A Practical Reconfigurable Hardware Accelerator for Boolean Satisfiability Solvers

John D. Davis
Microsoft Research
Silicon Valley Lab
joda@microsoft.com

Zhangxi Tan
EECS Department
UC Berkeley
xtan@cs.berkeley.edu

Fang Yu
Microsoft Research
Silicon Valley Lab
fangyu@microsoft.com

Lintao Zhang
Microsoft Research
Silicon Valley Lab
lintaoz@microsoft.com

ABSTRACT
We present a practical FPGA-based accelerator for solving Boolean Satisfiability problems (SAT). Unlike previous efforts for hardware accelerated SAT solving, our design focuses on accelerating the most time consuming part of the SAT solver ─ Boolean Constraint Propagation (BCP), leaving the choices of heuristics such as branching order, restarting policy, and learning and backtracking to software. Our novel approach uses an application-specific architecture instead of an instance-specific one to avoid time-consuming FPGA synthesis for each SAT instance. By avoiding global signal wires and carefully pipelining the design, our BCP accelerator is able to achieve much higher clock frequency than that of previous work. In addition, it can load SAT instances in milliseconds, can handle SAT instances with tens of thousands of variables and clauses using a single FPGA, and can easily scale to handle more clauses by using multiple FPGAs. Our evaluation on a cycle-accurate simulator shows that the FPGA co-processor can achieve 3.7-38.6x speedup on BCP compared to state-of-the-art software SAT solvers.

Categories and Subject Descriptors
C.3 Special-purpose and application-based systems

General Terms
Design, Experimentation, Verification.

Keywords
SAT solver, reconfigurable, BCP, co-processor, FPGA .

1. INTRODUCTION
Boolean Satisfiability (SAT) solvers are widely used as the underlying reasoning engine for electronic design automation, as well as in many other fields such as artificial intelligence, theorem proving, and program verification. Due to its wide adoption, much effort has been dedicated to design efficient SAT solvers. In recent years, tremendous progress has been made on SAT solving software in algorithm advancements, efficient implementation, and heuristic tuning. These improvements have enabled SAT instances with hundreds of thousands of variables and clauses to be solved in practice [1].

It is not surprising that researchers have studied using (reconfigurable) hardware accelerators for SAT solving. Designs based on Field Programmable Gate Arrays (FPGAs) have been described in [2][3][4][5][6][7], and were compared in a survey [8]. A parallel SAT solver using a reconfigurable processor is described in [9]. Unfortunately, unlike the software solvers, none of the hardware-assisted SAT solvers have gained much traction in practice. Most of the existing hardware accelerators were designed before the invention of the so called "chaff-like" modern SAT solvers [10][1]. Compared with modern software solvers, the hardware accelerators from previous work are usually slow and capacity limited, and they are unable to accommodate some important features in modern software solvers such as learning.

DAC 2008, June 8–13, 2008, Anaheim, California, USA

In this paper, we describe the design of an FGPA co-processor that focuses on accelerating the most time consuming component in a SAT solver, namely Boolean Constraint Propagation (BCP). BCP is a stable component in the modern SAT solver's implication engine, making it a good target for hardware acceleration. Our approach leaves the rest of the SAT solver functionality to software, thereby accommodating the constant innovations in SAT solver research. In our design, the co-processor is implemented in an FPGA. It heavily relies on modern FPGA's Block RAM (BRAM) to provide parallelism and memory bandwidth. The co-processor can load new SAT instances in a few milliseconds. Using a single FPGA it can handle SAT instances with tens of thousands of variables and clauses. The design is flexible enough to accommodate learning and non-chronological backtracking, as well as scaling to multiple FPGAs for increased capacity[1]. Due to careful pipelining, the synthesized design can achieve a clock frequency of up to 200MHz. Our full system cycle-accurate simulator shows that the BCP accelerator can achieve a speedup of 3.7~38.6x compared to a modern CPU for BCP operations. Finally, the co-processor can be easily used in the implication engines of many modern SAT solvers that use different heuristics and/or strategies.

2. OVERVIEW OF SAT ALGORITHM
We assume the readers are familiar with modern SAT solvers based on the Davis-Logemann-Loveland (DLL [11]) algorithm with learning (see [1] for a survey). In this section, we provide a brief overview of the algorithm that is relevant to the BCP co-processor and the hybrid solver design.

