Dynamic Circuit Generation for Boolean Satisfiability in an Object-Oriented Design Environment

Oskar Mencer and Marco Platzner
Computer Systems Laboratory, Stanford University
(oskar@umunhum.stanford.edu, marco.platzner@computer.org)

Abstract

We apply our object-oriented design environment PAM-Blox to dynamic generation of circuits for reconfigurable computing. Our approach combines the structural hardware design environment with commercial synthesis of finite state machines (FSMs). The PAM-Blox environment features a well defined hardware object interface and the ability to control the placement of hand-optimized circuits. We integrate the advantages of an object-oriented design environment with full control over placement at every level of abstraction, with commercial FSM synthesis and optimization.

As driving application we consider reconfigurable hardware accelerators for the NP-complete Boolean satisfiability problem. These accelerators require a fast compilation of circuits consisting of instance-specific datapaths and control automatons. By providing FSM optimization and control over placement, our design environment enables the maximization of performance.

1 Introduction

Reconfiguration models used in configurable computing can be classified into *compile-time reconfiguration* (CTR) and *run-time reconfiguration* (RTR) [6]. In CTR, the hardware compilation and the reconfiguration are done at compile-time. At run-time, the circuit loaded onto a reconfigurable resource is executed for many sets of input data. In RTR, the configuration of the reconfigurable resource is changed while the application is running. Here, the reconfiguration time becomes part of the application's run-time and has to be minimized.

Recently, another model of reconfigurable computing emerged: *instance-specific reconfiguration*. In this model, new hardware is generated for every problem instance, i.e., every set of input data, of a particular algorithm. In such a case, not only the reconfiguration time but also the hardware compilation time becomes part of the overall run-time. This is denoted as *dynamic circuit generation*.

Applications that can make use of the instance-specific reconfiguration model must show two characteristics, lots of fine-grained parallelism dependent on the actual input data and long run-times in software. The first characteristic ensures that fine-grained reconfigurable resources, such as Field Programmable Gate Arrays (FPGAs), achieve high speed-ups compared to microprocessors. The second characteristic is required to hide the hardware compilation overhead. The driving application for this reconfiguration model is the acceleration of Boolean satisfiability problems [13] [9] [2] [10] [12].

For this paper, we required a design environment that supports dynamic reconfiguration. Our approach combines PAM-Blox, a structural object-oriented design tool that gives control over placement, with commercial FSM synthesis and optimization. The concept of a hardware object demonstrated with PAM-Blox was introduced in [8] and has proven itself to be highly competitive with commercial tools such as Synopsys FPGA Express II. Circuits for solving Boolean satisfiability problems consist of some datapath blocks and many cooperating finite state machines. In terms of hardware area, the FSMs are dominating. A design environment that supports dynamic reconfiguration must therefore not only provide a method to re-use highly optimized datapath components, but also a method of specifying complex FSMs, both with control over placement.

In the remaining part of this section, we define the Boolean satisfiability problem, discuss the general design tool flow for reconfigurable satisfiability solvers, and introduce the PAM-Blox design environment. In Section 2, we describe our new environment that combines FSMs with PAM-Blox. Section 3 presents an example of a hardware architecture to solve satisfiability problems in reconfigurable hardware. First experimental results achieved are discussed in Section 4. Section 5 concludes this paper.

1.1 The Boolean Satisfiability Problem

The *Boolean satisfiability problem* (SAT) is a fundamental problem in mathematical logic and computing theory with many practical applications in areas such as computer-aided design of digital systems, automated reasoning, and machine vision. In computer-aided design, tools for synthesis, optimization, verification, timing analysis and test pattern generation use variants of SAT solvers as core algorithms. The SAT problem is commonly defined as follows [5]:

