FPGA Acceleration of Enhanced Boolean Constraint Propagation for SAT Solvers

Jason Thong and Nicola Nicolici
Department of Electrical and Computer Engineering, McMaster University, Canada
thongj@mcmaster.ca, nicola@ece.mcmaster.ca

***Abstract*— We propose a hardware architecture to accelerate boolean constraint propagation (BCP). Although satisfiability (SAT) solvers in software use varying search and learning strategies, BCP is a fundamental component and by far consumes the most CPU time. Our field-programmable gate array (FPGA) design uses on-chip SRAM to facilitate the acceleration of BCP. We discuss many insights to our innovative hardware memory layout, which is very compact and enables extremely fast BCP. It also supports multithreading to minimize the idle time in hardware and to fully utilize the multicore processor host. Additionally, many industrial SAT instances encode logic gates as constraints. We compact these to simultaneously reduce the hardware memory usage as well as speed up the computation (enhanced BCP). We implemented our enhanced BCP core and integrated it with a simple software SAT solver which communicates over PCI Express. Hardware performance counters show that a single processing engine is up to 4x faster than a state-of-the-art software SAT solver.**

I. Introduction

The boolean satisfiability problem (SAT) was one of the first NP-complete problems. Prior to the 1990s, SAT was arguably considered mostly a theoretical interest and mainly used to prove the intractability of new problems (by polynomial reduction). Although complete algorithms for solving SAT date back to the 1960s (e.g. DPLL [1]), much of the smartness in modern SAT solvers was developed in the late 1990s and early 2000s. This was when computation became more of a commodity rather than a privilege.

Among SAT algorithms based on DPLL, some noteworthy advancements include: GRASP [2] which introduced non-chronological backtracking, [3] which showed how to learn clauses from conflict analysis, and Chaff [4] which presented watched literal lists as well as an activity-based heuristic for choosing the next variable to assign.

Modern SAT solvers have greatly changed the picture for NP-complete problems. Today it is common to reduce a hard combinatorial problem to SAT so that it can be efficiently solved by any generic SAT solver. Using this approach, SAT has found itself at the heart of many applications including: electronic design automation (which often involves combinatorial optimization), bounded model checking, hardware and software verification, and cryptography. Further elaboration on such applications as well as their mapping to SAT are described in [5]. Verification is detailed in [6], [7].

Since the breakthrough of Chaff, many of the newer SAT solvers are variants which provide improvements by data structure optimization, or offer minor improvements by modifying the heuristics for which variable to assign next, learnt clause database management, restart policies, etc. Some widely used improvements include: luby restarts [8] which optimizes when to restart, and phase-saving [9] which caches whether a variable was last assigned true or false. The algorithmic advancement of SAT is slowing as further innovations beyond so much cumulative smartness become increasingly difficult.

A significant effort has also been placed on efficient boolean constraint propagation (BCP). In most modern SAT solvers, BCP typically consumes 80-90% of the CPU time (this seems to be the consensus within the SAT community, and we have also observed this behaviour).

BCP is the *fundamental pruning mechanism* of any DPLL-based SAT solver. Concisely speaking, as the SAT solver traverses the tree of all possible variable assignments, BCP enables it to prune off of unsatisfiable branches.

With such a large portion of the CPU time, many have proposed to accelerate BCP in hardware (see section II). Typically just BCP is accelerated, so a software host is needed. Advancements to implementing BCP are *orthogonal* to SAT algorithm development, as one can integrate the same BCP hardware with *any* DPLL-based software SAT solver.

In this paper, we propose a new field-programmable gate array (FPGA) architecture for accelerating BCP. We discuss many observations and insights which lead to our innovative hardware. We make no claim on advancing on SAT algorithms – our scope is strictly on the hardware architecture. We mostly focus on the design of a very compact SAT representation for hardware which also enables fast BCP.

The remainder of this paper is organized as follows. Prior work is discussed in section II. Section III examines the use of multithreading. Section IV illustrates the key insights which affect how BCP can be accelerated. This leads to our innovative and very compact memory layout as well as the BCP hardware architecture in section V. Section VI presents enhanced BCP, which operates on logic gate clauses. Finally, results are presented in section VII and we conclude in section VIII.

II. Prior Work

A. Older Hardware SAT Accelerators

A comprehensive survey of works from the late 1990s and early 2000s is provided in [10]. Many of these works used *instance-specific* hardware, so new hardware must be generated for each problem. In our opinion, this basically pushes the complexity of SAT into the CAD tool used for implementing the hardware. Most did not report run times of said CAD tools, and we suspect they were significantly longer than the hardware run times. We are not aware of any designs from the last decade that still use this approach.

978-1-4799-1071-7/13/$31.00 ©2013 IEEE

Many older non instance-specific designs made practical sense at that time when FPGAs were over an order of magnitude smaller in logic capacity, however they are unlikely to scale to today's large FPGAs. For example, [11] represents the SAT problem by storing a matrix of size $num_variables \times num_clauses$. This is practical for problems of tens of variables and hundreds of clauses, which would have been a reasonable problem size at that time. However, as the problem size grows, this matrix becomes increasingly sparse and the method itself less efficient.

