HW-BCP: A Custom Hardware Accelerator for SAT Suitable for Single Chip Implementation for Large Benchmarks

Soowang Park
Electrical and Computer Engineering
University of Southern California
Los Angeles, CA, USA
soowangp@usc.edu

Jae-Won Nam
Seoul National University
of Science and Technology
Seoul, South Korea
jaewon.nam@seoultech.ac.kr

Sandeep K. Gupta
Electrical and Computer Engineering
University of Southern California
Los Angeles, CA, USA
sandeep@usc.edu

Abstract

Boolean Satisfiability (SAT) has broad usage in Electronic Design Automation (EDA), artificial intelligence (AI), and theoretical studies. Further, as an NP-complete problem, acceleration of SAT will also enable acceleration of a wide range of combinatorial problems.

We propose a completely new custom hardware design to accelerate SAT. Starting with the well-known fact that Boolean Constraint Propagation (BCP) takes most of the SAT solving time (80-90%), we focus on accelerating BCP. By profiling a widely-used software SAT solver, MiniSAT v2.2.0 (MiniSAT2) [1], we identify opportunities to accelerate BCP via parallelization and elimination of von Neumann overheads, especially data movement. The proposed hardware for BCP (HW-BCP) achieves these goals via a customized combination of content-addressable memory (CAM) cells, SRAM cells, logic circuitry, and optimized interconnects.

In 65nm technology, on the largest SAT instances in the SAT Competition 2017 benchmark suite, our HW-BCP dramatically accelerates BCP (4.5ns per BCP in simulations) and hence provides a 62-185x speedup over optimized software implementation running on general purpose processors.

Finally, we extrapolate our HW-BCP design to 7nm technology and estimate area and delay. The analysis shows that in 7nm, in a realistic chip size, HW-BCP would be large enough for the largest SAT instances in the benchmark suite.

Keywords

Custom hardware, SAT, BCP, von Neumann machine, CAM

1 Introduction

Boolean Satisfiability (SAT) problem is a problem which determines if there exists an assignment of values to variables that satisfies a given Boolean formula. SAT solvers are widely used in various domains, especially in Electronic Design Automation (EDA) to test and verify hardware/logic designs. SAT solvers are also heavily used in AI, theorem proving, and so on.

Modern software SAT solvers are very efficient at solving large and difficult problem instances in practical runtimes. Most solvers are based on Davis-Putnam-Logemann-Loveland (DPLL) algorithm [2], heuristic local search [3], and conflict-driven clause learning (CDCL) [4]. DPLL approach tries a variable assignment, backtracks if there is a conflict, and repeats this process until a given formula is satisfied. DPLL performs Boolean Constraint Propagation (BCP), a process that finds an unsatisfied clause (if any) where all variables are assigned false except one. This one unassigned variable must necessarily be true. Recent software SAT solvers use additional methods to efficiently prune the decision tree, including a function that determines the level of decision to backtrack to. They also use CDCL, i.e., a clause learning step that identifies a new clause based on the conflict information. MiniSAT v2.2.0 (MiniSAT2) [1] was implemented with these advanced techniques and constitutes the basic structure of many leading software SAT solvers.

Performance of MiniSAT2 [1] is limited due to overheads of von Neumann machines. First, even for running a simple task, the fetch-decode-execute cycle is required. Second, due to the fact that BCP is memory bounded [5], it requires lots of table lookups and sustained memory accesses and hence high data movement overheads.

Gulati et al. showed a full implementation of a SAT solver in ASIC [6]. Clauses are partitioned into multiple banks, where clause/variable ID is used as row/column address. Each clause cell contains the value of a literal (0, 1, or x) and logic circuits for implication. It achieved considerable speedup on some SAT instances, but is not scalable to large SAT instances, due to the limitation of the addressing mechanism. They also demonstrated a similar design in FPGA [7]. However, to fit into small-sized memory on FPGA, the original instance is grouped into multiple bins and loaded/solved sequentially. Considerable bin-swapping overhead limits performance improvement.

Davis et al. proposed BCP accelerators on FPGA (FPGA-BCP) [8]. By implementing tree walk to maximize the utility of limited capacity of FPGA Block RAMs (BRAM), an efficient mechanism is developed. The original clauses are divided into 2^p groups so that in each group a specific variable appears at most once. To implement clause index tree walk, variable ID (k-bit) is divided into k/m chunks, and a multi-step index computation is used to find the clause. Each clause group can perform the tree walk to accelerate BCP. This design is limited by FPGA BRAM capacity. They achieve 6.1x speedup over MiniSAT2 [1] for SAT benchmark instances with up to 64K clauses.

Thong et al. [5] proposed a memory architecture that uses variables as addresses of hardware memory specially designed for multithreading. This design requires large memory and complex Network-on-Chip (NoC) between processing elements. Hence, their actual implementation is for instances with hundreds of clauses. Other approaches that implement fast SAT accelerators are also able to handle only small-sized instances [9, 10].