The DLL algorithm, which solves formulae in Conjunctive Normal Form (CNF), is a branch and search algorithm consisting of several major functions. The *branching* function heuristically chooses a *free* (unassigned) variable and assigns it a value. This is often called a *decision*. The *deduce* function propagates the effect of the decision based on the *unit implication rule*. A clause is said to be *unit* if all but one of its literals are assigned the value false and the remaining literal is *free* (unassigned). To satisfy the clause, the free literal needs to be implied to be true, and this clause is called the *antecedent* clause of the newly inferred variable. The iterative application of the unit implication rule is

[1] Not yet implemented in our current implementation.

called Boolean Constraint Propagation (BCP). If a variable is implied to be both true and false by different clauses, it is said that the current variable assignment is *conflicting*, and a *conflict analysis* function is invoked. The conflict analysis function may add extra clauses into the formula. This process is called *conflict-driven learning*. From conflict analysis, the solver needs to undo certain variable assignments, also known as *backtracking*.

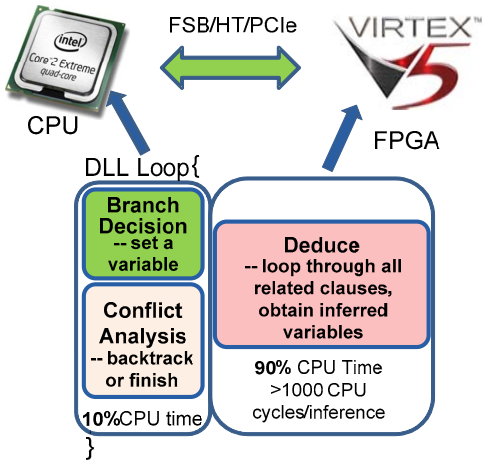

Figure 1. A modern SAT solvers and SW/HW partition.

While this work focuses on the deduction part of the SAT solver, the decision and conflict analysis part is equally important because they greatly affect the search space. In the last decade, SAT solver performance has been improved continuously with major innovations in the branching heuristics and learning schemes. Besides these major components, several other techniques such as preprocessing (e.g.[12]) and random restarts [13] also greatly contribute to the success of modern SAT solvers. Still, deduction remains to be one of the most important components of the SAT solvers and usually takes most of the time during SAT solving.

3. A HARDWARE ASSISTED SAT SOLVER

Our main goal in this research is to leverage hardware acceleration to build a *practical* SAT solver that performs better than pure software solvers for real world SAT instances. Due to the constant improvements of software SAT solvers on branching, learning and restarting heuristics, it is impossible to completely map a software SAT solver into hardware and build a practical hardware SAT solver, as previously attempted (e.g. [2][3][4]). Moreover, due to the huge research investment in so called "chaff-like" solvers; it would be difficult for algorithms that significantly deviate from it (e.g. [6]) to be competitive performance wise.

Therefore, we focus our effort on accelerating only Boolean Constraint Propagation (BCP), which accounts for 80%−90% of the CPU runtime in a highly optimized SAT solver [1][10][2]. Literature continues to provide optimizations for software BCP implementations, but the principle behind BCP (unit implication) has not changed, thus making it a good target for hardware acceleration. Moreover, most SAT solvers have clean interfaces between the implication engine and the rest of the solvers, thus providing an easy way to integrate the BCP co-processor into different solvers. Figure 1 provides the CPU time breakdown of the SAT solver from software profiling and the resulting partitioning we used in implementing our hybrid SAT solver using both a general purpose CPU and a reconfigurable BCP co-processor.

Modern SAT solvers have tuned the BCP engine so well that each implication only takes a couple thousand CPU cycles for typical SAT instances. To achieve reasonable speedup is very challenging for an FPGA implementation, due to the inherently lower clock frequency of FPGAs compared to CPUs and ASICs. Since the CPU is running in the Gigahertz range, the BCP co-processor must have a high clock frequency and each implication on average can at most take tens of cycles. Global signals must be avoided and the BCP co-processor must be fully pipelined (in contrast to [2][3]). Most SAT instances encountered in the industrial applications contain thousands of variables. Therefore, the implication time must be independent of the variable count (in contrast to [4][5]). Finally, for any practical solutions, the hardware resource requirement must be roughly proportional to the size of the instance being solved (in contrast to [7]).