B. Modern Hardware SAT Accelerators

From the mid 2000s onwards, the focus on accelerating BCP has shifted towards how to best store and process variable assignments and clauses *in the hardware's memory.* Many modern BCP accelerators are limited by either using slow memory or not making the most of fast memory in terms of the capacity.

A DRAM-based BCP engine was built in [12] to address the limited capacity of faster memory. They do solve the capacity issue, however DRAM (or any external memory) provides a limited amount of bandwidth to solve a fundamentally memory bound problem, as discussed in section IV-B. Interestingly, BCP results were not reported in [12].

A complete SAT solver is implemented in hardware in [13]. Moving decision making and conflict analysis into hardware eliminates the communication latency between a software host and the hardware accelerator (we address this in section III). Since they must also implement non-BCP computation in hardware, their hardware can only fit smaller problems (they only use on-chip memory, so capacity is an issue). Additionally, modern decision making and learnt clause database management involve things like memory allocation which is very cumbersome to implement efficiently in hardware. Consequently, these key components of a modern SAT solver are simplified in their hardware. Furthermore, any new advancements of these would be slower to integrate into hardware.

A radical approach is taken in [14], which developed a custom integrated-circuit cell just for BCP. As if SRAM is too slow, they store the SAT problem in registers. With BCP logic around the registers, their hardware is extremely fast. However, implementing an ASIC requires extremely large amounts of time and money, yet FPGAs typically have 1 to 2 orders of magnitude less registers than SRAM bits which would severely limit the supportable problem sizes.

C. Most Relevant Prior Work

The FPGA-based BCP accelerator in [15] and its follow up work [16] use on-chip memory to enable fast BCP. Among the prior work, this one is most similar to our proposed system. To briefly summarize their design, each newly assigned variable is broadcast to several parallel inference engines. They try to visit every clause that this variable occurs in at once, so each inference engine must determine which of these clauses(s), if any, it contains. This happens in parallel, so these engines use on-chip memory to avoid serialized access to external memory. After this, the variable assignments of those clause(s) are fetched to do BCP. Any newly implied variable is reported to a centralized location for broadcasting later.

We differ from [15], [16] in that we do not require a clause lookup. In our design, the incoming variable being assigned is a *hardware address* that points directly to what we need to access within the clause that contains this variable. This lack of translation is beneficial because:

1) Access is direct and therefore faster.
2) We do not store any clause translation table in hardware, so our memory layout is more compact, thus enabling larger SAT problem sizes.

We do not use a broadcast architecture. We start BCP locally, which is much faster than having to first report it to a centralized place. Although [15], [16] visits all clauses that contain a given variable at once, they need significantly more time to process one clause. To enable local BCP starts, and since we are faster per clause, we decided to visit these clauses sequentially. Our hardware is not purely serial, our work spreading mechanism is explained in section V-B.

In order to increase our hardware utilization, we propose to use two key techniques that [15], [16] did not consider:

1) Multithreading (section III)
2) Enhanced BCP (section VI)

We think these could be integrated into [15], [16], however an FPGA broadcast architecture will benefit minimally from multithreading due to the very limited broadcast bandwidth.

III. Multithreaded SAT

A. Advantages of Multithreading

The idea of parallelizing SAT to fully utilize multicore CPUs has been around for years. SAT competitions have included a parallel track since 2008. Even so, to the best of our knowledge, we are first to propose support for multithreading in hardware (as discussed in section III-C).

In addition to fully utilizing the software host, multithreading in a BCP hardware accelerator has many advantages:

1) It keeps the hardware busy (better utilization).
2) It hides the latency of host/FPGA communication.

As an example, consider what would happen if software BCP required 90% of the CPU time and hardware could accelerate it by 9×. The software solver would spend 10% of the original time doing non-BCP work and 10% of the original time waiting for hardware. Likewise the hardware would only be used 50% of the time. In this example, running 2 threads will keep both software and hardware busy.

Running more threads can facilitate load balancing between CPU cores and mask the communication overhead between software and hardware. As an example to illustrate this, with 3 threads, at any given point in time, one thread is being processed by software (e.g. conflict analysis), one thread is in hardware (BCP), and one thread is moving data around. Which thread is doing what task will rotate in time.

B. Parallel Software SAT Solvers

Parallel software SAT solvers typically employ one of the following strategies:

1) Assign n variables in all possible ways and distribute the 2^n subproblems to different threads.
2) Race several different serial SAT solvers against each other at the same time.

The first case uses divide and conquer – if any subproblem finds a satisfiable assignment, the original problem is satisfiable. In the second case above, by using orthogonal parameters for how to choose the next variable, when to restart, learnt clause database management, etc., on average we find a solution faster than with any one setting. Intuitively, different settings are better for different problems. This is the strategy used in ManySAT [17]. Further elaboration on parallel software SAT solvers is provided in [18].

From a BCP perspective, it does not matter which approach above is used so long as all of the serial SAT solvers (which collectively form a parallel solver) are solving the same problem. If we replicate the variable assignments and watched literal lists in hardware, we can integrate one multithreaded BCP accelerator with a parallel software SAT solver. To save hardware memory, clauses can be shared.