Among the studies mentioned above, we focus on FPGA-BCP [8] due to its scalability. Multiple clock cycles are required to complete a BCP operation, due to multiple table lookups for tree walk and multiplexing. We verified that memory used and latency are both optimized when the chunk size (m) is 4. We also profiled benchmark

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPDAC '21, January 18–21, 2021, Tokyo, Japan
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-7999-1/21/01…$15.00
https://doi.org/10.1145/3394885.3431413

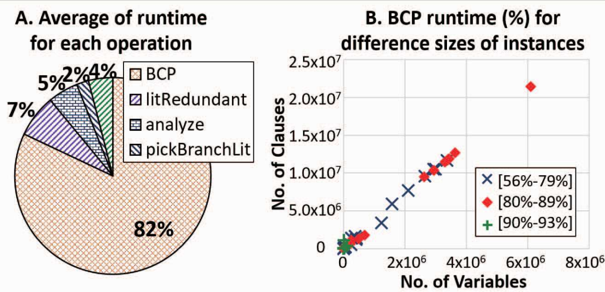

Figure 1: A. Execution profile for MiniSAT2 [1]: % CPU time required by key functions, average for 70 instances, B. Total runtime devoted to BCP for SAT instances with various sizes.

instances to derive the distributions of the numbers of clauses in which variables appear. This profile shows that there is no speedup for numbers of inference engines > 64 (i.e., for $p > 6$). Using above, for their study, which uses SAT instances with ≤ 64K clauses and a 65nm FPGA with a 5ns clock, the latency for each BCP operation is 10 clocks cycles and hence 50ns.

We propose a custom hardware design for BCP (HW-BCP), a co-processor that replaces software-BCP and accelerates MiniSAT2 [1]. Since BCP only performs Boolean operations once the clauses and variable values are identified, we design a fully parallel architecture using content-addressable memory (CAM) to eliminate table lookups and data movement, which is completely different from all above existing designs. We demonstrate that our design is scalable to the largest SAT benchmark instances for a realistic chip area, and provides significant acceleration via complete parallelization. (Specifically, in Section 4.3 we demonstrate this via a detailed comparison of performances of FPGA-BCP and our HW-BCP.)

2 Background: Profiling of MiniSAT2

Due to its high performance, MiniSAT2 [1] is used in this study as a target software SAT solver.

2.1 Focus on BCP

Generally, BCP operations take 80-90% of total CPU time of modern software SAT solvers like MiniSAT2 [1]. To better understand and characterize the instance-to-instance variations in the percentage of run-time required for BCP, we profiled SAT Competition 2017 benchmark suite using MiniSAT2 [1]. Whereas easy SAT instances are solved in a few seconds, hard ones take more than several hours.

We selected 70 instances that have medium-difficulty (total runtime in the 10-5000 sec range) for our profiling and show the summary in Fig. 1A. We confirmed that the average of total BCP runtime is 82% of the total runtime for these SAT instances. Further, on our system, the average runtime per BCP instance is 500ns, which corresponds approximately to 1000 clock cycles. Fig. 1B reports the range of total BCP time across benchmarks and shows that, regardless of instance sizes (number of variables and number of clauses), the average of total BCP time falls in the 56-93% range. Further, while small-sized instances have a tendency to have more than 90% of runtime devoted to BCP, generally speaking, BCP time is a dominant part and independent of the instance size. Hence, we focus on BCP as the target for hardware acceleration.

2.2 Opportunities for Acceleration

BCP is used in MiniSAT2 [1] and in most other software SAT solvers. Each instance of BCP starts with the SAT algorithm assigning a specific value to a particular variable. BCP must identify all clauses where the assigned variable appears, plug in the assigned value, compute the value of the clause, and generate and return the values of key signals. If this assignment causes a conflict in any of the clauses, BCP must return the backtrack signal which the SAT algorithm must use to reverse the previous decision. On the other hand, if in any clause, all-but-one variables are assigned specific values but the clause is still not satisfied, it is necessary that the sole unassigned variable is assigned a value that causes the clause to be satisfied. This is called a *unit propagation*, and its identification causes BCP to return a unit-propagation (UP) signal, along with the ID of the variable and the necessary value. In summary, BCP is the procedure that identifies all the variable value assignments that become necessary when the SAT algorithm assigns a specific value to a variable.

As is clear from above, software BCP has long serial execution [5] on von Neumann machines. While several researchers [8] have parallelized BCP at coarse-grain, however, each BCP instance is essentially executed serially in the manner summarized above.

The key challenge to parallelizing BCP beyond above in multi-core von Neumann systems is that each time BCP is called, it must visit the relevant parts of all the large data structures in SAT. Specifically, each BCP call necessitates visits to all the memories shown in Fig. 2. To identify all clauses where the assigned variable is used, it accesses global clause memory to retrieve one clause at a time. To evaluate the value of each clause, it needs the value of each variable in the clause. For this, it accesses global literal value memory, where a literal is a variable with its polarity (e.g., x_i or $\bar{x_i}$). This makes BCP a critically memory bounded process [5]. When the size of the SAT instance is large, entire clause information cannot be held in small-sized on-chip memory (SRAM caches), and hence requires the use of full memory hierarchy from cache to main memory (DRAM). Thus, extremely high memory latencies caused by cache misses become performance bottlenecks for the software SAT solvers.

