FPGA Acceleration of Decision-Based Problems using Heterogeneous Computing

FPGA ACCELERATION OF DECISION-BASED PROBLEMS USING HETEROGENEOUS COMPUTING

BY

JASON THONG, M.A.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Doctor of Philosophy (2014) McMaster University

(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE:	FPGA Acceleration of Decision-Based Problems using Heterogeneous Computing
AUTHOR:	Jason Thong M.A.Sc. (Computer Engineering), McMaster University, Hamilton, Ontario, Canada
SUPERVISOR:	Dr. Nicola Nicolici
NUMBER OF PAGES:	xxiv, 274

Abstract

The Boolean satisfiability (SAT) problem is central to many applications involving the verification and optimization of digital systems. These combinatorial problems are typically solved by using a decision-based approach, however the lengthy compute time of SAT can make it prohibitively impractical for some applications.

We discuss how the underlying physical characteristics of various technologies affect the practicality of SAT solvers. Power dissipation and other physical limitations are increasingly restricting the improvement in performance of conventional software on CPUs. We use heterogeneous computing to maximize the strengths of different underlying technologies as well as different computing architectures.

In this thesis, we present a custom hardware architecture for accelerating the common computation within a SAT solver. Algorithms and data structures must be fundamentally redesigned in order to maximize the strengths of customized computing. Generalizable optimizations are proposed to maximize the throughput, minimize communication latencies, and aggressively compact the memory. We tightly integrate as well as jointly optimize the hardware accelerator and the software host.

Our fully implemented system is significantly faster than pure software on real-life SAT problems. Due to our insights and optimizations, we are able to benchmark SAT in uncharted territory.

Acknowledgements

I am very grateful to those around me who have contributed to my development, both professionally and personally. I have been fortunate to experience and engage in many constructive environments where talent as well as perseverance are rewarded.

Firstly, I would like to acknowledge my academic colleagues. Many productive brainstorming sessions have and continue to happen in the Computer-Aided Design and Test group at McMaster. I wish to thank past and present members who have shaped my technical skills and provided valuable feedback. These include: Henry Ko, Adam Kinsman, Zahra Lak, Phil Kinsman, Amin Vali, Pouya Taatizadeh, Xiaobing Shi, and Peter Bergstra. I admire as well as thank my supervisor Nicola Nicolici for his no nonsense approach in challenging me to improve while sharing his sense of humor. I would also like to thank the examining committee for their efforts (Nicola Nicolici, Aleksandar Jeremic, Shahin Sirouspour, and Dejan Markovic).

I would like to acknowledge my colleagues from industry. I have had a wonderful experience working at Altera and Microsoft Research. This provided valuable insights on the development of CAD for FPGAs as well as FPGA applications, which has greatly furthered my own academic work. Special thanks goes to my supervisors John Freeman (Altera), Dmitry Denisenko (Altera), and Andrew Putnam (Microsoft) for giving me the opportunity to grow. I also wish to acknowledge Microsoft Research

for the equipment loan (Stratix V FPGA), without which my results would not have been as elaborate.

Finally, I am sincerely grateful for my loving and supportive family. My parents (Bruce Thong and Liza Thong) and girlfriend (Lisha Wang) provide an atmosphere of stability, where triumph and failure can be genuinely shared. Your commitment inspires me to keep on learning and improving.

Glossary

Abstraction A less detailed perspective of a complex problem which is often an approximation.

ASIC Application-specific integrated circuit.

Boolean A data type that can only be true or false.

BCP Boolean constraint propagation.

Cache A faster but smaller memory, provides faster access to commonly used data than main memory.

CAD Computer-aided design.

Clause A constraint that must be satisfied in Boolean SAT.

DMA Direct memory access, use of a dedicated memory controller to efficiently transfer large amounts of data.

DPLL The Davis-Putnam-Logemann-Loveland algorithm for solving SAT.

DRAM Dynamic random access memory.

EDA Electronic design automation.

Enhanced BCP The use of non-Boolean clauses for lossless compression.

Floorplan An overview of the physical placement of major functional blocks on the integrated circuit.

FPGA Field-programmable gate array.

Heterogeneous computing The use of more than one type of processor in a system.

Literal Either the positive occurrence or the negative occurrence of a variable in Boolean SAT.

Logic resources An abstraction of the amount of silicon required to implement a logic function.

Logical constructs The building blocks to describe the behavior of a digital system.

Macro clauses Multiply packed clauses, which facilitate better hardware memory utilization.

Multithreading The use of multiple execution threads, which is one technique to facilitate concurrent computation.

NOC Network-on-chip.

PCIe Peripheral component interconnect express, a high-speed hardware/software communication protocol.

Pipelining Insertion of registers to reduce the worst case propagation delay through combinational logic, thereby enabling a faster clock frequency.

Preprocessing Transformations applied to a problem which are expected to make it easier to solve but without affecting its solution.

Random access The ability to access data at any memory location in any order.

SAT Abbreviation for the satisfiability problem.

Search space The set of all possible solutions that can be examined by an algorithm.

SMT Satisfiability modulo theory, an extension or generalization of SAT.

SRAM Static random access memory.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The ability to efficiently compute solutions for discrete problems has many practical implications. Many real-world problems are inherently discrete or combinatorial. For example, suppose we are driving a car on streets organized in a two dimensional grid, e.g. running north-south and east-west. If we are currently driving north and want to go somewhere north-west of the current location, it is physically impossible to turn two-thirds to the left, for instance. The discreteness of the problem dictates that we must perform a combination of driving some distance north as well as some distance west in order to arrive at the desired location.

Combinatorial problems commonly exist in two forms: as a *satisfiability* problem or as an *optimization* problem. Continuing with examples, suppose some streets have construction which completely blocks vehicles from driving. We may be interested in solving a feasibility problem, e.g. a tour company may want to know if it is possible to visit every street intersection exactly once without reusing any section of any street. We may be interested in solving an optimization problem, e.g. knowing how long it takes to drive down any given street, what is the fastest way to make

such a tour? Whether we are optimizing the driving time of vehicles on the road or optimizing packets through a telecommunications network, the approach to solving such combinatorial problems tends to be similar.

Discrete problems are often solved using a *decision-based* approach. Returning to our first example, if we are currently driving north, at the next street intersection we must choose whether to keep driving north or turn left to the west. The structure of the problem determines which decisions are valid, and may also guide the decision procedure (e.g. we want to avoid construction which blocks certain streets).

For some discrete problems, the only guaranteed way to find a solution is with an exhaustive search. This results in an extremely large computational complexity. We discuss NP-completeness in section 2.1.1.

The lengthy compute time needed to solve some discrete problems can make them prohibitively impractical. This is presently one of the major challenges facing combinatorial problems. Some optimization problems use heuristics to reduce the compute time, however there is no substitute when using satisfiability for verification, e.g. does there exist a counter-example that causes the system to behave incorrectly?

Much of the research community has focused on developing new search strategies. This thesis addresses an orthogonal issue: *we explore how different implementation technologies can increase the computational throughput.* An efficient design requires an insightful remapping of the problem to maximize the strengths of each underlying technology. Algorithms and data structures have to be fundamentally rethought.

This chapter introduces the Boolean satisfiability problem (SAT) with a simple and intuitive example. The organization of the thesis is presented before we conclude this chapter with the scope of our contributions.

1.1 An Introduction to SAT

In general, a satisfiability problem consists of some *given* constraints which contain variables, and one must *find* an assignment to those variables to satisfy all constraints or prove that no such assignment exists. As an example, suppose we must satisfy:

$$x + 2y \leq 4$$
$$2x + y \geq 5$$

where x and y are the variables. One solution to this is $x = 2$ and $y = 1$.

In Boolean SAT, each variable can only be assigned true or false. Constraints are also referred to as "clauses". A clause must be satisfied by satisfying at least one of the variables, thus we OR the variables. For example, to satisfy the clause $(a + \overline{b})$, we can assign $a =$ true or $b =$ false, or both. All clauses in the SAT problem must be satisfied, hence we AND the clauses together. Conclusively, one example of a Boolean SAT problem could be:

$$(a + \overline{b})(\overline{a} + \overline{c} + d)(b + \overline{d}).$$

One solution is $a = 1$, $b = 1$, $c = 0$, and $d = 1$ (0 means false and 1 means true).

1.1.1 An Intuitive Example of Boolean SAT

We now present an example to provide an intuitive perspective of Boolean SAT. Suppose there are three friends Alice, Bob, and Charlie who want to buy some ice-cream. The store where they buy it from only sells 3 flavors: mint, strawberry, and chocolate. We can use Boolean variables m, s, and c to indicate whether or not they will buy each respective flavor.

Now let us introduce the constraints. Suppose Alice does not care for mint, but she will accept it if chocolate is purchased. Thus the first clause is:

$$(\overline{m} + c).$$

The SAT interpretation is that if $m = 1$ (mint was purchased), the only remaining way to satisfy the clause is to assign $c = 1$ (chocolate must also be purchased). This is the premise of Boolean constraint propagation, as examined in section 2.5.1.

Bob wants one of mint or strawberry but not both. In other words, if and only if mint is purchased, then strawberry should not be purchased. The example with Alice illustrated how to encode just a simple "if", so now we use two clauses to cover both halves of the "if and only if":

$$(m + s)(\overline{m} + \overline{s}).$$

Notice that the first clause is not satisfied if neither are purchased and the last clause is not satisfied if both are purchased.

Charlie is hungry and wants at least two flavors. This can be represented with three clauses:

$$(m + s)(m + c)(s + c).$$

If only one flavor is purchased (only one variable is assigned as true), then only two out of the three clauses will be satisfied. At least one more variable must be assigned as true in order to satisfy all three clauses.

Finally, we ask the question: is there a combination of ice-cream flavors that Alice, Bob, and Charlie can buy to satisfy their requirements? Removing the duplicate clause $(m + s)$, the corresponding SAT problem to solve is:

$$(\overline{m} + c)(m + s)(\overline{m} + \overline{s})(m + c)(s + c).$$

One solution is $m = 1$, $s = 0$, $c = 1$. For each flavor, the corresponding variable indicates whether or not to purchase it.

1.1.2 Variations of Boolean SAT

Continuing the previous example, now suppose that one unit of ice-cream has a different cost depending on the flavor, e.g. mint costs \$3, strawberry costs \$4, and chocolate costs \$5. We could instead solve an optimization problem: what is the specific combination of ice-cream flavors that Alice, Bob, and Charlie should buy to minimize the cost while satisfying their requirements?

For applications that operate primarily on integers instead of Boolean variables, one may encode the problem to preserve the high-level semantics. For example, we could instead represent that Charlie wants at least two flavors with:

$$(m + s + c \geq 2)$$

where m, s, and c are now integers in the range of 0 to 1. Integer linear programming is a close cousin of Boolean SAT, as both are based on discrete constraints.

Moving back to satisfiability (no optimization), suppose a problem is unsatisfiable. Some applications may be interested in extracting more information, such as what is the maximum number of clauses that can be satisfied (this count must be strictly less than the total number of clauses). Alternatively, one may be interested in which subset of clauses participated in causing the unsatisfiability.

1.1.3 Definitions and Terminology

To formalize SAT from our previously intuitive example, we now briefly provide some definitions to commonly used terminology in the satisfiability research community.

Definition 1. *A "clause" is a constraint describing a relationship among several variables that must be satisfied.*

In Boolean SAT, each clause takes the form $(x_1 + x_2 + x_3 + \ldots + x_n)$, where each x_i is a Boolean variable which may be assigned only true or false. To satisfy a clause, at least one *literal* must be satisfied. For example, to satisfy the clause $(x_1 + \overline{x}_2)$, we could assign $x_1 =$ true or $x_2 =$ false, or both.

Definition 2. *A "literal" is a variable with a particular sign.*

For example, x_1 and $\overline{x}_1$ are instances of the *same* variable x_1. Conversely, the positive literal x_1 and the negative literal $\overline{x}_1$ are *different* literals.

Definition 3. *A "satisfiability problem" consists of several clauses. For a given satisfiability problem, the objective is to find an assignment to the variables that satisfies every clause, or prove that no such assignment exists.*

For example, given the Boolean SAT problem $(x_1 + \overline{x}_2)(x_2 + \overline{x}_3 + x_4)(x_1 + \overline{x}_3 + \overline{x}_4)$, one solution is to assign all four variables as true. Conversely, the Boolean SAT problem $(\overline{x}_1 + \overline{x}_2)(\overline{x}_1 + x_2)(x_1 + \overline{x}_2)(x_1 + x_2)$ has no solution.

Definition 4. *The "clause size" refers to how many variables it contains.*

Intuitively, a larger Boolean clause is easier to satisfy. Smaller clauses typically participate in Boolean constraint propagation more frequently (see section 2.5.1).

1.2 Thesis Organization

Chapter 2 discusses the applications of SAT as well as software algorithms for solving SAT. We also present concrete examples which illustrate how some applications are mapped into SAT. In chapter 3, we examine current technology trends in detail. This motivates the use of hardware accelerated SAT and also identifies the limitations of most prior works on SAT in hardware.

We use technology trends to guide our design decisions, which are summarized in section 3.6. We only accelerate part of the SAT solver in hardware. Some types of computation are ideal for custom hardware while others are better suited for a CPU (software). Heterogeneous computing enables us to maximize the benefit of each.

We present our hardware processor in chapter 4. Chapter 5 then illustrates how these communicate with each other and with software. Chapter 6 examines advanced optimizations in hardware. Software integration is discussed in chapter 7 (SAT solver integration) and chapter 8 (additional software, such as SAT preprocessing).

Experimental results of our entire system are finally presented in chapter 9. A massive implementation effort was required to evaluate our ideas. Concluding remarks are summarized in chapter 10.

1.3 The Scope of Our Contributions

Many discrete or combinatorial problems are solved using a decision-based approach. This class of problems is typically difficult to accelerate due to the complex control flow. Our acceleration of SAT addresses many of these complexities as well as physical aspects which can no longer be ignored in the design of digital circuits.

Our contributions are manifested in the insightful restructuring of computation and data structures, as well as resolving several of the ensuing complications. The concepts, designs, and techniques used to solve these problems have generalizable properties. For example, chapter 4 demonstrates several ways to facilitate faster computation while also minimizing the amount of memory needed to represent SAT in hardware. Implicit representations and lossless compression are powerful tools, and these will be useful in other memory bound problems. In chapter 5, we classify different types of on-chip communication so that we can exploit the different properties of each class to obtain efficient implementations. In general, some applications can benefit from separating synchronization networks from data.

We make no claim of improving SAT algorithms, as surveyed in chapter 2. This is *orthogonal* to our acceleration of Boolean constraint propagation (BCP), which is the dominant computation within a SAT solver (see section 2.5.1). BCP is a fundamental part of all DPLL-based SAT solvers, so by implementing the remainder of our SAT solver in software, we facilitate an easy integration of *any* future DPLL enhancement.

Finally, our extensive use of lossless compression enables us to benchmark SAT problems in uncharted territory. We are significantly faster than software on real-life SAT problem sizes, unlike many existing hardware SAT designs which only support problem sizes too small to be of much practical interest. We are also the first to introduce multithreading into hardware SAT. Using a real implementation (not simulation), for the first time we are able to accurately observe system limitations such as thrashing the CPU's shared cache with numerous threads. Such factors are no longer trivial for highly accelerated computation. We have extensively designed our computing platform to mitigate these wherever it is practically possible.

Chapter 2

SAT Applications and Algorithms

This chapter begins by surveying the numerous applications of SAT. We then illustrate how some applications are mapped into SAT using concrete examples. The remainder of the chapter examines the basic approaches used in several different types of SAT algorithms. We also discuss many of the newer optimizations for DPLL, which is arguably the most common SAT algorithm used. Finally, we discuss preprocessing techniques, which simplify a SAT problem without changing its satisfiability.

2.1 Applications of SAT

2.1.1 NP-Completeness and a Historical Perspective

Boolean SAT was the first NP-complete problem, as proved by Cook [1] in 1971. NP stands for "non-deterministic polynomial". These problems must be deterministically verifiable and non-deterministically solvable (e.g. with the luckiest possible guess), both within polynomial time. NP does *not* mean "non-polynomial". This misconception often arises since there are currently no known algorithms to deterministically solve

NP-complete problems in polynomial time. Presently there is also no proof that such algorithms cannot exist. This is the famous P = NP problem. We refer the reader to [2] for more details on the theory and proofs of NP-completeness.

In 1972, Karp [3] demonstrated that 21 other problems were also NP-complete since there exists a polynomial time many-one reduction from Boolean SAT to each of the 21 problems. The solution to any NP-complete problem can be obtained by instead solving any other NP-complete problem and then applying a polynomial time transformation to obtain the solution to the original problem.

Prior to the 1990s, NP-complete problems were arguably mostly a theoretical interest and primarily used to demonstrate the intractability of new problems. In other words, it was a strong motivator for heuristics or approximate algorithms.

Complete algorithms for solving SAT date back as far as the 1960s, e.g. DPLL [4]. However, much of the smartness in "modern" SAT solvers was developed in the late 1990s and early 2000s. As the computing power grew over the decades, in the 1990s it eventually reached a threshold where solving SAT became practical. This instilled lots of research on ways to solve SAT more efficiently.

Modern SAT solvers have radically changed the picture for NP-complete problems. They can learn the internal structure of a SAT problem and aggressively prune the search space. Because of this smartness, the worst case exponential run time of a SAT solver is often not exposed by many real-world SAT problems.

Presently there are many efficient SAT solvers, so it is now common for one to solve a hard combinatorial problem by instead converting it to Boolean SAT. Using this approach, SAT has found itself at the core of many applications (an extensive survey is provided in [5]). SAT has therefore become an enabling technology.

2.1.2 Survey of SAT Applications

Throughout this section, we briefly discuss some common applications of SAT. New applications are emerging and thus it is impossible to provide an exhaustive list.

SAT plays a major role in Electronic Design Automation (EDA), which uses computer-aided design (CAD) tools to verify or optimize problems that are inherently discrete. On the verification side of EDA, SAT can be used for equivalence checking [6, 7]. When a compiler performs optimizations, the functionality of the optimized implementation should still exactly match the behavior as specified in the source code. A detailed example is demonstrated in section 2.2.2.

SAT can be used for hardware verification [8] and software verification [9], typically through model checking [10,11]. For example, we may want to ensure a hardware state machine always finishes its computation within 8 clock cycles. In software, we could ensure that we never access out-of-bounds memory or perform undefined computation, such as dividing by zero. Run-time assertions that simply *check* this, for example:

```
x = 0;
for (i=0; i<10; i++) {
  j = permute_function(i);
  assert(j>=0 && j<5); //array was allocated for 5 items
  x = x + array[j];
}
assert(x != 0);
y = z / x;
```

may not be particularly useful in safety-critical systems. Verification either proves we can avoid bad states by construction or it produces an actual counter-example.

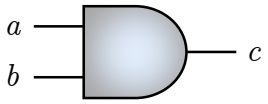
On the optimization side of EDA, SAT can be used for logic synthesis. A field-programmable gate array (FPGA) typically implements combinational logic using lookup tables (LUTs), thus SAT can be used to decompose the logic from a given hardware design to best utilize the LUTs [12]. SAT can also be used to solve routing (how to wire logic gates together) for both FPGAs [13] and standard cells [14].

Automatic test pattern generation (ATPG) screens logic circuits for manufacturing defects. SAT can be used to design ATPG test sequences that either maximize the coverage (how much of the circuit gets tested) for a limited number of test sequences, or minimize the number of test sequences (reduce testing time and costs) to achieve at least some threshold of coverage. Examples of SAT-based ATPG include [15, 16].

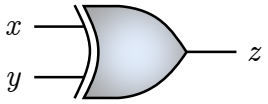
Cryptography is another major application of SAT [17]. For example, for all y with a Hamming distance of 1 from x, we could ensure that $encrypt(x) \neq encrypt(y)$.

SAT plays a major role in scheduling problems [18]. SAT is good at capturing constraints such as: any two tasks that require a shared resource cannot run in the same time interval. Graph coloring also captures this well, and it is one of the 21 problems proven NP-complete by Karp [3]. Many similar problems exist in artificial intelligence, especially for planning-based problems [19]. For example, knowing which moves a robot can make, find a legitimate plan of motion to reach the goal state.

As computation power continues to grow and SAT solvers become increasingly efficient, SAT could become an enabling technology for completely new applications. Recently, SAT has found itself in applications such as bioinformatics. A SAT-based approach to characterizing single nucleotide mutations in DNA is presented in [20]. Protein design can be assisted by SAT, which finds the lowest energy states to predict how a sequence of amino acids will fold into a protein [21].



$(\overline{a} + \overline{b} + c)(a + \overline{c})(b + \overline{c})$



$(\overline{x} + \overline{y} + \overline{z})(\overline{x} + y + z)(x + \overline{y} + z)(x + y + \overline{z})$

Figure 2.1: The Tseitin encoding for an AND gate and an XOR gate.

2.2 Mapping Logical Constructs to SAT

2.2.1 Logical Construct Encoding

We begin by illustrating how logical constructs are mapped to SAT. This serves as a precursor before we demonstrate the mapping of real applications to SAT.

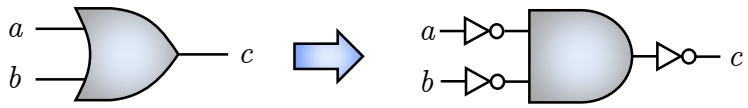
Logical constructs are the building blocks of digital systems. When describing the behavior of a digital system, there is no need to differentiate between a multiplexer in hardware or a software assignment made using an if-else statement.

Logical constructs are commonly mapped to Boolean SAT using the Tseitin encoding [22]. For example, the encodings of an AND gate and an XOR gate are shown in Figure 2.1. The clauses in the encoding have an intuitive explanation. For the AND gate:

- $(\overline{a} + \overline{b} + c)$ indicates that if $a = 1$ and $b = 1$, then we must have $c = 1$ (this is the only *remaining* way to satisfy the clause, Boolean constraint propagation is discussed in section 2.5.1).

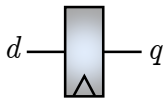
- $(a + \overline{c})$ indicates that if $a = 0$, then we must have $c = 0$. Alternatively, the contrapositive is that if $c = 1$, then we must have $a = 1$.

- By symmetry, the same idea applies to $(b + \overline{c})$ as it did to $(a + \overline{c})$.



$$(a + b + \overline{c})(\overline{a} + c)(\overline{b} + c)$$

Figure 2.2: Mapping an OR gate into AND to reuse the Tseitin encoding.



Unroll over n clocks: $(d_0 + \overline{q}_1)(\overline{d}_0 + q_1)\ldots(d_{n-1} + \overline{q}_n)(\overline{d}_{n-1} + q_n)$

Figure 2.3: Loop unrolling enables the tracking of values over time.

For the XOR gate, all possible combinations of signs for the first two variables x and y are considered, as shown below in blue:

$$(\overline{x} + \overline{y} + \overline{z})(\overline{x} + y + z)(x + \overline{y} + z)(x + y + \overline{z})$$

Once both x and y are assigned, exactly one of the four possible cases will require the last variable z (shown above in red) to take a certain value in order to satisfy that clause. For example, if $x = 1$ and $y = 0$, the second clause requires $z = 1$ to be satisfied. There is a symmetry between the variables, so if any two variables are assigned, the third variable can be implied.

Inversion comes for free in SAT by simply exchanging x and $\overline{x}$. Figure 2.2 shows the encoding of an OR gate. Using De Morgan's Law, we can reuse the Tseitin encoding of an AND gate.

Figure 2.3 illustrates how software loops or hardware state information can be unrolled to track how values will evolve over iterations or time, respectively. Model

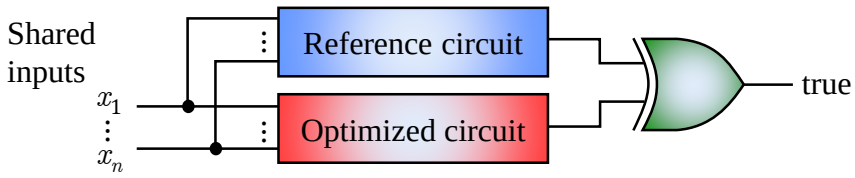


Figure 2.4: A miter circuit tests if the outputs can ever be different.

checking applications use a bounded model to avoid an infinite number of states. More recent work on unbounded model checking typically uses SAT to produce a proof (e.g. by induction) that certain behavior will be maintained in the steady state.

2.2.2 A Demonstration of Equivalence Checking

Equivalence checking can be used to verify the correctness of optimizations. For example, the behavior of a system optimized by a compiler should still match the reference source code. Figure 2.4 illustrates the "miter" circuit for verifying this. By constraining the XOR output to true, we are looking for a counter-example. Solving the corresponding SAT problem either produces one or guarantees that none exist.

As an example, in Figure 2.5 we illustrate two implementations of the majority function, which is true if at least two out of the three inputs are true. This arises in adders (specifically it is the carry logic for a 1-bit full adder). The blue subcircuit is the implementation that arises from cascading two half adders in order to create one full adder. The red subcircuit could be a speed optimization, as the longest combinational path has improved from 3 gates to 2 gates. The green XOR serves as the miter. We can force the output t to true by using a clause with only one variable. The only way to satisfy the clause (t) is by assigning $t = 1$. The derivation of the corresponding SAT problem is summarized in Table 2.1, which has a total of 32 clauses.

Table 2.1: The clauses for the SAT problem corresponding to Figure 2.5.

Description	Clauses
$d = b$ XOR c	$(\overline{b} + \overline{c} + \overline{d})(\overline{b} + c + d)(b + \overline{c} + d)(b + c + \overline{d})$
$e = a$ AND d	$(\overline{a} + \overline{d} + e)(a + \overline{e})(d + \overline{e})$
$f = b$ AND c	$(\overline{b} + \overline{c} + f)(b + \overline{f})(c + \overline{f})$
$g = e$ XOR f	$(\overline{e} + \overline{f} + \overline{g})(\overline{e} + f + g)(e + \overline{f} + g)(e + f + \overline{g})$
$w = a$ AND b	$(\overline{a} + \overline{b} + w)(a + \overline{w})(b + \overline{w})$
$x = a$ AND c	$(\overline{a} + \overline{c} + x)(a + \overline{x})(c + \overline{x})$
$y = b$ AND c	$(\overline{b} + \overline{c} + y)(b + \overline{y})(c + \overline{y})$
$z = w$ OR x OR y	$(w + x + y + \overline{z})(\overline{w} + z)(\overline{x} + z)(\overline{y} + z)$
$t = g$ XOR z	$(\overline{g} + \overline{z} + \overline{t})(\overline{g} + z + t)(g + \overline{z} + t)(g + z + \overline{t})$
$t =$ true	(t)

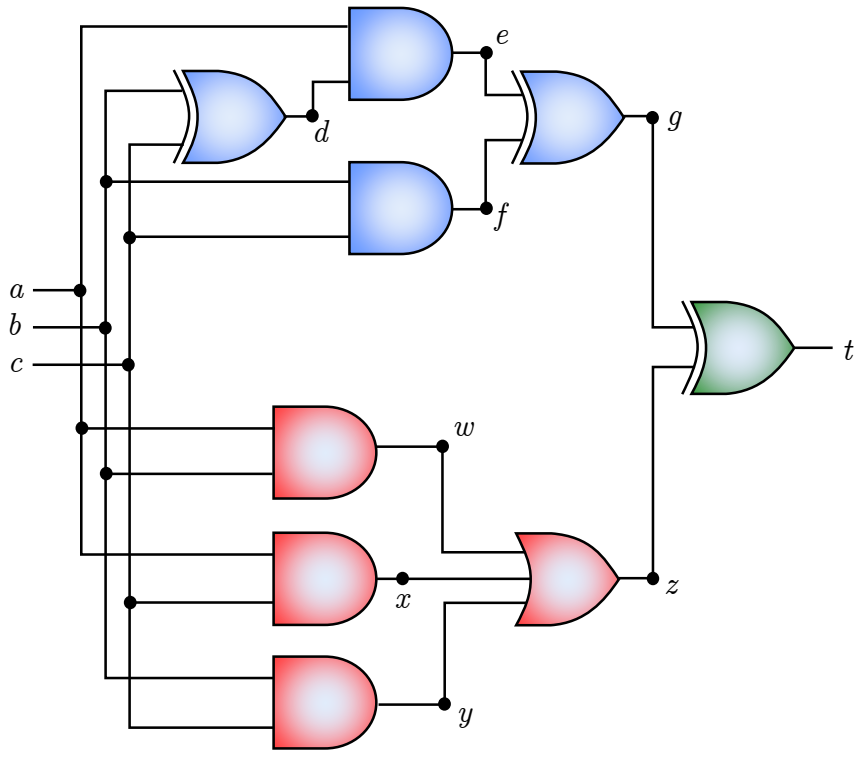


Figure 2.5: A miter to test two implementations of the majority function.

2.2.3 Using Integers in Boolean SAT

Most digital systems use integers which each have multiple bits. Conversely, SAT is limited to Boolean variables. The technique of "bit-blasting" can be used to represent one integer with several Boolean variables. For example, consider this Verilog code which describes a 4-bit adder in hardware:

```
module adder(a, b, s);
input [3:0] a;
input [3:0] b;
output [3:0] s;
assign s = a + b;
endmodule
```

We can introduce Boolean variables a_3, a_2, a_1, a_0 for `a`, and likewise for `b` and `s`. For simplicity, assume this is implemented as a ripple-carry adder. We will also need Boolean variables c_4, c_3, c_2, c_1, c_0 to represent the value of each intermediate carry signal. The logic for a ripple-carry adder is as follows. For $n = 0, 1, 2, 3$:

- Sum bit: $s_n = a_n$ XOR b_n XOR c_n.
- Carry bit: $c_{n+1} = \text{majority}(a_n, b_n, c_n)$.

We assign the initial carry $c_0 =$ false, and c_4 is not computed since it is not used. The logic for the majority function was shown in the example from section 2.2.2.

By using bit blasting, Boolean SAT can be used to find a feasible point in an integer linear programming problem, for example. Extensions of SAT into higher-level logic are known as Satisfiability Modulo Theories (SMT). SMT solvers typically follow

one of two strategies: so-called "eager" SMT solvers use bit blasting to obtain an equivalent Boolean SAT problem whereas "lazy" SMT solvers will internally use a "theory solver" to try to solve the problem in its high-level format (without bit blasting). The extensions that SMT provide over SAT are *orthogonal* to the contributions from this thesis. We refer the reader to [23, 24] for a comprehensive analysis of SMT.

2.3 Local Search SAT Algorithms

One way to solve SAT is to simply take a guess at how the variables should be assigned and then check if it satisfies the problem. If not then make another guess, and keep repeating until a solution is found. This is the basic strategy used in local search SAT algorithms, such as GSAT [25] and WalkSAT [26].

After a random assignment to all variables is initialized, typically local search algorithms iteratively flip the assignment of one variable (true becomes false and vice versa) until the variable assignments satisfy all clauses. Algorithms differ in how they choose which variable assignment to flip. GSAT flips the variable which minimizes the number of unsatisfied clauses using the new assignment. WalkSAT uses a similar strategy, but it first randomly chooses one unsatisfied clause to narrow the candidates of which variable may flip. Randomized restarts are also used, which is analogous to simulated annealing. A survey of more recent techniques is provided in [27].

Local search SAT algorithms are often referred to as *incomplete* SAT algorithms since they cannot prove a problem is unsatisfiable. Guaranteeing coverage of the entire search space requires storing an exponential number of previous assignments or using of a pre-defined variable order. Either would result in poor algorithm performance. Even so, not being able to prove unsatisfiability can be irrelevant for some applications.

Constrained random generation [28] is one such example. Suppose one wants to stress test a network card with random data. Arbitrary random data may not form legal network packets. To test it properly, one must ensure that the random data is conformant, e.g. certain fields in the network packet header must be set to certain legal values. In order to provide enough test coverage, the corresponding SAT problems must inherently be loosely constrained (and therefore easily satisfiable).

2.4 Unsatisfiable-Based SAT Algorithms

Goldberg introduced the stable set of points (SSP) method [29], which can prove a SAT problem is unsatisfiable by induction. The premise for this algorithm is that if a modern SAT solver is able to prove a problem with millions of variables is unsatisfiable within a reasonable amount of time, typically this is due to only a *small fraction* of the variables that actually participate in the conflict (e.g. a few thousand variables).

As a contrived example (the color scheme matches Figure 2.6):

$$(a+b)(a+\overline{b})(\overline{a}+b)(\overline{a}+\overline{b})(a+c+d)(\overline{b}+e)(f+\overline{g}+h)\ldots(x+y+z)$$

is unsatisfiable because of only the first four (red) clauses. Additional (blue) clauses may expand the search space since they contain more variables, however they do not influence the unsatisfiability (which only variables a and b affect). The search space of this problem is illustrated in Figure 2.6. If we could direct the SAT solver to stay within the search space of only a and b (the red region of Figure 2.6), we could obtain a minimalistic proof that the problem is unsatisfiable. Naturally we would expect such an algorithm to run faster than if it also had to visit much of the blue region.

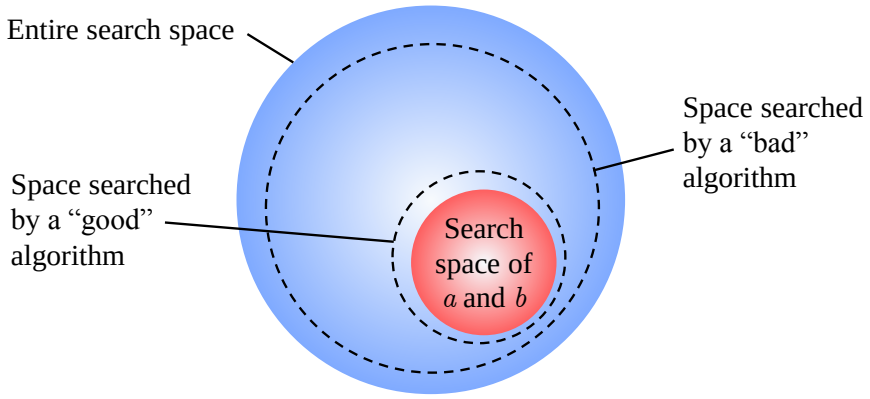


Figure 2.6: The search space for a subset of variables is significantly smaller.

Notice that local search algorithms (section 2.3) explore over unsatisfiable points since they terminate as soon as a satisfying variable assignment is found. However, if we can prove that these unsatisfiable points form a closed (self-containing) set, then problem is necessarily unsatisfiable. This is the main idea of SSP.

Continuing the above example, suppose we start a search with all variables assigned false. The first clause $(a + b)$ would not be satisfied. To fix this, we must flip the assignment of a or b. In general, by recursively looking at all possible updates that a local search algorithm could make (in this example, this would include flipping both a and b), we try to show that we can never escape the unsatisfiability. Like local search, we can also stop as soon as a satisfying variable assignment is found.

To improve the efficiency, the stable set can be searched in clusters [30] by using partial variable assignments (only some variables are assigned, the others are in a "don't care" state). Continuing the above example, assigning $a = 0$, $b = 0$, and leaving all other variables unassigned is still unsatisfiable. In other words, no matter how we assign the remaining variables, it is impossible to produce a satisfying assignment. However, it is unclear exactly how the clusters are represented and we think that the memory complexity could be exponential with respect to the number of variables.

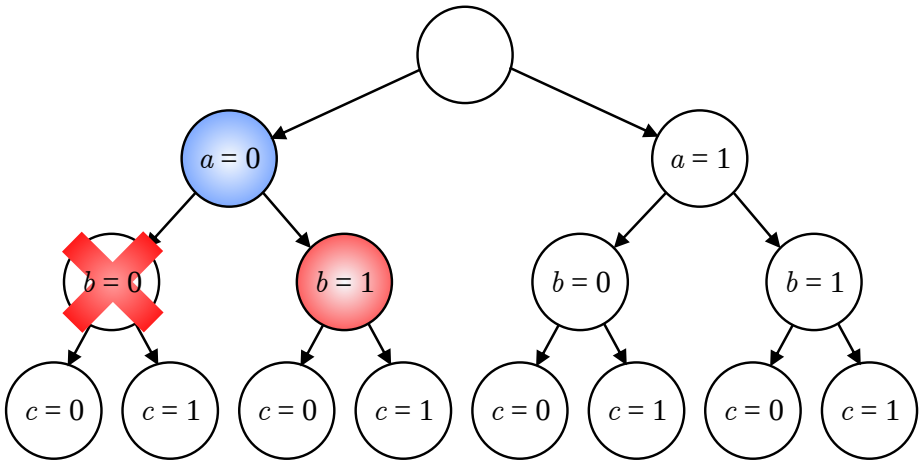


Figure 2.7: We begin the search on the tree of all possible variable assignments by assigning $a = 0$. The clause $(a + b)$ then implies $b = 1$ must also be assigned.

2.5 The DPLL Algorithm and its Enhancements

2.5.1 The Basics of DPLL

The Davis-Putnam-Logemann-Loveland algorithm [4] was created in the 1960s before SAT was proven NP-complete, yet it still serves as the basis for many modern SAT solvers (which also have sophisticated enhancements). DPLL performs a depth-first search on the tree of all possible variable assignments. We illustrate this with an example. Suppose our SAT problem is:

$$(a + b)(a + \overline{b} + \overline{c})(a + \overline{b} + c)(\overline{a} + b + c).$$

In Figure 2.7, we begin the tree search by assigning $a = 0$. At this point, all other variables are unassigned. If we were to then also assign $b = 0$, it would be impossible to satisfy the problem regardless of the assignment to the remaining variables. The clause $(a + b)$ implies that we must assign $b = 1$ given that $a = 0$.

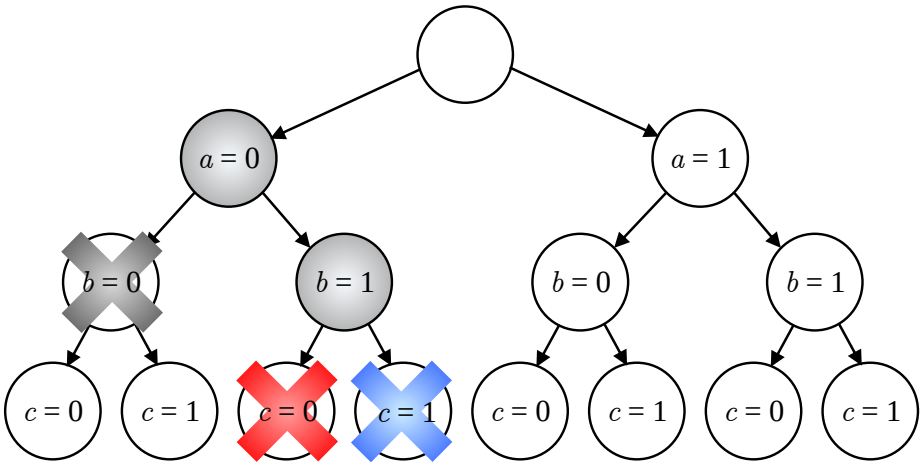


Figure 2.8: Under the partial assignment $a = 0$ and $b = 1$, the clause $(a + \overline{b} + \overline{c})$ implies $c = 0$ which cuts the $c = 1$ subtree, likewise $(a + \overline{b} + c)$ implies $c = 1$.