3.1 BCP Co-processor Overview

Due to the large amount of time required for FPGA logic synthesis and placement and routing, we avoided developing a BCP *instance-specific* SAT solver (e.g. [2][3]). Instead, we load SAT instances into an *application-specific* FPGA BCP co-processor. Similar to [7], we leverage the Block RAM (BRAM) in modern FPGAs to store instance specific data. This approach not only reduces the instance loading overhead, but also simplifies the design of the interface with the host machine. Our design is mainly bounded by the BRAM capacity of the FPGA. In our current design, we target the Xilinx Virtex 5 LX110T FPGA, which can handle up to 64K variables and 64K clauses. Using the largest Xilinx FPGA (Virtex 5 LX330T), the capacity can be extended to 64K variables and 176K clauses, by instantiating more inference engines. For even larger SAT instances, the BCP co-processor design can be easily expanded to utilize multiple FPGAs, as we will present in Section 3.3.

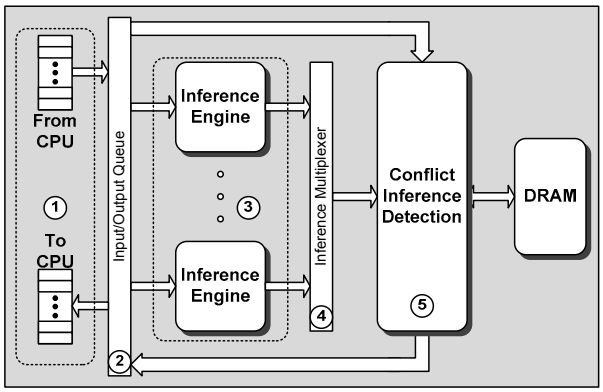

Figure 2. Overall FPGA-based BCP Accelerator Architecture.

The overall system architecture of the BCP co-processor is shown in Figure 2. It is composed of the following major components, described from left to right in Figure 2:

1. **CPU Communication Module**: This module receives branch decisions from and returns inference results back to the CPU. The co-processor communicates with the CPU through high speed links such as AMD HyperTransport (HT) [18], Intel's Front-Side

[2] More recent SAT solvers such as minisat and rsat have better heuristics for pruning, branching and restart, but their BCP performances are comparable to Zchaff.

Bus [17], and/or PCI Express (PCIe) [16] (we present simulation results using HT and PCIe in Section 4). To maximize effective bandwidth and minimize communication cost, inference results, including the status (conflict or not) and implied variables and their antecedents, are sent to the CPU in batches when the communication buffer is full or the current iteration is finished. Likewise, decisions are received from the CPU in batches. Modern SAT decision heuristics (e.g. VSIDS [10]) are *state-independent*, which means the decision order of a variable is independent of variable assignments. Therefore, the software on the CPU can compute several decisions in a batch and send them to the co-processor. The co-processor performs BCP on each decision variable one by one unless a conflict occurs. If a decision variable is already implied by previous decisions, the co-processor just ignores it. To accelerate conflict resolution, an undo operation is built into the communication module such that when a conflict occurs, it unassigns variables still in the buffer (i.e. variables assigned at current decision level), at the same time, reporting the results to the CPU.

2. **Input/Output Queue**: Decisions from the CPU and implications derived from the inference engines are queued in a FIFO (the equivalent of the implication queue in a software SAT solver) and sent to multiple implication inference engines. This module also sends the implication to a buffer to be sent to the CPU if there are no conflicts. Both input/output queues employ a banked architecture to sustain a bandwidth of up to 14.4 Gbps in each direction (running at 200 MHz). The communication is also asynchronous, potentially hiding communication latency.

3. **Parallel Inference Engines**: Clauses of the SAT formula are partitioned and stored in multiple inference engines. Given a variable index and its assigned value, the inference engines infer values of other literals within a fixed number of hardware cycles regardless of the number of literals in a clause. The new implications are put into a FIFO buffer to be dequeued for further processing. We present a more detailed description of the inference engines in Section 3.2.

4. **Inference Result Multiplexer:** This module serializes the data communications between the parallel inference engines and sequential *conflict inference detection* stage. This module uses a 2-level priority-encoded multiplexing bus architecture that supports up to 256 inference engines (physical resource in our target FPGA (LX110T) limits us to 64 parallel inference engines). Each data port is registered and has independent flow control allowing pipelined operation.