To accelerate BCP in new ways, we started by conducting detailed analysis of the BCP algorithm and data structure used in MiniSAT2 [1]. We first carried out qualitative analysis to derive a symbolic expression for BCP runtime complexity. We then complemented this with profiling to incorporate the actual runtime information into our symbolic expressions and used this to identify the key performance bottlenecks and completely new hardware designs to accelerate BCP.

The BCP implementation in MiniSAT2 [1] uses advanced approaches beyond the simple summary above. Specifically, to resolve the memory bottleneck, two-watched-literal (2WL) [11] and blocking literal [12] were developed to reduce the number of clauses to be observed by monitoring the activities of only two literals in each clause, which substantially improved performance of software SAT solvers. To implement the 2WL scheme, Watched List Table (WLT) and Watched List (WL) are required, as shown in Fig. 2. WLT is a table containing pointers to watched lists for each literal and the list sizes. Let the average size of watched list be denoted by α. Each WL, which is retrieved by directly indexing WLT with a literal, is a two-column table; clause references (*cref*s) and *blocker*s. *cref* is one of indices of associated clauses to global clause memory. *blocker* is a copied literal from the clause, which is used to reduce the probability of accessing the clause memory. This takes advantage of the observation that we don't need to visit the clause if one of its literals is known to be assigned true and thus the clause is already satisfied. This is why another level of indirection, WL, is inserted to minimize

Algorithm 1 Boolean Constraint Propagation

```
1:  procedure BCP
2:      p ← Enqueue(Queue)                         ▷ Enqueue a literal (lit) from BCP Queue
3:      wl ← WL(p̄)                                 ▷ WL(p̄): Watched list for p̄
4:      for i = wl; i < wl + α; i++ do              ▷ α is no. of clauses where p̄ is watched; to be parallalized
5:          blocker ← wl->blocker                   ▷ blocker: a copied lit from the cluase
6:          if value(blocker) == TRUE then          ▷ Check blocker
7:              continue                            ▷ Line 6, β0 cycles; mostly von Neumann overhead
8:          else                                    ▷ ε1, probability this branch reached
9:              cref ← i->cref
10:             c ← ca[cref]                        ▷ ca: global clause memory
11:             if c[0] == p̄ then
12:                 swap(c[0], c[1])                ▷ Make sure the clause that the second lit is p̄
13:             first ← c[0]
14:             if value(first) == TRUE then        ▷ Lines 9-14, β1 cycles; von Neumann overhead
15:                 blocker ← first
16:                 continue
17:             else                                ▷ ε2, probability this branch reached
18:                 for j = 2; j < γ'; j++ do       ▷ Find a new lit to watch (γ' no. of lits in the clause)
19:                     lit ← c[j]                  ▷ γ, actual no. of visits considering early termination (γ ≪ γ')
20:                     if lit != FALSE then        ▷ Check lit, β2 cycles; von Neumann overhead
21:                         swap(c[1], lit)
22:                         remove(c, WL_p̄)         ▷ This clause is not in the watched list of p̄ anymore
23:                         goto NextClause
24:                 if value(first) == FALSE then   ▷ New watch not found
25:                     return cref                 ▷ Line 24, ε3, probability this branch reached
26:                 else                            ▷ Line 24, β3 cycles; von Neumann overhead
27:                     unitAssignment(first, cref)
```

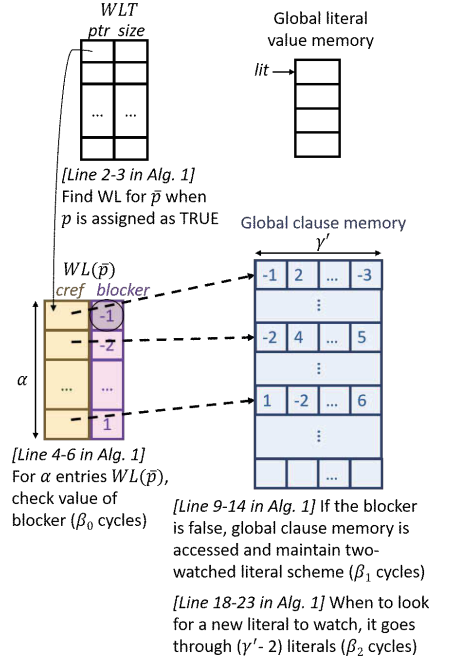

Figure 2: BCP algorithm and data structure.

the frequency of access to the global clause memory. Even with the above two very effective schemes, modern software SAT solvers like MiniSAT2 [1] still spend 80-90% of total CPU time on BCP.

As our goal is to fully parallelize BCP at very fine-grain, we design our custom hardware in ASIC and thus have complete freedom to maximize parallelism, specifically for MiniSAT2 [1]. To identify the opportunities for hardware acceleration, we analyze BCP algorithm's complexity and memory accesses qualitatively.