DPLL directs its search by trying to maintain satisfiability, as opposed to SSP from section 2.4 which directs its search in the unsatisfiable region of the space. DPLL prunes branches of the search tree once they are known to be unsatisfiable. This is done with Boolean constraint propagation (BCP). In general, constraint propagation is not restricted to just Boolean clauses. It operates in an intuitive manner:

If we have not yet satisfied a clause, and

If all variables in that clause are assigned except for one variable,

Then we must assign that last variable in the way that will satisfy the clause.

Continuing the above example, we now have $a = 0$ and $b = 1$. Following the rules of BCP, the clause $(a + \overline{b} + \overline{c})$ implies $c = 0$. This is illustrated by the blue X in Figure 2.8, which prunes the $c = 1$ subtree. Likewise $(a + \overline{b} + c)$ implies $c = 1$, so the red X cuts the $c = 0$ subtree. Having pruned both subtrees, it is clear that $a = 0$ and $b = 1$ cannot lead to a satisfying assignment.

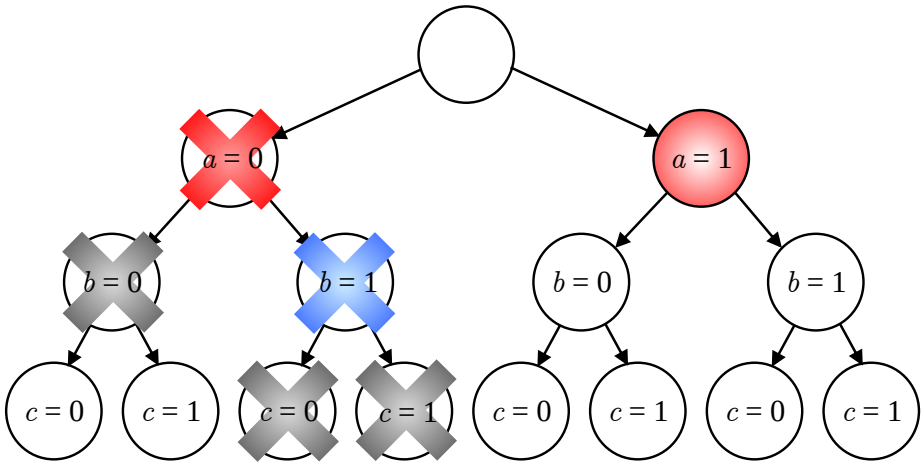


Figure 2.9: Backtracking, as a result of a conflict, helps to prune the search space.

Conflicts are useful because they help to prune the search tree. Now we must backtrack the search, as shown by the blue X in Figure 2.9. With only $a = 0$ assigned, notice that $b = 0$ was earlier pruned by BCP and now we have just pruned $b = 1$, therefore we continue backtracking. Eventually we conclude that if this problem actually has a satisfying assignment, it must have $a = 1$. This is shown by the red parts of Figure 2.9. Note that $a = 1$ is implied (it is not a search decision).

Upon inspection of the clauses, there are no variables that can be implied. Therefore we continue the tree search by assigning $b = 0$. The clause $(\overline{a} + b + c)$ implies $c = 1$, as shown in Figure 2.10. Once we reach a leaf node, the path from here to the root of the tree specifies a satisfying assignment. Otherwise we would eventually prune the entire tree, in which case the problem would be unsatisfiable.

One major degree of freedom in DPLL is the order in which variables are assigned. For example, we could have instead examined variable b first. Furthermore, different subtrees may have a different variable ordering, as illustrated in Figure 2.11. If there is nothing left to imply, any unassigned variable in the current subtree can be selected

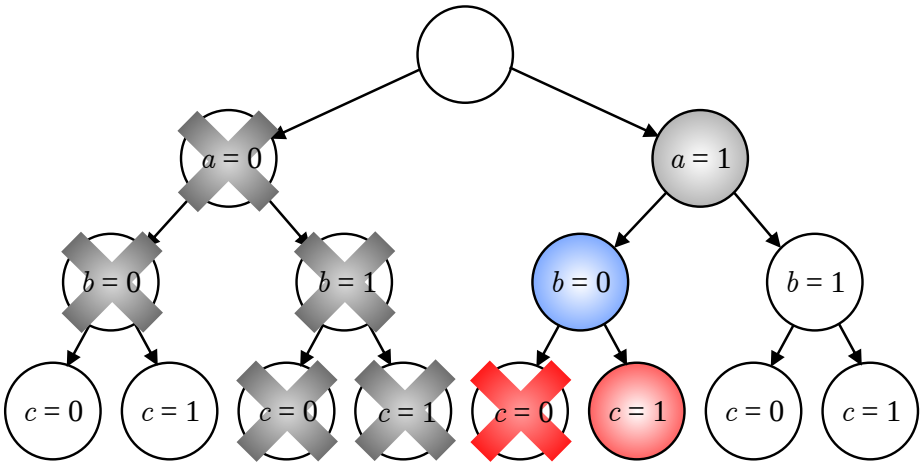


Figure 2.10: Reaching a leaf node means that a satisfying assignment has been found.

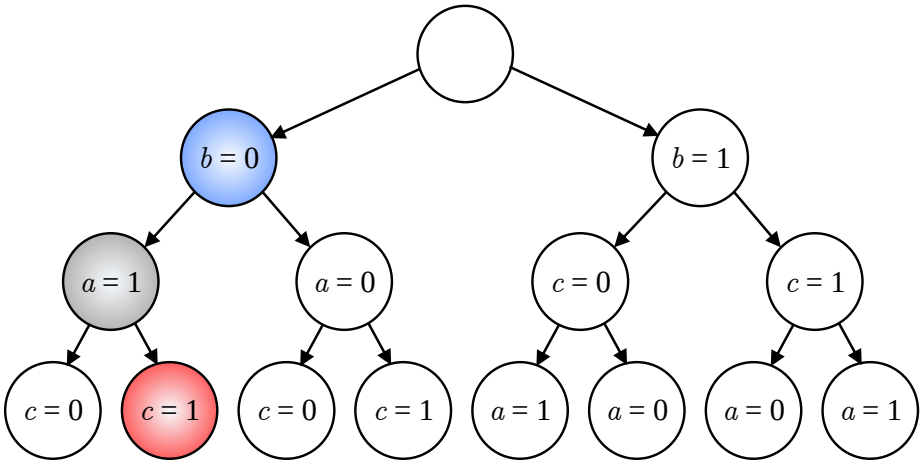


Figure 2.11: Variables can be assigned in any order so long as the entire space is enumerated. The same satisfying assignment from Figure 2.10 is shown here.

for assignment, and we can choose whether to first assign it true or false. Determining the best ordering of variables is an open research problem and it has a significant impact on the effectiveness and practicality of DPLL-based SAT solvers. One very commonly used heuristic is VSIDS, which we discuss in section 2.5.2.4.

2.5.2 Key Enhancements to DPLL

2.5.2.1 Non-Chronological Backtracking, Conflict Analysis, and Learnt Clauses

Conflicts closer to the root of the search tree result in a larger amounts of pruning and therefore are more beneficial. Maintaining a purely depth-first search can hinder this. As example, consider this SAT problem (only the blue clauses will be involved in what we will demonstrate):

$$(a + y + \overline{z})(x + \overline{y})(y + z)(\overline{b} + \overline{c})(\overline{d} + \overline{e} + \overline{f}) \ldots (\overline{v} + \overline{w})$$

Suppose a SAT solver assigns the variables as false in alphabetical order. We start by assigning $a = 0$, and no variables can be implied all the way to assigning $w = 0$. When we then assign $x = 0$, the second clause $(x + \overline{y})$ implies $y = 0$, then the first clause $(a + y + \overline{z})$ implies $z = 0$, yet the third clause $(y + z)$ cannot be satisfied. Figure 2.12 shows a graph of the dependencies. A conflict means we made a bad guess at the last assignment. In this case, we deassign x, y, and z, and then we can imply $x = 1$.

However, the search remains in the $a = 0$, ..., $w = 0$ subtree, so not much pruning is achieved. Notice the other variables (b through w) did not participate in the conflict. Essentially we have pruned the $a = 0$ and $x = 0$ subtree, which is a full one quarter of the search space. To obtain stronger pruning, we could jump the search to $a = 0$ and $x = 1$ (with none of the other variables assigned). This idea of conflict-driven "non-chronological backtracking" was introduced by the GRASP SAT solver [31].

To determine where we can jump the backtracking to, a root cause analysis is required when we encounter a conflict. Zhang *et al.* [32] introduced conflict analysis

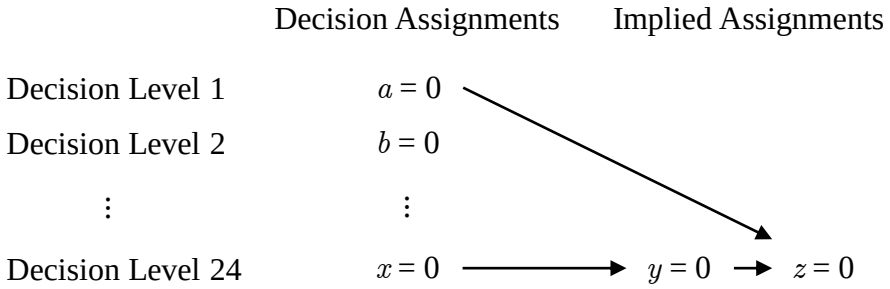


Figure 2.12: A conflict can be root-caused by analyzing the dependencies of implied assignments. Decision level refers to how deep in the search tree we currently are.

using the unique implication point (UIP). For instance, looking at the dependencies in Figure 2.12, only $a = 0$ and $y = 0$ would also produce the same conflict. UIP-based analysis favors this since it is closer to the conflict than our earlier example of $a = 0$ and $x = 0$. Notice that if we jump the search back to $a = 0$ and $y = 1$ instead, we can still imply $x = 1$ due to $(x + \overline{y})$. However, if we jump the search back to $a = 0$ and $x = 1$ as originally described, we know nothing about y since $(x + \overline{y})$ is satisfied.

Conflict analysis also produces a "learnt clause" to help us avoid the same conflict in the future. Notice we can add the clause $(a + y)$ to the problem without changing the satisfiability of the original problem. If we had known this clause earlier, as soon as $a = 0$ was assigned, we would have implied $y = 1$, essentially bypassing the conflict.

Adding too many learnt clauses can also slow down a SAT solver, since for each variable assigned there would be many more clauses that now need to be inspected. How to best manage the learnt clauses database is an open research problem [33].

2.5.2.2 Randomized Restarts

The depth-first nature of DPLL has another key problem. If one makes the wrong choice on a variable assignment near the root of the search tree, it is difficult to fix

because the entire subtree needs to be proven unsatisfiable before the bad variable assignment can be fixed. SAT solvers could waste lots of time if, for example, the opposite assignment easily leads to a satisfying assignment.

Randomized restarts can mitigate this problem, as introduced in [34]. To maintain the completeness of DPLL, the length of the search before the next restart grows over time in an unbounded manner. Learnt clauses can be preserved between restarts.

2.5.2.3 Watched Literals Lists

BCP is a key pruning mechanism, however it consumes a significant portion of the CPU time in DPLL-based SAT solvers. To improve this, if we somehow knew that a Boolean clause contained at least two unassigned variables, there is no possibility of producing a BCP. This idea was introduced in the SATO SAT solver [35] and then formalized into the modern "watched literals lists" by the Chaff SAT solver [36].

We illustrate the concept of watched literals lists with an example in Figure 2.13. For each clause, any two unassigned literals must be watched (the definition of a "literal" was provided in section 1.1.3). Suppose we arbitrarily decide to watch the first two literals of $(a + b + \overline{c} + d + \overline{e})$. First we assign $a = 0$. Since a is currently a watcher, we visit the clause. Watchers must be unassigned, so we need to find a new watcher (in this case, $\overline{c}$ becomes the new watcher and a will no longer be watched).

Suppose we assign $d = 0$ next. Since d is not a watcher, we can skip inspecting the clause. BCP is guaranteed to be impossible since our two watchers, b and $\overline{c}$, indicate that at least two variables are unassigned. Recall that BCP only happens when we reach the last unassigned variable. Likewise, when assigning $e = 1$, we can also skip the inspection. This lazy checking for BCP improves the compute efficiency.

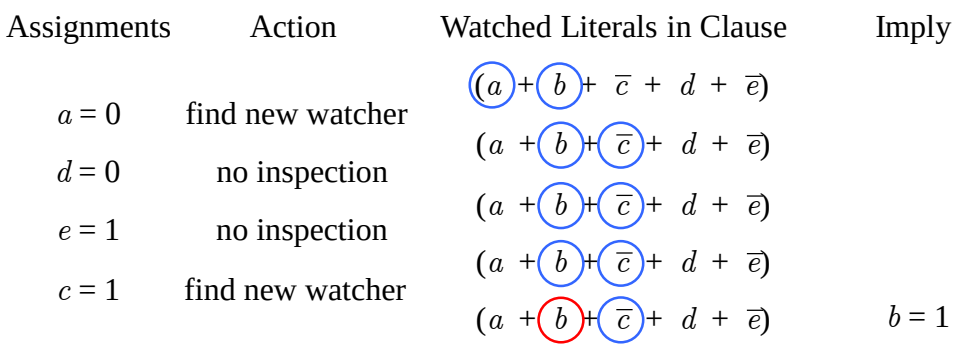


Figure 2.13: As we assign variables in the clause without satisfying it, we either find a new watcher (blue) or there is none left so we must imply a variable (red).

Finally, when we assign $c = 1$, we try to find a new watcher and fail. In other words, we could not find two unassigned variables. Therefore the last unassigned variable must be the other watcher (not the one we were trying to update, in this example it is b). If the clause is not yet satisfied then we must imply this variable, which is shown by the red circle in Figure 2.13.

2.5.2.4 Guiding the Choice of Which Variable To Assign

After implying all variables possible, the DPLL tree search continues by choosing the next variable to assign. One common method for guiding this is the variable state independent decaying sum (VSIDS) heuristic from the Chaff SAT solver [36].

With the advent of non-chronological backtracking, the search within DPLL is partly guided by conflicts. The intuition behind VSIDS is that variables which often participate in conflicts are more "active" in determining the satisfiability of the problem. When determining the next variable to assign (from the set of all currently unassigned variables), VSIDS chooses the most active variable. Each variable has an activity associated with it. Every time we encounter a conflict and therefore do a

root cause analysis of it, every variable that participated in the conflict will get its corresponding activity incremented.

The activity of all variables can be divided by 2 periodically (e.g. every 100 conflicts) to favor variables that participated in more recent conflicts. Newer SAT solvers, such as Minisat [37], divide the activity of all variables by 5% after each conflict. This provides a smoother favoring of more recent conflicts. Minisat's implementation does not sweep over all variable activities to divide them, but instead the amount to increment each activity by grows by 5% after each conflict. When choosing which variable to assign, the relative activities between different variables are compared.

2.5.3 Modern Enhancements to DPLL

Many of the major enhancements in modern DPLL (from the previous section) happened in the early 2000s when computing finally reached the threshold where SAT became practical. Since the breakthrough of Chaff in 2004, the algorithmic advancement of SAT has been slowing. Further innovations beyond so much cumulative smartness become increasingly difficult, yet there are some notable improvements.

The importance of data structure optimizations should not be understated. There are major practical implications for whether a memory read completes in 1 ns (e.g. from CPU cache) or in 100 ns (e.g. random access from DRAM). PicoSat [38] and its successor Lingeling [39] are two SAT solvers that have consistently performed well in the international SAT competitions [33]. A significant effort has been placed on such optimizations. For example, since the memory used by SAT solvers is typically far below 4 GB, 32-bit pointers can be used (even on 64-bit platforms) to facilitate more caching. Special data structures are used for clauses with 2 variables and with 3

variables since most BCPs are produced by these. The Siege SAT solver [40] would use bit fields to store a clause with 3 variables within a single 64-bit integer.

Many enhancements have been introduced for restarts. RSat [41] introduced "phase saving", which upon deassignment caches whether each variable was true or false. The next time a variable is assigned, RSat reuses its previous polarity since this tends to produce more conflicts. The Luby sequence [42] minimizes the expected time of Las Vegas algorithms (randomized algorithms that produce the correct answer once they terminate, but the run time is a random variable). Many modern SAT solvers use this as a guideline for how many conflicts are needed before the next restart [43].

Minisat also introduced learnt clause minimization [44]. After a learnt clause is produced from conflict analysis, a second pass with deeper analysis tries to remove redundant variables from the newly learnt clause. With fewer variables, a learnt clause is more likely to produce a BCP in the future.

Finally, some applications like cryptography tend to have symmetric constraints. For instance, $x = y$ XOR z can be encoded with 4 clauses:

$$(\overline{x} + \overline{y} + \overline{z})(\overline{x} + y + z)(x + \overline{y} + z)(x + y + \overline{z}).$$

XOR constraints have the property that they will produce a BCP if all except for one variable are assigned, regardless of the sign of each variable. The sign of the newly implied variable depends on the parity. Instead of treating these clauses independently, a 2x2 group of watched literals lists can natively support XOR clauses within the SAT solver. This is done in the SAT solvers MoRSat [45] (the successor of RSat) and CryptoMinisat [46] (as a modification built on top of Minisat). Both have performed well in the SAT competitions [33].

2.6 Preprocessing

2.6.1 Simplification

The goal of preprocessing is to apply transformations to a SAT problem to make it "easier" to solve but without changing whether the original problem was satisfiable or not. The exact meaning of easier varies for different SAT solvers, but empirically it typically means reducing the number of variables and reducing the number of clauses [47]. Our own preprocessor is discussed in section 8.2.

A unit clause contains only a single variable, so that lone variable must be assigned in a way that satisfies the clause. Preprocessors often remove unit clauses with BCP.

Suppose we find two clauses of the form $(a + \overline{b})(\overline{a} + b)$. Any satisfiable solution (if one exists) will have the same assignment to a and b. If one were false and the other were true, one of these clauses would be unsatisfied. All occurrences of a can be replaced by b (likewise for negative literals), thus effectively eliminating one variable.

Variables can also be eliminated by resolution [48]. Suppose we have two clauses of the form $C_1 = (x + a_1 + \ldots + a_n)$ and $C_2 = (\overline{x} + b_1 + \ldots + b_m)$. The resolvent of C_1 and C_2 is:

$$C = C_1 \otimes C_2 = (a_1 + \ldots + a_n + b_1 + \ldots + b_m).$$

Variable x can be eliminated as follows. Let S_x represent the set of all clauses in the SAT problem that contain positive literal x. Likewise, $S_{\overline{x}}$ is the set of all clauses that contain negative literal $\overline{x}$. We can replace the clauses from S_x and $S_{\overline{x}}$ with:

$$S = S_x \otimes S_{\overline{x}} = \{\ C_x \otimes C_{\overline{x}}\ \mid\ C_x \in S_x,\ C_{\overline{x}} \in S_{\overline{x}}\}.$$

Basically we apply resolution to every possible pairing between a clause that contains x and a clause that contains $\overline{x}$. The resulting set of clauses S captures the relationship among the *remaining* variables, hence these replace S_x and $S_{\overline{x}}$ thereby eliminating x.

Eliminating a variable reduces the DPLL search space size. However, if x and $\overline{x}$ occur in many clauses, the benefit of eliminating it can be outweighed by the excessive number of clauses produced, e.g. a SAT solver may take longer to solve the problem.

2.6.2 Assisting the SAT Solver

Sometimes adding clauses to a SAT problem (while preserving its satisfiability) can help a SAT solver run faster. One case that we examine is hyper resolution [49]. Suppose a SAT problem has the following clauses:

$$(a + b + c + d)(h + \overline{b})(h + \overline{c})(h + \overline{d})$$

If we assign $h = 0$, the last three clause imply $b = 0$, $c = 0$, and $d = 0$. The first clause can now imply $a = 1$. Because $h = 0$ eventually implies $a = 1$, we can add the clause $(h + a)$ without affecting the satisfiability.

Without this "learnt clause", assigning $a = 0$ does not lead to any BCP since the first clause has three unassigned variables. With the new clause $(h + a)$, a SAT solver is now capable of implying $h = 1$ from $a = 0$. In general, hyper resolution looks for $(x_1 + x_2 + \ldots + x_n)(h + \overline{x}_2) \ldots (h + \overline{x}_n)$ in order to add the learnt clause $(h + x_1)$.

Preprocessors essentially look for pre-defined patterns that are known to have a representation with fewer variables or clauses, or patterns that can be augmented to assist a SAT solver. We refer the reader to SatElite [47], blocked clause elimination [50], Niver [48], and Coprocessor [51] for more sophisticated preprocessing techniques.

2.6.3 Regulation of Clause Size and Variable Occurrence

Any SAT problem can be converted to an equivalent SAT problem where each clause has only up to 3 variables. A dummy variable can be used to split a large clause with $n+m$ variables into two smaller clauses with $n+1$ and $m+1$ variables. For example, $(a+b+c+d+e)$ is satisfiable if and only if $(a+b+x)(\overline{x}+c+d+e)$ is satisfiable. Applied recursively, it can be shown by induction that all large clauses can be brought down to 3 variables. Continuing the example leads to $(a+b+x)(\overline{x}+c+y)(\overline{y}+d+e)$.

The number of clauses that each variable occurs in can also be limited. This is done by creating equivalent variables. For example, if x occurs in 20 clauses, we could replace 10 of those occurrences with a dummy variable y as well as introduce 2 new clauses: $(x+\overline{y})(\overline{x}+y)$. Now x and y each occur in 12 clauses. Applied recursively, the variable occurrence can be limited to 5 (provable by induction).

Such transformations could be beneficial, for example, if specialized software data structures are used for clauses with 3 variables. However, this could be outweighed by the excessively large number of dummy variables created by this transformation, especially for SAT problems with mostly large clauses.

For hardware SAT solvers, these properties enable simpler designs. For example, if we enforce that clauses may only have up to 4 variables, in order to read an entire clause in one clock cycle, we only need 4 parallel memory controllers. Without this limit, since hardware has a finite number of memory controllers, the design would have to support reading one clause over several clock cycles, which complicates the design (e.g. we now have state information). Optimizations for software are not necessarily beneficial to hardware. Techniques such as variable elimination can be applied in *either direction*, depending on the optimization needed for a specific implementation.

2.7 Our Context of SAT Algorithms

Throughout this chapter, we surveyed and demonstrated applications of SAT as well as discussed algorithms for solving SAT. As mentioned in section 1.3, we make no claim of improving SAT algorithms in this thesis. Nonetheless, it is vital to understand the commonly used approaches, thereby enabling us to restructure the computation in order to maximize acceleration with custom hardware.

This restructuring is thoroughly examined in chapter 4. However, we first need to choose the appropriate underlying technology in order for acceleration to be practical. This is discussed in chapter 3.

Chapter 3

Motivations and Prior Work in Relation to Current Technology

This chapter discusses the use of hardware-accelerated SAT to augment the practicality of SAT. This is *orthogonal* to the development of good algorithms for solving SAT, which was presented in the previous chapter. We begin this chapter by motivating the use of custom hardware for SAT based on current technology trends. We then survey prior works on hardware SAT in relation to current technology. These discussions also serve as motivating factors for many of our own design decisions, which we summarize at the end of this chapter.

3.1 Motivation for Hardware-Accelerated SAT

SAT is at the core of numerous applications, as discussed throughout section 2.1. However, SAT solvers must become computationally efficient enough to become practical in new applications before SAT will see widespread adoption.

Table 3.1: Transistor scaling. The size scales by $S \approx 1.4$ per generation. The post-Dennard voltage scales by ν which is significantly smaller than S, typically $1.0 \leq \nu \leq 1.1$.

Transistor property (relative)	Dennard	Post-Dennard
N = number of transistors	S^2	S^2
F = clock frequency	S	S
C = capacitance	$1/S$	$1/S$
V = voltage	$1/S$	$1/\nu$
P = total power $\propto NFCV^2$	1	$S^2/\nu^2 \approx S^2$
U = utilization given fixed power budget	1	$\nu^2/S^2 \approx 1/S^2$
Raw computing ability $\propto NF$	S^3	S^3
Usable computing ability $\propto NFU$	S^3	$S\nu^2 \approx S$

Many proposals for accelerating SAT in hardware emerged in the early 2000s, around the time when the computing power reached the threshold where SAT became practical and lots of research resulted in the much of the smartness in today's SAT solvers. Although we currently do not see widespread adoption of hardware SAT, it does *not* imply that SAT is unsuitable for hardware. In fact, it is now very much the opposite due to *major changes in the computing technology* since then.

3.1.1 Moore's Law and its Complications

Moore's Law [52] is an observation from 1965 that every two years, the number of transistors per silicon die would double. Half a century later, this more or less still holds, although recently the interval appears to be slowing to three years per doubling.

Halving the area per transistor requires both the height and width to decrease by a factor of $S = \sqrt{2} \approx 1.4$. Dennard *et al.* [53] modelled how shrinking transistors would affect computing, as summarized in Table 3.1. With each generation of Moore's Law, smaller transistors can be clocked about 40% faster. Twice as many transistors gives a raw increase in computation by a factor of $S^3 \approx 2.8$.

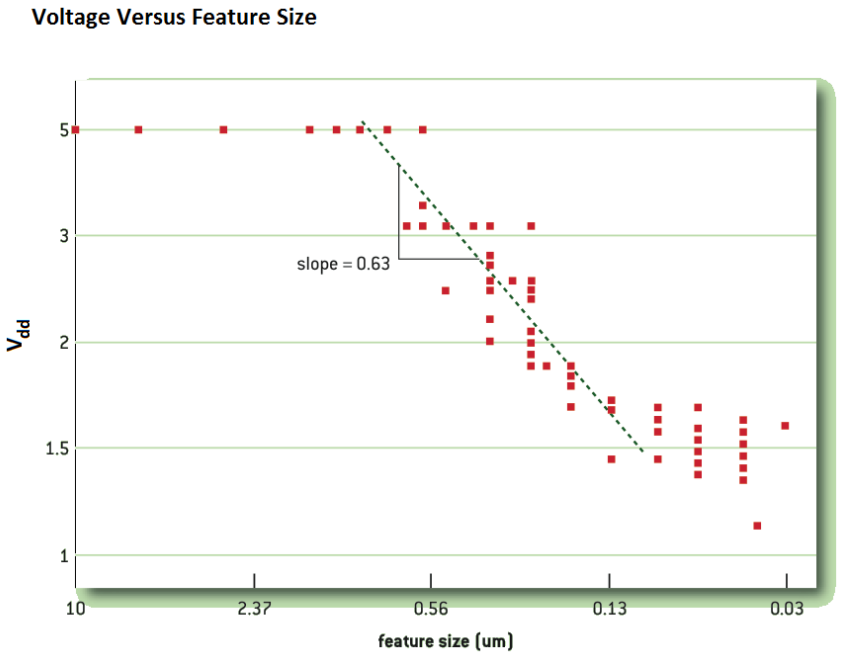


Figure 3.1: The trend of voltage versus transistor size (Figure 6 reproduced from [54]).

A 40% reduction in capacitance reduces the power by 40% at the same voltage. A plot of voltage versus transistor size is shown in Figure 3.1 (reproduced from [54]). As power became important (around year 1995 at the 600 nm node), voltage began to scale down. In theory the voltage could be scaled down (by $1/S$, see Table 3.1) so that even though we had more transistors, the total power would remain constant. However, in reality voltage scaling has not kept pace with the increase in frequency, thus power has increased over time. This is shown in Figure 3.2 (also from [54]).

In recent years (beyond the 130 nm node, which was around year 2004), we now follow the post-Dennardian scaling in Table 3.1. As physical sizes continue to shrink, insulators become increasingly leaky. Allowing current to flow consumes power. To address this, the voltage can no longer scale down by the historical $1/S$ [55]. As shown in Figure 3.1, the voltage now remains almost constant.

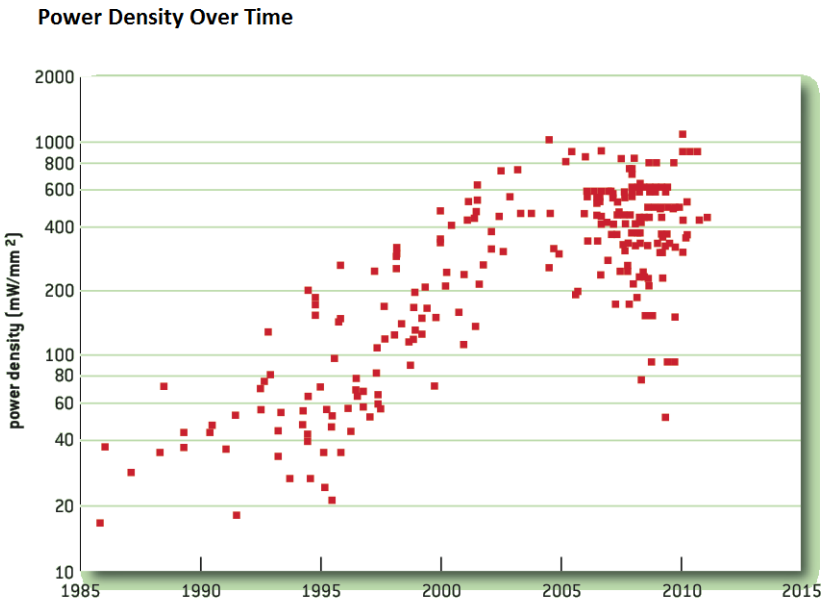


Figure 3.2: The trend of power density over time (Figure 8 reproduced from [54]).

With nearly constant voltage, power must exponentially increase to maintain the rate of computing improvement. Power dissipation also generates heat which will damage the silicon, and large scale cooling systems are impractical. This so-called "power wall" was the driving force that caused CPU frequencies to stop increasing around 2005, as shown in Figure 3.3.

With a limited power budget, performance continued to improve by doing more *useful* computation per transistor. In single-core CPUs, performance was improved by using many transistors as "helper" logic, such as branch predictors and speculative execution [56]. This was power inefficient, so multicore CPUs took back many of those helper transistors (by using a simpler branch predictor) and instead used them to make a second CPU core which can spend much more time doing useful work.

Multicore soon encountered the so-called "memory wall". External bandwidth to memory was not scaling as quickly, thereby causing data starvation. According

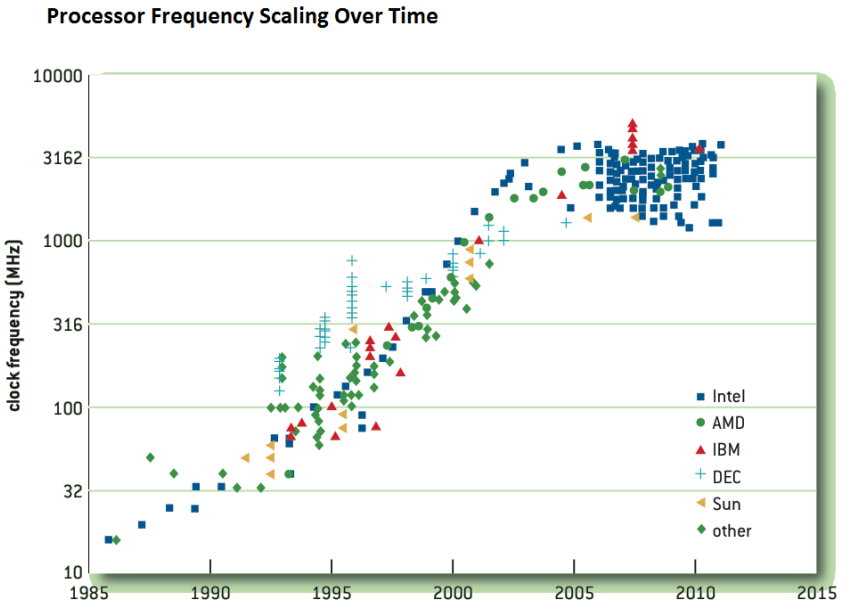


Figure 3.3: The trend of CPU speeds (Figure 7 reproduced from [54]).

to the JEDEC standards for DRAM, speeds have doubled approximately once every five years over four generations of DDR [57]. The data starvation problem lead to the development of extremely large and sophisticated CPU caches, which typically consume more than half of the transistors in CPUs from recent years [54].

Multicore is not the long-term answer. Although it did address the immediate problem of doing more useful work per transistor, it did not address the fact that for a limited power budget we must be *more energy efficient per transistor* in order to actually use more transistors. This lead to the so-called problem of "dark silicon", where we can now put more transistors on a die than the number we can power [55,58]. Perhaps more appropriately called "dim silicon", it is not wasted area but refers to transistors that cannot be clocked at the maximum frequency all of the time.

In the future of computing, designers will likely use the extra area to regain energy efficiency. For example, with different types of processors present, we can use the one

that is most energy efficient for a given task. Techniques such as clock-gating and power-gating are one-time enhancements [55], so further innovation will be needed.

Even with a limited power budget, Table 3.1 shows that shrinking transistors can still improve performance by a factor of 1.4 per generation, however this is a major shortfall from the traditional factor of 2.8. This is also an exponentially worsening problem, as each generation compounds the effect.

3.1.2 The Suitability of FPGAs for SAT

As we approach physical limitations like power, one solution is to break away from the artificial boundaries imposed by the instruction-based architectures that CPUs use. Field-programmable gate arrays (FPGAs) are *reconfigurable hardware*, meaning we can change the type of gates that the logic implements as well as change how the gates are connected together.

Such reconfigurability comes at an overhead in area and performance, yet this gap has narrowed in recent years and is expected to continue narrowing. CPUs are already power limited whereas FPGAs are not [59, 60]. Also, modern FPGAs contain both programmable logic and hardwired (non-reconfigurable) high-performance logic. Hardwired logic typically includes: embedded memories, multipliers, and high-speed external interfaces. These factors now give hardware SAT a more competitive edge against software.

FPGA clock speeds have typically been an order magnitude slower than CPUs, however this gap will likely narrow. A slower clock is an advantage in terms of power. To compensate for a slower computation clock, FPGAs can use specialized control paths to perform the equivalent of several instructions within a single clock cycle.

Table 3.2: Maximum capabilities of modern FPGAs.

FPGA Family	Year Announced	Logic Cells	On-chip SRAM
Altera Stratix IV	2008	813k	33 Mbits
Altera Stratix V	2010	1200k	63 Mbits
Altera Stratix 10	2013	>4000k	?
Xilinx Virtex 6	2008	758k	46 Mbits
Xilinx Virtex 7	2010	1955k	85 Mbits
Xilinx Virtex Ultrascale	2013	4400k	115 Mbits

Much of the computation in a SAT solver is *fundamentally serial*, as discussed in detail in section 3.5.1. Data-parallel architectures, such as general purpose graphics processing units (GP-GPUs), inherently perform poorly on this type of computation.

Larger CPU caches have and will likely continue to improve software performance, however that is about where the benefits end. Techniques for accelerating serial computation, such as branch prediction and speculative execution, are inherently power inefficient and would simply exacerbate the dark silicon problem.

Moore's Law has enabled FPGAs to double in logic capacity every two years. This trend will likely continue in the foreseeable future, as dark silicon is not yet a concern for FPGAs and transistor sizes are a few generations away from the atomic scale. Also, new techniques such as die stacking [61] may further prolong Moore's Law.

FPGA on-chip memory is directly related to the number of transistors. We expect this to grow faster than the size of SAT problems since SAT is NP-complete. This offers the opportunity to move what was previously stored in slower memory into now faster memory since there is enough faster memory available. SAT is a memory bound problem (as detailed in section 3.3.1), and bringing the data on-chip typically provides a 100× increase in memory bandwidth. *We believe that technology has just reached the threshold where it makes practical sense to attempt this for SAT.*

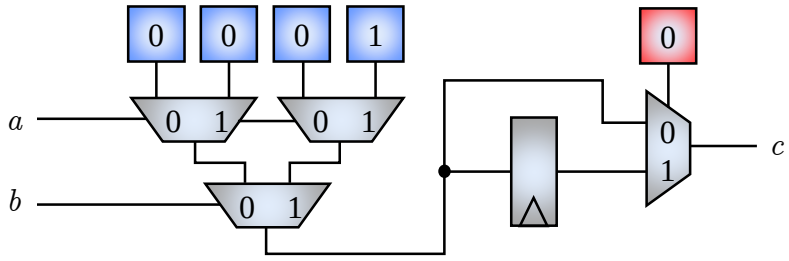


Figure 3.4: SRAM memory enables one to change the logic function implemented (blue) as well as whether a register is used (red). This LUT implements $c = a$ AND b.

To quantify this, in Table 3.2 we show the largest of the previous, current, and next generation FPGAs from Altera [59] and Xilinx [60]. These are by far the two largest FPGA vendors. Current generation FPGAs (Stratix V and Virtex 7) have thousands of embedded memories and the total bandwidth can exceed one terabyte/second. They typically operate in the range of 200-300 MHz while consuming 20-30 watts.

3.2 Unscalable Hardware SAT Designs

3.2.1 Instance-Specific Designs on FPGAs

FPGAs are reconfigurable hardware. The types of gates that the logic implements as well as how those gates are connected together can be changed. FPGAs use lookup tables (LUTs) to implement logic functions. By changing the contents of the SRAM memory inside a LUT, one can implement *any* logic function with up to 4 inputs. As an example, the architecture of a simpler 2-input LUT is shown in Figure 3.4, which implements a 2-input AND gate. Likewise for using registers and connecting the logic together, SRAM bits are used to control the configuration of the interconnect.

FPGAs can be used to create "instance-specific" designs. For each SAT problem, the corresponding hardware source codes are produced. These sources are then

compiled to obtain the FPGA configuration. For a different SAT problem, new hardware source codes are required. The FPGA design can be highly optimized for a specific SAT problem, yet the lengthy compile time makes this approach impractical. Intuitively, to obtain a very fast hardware SAT solver, some of the NP-complete complexity of SAT must be transferred to the FPGA compiler. The compile time was usually significantly longer than the SAT solving time, as practical SAT problem sizes were extremely small (e.g. 100 clauses) when this approach was commonly used.

In fairness, many FPGA designs that used this approach in the late 1990s and early 2000s [62–64] only had significantly smaller FPGAs available. Moore's Law indicates that 7 generations or 14 years ago, FPGAs were over 100× smaller than today. Furthermore, the computer-aided design (CAD) tools, e.g. optimizations within the FPGA compiler, were much less sophisticated than they are today. Consequently, using instance-specific designs enabled FPGA designers to squeeze every last bit of logic and performance out of the tiny and slow FPGAs of the time.

Conclusively, most modern hardware-accelerated SAT solvers represent the problem in *memory*, such as embedded SRAM within the FPGA or external DRAM. A different SAT problem would simply require reloading the memories.

3.2.2 Other Unscalable Designs

Many older non instance-specific designs made practical sense at that time when FPGAs were significantly smaller, however they are unlikely to scale to the large FPGAs of today. For example, in [65], a SAT problem is represented in memory using 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.

A survey of hardware SAT from 2004 [66] shows that almost all of the older works that did make practical sense at the time are not suitable for modern FPGAs. Technology has considerably changed since then, and consequently hardware designs must be *fundamentally rethought*.

3.3 The Importance of Hardware Memory Types

There are major practical implications due to whether a memory access takes 1 ns or 100 ns, for example. In this section, we begin by demonstrating much of the computation in a SAT solver is random access. We then demonstrate how this affects the usability of different types of memory.