5. **Conflict Inference Detection**: This is a serialized pipelined process to detect conflict inference results generated by the parallel inference engines. This module maintains a global copy of the variable states. Given a new implication from the inference engine, it detects if there is a conflict, and whether the same implication was already produced by a different clause. If a conflict is detected, the BCP co-processor notifies the CPU using an interrupt. This invokes the conflict analysis and backtracking process in the software. In our prototype, a total of 64K variables are supported by the on-chip BRAM global status table. In order to save BRAM overhead in the inference engines, the input to the conflict detection module is locally indexed, i.e. a tuple $<l, k, n>$ indicates that the l-th literal in clause k of inference engine n is implied. A separate pipelined DRAM is used to translate the local index to its global variable and clause ID.

3.2 Inference Engines

The inference engine is the key component of our co-processor design. Due to space limit, in this section, we can only provide a high level description of how it works.

In a pre-processing step, we partition clauses into groups so that they can be processed by multiple inference engines in parallel, one group per engine. If there are multiple clauses associated with a particular variable, these clauses are distributed over different engines so that the clauses can be processed in parallel to produce at most one new implication per inference engine. This partitioning scheme also ensures that each group has a limited number of clauses (limited by the BRAM capacity).

Each inference engine is pipelined and has two stages, as shown in Figure 3. The first stage is called *clause index walk*, which takes a variable assignment from the implication queue (Input/Output Queue) and locates the corresponding clause. In the second *implication* stage, the clause's status is retrieved from the clause status table, which includes the value of each literal in the clause. The clause status is then examined to identify new implications. In the rest of this section, we briefly go through these two stages.

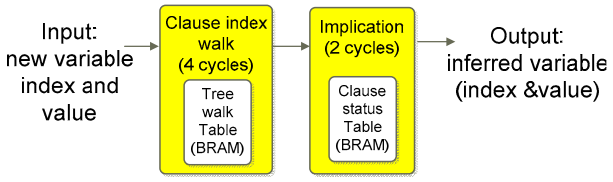

Figure 3: Inference engine overview.

The clause index walk module uses a tree to efficiently locate the clause associated with the input variable. The tree is stored in the *tree walk table* in an on-chip BRAM block local to the module. Suppose the variable index has a width of k (so that the BCP co-processor can handle 2^k variables), and every non-leaf tree node has 2^m child nodes, then the tree will be k/m deep. Here both k and m are configurable. Given a non-leaf node, the address of its leftmost child in the tree walk table is called the *base index* of this tree node. The rest of the children are ordered sequentially, following the leftmost child. Therefore, to locate the ith child, the index can be calculated by adding i to the base index. If a child is not associated with any clauses, we store a no-match (-1) tag in the entry. If for a node, all of its 2^m children have no match, then we do not expand the tree node and just store a no-match tag in the node itself. The entry of a leaf node stores the clause ID where the variable occurs, as well as the literal index in the clause that corresponds to the variable.

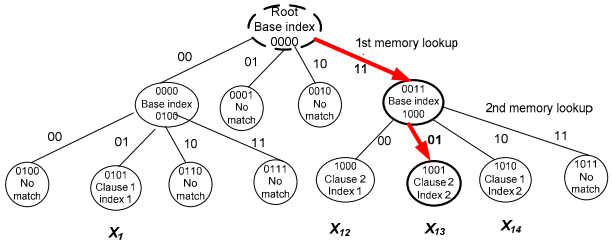

Figure 4. Clause index tree walk in the inference engine.

Figure 4 provides a simple example with the literal index size k =4 and the tree branch width m=2. There are two clauses, (X_1 V X_{14}) and (X_{12} V X_{13}), with variable X_1's index is 0001, X_{12}'s index is 1100, X_{13}'s index is 1101, and X_{14}'s index is 1110. Suppose the

new input variable is 1101. The base index of the root node is 0000 and the first two bits of the input are 11. The table index is the sum of two: 0000+11= 0011. Using this table index, the first memory lookup is conducted by checking the 0011 entry of the table. This entry shows that the next lookup is an internal tree node with the base index 1000. Following this base index, adding it to the next two bits of the input 01, we reach the leaf node 1000+01 = 1001. This leaf node stores the variable association information; in this case, the variable is associated with the second variable of clause two.

After identifying the related clause, the *implication* module retrieves its status from the clause status table in the BRAM in one cycle. In the next cycle, the inference module checks whether the clause generates a new implication. The condition for generating a new implication is that only one variable is unassigned and all the rest are set to false. The implication generation algorithm is based on checking this condition using bit-wise logic and the algorithm is shown in Figure 5. It is implemented with combinational logic, which can be computed in a *single cycle*.