C. Hardware Multithreading

We believe the *previous lack of sufficiently large amounts of fast memory* is the dominant reason why prior works did not attempt multithreaded BCP in hardware. For SRAM-based BCP accelerators, the amount of memory was the limiting factor. Replicating the variable assignments (and possibly also the watched literal lists) was too large of an overhead. DRAM-based BCP accelerators can easily exhaust the external memory bandwidth, so a multithreaded system would ultimately time-share the access to this slow memory. Given the overhead of parallelization, such a system would not make practical sense. Conclusively, *in order to effectively implement multithreading, we must use on-chip memory and have a very compact memory layout.*

IV. KEY OBSERVATIONS FOR ACCELERATING BCP

A. A Sober Look at Present Technology and Trends

Many proposals for accelerating SAT in hardware emerged in the late 1990s and early 2000s, yet we do not see widespread adoption today. This does *not* imply that SAT is unsuitable for hardware – in fact it is now the opposite due to dramatic changes in computing technology since then.

BCP is a *fundamentally serial process* (as illustrated in section IV-B), so as CPU frequencies no longer increase due to power dissipation, we must look for new techniques to accelerate BCP. Some gain is achievable by multithreading, however it is limited due to shared memory bandwidth.

In recent years FPGAs have been closing the performance gap to ASICs. This gives hardware SAT a more competitive edge against pure software. In addition to CPUs not being clocked faster, FPGAs now contain a mix of soft programmable logic and hardened high performance logic (e.g. embedded memories, multipliers, high speed external interfaces, etc.). Every 18-24 months, Moore's Law has enabled FPGAs to double in logic capacity.

We expect the memory capacity (which is directly related to transistor count per die) to grow faster than the size of SAT problems, as SAT is NP-complete and pruning is only so smart. *This offers the opportunity to move what was previously stored in slower memory into now faster memory because there is enough faster memory available.* Software based BCP has and will likely continue to benefit from a larger CPU cache. More than half of the transistors in many modern CPUs are used for cache.

For FPGAs, this enables a design to improve performance simply by migrating from DDR2 to DDR3 DRAM, for example. Since SAT is a memory bound problem (see section IV-B), the biggest game-changer is bringing data on-chip, since the bandwidth is typically 2 orders of magnitude higher. *We believe we are nearing the threshold where it makes practical sense to do this for SAT.*

Others have also considered BCP acceleration using on-chip memory, such as [15], [16]. The largest FPGAs today have tens of megabits of embedded memory and it is very likely that this will continue to grow in accordance with Moore's Law in the foreseeable future.

As a final point, the largest FPGAs available now have millions of logic elements and thousands of embedded memories. It is likely that this much logic would be under utilized if the innermost loop of BCP required serialized access to slower external memory. In fairness, the prior works that used this approach only had significantly smaller FPGAs available and such designs made practical sense at the time.

B. The Computation Pattern of BCP

We assume the reader is familiar with the typical conjunctive normal form of SAT as well as the typical notation like $(A \vee \neg B) \wedge (B \vee C) \wedge (\neg C \vee D \vee \neg E)$.

In the simplest terms, BCP prunes unsatisfiable branches as the SAT solver traverses the tree of all possible variable assignments. As an example, if we decide to assign $A = false$, then the above clause $(A \vee \neg B)$ requires that $B = false$. This then implies $C = true$ due to $(B \vee C)$. At this point we cannot infer anything about D or E.

The computation pattern of BCP is as follows:

1) For each variable that is assigned, *fetch* a list of all clauses that contain this variable. This was improved to a smaller "watched literals" list in [4].
2) For each of these clauses, *fetch* the variable assignments. If all variables except for one are assigned such that the clause is not satisfied, we know we must assign the last unassigned variable to satisfy the clause.
3) As BCP implies new variables, repeat step 1 for each.

Notice that what is fetched in step 1 indicates what to fetch in step 2. Once we have the variable assignments, we apply the BCP rules and may imply a new variable. Eventually that new variable will later tell us what to fetch in its step 1. This access pattern is analogous to traversing a link list, as what we fetch now tells us where to look next.

This leads to some critical observations about BCP:

1) BCP is inherently a serial process.
2) BCP is memory bound.
3) The memory access pattern is random access, so read latency is extremely important.

The first point above comes from dependency analysis. In the above example, $A = false$ implied $B = false$ which in turn implied $C = true$, but until B was assigned it is impossible to know that C would be affected.

For the second point above, most of the work is in *fetching* data. The only math involved is in computing offsets (e.g. pointer arithmetic) and loop iterators.

Memory is randomly accessed because for arbitrary clauses in a SAT problem, there is no way to implicitly know:

1) Which clauses to inspect for a newly assigned variable.
2) Which variable assignments must be fetched for each clause inspected.

It is actually this lack of implicit knowledge that causes so much fetching. This can be evaded by using instance-specific hardware, however pushing the complexity of SAT into the CAD tool for implementing the hardware has proven futile.

Random access is inherently sensitive to read latency. In software, all one can do is optimize data structures. In hardware, in addition to this, which parts of the problem we decide to place in registers, on-chip memory, off-chip SRAM, and off-chip DRAM has a *significant* effect on the system performance.

In summary, *we need an underlying technology that facilitates low latency random access to maximize the acceleration of BCP*, hence our choice of on-chip SRAM.

C. Key Insights to Accelerate BCP in Hardware

We now present several key insights to address the limited amount of on-chip SRAM. This eventually leads to a *very compact representation for SAT without sacrificing the ability to process data extremely fast*.