Definition 1 *Given i) a set of n Boolean variables $x_1, x_2, \ldots, x_n$, ii) a set of literals, where a literal is a variable x_i or the negation of a variable $\bar{x}_i$, and iii) a set of m distinct clauses $C_1, C_2, \ldots, C_m$, where each clause consists of literals combined by the logical* or *connective $\vee$, determine, whether there exists an assignment of truth values to the variables that makes the Conjunctive Normal Form (CNF) $C_1 \wedge C_2 \wedge \ldots \wedge C_m$ true, where $\wedge$ denotes the logical* and *connective.*

An example for a SAT problem with 4 variables and 3 clauses is $(x_1 \vee \bar{x}_2) \wedge (x_1 \vee \bar{x}_3 \vee x_4) \wedge (x_2 \vee \bar{x}_4)$. The vector $(x_1, x_2, x_3, x_4) = (1, 1, 0, 0)$ is one possible solution to this SAT problem.

Since the general SAT problem is NP-complete, exact methods to solve SAT show an exponential worst-case run-time complexity. This limits the applicability of exact SAT solvers in many areas.

The SAT problem is a *discrete, constrained decision problem* [5]. A straightforward but inefficient procedure to solve it exactly is to enumerate all possible truth value assignments and check if one satisfies the CNF. Many of the improved techniques that have been proposed to solve SAT problems eliminate one variable from the CNF at a time. There are two basic methods: *splitting* and *resolution*. Resolution was implemented in the original Davis-Putnam (DP) algorithm [4]. Splitting was used first in Loveland's modification to DP, the DPL algorithm [3]. In splitting, a variable is selected from the CNF and two sub-CNFs are generated by setting the variable to 0 and 1, respectively. The iterative application of splitting generates a *search tree*; a leaf of the tree denotes a full assignment of values to variables. Most practical SAT solvers use the splitting technique and combine it with *backtracking*. Backtracking searches the search tree in a depth-first order and thus avoids excessive memory requirements.

Existing software SAT solvers use a wide variety of backtracking methods and strategies for decision, deduction, and diagnosis. A very sophisticated example is the GRASP algorithm [11], which is also used as reference in our work. The powerful strategies that are implemented by sophisticated SAT solvers reduce the number of variable assignments required to find a solution or to prove that there is no solution. However, these strategies can be computationally very expensive.

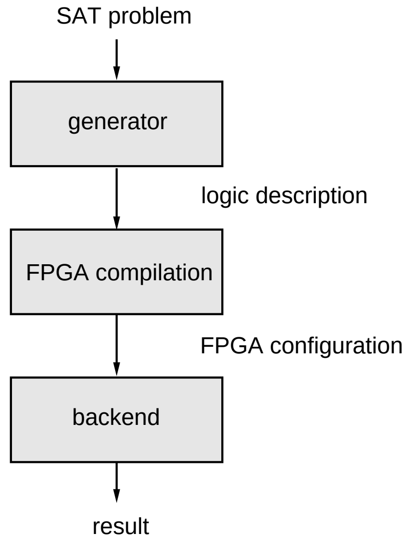

Figure 1: Steps for solving SAT problems with instance-specific hardware accelerators.

Recently, reconfigurable hardware architectures have been proposed to solve hard instances of the SAT problem [13] [14] [9] [2] [10] [12]. For each instance of a SAT problem, i.e., for each CNF, new hardware is generated on-the-fly reflecting the particular structure of the CNF. These instance-specific architectures rely on fine-grained computing structures as provided by FPGA technology.