Algorithm: Find implication in a clause
Input: Clause status bit vector, X
Output: Literal position if an inference is made, otherwise an all-one position vector
1 Begin
2 For each literal X_i calculate two or-reduced signal using clause status bit vector, i.e.
3 HI_i = or-reduce all status bit of literal X_j, where $j > i$
4 LO_i = or-reduce all status bit of literal X_j, where $j < i$
5 End for;
6 Return X_i is the new implication, when $HI_i = LO_i = false$ and $X_i = unassigned$, otherwise an all-one vector
7 End

Figure 5. Inference algorithm.

The hardware inference algorithm tries to make inferences for each literal concurrently (line 2-5 in parallel). For each literal in the clause, it checks whether the inference condition has been met − whether it is an unassigned variable (line 6), and all the literals before it (line 3) and after it are all false (line 4). If so, it reports a new inference. Finally, the inference engine updates the clause status table in the BRAM with the new clause status.

In our current implementation of the inference engine, *m=4* and *k=16*. Therefore, we need 16/4=4 BRAM reads in the clause index tree walk, each takes one cycle. One engine can support up to 1024 clauses. The status of these clauses is stored in a BRAM block, which supports clauses with up to 9 literals. Note this can be easily extended to support many more literals (see Section 3.3). FPGA BRAMs are dual-ported. The clause index tree walk uses one of the ports. The other port is designated to the tree walk table initialization and reprogramming interface.

3.3 Discussions on the BCP Co-Processor

As mentioned before, our inference engine supports clauses with up to 9 literals. For longer clauses, extra variables could be introduced to break them into smaller ones. Our single chip design using an LX110T currently supports up to 64K variables and 64K clause and is limited by on-chip BRAM resources. Using the largest Xilinx FPGA (LX330T) available, the capacity of the BCP co-processor can be extended to 176K clauses. In both cases, the *Input/Output Queue* and *Inference Result Multiplexer* are underutilized. This makes it relatively easy to extend the design to multiple FPGAs without changing the architecture. Our architecture supports adding more inference engines by connecting multiple FPGAs using the *Input/Output Queue* and the *Inference Result Multiplexer* across chip boundaries, as shown in Figure 6. Our current design can support up to 256 inference engines without modifying the existing RTL code. The system shown in Figure 6 can accommodate up to 256K clauses using five Virtex 5 LX110T FPGAs. The potential downside of this multi-FPGA design is increased overall latency due to crossing chip boundaries. The overall BCP co-processor design is highly modularized and can support additional levels of hierarchy, which enables connecting even more FPGAs and scaling up the solvable SAT instances linearly. Finally, like the clause capacity, both the maximum number of variables and literals for each clause can also be scaled by using more BRAM resources.

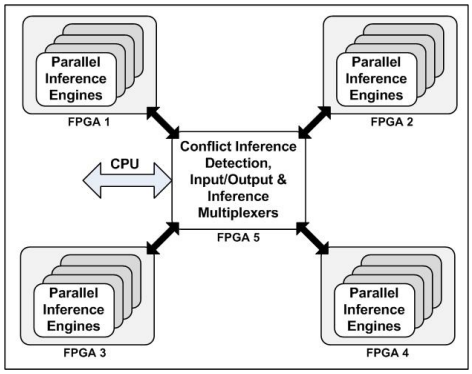

Figure 6. Scaling the BCP co-processor with multiple FPGAs.

Unlike many existing designs, our architecture does not require re-synthesis and subsequent placement and routing of the FPGA before solving a new SAT instance. The time for software pre-processing of clauses to create the clause partitions and the FPGA reprogramming time can all be done in less than a second. The actual FPGA "reprogramming" can be done by configuring the clause index walk tables in the inference engines, requiring about 1 millisecond in the worst case. For this purpose, we designed a 3.6 Gbps CPU-to-FPGA programming interface that is independent of the normal data path using the second port of the dual-ported BRAM. The rest of tables and state in FPGA (including the FIFOs, clause status tables and variable states in the conflict detection module) are reinitialized the same way. This FPGA programming interface allows dynamic clause addition during the solving process, if required. This gives us the flexibility of being able to add learned clauses [14].

3.4 Implementation and Circuit Performance

There are numerous physical optimizations used to make the FPGA BCP co-processor run as fast as possible. First, the whole design is heavily pipelined. Each stage described in Section 3.1 represents a high-level pipeline stage. Furthermore, within each high-level stage, pipelining is used to improve performance. For example, on some timing critical paths, such as the inference engine, internal pipeline stages are added to break down global wires and help route signals. Manually mapping FPGA primitives was another method used to achieve the highest clock frequency. For example, special low-latency FPGA structures, such as fast carry-chains, are used in pattern detection and priority encoding.