Key Insight #1: When we assign a variable, we actually want the *variable assignments* of the clauses that this variable occurs in, not the clauses themselves.

Another way of looking at this is we want to *reorder* the memory access by dereferencing what matters the most first. We only need to access a clause itself if there is a BCP. Lazy execution saves bandwidth and processing.

Key Insight #2: Variable assignments only require 2 bits per variable, so multiple assignments can be placed in the same memory word enabling them to be read out all at once.

As a follow up to the first insight, packing all the variable assignments of one clause in the same word reduces the number of references needed for a newly assigned variable.

One property of SAT is that the clause size can be regulated. We can split large clauses by introducing dummy variables, for example $(A \vee B \vee C \vee D)$ is satisfied if and only if $(A \vee B \vee E) \wedge (\neg E \vee C \vee D)$ also is, where E is the dummy variable. Applied recursively, any SAT problem can be transformed into 3-SAT (all clauses size 3).

We need not bring clauses all the way down to size 3, as this can create an excessive number of dummy variables. Many FPGA embedded memories have 16 or 18 bits per location, giving a maximum clause size of 8 or 9 variables.

Key Insight #3: Given all the variable assignments of one clause at once, constraint propagation is just combinational logic and therefore extremely fast.

This is yet another good reason to pack the variable assignments together. In hardware it is easy to evaluate in parallel if any variable needs to be implied. There is no one single equivalent software instruction, so software would need several operations. Typically software uses if-else constructs to do BCP. Eventually some branches will be mispredicted, thus giving hardware BCP more advantage.

Key Insight #4: Favor implicit linking over explicit linking whenever possible. For example, it is better to walk down an array (which is implicitly linked by proximity in memory) rather than to traverse a link list (which explicitly stores where to look next every time).

This is beneficial because:

1) We save space by not storing pointers/addresses.
2) Accessing the next item in the list is typically faster since implicit linking depends on bandwidth whereas explicit linking depends on read latency. Notice that array access can be pipelined.

This guideline leads us to two potential memory layouts:

1) Use an array for each variable occurrence list (implicitly linked by proximity), which means we must explicitly link the clauses. This is shown in Figure 1.
2) Use an array for each clause, which means we must explicitly link the variables (Figure 2).

Figures 1 and 2 both represent the same SAT problem of $(A \vee B) \wedge (A \vee C) \wedge (A \vee B \vee C)$. Adding some flags to allow inverted variables is trivial and thus not shown.

The array-based variables memory layout has some major challenges. In Figure 1, array B should start immediately after array A so as to not waste any memory locations, however we would then need some other mechanism for adding a learnt clause containing variable A. Also, constraint propagation would have to gather the variables assignments

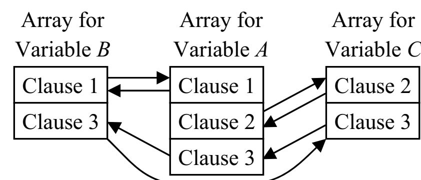

Fig. 1. Array-based variable occurrence lists require clauses to be linked.

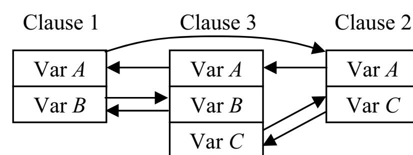

Fig. 2. Array-based clauses require explicitly linked variables.

over several clock cycles, which does not enable insight #3. For these reasons, we chose to use array-based clauses.

Key Insight #5: When explicit linking is required, favor localism so that addresses can be represented on fewer bits.

This can be regarded as a form of lossless compression. As an example, if the next item to access is within the same block of 4096 addresses as the current item, we only need 12 bits to reference it (plus a 1 bit flag to indicate it is a "local" link). Otherwise we would need to use, e.g. 32 bits to globally reference the next item.

V. Our Hardware BCP Engine

A. Our Compact Hardware Memory Layout for Fast BCP

The memory layout for one clause is presented in Figure 3. As discussed above in key insight #4, we use array-based clauses – each clause consumes a contiguous section of memory with no space wasted between the end of one clause and the start of the next. Clauses cannot cross a boundary of X addresses to facilitate using localized links on fewer bits (key insight #5). Our implementation uses $X = 4096$, however there is no restriction on X in general.

To facilitate fast BCP, for each clause, all of its variable assignments are stored in one memory location (key insight #3). A variable occuring in n clauses has n copies of its variable assignment distributed to each of the n clauses. This actually consumes *less* memory than using one single variable assignment, as we would then need a link on tens of bits at each clause to locate this one variable assignment. Simply storing the 2-bit variable assignment itself is smaller.

The multiple copies of each variable assignment must be kept synchronized. Upon updating a variable assignment, we must visit *every* clause the variable occurs in. We avoided using watched literals lists from [4], as they prevent enhanced BCP (section VI) and would also need to be replicated per thread (since these are links each on tens of bits, they are *much* more expensive to replicate than variable assignments). Finally, multithreading is supported by replicating the variable assignments (one copy per thread).