1.2 Design Tool Flow for Reconfigurable SAT solvers

A design tool flow for instance-specific computation of SAT problems includes basically three steps, as shown in Figure 1. 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

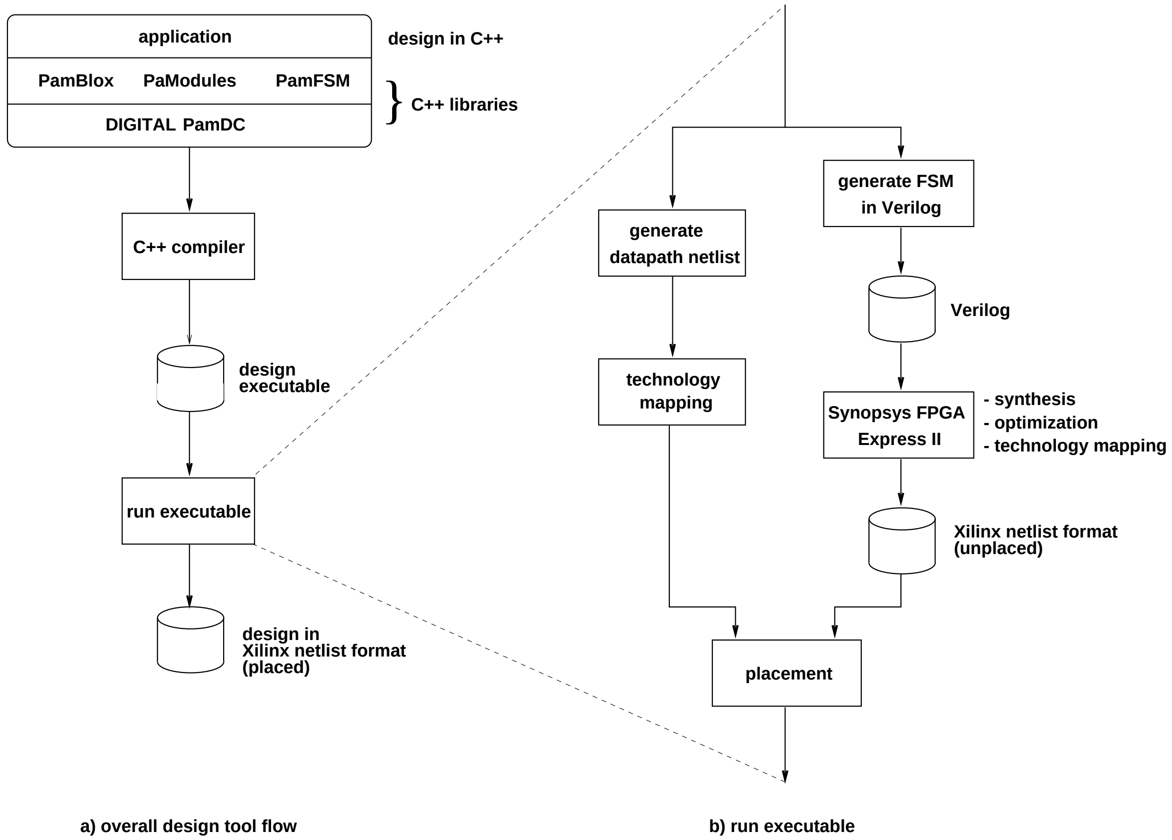

Figure 2: a: Tool flow for designing and placing finite state machines within the PAM-Blox environment. b: Running the C++ executable creates a behavioral Verilog description of the state machine which is optimized by Synopsys FPGA Express II and merged with the PAM-Blox design on the Xilinx netlist level.

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.

Zhong et al. [14] presented a tool flow for a reconfigurable SAT solver where the instance-specific logic is described in VHDL. This description is then partitioned and mapped onto an array of Xilinx XC4000 series FPGAs by an IKOS logic emulation system. The advantage of this approach is that large FPGA systems can be targeted. The drawback is that the utilization of the FPGAs and the achieved clock frequencies are usually rather low.

A different approach is proposed by Rashid et al. [10]. SAT-specific CAD tools for synthesis, partitioning, placement, and routing are being developed for an open FPGA architecture, namely the Xilinx XC6200 family. The proposed design tool can generate either a logic description in VHDL, requiring commercial tools for FPGA compilation, or directly a configuration bitstream for the Xilinx XC6216.