We use Xilinx ISE 9.2i SP3 to synthesize and place and route our design, which has about 4000 lines of VHDL code. The target device is Xilinx Virtex 5 LX110T with a speed grade of -3. The timing of our post placed and routed design is reported to be 5.0

ns (200 MHz) without any manual floor planning. Changing to an FPGA with a speed grade of -2 degrades performance by 15%, to approximately 170 MHz. Manual floor planning is a potential future performance optimization opportunity, which is especially useful when pushing the BRAM utilization to its limit. The FPGA resource consumption is shown below in Table 1. Because we translate Boolean expressions into a collection of memory addresses, it is no surprise that on-chip BRAM (also used for FIFOs) is heavily consumed (91%). If aggressive BRAM block packing is used by not maintaining the design hierarchy, the BRAM blocks usage can be reduced to about 70%. The other on-chip FPGA resources are used moderately (<20%) with registers having slightly higher utilization because of additional pipeline registers required to achieve the aggressive clock rate.

Table 1. Resource Utilization on XC5VLX110T-3FF1136.

Resources	Used	Utilization
Registers	14,221	20%
LUTs used as logic	12,144	17%
LUTs used as memory	1,536	8%
BRAMs	138	91%
Equivalent ASIC Gate Count	13,861,127	

An alternative to an FPGA-based BCP co-processor would be to develop an ASIC. Kuon and Rose compared FPGAs to ASICs with respect to performance, area, and power [15]. They showed that ASIC performance was 3.5 to 4.8 times better depending on the speed grade of the FPGA for benchmarks using both logic and memory. Likewise, ASIC area was about 33 times smaller, but this could be reduced dramatically to as low as 18 to 5 times smaller if using the built-in hard blocks. Thus, we expect a single chip ASIC BCP co-processor can operate at approximately 700 MHz and support well over 256 parallel inference engines.

4. RESULTS

The FPGA implementation was simulated and tested using ModelSim 6.3. We developed a cycle-accurate simulator to evaluate the design. The ModelSim results from the FPGA BCP co-processor were used for the timing information in our cycle-accurate simulator, which faithfully simulates events in the FPGA. We compare the FPGA co-processor running at 200 MHz with a state-of-the-art software-only SAT solver based on a modified version of Zchaff [10]. It runs on a 3.6 GHz Pentium 4 with 2 GB of RAM. Since the FPGA co-processor only performs the BCP part of the solver, we compare it with the software BCP module, which takes around 90% of total running time (excluding initial I/O cost for loading instances). We disabled learning in the software SAT solver because our initial co-processor design does not support learning. For each SAT instance, we run the solver for one million implications. The execution time of the BCP co-processor is divided by the CPU cycle time to report the performance results in terms of CPU cycles.

For the link between the FPGA and CPU, we simulate two types of connection: HyperTransport (HT) [18] and PCI-Express (PCIe) [16]. The simulated HT operates at 800 MHz with 16 lanes using DDR technology. Each packet has a maximum size of 64-bytes and uses 8b/10b encoding. For writes, the packet control overhead is 8 bytes and for reads, the packet control overhead is 12 bytes. The round trip delay between the FPGA and CPU is 300ns using HT. PCIe has higher latency (560ns), but the max packet size is bigger (128 bytes) and the bandwidth is higher (2Gbps effective bandwidth per lane with 16 lanes).

We use two types of SAT instances, the first is a set of randomly generated k-SAT instances at phase transition point across various clause sizes and the second is a collection of real SAT instances from commonly used SAT benchmarks. Using both real world and synthetic SAT instances prevents the system from being optimized for a specific instance and demonstrates good performance across a range of SAT instances and sizes.

Table 2. Parameters for generating random SAT instances.