As shown in Alg. 1 in Fig. 2, BCP algorithm starts when a literal, p, is assigned true. MiniSAT2 [1] uses the 2WL scheme where the clauses containing the watched literal just assigned false needs to be updated. Thus, a WL for $\bar{p}$ is retrieved from WLT as shown in Fig. 2. Since the size of WL is α, α-size iterations are required to check whether each *blocker* of the associated clauses is true or not. For each time to get the value of the literal, global literal value memory (shown in Fig. 2) must be accessed. Since checking *blocker* (line 6 in Alg. 1) is performed for all the related clauses being watched, this becomes a dominant part of performance bottlenecks.

If *blocker* is not true, then BCP algorithm must visit global clause memory (lines 9-10 in Alg. 1). Thus, we denote a probability that this branch is reached, by ϵ_1. After getting the header of the clause by directly indexing with *cref* on global clause memory, a preliminary job to maintain the 2WL structure (the second literal is $\bar{p}$) is executed (lines 11-13 in Alg. 1). Then, BCP algorithm tries to check whether the first watched literal is already true (β_1 cycles; lines 9-14 in Alg. 1). If it is true, BCP algorithm does not need to examine this clause anymore. Thus, *blocker* is set to this first literal and skip to the next clause. If it is not true (probability, ϵ_2, this branch is reached), non-watched literals must be examined serially to find a new watched literal (the clause has γ' literals). If there is a literal not assigned false, the 2WL structure is updated with this literal and skip to the next clause (β_2 cycles; lines 18-23 in Alg. 1). Since there is a early termination of this loop (line 18 in Alg. 1), the actual number of visits in this loop, γ, is much less than γ' ($\gamma/\gamma' \simeq 0.1$). If a non-false literal is not found (probability, ϵ_3, this branch is reached), this clause has either an UP or a conflict, which is determined by checking whether the first literal is false (β_3 cycles; line 24 in Alg. 1). If it is false, this clause cannot be satisfied and BCP algorithm returns *cref* to the SAT algorithm to deal with the conflict. If it is non-false (unassigned), it causes the UP and thus this literal must be assigned true.

Overall, the number of cycles taken on von Neumann machine for BCP algorithm, h, is approximately represented as follows:

$$h = \alpha(\beta_0 + \beta_1\epsilon_1 + \beta_2\gamma\epsilon_2 + \beta_3\epsilon_3). \quad (1)$$

Eq. (1) gives a qualitative view how BCP is organized in terms of algebraic coefficients; nested two loops (α and γ), cycles for each task ($\beta_i; i = 0, 1, 2, 3$), and its probabilities ($\epsilon_i; i = 0, 1, 2, 3$).

To understand BCP algorithm quantitatively, in other words, how large these coefficients are, we profiled benchmark instances and revealed that α, β_0, β_1, β_2, β_3, and γ are 10, 25, 30, 30, 21, and 1.8, respectively. Also probabilities ϵ_1, ϵ_2, and ϵ_3 are 39%, 35%, and 11%, respectively. We noticed that even a simple task, only checking a value of a blocker, requires 25 cycles due to data movement overheads across all the memories (shown in Fig. 2) and the fetch-decode-execution cycle of von Neumann machines. Above that, BCP algorithm executes this task for α (= 10) times. As a result, h is around 500+ cycles with an assumption that we don't have any cache misses. With cache misses and subsequent memory accesses, overall h grows over 1000 cycles. So far, we identified that we have opportunities to parallelize BCP algorithm. Thus, in the next section, we will discuss that how we design our custom hardware to parallelize BCP and process it efficiently, and thus eliminate data movement and von Neumann overheads depicted in Eq. (1).

3 Proposed Custom Hardware

3.1 Key Ideas

Our goal is to design unique chip architecture which can increase parallelism and significantly improve the performance of BCP by optimizing memory architecture, logic processing, and wire design, and also reduce the overheads (shown in Eq. (1)).

First, as mentioned above, the required operations for BCP are quite simple: checking values, chasing pointers, and logic operations. This fact allows us to design simple custom logic for processing and eliminate most of the high-area modules. Further, the simplicity allows us to incorporate a large number of processing elements and place them next to the data. This helps with parallelization as well as elimination of von Neumann overheads, especially data movement.

Second, our above study showed that BCP needs to access all the large data structures used by MiniSAT2 [1] (see Fig. 2). To avoid the high delays associated with off-chip DRAM accesses, our goal is to fit all the memories required for large SAT instances on-chip. The above elimination of high-area modules already frees up area.

Generally, CAM is expensive structure in terms of area and delay. However, we remove the high-area high-delay part, namely the priority encoder. For each row of CAM array, CAM matchline is horizontally connected to SRAM wordline, thus we remove the decoder for SRAM as well. Further, we place logic circuitry for clause evaluation right next to SRAM and directly use data stored on SRAM to evaluate the clause.