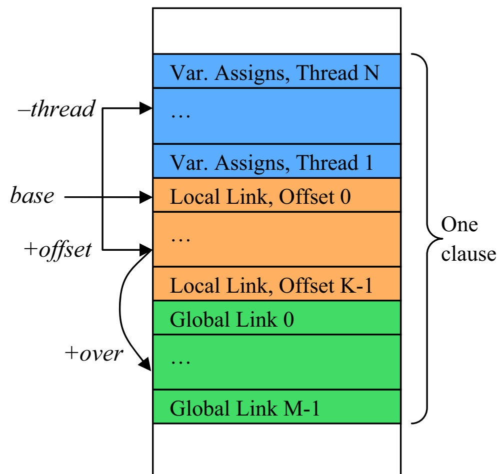

Fig. 3. Our proposed hardware memory layout for BCP.

In Figure 3, $offset$ refers to the position of a variable within the clause. For example, variable A is at offset 2 in the clause $(C \vee \neg B \vee A)$. Note the first offset is 0.

A "job" (the visiting one clause) is characterized by:

1) $base$: the address of offset 0 (where is the clause).
2) $thread$: which thread (note the first thread is 1).
3) $offset$: which variable in the clause.
4) $assignment$: are we deassigning, assigning true, or assigning false (2-bit value).

Subtracting $base - thread$ gives the location of all variable assignments for this clause and this $thread$ (the blue region in Figure 3). $offset$ indicates which 2 bit variable assignment we need to overwrite with the value of $assignment$. After doing this update, all variable assignments are written back to memory and are also sent to a constraint propagation unit to do BCP (as shown in Figure 6).

Adding $base + offset$ gives the location of the local link (the orange region in Figure 3). The memory contents here indicate where the next clause is that contains this variable. For example, if the current clause is $(C \vee \neg B \vee A)$ and $offset$ is 2 (so we are updating variable A), then the contents in the memory location $base + 2$ indicate where we can find the next clause that contains variable A.

Getting to the next clause of the same variable requires:

1) The next $base$ – where is next the clause.
2) The next $offset$ – which variable within the clause.
3) Is the next variable occurrence negated – if so we need to exchange assigning true/false.

The $thread$ stays the same. The next $offset$ and the 1 bit flag that indicates negation are always stored in the local link (orange region of Figure 3).

The next $base$ can be encoded in two ways. If it is within the same block of X (e.g. 4096) addresses, the local link simply contains a 12 bit local address. Otherwise we need a global address, which in our implementation is stored in 2 memory locations. In this case, the local link (orange region) stores some of the bits of this global address. The local link also stores $over$, which identifies which "overflow bin" (green region labeled as global link) contains the remaining bits of this global address.

We require N memory locations to store variable assignments for N threads. The number of local links (parameter K in the orange section of Figure 3) is always equal to the number of variables in the clause, so smaller clauses consume less memory. We may have fewer global links (parameter M in the green section) if many variables in this clause have their next clause within the same block of X addresses. Thus $K \geq M \geq 0$. Note that $M = 0$ can happen but $K \geq 1$.

Clauses with up to 8 variables are supported. The clause size is not stored anywhere in hardware. It is illegal to specify an $offset$ larger than the clause size. Doing BCP on $(A \vee B)$ is identical to doing BCP on $(A \vee B \vee Y \vee ... \vee Z)$ where $Y ... Z$ were permanently and deliberately assigned false.

For each variable, the linked list of clauses is cyclic. We have visited all clauses when the $assignment$ in memory (which we wrote earlier) matches the incoming one. A similar technique is used to detect conflicts.

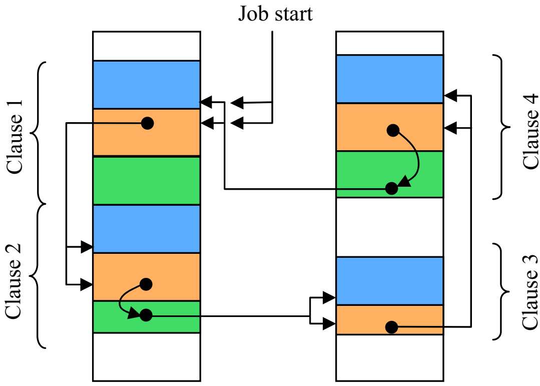

Fig. 4. An example traversal for a variable that occurs in 4 clauses.

Figure 4 illustrates the access pattern if we assign a variable that occurs in 4 clauses. At each clause, we access both the variable assignments (the blue section) and the link to the next clause (if local then only the orange section, otherwise a globally addressable link requires a second access to into the green section). In clause 3, all variables have their next occurrence within the same block, hence there are no global links here. The clause sizes can vary (note the different heights of each orange section) and the offset can vary (position within each clause). The thread always stays the same. In this example, we always access the variable assignments of thread 1 (bottom of the blue section). Upon visiting clause 1 again (after clause 4), the existing variable assignment matches the incoming one, hence we are done.

B. Full System Architecture

Our proposed full system architecture is illustrated in Figure 5. Many distributed PEs (processing engines) each contain a memory block of, e.g. 4096 addresses. The PEs communicate to each other via a NOC (network-on-chip). The software SAT solver does everything except for BCP and communicates with hardware over PCI Express.

To minimize latency and thus maximize BCP acceleration, we must process clauses and variable assignments *as close to the memory interface as possible*. Since embedded memories are distributed throughout an FPGA, we likewise distribute the processing. This enables massive parallelism, as different PEs can simultaneously process different clauses.