	3-SAT	4-SAT	5-SAT	6-SAT
Clause-Variable Ratio	4.3	9.9	21	43
FPGA to CPU speedup (HT)	7.7	10.7	16.3	38.6
FPGA to CPU speedup (PCIe)	6.7	8.3	13.3	30.5

We randomly generated *k*-SAT instances ($3 \leq k \leq 6$) with 200 variables at the phase transition point [19], producing non-trivial *k*-SAT instances. The clause-variable ratios at the phase transition point are shown in the first row of Table 1. The second and third rows in the table show the speedup ratios of the BCP FPGA co-processor using HT and PCIe links vs. the software BCP implementation. From the table, we can see that the speedup ratio varies from 6.7 to 38.6. We notice that the speedup ratio increases as the clause-variable ratio increases. Intuitively, there are two reasons behind this. First, a higher clause-variable ratio means one variable appears in more clauses, producing greater opportunity for parallelism. Second, the CPU spends more cycles per implication sequentially processing the variables when there are more variables per clause. In Figure 7, the left bar in the cluster of three bars shows how the number of clock cycles per implication grows using a software-only SAT solver. In contrast, using the BCP co-processor, multiple clauses can be evaluated in parallel in the FPGA instead of sequentially as in the software SAT solver. In addition, for each clause, the FPGA can examine all the literals in one clause in constant time regardless of the clause length (as long as it is less than 9). As a result, the number of cycles needed per implication remains almost constant when moving from 3-SAT to 6-SAT, as shown in Figure 7.

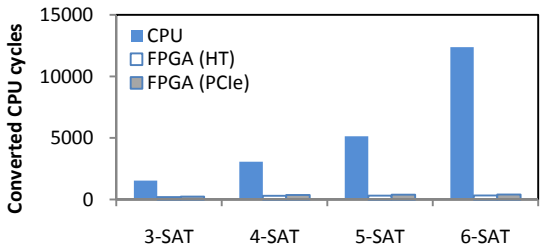

Figure 7. Number of converted CPU cycles per implication.

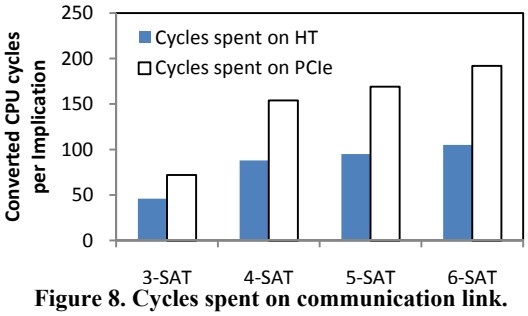

Figure 8. Cycles spent on communication link.

It should also be noted that in our experiments, the performance of HT is slightly better than PCIe as shown in Figure 8. As mentioned earlier, PCIe has higher bandwidth and latency, while

HT has lower bandwidth and latency. Our design tries to reduce communication overhead by using the largest packets possible, and batch communication as much as possible. However, only a small amount of batched data is transferred between the co-processor and the CPU. Therefore, link latency plays a more important role than link bandwidth even with the batch asynchronous communication and difference in packet size. Under such scenario, HT performs better than PCIe. Among all the cycles consumed in the FPGA, HT constitutes 15-20 % of total cycles, while PCIe is 30-40%. We also observed that the benchmarks with higher clause-variable ratios generate more conflicts (we fix the total number of implications). As one might expect, the more conflicts, the more round trip interactions between CPU and FPGA. Therefore, the total cycles spent on communication go up for benchmarks with a higher clause-variable ratio.

Next, we tested the performance of our SAT co-processor using real world benchmarks, which have clauses with mixed length. Table 3 shows the speedup ratio of the FPGA co-processor compared to the software-based BCP running on the CPU. On average, one implication takes around 5-15 FPGA cycles. Converting FPGA cycles to CPU cycles and comparing it to the software solvers, the speedup ratio varies from 3.73 to 14.1. *Crypto* benchmarks (names starting with *crypto*) have the greatest speedup. Similar to the synthesized instances, the performance of HT is better than PCIe. The difference in speedup is mainly due to the clause-variable ratio. In particular, the *miters* benchmarks tend to have smaller clause-variable ratios. In contrast, *crypto* benchmarks tend to have higher clause-variable ratios. Currently, our co-processor does not support clause learning; therefore, learning is disabled in both the software and the hybrid solver. Since learned clause tends to contain many literals; we expect our co-processor to be even more competitive when learning is enabled.

Table 3. Performance speedup on real word instances.