The design tool flow from Suyama et al. [12] generates a logic description in the hardware description language SFL. This description is synthesized by the PARTHENON CAD tools and mapped onto a ZyCAD system, which consists of Xilinx XC4000.

In our approach, we generate a logic description in form of a Xilinx FPGA netlist and use the commercial Xilinx design implementation tools for mapping, placement, and routing. We address the issue of fast circuit generation by controlling the placement of the FSMs with the object-oriented hardware design environment PAM-Blox/PamDC. FSMs are optimized with a commercial synthesis and optimization tool such as Synopsys FPGA Express II.

1.3 PAM-Blox: Object-Oriented Structural Design

PAM stands for Programmable Active Memories described in [7]. PamDC was developed for the PAM project to offer circuit generation on the register-transfer-level (RTL) in C++. For datapaths, hand de-

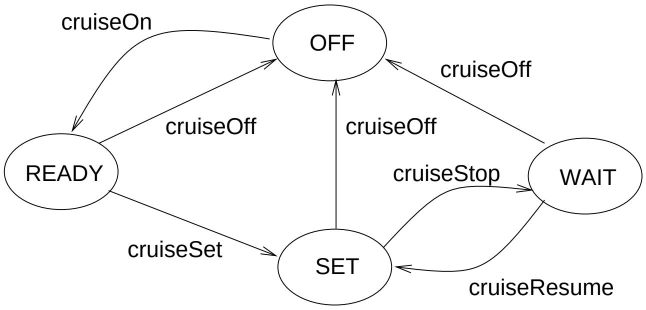

Figure 3: CRUISE1, a finite state machine for cruise control in a car.

signs are typically more efficient than compiled behavioral descriptions. In order to exploit the efficiency of hand design while simplifying the design process, PAM-Blox, introduced in [8], offer a bottom up approach to compilation for custom computing machines. By using a powerful and highly optimized parameterizable library of hardware object generators, PAM-Blox, we add levels of abstraction that preserve optimal area and performance while simplifying the design process. The first level, PamBlox, consists of parameterizable simple elements such as counters and adders. Automatic placement of carry chains and flexible shapes are supported. PaModules are more complex elements possibly instantiating PamBlox. PaModules have fixed shapes and are usually optimized for a specific data-width. Examples for PaModules are multipliers, Coordinate Rotations (CORDICs), and special arithmetic units for encryption.

PAM-Blox simplifies the design of datapaths for FPGAs by implementing an object-oriented hierarchy in PamDC/C++. With PAM-Blox, hardware designers can benefit from all the advantages of object-oriented system design that the software industry has learned to cherish during the last decade. Efficient use of function overloading, virtual functions, and templates makes PAM-Blox a very powerful and yet simple to use design environment.

By implementing PAM-Blox together with the actual design within a C++ hierarchy, we simplify the task of adapting library modules to the specific needs of the application. Therefore PAM-Blox circuit generators are easily scalable and allow FPGA designers to share and reuse pieces of designs by writing new PamBlox and PaModules.

2 Combining PAM-Blox with FSM Synthesis

PAM-Blox has proven itself to be very useful for designing high performance data intensive applications.

```
FSM *myFSM = new FSM("cruise");
...
myFSM->set_output("offOut");
myFSM->set_output("readyOut");
myFSM->set_output("setOut");
myFSM->set_output("waitOut");

myFSM->set_input("cruiseOn");
myFSM->set_input("cruiseOff");
myFSM->set_input("cruiseSet");
...
myFSM->set_state("OFF",   "0001");
myFSM->set_state("READY", "0010");
myFSM->set_state("SET",   "0100");
myFSM->set_state("WAIT",  "1000");
myFSM->set_init_state("OFF");

myFSM->set_trans("OFF",   "READY", "cruiseOn");
myFSM->set_trans("READY", "SET",   "cruiseSet");
myFSM->set_trans("SET",   "WAIT",  "cruiseStop");
myFSM->set_trans("WAIT",  "SET",   "cruiseResume");
...
myFSM->set_output_loc("offOut",   dx, dy0, FFX);
myFSM->set_output_loc("readyOut", dx, dy0, FFY);
myFSM->set_output_loc("setOut",   dx, dy1, FFX);
myFSM->set_output_loc("waitOut",  dx, dy1, FFY);

myFSM->generate_instance("cruise1", x, y);
```

Figure 4: Code fragment showing the specification of the state machine CRUISE1 in C++. The state machine will be implemented as Moore automaton. Thus, the state encoding corresponds to the specified order of outputs.