Multithreading helps to keep many PEs busy. Work spreading of BCP also helps this. For example, suppose $A = true$ implies $B = true$ and $C = true$. If B and C are in different PEs, we can inspect some clauses that contain B at the same time as we inspect some clauses that contain C.

Figure 6 shows the structure of each PE. The NOC provides a layer of abstraction which *separates communication from processing*. At this boundary, there is an input queue for jobs to be processed here as well as an output queue for data ready to leave (a job or reporting BCP to software). Jobs are started by software and a job can be passed from one PE to another, which happens if we need a global link.

As explained in section IV-C, given all variable assignments in a clause at once, constraint propagation detects in parallel if we can imply any new variable. If so, constraint

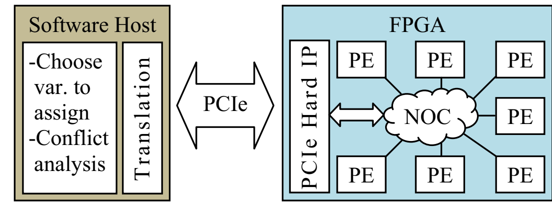

Fig. 5. Our proposed system architecture for accelerating BCP in hardware.

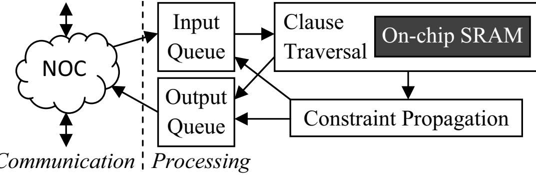

Fig. 6. The structure of each PE (processing engine) is shown on the right. The NOC (network-on-chip) provides communication between PEs.

propagation reports which $offset$ in this clause is the newly implied variable. Since all of the clauses that contain this newly implied variable are accessed by a *cyclic link list*, we can visit all of them by starting *anywhere* in the list. Hence, we simply start a new job at the same clause that caused this BCP but with whatever $offset$ was reported by constraint propagation (this job is pushed into the input queue). We also report it to software via the output queue.

Our hardware BCP does not require a clause lookup because it is completely implicit. If A implies B, we can very quickly start inspecting clauses that contain B. This is achieved by representing BCP as a *hardware address and an offset* (recall how jobs were characterized in section V-A).

A translation is required for software to know which variable this BCP hardware address and offset represent. Likewise, if software wants to assign a variable, it must translate it into a hardware address and offset. This translation is currently implemented in software.

We are currently investigating different network topologies and arbitration policies for the NOC, which is beyond the scope of this work and will be discussed in future work. We currently connect the PCIe interface directly to one PE.

C. BCP State Machine

Up to four memory accesses are needed per clause visit:

1) Read the local link (the orange region in Figure 3).
2) Read the variable assignments (blue region).
3) Read the global link (green region) if necessary.
4) Write back the updated variable assignments.

Accesses 3 and 4 above must come after 1 and 2. FPGAs typically have dual-port embedded memories, so we could visit one new clause every 2 clock cycles, however this resulted in long combinational paths. By adding pipelining, we were able to nearly double the clock frequency. Although we visit the next clause every 4 clocks, the *absolute* time is nearly the same but now the throughput is twice as much.

Our implemented hardware clause traversal state machine is shown in Figure 7. Pipelining adds read latency, so to fully utilize the bandwidth, we use 2 independent “execution strands” which time-share the memory.

Clock		0	1	2	3	4	5	6	7	8	9	10	11
Execution strand 0	Addr1	Read local link 0			Read global link 0	Read local link 1			Read global link 1	Read local link 2			Read global link 2
	Addr2	Read var assign 0			Write var assign 0	Read var assign 1			Write var assign 1	Read var assign 2			Write var assign 2
Execution strand 1	Addr1			Read local link 0			Read global link 0	Read local link 1			Read global link 1	Read local link 2	
	Addr2			Read var assign 0			Write var assign 0	Read var assign 1			Write var assign 1	Read var assign 2	

Fig. 7. Our pipelined hardware clause traversal state machine has two execution strands to fully utilize the dual port embedded memory bandwidth.

VI. Enhanced BCP

It is commonly known that many "industrial" SAT instances encode logic gates as boolean clauses, typically using the Tseitin encoding [19]. This naturally arises in SAT applications such as formal verification, since all digital system are based on *logical constructs* regardless of whether it is software or hardware.

In SAT problems derived from logical constructs, our hardware can operate directly on "logic gate clauses". Recall key insight #3 from section IV-C: given all variable assignments in a clause at once, constraint propagation is simply combinational logic. We can extend this idea by also supplying a clause type to indicate which specific type of custom constraint propagation to apply.

For a collection of boolean clauses that specify a logic gate, in hardware we only need to store one representative clause which contains all of the variables as well as the clause type. When our hardware constraint propagation unit sees the clause type, it *derives all of the underlying boolean clauses* and applies BCP to each of these in parallel.

For example, if the constraint propagation unit is given:

$$(A \vee B \vee C),\ type = \text{``}AND\text{''}$$

it derives all of the underlying boolean clauses:

$$(\neg A \vee B) \wedge (\neg A \vee C) \wedge (A \vee \neg B \vee \neg C)$$

which is the Tseitin encoding for the AND gate $A = B\&C$. If we assigned $A = true$, constraint propagation implies both $B = true$ and $C = true$.