Third, to achieve very high degree of parallelism, we propose to use CAM to store clauses. Specifically, each row of CAM cells (see bottom of Fig. 3) stores the IDs of the variables in a particular clause, i.e., i is stored if a variable x_i or $\bar{x_i}$ appears in the clause. This enables completely parallel search across all clauses to look for the specific variable for each BCP run and avoids all indexing operations (pointer chasing). Specifically, this parallelization eliminates α in Eq. (1). Our use of CAM also avoids serially going through each literal in the clause to find a non-false literal. This also eliminates γ in Eq. (1).

Fourth, next to each row of CAM cells mentioned above, we place SRAM cells to store literal values as shown in Fig. 3. Three SRAM cells are required for each literal: two-bits to store the value of the three values (0, 1, or x) of the literal and one bit to store its polarity (0 for x_i and 1 for $\bar{x_i}$). Further, to enable HW-BCP to output the ID of the clause identified to have a conflict or a unit propagation, we have SRAM cells to store the clause ID.

Finally, to parallelize evaluation of the clause to identify whether the clause is satisfied, has a conflict, or a unit propagation, in each row, we incorporate logic elements to evaluate the SAT clause with the updated literal values. This also eliminates much of the data movement overheads. In this manner, we also eliminate the need for blocking literals [12], indirection memory, and the 2WL [11] structure. More importantly, we eliminate the complex sequence of operations required to update two-watched literals, which are expensive to implement in custom hardware.

In this manner, via parallelization, we eliminate all the algebraic coefficients shown in Eq. (1). Consequently, the proposed HW-BCP takes only one cycle but with an increased clock period. In comparison, 1000+ cycles are required on general purpose processors. In our HW-BCP, the fanout of variable ID (VID) increases by order α, since the VID is broadcast to all the CAM rows. However, careful wire and buffer design in the H-tree structure (shown in Fig. 3) to optimize the delay of VID broadcast, reduces this delay to order of $log(\alpha)$. $log(\alpha)$ primarily determines the clock period for HW-BCP.

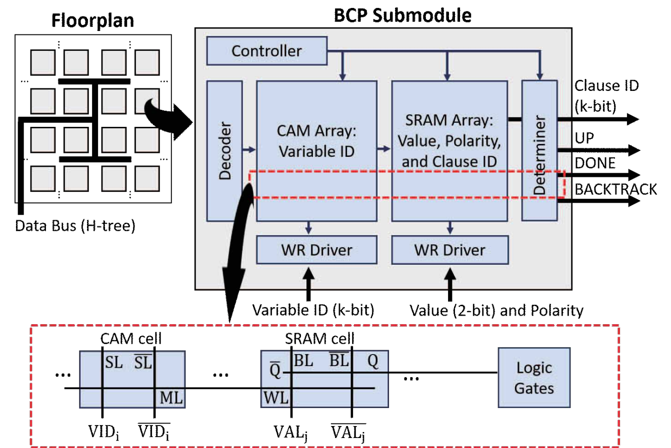

Figure 3: Floorplan with H-Tree data bus and BCP submodules.

Hence, overall our HW-BCP design reduces BCP delay by a factor that is greater than:

$$O(\alpha/log(\alpha)). \tag{2}$$

3.2 Architecture Design

To search all clauses in parallel where the given variable (VID) appears, in our design the VID and its value are broadcast to all BCP submodules using the data bus in the H-tree structure as shown in Fig. 3. The data bus is bidirectional to get clause ID (CID) out when there is a unit propagation. Once VID arrives at each BCP submodule, it goes to the CAM array as a search key. If there is a matched VID stored in the CAM, it activates a CAM matchline which is connected to the wordline of the SRAM cells in the same row. The SRAM stores associated clause information: CID (k-bit) and the values (2-bit) and polarity (1-bit) of each variable. The stored data directly drive the adjacent logic circuity, namely the determiner, which evaluates whether the clause is satisfied (*DONE*), or there is a conflict (*BACKTRACK*), or a unit propagation (*UP*). The *BACKTRACK* signal has the highest priority, since it requires MiniSAT2 [1] to backtrack (handled by software). If there is no conflict, *UP* signal has a higher priority than *DONE* signal. After these signals are output at each row, we incorporate logic circuitry to combine these signals up the H-tree and deliver the final output to the root of the tree.

3.3 VLSI Design

We use custom design flow for memory cells and general digital design flow for logic cells.

TSMC 65nm GP PDK is used in this design. Custom schematics and layouts of memory cells are designed using the minimum metal/wire pitches and shown in Fig. 4. Both the CAM and SRAM cells are abuttable and can be placed side by side, horizontally and vertically. Also, CAM matchlines are aligned with SRAM wordlines and tri-state buffers are placed between matchlines and wordlines to control a write operation on the SRAM cells. Literal values stored on the SRAM cells directly drive the logic circuitry next to the SRAM cells (Determiner in Fig. 3) to evaluate the clause. (This eliminates a SRAM read operation. In a future implementation, we will incorporate the read mode to support post-fabrication testing.) Each clause occupies 3 rows. Finally, CIDs are also stored on the SRAM cells.