However, the tediousness of creating control units remained a major drawback of the object-interface.

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 2. 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.

Figure 3 shows a simple state machine controlling the cruise control of a car. The PamFSM specification is presented in Figure 4. 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.

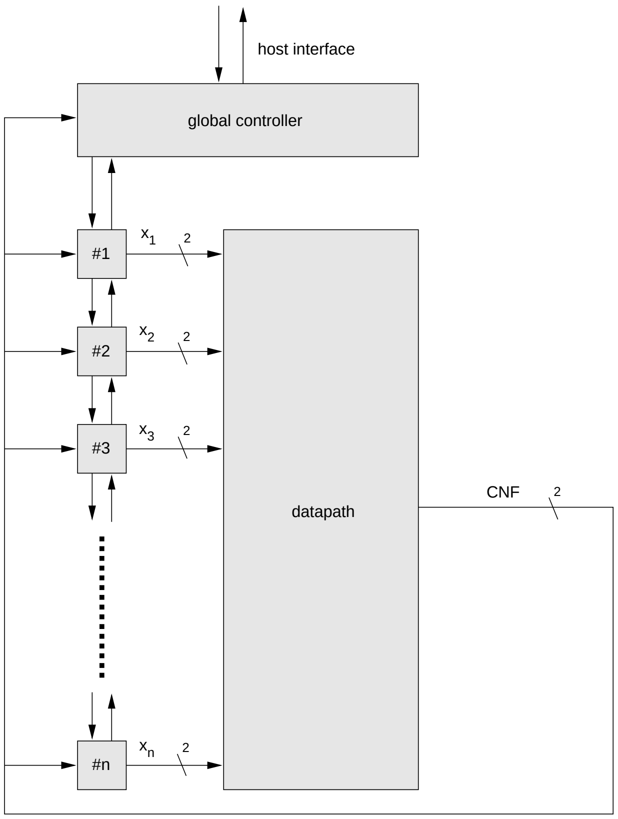

Figure 5: Block diagram for the basic SAT architecture. It consists of an array of FSMs (#1, ... ,#n), a datapath, and a global controller. The variables x_i and the CNF are modeled in 3-valued logic.

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.

3 Reconfigurable Architectures for SAT

The block diagram of the basic architecture for solving SAT in hardware is shown in Figure 5. 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.

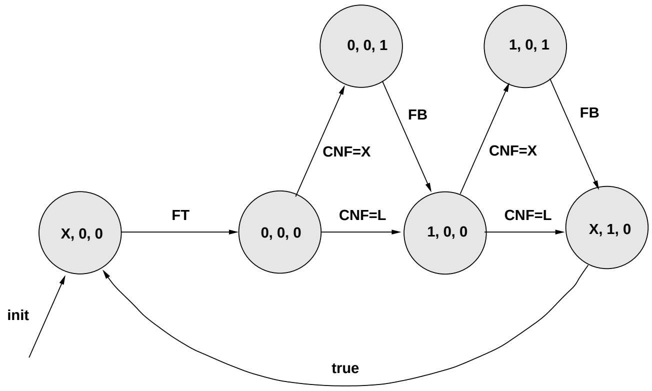

Figure 6: State diagram for an FSM of the architecture CE. The inputs are `FT` (from top) and `FB` (from bottom) that activate the FSM, and the 3-valued CNF. The output signals displayed inside the states are the variable value, and the signals `TT` (to top) and `TB` (to bottom) that activate the previous and next FSM.