3.3.1 The Computation Pattern of BCP

A survey of the international SAT competitions [33] indicates that the majority of SAT solvers are based on DPLL. Although one can get lucky with a guess-and-check approach (this is the basis for local search algorithms from section 2.3), in order to prove unsatisfiability, the entire search space must been covered. An exponential amount of memory is needed to search this in an arbitrary order whereas a depth-first search introduces *hierarchy* to avoid this memory explosion. We believe this is the primary reason why many modern SAT solvers use some variant of DPLL.

Among modern DPLL-based software SAT solvers, the consensus in the SAT research community is that Boolean constraint propagation (section 2.5.1) typically

takes 80-90% of the CPU time. We have also confirmed this with our own experimentation. Consequently, many optimizations have been developed to optimize BCP, such as the watched literals lists (section 2.5.2.3).

BCP is *memory bound* (limited by memory access, not by computation) and requires *random access*. For each variable v assigned, the computation pattern is:

1. Fetch a list of clauses to visit. This could be all clauses that contain v, or we could use an optimized list such as the watched literals list.

2. For each clause to visit, fetch which variables that clause contains.

3. For each of those variables, fetch its assignment.

Only once we have the variable assignments can we check for new variables to imply. Notice that what is fetched in step 1 indicates what to fetch in step 2, and likewise from step 2 to step 3. If we assign a new variable, that new variable will ultimately indicate what needs to be fetched in its own step 1.

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, and

2. Which variable assignments must be fetched for each clause inspected.

It is actually this lack of implicit knowledge that causes so much fetching. In other words, we must *explicitly* store the next item to access because there is no way of constraining its placement in memory (it could be anywhere).

In conclusion, BCP is the dominant computation in most modern SAT solvers and it mostly involves random access to memory.

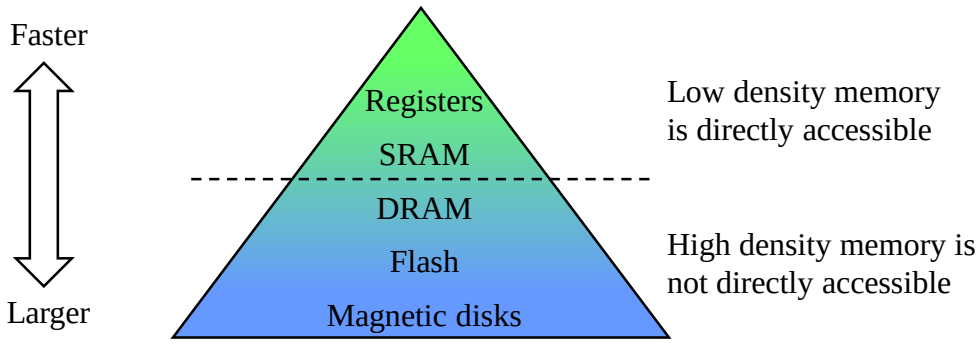


Figure 3.5: Different types of memory offer a tradeoff between speed and capacity.

3.3.2 Memory Speed Versus Capacity

The existence of multiple types of memory technologies enables a designer to choose the appropriate tradeoff between the speed and the capacity of memory. This is illustrated in Figure 3.5. The tradeoff points are *discrete* because each represents a specific technology with distinct underlying physical characteristics, although there is a range of speeds and capacities covered by each technology.

With increasing memory density, fewer electrons are used to store each bit. Eventually sense amplifiers are needed to boost the analog signal strength before the digital logic may access the memory. As shown in Figure 3.5, this threshold exists between SRAM (typically 6 transistors per bit) and DRAM (typically 1 transistor and 1 capacitor per bit, parasitic capacitance can be used instead of an actual capacitor).

The *underlying physical characteristics* affect the random access performance of memory. SRAM incurs no performance penalty for this whereas DRAM incurs a large penalty for random access because DRAM requires sense amplifiers.

DRAM operates as follows. To gain access to a "row" of memory, the row must be opened. The sense amplifiers perform a destructive read from the row of DRAM, capturing a local copy in registers. Once a row is opened, random access reads and

Table 3.3: DRAM random access is about 6% of achievable performance and is comparable with software BCP. The DE4 board has 2 DRAM controllers.

Property	DRAM on FPGA	Software
Memory type and speed	DDR2-800	DDR3-1067
Latency (CL-T_{RCD}-T_{RP}-T_{RAS})	6-6-6-18	7-7-7-20
256 bit random accesses $\times 10^6$ / second	$2 \times 12.5 = 25$	——
256 bit sequential accesses $\times 10^6$ / second	$2 \times 194.5 = 389$	——
Software BCP $\times 10^6$ / second	——	23
CPU cache size	——	12 MB

writes *within* the row are permitted. Once all accesses within the row are complete, the row is closed by writing back the locally updated contents to DRAM. A row must be closed before another one can be opened since reads are destructive.

A DRAM row is small (typically 1 kilobyte) compared to its capacity (typically several gigabytes), thus random access throughout the entire memory would often require a row change. This results in an access pattern that would repeatedly: open a row, perform one memory access, then close the row.

To demonstrate the overhead opening and closing DRAM rows, we did an experiment on the Terasic DE4-230 board [67], which uses an Altera Stratix IV FPGA [59]. We used DDR2-800 DRAM which gives 64 bits of data on both the rising and falling edges of a 400 MHz clock. The FPGA contains hardwired logic for the high-speed external interface. This acts as a gearbox, providing a data interface of 256 bits at 200 MHz (4 times wider and slower, byte enables provide control within a clock cycle).

In our experiment, we performed random access reads in blocks of 256 bits. The DE4 has two independent DRAM controllers. As shown in Table 3.3, only about 25 million random accesses per second were achieved (800 MB/s throughput). Conversely, sequential access (which amortizes the cost of opening and closing rows over numerous

accesses), achieves about 389 million reads per second (12448 MB/s throughput, which is 97% of the 12800 MB/s theoretical maximum). DRAM random access achieves about 6% utilization due to the large overhead for opening and closing rows.

As a second experiment, we ran 12 instances of Minisat [37] at a time on the entire SAT competition 2013 applications benchmark [33] to measure the BCP performance. We used a timeout of 60 seconds. Our test computer uses the Intel Core i7-980 CPU (6 cores, 3.33 GHz, 12 MB L3 cache) with 3 DRAM controllers using DDR3-1067. As shown in Table 3.3, the aggregate performance over 12 threads was 23 million BCP/s.

It is unlikely that each memory access will produce a BCP (several clauses may need to be inspected to produce one BCP). Even in the best case of 25 million random accesses = 25 million BCP/s, this would be barely faster than software.

The DRAM latencies are slightly lower on the CPU (7 clocks at 533 MHz) than on the FPGA (6 clocks at 400 MHz), thus favoring random access on the CPU's DRAM (the latency timings are explained in [57]). The CPU also has an extra DRAM controller and a large on-chip cache, collectively giving the CPU a major advantage.

Not all researchers share our opinion that DRAM is not ideal for DPLL-based SAT solvers. A DRAM-based solution using the BEE3 computing platform is presented in [68]. By using DRAM, they can support significantly larger SAT problem sizes compared to using SRAM. Unfortunately no results were presented in [68]. According to our BCP analysis and DRAM experiment, it would have been difficult for them to provide acceleration over software.

On the other side of the spectrum, [69] stores the SAT problem in registers. With essentially unlimited memory bandwidth, they produced a SAT solver about 2 orders of magnitude faster than others. However, FPGAs typically have 2 orders of magnitude

Table 3.4: A summary of the suitability for SAT for various memory types. SRAM offers the least severe disadvantage and therefore is the best choice.

Memory	Key Advantage	Key Disadvantage (Severity)
Registers	Unlimited memory bandwidth	Extremely limited capacity (3)
SRAM	Good performance on random access	Somewhat limited capacity (1)
DRAM	Sufficiently large capacity	Poor performance on random access (2)

fewer registers than SRAM. Supporting only very small SAT problems limits the practicality. Conversely, implementing an application-specific integrated circuit (ASIC) requires extremely large amounts of time and financial resources.

In conclusion, Table 3.4 summarizes the key advantage and disadvantage for each different type of memory. There is no need to consider memory densities higher than DRAM, as DRAM provides more than sufficient amounts of memory for SAT. We ranked the memories in terms of suitability for SAT based on the severity of the disadvantage, with a lower number being less severe. SRAM offers the highest memory density (to support practical SAT problem sizes) without reaching the threshold of memory density that inhibits random access (by requiring the use of sense amplifiers). *This is our motivation for using SRAM in our hardware SAT accelerator.*

3.4 Heterogeneous Computing

As discussed in section 3.3.1, Boolean constraint propagation is the dominant computation in most modern software SAT solvers, typically consuming 80-90% of the CPU time. In this section, we discuss the advantages and disadvantages of accelerating *only* BCP in hardware versus implementing the *entire* SAT solver in hardware.

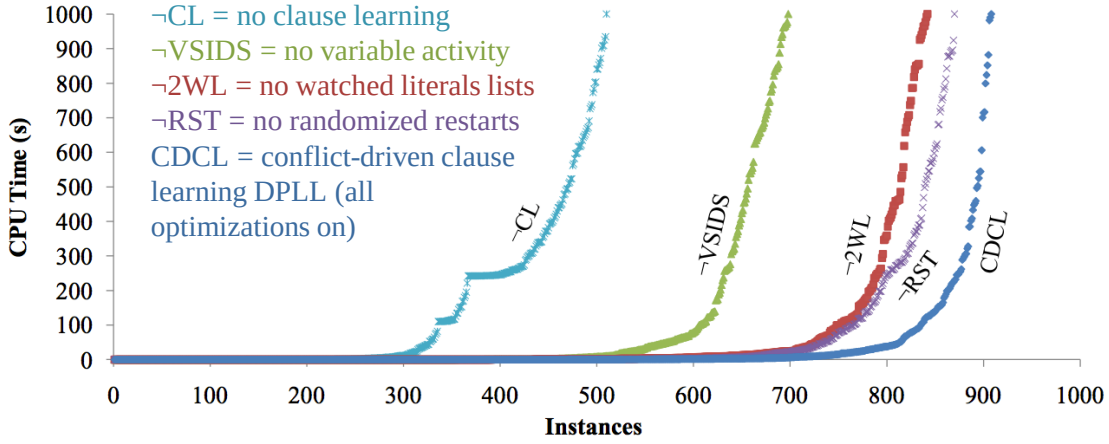


Figure 3.6: As key enhancements are removed from DPLL, fewer SAT problems are solvable within the same amount of time. This is Figure 3 reproduced from [70], and we have added a description of the labels.

3.4.1 Challenges of Hardware Support for Non-BCP Tasks

The remaining 10-20% of the CPU time is typically spent on conflict analysis, choosing the next variable to assign, and managing the learnt clauses database. These key enhancements to DPLL were discussed throughout section 2.5.2.

Figure 3.6 (reproduced from [70]) illustrates the penalty for removing key enhancements from DPLL. Empirically, removal of learnt clauses induces the largest handicap. Managing the database of learnt clauses is arguably the most challenging aspect to implement in an FPGA. In particular, this involves *memory allocation and deallocation*. One learnt clause is produced by conflict analysis upon encountering each conflict, however keeping every learnt clause will significantly slow down BCP.

Memory management in hardware is extremely cumbersome. Hardware typically gains its advantage over software by enforcing certain *memory alignments* so that the equivalent of multiple CPU instructions can be executed in a single FPGA clock cycle. In hardware, we would likely need to pay at least one of the following penalties:

1. Over allocate the block size to maintain alignment (the penalty is the wasted memory capacity and/or bandwidth).

2. Use a complex memory controller to tolerate misalignment (the penalty is that intricate logic produces longer compute paths, which will decrease the clock frequency and/or require more clock cycles to compute).

As shown in section 2.6.3, the number of variables in a clause can be regulated to assist alignment, however it can create an excessive number of dummy variables.

Given these inefficiencies, one could argue that spending e.g. 40% of the FPGA for only 10-20% of the compute time is a poor allocation of hardware resources.

3.4.2 Prior Works with Entire SAT Solvers in Hardware

An entire SAT solver in hardware is presented in [71]. Conflict analysis is performed yet learnt clauses were not created, likely due to the challenges of hardware memory management that we discussed above. Also, variables were assigned in a pre-defined order, e.g. variable 1 must be assigned before variable 2 can be assigned. Pipelining was added in [72] to improve the clock frequency. The hardware only supports clauses with up to 3 variables, thus many dummy variables would be created if the original SAT problem has large clauses. Due to the overhead of the non-BCP tasks, the hardware only supports very small problem sizes, which limits the practicality.

Although faster BCP may compensate for removing or simplifying the key DPLL enhancements from Figure 3.6, ideally it should not be at the expense of these enhancements. Acceleration was still achieved in [72] since features such as non-chronological backtracking, which enable one to jump back hundreds of decision levels, have less influence on smaller problem sizes.

3.4.3 Motivation for Heterogeneous Computing

Our design only accelerates BCP in hardware and we implement the remainder of the SAT solver in software. We provide our justification for this decision in this section.

The primary advantage of implementing an entire SAT solver in hardware is it minimizes the communication latency between the computation engines. The latency between the decision making engine (which variable to assign) and the BCP engine is not trivial. We address this latency with multithreading, as discussed in section 3.5.

Except for this communication latency, many factors actually favor software over hardware. The most important ones include:

CPU is hardwired: The CPU has a significantly faster clock, so operations like loads and stores in conflict analysis will inherently be faster in software. Furthermore, the last layer cache on CPUs is typically faster and larger than on-chip memory in FPGAs. CPUs also typically have faster and more DRAM memory controllers.

Memory allocation: Software has memory allocators that have been optimized over years. The CPU also has dedicated memory management units (MMUs) to decouple virtual addresses in software from the physical addresses. This enables further optimization such as balancing the bandwidth between several memory controllers.

Integration of new developments: SAT algorithms are still evolving with new approaches and better heuristics. As new developments are made, it is significantly easier (and faster) to integrate this in software.

Although modern SAT solvers may have very different heuristics, BCP is such a fundamental pruning mechanism that it is used by *every* modern DPLL-based SAT solver, many of which have highly optimized software implementations of BCP.

3.5 Multithreaded SAT

3.5.1 Multithreaded Software SAT

With the advent of multicore, for software to maximize the utilization of a CPU with n cores, we would like to run n threads of something. For example, to parallelize BCP, upon assigning variable v, we could visit up to n clauses that contain v in parallel. Unfortunately this type of "fine-grained" parallelism is not ideal for multithreaded software due to the large overhead of synchronization between threads.

This overhead further exacerbates the problem that BCP involves fundamentally serial computation, which is challenging to accelerate. BCP is inherently serial since if variable a implies b and then b implies c, we cannot go straight from a to c.

To amortize the large synchronization overhead, software favors "coarse-grained" multithreading in which threads execute large amounts of work before synchronizing. This has lead to two common strategies for parallelizing SAT:

1. Assign S variables in all possible ways, then distribute the 2^S subproblems to different threads.

2. Race many different SAT solvers against each other on the same SAT problem.

In the first case, the original problem is satisfiable if any subproblem is satisfiable. Proving unsatisfiable may benefit from dynamic load balancing so that we are not often waiting for one last thread to finish. In general, there is no reason to force partitioning to just the first S variables. As an example, we could divide a SAT problem as shown in Table 3.5. Thread 0 uses $a = 0$ by adding the clause $(\overline{a})$ to the original problem. As long as the entire space is covered, partitioning can be done as a

Table 3.5: An example partitioning of a SAT problem.

Thread	Subproblem
0	$a = 0$
1	$a = 1, c = d, b = 0$
2	$a = 1, c = d, b = 1$
3	$a = 1, c \neq d, e = 1$
4	$a = 1, c \neq d, e = 0, f = 0$
5	$a = 1, c \neq d, e = 0, f = 1$

prefix code. Equality-based partitioning can also be used. For example, adding the clauses $(c + \overline{d})(\overline{c} + d)$ creates the $c = d$ partitioning, likewise $(\overline{c} + \overline{d})(c + d)$ enforces $c \neq d$. Different subtrees do not need to branch on same variables. For example, threads 1 and 2 branch on b whereas threads 4 and 5 branch on f.

Subproblems can also be distributed over a grid of computers, as done in GridSAT [73]. We refer the reader to [74] for details on the heuristics for partitioning and the synchronization mechanisms, which are beyond the scope of this thesis.

The second approach above involves racing different SAT solvers against each other. The premise is that different SAT solvers perform better on different types of problems. One may have its heuristics optimized for cryptography-based problems whereas another may be optimized for equivalence checking. There are a wide range of applications for SAT (section 2.1.2), and consequently one SAT solver can almost never outperform another SAT solver *on every single benchmark* [33].

This diversity lead to so-called "portfolio solvers" such as SatZilla [75], which *choose* which SAT algorithm to run for a given SAT problem based on its characteristics. Common characteristics include: the ratio of variables to clauses, the distribution of clause sizes, the distribution of how many clauses each variable occurs in, the polarity of variables (e.g. does $\overline{x}$ occur more often than x), and so forth.

Given 4 CPU cores, one could choose and then race the 4 best algorithms against each other on the same SAT problem. On average, a solution is found faster than any individual SAT algorithm. ManySAT [76] uses this approach, however it always uses the same 4 algorithms (regardless of the given SAT problem). A common implementation facilitates the sharing of learnt clauses between their SAT algorithms. In general, SAT algorithms are written in various programming languages and with different data structures, thus it is not trivial to efficiently implement a multithreaded portfolio SAT solver. Other approaches for parallelizing SAT are discussed in [77].

Finally, portfolios may also include algorithms not based on DPLL. For example, we can include local search algorithms or run a preprocessor on a background thread and share the simplifications as they are found. Even hardware versions of these can be used, such as FPGA-accelerated WalkSAT [78] and FPGA-accelerated SatElite [79] (SatElite is the preprocessor of Minisat [37]). These non-DPLL hardware implementations are *complementary* to our work, hence we will not discuss them in detail.

3.5.2 Multithreaded Hardware SAT

Thus far, our examination of technology trends has motivated:

- Storing the SAT problem in the FPGA's SRAM (section 3.3), and
- Accelerating only BCP in hardware, so the remainder of the SAT solver is implemented in software (section 3.4).

In addition to fully utilizing the multicore CPU, one key advantage of multithreading is that it can *hide the communication latency between hardware and software.*

Figure 3.7 illustrates one round of BCP. These three steps are repeated within a SAT solver. Without multithreading, there are significant periods of inactivity for

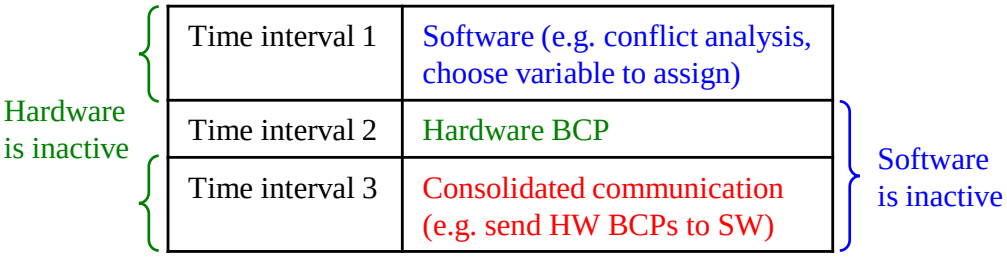


Figure 3.7: Without multithreading, there are idle times for both hardware and software. One round of BCP involves 3 steps (this cycle repeats within a SAT solver).

	Thread 0	Thread 1	Thread 2
Time interval 1	Software	Communication	Hardware BCP
Time interval 2	Hardware BCP	Software	Communication
Time interval 3	Communication	Hardware BCP	Software

Figure 3.8: By running three threads, we can fully utilize the CPU and the FPGA. It also hides the communication latency.

both software and hardware. Even disregarding the communication latency, if software BCP required 90% of the CPU time and hardware accelerated it by 9×, software would spend 10% of the original time on non-BCP tasks and 10% of the original time waiting on hardware. Likewise hardware would only be active half of the time.

Running additional threads keeps both software and hardware busy. Multithreading breaks the data dependencies which cause the serialization in Figure 3.7, thereby providing a higher probability that some work will be available. Although Figure 3.8 is idealistic, with 8 to 12 threads we can usually keep both the CPU and the FPGA reasonably active, as quantified by our experimental results throughout chapter 9.

Multithreaded hardware SAT has its challenges. BCP is a memory bound problem, as the data we read either enables us to apply BCP (when variable assignments are read) or tells us what to read next. This was discussed in detail in section 3.3.1.

The bandwidth of on-chip memory in FPGAs is typically 2 orders of magnitude larger than external bandwidth. Typically an FPGA contains SRAM (distributed as thousands of embedded memories) whereas DRAM is accessed through one or two external interfaces. This further strengthens our choice to use SRAM rather than DRAM. Without sufficient bandwidth, multithreading results in *time-shared* access to memory. Although it still hides the communication latency, it does not provide further *aggregate* acceleration over all threads. To put it in perspective, a BCP hardware accelerator that is only 2× faster than a single thread will most likely be slower than 4 threads of pure software (unless the CPU has extremely slow shared memory).

In hardware, we must replicate the variable assignments per thread in order to facilitate an independent search by each of the different software SAT solver threads. This results in a significantly more difficult challenge to support practical SAT problem sizes with the limited amount of on-chip SRAM.

To the best of our knowledge, we are the first to propose the use of multithreading for FPGA-accelerated SAT. We are reaching the threshold of technology where this is starting to become practical. In order to implement hardware SAT multithreading at this early stage, we will need very aggressive optimizations including the development of an *extremely compact representation of SAT in hardware*.

3.5.3 Most Relevant Prior Work

The FPGA design by Davis *et al.* [80] uses on-chip SRAM and accelerates only BCP in hardware (the remainder of the SAT solver is in software). Among all of the prior work, this is by far the most similar to ours in terms of technology and the partitioning of the SAT problem. Support for learnt clauses was later added in [81].

For a given SAT problem, the clauses are partitioned so that they can be distributed to several inference engines. This facilitates the parallel inspection of many clauses. The design operates as follows. For each variable v assigned:

1. The variable v is broadcast to every inference engine.

2. In parallel, each inference engine determines whether or not it has a clause containing v. Each engine only inspects the clauses within its own partition (using a trie). Upon finding it, we know which specific clause has v.

3. In parallel, for each inference engine that has a clause containing v, the assignment to v is updated, and all of the variable assignments of this clause are locally retrieved to check for BCP.

4. Any newly implied variable is reported through a pipelined multiplexer to a centralized controller, which will eventually broadcast it out.

The centralized controller also detects conflicts. For example, if inference engine 0 contains $(v + \overline{a})$ and engine 1 contains $(v + a)$, broadcasting $v = 0$ will result in engine 0 implying $a = 0$ at the same time as engine 1 implies $a = 1$.

As discussed in section 2.6.3, the number of clauses that a variable occurs in can be regulated. Given that hardware can fit e.g. 64 inference engines, a SAT problem can be preprocessed to ensure that each inference engine contains no more than n occurrences of a variable. Typically n is limited to 2 since FPGAs typically have dual-port embedded memories.

For each inference engine to determine which specific clause(s) contain the broadcasted variable (if any), a trie is used and an example is provided in [80]. Four sequential reads are required to perform this "translation".

The authors did not consider multithreading and their architecture does not enable an efficient integration of it. In fairness, hardware SAT multithreading would not have been practical back in 2008. Consequently, there is no mechanism for mitigating the communication latency between hardware and software, which accounts for 10-40% of the total hardware BCP time according to their results.

If a variable occurs in a only few clauses, most of the inference engines will be performing *wasted computation*. When a variable is broadcast, *every* engine is required to respond. Multithreading essentially multiplies the workload by the number of threads. Since the inference engines are already active much of the time, multithreading would result in time-shared use of the inference engines.

Without increasing the aggregate acceleration over all threads, integration with a multithreaded software host will result in hardware being the computation bottleneck. Such a system can actually be slower than pure software (recall the earlier example of 4 pure software threads versus a hardware accelerator which is only 2× faster than a single thread). For hardware to support the larger workload, entire inference engines in their design must be replicated, which is impractical.

Our design differs in two fundamental ways:

1. Do only the work that is required. Every compute task must be targeted. We cannot broadcast tasks, yet multithreading and the spreading of work through global links (section 4.4.2) balances the workload over several processors.

2. Eliminate translation in the innermost loop. The trie (or any kind of dictionary) consumes valuable on-chip memory and translation is slower than directly accessing what is needed. In our design, once a BCP is produced, we can immediately begin visiting clauses (as detailed in section 4.3.2).

3.6 Summary of Our Design Decisions

Throughout this chapter, we have motivated our design decisions based on the underlying physical characteristics of current technology.

In section 3.1, we motivated the use of FPGAs because:

- The increase of CPU performance is tapering due to power limitations.
- FPGAs have grown significantly in speed and size. They now also have hardwired high-performance logic, such as embedded memories and high-speed external interfaces. CAD tools for FPGA design have also improved.

We performed an experiment in section 3.3 to illustrate the poor performance of DRAM on random access. BCP is the dominant computation in SAT and mostly involves random access. We choose to use SRAM since it provides the highest memory density (to support practical SAT problem sizes) without crossing the threshold where sense amplifiers are needed (SRAM still has no penalty for random access).

In section 3.4, we chose to accelerate only BCP in hardware and implement the remainder of the SAT solver in software. This is beneficial because:

- CPUs are hardwired, which enables a faster clock and a larger cache.
- Memory (de)allocation is substantially simpler to manage in software.
- New developments to DPLL are easier to integrate.

Heterogeneous computing can benefit from the expanded use of different underlying technologies, however there is an inherent communication overhead. We address this with multithreading in section 3.5.

Chapter 4

Hardware Memory Layout and BCP Engine

In this chapter, we present many insights which lead to a very compact representation of SAT in hardware memory. This layout also facilitates fast BCP computation. The key parts of this contribution have been published [82]. Our memory layout ultimately defines the computation used to perform BCP in hardware. We conclude this chapter by discussing the timing of our hardware state machine.

4.1 Variable Occurrence Lists

In the most basic form of DPLL (section 2.5.1), we need memory to store:

1. The clauses of the SAT problem.
2. The assignments to the variables.
3. Variable occurrence lists (for each variable v, which clauses contain v).

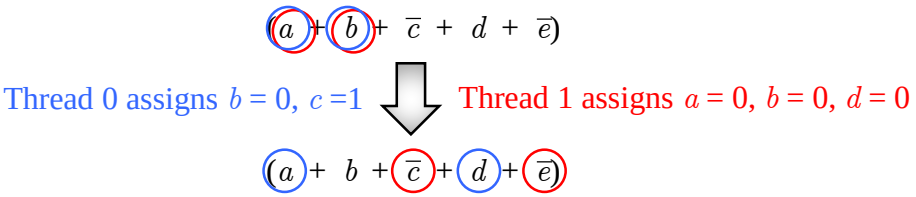


Figure 4.1: Watched literals lists must be replicated per thread because the watchers depend on which variables are currently assigned.

Although only the first two are absolutely necessary, without variable occurrence lists, we would need to sweep through all of the clauses to perform BCP. This is impractical, as modern SAT problems contain thousands to millions of clauses.

BCP is a fundamental pruning mechanism of DPLL, as illustrated in section 2.5.1. To apply BCP, for each variable v assigned, we could inspect every clause that contains v to check if any new variables can be implied. A variable occurrence list for v would indicate which specific clauses we need to visit. Every variable has its own occurrence list. Watched literals lists allows one to prune these lists, however it requires significantly more memory for multithreaded SAT.

In multithreaded SAT (section 3.5), each thread acts as an individual SAT solver. To benefit from partitioning or racing (section 3.5.1), the concurrent searches must be independent, so we must replicate the variable assignments (one copy per thread).

Recall from section 2.5.2.3 that the watched literals lists approach requires both "watchers" to be currently unassigned variables. Because of this, we only need to visit a clause if one of the watchers gets assigned (this prunes the variable occurrence lists). Upon visiting the clause, we try to find a new watcher. If there are no watchers available, then we have a BCP. As illustrated in Figure 4.1, due to the dynamic updating of watchers, *watched literals lists would need to be replicated per thread.*

Data must be static or "read-only" to be sharable over all threads. Obviously this is the case for the clauses of the SAT problem. This is also the case for full variable occurrence lists (the list for variable v includes every clause that contains v).

In order to support practical SAT problem sizes with the limited amount of on-chip SRAM, we choose to use full variable occurrence lists over watched literals lists. Although variables assignments must be replicated, these are relatively small (notice that only 2 bits are needed to represent whether a variable is currently true, false, or unassigned). Conversely, each item in a variable occurrence list or a watched literals list is a clause index, which is substantially larger than a variable assignment (e.g. 16 bits are needed to support problems with up to 65536 clauses).

Using full variable occurrence lists has other advantages:

- It allows one to operate on non-Boolean constraints (enhanced BCP in section 6.1 uses lossless compression to augment the problem sizes we can support).
- Memory allocation and deallocation are not needed in hardware. When we find a new watcher, the watched literals list of one variable will remove one clause index and the watched literals list of another variable will add one clause index. The challenges of managing memory in hardware were discussed in section 3.4.1.
- Read-only information is easier to distribute over multiple processing engines because it avoids the challenges of concurrent updating. This facilitates better workload balancing and possibly more parallel processing.

The primary disadvantage of full variable occurrence lists is that we must visit all clauses. However, there are techniques for mitigating this, such as enhanced BCP (section 6.1) and workload splitting with equivalent variables (section 4.4.2).

4.2 Key Insights for Compacting SAT

In this section, we provide several key insights which jointly minimize the amount of memory needed to represent SAT in hardware memory as well as facilitate fast BCP computation. Essentially we attempt to derive one of the simplest intrinsic representations of SAT.

4.2.1 Implicit Representations

Our first design guideline is primary focused on minimizing the amount of memory needed. This enables better support for practical SAT problem sizes.

Key Insight 1. *Favor implicit linking over explicit linking whenever possible.*

For example, it is better to sequentially access items in an array (the addresses are *implicitly* adjacent) rather than to traverse a link list (which *explicitly* indicates where in memory the next item is). This has two key advantages:

- We save space by not storing pointers/addresses.
- Several implicit accesses are faster than explicit accesses. Implicit access depends on memory bandwidth whereas explicit access depends on read latency.

To illustrate the second point, sequential access to an array can be pipelined, e.g. a continuous access to the memory of one value per clock cycle. Conversely, traversing a link list typically requires 2 or 3 clocks per access, depending on how much logic is in the feedback path from the memory's read data to its address. Alternatively, without any pipeline registers, achieving one value per clock cycle on a link list traversal will result in a significantly slower clock frequency than pipelined array access.

Table 4.1: Description of the two candidate memory layouts.

	Implicit Feature	Explicit Feature
Variable-Centric Memory Layout	Each variable has its own section of contiguous memory, hence the variable occurrence list (which indicates which clauses we need to visit) is an array	Each clause is described as a cyclic link list, therefore inspecting each clause involves a link list traversal
Clause-Centric Memory Layout	Each clause has its own contiguous memory section, and a pre-defined layout for clauses simplifies the inspection of each	Each variable occurrence list forms a cyclic link list, thus a link list traversal is needed to find all of the clauses

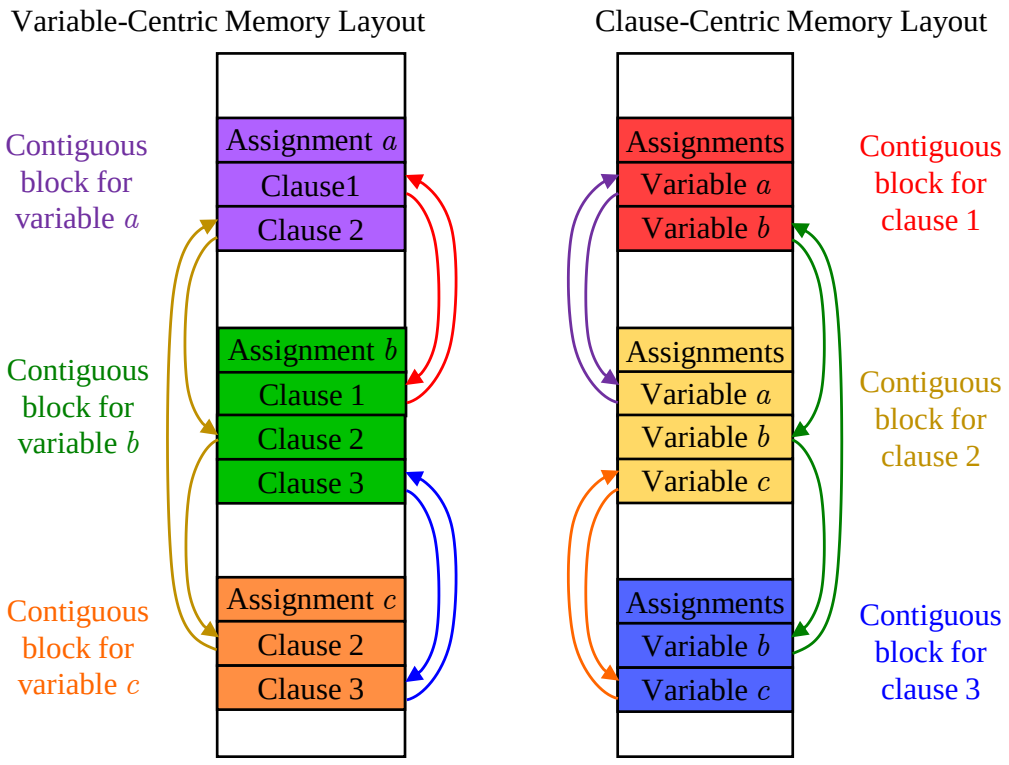


Figure 4.2: Two memory layouts for the same SAT problem of $(a+b)(a+b+c)(b+c)$. The first clause $(a+b)$, which is colored red, occupies a contiguous memory section in the clause-centric layout whereas that same clause $(a+b)$ is represented as a cyclic link list in the variable-centric layout. The color purple represents the variable occurrence list for a, which indicates which clauses contain variable a. Likewise, it has a different representation in each memory layout.

Table 4.2: Comparison of the two memory layouts. Properties are compared in pairs.

Property	Variable-Centric	Clause-Centric
Inspecting one clause	Slow	Fast
Finding all clauses for a given variable	Fast	Slow
How many link list traversals per variable	Number of clauses that this variable is in	1
Length of each link list traversal	Number of variables in current clause	Number of clauses that this variable is in
Replication of variable assignments	One copy per thread	One copy per thread *and per clause*
Each implicit operation	Access link to next variable and *one* variable assignment	Access link to next clause and *all* variable assignments

This design guideline leads to two candidate memory layouts: the variable-centric layout and the clause-centric layout, as explained in Table 4.1 and illustrated in Figure 4.2. If we decide to implicitly represent the variables, then we must explicitly link the clauses, or vice versa. We cannot implicitly represent both. For a SAT problem with *arbitrary* clauses, there must be at least one degree of freedom.

In the variable-centric layout, which specific clauses to visit is immediately available, however actually inspecting each clause requires a link list traversal. Cyclic link lists enable inspection of a clause starting from any of its variables. Conversely, in the clause-centric layout, a link list traversal is needed to visit each clause, yet the inspection of each clause can be implicit (e.g. our pre-defined layout always has the variable assignments at the top). Likewise, a cyclic link lists allows us to start anywhere, and the traversal finishes upon re-visiting a clause.

A comparison of the two memory layouts is summarized in Table 4.2. We need more memory for the clause-centric layout, although we can do more work with each implicit operation. So far, it is not immediately obvious which layout is better.

4.2.2 Fast BCP

The next two design guidelines are primary focused on facilitating fast BCP in hardware. Ideally, compaction should not impede computation, and vice versa.

Key Insight 2. *When a variable is assigned, we are most interested in the variable assignments of the clauses that this variable occurs in, not the clauses themselves.*

Recall from section 3.3.1 that in order to perform BCP, there are typically three levels of dereferencing. For each variable v assigned, first we need to know which clauses to visit (from the variable occurrence lists), then we need to know which variables those clauses contain, and finally we read the assignments of those variables to apply BCP. Ideally, we should first access what matters the most, e.g. when assigning v, go straight to the corresponding variable assignments. Only if there was a BCP, we can then go back to determine which variable and clause it was. As a lower bound, at least one full list must be accessed because a SAT problem can have an arbitrary number of clauses containing v.

Key Insight 3. *To maximize the speed of constraint propagation, all of the variable assignments in a clause should be read simultaneously.*

Once we know where the variable assignments are for a given clause, we should minimize the amount of time needed to determine if there is a new variable to imply. This is minimized by reading all variable assignments at once (on the same clock cycle). This also enables constraint propagation to become *purely combinational logic*, which is extremely fast.

These new insights heavily favor the clause-centric memory layout. The variable-centric layout slowly gathers variable assignments by traversing a link list.

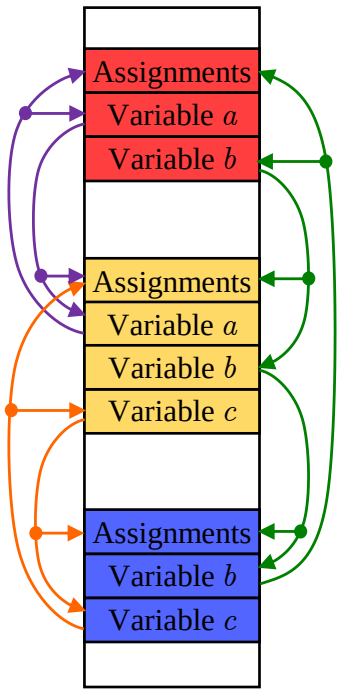


Figure 4.3: Two pointers are placed in each memory location used by the link list. This way we can simultaneously traverse the link list (visiting different clauses) as well as access all variable assignments of the current clause.

Boolean variables can only be assigned true or false, and the tree search of DPLL also adds the unassigned state, hence only 2 bits are needed per variable assignment. Embedded memories in FPGAs can typically be configured as 18 bits per location [59,60] for example, so we could pack up to 9 variable assignments in the same memory location. As shown in section 2.6.3, the maximum number of variables in any clause can be regulated by splitting large clauses and introducing dummy variables.

Figure 4.3 shows that the clause-centric layout uses two pointers (which are packed into one memory location) to indicate:

- Where is the next clause in the link list, and
- Where are the variables assignments for this next clause.

Figure 4.2 was a simplification, as its primary objective was to illustrate two different ways to exploit implicit representations. With only one pointer, we can only traverse the link list (we would not know where the variable assignments are). By a similar argument, the variable-centric memory layout would have also needed two pointers.

One challenge unique to the variable-centric memory layout is the inability to easily add learnt clauses (produced by conflict analysis, see section 2.5.2.1). To maximize compaction, no space can be wasted between the contiguous memory sections of each variable. For example, on the left side of Figure 4.2, the last clause of variable a should be immediately followed by the assignment for variable b. However, this makes it impossible to add any new clause containing a unless everything underneath it is shifted down. Conversely, in the clause-centric memory layout, for each variable v in the added clause, we simply perform one insertion to the cyclic link list of v.

The primary disadvantage for packing together variable assignments is that we must replicate them, one copy per clause. However, this is relatively inexpensive because each assignment is only 2 bits. This is further elaborated below.

Notice that in both memory layouts, the total number of links needed is equal to the total number of variable occurrences in all clauses. Intuitively, *each* occurrence of some variable needs to be stored somewhere. From the example in Figure 4.2, variables a, b, and c occur in 2, 3, and 2 clauses respectively, and a total of 7 links are used in each layout. If we add a new clause with 3 variables, we must add space in memory for 3 new links. Each link requires tens of bits of memory, so adding a total of 6 extra bits for variable assignments is relatively inexpensive.