SAT instances	Clause variable ratio	FPGA to CPU speedup ratio	
		HT	PCIe
miters-c2670	2.50	3.92	3.73
miters-c3540	2.70	6.07	5.91
miters-c499	3.09	4.59	4.10
miters-c5315	2.96	4.32	4.17
miters-c7552	2.67	4.33	4.26
mites-c880	5.08	4.08	3.82
bmc-galileo-8	5.08	7.53	7.70
bmc-galileo-9	5.14	7.54	7.72
bmc-ibm-10	5.48	11.31	11.25
bmc-ibm-11	4.67	8.80	8.66
crypto-md4_wang5	4.16	12.64	12.26
crypto-md5_48	4.17	14.10	14.00
crypto-sha0-34	4.17	8.52	8.20

5. CONCLUSION AND FUTURE WORK

We have described a practical hybrid SAT solver that uses an application-specific FPGA-based BCP co-processor. The key to this co-processor architecture is our novel approach to transform Boolean logic into a compact memory structure to take advantage of the FPGA's high bandwidth and low latency on-chip BRAM. The modular design of the BCP co-processor can be scaled up to support hundreds of thousands of clauses.

Our results demonstrate speed-up over an optimized software-only BCP implementation by approximately 3.7 to 38.6 times. Currently, the non-BCP parts (decision, conflict resolving, and backtracking) take around 10% of running time in software. If BCP is sped up by the accelerator, the non-BCP modules become the system bottleneck. We believe the non-BCP parts can be further optimized and potentially run in parallel with the FPGA accelerator, a potential future direction for the SAT community. We are currently in the process of mapping the design to real hardware and integrating it with the software solver to do full system comparison, which will also enable thorough investigation of the BCP co-processor design using a multi-FPGA platform.

ACKNOWLEDGEMENTS

We would like to thank Chuck Thacker and Satnam Singh for their helpful comments and suggestions. We would also like to thank all of the anonymous reviewers for their valuable feedback.

6. REFERENCES

[1] L. Zhang and S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers," *Proc. of CAV 2002*, July 2002

[2] T. Suyama, M. Yokoo, H. Sawada, and A. Nagoya, "Solving Satisfiability Problems Using Reconfigurable Computing," IEEE Trans. VLSI Systems, vol. 9, no. 1, pp. 109-116, 2001

[3] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Using Configurable Computing to Accelerate Boolean Satisfiability," IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 18, no. 6, pp. 861-868, 1999

[4] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Solving Boolean Satisfiability with Dynamic Hardware Configurations," FPL 1998: 326-335

[5] M. Redekopp and A. Dandalis, "A Parallel Pipelined SAT Solver for FPGA's", Proc. of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Application, 2000

[6] M. Abramovici and D. Saab, "Satisfiability On Reconfigurable Hardware," *Proc. Intn'l. Workshop on Field-Programmable Logic and Applications*, Sept. 1997

[7] I. Skliarova and A.B. Ferrari, "A Software/Reconfigurable Hardware SAT Solver," IEEE Trans. Very Large Scale Integration (VLSI) Systems, vol. 12, no. 4, pp. 408-419, Apr. 2004

[8] I. Skliarova, A.B. Ferrari, "Reconfigurable Hardware SAT Solvers: A Survey of Systems," IEEE Transactions on Computers, vol. 53, issue 11, Nov. 2004, pp. 1449-1461.

[9] Y. Zhao, S. Malik, M. Moskewicz, and C. Madigan, "Accelerating boolean satisfiability through application specific processing," ISSS 2001, 244-249

[10] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *38th Design Automation Conference*, Las Vegas, June 2001

[11] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem Proving," Comm. ACM, no. 5, pp. 394-397, 1962

[12] N. Een and A. Biere, "Effective Preprocessing in SAT through Variable and Clause Elimination," SAT 2005

[13] C. P. Gomes, B. Selman, and H. Kautz, "Boosting Combinatorial Search through Randomization," AAAI 1998.

[14] J. Davis, Z. Tan, F. Yu, and L. Zhang, "Designing an Efficient Hardware Implication Accelerator for SAT Solving," SAT 2008

[15] I. Kuon and J. Rose, "Measuring the Gap between FPGAs and ASICs," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, NO. 2, Feb. 2007, pp. 203 - 215.

[16] "PCI Express Base 2.0 Specification," http://www.pcisig.com/members/downloads/specifications/pciexpress/PCI_Express_Base_Rev_2.0_20Dec06a.pdf

[17] Open FSB Initiative, Intel IDF, Spring 2007, Beijing

[18] HyperTransport Technology I/O link, AMD, 2001

[19] S. Kirkpatrick and B. Selman, "Critical Behavior in the Satisfiability of Random Boolean Formulae," Science, Vol. 264, pp. 1297-1301, May 27, 1994