Our hardware BCP is *fundamentally boolean* and fully supports pure boolean SAT. However, if there *happens to exist* a more compact representation in a SAT problem, we exploit it because:

1) Hardware processing is faster since we inspect the equivalent of several boolean clauses at once (3 clauses in this example).
2) It enables support for larger problem sizes in the same amount of hardware memory.
3) It may enable new propagations which are not immediately visible in the boolean domain.

As an example to illustrate the third point, suppose we allow a 1-bit half adder clause type consisting of 4 variables:

$$(A \vee B \vee C \vee D),\ type = \text{``}half\ adder\text{''}$$

Let the carry be $C = A\&B$ and let the sum be $D = A \oplus B$. The set of underlying boolean clauses are:

$$(\neg A \vee \neg B \vee \neg D) \wedge (A \vee B \vee \neg D) \wedge (A \vee \neg B \vee D) \wedge$$

$$(\neg A \vee B \vee D) \wedge (\neg C \vee B) \wedge (\neg C \vee A) \wedge (C \vee \neg B \vee \neg A)$$

Assigning $D = true$ does not imply anything, as D does not occur in a size 2 clause. However, we know that if the sum of a half adder is 1, the carry must be 0. Hardware constraint propagation accounts for this by adding the clause $(\neg D \vee \neg C)$ to the set of underlying boolean clauses.

A method for extracting gates from boolean clauses is presented in [20]. As the complexity of the combinational logic for constraint propagation has increased, we now use a 2-stage pipeline to maintain a fast clock. Our current implementation supports AND and XOR clauses with up to 8 variables. Our memory has 18 bits per location, thus leaving 2 bits for the clause type. Inverting a variable comes for free in SAT, so AND also represents NAND, NOR, and OR. As future work, we plan to support additional clause types, such as single bit 2:1 muxes, half adders, and full adders.

VII. Results

Our *real life* hardware implementation (not simulation) features one fully functional PE (processing engine) that is directly connected to the PCI Express interface. The PE contains a pipelined clause traversal engine (see section V-C for the state machine) as well as a constraint propagation unit that understands boolean, AND, and XOR clauses (from section VI). As stated in section V-B, the NOC is under development and its design is beyond the scope of this paper, hence our limitation of implementing only one PE.

We integrated our BCP core with a simple C++ software SAT solver along with a Linux PCIe driver, both of which we wrote. We have tested the *correctness* of multithreading in our hardware (in simulation and in system), however we leave it as future work to fully integrate our hardware with a state-of-the-art parallel SAT solver. The acceleration of BCP is *orthogonal* to SAT algorithmic development, hence the software host just needs to *exercise* our BCP hardware to demonstrate our contribution.

Our design runs at 250 MHz on the Terasic DE4-230 board (Altera Stratix IV EP4SGX230 FPGA). Our PE uses an embedded memory with 4096 addresses and 18 bits per location. Resource estimates indicate we can replicate this 150 times to exhaust the on-chip memory and still use less

TABLE I

SUMMARY OF OUR RESULTS FOR ACCELERATING BCP IN HARDWARE.

	Boolean clauses only			Mixed clauses		
Variables	225	200	200	500	525	450
Clauses	500	500	500	300	500	500
Max Clause Size	6	4	3	6	4	4
% SAT	87.1	47.5	12.8	92.9	44.2	4.1
% UNSAT	12.3	52.5	87.2	2.3	48.7	95.4
Total BCPs ($\times 10^6$)	1349	716	9	8023	18303	1300
Average clocks/BCP	46.7	28.0	21.3	22.1	13.5	12.9
BCPs/second ($\times 10^6$)	5.4	8.9	11.7	11.3	18.5	19.4
Speedup vs software	1.1×	1.8×	2.3×	2.3×	3.7×	3.9×

than half the LUTs and registers (leaving plenty of space for the NOC). This would support problems of approximately 50,000 variables and 50,000 clauses. The largest FPGAs today have more than 5× our memory and would support problems with 250,000 variables and clauses.

We can support even larger SAT problems if they contain logical constructs. Using enhanced BCP (section VI), we only store one representative clause for all of the underlying boolean clauses (e.g. a set of boolean clauses that describe an AND gate). We typically obtain a 3× to 4× compression ratio by converting the same boolean SAT problem into enhanced BCP for problems rich in logical constructs.

To fit 50,000 clauses in $4K * 18 * 150 = 11M$ bits of memory, we average about 28 bytes (or seven 32-bit integers) per clause. This includes the variable assignments, clauses, and the variable occurrence lists which are needed to traverse the clauses. We restructured and compacted how the SAT problem is represented in hardware mostly by exploiting localism and implicit linking.

Table I summarizes our results. Timing was obtained from hardware performance counters, which exclude the PCIe communication overhead, as this can be hidden as explained in section III-A. We randomly generated 1000 SAT problems in each of the 6 categories with uniformly distributed clause sizes between 2 and Max Clause Size inclusive. Mixed clauses contain one third each of boolean, XOR and AND clauses. As expected, the gains are larger with enhanced BCP.

We adjusted the problem characteristics to show benchmarks that are mostly SAT, balanced, and mostly UNSAT. More constrained problems offer more BCP, hence the improvement in results for more UNSAT problems. A 5 minute timeout was used in the software host (as we just need to *exercise* BCP), so %SAT+%UNSAT may be less than 100.