In conclusion, the variable-centric layout demonstrates that certain compactions can hinder computation, and therefore *we have chosen the clause-centric layout.*

4.2.3 Lossless Compression and Partitioning

Our final design guideline not only enables memory compaction, it also examines ways to efficiently use the thousands of embedded memories that modern FPGAs contain.

Key Insight 4. *When explicit linking is required, favor localism so that addresses (or pointers) can be represented with fewer bits.*

This is actually a form of *lossless compression*. For example, recall from Figure 4.3 that two pointers are needed for each clause that we visit: one pointer indicates the location of all variable assignments packed together, and the other pointer is for the link list. If we only support clauses with up to 9 variables, then the *difference* between these two pointers is at most 9. Storing one of the pointers as the difference requires fewer bits, hence reducing the amount of memory needed.

For the non-differential pointer, the number of bits needed depends on the amount of memory in the system. For example, in an FPGA with 36 Mbits of on-chip SRAM organized as 18 bits per location, this gives 2^{21} locations, so the address needs 21 bits. Unlike software, we are not constrained to pointers on 32 or 64 bits.

We illustrate Key Insight 4 with an example. Suppose we are traversing the link list to visit each clause that contains some variable v. To arrive at the current clause, we obviously know its address. Suppose the location of the next clause in the link list is within the same block of X addresses as our current address (e.g. $X = 4096$). In this case, an address on $\lceil log_2(X) \rceil$ bits (e.g. 12 bits) is sufficient to specify the location of the next clause (the upper address bits remain the same). We also need a 1-bit flag to indicate whether the next address is "local" or "global".

We could introduce more than 2 pointer sizes, however the key practical implication of this technique is whether a pointer will fit in 1 or 2 memory locations. Modern

FPGAs use hardwired logic for the embedded memories to improve the maximum clock frequency and the density of SRAM. Consequently, the width of the memory is *quantized*, typically 9, 18, or 36 bits per location.

A narrow memory (e.g. 9 bits per location) provides insufficient bandwidth and we would need several clock cycles to read all of the information. A wide memory (e.g. 36 bits per location) cannot easily exploit small or localized pointers. We chose 18 bits per location to balance these two factors.

In addition to the local pointer, we would like to pack other information in the same memory location, such as the differential pointer (as discussed above) which would need 4 bits for clauses with up to 9 variables. *Given* a memory with 18 bits per location and *knowing* how many bits the other information takes, the *remaining space determines* the maximum number of bits that the local pointer can use. This ultimately specifies the block size X (which was 4096 in the above example).

One immediate consequence of exploiting lossless compression for pointers is that it inherently *partitions* the clauses of the SAT problem. We present an algorithm for partitioning SAT in section 8.1. Partitioning is beneficial because:

- Embedded memories are physically spread across the entire FPGA. Using all of them as one giant logical memory would result in very long routing paths and an extremely slow clock frequency.

- Partitioning induces a natural division of work which facilitates massively parallel processing, thereby improving the overall computational throughput.

To distribute the processing, we have the option to distribute the tasks or to distribute the data. Let us examine the implications of each:

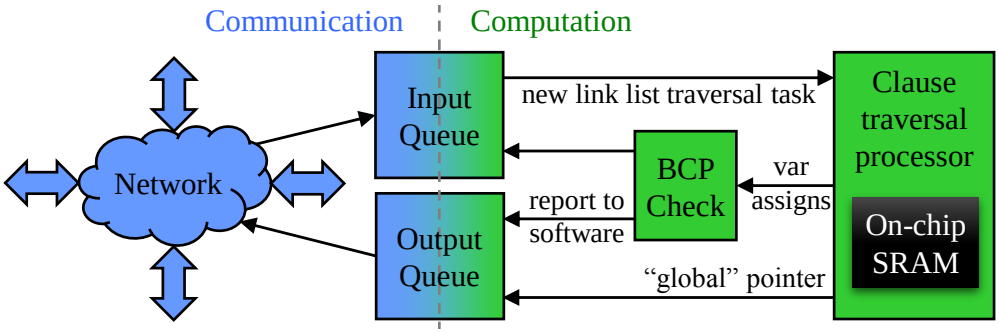


Figure 4.4: Separating communication from computation enables a seamless transfer of link list traversal tasks between processors. Each newly implied variable creates a task and is reported to software. Queues decouple the network and processor timing.

Distributed tasks: For hardware BCP, a "task" is a link list traversal to visit clauses. This allows each partition of clauses to stay within its own processor. To inspect a clause C, the task must be given to the processor that contains C. Every use of a "global" pointer transfers the link list traversal to a different processor.

Distributed data: Each link list traversal stays within its own processor. If we need clause C but some other processor has it, that other processor must send us C.

SAT is a memory bound problem (as examined in section 3.3.1) so to maximize acceleration, we must *operate as close to the memory interface as possible*. Moving clauses between processors hinders this, therefore we distribute tasks. Without moving clauses, each processor essentially has "ownership" of one partition of clauses, which facilitates a tighter integration between data access and processing of that data.

In order to facilitate the seamless transfer of link list traversal tasks between processors, we can *separate communication from computation*. Figure 4.4 illustrates a processing model in which the clause traversal processor can disregard how a task arrived and the network can ignore the specifics of each task. The details of the hardware system integration will be examined throughout chapter 5.

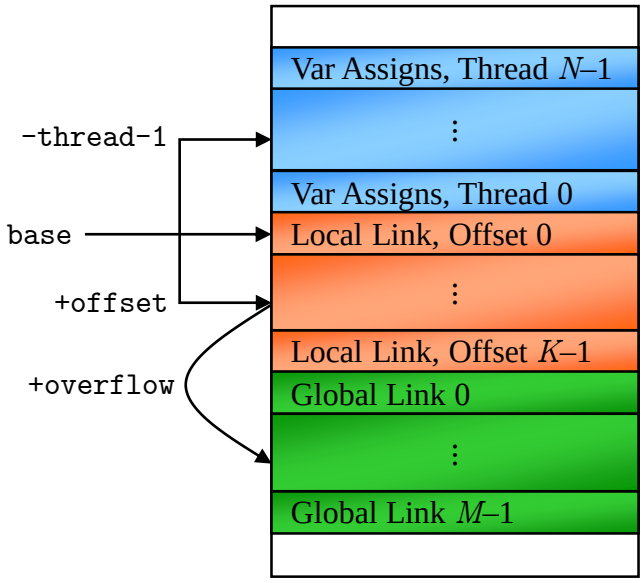


Figure 4.5: The detailed memory layout for one clause.

4.3 Hardware BCP Memory Layout

Throughout section 4.2, we presented several key insights to guide the high-level structure of the memory layout for hardware BCP. We now present the details of our compact representation of SAT that facilitates fast BCP.

4.3.1 Specification to Visit One Clause

Figure 4.5 illustrates the layout for one clause. Each clause occupies a contiguous section of memory, and there is no wasted space between clauses. For each variable v, all occurrences of v are connected with a cyclic link list. Upon assigning v, we visit each clause that contains v by traversing this link list in order to check for BCP. Because the link list is cyclic, we can start from any clause containing v, and the traversal finishes upon re-visiting a clause.

In order to visit one clause (one step in the link list traversal), the following information is required:

- `thread` indicates which software thread our hardware is performing BCP for (multithreading was discussed in section 3.5).
- `base` indicates where this clause is located in hardware memory. Local link offset 0 is used as an arbitrary anchor point.
- `offset` indicates which variable within the clause we are interested in. For example, if we are visiting clauses containing variable d and the current clause is $(a + \overline{b} + c + d + \overline{e})$, then `offset` is 3 ($d$ is the fourth variable and offset starts at 0).
- `assignment` is a 2-bit value indicating whether we are assigning true, assigning false, or deassigning the variable.

Each software thread races its own SAT solver against other threads on the same SAT problem, so clauses can be shared between threads. Variable assignments in hardware are replicated per thread to enable independent searches, hence `thread` is used to select which set of hardware variable assignments to use. To determine the memory location of the variables assignments for this `thread`, we compute:

$$\texttt{var\_assign\_location} = \texttt{base} - \texttt{thread} - 1.$$

For one thread, all variable assignments for the clause are packed together in one memory location. For N threads, we need N memory locations, as shown by the blue section of Figure 4.5.

Parameter K in Figure 4.5 indicates the number of variables in the clause, as each variable requires exactly one local link. It follows that

$$\texttt{offset} < K$$

which is enforced *by construction* during the generation of the memory layout. In other words, we can never be trying to access the fifth variable ($\texttt{offset} = 4$) in the clause $(\overline{a} + b + \overline{c})$ which has $K = 3$, for example.

In addition to specifying the local link, `offset` also indicates which specific variable assignment we need to update. While `thread` selected the memory location, `offset` indicates which 2 bits *within* that memory location must be updated. For example, if `offset` is 3, then we will update bits 7 and 6 with the value of `assignment`. In general, we can store the variable assignment of `offset` k in bits $2k + 1$ and $2k$.

In order to visit the next clause that contains variable v (taking the next step in the link list traversal), the following updates are applied:

- `thread` implicitly stays the same.
- `base` of the next clause must be explicitly stored.
- `offset` of the next clause must be explicitly stored.
- `assignment` is incrementally updated, as shown in section 4.3.3.

We store `offset` in full because it is relatively small (e.g. it is only 3 bits for clauses with up to 8 variables). This is actually the differential pointer that we discussed at the beginning of section 4.2.3. Using lossless compression, the next `base` can be encoded in two ways:

- If the next clause is located within the same block of X addresses (e.g. $X = 4096$), a "local" pointer on $\lceil log_2(X) \rceil$ bits (e.g. 12 bits) is sufficient to specify the next `base`, as the upper address bits will remain the same.
- Otherwise, a "global" pointer is needed.

Each local link needs a 1-bit flag to indicate whether the next `base` is local or global. A local pointer is small enough to fit in one memory location, however a global pointer must be stored across two memory locations. Some bits of the global pointer are stored in the local link (the orange section of Figure 4.5) and the remaining bits are stored in the global link (the green section). The specifier `overflow` in Figure 4.5 indicates where the global link is located relative to the local link.

An explicit `overflow` is required. Let us illustrate with an example. Suppose our clause is $(a + b + c)$.

- Variable a uses local link `offset` 0. Suppose the next clause containing a occurs in a different partition, then we can assign global link 0 to a. Thus, `overflow` is 3 since this clause has 3 variables.
- Variable b uses local link `offset` 1. Suppose the next clause containing b occurs in the same partition, hence no global link is needed.
- Variable c uses local link `offset` 2. Suppose the next clause containing c occurs in a different partition, then we can assign global link 1 to c. Notice that this `overflow` is 2 since b did not use a global link.

Parameter M in Figure 4.5 represents the number of global links this clause uses ($M = 2$ in the above example). In general, $K \geq M \geq 0$. In other words, we cannot

use more global links than the number of variables in the clause. Note that $M = 0$ is possible, but $K \geq 1$.

We do not store the number of variables in the clause, hence rearranging the variables so that the ones that need global links go first does not make `overflow` implicit. Hardware still functions correctly without knowing K because:

1. `offset` $< K$ is enforced by construction, as discussed above.

2. Applying BCP to $(\overline{a} + b)$ is the same as applying BCP to $(\overline{a} + b + 0 + \ldots + 0)$.

Since the constraint propagation logic has to support clauses with up to e.g. 8 variables anyways, clauses with fewer variables can reuse this by simply tying off all remaining variable assignments to false.

4.3.2 Variable Assignments and Constraint Propagation

The previous section focused on how we traverse the link list. As each clause is visited, we now examine the updating of variable assignments and checking of BCP.

All variable assignments for a clause are packed within one memory location. While `thread` indicates which memory location to use, `offset` indicates which specific 2 bits *within* this location to update with the value of `assignment`. The 2-bit value of `assignment` uses the following binary encoding:

- `00` = deassign.
- `01` = invalid (by construction this is never used).
- `10` = assign false.
- `11` = assign true.

The variable assignment of `offset` k is stored in bits $2k+1$ and $2k$. For example, in the clause $(a+b+c+d+e)$, if we assign variable b as false, then bits 3 and 2 becomes 1 and 0 respectively. If we deassign variable d, then bits 7 and 6 both become 0 (deassignment happens after the conflict analysis part of the SAT solver).

We know we have finished traversing the cyclic link list when the *incoming* `assignment` matches the *existing* `assignment`. For example, suppose $(a+b+c+d+e)$ is the first clause we visit containing variable a. We assign $a=1$, so bits 1 and 0 both become 1. We go along visiting other clauses that have a, and eventually we come back to $(a+b+c+d+e)$. When we try to assign $a=1$ again, we will notice that bits 1 and 0 are already both 1, hence we have been here before. On the first visit, a would have been in a deassigned state.

Likewise, conflicts can be detected if the incoming `assignment` is true yet the existing `assignment` is false, or vice versa.

Once we have all of the updated variable assignments for the clause, we send them to a BCP Check engine, as illustrated in Figure 4.4. Given all variable assignments on the same clock cycle, constraint propagation can be *purely combinational logic* which is extremely fast (check the clause is not satisfied and one variable is unassigned).

For example, suppose we are visiting clauses with variable a to update $a=1$. The current clause is $(\overline{a}+b+\overline{c}+\overline{d})$ and we already assigned $b=0$ and $c=1$ earlier. Sending $a=1$, $b=0$, $c=1$, and $d=$ unassigned to the BCP Check engine results in $d=0$. We report this to software and also enqueue a new link list traversal task to later visit the clauses containing d.

We start traversing the link list at the same clause where the variable was implied. Having visited $(\overline{a}+b+\overline{c}+\overline{d})$, we know its `base` and `thread`, and these are reused for

starting d. The constraint propagation engine provides the `offset` (3 in this example) and the `assignment` (10 in binary or false in this example).

The constraint propagation engine and clause traversal engine use *compatible encodings*, thus avoiding the translation tables that [80] uses in the innermost loop of BCP (section 3.5.3). However, we still need a translation between software variables and hardware addresses and offsets. Even so, any latency incurred by this lookup table can be consolidated with the communication latency between hardware and software, which we already mitigate with multithreading, as discussed in section 3.5.2.

4.3.3 Incremental Update to Assignment

It is possible to simplify the BCP Check engine so that it only operates on positive variables. In other words, the clause $(\overline{a} + c)$ must somehow remap variable a to avoid negation. As illustrated in Figure 4.6, each variable in each clause can use a 1-bit flag to indicate whether the next occurrence of that variable has the same sign. In the example in Figure 4.6, as we traverse the link list for variable a, the first and third clauses have the "next is negated" flag as true, so we see a change in `assignment` by the time we reach the second and fourth clauses, respectively.

Essentially we are using differential coding to store whether each variable occurs as positive or negative within each clause. By pushing the complexity of sign management into the link list traversal, the constraint propagation engine becomes simpler because clauses now implicitly have only positive variables.

Although the value of `assignment` is 2 bits, we only need 1 bit to incrementally update it. Notice that if variable v occurs in 5 clauses, each thread has 5 copies of the variable assignment to v and these are distributed (one at each clause). Consequently,

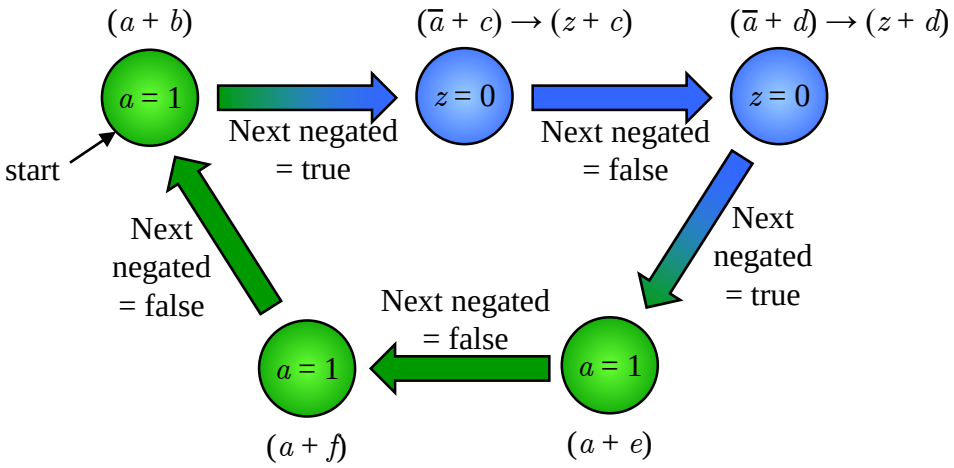


Figure 4.6: By dynamically updating `assignment` as we visit each clause, the original clauses can be modified to have only positive variables. We introduced a dummy variable z to represent the negated version of a. The first, fourth, and fifth clauses see `assignment` = true whereas the second and third clauses see `assignment` = false.

after each conflict in the SAT solver, deassignment must also traverse the link lists to update all occurrences of each variable assignment. Furthermore, we cannot start the new round of BCP until all deassignments are complete. For example, suppose $a = 0$ is the next variable that software will assign after a conflict. If this happens before we deassign $b = 0$, then the clause $(a + b + c)$ will incorrectly imply $c = 1$.

Each thread in hardware operates in two distinct modes: assignment mode and deassignment mode. Operations from one mode must completely finish before we can change modes. Thus, if we are deassigning at the current clause, we must deassign at the next clause. If we are assigning (either true or false) at the current clause, we may assign true or assign false at the next clause. Knowing the current value of `assignment`, the next value is limited to at most two choices, therefore 1 bit is sufficient for an incremental update.

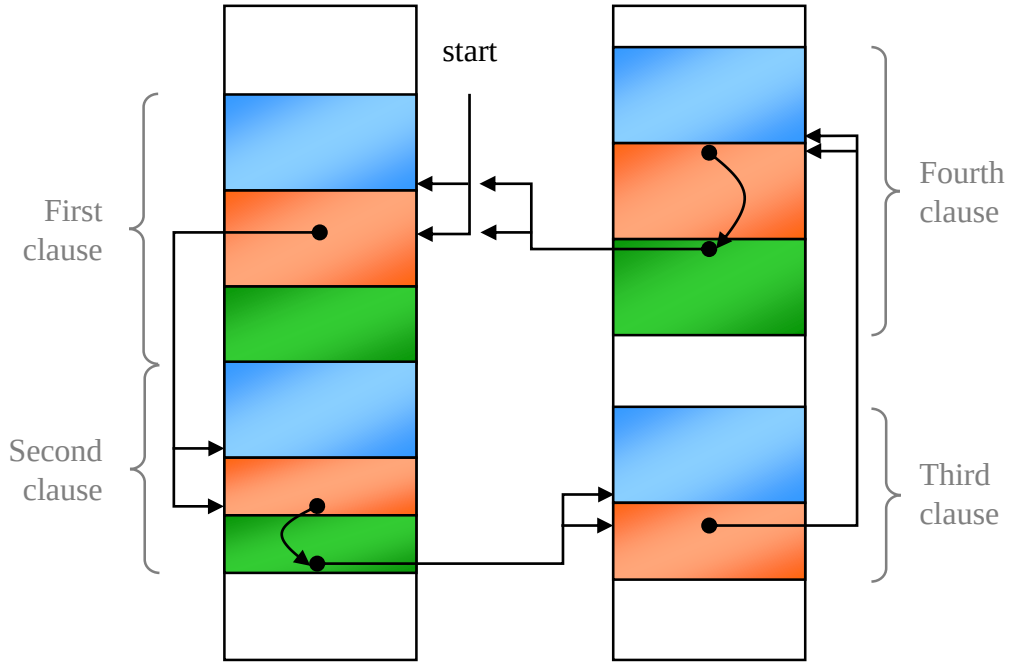


Figure 4.7: An example traversal of four clauses. The same color scheme is used as in Figure 4.5. Blue is for variable assignments, orange is for local links, and green is for global links.

4.4 Concurrency Analysis

4.4.1 Synchronization of Distributed Variable Assignments

A variable v that occurs in C clauses will have C copies of its variable assignment per thread. These are *distributed* to the location of those clauses. As we will show, these copies inherently stay synchronized if no conflict occurs.

Figure 4.7 illustrates the access pattern that results from assigning a variable that occurs in four clauses, which are spread over two partitions. At each clause, we access both the variable assignments (the blue section) and the link to the next clause. This always involves a local link (the orange section), and a global link is also needed (the green section) if the next clause is in a different partition.

The height of each section corresponds to the number of memory locations used in that section. The blue sections always have the same height since the number of threads does not change. The `thread` also stays the same, notice we always access the same positions in the blue section (the bottom corresponds to the `thread` = 0).

The first clause has more variables than the second clause, as indicated by the taller height of the orange section in the first clause. The `offset` can also change between clauses. For example, we access the last variable in the second clause and the first variable in the fourth clause.

The number of global links is at most equal to the number of local links (as in the fourth clause, where all variables have their next occurrence in a different partition). Clauses may not need any global links, as demonstrated by the third clause in which all variables have their next occurrence within the same partition.

A link list traversal task can be created by a BCP in hardware or a variable assignment from software (to continue the tree search of DPLL). Before the link list traversal, all copies of the variable assignment should be in the unassigned state.

Without a conflict, eventually we will visit all clauses and update all copies of the variable assignment. The correctness of this is obvious if the variable is implied (or assigned) exactly once.

Let us examine the scenario where the *same* variable is implied *multiple* times. For example, assigning $a = 0$ in:

$$(a + \overline{b})(a + \overline{c})(b + d)(c + d)$$

will imply $b = 0$ and $c = 0$. Now there is a race to determine whether b or c will imply $d = 1$, and any of the following situations may occur:

1. $b = 0$ implies $d = 1$ first. This causes us to visit clauses containing d, and we arrive at $(c + d)$ before the link list traversal for c arrives here. Thus $(c + d)$ will already be satisfied by the time c arrives later, so it cannot re-imply $d = 1$.

2. By symmetry, $c = 0$ could imply $d = 1$ first, and the above analysis applies.

3. $(b + d)$ and $(c + d)$ could *both* imply $d = 1$ if they do so before the link list traversal for d arrives.

The third case is more likely to happen when $(b + d)$ and $(c + d)$ occur in different partitions, as $d = 1$ would be implied by two different processors that operate concurrently. The third case actually causes two separate link list traversals for the same variable d, one starting at $(b + d)$ and the other traversal starting at $(c + d)$.

When the traversal that started at $(b + d)$ eventually reaches $(c + d)$, the incoming `assignment` will match the existing `assignment`. Note that the existing `assignment` was actually written by the other traversal that started at $(c + d)$. Nonetheless, the traversal that started at $(b + d)$ will think that it is finished. Likewise, the same thing will happen when the traversal that started at $(c + d)$ eventually reaches $(b + d)$. Essentially each traversal covered half of the clauses, and together both traversals visited every clause.

In general, the cyclic link list traversal can be split into several sections, and these can run concurrently. If each section of the traversal continues until it visits an already visited clause (as detected by the incoming `assignment` matching the existing `assignment`), then all clauses will be visited by the time every traversal has finished. This is illustrated in Figure 4.8.

As mentioned in section 4.3.2, conflicts are detected if the incoming `assignment` is true and the existing `assignment` is false, or vice versa. This is not possible with only

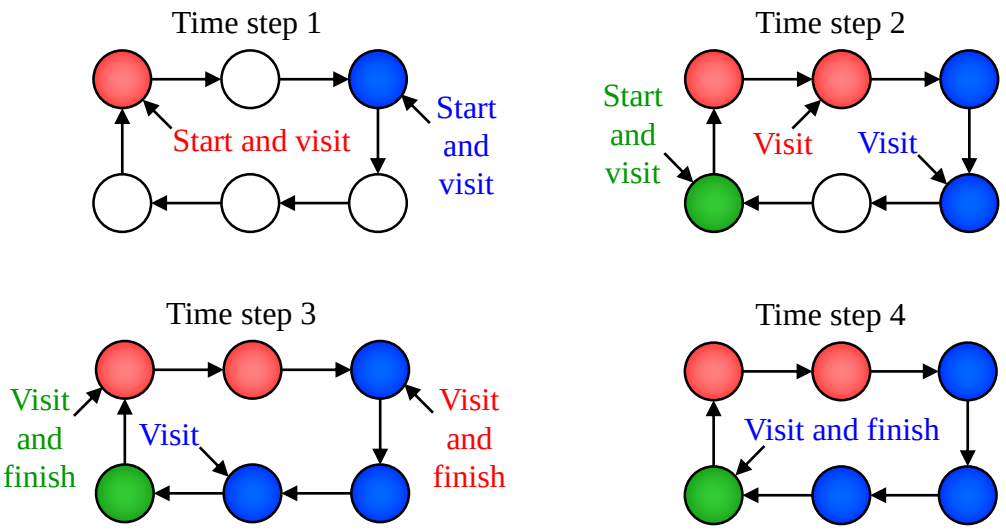


Figure 4.8: Concurrent traversal of a cyclic link list. We can start at different points and at different times. A node is colored once it is visited. Each component of the traversal finishes when it visits an already visited node. When all traversals finish, every node must be visited.

a single traversal, as that would end in a consistent state. In other words, *conflicts require a race between at least two concurrent traversals of the same cyclic link list.*

For example, assigning $a = 0$ in:

$$(a + b)(a + c)(\overline{b} + \overline{c})$$

will induce a conflict. We have a race, and one of the following situations must occur (we use $\rightarrow$ as a short form for "implies"):

- $a = 0 \rightarrow b = 1 \rightarrow c = 0 \rightarrow a = 1$ (race on the cyclic link list of a).
- $a = 0 \rightarrow b = 1 \rightarrow c = 0$ and $a = 0 \rightarrow c = 1$ (race on cyclic link list of c).
- $a = 0 \rightarrow b = 1$ and $a = 0 \rightarrow c = 1 \rightarrow b = 0$ (race on cyclic link list of b).
- $a = 0 \rightarrow c = 1 \rightarrow b = 0 \rightarrow a = 1$ (race on cyclic link list of a).

Notice that there is a *cyclic dependency* between variables a, b, and c. The exact manifestation of the conflict depends on how fast we go around each direction of the dependency loop (e.g. the order in which variables are processed).

In conclusion, without a conflict, all copies of a variable assignment must inherently stay synchronized. If there is a conflict, at least one variable must have a disagreement between the copies of its assignment. In terms of DPLL, a conflict means the tree search made a bad decision and therefore needs to backtrack. It follows that any inconsistent variable assignment will be immediately deassigned (after conflict analysis), hence restoring it a consistent unassigned state.

4.4.2 Distribution of Work Between Processors

BCP can be regarded as a form of *dynamic task creation*. Upon *discovering* that a variable should be implied (to prune the tree search of DPLL), we enqueue a link list traversal task (which will be executed some time later). Although each newly implied variable starts its link list traversal in the same processor that implied it, global links facilitate a spreading of the work between processors.

An example of work spreading is illustrated in Figure 4.9. After $a = 0$ implies $b = 0$ using $(a + \overline{b})$, we can visit clauses containing a at the same time as we visit clauses containing b because different partitions of the SAT problem reside in different processors. In general, the first variable a could have implied several other variables, and the implied variables like b could imply others as well. Applied recursively, numerous variables could be implied and several link list traversal tasks could be simultaneously executed. Depending on the amount of memory and logic resources in a given FPGA, several tens of processors can operate concurrently, which jointly

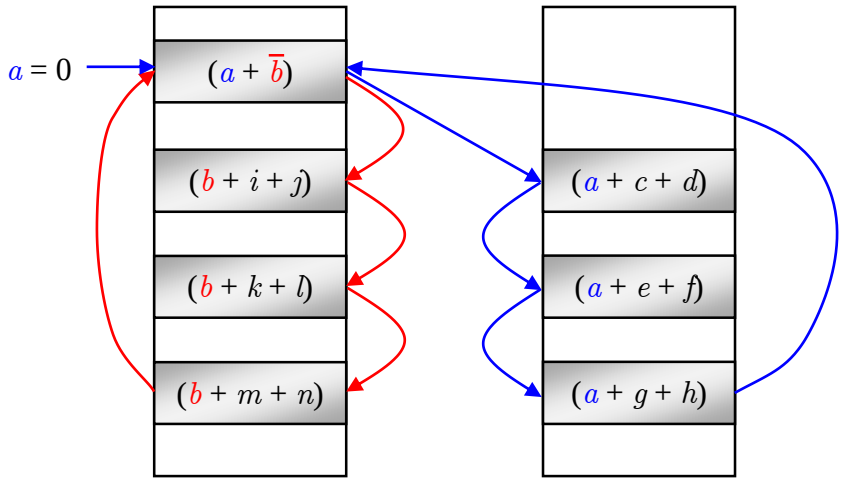


Figure 4.9: When $a = 0$ visits $(a + \bar{b})$, it implies $b = 0$. The link list traversal for a is then passed to another processor, hence the clauses containing a and the clauses containing b can be visited concurrently.

support a massive amount of work. Even if a particular SAT problem does not create many BCPs, multithreading also helps to spread the work across processors.

The maximum number of clauses that a variable occurs in can be regulated, as shown in section 2.6.3. This technique can be used to limit the latency of a long traversal from a variable that occurs in 1000 clauses, for example. Several equivalent variables can be created so that, e.g. we do 10 traversals of length 100.

Finally, it appears that excessive compaction (with few global links) can hinder the aggregate computing throughput over all processors due to limited work sharing. Nonetheless, too many global links may impede computation because there is a nonzero latency to transfer a link list traversal to a different processor. It remains an open problem to find the optimal proportion of global links for a given SAT problem. From a practical perspective, our brief experimentation suggests that it may be better to use more global links, so long as the SAT problem still fits in memory. This is further discussed in section 8.1.5.

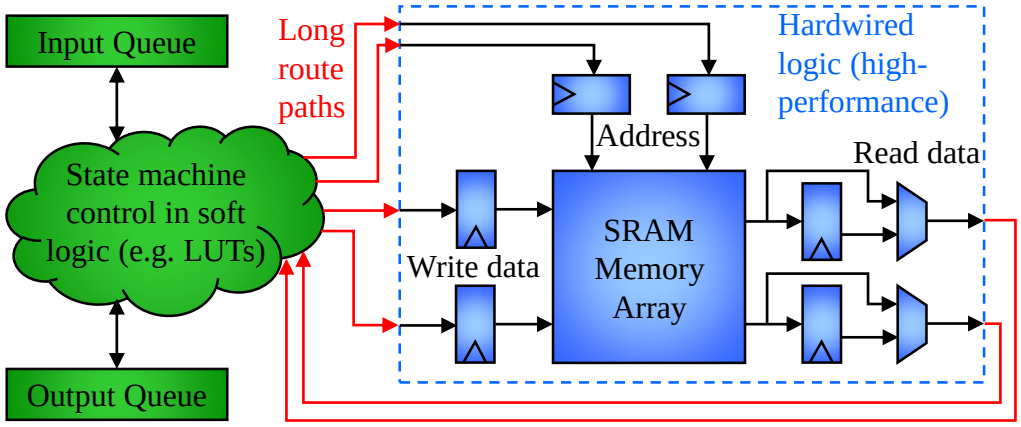


Figure 4.10: A long feedback path can limit the clock speed of the state machine.

4.5 Clause Traversal State Machine

From the analysis of our hardware memory layout in section 4.3, it follows that we need up to four memory accesses per clause visit:

1. Read the local link (located at `base` + `offset`).
2. Read the existing variable assignments (located at `base` − `thread` − 1).
3. If necessary, read the global link (located at `base` + `offset` + `overflow`).
4. Write back the updated variable assignments (same memory location).

Because of the data dependencies, accesses 3 and 4 must happen after 1 and 2. Modern FPGAs have dual-port embedded memories, so we could visit one clause every two clock cycles. The simplest solution is to visit clause n on clock cycle $2n$ (accesses 1 and 2) as well as on clock cycle $2n + 1$ (accesses 3 and 4). However, using data that was just read to access new data on the next clock cycle results in an extremely long feedback path thereby significantly reducing the clock frequency.

Modern FPGAs use hardwired logic for the embedded memories to increase the clock frequency and density of SRAM memory bits. Consequently, they are physically placed some distance away from the reconfigurable logic where we implement the state machine. This creates long route paths which are therefore slow, as shown by the red arrows in Figure 4.10. Using data read on clock cycle $2n$ to control the address and write data for clock cycle $2n + 1$ requires absolutely no registers in the feedback path. This results in a very long propagation delay through the combinational logic:

1. Starting from the hardwired address and write data registers (in blue in Figure 4.10), there is delay through the SRAM memory array itself.

2. A long route path from the unregistered read data adds delay (shown in red).

3. The state machine must arbitrate whether it will continue its current link list traversal, whether it will start a new traversal from the input queue, or whether it must stall because the output queue is full. These queues are the same ones as in Figure 4.4 and they interface to the network.

4. A long route path from the state machine to the hardwired address and write data registers adds delay.

This ultimately results in a very slow clock frequency, hence our implementation uses aggressive pipelining. We choose the registered path for the hardwired read data from memory (in blue in Figure 4.10). We also register the address and write data within our state machine (in green) to mitigate the long route path that follows.

As a result of our pipelining, reads from memory now have a 3 clock cycle latency. We now use two independent execution strands to use up what would otherwise be clock cycles of unused memory access since we are waiting on read data. We can visit

Clock cycle	Execution Strand 0		Execution Strand 1	
	Address A	Address B	Address A	Address B
0	Read local link 0	Read var assign 0		
1				
2			Read local link 6	Read var assign 6
3	Read global link 0	Write var assign 0		
4	Read local link 1	Read var assign 1		
5			Read global link 6	Write var assign 6
6			Read local link 7	Read var assign 7
7	Read global link 1	Write var assign 1		
8	Read local link 2	Read var assign 2		
9			Read global link 7	Write var assign 7
10			Read local link 8	Read var assign 8
11	Read global link 2	Write var assign 2		
12	Read local link 3	Read var assign 3		
13			Read global link 8	Write var assign 8

Figure 4.11: The timing of our state machine for traversing link lists to visit clauses. The two execution strands operate independently (we arbitrary start at clause 6 in strand 1, just to show we are not revisiting clauses from strand 0).

two clauses every four clock cycles (this is still the same throughput), yet the clock frequency should be significantly higher, as our pipelining sliced the feedback path into three sections. The timing of our state machine is illustrated in Figure 4.11.

Address A and Address B access mutually exclusive regions of memory. We avoid one address reading from the same memory location written to by the other address on the same clock cycle, as this can have different behavior on different FPGAs.

One minor complication that arises with two execution strands is the pipeline hazard of potentially reading stale variable assignments. For instance, on clock cycle 4, strand 0 may happen to read the same memory location that strand 1 writes to on clock cycle 5. Since we already have a backpressure mechanism (if the output queue is full), we can redo the clause visit on strand 0 at clock cycle 8.

4.6 Summary of Hardware BCP

In this chapter, we have derived an intrinsic representation of SAT, which resulted in a compact memory layout. Ultimately the memory layout defines the processing required to compute BCP in hardware.

In general, implicit representations are beneficial for custom hardware. FPGAs have significantly slower clocks than CPUs, so they compensate by using specialized control and data paths to perform the equivalent of several CPU instructions within a single FPGA clock cycle.

Certain types of memory compaction enable one to do more computation with the same amount of memory bandwidth. This also facilitates the use of faster memory, which is typically only available in smaller capacities. In particular, we have restructured the representation of SAT to minimize the amount of dereferencing as well as use lossless compression. Similar techniques could be used in other memory bound applications, where it is often important to maximize the functionality per bit of memory.

Finally, another way to mitigate the slower FPGA clock is with parallel processing. We partition the SAT problem across numerous hardware processors which each operate on local and private memory. Beyond some threshold of parallelism, access to shared memory would become the bottleneck. We are already seeing data starvation on multicore CPUs, and the effects are observable in our results throughout chapter 9.

Chapter 5

Hardware Communication and Integration

The previous chapter established the use of multiple processors, each operating on one clause partition of a SAT problem. In this chapter, we examine the *communication* between hardware BCP processors as well as to software which performs all of the non-BCP aspects of the SAT solver. We classify different types of communication so that we can exploit their properties to obtain efficient implementations. Finally, we summarize the interconnect and discuss high-level FPGA floorplanning.

5.1 System Overview

We motivated the use of heterogeneous computing in section 3.4. In our system, only Boolean constraint propagation is offloaded into custom hardware, thus software implements the remainder of the SAT solver (choose the next variable to assign, conflict analysis, etc.).

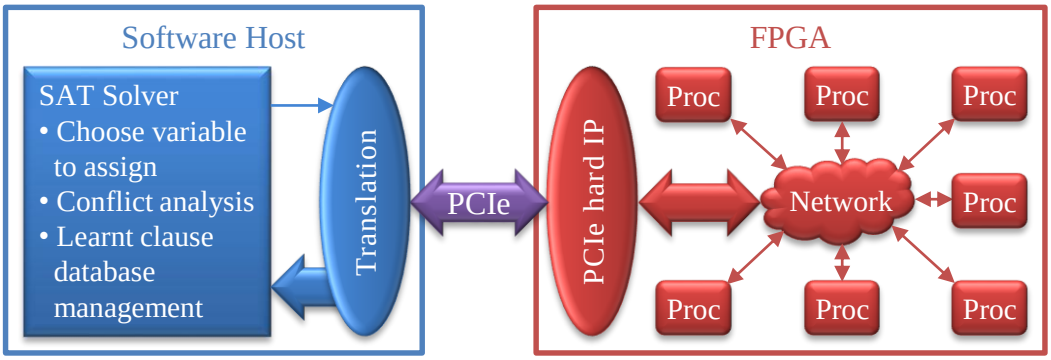


Figure 5.1: Our SAT system accelerates only BCP in custom hardware with multiple processors that connect using an on-chip network. Translation converts software variables to/from hardware addresses and offsets. Arrow thickness is an approximation of the required bandwidth.

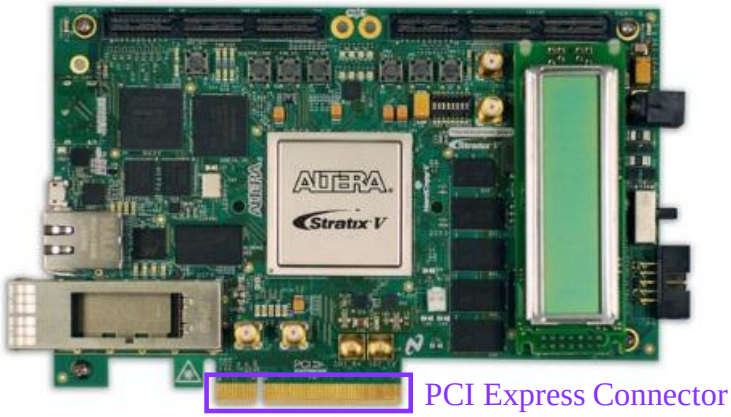


Figure 5.2: We used the Stratix V DSP Development Kit from Altera [59].

Lossless compression can be used to reduce the number of bits needed to represent pointers or addresses, as detailed in section 4.2.3. The resulting localism of addresses inherently partitions the problem, which facilitates the distribution of one SAT problem over multiple processors on the same FPGA. Figure 4.4 briefly illustrated that these processors can be connected using some kind of network. Obviously this network must also connect to some FPGA external interface so that BCPs produced in hardware can be reported to software.