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 6. 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 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.

Most reconfigurable architectures that have been proposed for solving SAT in CNF form share the basic block diagram shown in Figure 5. They differ in the modeling of the variables (2-valued, 3-valued, or 4-valued logic) and in the used deduction strategy, which is reflected in the actual implementation of the datapath and the FSM. For all architectures, the datapath

benchmark	*variables*	*clauses*	t_{sw} $[s]$
hole6	42	133	0.31
hole7	56	204	4.56
hole8	72	297	54.98
hole9	90	415	627.52
hole10	110	562	7616.40

Table 1: *hole* benchmarks from the DIMACS benchmark suite. The software SAT solver GRASP was executed with parameters `+bD +dDLIS` on a Pentium-II/300MHz/128MB RAM PC platform running Linux.

and the number of FSMs is instance-specific. However, the global controller and the single FSM do not change with the CNF.

Architectures that implement more powerful deductions strategies also have more complex datapaths and FSMs. The architecture presented here offers the least powerful deduction strategy and has the smallest hardware requirements. As discussed in [9], this basic architecture can be a viable option for smaller SAT problems or in cases with resource limitations. In this paper, we restrict our discussion to this basic SAT architecture, as the more complex alternatives lead to the same issues.

4 Experimental Results

In this section, we report experimental results for the class of benchmarks *hole* taken from the DIMACS satisfiability benchmarks suite [1]. The *hole* benchmarks are instantiations of the pigeon hole problem, formulated as a SAT problem in CNF. This benchmark class is well-suited for evaluation, as all the examples are unsatisfiable and hard to solve, i.e., software SAT solvers have long run-times. Table 1 lists the benchmark examples with their problem size, given by the numbers of variables and clauses, and the run-time of the software SAT solver GRASP [11] on a PC platform.

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}.

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

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

Table 2 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 2 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. 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 7. 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 2 shows that the raw speed-up S_{raw} is decreasing with the problem size. This is for two reasons. First, as simulations [9] 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.

5 Conclusion and Future Work

Although hardware compilation for current FPGAs takes minutes to hours, instance-specific SAT solvers are promising for hard SAT problems, i.e., problems

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

Table 2: Results for running the *hole* benchmarks. For each benchmark, the software run-time t_{sw}, the hardware cost in configurable logic blocks (CLBs), the minimum clock period τ_{min}, the hardware execution time t_{hw}, the hardware compilation time t_{comp}, the overall run-time for the reconfigurable system $t_{overall}$, and the raw and overall speed-up are shown.

where software solvers show very long run-times. The sources of the potential speed-ups are the deduction steps that show large amounts of fine-grain parallelism. This makes FPGAs with their fine-grained logic blocks an optimal target for instance-specific hardware accelerators for SAT problems.

Our design environment that combines PAM-Blox/PamDC with commercial FSM synthesis and optimization has proven itself to be very convenient for dynamic hardware generation. The entire application, including the code generation for the instance-specific circuits as well as the run-time functions for downloading bitstreams and reading back results, can be handled in a single representation in C++. This greatly simplifies the construction and usage of instance-specific systems.

The toolflow is still very clumsy as it requires us to run Xilinx place-and-route tools. A specialized place-and-route tool together with the availability of larger FPGAs would make the reconfigurable solution even more competitive.

In the future, we will extend the SAT architectures by assigning cost values to the variables. This will allow to solve the important class of minimum-cost problems. We plan to apply these reconfigurable SAT solvers to real-world problems in CAD.

In the current implementation, we use only one FPGA of the Pamette board. To employ arrays of FPGAs, we are investigating different partitioning techniques. This includes mapping large SAT circuits onto several FPGAs and splitting large SAT problems into subproblems that can be run independently.

In addition, we consider to extend PamFSM to more general models, including Mealy machines and derivatives.

