.40	658	37.5	431.110	597	1028.11	17.667	7.408

With this strategy, we never ran into routing problems for the datapath logic and we were able to achieve a rather high performance for these irregular designs.

The hardware and software execution times are shown in Figure **??**. The raw execution times of hardware and software SAT solvers increase more rapidly with the problem size than the hardware compilation time. This leads to a *cross-over* point in the overall speed-up around *hole9*. For this benchmark, SAT solvers in instance-specific hardware and software have similar overall run-times. For *hole10* we achieve a speed-up of 7.408, which reduces the run-time from more than 2 hours in software to about 17 minutes in hardware.

Table **??** shows that the raw speed-up S_{raw} is decreasing with the problem size. This is for two reasons. First, as simulations [31] have shown, a slightly decreasing speed-up seems to be an artifact from applying the presented deduction strategy to this particular SAT problem class. Second, larger problems result in more complex circuits which lead to longer clock cycles times.

VI. Conclusions and Future Work

Domain specific compilers enable the compiler designer to focus all optimizations and options on a particular application domain and architecture. The programmer matches an application domain with the particular application and gets access to a specialized tool that focuses on the problem at hand. Thus, domain-specific compilers simplify the effort to develop the compiler and at the same time reduce design time significantly.

The key to efficient and scaleable design of a compiler framework and the hardware description methodology is object-oriented programming. Object-oriented module generators in C++, PAM-Blox, form a very flexible and convenient substrate for domain specific compilers based on C++. An important next step is to adapt the state-of-the-art in hardware/software co-design of parallel and pipelined systems[22] into a compiler for FPGAs or reconfigurable resources closely coupled with a microprocessor or memory. Ideally, such a compiler would be able to explore parallelism and pipelining on the algorithm level, instruction level, arithmetic level and bit level.

The advantages of object-oriented design with C++ which have been recognized in the software industry find their way into VLSI CAD[23][24] and into programming of FPGAs shown in this paper. Our first example, **StReAm**, applies the object-oriented design methodology to high-level programming of FPGAs. 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.

Our second example, 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, BSAT offers fast design time, high flexibility and high performance of the final designs.

VII. Acknowledgments

We would like to thank Compaq Systems Research Center for support of this work, and M. Shand for maintaining PamDC. Thanks to H. Huebert for implementing operator overloading and scheduling for **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, February 1992.

[2] H. T. Kung, *Why Systolic Arrays,* IEEE Computer, 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, Jan. 1982.

[4] C.A.R. Hoare *Communicating Sequential Processes,* Prentice Hall International, London, 1985.

[5] Embedded Solutions *Handel C,* http://www.embeddedsol.com/

[6] G. DeMicheli *Synthesis and Optimization of Digital Circuits,* McGraw-Hill, 1994.

[7] W.H. Mangione-Smith, et.al. *Seeking Solutions in Configurable Computing,* IEEE Computer Magazine, December 1997.

[8] W. Luk, S. McKeever, *Pebble: A Language for Parametrised and Reconfigurable Hardware Design,* Field-programmable Logic and Applications (FPL), Tallinn, Estonia, Aug. 1998.

[9] O. Mencer, M. Morf, M. J. Flynn, *PAM-Blox: High Performance FPGA Design for Adaptive Computing,* Field-programmable Custom Computing Machines, Napa Valley, CA, 1998.

[10] P. Bellows, B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, M. Rytting, *A CAD Suite for High-Performance FPGA Design,* Field-programmable Custom Computing Machines, Napa Valley, CA, 1999.

[11] M. Chu, N. Weaver, K. Sulimma, A. DeHon, J. Wawrzynek, *Object Oriented Circuit-Generators in Java,* Field-programmable Custom Computing Machines, Napa Valley, CA, 1998.

[12] M.B. Gokhale, J.M. Stone, *NAPA C: Compiling for a Hybrid RISC/FPGA Architecture,* Field-programmable Custom Computing Machines, Napa Valley, CA, 1999.