An overview of our SAT system is illustrated in Figure 5.1. We use PCI Express [83] to communicate between the FPGA and the CPU motherboard (software). PCIe is commonly available on many modern FPGAs, such as the Stratix V DSP Development Kit from Altera (Figure 5.2) which we prototyped on. PCIe is a high-speed interface which is also backwards compatible with legacy PCI. Almost all modern video cards (GPUs) use PCIe to connect to the CPU motherboard. Stratix V (Altera [59]) and Virtex 7 (Xilinx [60]) FPGAs both have third generation PCIe, which operates at 8 gigabits per serial lane. Consequently, FPGAs have hardwired logic to integrate this high-speed external interface with the slower reconfigurable logic.

Other external interfaces could be used instead of PCIe, such as HyperTransport [84]. In fact, our results could potentially be better with HyperTransport because it typically has lower latency than PCIe.

In software, to continue the DPLL tree search after finishing a run of BCP, software sends the next variable to assign to hardware. This must be translated from a software variable to a hardware address and offset (as required by our hardware memory layout in section 4.3). Translation is implemented as a lookup table in software. For each new assignment, hardware may report back numerous BCPs, hence the asymmetrical bandwidth in Figure 5.1. Likewise, hardware BCPs require translation to software variables. If a conflict is found, software issues a deassign index to hardware (e.g. deassign starting with the most recent variables until there are only 25 variables still assigned). Deassignment is managed entirely in hardware, as shown in section 5.9.

We discuss the handshaking between hardware, our own Linux PCIe driver, and the application software in section 7.2. The remainder of this chapter examines the details of the on-chip network within the FPGA.

5.2 Classification of On-Chip Communication

As already hinted thus far, some communication infrastructure is required to transfer a link list traversal task between processors.

On-Chip Communication Type 1. *Network-on-chip architectures are suitable for randomized communication patterns.*

Each processor performs BCP which involves random access, as examined in detail in section 3.3.1. Therefore it is reasonable to expect the communication patterns to also exhibit randomized behavior. As discussed in section 5.7, network-on-chip architectures offer a trade-off between the amount of connectivity and resource usage.

Our random access network is presented in section 5.7. Unfortunately, it is actually not suitable for *every* type of communication. One of the key complications is that Boolean constraint propagation is a form of *dynamic task creation*.

When we start BCP, it is impossible to know how many variables will be implied. This can only be *discovered* by actually doing the implication. Upon discovering a variable must be implied, for each of these variables we must perform extra work (in the form of a link list traversal). In case the processor is currently busy with other work, this new extra work can be enqueued. In practice, the depth of this queue is finite and therefore this limits the maximum amount of work that can be deferred.

On-Chip Communication Type 2. *Dynamic task creation can be resolved with communication infrastructure to offload and reload deferred work.*

Some buffering is required for the offloaded items, and this is shared between several processors to amortize the cost. In section 5.3, we explain why independent

control flow for different network traffic classes is required in order to prevent the network from clogging, which ultimately leads to deadlock.

Another key complication is determining when a run of BCP has finished. This arises due to the *decentralization* of dynamic task creation.

One solution is for each processor to send a "start" message upon creating a new link list traversal task (when a newly implied variable is discovered). Each processor also sends a "finish" message upon finishing a traversal (when the incoming variable assignment matches the existing one). By collecting these messages at a centralized place, we can determine whether any link list is still being traversed.

This scheme will fail if the network cannot guarantee the latency. For example, if variable a implies b, then the start message for b must be *produced* before the finish message for a. However, if for any reason these *arrive* in the opposite order at the centralized place, we will erroneously think that BCP had finished with variable a (variable b would not be considered part of this run of BCP).

On-Chip Communication Type 3. *Synchronization involves lossy communication, for which counting-based networks are more suitable than message-based networks.*

Notice that at any given point in time, either BCP is active or it is not. We are not interested in which specific variables are currently being traversed or which specific processors are active. We exploit this *lossy* characteristic by counting the number of start and finish messages, as presented in section 5.6.

Instead of significantly over-designing the random access network to guarantee latency and tolerate overflow offloading, we have isolated each class of communication in order to exploit properties which are unique to that specific class. This results in a collectively more efficient implementation.

5.3 Network Design Guidelines

Communication infrastructure can be regarded as an overhead, since no computation is being performed yet FPGA logic resources are used. Ideally the processors should use most of the logic resources. Unlike much of the networks-on-chip research community [85], *we are not interested in sophisticated (and typically expensive) optimizations to obtain incremental improvements in performance*. From a practical perspective, we need just enough network resources to avoid starving the processors of data.

All networks absolutely must be pipelined for physical routability reasons. Trying to send a signal physically from one side of the FPGA to the other side within the same clock cycle will result in an extremely low clock frequency due to extensive routing delay. We must also pipeline backpressure signals, which indicate someone downstream cannot absorb any more network traffic. Consequently, we must either use physically localized backpressure, or we must send an early notification to compensate for the pipeline delay (e.g. when a queue is almost full).

The registers that are required for the network pipeline stages effectively act as a queue with a capacity of 1 item. Minimal buffering for network traffic is already established, thus to minimize resources we should avoid additional buffering.

5.4 Separation of Traffic Classes

As discussed in section 5.2, BCP is a form of dynamic task creation. Upon executing BCP, we discover how many variables need to be implied. For each implied variable, we add a link list traversal task to the input queue. When the processor later accepts this task from the input queue, it will visit all clauses that contain this variable.

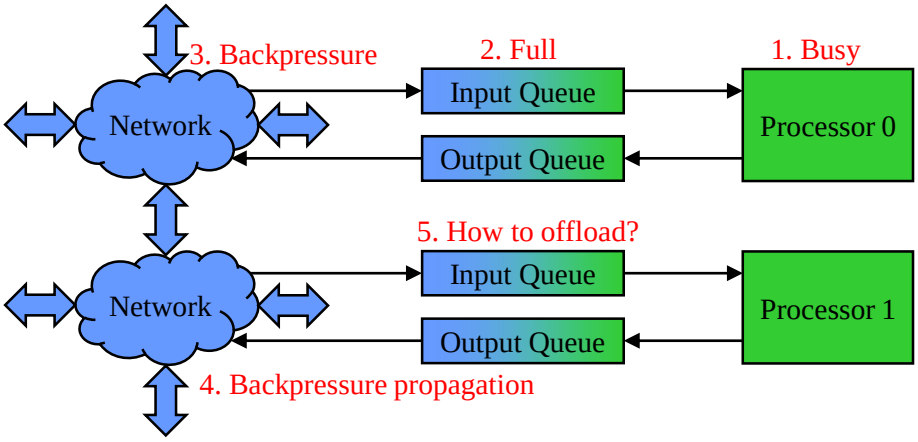


Figure 5.3: Backpressure can propagate through the network.

The input queue will fill up if tasks are added faster than the processor can empty it. Tasks can be added by discovering a BCP or by other processors transferring a link list traversal to here. When the input queue fills up, we apply backpressure into the network. Although this stalls other processors from transferring a link list traversal to here, it does not resolve the dynamic task creation overflow problem.

As shown in Figure 5.3, as the input queue to processor 0 fills up, backpressure is applied into the network. Networks have a finite capacity (a limited number of packets that can be buffered), and thus the backpressure can propagate and potentially block output queues in other processors from emptying.

In Figure 5.3, if processor 1 now produces many BCPs, it is extremely difficult to guarantee that these can be offloaded via the output queue. This is due to the *cyclic dependencies* inherent to *any* network in which node A can send to node B and node B can also send to node A (our network requires this due to the random access nature of BCP).

To break this cyclic dependency, we must use *independent control flow* for each class of network traffic. In addition to synchronization, three other classes are needed:

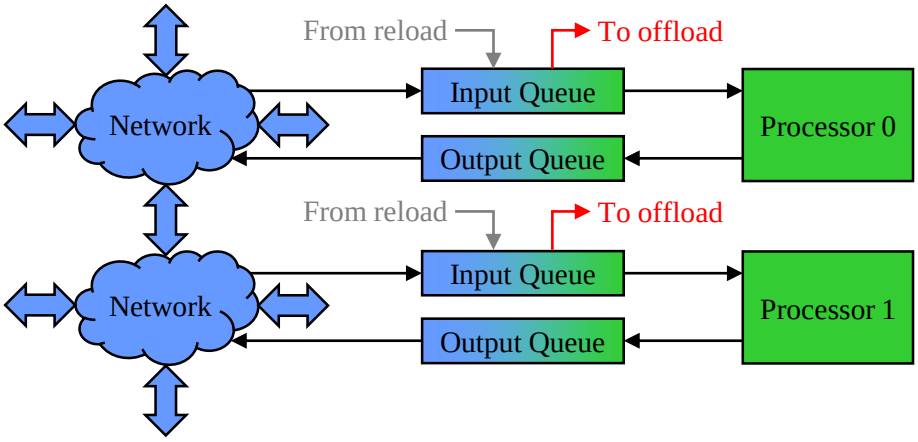


Figure 5.4: Using separate paths to offload and reload tasks can hide the small capacity of the input queue from the network on the left.

1. Offload a task (highest priority).

2. Transfer a link list traversal task to another processor (normal priority).

3. Reload an earlier offloaded task (lowest priority).

Independent control flow means that a task being offloaded cannot be blocked by backpressure from transferring a link list traversal task to a processor whose input queue is full. Consequently, we must replicate network buffers per traffic class.

In the case where these buffers are registers, this is essentially a replication of the data path itself. This is actually beneficial because for each traffic class, we can individually optimize many features of that particular network, such as the number of bits in the data path, the arbitration logic, the backpressure mechanism, etc.

As shown in Figure 5.4, tasks would no longer need to be offloaded via the output queue. Reasonable priorities must also be set, for example to avoid clogging the network by prematurely reloading tasks. Therefore the arbitration for accepting tasks into the input queue should always favor the network over the reload pathway.

5.5 Distributed Shift Register Networks

In this section, we present the pathways for offloading and reloading tasks, which takes the form of a distributed shift register. This infrastructure enables BCP to dynamically create more tasks than the input queue in each processor can store.

As motivated in the previous section, separate data pathways are used for offloading, transferring, and reloading tasks. One key distinction between these three types of communication is the network topology:

1. Offloading uses a many-to-one topology (several processors may offload to one shared buffer).

2. Transferring a link list traversal is a many-to-many topology (random access).

3. Reloading is a one-to-many topology (the reverse of offloading).

We obviously need some on-chip memory for the overflow buffers. Each buffer is shared over many processors to amortize the cost. Given this necessary buffer, we attempt to minimize the use of extra resources.

5.5.1 The Offload Network Structure

The structure of the offload network is shown in Figure 5.5. Processors are physically spread apart, thus pipeline registers are required. The distributed shift register is a *stall-free* pathway, so there must be space available in the overflow buffer before a processor is allowed to offload into the shift register. Arbitration in the distributed shift register always favors items that are already inside of the shift register. Conclusively, once an item has entered the shift register, that item will eventually be committed to the overflow queue.

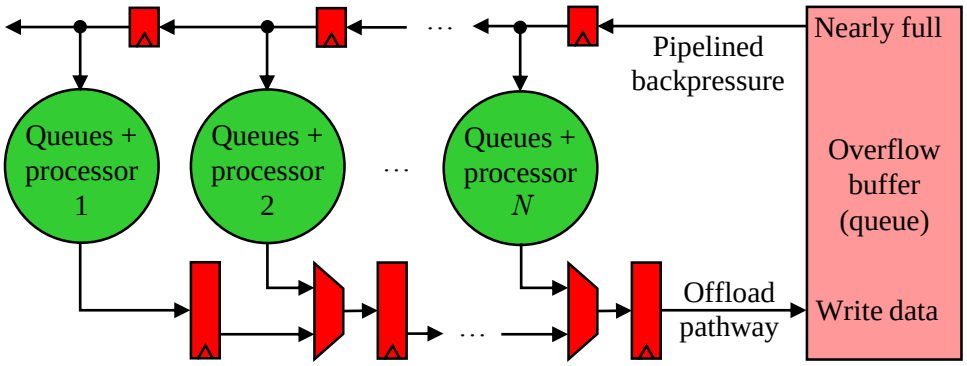


Figure 5.5: The offload pathway uses a distributed shift register.

Each pipeline stage in the offload pathway consists of:

- The offload data itself,
- A 1-bit valid signal, and
- An address to identify which processor it came from (so that we will later know where to reload this task to).

When the queue is almost full, it sends a pipelined early backpressure signal to disable all sources from inserting into the shift register. If there are S pipeline stages, then the queue must issue a backpressure signal when it is $2S$ items from being full. Up to S items could arrive over the S clocks it will take for the backpressure signal to propagate all the way to the first processor. Processors are no longer allowed to offload at this point in time, however the shift register could already be fully populated, so up to S more items may still arrive as the shift register empties.

Because we use a stall-free pathway, we do not need extra stall buffers (in addition to the pipeline stages). If a pathway could stall, there would be a point in time where the pipelined backpressure signal is stalling downstream traffic but not yet stalling

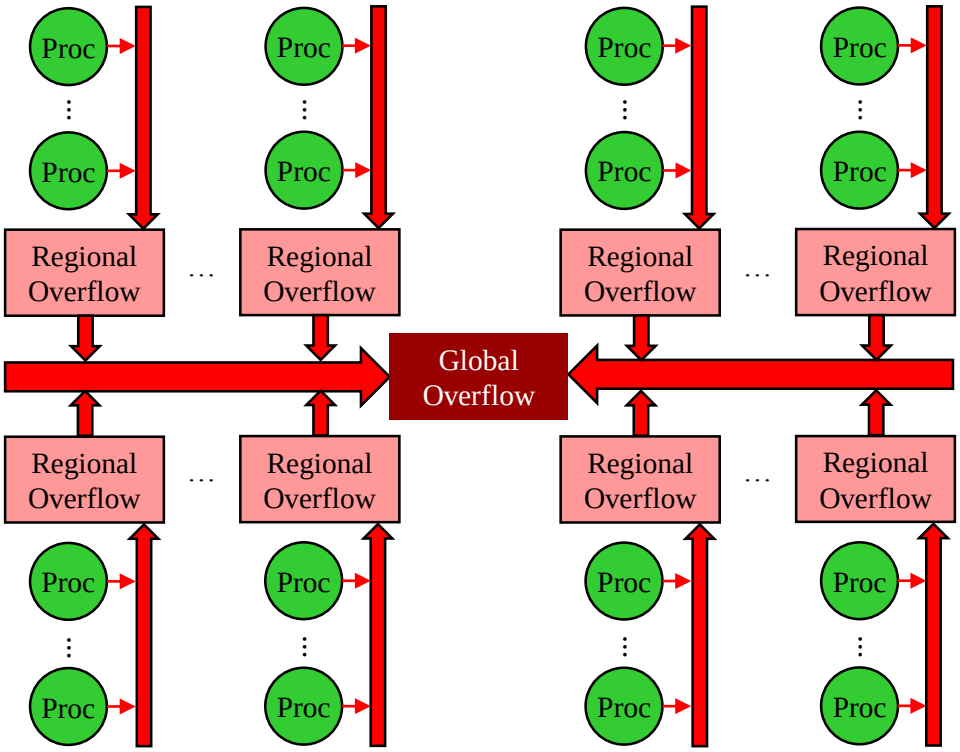


Figure 5.6: Processors offload to regional overflow buffers, which then offload to a global overflow buffer, thus forming a hierarchy. Arrow thickness represents the supported bandwidth. Pipelined backpressure signals are not shown for ease of presentation.

upstream traffic (since the pipelined backpressure has not yet propagated all the way upstream). Consequently, additional stall buffers would be needed to absorb the network traffic caught in between. We avoid this by using an early backpressure to prevent all sources from offloading instead of stalling within the pathway.

If an overflow queue itself fills up, it offloads to an even larger queue downstream. Applied recursively, this leads to a hierarchy: each group of processors offloads to a regional overflow buffer, and the regional overflow buffers offload to a global overflow buffer. This 2-dimensional structure is shown in Figure 5.6.

The regional overflow buffers (and each of their corresponding group of processors) operate concurrently. Backpressure is applied independently within each group.

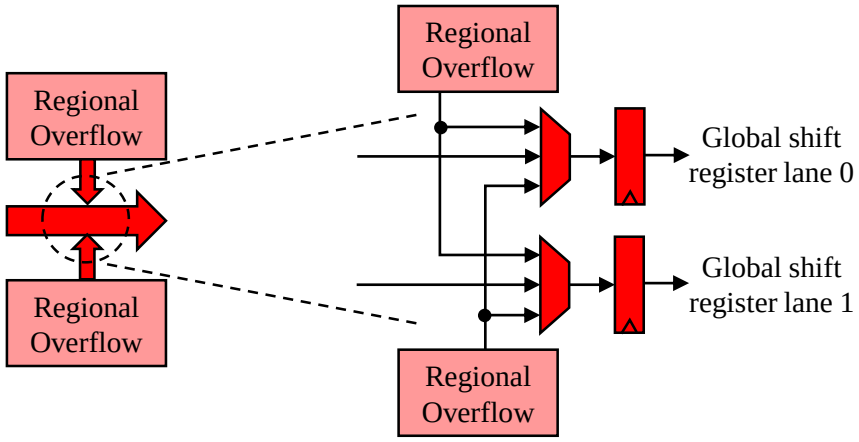


Figure 5.7: The global offload shift register has two lanes for increased bandwidth.

Using several small regional overflow buffers facilitates a high-throughput offload infrastructure for small amounts of overflow, which may not be rare events.

As presented in section 5.5.3, this distributed shift register is not used exclusively for offloading. To increase the performance, we intentionally used half-row and half-column groups to increase the amount of parallelism at the expense of more resources. Furthermore, we have two lanes of global offload shift registers for higher bandwidth, as illustrated in Figure 5.7.

With the appropriate sizing of buffers, large amounts of overflow (which exceed the capacity of regional overflow) should be uncommon events. Only a single global overflow buffer needs to be sized to tolerate the worse case maximum number of tasks that can be dynamically created. Moreover, part of this buffer can exist in *off-chip memory*. For example, if the on-chip global overflow buffer fills up, it can *temporarily* apply backpressure to all regional overflow buffers while it offloads some data into the CPU's memory via PCIe. Later when the global overflow buffer is nearly empty, it can ask for its data back from software's memory. The larger reload latency of the external interface can be hidden from the processors by the regional overflow buffers.

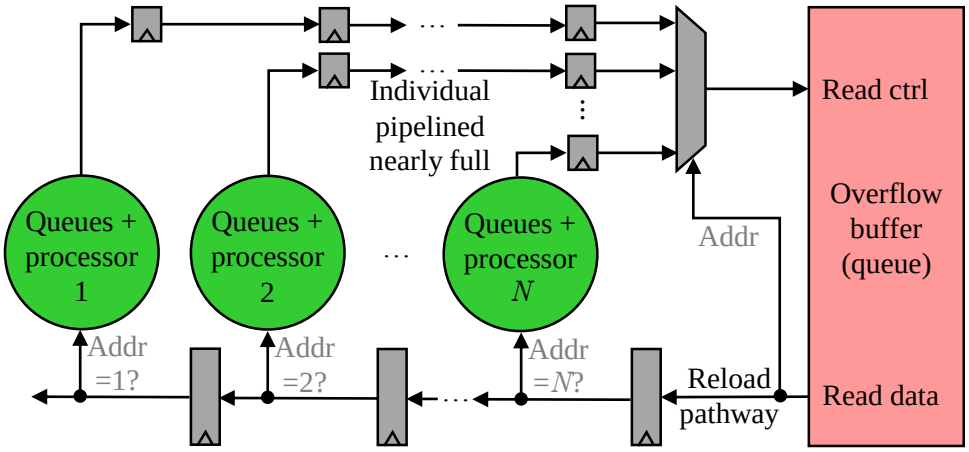


Figure 5.8: The reload network also uses a distributed shift register. Each processor must decode whether the reloaded task is for itself. The overflow buffer does not release the task unless it is sure the that destination has space available for it.

5.5.2 The Reload Network Structure

Any task that was offloaded must eventually be reloaded to the processor that originally offloaded it. The reload network essentially works in the opposite direction of the offload network: the global overflow buffer feeds the regional overflow buffers, and each regional overflow buffer feeds its own group of processors. Like offload, reload ensures that the destination has space available for the task before it sends that task. This enables the use of a stall-free pathway which eliminates the need for stall buffers, as discussed above.

Unlike each offload which only has one destination, each reload has several possible destinations. Thus, deciding whether the destination has space available requires:

- All destinations must send their own "nearly full" signal (early backpressure).
- Recall that each offloaded task had an address component that indicates which specific processor it came from. We use this to select the appropriate nearly full signal for the current task that we want to reload.

The reload network also uses a distributed shift register. However, unlike offloading which distributes the arbitration of whether a task can enter the shift register, reload does all of the arbitration at the single point of insertion. In addition, since reload has multiple destinations, each destination must decode the address component to determine whether it should extract the reloaded task. This is shown in Figure 5.8.

5.5.3 Reusing the Offload Network for BCP Collection

In general, the offload network structure is suitable for *any* type of communication that uses a many-to-one topology. As BCPs are produced by the processors on the FPGA, eventually we need a way to report them to software. The FPGA obviously requires an external interface, and ultimately this is a *point of serialization*.

We expect most of the traffic on the global distributed shift register to be BCP collection. As mentioned at the end of section 5.5.1, with the appropriate sizing of regional overflow buffers, global overflow is not expected to be common.

The distributed shift registers network as a whole can be regarded as a distributed arbitration to a serial interface. Every BCP that uses this network eventually needs to go across PCIe. Furthermore, the bandwidth of the external interface is known, so we can adjust the bandwidth of the distributed shift registers accordingly.

The bandwidth requirement is larger for the global shift registers (which run horizontally in Figure 5.6) since they collect the network traffic from all regional shift registers (which run vertically in Figure 5.6). We only used two lanes. Even at 200 MHz, this provides up to 400 million BCP/s, which is about 80× the speed of single-threaded software. Additional lanes are unlikely to help since the non-BCP parts of the SAT solver in software would impose a lower speed limit.

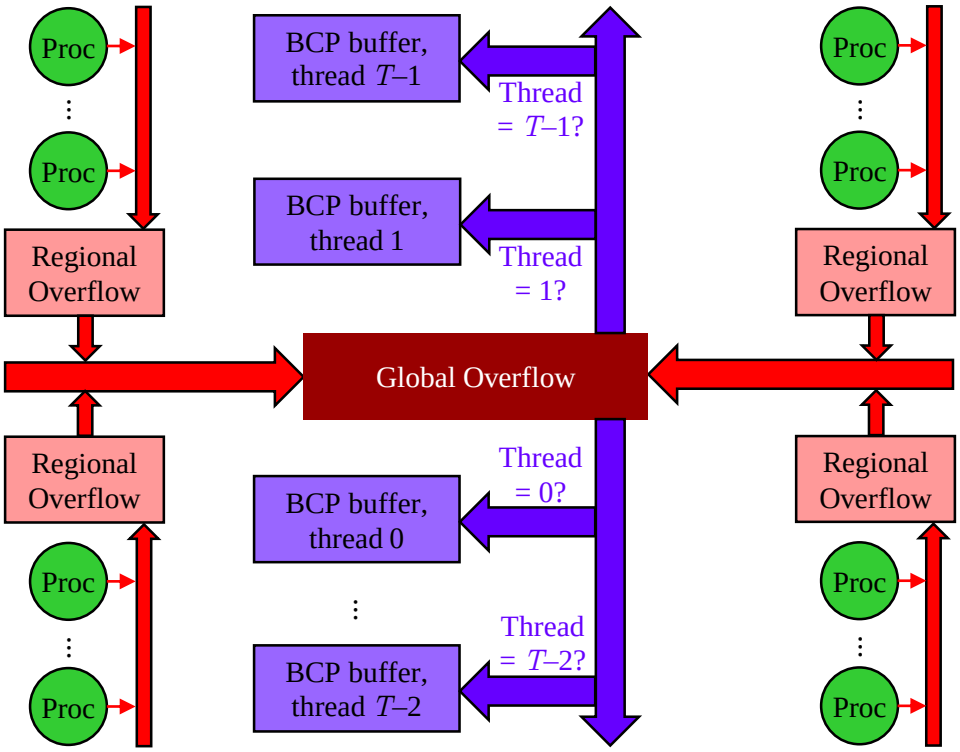


Figure 5.9: BCP collection reuses the offload infrastructure (we removed some processors and regional overflows for ease of presentation). When a BCP arrives at the centralized global overflow buffer, it is transferred to a reload network (shown in purple) where each thread-private buffer will extract its own BCPs.

We separate BCPs according to thread in hardware (before these are sent over PCIe) in order to minimize the overhead of synchronization between different software threads. Once items reach the central global overflow buffer, anything that is not an offloaded task must be a BCP which needs to be reported to software. These BCPs are sent to a *reload* network where thread-private BCP buffers extract them from the distributed shift register based on the thread. This is shown in purple in Figure 5.9.

As one can probably infer, the BCP buffers use an offload network to arbitrate sending data to the PCIe interface. This is yet another example of a many-to-one topology. The specifics of the BCP buffers are discussed in section 5.9.

5.5.4 Reusing the Reload Network for Memory Initialization

We begin by examining the *entire* offload/reload hierarchy for resolving an excessive number of dynamically created tasks:

Localized: If a processor discovers more BCPs than it can enqueue, it offloads some of them to the regional overflow buffer within its own processor group. The regional overflow buffer will reload a task if the corresponding processor has space available. Processor will not re-imply a variable and since there are a finite number of variables, eventually all processors will empty their queue as they perform link list traversals.

Regional: If a regional overflow buffer fills up, it offloads to the global overflow buffer. The global overflow buffer will reload a task if the corresponding regional overflow has space available. Since processors will eventually empty their queues, regional overflow buffers will also eventually be able to reload.

Global: If the global overflow buffer fills up, it offloads into the CPU's memory via PCIe. This is entirely managed by hardware, so hardware must also retrieve the data when the global overflow buffer later becomes sufficiently empty.

Software knows where in the CPU's memory the FPGA will offload global overflow data. In this memory region, software can write the data for initializing the FPGA's memory (to load an arbitrary SAT problem). To bring this into the FPGA, we "trick" the global overflow buffer into thinking it had previously offloaded some data. All buffers are empty at power-up (after a hardware reset), so the global overflow buffer will retrieve the data and immediately reload it to the regional overflow buffers, which pass it along to the processors. To inform all processors that this data is not a link list traversal, we use a pipelined broadcast signal, as discussed in section 5.6.3.

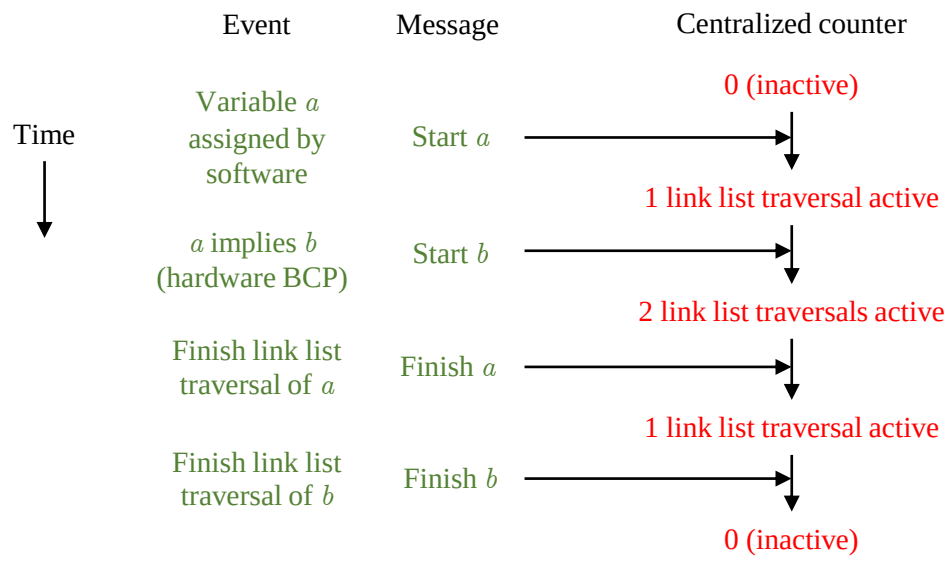


Figure 5.10: By counting start and finish messages at a centralized place, we can detect when BCP is finished.

5.6 Synchronization Networks

5.6.1 Detecting the Completion of BCP

One of the major challenges of distributing Boolean constraint propagation over several processors is determining when BCP will finish. Without dynamic task creation, if we start T tasks, then we would implicitly know that we have finished once T done messages arrive at a centralized place. Unfortunately in the case of BCP, we do not know ahead of time whether 100 variables will be implied or none at all, for example.

As illustrated in Figure 5.10, processors could send "start" and "finish" messages respectively for discovering a BCP and completely finishing a link list traversal (incoming variable assignment matches the existing one). These would be collected at a centralized place. However, this approach is sensitive to timing. If variable a implies b, then start for b must be *produced* before finish for a, however unless the network

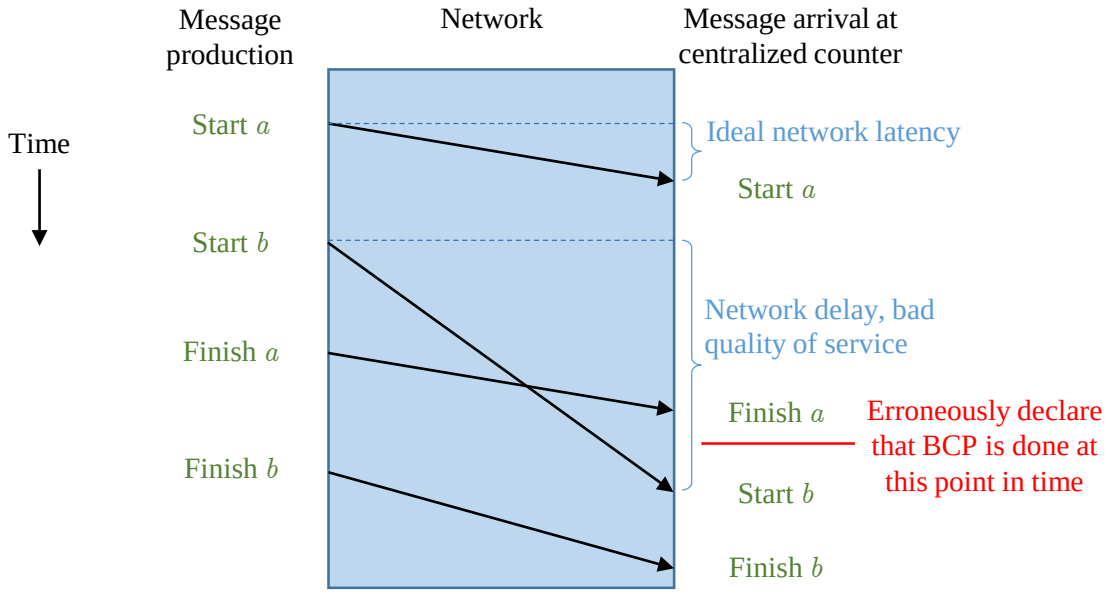


Figure 5.11: Synchronization will fail if the network cannot guarantee the latency.

can always guarantee the latency, the centralized counter that collects these messages may *observe* them in the opposite order. In that case, it would erroneously declare that BCP is done upon receiving finish for a, as shown in Figure 5.11.

Furthermore, if hardware synchronization fails, software will think that b was not implied thereby causing mismatched variable assignments between software and hardware. All of these ensuing complications can be avoided by construction if we use a guaranteed latency network in hardware.

Synchronization is unique from other types of communication because:

1. Synchronization is *latency sensitive*. Unlike the offloading of overflow tasks or the transferring of link list traversals between processors, the system will fail if synchronization messages arrive out of order.

2. Synchronization is *lossy*. At any point in time, either BCP is done or it is not. We are not concerned with which specific variables or processors are still active.

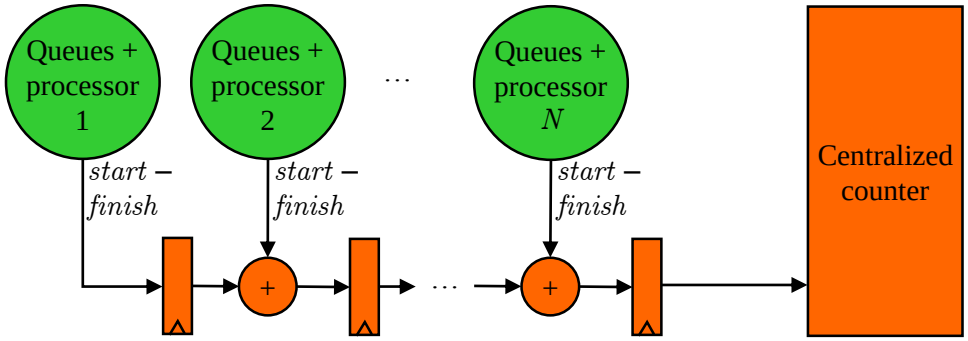


Figure 5.12: A counting-based network composed of pipelined adders.

We exploit these properties by using a lossy counting-based network instead of the typical lossless message-based network. BCP is done when the centralized counter receives an equal number of start and finish messages. We do not need to match finish for a with start for a. If two start messages arrive at the same time in the same location in the network, we can simply add them (merging is our collision resolution strategy). Likewise, a start and finish can cancel each other. Conclusively, our counting-based network has *guaranteed latency* because it never uses backpressure.

To mitigate long physical distances, we use pipelined adders, as shown in Figure 5.12. A centralized counter keeps track of the cumulative difference between the total number of start and finish signals received. Using purely combinational adders, it could declare BCP is done as soon as this difference is zero. If the longest pipeline path adds P clock cycles of latency, BCP is done if this difference is zero over P consecutive clock cycles.

To reduce this latency P and thereby more concisely detect when BCP is done, we must shorten the path from the centralized counter to the farthest processor. This is achieved by using a 2-dimensional hierarchy similar to the offload and reload networks from section 5.5. As illustrated in Figure 5.13, the longest path is the length of a

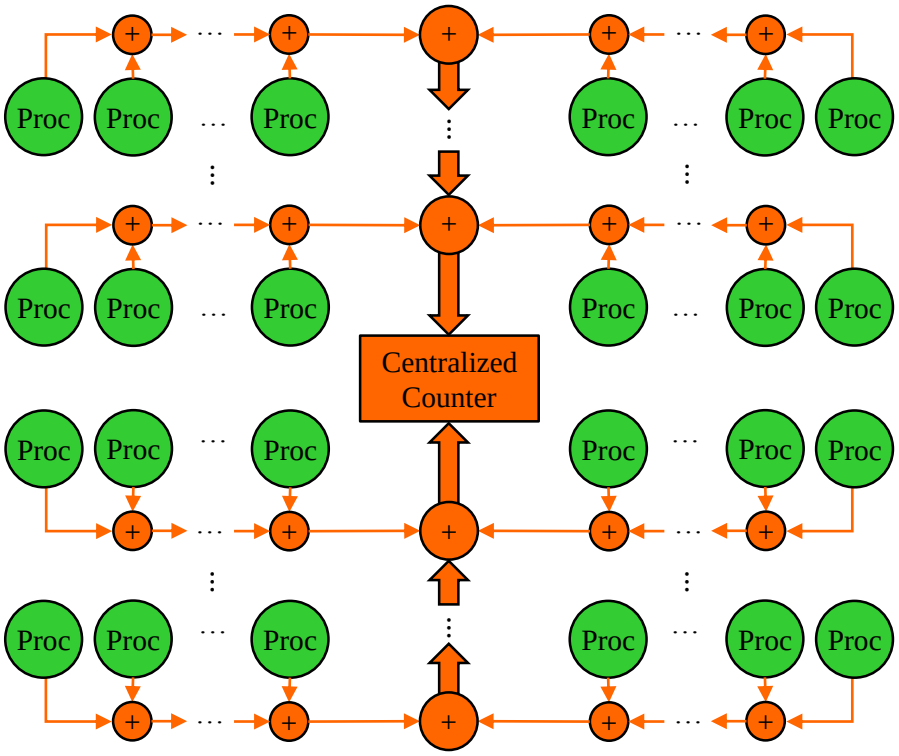


Figure 5.13: Using a 2D topology reduces the network latency. The pipeline registers are not shown for ease of presentation. Arrow width and adder size represent the number of bits needed.

half-row (for adding individual votes from a group of processors) plus the length of a half-column (for adding regional sums). The centralized counter is physically placed in the center of the FPGA to minimize the worst case distance.

This entire pipelined adder structure is replicated per thread. Each processor actually reports the *difference* of start minus finish. For example, if a processor discovers a new BCP on thread 1 and finishes the link list traversal for thread 3 on the same clock cycle, it would report [0 +1 0 −1] to the synchronization network for threads 0 to 3 respectively. Since there is no backpressure, every processor must report to the synchronization network on every clock cycle (all zeros can be reported).

5.6.2 Ensuring Software Receives Every BCP

The synchronization network described so far will correctly detect when BCP has finished, however BCPs are collected via the offload network before being sent across PCIe to software. The offload network provides no guarantee on the latency since arbitration favors items that are already inside of the distributed shift register, so one could wait for an arbitrary amount of time before finally making an insertion.

In other words, if variable a implies b, we should not declare BCP is done until b arrives at the BCP central buffer (the purple central column in Figure 5.9). The fundamental problem with the existing synchronization scheme is that the finish message for a can cancel the start message for b *within* the pipelined adder network. In this case, the BCP central buffer has no way of knowing to expect b.

If b is significantly delayed by the offload network, it could be reported in the wrong round of BCP. For example, by the time b arrives at the BCP central buffer, hardware had already finished reporting the BCPs to software, software had already issued the next variable to assign (continuing the DPLL tree search), and hardware has already started running the next round of BCP.

To avoid such synchronization problems, our actual implementation uses the following voting (this is still replicated per thread):

Start: Discovery of a BCP contributes a vote of -2 on that specific thread.

Finish: Completely finishing a link list traversal (incoming variable assignment matches the existing one) contributes a vote of $+1$ on that specific thread.

Collect: Arrival of a BCP at the central buffer contributes a vote of $+1$ on that specific thread.

Every BCP must eventually end its link list traversal as well as arrive at the BCP central buffer. As before, the centralized counter keeps a cumulative sum of all the votes, and BCP is done when this sum is zero for P consecutive clock cycles.

In general, any combination of nonzero weights can be used for start, finish, and collect so long as they sum to zero. Also, finish and collect must have the same sign. In 2's complement binary, the values of -2 (1110) and +1 (0001) occupy mutually exclusive bits, thus processors can report the signed value of "start + finish" to the synchronization network without using an adder (bit replication and rewiring cost no FPGA logic resources).

5.6.3 Conflict Detection and Reporting

A conflict is an inconsistent assignment to the variables. For example, if we assign $a = 0$ in $(a + b)(a + \overline{b})$, the first clause implies $b = 1$ whereas the second one implies $b = 0$. A processor detects a conflict when the incoming assignment is true and the existing assignment is false, or vice versa. We simply replace the adders with OR gates in the pipelined counting-based network to determine if any processor has detected a conflict. As before, the entire structure is replicated per thread.

Upon *any* processor detecting a conflict on thread t, we should inform *all* processors as soon as possible to minimize wasted work. If a processor knows that thread t is in a conflicting state, it will abort all link list traversal tasks that use thread t (by sending a finish vote even though the link list is not fully traversed). As explained in section 4.4.1, any variable assignments left in an inconsistent state will be deassigned immediately after conflict analysis in software, thus restoring them to a consistent unassigned state.