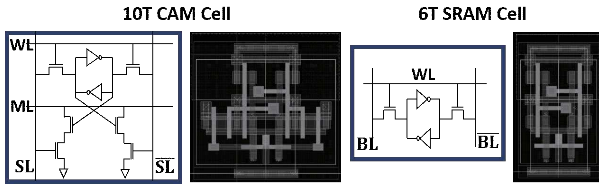

Figure 4: Our schematics and layouts of CAM and SRAM cells.

The entire memory part of the BCP submodule is designed at netlist level with complete floorplan and precise wire information. The logic part of the BCP submodule is designed using standard library cells. Thus we create a detailed netlist of the BCP submodule, which includes all extracted RC parasitics from our memory cell layouts (shown in Fig. 4) and perform accurate simulations.

Once we know the precise dimensions and parasitics of each BCP submodule, as shown in Fig. 3, we design and optimize layout of the H-tree data path.

We repeat the above process by exploring the design space across different sizes of BCP submodules, i.e., different numbers of clauses in each BCP submodule, as well as different buffer sizes for memory write drivers, and different *place and route* options on the logic circuitry. Finally we select the H-tree and BCP submodule design that optimizes a desired combination of area and delay.

Specifically, in the selected design, the number of rows for the BCP submodule is 384 and the number of BCP submodules is 2^{18} for the largest SAT instances. All delay and area evaluations ahead are for this detailed design.

4 Evaluation and Extrapolation

Based on the 65nm HW-BCP design, we want to see how area, delay, and largest SAT instance size change, especially as we move from 65nm to 7nm technology. For this 65nm to 7nm transformation, simply applying a scaling factor would be imprecise, because 7nm technology uses FinFET and the wire pitch/spacing rules are very different. Hence, we carry out a more careful transformation.

By maintaining the same design floorplan and configuration, we estimate area and delay based on the size of library SRAM cells for 65nm and 7nm. Based on our memory cells as well as from the literature [13], we assume that CAM cell area is two times the SRAM cell area. Thus, for extrapolation, based on the SRAM cell size, we calculate the size of the BCP submodule, overall chip size including space of the data bus and logic circuitry, and wire length of the data bus. Since we do not have delay models of 65nm library memory cells and 7nm library cells, we estimate an upper bound of total delay for the 65nm library cells and only wire delay for the 7nm design.

4.1 Area and Scaling

As mentioned above, in our HW-BCP, SRAM cell area is a key factor which decides entire chip area, the H-tree data path length (shown in Fig. 3) and the maximum instance size loaded on the chip.

To extrapolate area as well as delay (discussed in Section 4.2) realistically, we start by designing our own CAM and SRAM cells (shown in Fig. 4). Since we use the standard PDK and follow its design rules to design our memory cells, its size is quite a bit larger than industry's library memory cells. Thus, we extrapolate area and delay based on the library cells. The library SRAM cell area suggested by TSMC is $0.520um^2$ at 65nm [14] and $0.027um^2$ at 7nm [15]. Considering theoretical area scaling from 65nm to 7nm, we expect SRAM cell area would decrease by 86x (= $(65/7)^2$), but in reality

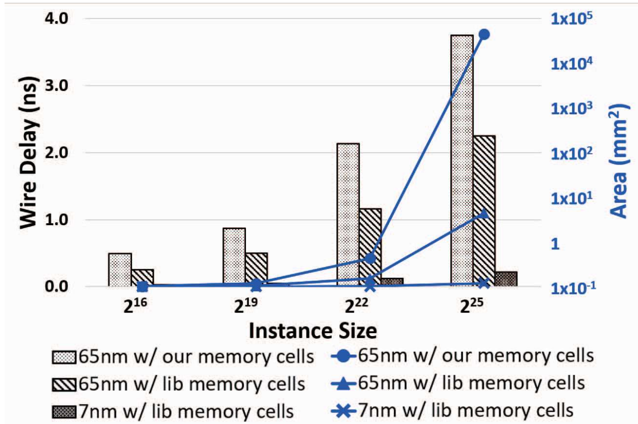

Figure 5: Area and wire delay with different instance sizes on different technologies: (1) 65nm with our memory cells, (2) 65nm with library memory cells, and (3) 7nm with library memory cells. In 65nm, wire delay is the dominant part (>75%) of total delay for the largest benchmark instances.

it is just 19.2x smaller. This is because it is difficult to make a very compact SRAM cell in recent technologies like 7nm, as FinFET front-end is combined with much more stringent back-end metal routing rules. Thus, anticipated performance enhancement from 65nm to 7nm may not be proportional to the gate length of the transistor.

Fig. 5 shows how area and wire delay change with different target instance sizes for three cases; (1) 65nm with our memory cells, (2) 65nm with library memory cells [14], and (3) 7nm with library memory cells [15]. The instance size, x-axis, has four cases; the numbers of clauses are 2^{16}, 2^{19}, 2^{22}, and 2^{25}, respectively. Total area increases with the instance size. In 65nm, target instance size for practical chip size would fall in the range 2^{16}-2^{20}.