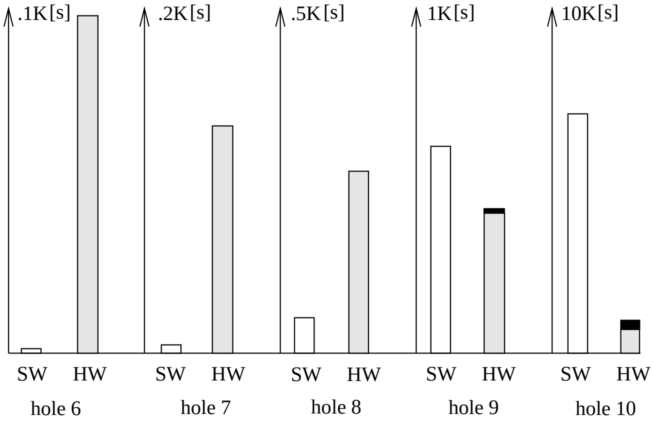

Figure 7: Software and hardware execution times in seconds, with scaled y-axes for the different benchmarks. The right bars are the hardware compilation times, including hardware execution time shown as black area. Only *hole 10* spends a significant amount of time executing in hardware.

6 Acknowledgment

We would like to thank Prof. De Micheli and Prof. Flynn at the Computer Systems Laboratory, Stanford University for their support and constant encouragement that led to this work. Oskar Mencer is supported by DARPA contract No. DABT63-96-C-0106. Marco Platzner was partially supported by the Austrian National Science Foundation *FWF* under grant number J01412-MAT.

References

[1] DIMACS satsifiability benchmark suite, available at *ftp://dimacs.rutgers.edu* in the directory */pub/challenge/sat/benchmarks/cnf/*.

[2] Miron Abramovici and Daniel Saab. Satisfiability on Reconfigurable Hardware. In *International Workshop on Field-Programmable Logic and Applications (FPL)*, pages 448–456, 1997.

[3] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, (5):394–397, 1962.

[4] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, (7):201–215, 1960.

[5] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the Satisfiability (SAT) Problem: A Survey. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 35:19–151, 1997.

[6] Brad L. Hutchings and Michael J. Wirthlin. Implementation Approaches for Reconfigurable Logic Applications. In *International Workshop on Field-Programmable Logic and Applications (FPL)*, 1995.

[7] Oskar Mencer, Martin Morf, and Michael J. Flynn. PAM-Blox: High Performance FPGA Design for Adaptive Computing. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.

[8] Marco Platzner and Giovanni De Micheli. Acceleration of Satisfiability Algorithms by Reconfigurable Hardware. In *International Workshop on Field-Programmable Logic and Applications (FPL)*, pages 69–78, 1998.

[9] Azra Rashid, Jason Leonard, and William H. Mangione-Smith. Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.

[10] J. Silva and K. Sakallah. GRASP – A New Search Algorithm for Satisfiability. In *IEEE ACM International Conference on CAD '96*, pages 220–227, November 1996.

[11] Takayuki Suyama, Makoto Yokoo, and Hiroshi Sawada. Solving Satisfiability Problems on FPGAs. In *International Workshop on Field-Programmable Logic and Applications (FPL)*, pages 136–145, 1996.

[12] Jean E. Vuillemin, Patrice Bertin, Didier Roncin, Mark Shand, Herve H. Touati, and Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. In *IEEE Transactions on VLSI*, March 1996.

[13] Peixin Zhong, Margaret Martonosi, Pranav Ashar, and Sharad Malik. Accelerating Boolean Satisfiability with Configurable Hardware. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.

[14] Peixin Zhong, Margaret Martonosi, Pranav Ashar, and Sharad Malik. Solving Boolean Satisfiability with Dynamic Hardware Configurations. In *International Workshop on Field-Programmable Logic and Applications (FPL)*, pages 326–335, 1998.