We use a pipelined broadcast signal (1 bit per thread) to report the conflict status on every clock cycle. Once the conflict status turns on (for a given thread), it does not turn off until BCP is done (all link list traversal tasks are finished or aborted, and all BCPs are collected at the BCP central buffer). Since the conflict status is a slowly changing signal, one could instead send *updates* by sending the thread id (e.g. only 3 bits for 8 threads), however each processor would need to store the conflict status locally. By reporting the status itself, each processor can always read it from the registers in the pipelined broadcast (without additional registers for storage).

Another usage of pipelined broadcast is to inform all processors whether data arriving is initial memory data (for loading an arbitrary SAT problem onto the FPGA) or whether it is a link list traversal task. Although this information could be contained within the data itself, it requires fewer logic resources to use a pipelined broadcast than to widen all queues by 1 bit.

5.7 Random Access Network

Any processor can transfer a link list traversal task to any processor (including itself, as demonstrated in section 5.10). We must ensure complete connectivity, however we can share intermediate terms to reduce the amount of resources used. This essentially *distributes* the connectivity. For example, instead of directly connecting every pair of processors (which would require large multiplexers), we could arrange them in a ring where each processor can only send data to its left neighbor. By decomposing the multiplexers that connect every pair of processors, each shared intermediate term essentially becomes a network node that may be visited in order to eventually get to the desired destination.

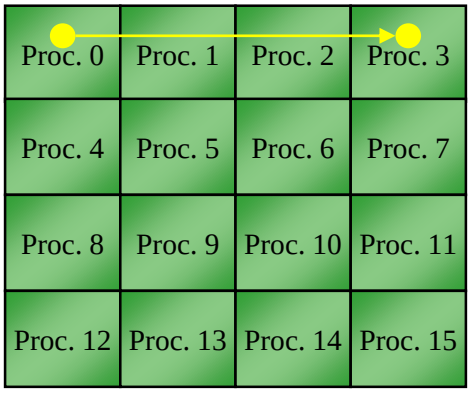


Figure 5.14: To send data from processor 0 to processor 3, it physically has to pass through the area occupied by processors 1 and 2.

The above examples are extremities. Network-on-chip architectures offer a middle ground where one can choose the appropriate amount of connectivity to balance network resources versus performance for the given application. In our case, each processor communicates with up to four neighbors, as discussed below.

5.7.1 2-Dimensional Grid Network Topology

Numerous processors are *physically spread* across the entire two dimensional surface of the FPGA. Figure 5.14 shows that any data which physically travels from processor 0 to processor 3 must pass through the area occupied by processors 1 and 2, assuming we take the shortest path. Notice that if processor 1 sends data to processor 3, part of this path can be reused or shared. By symmetry, the same applies in the vertical direction. This is analogous to having several cities arranged in a 2-dimensional grid with horizontal and vertical streets connecting only the neighboring cities (to get from city 0 to city 3, we have to travel east through cities 1 and 2). Additional roads would not affect the completeness of the connectivity, but it would allow more cars on the roads to travel between cities (e.g. improve network performance).

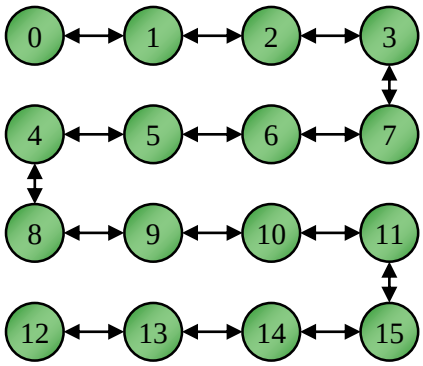


Figure 5.15: A 2-dimensional layout can be unrolled into a 1-dimensional network topology to simplify arbitration (fewer neighbors) at the expense of routing detours.

Because of the long physical distance between processors 0 and 3, we must place pipeline stages in the area occupied by processors 1 and 2. If processors 0 and 1 both need to send data to processor 3 at the same time, they must first compete for the pipeline register in the area of processor 2. Conclusively, pipelining *isolates* the distribution of the arbitration in the network.

To match the physical layout, we have chosen to use a 2-dimensional grid network topology where each processor can communicate with its horizontal and vertical neighbors. As shown in Figure 5.15, a 2-dimensional arrangement of processors can be unrolled into a 1-dimensional path to reduce the number of neighbors (which requires fewer resources for the simpler arbitration), however it causes major detours in the routing paths. For example, a 2-dimensional network can directly send data from processor 7 to 11 whereas a 1-dimensional network must go through 6, 5, 4, 8, 9, and 10. Extreme routing detours is an inefficient use of network resources.

The most general form of a 2-dimensional network switch is shown in Figure 5.16. In order to reduce the size of the multiplexers at each network output port, we can employ a *routing policy* such as dimensional routing:

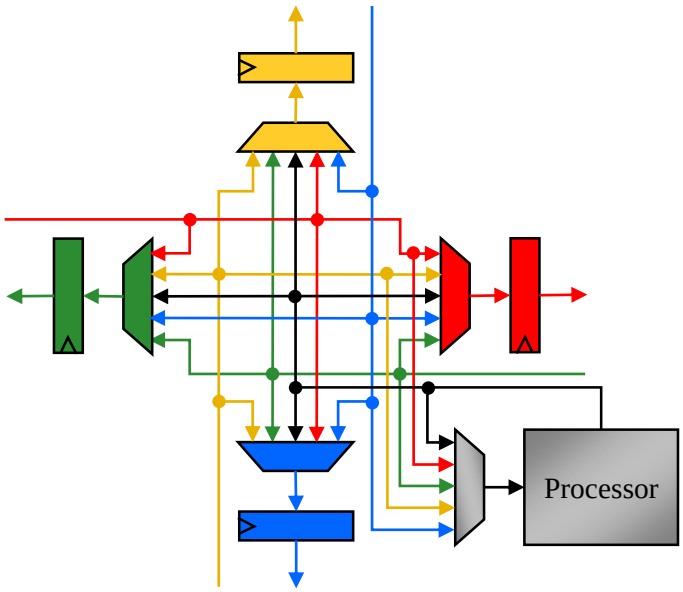


Figure 5.16: A generalized 2-dimensional network switch. The incoming red path originates from the left neighbor and may proceed in any direction. The red multiplexer determines which data to send to the right neighbor. A similar color scheme is used for the other directions. A fifth port is needed for the processor to insert and extract data from the network.

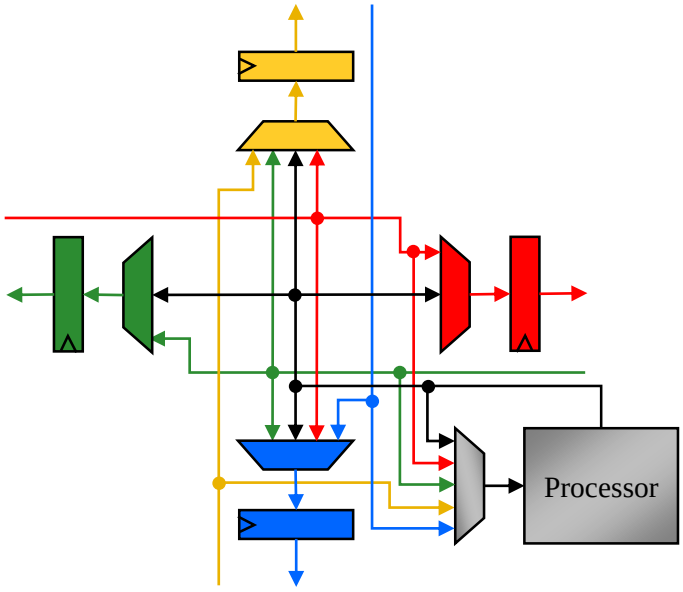


Figure 5.17: Dimensional routing (first route in x, then route in y) and removal of turn-around pathways simplifies the network arbitration.

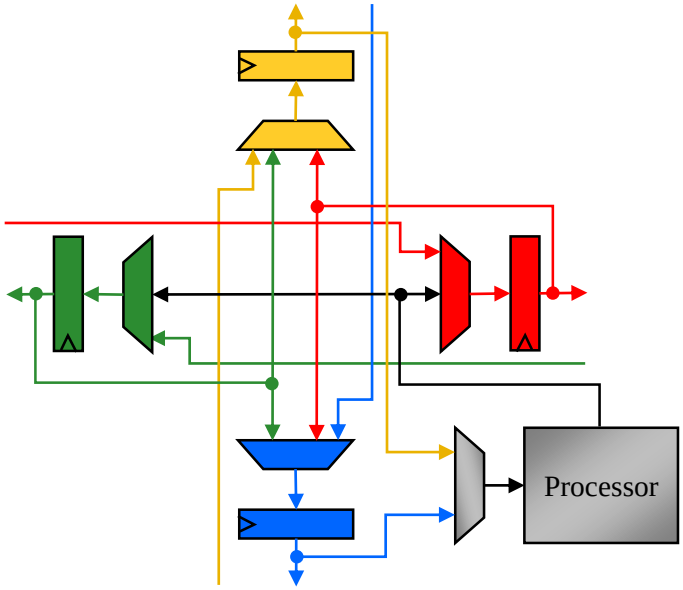


Figure 5.18: Using the outputs of the network ports to drive the inputs of the next stage of routing (first x, then y, then extraction) further simplifies the network arbitration.

1. First we route in the x direction, which brings us to the correct column.

2. We then route in the y direction to arrive at the correct row.

Once we are routing in the vertical direction, we cannot go back to horizontal routing. Therefore the multiplexers for arbitrating left and right network traffic (green and red in Figure 5.16) reduce from a 5-way decision to only a 3-way decision.

If we always route network data in the correct direction, we can remove the turn-around pathway, e.g. data that was moving left should never need to turn around and travel right. Thus the horizontal multiplexers further reduce to a 2-way decision and the vertical multiplexers reduce to a 4-way decision. This is shown in Figure 5.17.

As a final optimization, we can use the *outputs* of the horizontal routing to drive the *inputs* of the vertical routing. This is illustrated in Figure 5.18. Notice that the network insertion path (from the processor) does not need to directly supply the

vertical multiplexers, hence reducing them to a 3-way decision. Likewise the outputs of the vertical routing can be used for the network extraction path (to the processor), which reduces the processor's multiplexer to only a 2-way decision. If the processor sends data to itself through the network, this requires a route in x (to self, e.g. through the red pipeline register) followed by a route in y (also to self, e.g. through the blue pipeline register).

In conclusion, we have maintained the complete connectivity of 2-dimensional grid network topology, yet the above optimizations have significantly reduced the multiplexer cost and the arbitration complexity.

5.7.2 Network Design Guidelines and Simplifications

Certain network features are desirable in order to simplify the system-level design:

Deadlock free: Designing the network to be deadlock free *by construction* eliminates the need for timeout counters and back-off arbitration policies to try to avoid future deadlocks. This also applies to livelock, where network traffic is moving yet no functional progress is being made.

Lossless: When a link list traversal task leaves a processor's output queue, it would be beneficial to recover this space *immediately*. Furthermore, acknowledgement messages would be needed if the network could lose link list traversal tasks, which would create additional network traffic. This can be avoided if the network is designed to be lossless by construction.

Minimize buffering: Due to the limited amounts of on-chip SRAM, as much of the memory as possible should be preserved for the variable assignments and clauses.

Major simplifications can be applied to the random access network due to the co-existence with the distributed shift register networks (section 5.5) and synchronization networks (section 5.6):

Latency insensitive: The network does not need to maintain any worst case quality of service. A link list traversal task only eventually needs to arrive at its destination. Until a task arrives at the appropriate processor, the corresponding "finish" vote cannot be sent to the synchronization network, thus preventing the centralized counter from declaring that BCP is done.

Pseudo-infinite capacity queues: When a processor's input queue fills up, it will only *temporarily* apply backpressure into the network. The offload network guarantees that on every clock cycle, at least one processor can offload a link list traversal task to a sufficiently large external buffer. Only in the case where BCPs are produced faster than the offload network can tolerate, then the input queues of some processors will need to wait, but eventually they will be able to offload. From the network's perspective, each input queue behaves as if it has an infinite capacity but it may not service the network on every clock cycle. Therefore the network requires backpressure support, however we do not need to use e.g. credit-based flow control to ensure that the destination has space available before sending it some data.

As summarized in section 5.8, the random access network only needs to support the transfer of link list traversal tasks. Recall from section 4.3.1 that a link list traversal task is characterized by: `thread`, `base`, `offset`, and `assignment`. This can typically be represented with a total of 30-35 bits.

Modern FPGAs have sufficient amounts of logic resources to use 30-35 bit data widths in an on-chip network. This allows an entire link list traversal task to stay

completely synchronous (all bits are transferred together on the same clock cycle at each stage in the network).

With only single clock cycle network packets, a similar behavior is exhibited by many different network switching policies, such as store-and-forward and wormhole switching. Ultimately, at each output port of each network switch, either an entire network packet is accepted or it is not. On every clock cycle, we can decide what to do with all of the incoming network packets *within the current clock cycle*.

If two or more incoming packets at a network switch compete for the same available output port, we must backpressure all except for one of them. Each node in the network may apply point-to-point backpressure to its neighbors, and this backpressure can propagate. If an output port is not available due to backpressure somewhere downstream, we must backpressure all incoming packets that require this output port. Conclusively, our network is lossless by construction.

Once pipeline registers are added to resolve physically long routes, we do not need additional memory. Registers effectively act as queues with a capacity of 1 item, which is sufficient for our single clock cycle network packets.

Finally, it can be shown by induction that our random access network is deadlock and livelock free by construction due to the following properties:

- On every clock cycle, every packet either moves closer to its destination or maintains its current position (due to temporary backpressure). Packets never move farther away from their destination. This is enforced in both the x and y directions since our network topology excludes the turn-around pathway.

- On every clock cycle, at least one network packet can move forward due to the offload network.

Table 5.1: Summary of the on-chip networks.

	Distributed Shift Reg.	Synchronization	Random Access
Used for	Resolving dynamic task creation, collecting BCPs at central buffers, and loading initial data into FPGA memory	Determining when BCP is finished, and collecting and reporting conflicts from/to all processors	Transferring link list traversal tasks between processors
Requirement	Ensure that no queue ever overflows	Collect "start" and "finish" votes from every processor on every clock cycle	Provide connectivity between all pairs of processors
Optimization	Overflow hierarchy hides latency of single large external buffer, regional buffers increase bandwidth	Reduce latency with half-row and half-column 2-dimensional topology	Simplify network arbitration with smaller multiplexers and single clock cycle packets

5.8 Summary of the On-Chip Networks

Throughout this chapter, we have identified three distinct types of communication that arise from distributing the computation of BCP over several processors. The random access network is arguably the most obvious, as we need some mechanism to send data between any two processors. However, the *co-existence* with the other two networks is what ultimately facilitates an efficient implementation of the random access network. By isolating the features which are challenging (and costly) to implement within the random access network, we can customize each implementation to exploit any special property of that feature. For example, synchronization is lossy, hence we can resolve network collisions by simply merging the data. This can be further optimized by using a half-row and half-column 2-dimensional topology, as discussed in section 5.6.1.

Table 5.2: Comparison of the features of the on-chip networks.

	Distributed Shift Reg.	Synchronization	Random Access
Lossless	Yes	No	Yes
Latency	Guaranteed only once inserted into pathway	Always guaranteed	No guarantee
Space at destination	Space must be available before sending data	N/A	Every receiver has a pseudo-infinite capacity
Arbitration	Items already inside the shift register take priority	Votes are merged by adding them	Favor vertical network traffic, then horizontal, finally insertion from processor
Backpressure	Early backpressure if queue nearly full disables all sources from inserting	None	Point-to-point backpressure between neighbors
Topology	Offload: many-to-one Reload: one-to-many	Detect: many-to-one Report: one-to-many	Many-to-many (2D grid)
Memory usage	Overflow queues and pipeline registers	Pipeline registers only	Pipeline registers only

Table 5.1 summarizes which types of communication are supported by each network while Table 5.2 compares several features between the networks. The features are very different because each network is customized for a *distinct* type of communication.

Although we originally designed the distributed shift register to resolve dynamic task creation, its offload architecture is suitable for BCP collection and its reload topology is suitable for loading initial data into the FPGA's memory.

Except for the buffers which are required for overflow management, the networks use no additional memory (pipeline registers are primarily used to resolve physically long route paths). This leaves much of the FPGA resources for the processors.

5.9 BCP Central Buffers

Every BCP produced by a processor on the FPGA is reported to software. However, our FPGA only has one PCIe interface, hence this is a point of serialization. The PCIe interface must be time-shared between different threads. Several processors may each produce a BCP for different threads at the same time, therefore we require some buffering on the FPGA.

The BCP central buffers are replicated per thread, hence *each* BCP central buffer is responsible for collecting all BCPs from *one* thread. By grouping BCPs according to their thread in hardware, we avoid the overhead of synchronizing different software threads upon each BCP report.

As discussed in section 5.5.3, every BCP (from all threads) can be collected using the offload network. Once these reach the central global overflow buffer, every BCP is sent to a reload network so that exactly one of the BCP central buffers will extract it based on its thread. This reload pathway was shown in purple in Figure 5.9.

Each software thread has its own buffer (a region in the CPU's memory) where it expects hardware to write the BCPs. For each thread, this buffer contains a list of variables that have been assigned. When we assign or imply a variable, it is added to the end of the list. When we deassign a variable, it is removed from the list. DPLL uses a depth first search, thus we deassign variables in exactly the opposite order in which they were assigned. Conclusively, this buffer acts as a *stack* (it has an access pattern of first in, last out).

A SAT algorithm proves that a SAT problem is satisfiable by assigning all variables without a conflict, thus each software buffer (one per thread) must be at least this large. However, we have insufficient amounts of on-chip FPGA memory to size the BCP

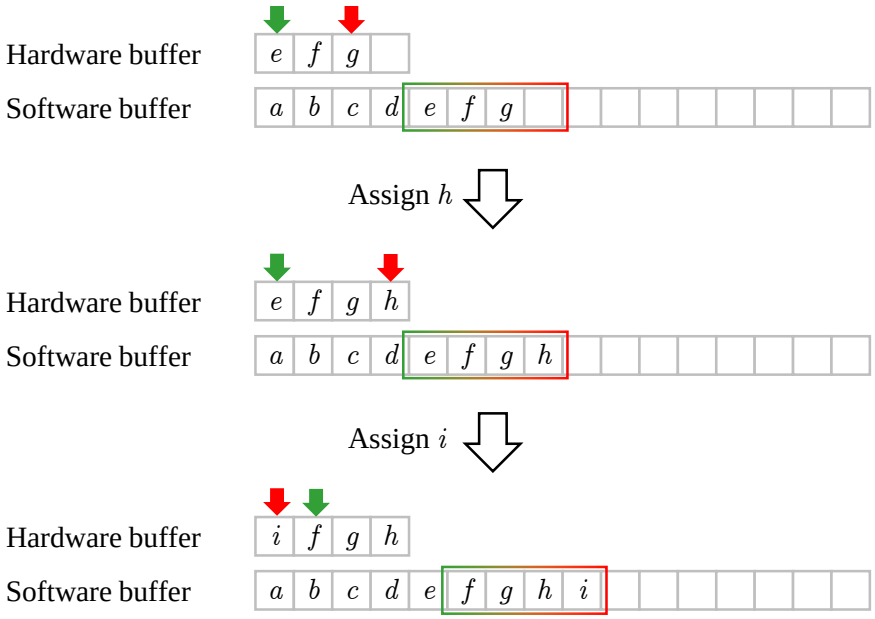


Figure 5.19: As BCPs are produced in hardware (new variables are assigned), older variables are flushed to software before being overwritten in hardware.

central buffers for this worst case. Since the buffer is a stack, we only need immediate access to where we can push or pop an item. Therefore we have implemented the BCP central buffers as a *sliding window* which caches the most recent BCPs.

Figure 5.19 illustrates how newly implied variables are pushed onto the stack. We must keep track both the highest and lowest position since the sliding windows is not always full. If we currently do not have enough variables assigned, we cannot fully populate the sliding window. Also, if all of the variables we need to deassign are currently in hardware, we can avoid using the PCIe interface thereby saving the bandwidth for other threads.

Due to the limited capacity of on-chip memory in hardware, the oldest variables are overwritten during BCP. Obviously these must be flushed to software before being overwritten. If a BCP central buffer fills up (e.g. BCPs are produced faster than

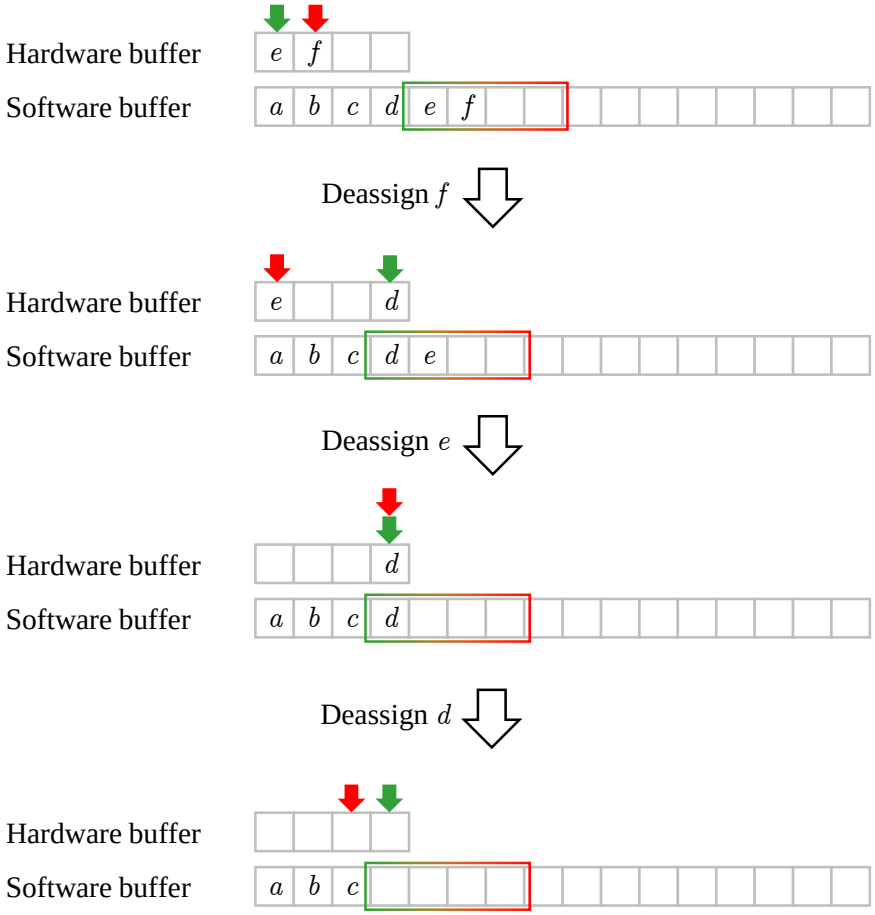


Figure 5.20: As variables are deassigned (following a conflict), we may need to retrieve older variables from software's memory.

PCIe can send them to software), we *temporarily* apply backpressure to the offload network. Once BCP is done, we must ensure that all BCPs have been sent to software (regardless of whether they have been overwritten or not).

Following a conflict, software will perform conflict analysis to determine how many variables to deassign. This involves popping them from the stack as shown in Figure 5.20. This may involve retrieving previously overwritten variables from the CPU's memory (such as variable d in this example). This retrieval is done as soon as the sliding windows has space available.

The capacity of the sliding window in hardware should be sufficiently large to hide the communication latency of PCIe. In Figure 5.20, ideally we would like variable d to arrive from software before variables e and f are deassigned. In our implementation, each BCP central buffer can store up to 4096 BCPs, thus thousands of deassignments can be used to hide the external interface latency.

As a clarification, both the hardware sliding window cache and the software buffer contain variables in the form of `base`, `offset`, and `assignment`, which must be translated into the corresponding software variables. Note that `thread` is not stored since each thread has its own buffer.

Each variable assigned in hardware has a corresponding `base` and `offset` collected at the BCP central buffer. To deassign this variable, we create a new link list traversal task with the same `thread`, `base` and `offset`, but with `assignment = 00` (the code for unassigned). These tasks can be delivered to the processors via the random access network or the reload network (or both). Our implementation only uses the random access network since it has significantly higher bandwidth.

Conclusively, hardware manages the entire deassignment process. All that software needs to indicate is how many variables to deassign.

The BCP central buffers use an offload network to send data to the PCIe interface, as this is yet another example of a many-to-one topology requiring distributed arbitration. In the reverse direction, the PCIe interface writes data to its own regional overflow buffer so as to reuse the existing communication infrastructure.

As a final note, hardware manages all of the reading and writing to the software buffers. We implemented a direct memory access (DMA) controller on the FPGA to optimize the PCIe interface, which is discussed in section 7.2.1.

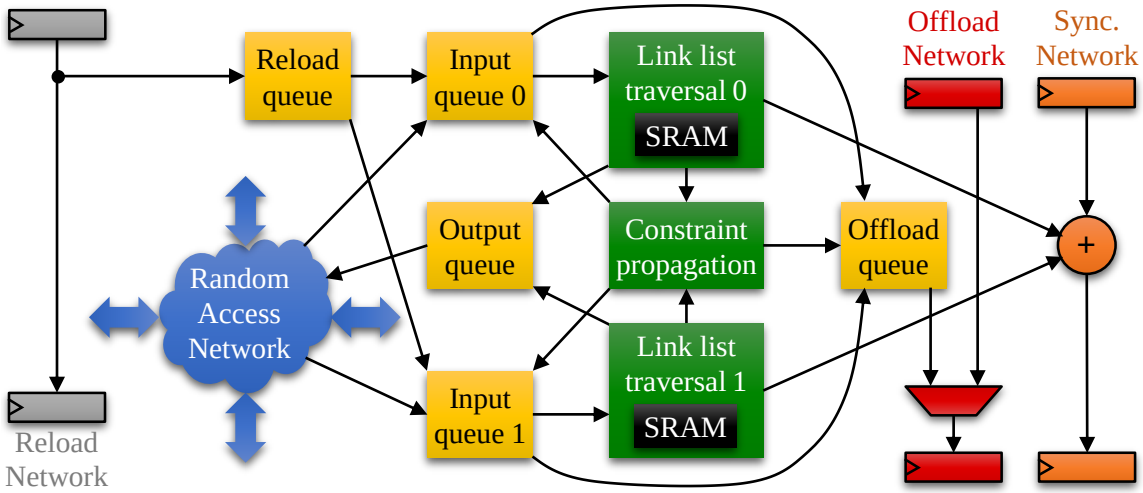


Figure 5.21: This entire figure illustrates the components in *one* processor as well as the connectivity to the networks. The left side primarily supplies data for processing. Processing units are shown in green in the middle. On the right side, data is exported to a centralized place (overflow tasks, BCPs, synchronization).

5.10 Processor Interface

With the establishment of our on-chip communication infrastructure, we can finally present exactly what constitutes one processor. To maximize the utilization of logic resources, the throughput of the components in each processor should be balanced.

Each link list traversal engine can visit one clause every two clock cycles. Four memory accesses are needed (see section 4.5) and we use dual-port embedded memories. On each clause visit, the variable assignments are checked for newly implied variables using combinational logic. To facilitate resource sharing, the constraint propagation logic is time-shared between two link list traversal engines (on odd and even clock cycles, every pair of link list traversal engines are always out-of-phase with each other).

As shown in Figure 5.21, one processor contains two link list traversal engines. Therefore each processor has a total of four execution strands (two per link list traversal engine) and may produce a BCP on every clock cycle.

To transfer a link list traversal task to another engine, the task is sent to the output queue which interfaces to the random access network. If link list traversal engine 0 needs to transfer a task to link list traversal engine 1 in the same processor, this must go through the random access network. The output queue can accept a new value on every clock cycle, hence it is also time-shared.

Queues decouple the timing of computation and communication. Even if the random access network is temporarily unavailable due to backpressure, the link list traversal engines can continue processing. Ultimately if the output queue fills up, it must backpressure both link list traversal engines.

The constraint propagation engine is given the variable assignments from the link list traversal engines. If a newly implied variable is found, two things must happen:

1. This variable must be reported to software. BCP collection is done via the offload network. Arbitration favors items already inside the distributed shift register, hence the need for the offload queue.

2. A new link list traversal task must be created (to visit all clauses that contain this newly implied variable). This is added to the appropriate input queue.

To resolve the overflow problem of dynamic task creation, items from the input queues must first be moved to the offload queue before leaving the processor. After the link list traversal engines have processed some tasks, some space will become available to restore the offloaded items. However, as these are being reloaded, other tasks could be arriving through the random access network. To avoid clogging the networks, arbitration into the input queue always favors the random access network, hence the need for the reload queue. We backpressure the reload network using the fullness of the reload queue instead of the input queues.

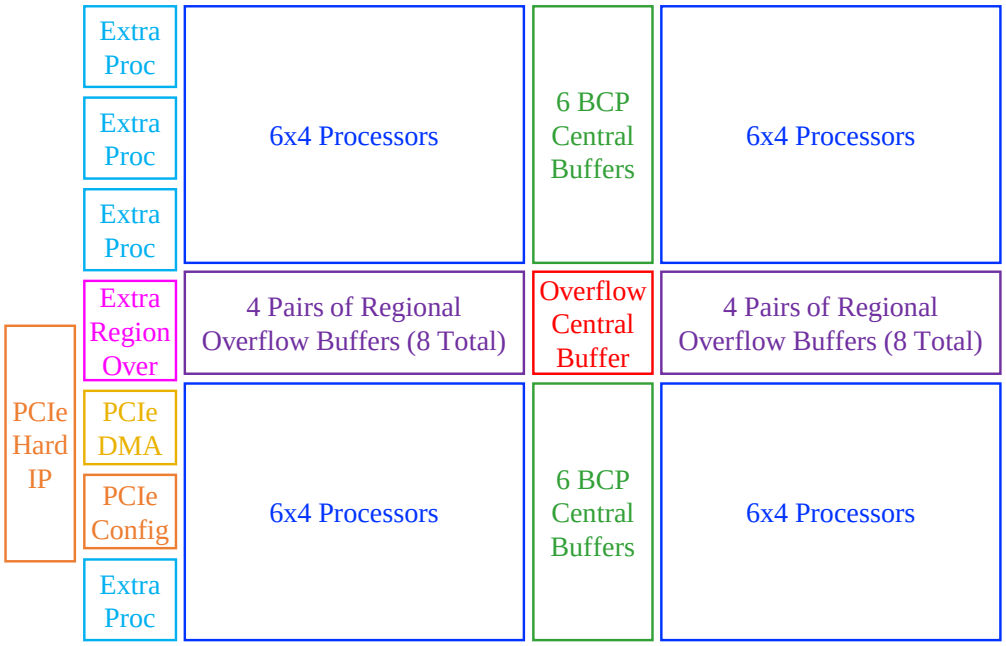


Figure 5.22: An overview of where processing blocks should be physically placed on the FPGA. This is not drawn to scale.

5.11 High-Level Floorplanning

In synchronous digital design, the longest propagation delay between any pair of registers determines the maximum clock frequency. Our experience with FPGA design is that this delay tends to be dominated by routing delay instead of delay through combinational logic. In timing-sensitive paths, routing delay can exceed 80% of the clock period. Conclusively, the *physical placement* of the logic on the FPGA has a tremendous impact on the performance.

We pipelined all of our communication infrastructure with the expectation that processors will be physically spread across the FPGA. However, CAD tools for compiling Verilog code into an FPGA implementation (Quartus 13.1 in our case) are not globally optimal. Many NP-hard problems are involved, thus heuristics are

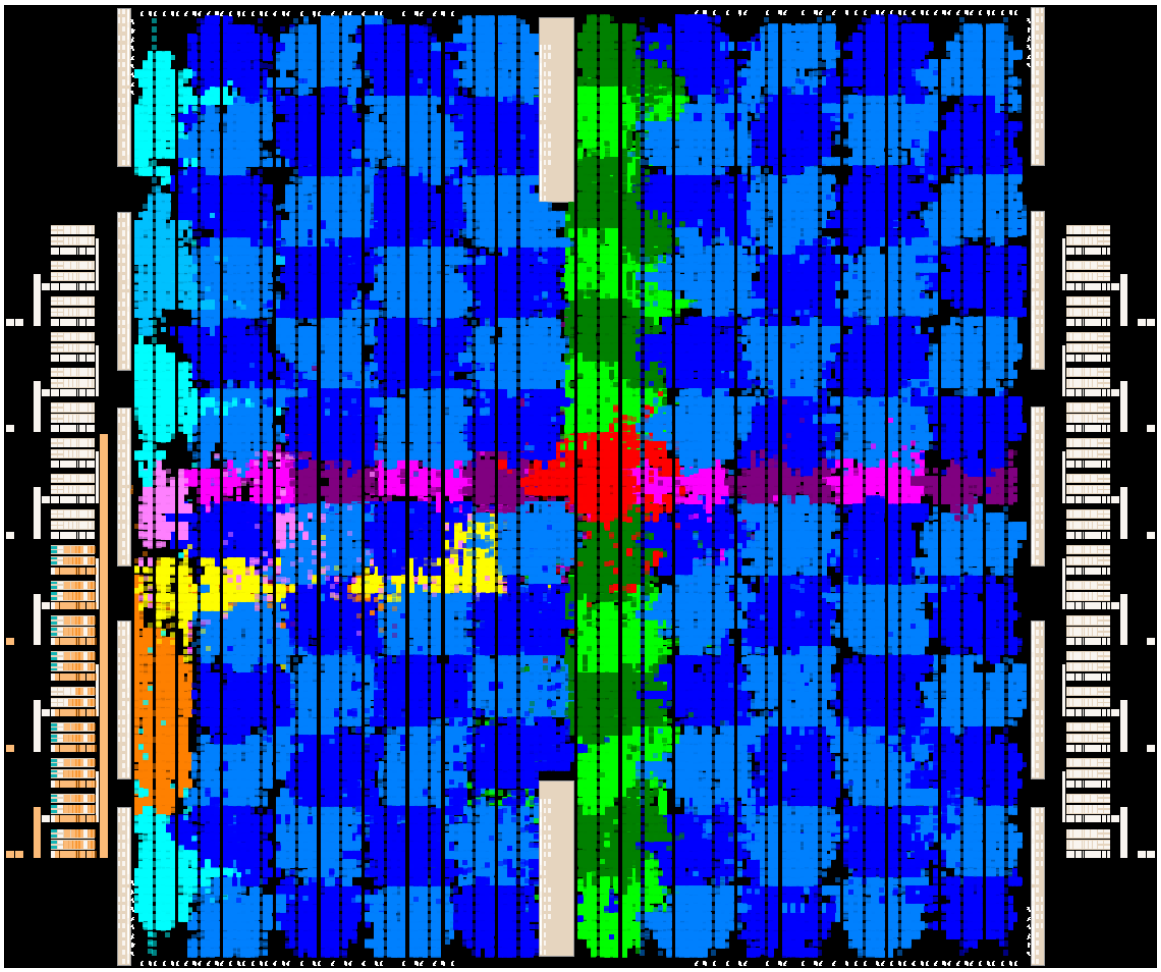


Figure 5.23: The floorplan of our final implementation shows that processing blocks are generally placed the correct region, however the boundaries are left to the discretion of the CAD tool. The color scheme is consistent with Figure 5.22.

needed for practicality reasons. Consequently, even if processors A and B are *logical* neighbors, there is a chance that they may be *physically* placed far apart. This is problematic since we only use one pipeline stage between logical neighbors, for example. Generally as the size of the design increases, the probability and severity of performing a poor placement (or using bad routing) increases, and it is the *single* worst case that determines the maximum clock frequency.

In order to guide the CAD tool, we assigned the physical placement of every embedded memory on the FPGA. Figure 5.22 illustrates an overview of the desired placement, and Figure 5.23 demonstrates that each block is placed in approximately the desired area. Notice that the boundaries are irregular. By allowing some flexibility, the CAD tool can still perform low-level optimizations.

We used the Stratix V DSP Development Kit from Altera [59], which was illustrated in Figure 5.2. The FPGA contains an equivalent of 457,000 logic elements and 345,200 usable registers. We were able to fit 100 processors, of which 96 are arranged in a grid of 12 rows by 8 columns. We squeezed 4 additional processors into the leftover space on the far left side (shown in cyan in Figures 5.22 and 5.23). This leftover area is half the usual width of a processor, hence processors here have twice the height.

Each of the 200 link list traversal engines has 8192 memory locations. Every FPGA has a *pre-defined* ratio of memory capacity versus the amount of logic. These can be balanced by choosing the appropriate memory size per processor *for the given FPGA*.

Each half-column of processors has its own regional overflow buffer (shown in purple). Every pair of regional overflow buffers from the same column will insert to the same place in the global distributed shift register, hence the pairing in Figure 5.22.

With 12 BCP central buffers (shown in green), our hardware supports up to 12 threads. This was chosen to match our 6-core CPU host (hyper-threading produces 12 logical cores). Our memory layout supports fewer threads with no change to hardware.

The PCIe interface is shown in orange, including both hardwired high-speed logic and a calibration controller which is implemented in reconfigurable logic.

Our own PCIe DMA engine is shown in yellow. This also acts as a clock domain crossing between the 250 MHz PCIe interface and all of the remaining logic which is

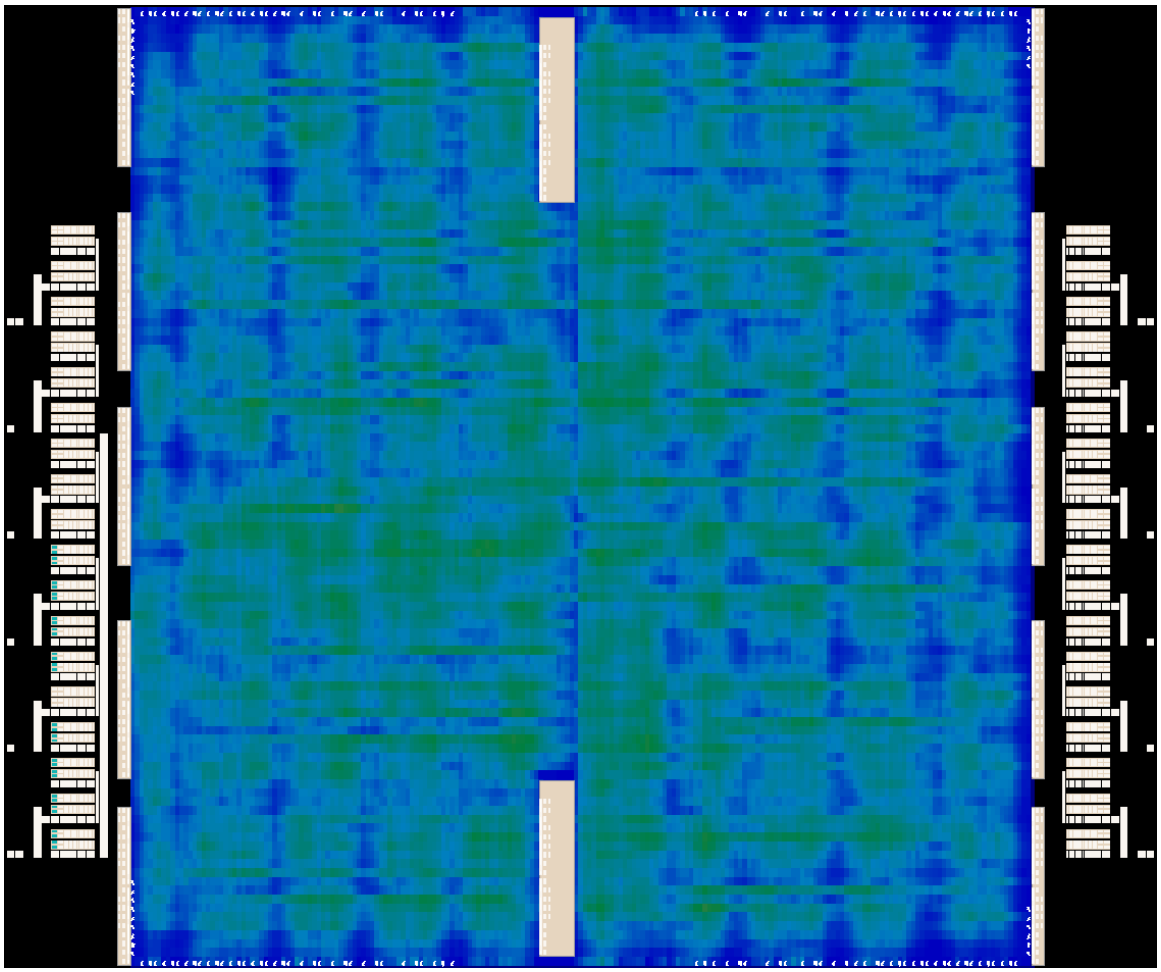


Figure 5.24: The routing usage tends to be concentrated within each processing block. Blue represents low routing usage and green represents high usage. The areas of the highest routing density only use approximately 60% of the routing resources.

clocked at 400 MHz. This high frequency (for a Stratix V FPGA) was achieved with aggressively pre-planned pipelining and high-level physical placements, which reduced the propagation delay due to physically long route paths between registers.