The speedup over software was approximated by using the typical 5 million BCPs/second performance of Minisat [21], a state-of-the-art SAT solver. Obviously this number varies with different CPUs (we used an Intel Core i7 980, 3.33 GHz, 12 MB cache). Although more recent solvers have improved SAT strategies, many like Glucose [22] are built on-top of Minisat and thus share the same BCP implementation.

The only reason we are limited to small problem sizes is because the NOC is currently under development. The design of an effective NOC is orthogonal to and beyond the scope of this paper. As detailed in section IV-C, our key insights prove to be effective even without the NOC, as our limited implementation still outperforms software BCP.

VIII. CONCLUSION

We have proposed a new FPGA-based accelerator for BCP. We presented several key insights which lead to our very compact representation of SAT in hardware. Our memory layout does not sacrifice any ability to process data extremely quickly. Our system is poised to further accelerate BCP with the future integration of a NOC and a multithreaded software SAT solver host.

REFERENCES

[1] M. Davis, G. Logemann, and D. W. Loveland, “A machine program for theorem-proving,” *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[2] J. P. M. Silva and K. A. Sakallah, “Grasp - a new search algorithm for satisfiability,” in *ICCAD*, 1996, pp. 220–227.

[3] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik, “Efficient conflict driven learning in boolean satisfiability solver,” in *ICCAD*, 2001, pp. 279–285.

[4] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: Engineering an efficient sat solver,” in *DAC*. ACM, 2001, pp. 530–535.

[5] J. Marques-Silva, “Practical applications of boolean satisfiability,” in *WODES*, 2008, pp. 74–80.

[6] A. Gupta, M. K. Ganai, and C. Wang, “Sat-based verification methods and applications in hardware verification,” in *SFM*, M. Bernardo and A. Cimatti, Eds., vol. 3965. Springer, 2006, pp. 108–143.

[7] M. R. Prasad, A. Biere, and A. Gupta, “A survey of recent advances in sat-based formal verification.” *STTT*, vol. 7, no. 2, pp. 156–173, 2005.

[8] J. Huang, “The effect of restarts on the efficiency of clause learning,” in *IJCAI*, M. M. Veloso, Ed., 2007, pp. 2318–2323.

[9] K. Pipatsrisawat and A. Darwiche, “A lightweight component caching scheme for satisfiability solvers,” in *SAT*, ser. Lecture Notes in Computer Science, J. Marques-Silva and K. A. Sakallah, Eds., vol. 4501. Springer, 2007, pp. 294–299.

[10] I. Skliarova and A. d. B. Ferrari, “Reconfigurable hardware sat solvers: A survey of systems,” *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1449–1461, Nov. 2004.

[11] ——, “A software/reconfigurable hardware sat solver,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 12, no. 4, pp. 408–419, Apr. 2004.

[12] L. Haller and S. Singh, “Relieving capacity limits on fpga-based sat-solvers,” in *FMCAD*, R. Bloem and N. Sharygina, Eds. IEEE, 2010, pp. 217–220.

[13] M. Safar, M. W. El-Kharashi, M. Shalan, and A. Salem, “A reconfigurable, pipelined, conflict directed jumping search sat solver,” in *DATE*. IEEE, 2011, pp. 1243–1248.

[14] K. Gulati, M. Waghmode, S. P. Khatri, and W. Shi, “Efficient, scalable hardware engine for boolean satisfiability and unsatisfiable core extraction.” *IET Computers and Digital Techniques*, vol. 2, no. 3, pp. 214–229, 2008.

[15] J. D. Davis, Z. Tan, F. Yu, and L. Zhang, “A practical reconfigurable hardware accelerator for boolean satisfiability solvers,” in *DAC*, L. Fix, Ed. ACM, 2008, pp. 780–785.

[16] ——, “Designing an efficient hardware implication accelerator for sat solving,” in *SAT*, ser. Lecture Notes in Computer Science, H. K. Büning and X. Zhao, Eds., vol. 4996. Springer, 2008, pp. 48–62.

[17] Y. Hamadi, S. Jabbour, and L. Sais, “Manysat: a parallel sat solver,” *JSAT*, vol. 6, no. 4, pp. 245–262, 2009.

[18] S. Holldobler, N. Manthey, V. H. Nguyen, J. Stecklina, and P. Steinke, “A short overview on modern parallel sat-solvers,” in *ICACSIS*, 2011, pp. 201–206.

[19] G. S. Tseitin, “On the complexity of derivation in the propositional calculus,” *Zapiski nauchnykh seminarov LOMI*, vol. 8, pp. 234–259, 1968, english translation of this volume: Consultants Bureau, N.Y., 1970, pp. 115–125.

[20] Z. Fu and S. Malik, “Extracting logic circuit structure from conjunctive normal form descriptions,” in *VLSI Design*. IEEE Computer Society, 2007, pp. 37–42.

[21] N. Eén and N. Sörensson, “An extensible sat-solver,” in *SAT*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., vol. 2919. Springer, 2003, pp. 502–518.

[22] G. Audemard and L. Simon, “Predicting learnt clauses quality in modern sat solvers,” in *IJCAI*, C. Boutilier, Ed., 2009, pp. 399–404.