However, if we can re-design the proposed HW-BCP using 7nm technology, we can have HW-BCP which is able to load the largest SAT instances with 2^{25} clauses on a reasonable chip size (less than $4cm^2$). Due to the much smaller 7nm SRAM cell area, overall chip area and wire delay shrink significantly as shown in Fig. 5.

4.2 Delay and Scaling

By estimating total BCP delay for our 65nm cells and further wire delay for 65nm/7nm library cells, we evaluate accelerated BCP performance and see potential for the advanced technology.

Total delay for BCP is composed of three parts: 2x the wire delays on the H-tree (shown in Fig. 3, 1x for data-in and 1x for data-out), memory operation delay from the point that data arrive at the BCP submodule to the point that a new value is updated to the SRAM, and subsequent logic circuit delay. The wire delay is the dominant part of the proposed HW-BCP. Using our memory cells, we achieve the optimal total delay of 4.5ns, in which wire delay, memory operation delay, and logic delay are 3.8ns (84%), 0.5ns (11%), and 0.3ns (5%), respectively. When the 65nm library memory cells are used, wire delay is estimated to 2.3ns and the upper-bound total delay is estimated by adopting the rest delay from our 65nm cells, which is 3.0ns.

Thus with the HW-BCP using our 65nm cells, BCP speedup over MiniSAT2 [1] on general purpose processors is 62-123x. In a version of our HW-BCP design with the 65nm library memory cells, the speedup is a minimum of 93-185x. (We assume the range of clock frequency of general purpose processors is 2-4GHz.)

Table 1: BCP performance comparison between MiniSAT2 [1], FPGA-BCP [8], and the proposed HW-BCP.

	MiniSAT2 [1]	FPGA-BCP [8]		HW-BCP	
Technology	N/A	65nm	7nm	65nm	7nm
Largest instance (No. of clauses)	N/A	64K	2M	670K	32M*
Avg. clock cycles per BCP	1000	10	11+	1	1
Clock period (ns)	0.25-0.5	5	2.5	1.6	1.24-
Avg. BCP delay (ns)	250-500	50	27.5+	1.6	1.24-

* Largest instance size in SAT Competition 2017 benchmark suite

Then, we carefully address the estimation of wire delay on the 7nm node. By using the normalized inverter FO4 delay which decreases by 4x on 7nm, logic circuit delay is estimated. RC ($\Omega \cdot F/um^2$) increases by 8-18x when technology changes from 180nm to 35nm [16]. With this tendency, we extrapolate RC for 7nm. RC is expected to increase by 15x when the technology changes from 65nm to 7nm. We estimate delay of the data bus (shown in Fig. 3) by assuming that it is a semi-global wire and RC estimation is in the middle of conservative and aggressive estimation.

Considering the above factors, wire delay using the 7nm library cells is estimated and shown in Fig. 5. Wire delay significantly reduces to 0.22ns, which is 10.2x enhancement compared to the HW-BCP with the 65nm library cells.

4.3 Feasibility and Comparison

Since the proposed HW-BCP is an ASIC design and scalable, the largest SAT instance that can be solved is limited by chip size. We assume that 4cm^2 is a feasible chip size for each technology.

Table 1 shows that, on HW-BCP using our 65nm cells, the largest instance that fits in a 4cm^2 chip has 670K clauses and has a clock period of 1.6ns. Compared to FPGA-BCP [8], setting aside the fact that our HW-BCP can load 10x larger instances, in 65nm, HW-BCP achieves at least 30x speedup for BCP operations. Also, on SAT instances with up to 670K clauses, in 65nm, our HW-BCP provides 156-312x speedup over MiniSAT2 [1] for BCP operations.

In 7nm, we estimate that the HW-BCP can hold the largest SAT instances (32M clauses), in a 3.3cm^2 chip. In our HW-BCP, the clock period can be considerably reduced due to lower wire delays in 7nm. As an upper bound, even with a pessimistic assumption that the memory operation delay is the same as that for 65nm technology, we can achieve 1.24ns clock period. FPGA-BCP can also be implemented on Xilinx's largest 7nm FPGA [17] and load an instance with 2M clauses due to 32x larger BRAM capacity on the 7nm chips. FPGA clock period can decrease by 2x [17]. Compared to FPGA-BCP, our HW-BCP can load 16x larger SAT instances and achieve at least 22x speedup on BCP. On the largest SAT instances, in 7nm, our HW-BCP has 201-403x speedup over MiniSAT2 [1] for BCP.

4.4 MiniSAT2-level Speedup

We evaluate the overall speedup at the MiniSAT2 [1] level, not just at the level of BCP operations. We assume that our HW-BCP is a co-processor along with MiniSAT2 [1] on general purpose processors, namely MiniSAT2+HW-BCP, where BCP runs on our HW-BCP and the rest of the MiniSAT2 [1] runs on general purpose processors. Compared to the software MiniSAT2 [1], on the 70 benchmark instances, in 65nm, the overall SAT speedup of MiniSAT2+HW-BCP is shown in Table 2. Since total BCP time is 80-90% of total SAT solving time, the maximum speedup we can achieve at the entire MiniSAT2-level is 5-10x. Even though we achieved significant speedup on BCP operations, MiniSAT2-level speedup is bounded by Amdahl's Law.