Finally, Figure 5.24 illustrates that the routing tends to be concentrated within each processing block. As shown in section 9.2.1, our communication infrastructure is able to preserve much of the logic resources for the processing itself.

5.12 Summary of Hardware Communication

Throughout this chapter, we have presented the communication infrastructure required to distribute BCP computation over numerous processing engines as well as to interface with software. This included: three separate networks, queues for interfacing with processors, and buffers for overflow tasks and collecting BCPs.

One major complication with BCP is that it is a form of dynamic task creation. BCPs are discovered as clauses are inspected, thus we must dynamically determine when the round of BCP has finished. We efficiently resolved this with a lossy counting-based network. This infrastructure would likely be useful in any application in which the number of distributed tasks to execute is not known in advance or is expensive to determine. In general, some applications may benefit from separating synchronization from data networks.

Since the maximum number of dynamically created tasks was impractically large, we added offload and reload networks. Even in other applications where this bound is practical, it could still be less expensive to use offload and reload networks rather than size every queue for the worst case. In any application that distributes dynamic task creation, it is difficult to ensure that several processors will never "victimize" one processor (by sending it excessive amounts of data), thus some offload and reload infrastructure will very likely be required.

In general, to maximize the computation per transistor, we typically have to trade off the functionality per transistor. By exploiting properties specific to each network, we obtained a collectively more efficient implementation (instead of significantly over designing one network to handle everything). Our random access network was greatly simplified due to the co-existence with the other networks.

Finally, the hardware/software interface is ultimately a point of serialization. In order to efficiently time-share it, buffers on the FPGA must be sufficiently large. In the case of BCP collection, a large sliding window decouples BCPs produced in hardware versus BCPs reported to software. For overflow tasks, our hierarchy uses regional overflow buffers to hide the PCIe latency from the BCP processors. The communication latency overhead inherent to every heterogeneous compute system is typically not trivial, thus architectural features should be designed to mitigate it.

Chapter 6

Advanced Hardware Optimizations

Throughout the previous two chapters, we have established the design of a hardware BCP engine and the interconnect between these engines. Although this is sufficient to specify the hardware aspects of our heterogeneous computing system, we present additional hardware optimizations in this chapter to further extend the practicality of our SAT solver accelerator. In particular, our choice to use SRAM (because of its high-performance in random access) limits our ability to support very large SAT problems. We further exploit lossless compression in order to fully maximize the use of the on-chip memory.

6.1 Enhanced BCP

Many applications of SAT, as surveyed throughout section 2.1.2, involve the verification or optimization of digital systems. In order to verify or optimize any system, the behavior of that system must be encoded into the corresponding SAT problem. Consequently, many SAT problems are composed of *logical constructs*.

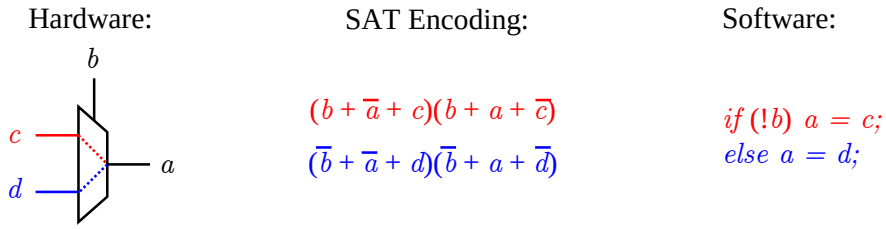


Figure 6.1: A hardware multiplexer and a software if-else assignment share the same SAT encoding.

Logical constructs are implementation independent building blocks for digital systems. For example, a hardware multiplexer and a software assignment made using an if-else statement can use the same encoding in a SAT problem. This is illustrated in Figure 6.1.

If a Boolean SAT problem happens to contain the four clauses listed in Figure 6.1, we can use *lossless compression* to represent all four clauses with one representative clause. This can also be done for other types of logical constructs, such as AND gates and XOR gates (their SAT encodings were illustrated in section 2.2.1). Therefore we also need to provide a "clause type" so that we know whether the representative clause $(a+b+c+d)$ means $a = b$ AND c AND d or whether it means $a = \{c$ if not b, otherwise $d\}$, for example.

This is the basic idea of enhanced BCP. Note that the SAT solver hosted in software is still *fundamentally Boolean* and our hardware still accelerates the Boolean constraint propagation part of the SAT solver. Our system supports non-Boolean clauses primarily to exploit lossless compression.

We have targeted logical constructs because of the high ratio of lossless compression. Each logical construct requires only one clause to represent the equivalent of several Boolean clauses. This is possible because these Boolean clauses are *highly structured*.

Minor changes are required to the hardware memory layout. Assuming the embedded memories have 18 bits per location, instead of storing 9 variable assignments (each 2 bits), we could store 8 variable assignments and a 2-bit clause type, for example. This would allow up to four logical construct types.

It follows that the link list traversal engine needs to send both the variable assignments and the clause type to the BCP check engine. Given the clause type, the BCP check engine first derives all of the underlying Boolean clauses. For example, given:

$$(a + b + c), type = XOR$$

it would derive the following Boolean clauses:

$$(\overline{a} + \overline{b} + \overline{c})(\overline{a} + b + c)(a + \overline{b} + c)(a + b + \overline{c}).$$

We can check for BCP in all four of these clauses in parallel, hence the BCP check engine is still purely combinational logic. Conceptually this is true, however our implementation adds pipeline registers to cut the now longer combinational paths. Nonetheless, the throughput is still one clause (of any type) per clock cycle.

Enhanced BCP offers several advantages:

1. Lossless compression allows us to support larger SAT problems.

2. It may enable faster processing. In the above example, we can visit one XOR clause instead of visiting four separate Boolean clauses. Lossless compression improves the functional usage of memory bandwidth. It is advantageous to improve this at the expense of more logic resources (constraint propagation is now more complex) because BCP is a memory bound problem.

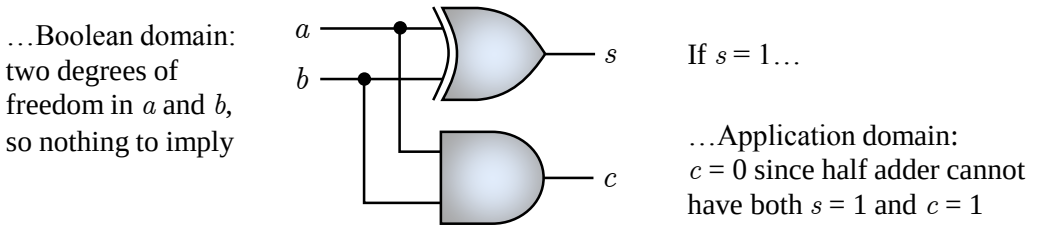


Figure 6.2: Enhanced BCP enables new propagations not immediately visible from the Boolean domain. In a half adder, the sum and carry outputs cannot both be 1, hence $s = 1$ implies $c = 0$ even though nothing is known about a or b.

3. It may reduce traffic in the random access network. For example, instead of variable v occurring in a total of 10 clauses spread over 4 partitions, using enhanced BCP may result in v occurring in 3 clauses spread over 2 partitions. Visiting only 2 different processors instead of 4 reduces network traffic.

4. It enables new propagations which are not immediately visible from the Boolean domain. An example is illustrated in Figure 6.2. For a purely Boolean SAT solver to imply $c = 0$ from $s = 1$, it would first need to encounter the conflict state of $c = 1$ and $s = 1$ in order to *learn* the clause $(\overline{c} + \overline{s})$. Enhanced BCP therefore provides more aggressive pruning in the DPLL tree search.

Table 6.1 summarizes the types of clauses that our implementation supports. Inversion comes for free in SAT by simply exchanging variable v with $\overline{v}$. Using inversions and De Morgan's Law, an AND gate can be transformed into NAND, NOR, and OR, hence we only need to support one of these. We have chosen NOR since it complements Boolean clauses. In Boolean clauses, we look for variables assigned false, and if there is a BCP we imply it as true. This is reversed in partial NOR clauses, where we look for true variables and imply false. A (full) NOR clause uses the propagation rules of both Boolean and partial NOR clauses.

Table 6.1: Summary of all enhanced BCP clause types currently supported in hardware. Partial NOR is equivalent to NOR without the first Boolean clause.

Type	Underlying Boolean clauses
Boolean	$(x_1 + x_2 + x_3 + \ldots + x_n)$
XOR	All 2^{n-1} unique clauses of the form $(x_1 + x_2 + x_3 \ldots + x_n)$, e.g. they contain all n variables, and the number of positive variables in each clause is even
NOR	$(x_1 + x_2 + x_3 + \ldots + x_n)(\overline{x}_1 + \overline{x}_2)(\overline{x}_1 + \overline{x}_3) \ldots (\overline{x}_1 + \overline{x}_n)$
Partial NOR	$(\overline{x}_1 + \overline{x}_2)(\overline{x}_1 + \overline{x}_3) \ldots (\overline{x}_1 + \overline{x}_n)$
Inverted Majority	$(x_1 + x_2 + x_3)(x_1 + x_2 + x_4)(x_1 + x_3 + x_4)$ $(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_4)(\overline{x}_1 + \overline{x}_3 + \overline{x}_4)$
Inverted Half Adder	NOR: $(x_1 + x_2 + x_3 + x_4)(\overline{x}_1 + \overline{x}_2)(\overline{x}_1 + \overline{x}_3)(\overline{x}_1 + \overline{x}_3)$ XOR: $(\overline{x}_2 + \overline{x}_3 + \overline{x}_4)(x_2 + x_3 + \overline{x}_4)(x_2 + \overline{x}_3 + x_4)(\overline{x}_2 + x_3 + x_4)$
2:1 Multiplexer	$(x_2 + \overline{x}_1 + x_3)(x_2 + x_1 + \overline{x}_3)(\overline{x}_2 + \overline{x}_1 + x_4)(\overline{x}_2 + x_1 + \overline{x}_4)$
Double Packed Size 2 Booleans	$(x_1 + x_2)(x_3 + x_4)$ This is created specifically for the optimization in section 6.2.5.

By inverting the inputs of a half adder, the AND gate becomes a NOR (the XOR remains the same). This allows us to reuse some logic in the constraint propagation engine. The majority function occurs as the carry logic in a full adder, and by inverting the inputs we can share some logic with the inverted half adder. The multiplexer propagation logic operates independently.

To facilitate new propagations that are not immediately visible from the Boolean domain, notice that the NOR component of the inverted half adder involves all four variables. Due to the symmetry of an XOR gate, the sum output of the half adder performs propagation in exactly the same manner as the inputs. This can also be verified by enumerating every possible variable assignment and checking for propagation equivalence.

One of the key complications of operating on non-Boolean clauses is that some clauses can produce *multiple propagations at once*. This is impossible for Boolean clauses since only the last unassigned variable can be implied.

As an example, if we visit the clause $a = b$ NOR c and we have assigned $a = 1$, this will imply both $b = 0$ and $c = 0$. As discussed in section 5.10, every BCP that is produced must be reported to software via the offload network and a new link list traversal task must be enqueued. Both of these pathways have a *limited bandwidth*, hence we can only imply one of b or c. We arbitrarily choose the first unassigned variable (b in this case), but without extra bandwidth we have no choice but to drop all of the other propagations.

Conceptually, we are not yet done with this $a = b$ NOR c clause. However, it will be the first clause visited in the newly created link list traversal task for variable b. Upon this revisit, we can now imply c. Upon the revisit from c, we would be able to imply another variable, e.g. for NOR clauses with more variables. This revisiting process continues until all of the propagations have been invoked.

There is little challenge to add more sophisticated logical constructs, such as a full adder or a 3:1 multiplexer, however these are more specialized and thus typically occur less frequently in real-life SAT problems. Furthermore, as we increase the number of supported clause types, we would need more bits to identify each clause type, and this requires additional memory.

Finally, a method for extracting logic gates from an arbitrary Boolean SAT problem is provided in [86]. However, during our own experimentation, we figured out how to extract logical constructs using template matching. We support only certain types of logical constructs, and the corresponding templates are shown in Table 6.1.

6.2 Ultra Compact Variable Assignments

As discussed in section 3.3.2, we require the use of SRAM due to the poor random access performance of denser memory. However, this limits our ability to support large SAT problems. Consequently, we presented several key insights throughout section 4.2 in order to compact our hardware memory layout. In this section, we present sophisticated optimizations to further reduce the memory usage. Supporting even larger SAT problems further extends the practicality of our system.

6.2.1 Identification of Unused Memory

Our memory layout from section 4.3 is already very compact. Every clause occupies a contiguous section of memory, and no space is wasted between clauses. The only potentially small amount of wasted space is at the end of a partition. Clauses cannot cross partition boundaries because each link list traversal engine operates on its own memory partition. To put this in perspective, typically each partition has 8192 memory locations yet each clause uses only 10-20 locations.

Except for the end of a partition, every memory location is used to represent the SAT problem. Variable assignments must be replicated per thread (to enable independent DPLL searches), and we only use N memory locations for N threads. If a clause has K variables, only K local links are used. Global links are only used if necessary. Our memory layout was designed to utilize every single memory location. However, further optimization can be obtained by maximizing the usage *within* each memory location.

We pack the assignments of all variables in the clause into one memory location (per thread) to facilitate fast BCP. However, the entire memory location is used,

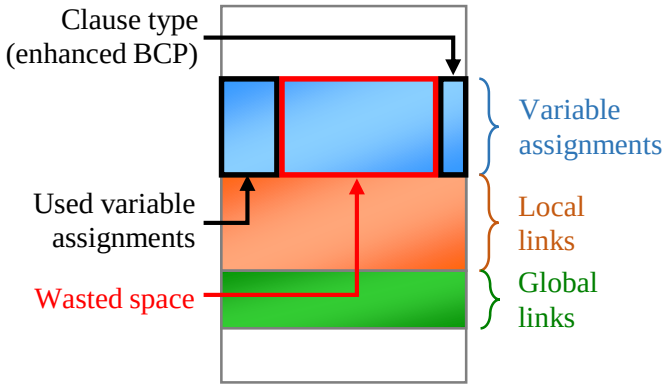


Figure 6.3: If a clause has few variables, much of the space for the variable assignments is unused. The color scheme is consistent with Figure 4.5.

regardless of whether the clause has 2 variables or 8 variables, for example. Figure 6.3 illustrates that this is wasteful for small clauses. Unused variable assignments are tied to false since performing BCP on $(\overline{a} + b)$ produces the same result as doing it on $(\overline{a} + b + 0 + \ldots + 0)$. This is still the case for enhanced BCP from section 6.1.

SAT problems typically contain many clauses with 2, 3, or 4 variables. Furthermore, some of the enhanced BCP clause types (half adder, majority, and 2:1 multiplexer) require exactly four variables. Conclusively, we can improve the memory layout by customizing it for small clauses and for large clauses.

6.2.2 Double Packed Clauses

We must use large clauses in order to recover the wasted variable assignment space, even the if the *given* SAT problem only has small clauses. With the establishment of enhanced BCP, our constraint propagation engine already understands complex non-Boolean constraints. We generalize the "clause type" from enhanced BCP into "clause information" in order to pack two original clauses into one clause in hardware.

For example, suppose we pack $(a + b + c + d)$ and $(e + f + g)$ together as:

$$(a + b + c + d + e + f + g), info = DoublePackedBoolean$$

In the existing design, when a link list traversal engine visits a constraint, it would send the variable assignments and clause type to the constraint propagation engine. Now it also needs to send the `offset`:

- If `offset` is between 0 and 3 inclusive, we are targeting the first original clause. By *masking* the variable assignments of e, f, and g (setting them to false), the constraint propagation engine will think it received a normal size 7 clause.
- If `offset` is between 4 and 6 inclusive, we are targeting the second original clause. In order to reuse the constraint propagation engine, we must first *align* the variable assignments (by shifting them by four offsets). Therefore e, f, g, and false must take the place of a, b, c, and d respectively. As before, the variable assignments of the upper offsets must be tied off to false.
- Our memory layout never allows `offset` to be 7 or higher (for a size 7 clause).

The software that generates the binary values for the hardware memory layout must be updated, however no changes are needed to the SAT solver itself in software. When hardware reports BCPs to software, we translate the hardware address of the local link into a software variable. As specified in section 4.3.1, each variable in a clause always has exactly one local link at the memory location `base` + `offset` (in the orange region of Figure 6.3). Therefore even with double packing, each variable can still be uniquely identified.

We use "clause information" to generalize the enhanced BCP "clause type". This has more degrees of freedom because:

- We still support large clauses, hence we need some way to differentiate one clause with 8 variables versus two clauses with 4 variables each, for example.
- Some clause types have arbitrary size, so if there are two clauses we need to know whether it is packed as $2 + 6$, $3 + 5$, or $4 + 4$, for example.
- If we allow two different clause types to be packed together (to maximize the recovery of wasted space), we must specify each clause type.

More bits are needed to specify clause information, and this is problematic since it requires *extra storage*. For double packed clauses to be worthwhile, this overhead must be smaller than the amount of recovered space from the unused variable assignments.

6.2.3 Creating Space for Extra Clause Information

When we added the clause type for enhanced BCP, we lost one variable assignment to make space for this (assuming only four clause types are supported, as each variable assignment is 2 bits). This was illustrated on the far right side of Figure 6.3.

The clause type was replicated per thread to ensure that *any* thread can access it, since we already need to access the variable assignments for that specific thread anyways. However, more bits are needed as we add packing data to the clause information. Eventually this overhead becomes prohibitive.

With a sufficient number of threads (e.g. 8 threads), the total storage for the clause type over all threads becomes comparable to an entire memory location (e.g. 16 bits). This is shown in Figure 6.4. The key advantage of the illustrated transformation

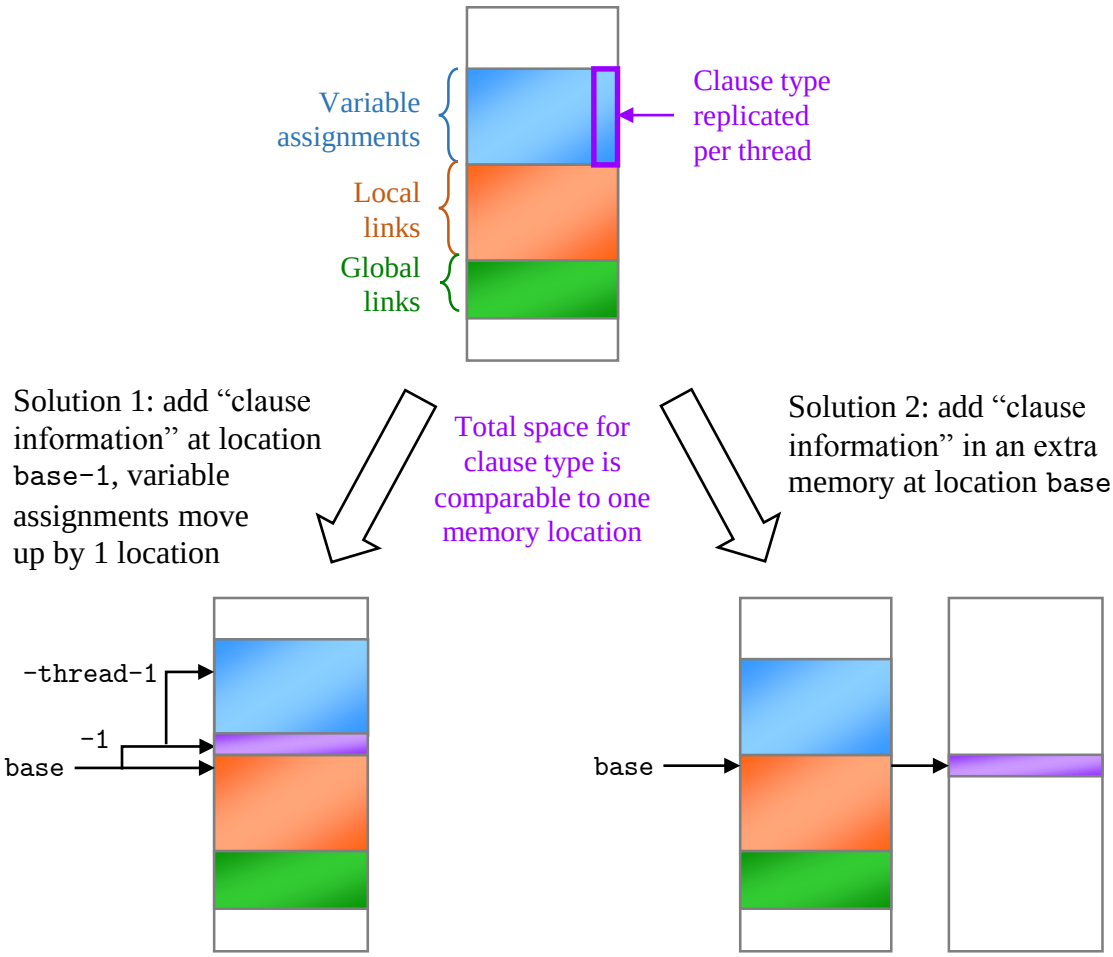


Figure 6.4: The existing memory layout is shown on the top, where the clause type replaces one variable assignment. Over many threads, the total space occupied is comparable to one memory location. Moving from a vertical arrangement to a horizontal arrangement in memory provides more bits of storage space.

is that we obtain the many more bits of storage that we will need for the "clause information". This data is now shared, and thus the cost is amortized over all threads. We also regain space for more variable assignments.

Two solutions are presented. Solution 1 in Figure 6.4 preserves the implicit features of the memory layout. Clause information would be stored in memory location `base-1`. The variable assignments for a given `thread` would now be located at `base-thread-2`.

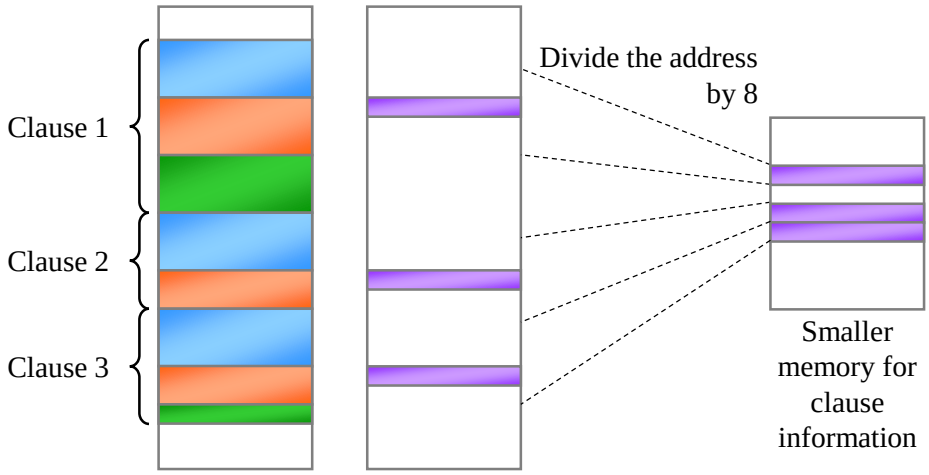


Figure 6.5: Because the clauses have minimum spacing (e.g. no closer than 8 addresses apart), we can compact the clause information memory.

However, one major complication is that by placing the clause information in a different memory location than the variable assignments, we would need an extra memory access. As shown in section 4.5, our link list traversal state machine already fully utilizes the bandwidth of the dual-port embedded memories.

We instead placed the clause information in an additional embedded memory to increase the total bandwidth. However, without any spare bandwidth in the original memory, we must be able to *implicitly* find the clause information. As shown in Solution 2 of Figure 6.4, we can place the clause information at an implicit location, such as `base`. In order to visit a constraint, the link list traversal engine must be given the value of `base`, however this makes the clause information memory very sparse.

To reduce the sparsity of the clause information memory, suppose that the `base` of every clause was at least 8 memory locations away from the `base` of any other clause. If we divide the `base` of every clause by 8, the values would all be *distinct*. This is illustrated in Figure 6.5. In section 6.2.6, we will demonstrate how to enforce this minimum spacing between clauses.

Exploiting the minimum clause spacing allows us to use a significantly smaller memory for the clause information. Also, by using separate memories, the number of bits per memory location can be different, which can further reduce the size.

Using a separate smaller memory for the clause information requires memories with different capacities on the same FPGA. However, this is not expected to be an issue. Each FPGA has a pre-defined ratio of the amount of embedded memory versus the amount of logic resources. To balance the usage of resources, we must choose the appropriate amount of memory per processor. Two scenarios can happen:

1. With relatively more logic than memory on the given FPGA, each clause traversal engine should use only one embedded memory (M20K for Stratix V from Altera, Block RAM for Xilinx). The clause information memory could be implemented in LUT-based memory (MLAB for Altera, Distributed RAM for Xilinx).

2. With relatively more memory, each clause traversal engine should use several embedded memories. If it is too expensive to use LUT-based memory for the clause information, one embedded memory for this can be time-shared between two link list traversal engines (our implementation does this).

6.2.4 Variable Assignment Compression

Having established an *adjustable* amount of space for the clause information, we now seek to compress the variable assignments themselves. Boolean variables can only be assigned true or false, and the DPLL tree search adds the unassigned state. Two bits are required for 3 states, however by grouping k variables together, eventually the number of bits needed to specify 3^k combinations will drop below $2k$. The first occurrence is with 3 variables, as 27 states can be represented on 5 bits.

Stratix V embedded memories are natively 20 bits per location, which cleanly divides 5. Thus we support clauses with 12 variables. For consistency with section 4.2.3 where we chose 18 bits per memory location (wide enough to provide sufficient bandwidth yet narrow enough to exploit lossless compression), our design began on Stratix IV which natively has 18 bits per location. We migrated our implementation to Stratix V before variable assignment compression was introduced.

Other FPGAs may have different memory widths. There are several options:

- Use several *physical* memories to construct one *logical* memory that has 20 bits per location (an example is illustrated in Figure 10.1).
- If the native memory width is 16 bits, we could use 3 groups of 5 bits.
- If the native memory width is 18 bits, we could add one unencoded variable assignment to the above scenario (for clauses with up to 10 variables).

As we traverse a link list, only one variable assignment is updated at each clause. The following steps are involved:

1. Decode: every group of 5 bits read from memory is encoded in base 3. These need to be converted into individual variable assignments each on 2 bits.
2. Update: this is the same as before. The bits at `2*offset+1` and `2*offset` are overwritten by the 2-bit value of `assignment`.
3. Encode: every group of 3 variable assignments needs to be encoded into base 3 before being written back to memory.

No changes are needed to the constraint propagation engine, the link list traversal engines must send it variable assignments from after the update but before the encode.

6.2.5 Elimination of Most Small Clauses

With support for clauses with up to 12 variables, we could pack 6 clauses (each with 2 variables) together with no wasted space, however storing 6 clause types is a large overhead. Instead, we eliminate most of the size two clauses.

XOR and Boolean are the only clause types that we support that may have 2 variables (all other types require at least 3 variables). An XOR with 2 variables means that those two variables are equivalent. During SAT preprocessing, we can relabel one variable as the other and then eliminate this clause.

As shown in the bottom of Table 6.1, two Boolean clauses each with 2 variables can be combined. This results in one "Double Packed Size 2 Booleans" clause, and it is treated like any other enhanced BCP clause with 4 variables.

Only a few size 2 clauses may still exist. If there was an odd number of size 2 Boolean clauses, one would be leftover after double packing. Also, a very small number of size 2 XOR clauses may be intentionally created to facilitate work spreading, as discussed in section 4.4.2.

Finally, size 1 clauses are eliminated with BCP during SAT preprocessing. The remaining size 2 clauses are treated as size 3 with minimal wastage. Therefore only up to 4 clauses can be packed together, which limits the storage for different clause types.

6.2.6 Macro Clauses

Macro clauses are *multiply packed* clauses and they provide the ultimate compaction of variable assignments. By compressing the variable assignments, we support clauses with up to 12 variables, and 12 cleanly divides both 3 and 4. With the elimination of most size 2 clauses, clauses with 3 and 4 variables are typically the most common

Table 6.2: Summary of all of the supported clause packings.

Packing	Allowed clause sizes
$\{12\}$	$\{12\}$, $\{11\}$, $\{10\}$, $\{9\}$, $\{8\}$, $\{7\}$, $\{6\}$, $\{5\}$, $\{4\}$, $\{3\}$
$\{6,6\}$	$\{6,6\}$
$\{5,7\}$	$\{5,7\}$, $\{5,6\}$, $\{5,5\}$
$\{4,8\}$	$\{4,8\}$, $\{4,7\}$, $\{4,6\}$, $\{4,5\}$, $\{4,4\}$
$\{3,9\}$	$\{3,9\}$, $\{3,8\}$, $\{3,7\}$, $\{3,6\}$, $\{3,5\}$, $\{3,4\}$, $\{3,3\}$
$\{3,5,4\}$	$\{3,5,4\}$
$\{4,4,4\}$	$\{4,4,4\}$, $\{4,4,3\}$
$\{3,3,6\}$	$\{3,3,6\}$, $\{3,3,5\}$, $\{3,3,4\}$, $\{3,3,3\}$
$\{3,3,3,3\}$	$\{3,3,3,3\}$

in many SAT problems. With no wasted space, we can pack 4 clauses each with 3 variables, or we can pack 3 clauses each with 4 variables.

Packings of other clause sizes are allowed, such as $\{3,3,6\}$ or $\{4,8\}$. A summary of every supported clause packing is provided in Table 6.2. Given a minimum clause size of 3 (as explained in section 6.2.5), we can pack any group of clauses in which the total number of variables does not exceed 12.

As shown in Table 6.1, we support 8 clause types, so 3 bits are needed to specify the type of each clause. However, four of the clause types (Majority, Half Adder, Multiplexer, and Double Packed Size 2 Booleans) require exactly four variables. Thus if a clause does not have four variables, only 2 bits are needed to specify its type.

Clause information needs to specify the packing (from Table 6.2) as well as the type of each clause. As the number of clauses that are packed together increases, the total space required by all of the clause types increases. To fit this in a fixed amount of space, we would need a smaller codeword to specify the packing, as illustrated in Figure 6.6.

Packing	Prefix	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
{3,3,3,3}	11XXX	prefix		type3		type2		type1		type0	
{4,4,4}	10XXX	prefix		count		type2		type1		type0	
{3,3,6}	011XX	prefix			type2			type1		type0	
{3,5,4}	010XX	prefix			type2			type1		type0	
{4,8}	0011X	prefix				type1			type0		
{3,9}	0010X	prefix				type1				type0	
{6,6}	0001X	prefix					type1			type0	
{5,7}	00001	prefix					type1			type0	
{12}	00000	prefix							type0		

Figure 6.6: Clause information specifies the packing with a prefix code as well as each clause type. The colors in this diagram have no relation to colors elsewhere.

In the Stratix V FPGA that we used, memories can be configured as 10 or 20 bits per location, hence using 11 bits is very wasteful. In order to fit into 10 bits, we used a non-standard encoding of the clause types in the $\{4, 4, 4\}$ packing. Each size 4 clause is supposed to need 3 bits to specify its clause type, however there is some symmetry that can be exploited. We separate the clauses into two groups based on the most significant bit of the clause type (all clauses in one group have type ≤ 3, in the other group they have type ≥ 4). By indicating *how many* of the clauses have type ≥ 4, we only need to specify the 2 least significant bits of each clause type.

Let us illustrate this with an example encoding. Suppose our count is 01 (in binary) and the truncated clause types are 01, 10, and 11. The count indicates that only the first clause type should have its most significant bit set to 1. Therefore the original clause types are 101, 010, and 011.

When the link list traversal engine visits each clause, it sends the updated variable assignments, the `offset`, and the entire clause information to the constraint propagation engine. To reuse the constraint propagation engine, after the link list traversal engine but before the constraint propagation engine, we need the following features:

- Alignment: because of all the different clause packings, we need a barrel shifter to align the variable assignments. For example, if we want to access the third clause in $\{3, 3, 3, 3\}$, we need to shift by 6. Constraint propagation expects a size 3 clause to occupy offsets 0, 1, and 2.

- Masking: upper variable assignments must be tied off to false. For example, after shifting by 6 to access the third clause in $\{3, 3, 3, 3\}$, the variable assignments of the first two clauses disappear, however we still need to mask the fourth clause. The upper 6 variable assignments (that were shifted in) are always false.

- Decoder: we use the prefix code to identify the packing. The `offset` identifies which *individual* clause to use from the packing. Once we know all of this, we can extract the correct individual clause type from clause information as well as control the alignment and masking.

Due to this added complexity, we have added an extra pipeline stage before the constraint propagation engine. Even so, constraint propagation is still *conceptually* combinational logic since a large lookup table could be used to check for BCP given the variable assignments, `offset`, and clause information. The constraint propagation pipeline has a throughput of one input per clock cycle and never applies backpressure.

Finally, to guarantee a minimum clause spacing of 8, since we need at least one thread (at least one memory location is used for variable assignments), at least 7 local links must be used. Notice from Table 6.2 that any clause with up to 6 variables can at least be double packed. The worst case packing for macro clauses is if the given SAT problem only has size 7 clauses. Due to the leftovers from packing, only the last macro clause in each partition may not be fully populated, however the clause spacing depends on the size of the previous clause. Conclusively, the minimum spacing is 8.

6.3 Distributed Encoding of Local Length

The link list traversal of clauses finishes when the incoming assignment matches the existing one, as discussed in section 4.3.2. Therefore if there are L clauses in the link list, we actually perform $L + 1$ visits. If we could indicate that the length of this link list is L, then we could stop traversing after L visits, thereby reducing the computation time by avoiding the revisit.

We only apply this optimization to *short* and *entirely local* link lists because:

1. A length of L requires $\lceil log_2(L) \rceil$ bits to encode ($L = 0$ is not allowed). The storage overhead becomes prohibitively large if L is large. We fall back to the existing mechanism (incoming assignment matching the existing one) for long link lists.

2. As we traverse the link list, we need *state information* to indicate how far along the traversal we currently are. If the link list spans more than one partition, this state information must also be passed through the random access network, thus we must widen the data path and the queues. This is significantly more expensive than keeping the state information entirely localized.

3. From a practical perspective, the improvement from 3 visits to 2 visits is more noticeable than from 100 visits to 99 visits, for example.

Note that a link list of length 1 means that the variable only occurs in one clause in the entire SAT problem. We can therefore assign the variable such that the clause is satisfied. Thus, for all practical purposes, we have $L \geq 2$.

Ideally we would like to distribute the storage of the length over all clauses in the link list. However, one major complication is that the link lists are cyclic, thus the

traversal can start *anywhere* in the link list. This property is required because when a new variable is implied, we must start the link list traversal of that variable at the same clause where it was implied (as explained section 4.3.2).

As an example, suppose we distribute a 3-bit length over several clauses where each clause stores 1 bit of the length. We must somehow indicate that clause X contains the least significant bit, clause Y contains the middle bit, and clause Z contains the most significant bit. This indicator requires 2 bits, so this is no better than storing the original 3-bit value at each clause.

Since state information is required to keep track of how far along the traversal we currently are, we can design the encoding of the length to directly affect these states. This leads to our so-called "three counter system":

- Our state information consists of three counters:
  - Counter A is a modulo 2 counter (1 bit).
  - Counter B is a modulo 3 counter (2 bits).
  - Counter C is a modulo 4 counter (2 bits).
- When we start a link list traversal, all three counters are set to 0.
- At each clause, we use 2 bits to indicate which counter to increment. This is stored as part of the *local link* (in the orange section of Figure 4.5).
- When the three counters return to a value of all zeros, we have finished the link list traversal.

We will illustrate this with several examples. We use the notation $counters = \langle 1, 2, 3 \rangle$ to respectively indicate that counter A is 1, counter B is 2, and counter C is 3.

Example 1. *A link list of length 2 can use the encoding AA.*

When we start the traversal, $counters = \langle 0, 0, 0 \rangle$. Upon visiting the first clause, we see symbol A so we update $counters = \langle 1, 0, 0 \rangle$. Upon visiting the second clause, we see another A so we update $counters = \langle 0, 0, 0 \rangle$. Recall that counter A is a modulo 2 counter, hence $1 + 1 = 2 \equiv 0 \pmod 2$. Since we have returned to the all zeros state, we are done traversing the link list (we do not revisit the first clause).

Example 2. *A link list of length 5 can use the encoding ABABB.*

We start with $counters = \langle 0, 0, 0 \rangle$. At the first clause, we see symbol A so we update $counters = \langle 1, 0, 0 \rangle$. At the second clause, we see symbol B so we update $counters = \langle 1, 1, 0 \rangle$. At the third clause, we see symbol A so we update $counters = \langle 0, 1, 0 \rangle$. Although counter A has returned to zero, not all of the counters have returned to 0. Counter B keeps the link list traversal active. At the fourth clause, we see symbol B, so we update $counters = \langle 0, 2, 0 \rangle$. Finally, at the fifth clause, we see symbol B so we update $counters = \langle 0, 0, 0 \rangle$. Recall that counter B is a modulo 3 counter. Having returned to the all zeros state, the link list traversal is finished.

Since we can start anywhere in the cyclic link list, we may actually observe *any cyclic rotation* of $ABABB$. These include: $BABBA$, $ABBAB$, $BBABA$, and $BABAB$. Each of these encodings will cause the counters to return to the all zeros state after exactly five updates. Only cyclic rotations of the encoding are allowed. An arbitrary permutation, such as $AABBB$, cannot happen because we traverse the link list in a pre-defined order. Actually $AABBB$ is an example of an invalid encoding since we will erroneously return to the all zeros state after only two visits. Any cyclic rotation of an invalid encoding is also invalid (since we can start anywhere).