[13] A. Koch, *Enabling Automatic Module Generation for FCCM Compilers,* Poster Session 1, Field-programmable Custom Computing Machines, Napa Valley, CA, 1999.

[14] S.A. Guccione, D. Levi, P. Sundararajan, *JBits: A Java-based Interface for Reconfigurable Computing,* 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD).

[15] R. Laufer, R. Reed Taylor, H. Schmit *PCI-PipeRench and SWORDAPI: A System for Stream-based Reconfigurable Computing,* IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, 1999.

[16] T.J. Callahan, J. Wawrzynek, *Instruction-Level Parallelism for Reconfigurable Computing,* Field-Programmable Logic and Applications (FPL), Tallinn, Estonia, Aug-Sep 1998

[17] X. Lai, J.L. Massey, S. Murphy, *Markov Ciphers and Differential Cryptanalysis,* EUROCRYPT '91, Lecture Notes in Computer Science 547, Springer-Verlag, 1991.

[18] E. Linzer, E. Feig, *New Scaled DCT Algorithms for Fused Multiply/Add Architectures,* International Conference on Acoustics, Speech, and Signal Processing, Proceedings ICASSP .91, Vols.1-5, pp.2201-2204, 1991.

[19] T.J. Callahan, P. Chong, A. DeHon, J. Wawrzynek, *Fast Module Mapping and Placement for Datapaths in FPGAs,* Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays, Feb. 1998.

[20] S. Rixner, et. al. *A Bandwidth-Efficient Architecture for Media Processing,* Symposium on Microarchitecture, Dallas, Texas, Nov. 1998.

[21] C. Ebeling, D.C. Cronquist, P. Franklin, J. Secosky, S.G. Berg, *Mapping Applications to the RaPiD Configurable Architecture,* Field-programmable Custom Computing Machines, Napa Valley, CA, 1997.

[22] S. Bakshi, D.D. Gajski, *Partitioning and Pipelining for Performance-Constrained Hardware/Software Systems,* IEEE Transaction on VLSI Systems, Dec. 1999.

[23] *The SystemC Community,* http://www.systemc.org/

[24] S. Vernalde, P. Schaumont, I. Bolsens, *An Object Oriented Programming Approach for Hardware Design,* IEEE Workshop on VLSI'99, Orlando, April 1999.

[25] O. Mencer, M. Morf, M. Flynn, *Hardware Software Tri-Design of Encryption for Mobile Communication Units,* Proceedings ICASSP, Seattle, May 1998.

[26] O. Mencer, L. Séméria, M. Morf, J.M. Delosme, *Application of Reconfigurable CORDIC Architectures,* The Journal of VLSI Signal Processing, Special Issue: VLSI on Custom Computing Technology, Kluwer, March 2000.

[27] P. Zhong, M. Martonosi, S. Malik, P. Ashar, *Implementing Boolean Satisfiability in Configurable Hardware,* Logic Synthesis Workshop, May, 1997.

[28] T. Suyama, M. Yokoo, H. Sawada, *Solving Satisfiability Problems on FPGAs,* International Workshop on Field-Programmable Logic and Applications (FPL), 1996.

[29] DIMACS satsifiability benchmark suite,
ftp://dimacs.rutgers.edu/
in directory: pub/challenge/sat/benchmarks/cnf/

[30] J. Silva, K. Sakallah, *GRASP – A New Search Algorithm for Satisfiability,* IEEE ACM International Conference on CAD, Nov. 1996.

[31] M. Platzner, G. De Micheli, *Acceleration of Satisfiability Algorithms by Reconfigurable Hardware,* International Workshop on Field-Programmable Logic and Applications (FPL), 1998.

[32] O. Mencer, *Rational Arithmetic Units in Computer Systems,* PhD Thesis (with M.J. Flynn), E.E. Dept., Stanford, Jan. 2000.