Table 2: BCP speedup on HW-BCP and SAT speedup on MiniSAT2+HW-BCP compared to software MiniSAT2 [1].

	BCP speedup on HW-BCP* (range)	SAT speedup on MiniSAT2+HW-BCP* (range)
w/ our cells	98.9x (62-123x)	6.6x (2.2-15.3x)
w/ lib cells	148.8x (93-185x)	6.7x (2.3-15.5x)

* Designed in 65nm technology

We expect that on 7nm, HW-BCP achieves significantly increased BCP speedup over MiniSAT2 [1], which is anticipated to be 300-1000x. In our future work, we plan to simplify our HW-BCP design to save chip area and use it to accelerate other parts of MiniSAT2 [1] to improve overall SAT performance. We also plan to optimize HW-BCP delay by pipelining our HW-BCP architecture.

5 Conclusion

We have designed a custom hardware accelerator for BCP (HW-BCP) to fully parallelize BCP operations and eliminate von Neumann overheads, especially data movement. Also by storing on-chip all major large data structures (clauses, variable values, etc.), we completely eliminate cache misses and associated performance overheads. In 65nm technology, HW-BCP achieves at least 62x speedup over general purpose processors with full custom combinations of memory, logic circuitry, and interconnect design. We also show that on 7nm, besides performance enhancements, HW-BCP can support the largest SAT benchmark instances in a practical chip size.

Acknowledgments

We gratefully acknowledge the support of NSF.

References

[1] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *Int'l Conference on Theory and Applications of Satisfiability Testing*, May 2003, pp. 502–518.
[2] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem-Proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962.
[3] B. Selman and H. Kautz, "Domain-Independent Extensions to GSAT: Solving Large Structured Satisability Problems," in *Proceedings of the 13th Int'l Joint Conference on Artifical Intelligence*, Aug. 1993, pp. 290–295.
[4] J. Marques-Silva and K. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Trans. on Computers*, vol. 48, pp. 506–521, May 1999.
[5] J. Thong and N. Nicolici, "Fpga acceleration of enhanced boolean constraint propagation for sat solvers," in *IEEE/ACM Int'l Conference on Computer-Aided Design (ICCAD)*, Nov 2013, pp. 234–241.
[6] K. Gulati, M. Waghmode, S. Khatri, and W. Shi, "Efficient, scalable hardware engine for Boolean satisfiability and unsatisfiable core extraction," *IET Computers Digital Techniques*, vol. 2, no. 3, pp. 214–229, May 2008.
[7] K. Gulati, S. Paul, S. P. Khatri, S. Patil, and A. Jas, "FPGA-based hardware acceleration for Boolean satisfiability," *ACM Trans. on Design Automation of Electronic Systems*, vol. 14, no. 2, pp. 1–33, Apr. 2009.
[8] J. D. Davis, Z. Tan, F. Yu, and L. Zhang, "A Practical Reconfigurable Hardware Accelerator for Boolean Satisfiability Solvers," in *ACM/IEEE Design Automation Conference (DAC)*, Jun. 2008, pp. 780–785.
[9] M. Safar, M. W. El-Kharashi, M. Shalan, and A. Salem, "A reconfigurable, pipelined, conflict directed jumping search SAT solver," in *IEEE/ACM Design, Automation Test in Europe*, Mar. 2011, pp. 1–6.
[10] X. Yin, B. Sedighi, M. Varga, M. Ercsey-Ravasz, Z. Toroczkai, and X. S. Hu, "Efficient Analog Circuits for Boolean Satisfiability," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 1, pp. 155–167, Jan. 2018.
[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *ACM/IEEE Design Automation Conference (DAC)*, 2001, pp. 530–535.
[12] N. Sörensson and N. Eén, "MiniSat 2.1 and MiniSat++ 1.0 - SAT Race 2008 Editions," *The SAT race 2008: Solver descriptions*, 2008.
[13] K. Pagiamtzis and A. Sheikholeslami, "Content-Addressable Memory (CAM) Circuits and Architectures: a Tutorial and Survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, Mar. 2006.
[14] ITRS, "ITRS Report 2008," in *http://www.itrs2.net/itrs-reports.html*, 2008.
[15] J. Chang *et al.*, "A 7nm 256Mb SRAM in high-k metal-gate FinFET technology with write-assist circuitry for low-VMIN applications," in *IEEE Int'l Solid-State Circuits Conference (ISSCC)*, Feb. 2017, pp. 206–207.
[16] R. Ho, K. Mai, and M. Horowitz, "The Future of Wires," *Proceedings of the IEEE*, vol. 89, no. 4, pp. 490–504, Apr. 2001.
[17] S. Ahmad *et al.*, "A Versatile 7nm Adaptive Compute Acceleration Platform Processor," in *IEEE Hot Chips Symp.*, Aug. 2019.