Example 3. *A link list of length 6 can use the encoding ACCACC.*

The values of *counters* will proceed as follows: $\langle 0,0,0\rangle$, $\langle 1,0,0\rangle$, $\langle 1,1,0\rangle$, $\langle 1,2,0\rangle$, $\langle 0,2,0\rangle$, $\langle 0,3,0\rangle$, $\langle 0,0,0\rangle$. Recall that counter C is a modulo 4 counter.

Several valid encodings for a given length may exist. For example, we could instead use $ACACCC$, which is not a cyclic rotation of $ACCACC$. *Any* valid encoding can be used. Three counters are needed to encode the base cases of length 2, 3, and 4. Using an exhaustive search, it turns out that this is sufficient to generate valid encodings for lengths 2 to 24 inclusive (shown in Table 6.3). Our C++ code runs in about 1 second and also proves that no encoding for length 25 exists. Intuitively, the counters only provide a state space of $2 \times 3 \times 4 = 24$. Our search is summarized below:

- We used an exhaustive depth first search. Given an encoding of length L, we would recursively append A, B, and C (one at a time) and check the validity of the resulting $L+1$ length encoding.
- Search pruning was used. If the current encoding is valid (its traversal results in a return to the all zeros state only at the end), it cannot be extended to a longer valid encoding.

We expect it be to extremely rare that more than 24 occurrences of a variable occur *within the same partition*. In case it does happen, we induce a global link. The random access network already allows routing to self since there are two link list traversal engines within each processor, as discussed in section 5.10.

By transferring the link list traversal task to ourselves after at most 24 steps, this task will resume itself after we run the other tasks which are already waiting in the input queue. This helps to improve the fairness in the quality of service, as different software threads time-share each link list traversal engine.

Table 6.3: Encodings for link lists of length 2 to 24 inclusive.

Length	Encoding
2	AA
3	BBB
4	CCCC
5	ABABB
6	ACACCC
7	BBCBCCC
8	ACACACAC
9	ABABCBCCC
10	BBCBBCBBCC
11	ABABACABCCC
12	ABABCBBCBBCC
13	ABABACABACACC
14	ABABACABBCBBCC
15	ABABCCACCCBACCC
16	ABABACABABACBBCC
17	ABABACBABBCBBCBBC
18	ABABACABABACABABCC
19	ABABACABABCBABBCBBC
20	ABABACABABACABABACAC
21	ABABACABABACABBCBABBC
22	ABABACABBCBBABBCBBABBC
23	ABABACABABACABABACBABBC
24	ABABACABABACBBABBCBBABBC

Finally, this encoding also needs to be compatible with link lists that are not entirely contained within one partition. These link lists will need *global links*.

Two bits were used to specify which counter to increment, but we only used 3 out of the 4 possible cases. The fourth case had been reserved to indicate that we currently have a global link. If a link list uses a global link (because it spans multiple partitions or it is too long), we must use the existing mechanism of the incoming assignment matching the existing one to detect the completion of the link list traversal.

6.4 Summary of Hardware Optimizations

In this chapter, we have presented techniques to both compact the memory as well as attempt to improve the utilization of memory. Our choice to use SRAM (due to its high-performance on random access) limits our ability to support large SAT problems in hardware. To improve the practicality of our system, we have traded off more computation for less memory.

In general, this tradeoff is often desirable in systems with limited memory. For example, embedded computing typically uses bit masking. In our case, constraint propagation became more complex to support enhanced BCP clause types and also due to the alignment introduced by clause packing. In section 9.7.2, we examine the effect of enhanced BCP and macro clauses on the amount of FPGA memory used by real-life SAT problems.

Finally, using a very small amount of extra memory for a significant decrease in the amount of computation can be advantageous. This was demonstrated by our distributed encoding of the length for localized link lists. If the extra amount of memory is sufficiently small, no extra cost may be incurred since FPGAs typically contain hardwired embedded memories with quantized bit widths. For example, given a memory with 18 bits per location, if the application only needs 16 bits, two leftover bits are freely available to encode extra information which may assist the processing.

Chapter 7

Software SAT Solver Integration

With BCP accelerated in highly optimized hardware, the software performance of the remainder of the SAT solver is no longer insignificant. Software integration is far from trivial, especially to support hardware optimizations such as enhanced BCP, which software must perform conflict analysis on. In this chapter, we also discuss our PCIe interface, the integration of learnt clauses into hardware BCP, and memory minimization optimizations.

7.1 System Flow

Our SAT solver accelerator uses heterogeneous computing to encompass the different strengths of FPGAs and CPUs. Hardware and software can communicate using *any* protocol. We used PCI Express, which is further discussed in section 7.2. The flow of our entire system is illustrated in Figure 7.1.

The preprocessor can apply any transformation to a SAT problem so long as it does not change the satisfiability of the given problem. Clauses with only one variable

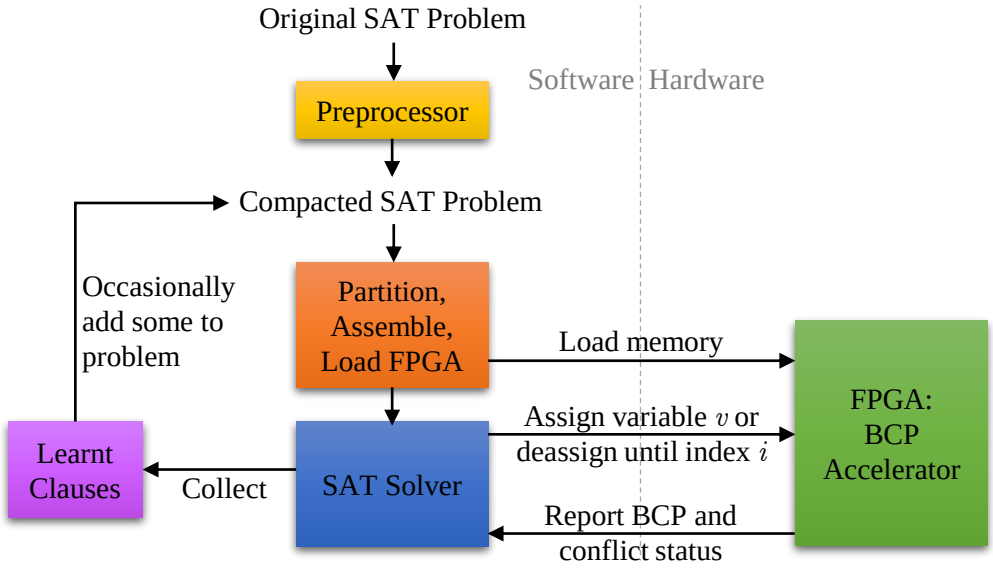


Figure 7.1: The flow of our entire SAT solver system.

are propagated (and thus eliminated). Our hardware only supports clauses with up to 12 variables, hence large clauses must be split. This was demonstrated in section 2.6, which also discusses other preprocessing techniques.

Since we support enhanced BCP, gate extraction (from a Boolean SAT problem) is also performed in the preprocessor. For some SAT applications such as hardware verification, ideally we should be given the logic gates directly as our input (instead of the application encoding them to Boolean clauses only for us to then extract it).

Software preprocessors may enlarge the SAT problem (by adding learnt clauses) to try to make it easier to solve. Conversely, our preprocessor typically favors compaction since we want to ensure that a SAT problem will actually fit in the FPGA's on-chip memory. Our preprocessor is discussed in more detail in section 8.2.

Our hardware BCP accelerator consists of several link list traversal engines which each operate on their own section of memory. We must therefore partition the SAT problem so that we can distribute some clauses to each hardware engine.

Our partitioning algorithm is presented in section 8.1. Once a partition is assigned to each clause, our assembler (step 2 in orange in Figure 7.1) generates the binary values which conform to our hardware memory layout, which was specified in section 4.3. In section 7.2, we describe how this is loaded onto the FPGA.

The multithreaded software SAT solver offloads only the BCP computation into hardware. Software must therefore choose the next variable to assign, perform conflict analysis, and manage the learnt clauses database.

Each conflict produces a learnt clause, and these can be shared between threads. Clauses in hardware are connected by cyclic link lists, so attaching a new clause is straightforward. However, each hardware clause is shared by all threads, and since BCP involves dynamic task creation, the only way to guarantee that no thread is accessing a clause is to stop all threads (in software). When we add a clause, we need the current variable assignments from all threads. If any thread performs BCP after the variable assignments are captured for the new clause and before the new clause is actually added in hardware, we may *desynchronize* the variable assignments.

We avoid this problem by stopping all software threads when we add learnt clauses into hardware memory. To amortize this overhead, we only do this *occasionally*. We discuss this update as well as learnt clause database management in section 7.5.

Upon adding learnt clauses to the SAT problem, we perform a new partitioning, which means we also need to run the assembler and load the FPGA memory again. For ease of software synchronization, we stop the SAT solver during this update. Our results show that this overhead is minimal, so even if we let the SAT solver continue during this update (except during the memory load) or if we parallelized the partitioner and assembler, there is little that can actually be gained.

7.2 Hardware/Software PCI Express Interface

Hardware and software may communicate using *any* protocol. Among the options available on both our FPGA board and CPU motherboard, we chose PCIe since it provides sufficiently high bandwidth for our application. Although not tested, our results could potentially be improved by instead using a CPU-based protocol, such as HyperTransport [84], which typically has lower latency than PCIe.

7.2.1 Hardware/Software Interactions

Software uses 32-bit PCIe register writes to send commands to hardware. As shown in Figure 7.1, software may instruct the following:

1. Request the FPGA to start loading initial memory data (load the SAT problem into hardware memory).

2. Assign variable v (specified as `thread`, `base`, `offset`, and `assignment` from section 4.3, so we must translate the variable before sending it to hardware).

3. Deassign until index i (the deassignment itself is entirely managed by hardware, as demonstrated in section 5.9).

In section 5.5.4, we described how initial memory data could be loaded onto the FPGA by "tricking" the global overflow buffer into thinking it had previously offloaded some data. The global overflow buffer is empty when we start, so it will ask for data back thereby allowing us to bring data from the CPU's memory into the FPGA.

The loading of memory data via PCIe is outlined in Figure 7.2. Although software starts the entire process, it is actually the FPGA that manages the data transfer.

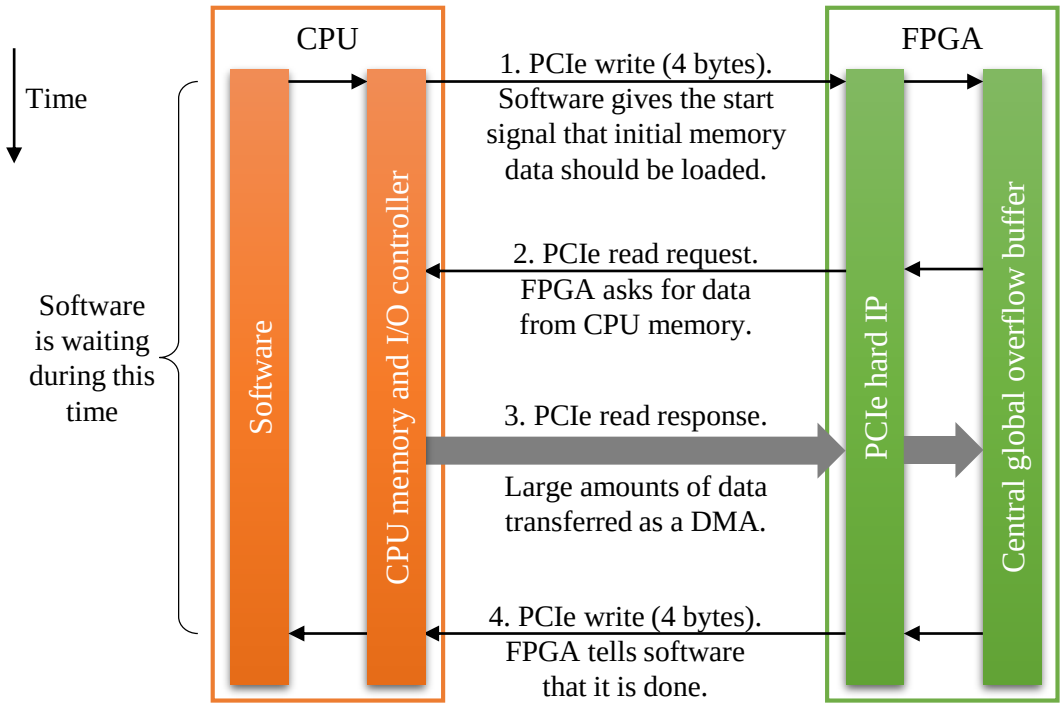


Figure 7.2: The process of loading initial memory data into the FPGA's memory via PCIe. The color scheme is consistent with Figure 7.1.

In steps 2 and 3 in Figure 7.2, the FPGA interacts with the CPU memory and I/O controller *without any involvement from software.*

In Figure 7.3, we illustrate one round of hardware BCP. Software starts the process by sending a 4-byte PCIe write to indicate which specific variable to assign. As hardware produces BCPs, these are written into the CPU's memory with no involvement from software. Finally, the FPGA sends a synchronizing write to software to indicate that it is done.

We have implemented a direct memory access (DMA) controller on the FPGA to maximize the bandwidth available over PCIe. PCIe operates with data packets. Each packet has a header field which consumes bandwidth, thus leaving less PCIe bandwidth for the application. It is therefore better to send one large packet with large

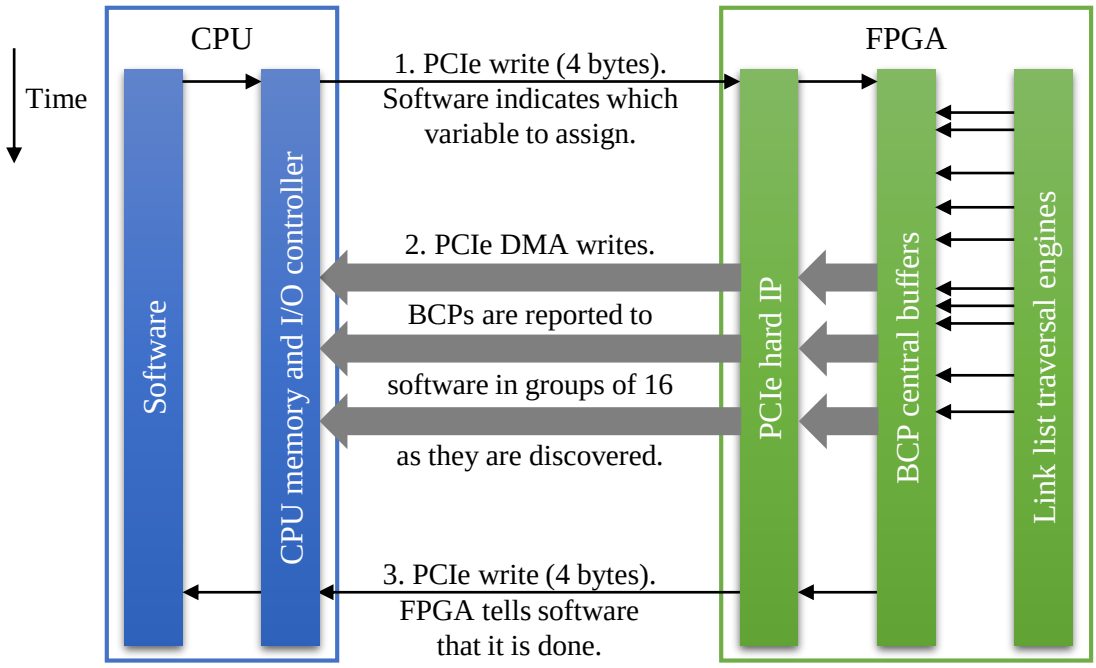


Figure 7.3: Hardware BCP starts with software indicating which variable to assign. As BCPs in hardware are discovered, they are directly written into the CPU's memory via PCIe (using DMA). The color scheme is consistent with Figure 7.1.

amounts of data rather than several small packets. This is what our DMA enables. Notice that in Figure 7.3, individual BCPs are produced by each link list traversal engine, and these are collected at one BCP central buffer (every BCP illustrated in Figure 7.3 belongs to the same thread). Grouping several BCPs together results in a more efficient use of PCIe bandwidth. Other threads also need PCIe bandwidth as well, hence each thread needs to be efficient.

Although not illustrated, deassignment operates in a similar manner. Software starts the process with a PCIe write to indicate how many variables to deassign. As hardware sweeps backwards over the contents of the BCP central buffers, older variable assignments may need to be retrieved from the CPU's memory (as shown in Figure 5.20). The FPGA controls this retrieval using steps 2 and 3 from Figure 7.2.

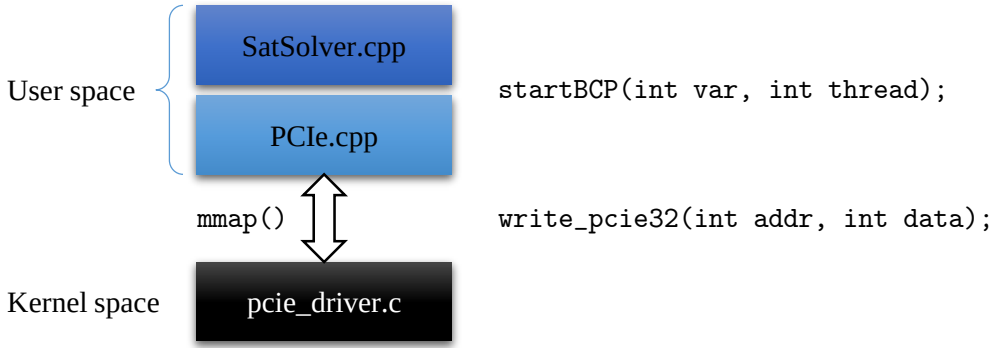


Figure 7.4: PCIe access is passed through the hierarchy of our software. The function `startBCP()` is called by SatSolver.cpp and is implemented in PCIe.cpp.

7.2.2 Software Interfacing

To gain access to PCIe in software, we wrote our own Linux [87] driver. The following DMA buffers (in the CPU's memory) are allocated when the driver is loaded:

- Each thread has its own BCP buffer where hardware can directly write every BCP that it discovers.
- The global overflow buffer (in hardware) offloads into the CPU's memory upon excessive dynamic task creation. Software also borrows this same buffer to load initial memory data (to load an arbitrary SAT problem into the FPGA).
- Hardware writes into a synchronization buffer to indicate to software that it is done (the last step in Figures 7.2 and 7.3). Software polls from here (interrupts have a context switching overhead, also our hardware is relatively fast).

The driver operates in *kernel space* in order to access PCIe (32-bit reads and writes) and the DMA buffers. PCIe reads from software were included for debug purposes. To bring this access into *user space* (where software applications reside), we use the `mmap()` function. Our software hierarchy is summarized in Figure 7.4.

7.2.3 Same Thread Hardware/Software Concurrency

Multithreading facilitates heterogeneous concurrency, and in this section we examine concurrency *within* the same thread. Unfortunately, each software thread must wait for its own thread to finish hardware BCP. One of the following will occur:

- No conflict was found. Software must be informed which variables were implied in hardware so that software can update its own variable assignments. This ensures that the next variable to assign is actually currently unassigned.
- A conflict was found. Software needs the dependencies of each implied variable (reported by hardware BCP) in order to perform conflict analysis.

Nonetheless, there is one situation where we can overlap computation in hardware and software on the same thread. Immediately after conflict analysis in software:

1. Software tells hardware how many variables to deassign (hardware performs the deassignment itself, see section 5.9). Software will then immediately send the next variable to assign. Hardware buffers this until it finishes the deassignment, thus avoiding a second communication latency with software.
2. *While hardware is running*, software deassigns its own variable assignments.

When software finishes its deassignment, it waits for BCP to finish on the FPGA. It is possible that before software could finish, hardware may have already finished its deassignment and then started and finished the next round of BCP. In this case, software will not spend any time waiting for hardware, thus masking the hardware/software communication latency. The FPGA only uses fast SRAM whereas software can be delayed by CPU cache misses (CPU memory is shared by all threads).

7.3 Conflict Analysis for Enhanced BCP

Every time a conflict is found during the DPLL tree search, conflict analysis is used to identify the root cause. This facilitates non-chronological backtracking and also produces a learnt clause, as shown in section 2.5.2.1. In this section, we discuss how to extend conflict analysis to support the non-Boolean clauses of enhanced BCP.

7.3.1 A Review of Boolean Conflict Analysis

In order to perform a root cause analysis, during BCP we must keep track of which specific clause was used to imply each variable. This will be sufficient to later derive the dependencies. For example, suppose variable b was implied due to the clause:

$$(a + b + c + d + e).$$

BCP only implies a variable if it is the last unassigned variable in the clause, thus variables a, c, d and e must have been *already assigned* by the time we implied b.

A conflict essentially means we made a bad decision in the DPLL tree search. Using the example in Figure 7.5, if we assign $x_1 = 1$ and $x_2 = 0$, then we should not have assigned $x_3 = 0$. The decision variables specify a valid cause of the conflict, however the purpose of conflict analysis is to identify a *simpler cause*.

Conflict analysis starts at the conflict and sweeps backwards (using the dependencies of each variable) in order to find the root cause. Every variable that was implied has a corresponding clause to indicate the dependencies of that variable. We also need to know the order in which variables were implied, as discussed below. The conflict analysis as illustrated in Figure 7.5 proceeds as follows:

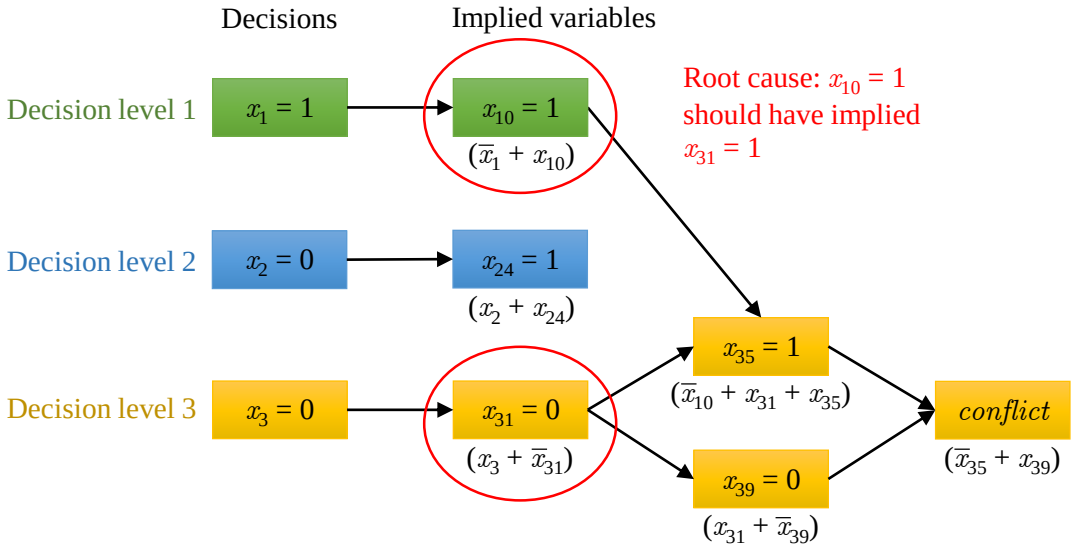


Figure 7.5: Conflict analysis sweeps backwards from the conflict to identify the root cause. The clause that was used to imply each variable is shown below that variable.

1. We start at the conflict clause, which is a clause that could not be satisfied due to the way that other clauses have implied their variables. In this example, both variables x_{35} and x_{39} together are responsible for causing the conflict.

2. Next we "visit" x_{35} to determine which variables are responsible for implying it, since x_{35} was part of what lead to the conflict. Since x_{35} was implied from the clause $(\overline{x}_{10} + x_{31} + x_{35})$, we know that variables x_{10} and x_{31} are responsible for implying x_{35} (both are needed). So far in our backtracking, we know that x_{10}, x_{31} and x_{39} are responsible for the conflict.

3. Conflict analysis sweeps backwards in *exactly the reverse order* in which variables were assigned (the reason is discussed later). Among the variables we could visit (x_{10}, x_{31} and x_{39}), from Figure 7.5 we can see that x_{39} was the last one of these to be assigned. Upon visiting x_{39} which was implied with $(x_{31} + \overline{x}_{39})$, we deduce that x_{31} is solely responsible for implying x_{39}.

4. The next variable to visit is x_{31}. If we only consider variables assigned on the current decision level (decision level 3 in our example), x_{31} is solely responsible for the conflict. Conflict analysis finishes when the backwards graph traversal converges on a single variable in the current decision level.

When we finish, x_{10} has not been visited. In general, conflict analysis will not visit any variable that was assigned on a previous decision level. Since there was no conflict when x_{10} was assigned, there was nothing wrong with deciding to assign $x_1 = 1$.

In general, conflict analysis identifies the root cause due to:

- A *single* variable on the current decision level which is solely responsible for leading to the conflict (x_{31} in our example), and
- Some (possibly zero) variables on previous decision levels which *contribute* to the conflict, but these cannot actually *produce* the conflict without that one variable on the current decision (or else we would have had a conflict earlier).

To conclude the analysis of Figure 7.5, variables x_{10} and x_{31} are *entirely* responsible for the conflict (highlighted in red). The same conflict can be induced by only assigning $x_{10} = 1$ and $x_{31} = 0$ (all other variables unassigned). Thus, we can produce the learnt clause $(\overline{x}_{10} + x_{31})$. The DPLL tree search will then non-chronologically backtrack to decision level 1, where this new learnt clause uses $x_{10} = 1$ to imply $x_{31} = 1$.

Conflict analysis must sweep backwards in exactly the reverse order in which variables were assigned to ensure that we *do not bypass the point of convergence*. For example, at step 3 above, if we had instead visited x_{31}, we would infer that x_3 was responsible for implying it. Since we visited x_{31} instead of x_{39} (x_{39} is not yet visited), we therefore cannot conclude that we have converged, yet we have already backtracked past the point of convergence (by adding x_3 to the list of variables still to visit).

7.3.2 Integration of Hardware BCP into Conflict Analysis

As a starting point, we first discuss how our hardware BCP integrates with software conflict analysis assuming that only Boolean clauses are used (no enhanced BCP).

Hardware BCP operates with `thread`, `base`, `offset`, and `assignment`, as specified in section 4.3. An implied variable in hardware is reported to software by providing both the `base` and the `offset`. Given `base`, software identifies which specific clause was used, and `offset` indicates which specific variable in that clause was implied.

Translation still works properly with macro clauses, which are multiply packed clauses (section 6.2.6). Now `base` identifies which macro clause was used (which *group* of original clauses), and `offset` identifies which original clause within that group was used as well as which variable was implied. For example, an `offset` of 3 in the macro clause:

$$\{\ (a+b+c),\ (d+e+f+g+h),\ (i+j+k+l)\ \}$$

indicates that variable d from the second original clause was implied.

Due to our translation process, hardware is already providing a sufficient amount of information to perform conflict analysis with Boolean clauses.

7.3.3 Enhanced BCP Conflict Analysis

With non-Boolean clauses, we can no longer implicitly determine the dependencies from the clause used to imply a variable. This is because the rules of constraint propagation have changed. For example, suppose that we have a NOR clause where $x = y$ NOR z. If we implied $x = 0$, it could have been due to $y = 1$ or due to $z = 1$. Given only the clause, it is impossible to uniquely determine the dependency of x.

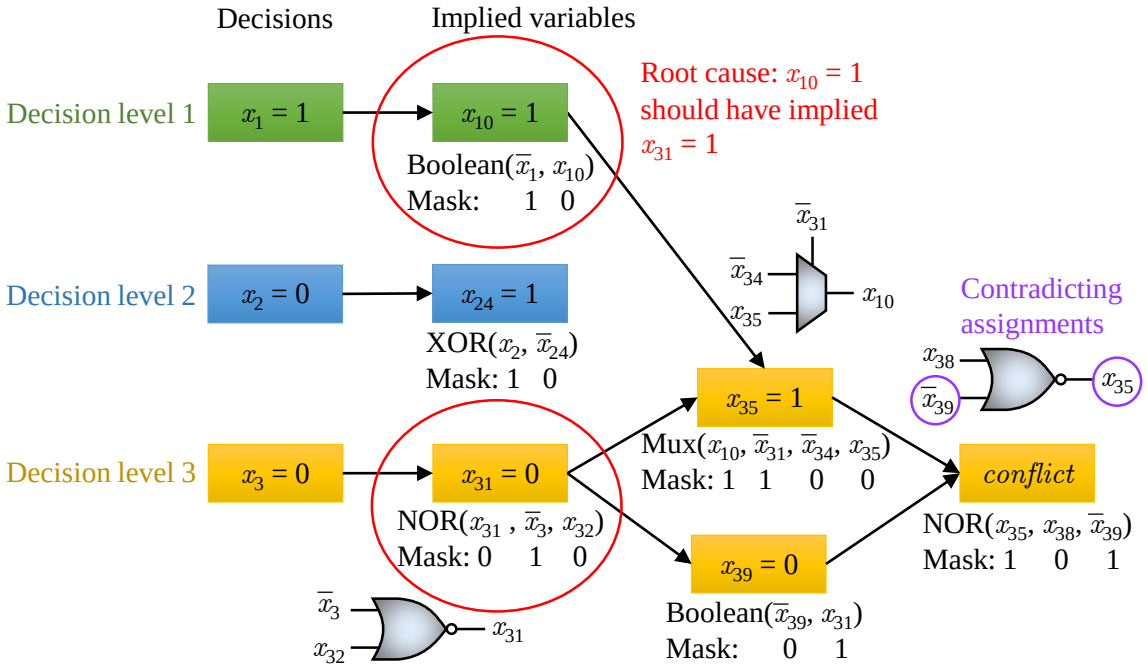


Figure 7.6: Different clauses are used to produce the same conflict as in Figure 7.5. The clause and the binary mask that were used to imply each variable is shown below the variable. Illustrations of the logic gates are provided to clarify our notation.

Hardware must now explicitly indicate the dependencies for every implied variable. We use a binary mask. Suppose $z = 1$ implied $x = 0$ in the clause $x = y$ NOR z. The binary mask would be 001, corresponding to variables $\langle x, y, z \rangle$ respectively.

In general, bit n of the mask indicates whether the variable at `offset` n participated in the constraint propagation. Since our hardware supports clauses with up to 12 variables, each implied variable requires a 12-bit mask.

Recall from section 6.1 that in order to apply BCP to an enhanced BCP clause, the hardware constraint propagation engine derives all of the underlying Boolean constraints and applies BCP to each of them in parallel. Therefore the binary mask provides a means for hardware to indicate which specific underlying Boolean clause was used when it implied a variable.

As an example, in Figure 7.6 we have used enhanced BCP clauses to produce the same conflict as in Figure 7.5. The backtracking dependency analysis is identical:

1. We start at the conflict clause NOR($x_{35}, x_{38}, \overline{x}_{39}$). As illustrated in Figure 7.6, we cannot assign both an input and the output of a NOR gate as true. The binary mask indicates that x_{38} does not contribute to the conflict. As consistent with the analysis of Figure 7.5, variables x_{35} and x_{39} are responsible for causing the conflict.

2. Next we visit x_{35}. As illustrated in Figure 7.6, if the output of the multiplexer is 1 and the select is 1, then the lower input must be a 1. The binary mask indicates that only variables x_{10} and x_{31} are responsible for implying x_{35}. So far x_{10}, x_{31} and x_{39} are responsible for the conflict (consistent with Figure 7.5).

3. Continuing the backtrack in the reverse order that variables were assigned, next we visit x_{39}. We used the same Boolean clause $(\overline{x}_{39} + x_{31})$ as in Figure 7.5. Note that the order of the variables in a Boolean clause does not matter, however in general it does matter for enhanced BCP (as demonstrated by the multiplexer).

4. Finally we visit x_{31}. We are finished since the backtracking has converged, hence conflict analysis does not need to check the dependencies of x_{31}.

Although conflict analysis is not concerned, x_{31} was implied since if we assign any input of a NOR gate to 1, the output must be 0. As before, we produce the same learnt clause $(\overline{x}_{10} + x_{31})$ and DPLL will non-chronologically backtrack to the same place. Learnt clauses implicitly have a clause type of Boolean, since the SAT solver is still fundamentally Boolean and enhanced BCP is a form of lossless compression.

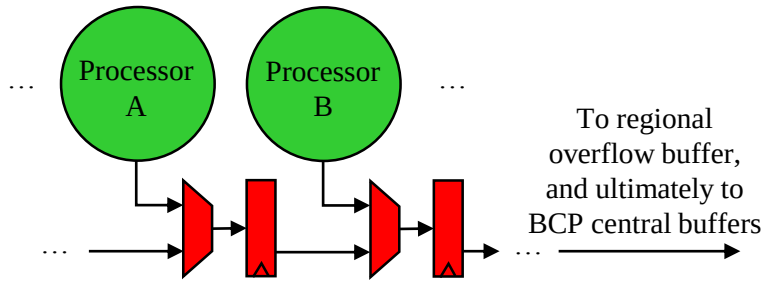


Figure 7.7: Hardware processors report BCPs via the offload network.

7.4 The Hardware BCP Reordering Problem

Recall from section 5.5.3 that the hardware link list traversal engines report BCPs via the offload network. Arbitration favors items already inside of the distributed shift register. In Figure 7.7, if processor A continuously produces BCPs (on every clock cycle), it could starve processor B from offloading. Consequently, processor B could imply variable b before processor A implies variable a, yet a could arrive at the same BCP central buffer before b (assuming a and b belong to the same thread).

Reordering is rare, but it does occur. Every processor upstream of processor B could contribute to the starvation of B. In each processor, every offloaded item shares the same queue. If processor B produced many BCPs before implying b yet processor A was idle before implying a, this increases the chance of a reordering. The same problem also occurs when regional overflow buffers insert into the global shift registers.

Reordering between threads does not matter, however the reordering of variables belonging to the same thread can cause conflict analysis to fail. As explained in section 7.3.1, conflict analysis must sweep backwards in exactly the opposite order that variables were implied to ensure that it does not bypass the convergence point. With variables in an arbitrary order, backtracking could continue indefinitely (potentially visiting every variable) and therefore fail to identify the root cause of the conflict.

It is possible to resolve this reordering problem entirely in software. Since conflict analysis needs the dependencies of each implied variable, we can reuse these to ensure a proper ordering. For example, suppose that variable b was implied due to:

$$\begin{aligned} Clause: &\quad (a + b + c + d + e) \\ Mask: &\quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \end{aligned}$$

Thus variable b can only be assigned *after* all of a, c, d and e are assigned in software. As software sweeps through all of the BCPs that hardware has reported, suppose by the time it processes variable b that only variables a and d have been assigned so far. The mask indicates that c must have been assigned to imply b, so we can infer c had been delayed in hardware due to backpressure (and likewise for e).

Variable b is waiting on variable c. Therefore, later on when we assign c, we should revisit b to see whether all of the dependencies of b have been assigned (thus allowing b to be assigned). To facilitate this, every variable in software has a "waiting list". In our example, we add b to the waiting list of c.

When we later assign variable c, because b was in the waiting list of c, we know that we are closer to assigning all of the dependencies of b. If e had since been assigned, then b is no longer waiting on anyone, so now we can assign it. Otherwise e is not yet assigned, so we transfer b from the waiting list of c to the waiting list of e (eventually when e is assigned, we can then assign b).

In general, several other variables could have been waiting on variable c, in which case the waiting list of c would have contained multiple items. When we assign c, we empty the entire waiting list of c. Each variable in this list is either ready to be assigned or is transferred to the waiting list of another variable.

7.4.1 Motivation for Timestamp-Based Conflict Analysis

Notice that computation is required just to *verify* that the variables happened to be in the correct order. This happens for *every* variable. The checking of dependencies is similar to the computation involved in conflict analysis. However, conflict analysis only runs when there actually is a conflict and it only visits the variables that participate in the conflict. Conclusively, reordering has added a significant computational overhead which did not exist in purely software SAT solvers (BCP in software).

Unfortunately it is difficult to absolutely prevent reordering in hardware. Reordering fundamentally arises due to backpressure when BCPs are produced faster than they can be collected. BCP involves random access and dynamic task creation, hence the *rate* at which BCPs are produced will fluctuate over time. A short burst of numerous BCPs compensates for idle periods at other times, thus for performance reasons we do not wish to limit the number of BCPs per clock cycle. To ensure we collect BCPs in the correct order, we could:

1. Timestamp every BCP (record which clock cycle it was produced on), and
2. Collect the BCP with the smallest timestamp on each clock cycle.

The second point is problematic since processors are physically spread across the entire FPGA. A globalized decision involves long route paths and extremely complex arbitration. For physical reasons, we have favored the use of distributed arbitration in which the order could have been preserved in each localized area, but it is difficult to enforce this at the scale of the entire FPGA.

Although it is inefficient for hardware to *sort* the BCPs, hardware can still provide "helper information" to make facilitate a simpler reordering in software. By sharing

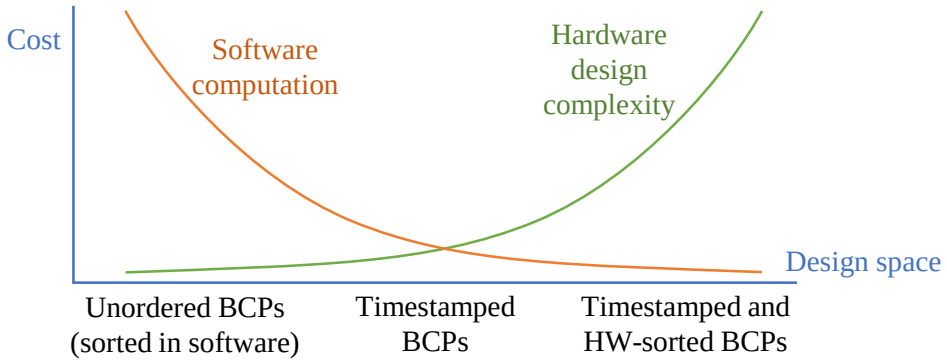


Figure 7.8: Pushing the complexity of the reordering problem into entirely software (sorted in software using dependency analysis) or entirely hardware (guaranteed prevention of reordering) is costly.

the cost between hardware and software, the total complexity can be reduced. This is illustrated in Figure 7.8.

The *timestamping* of BCPs in hardware refers to the recording of the clock cycle number that each BCP was produced on. It is significantly faster for software to sort each variable based on *one integer* rather than several dependencies.

7.4.2 Adding Timestamps to Hardware BCP

At each hardware processor, we use a 31-bit counter to record the time at which each BCP was produced. Since we also store this with the newly implied variable, we must widen the entire data path to software. This includes:

- The offload queue in each processor (see section 5.10),
- Distributed shift registers in each offload network and regional overflow buffers,
- Global distributed shift registers and BCP central buffers, and
- Both the offload to PCIe and reload from PCIe paths.

Each processor has its own local timestamp counter, and these are synchronized by coming out of reset on the same clock cycle. Each BCP central buffer (one per thread) also has its own local timestamp counter. Every time hardware starts a round of BCP, the BCP central buffer for that thread captures the timestamp. This is used to offset the timestamps in order to avoid counter wrap-around. For example:

- Suppose we used an 8-bit timestamp counter.
- Suppose that software starts a round of BCP at time 240.
- Suppose that hardware produces BCPs at times 245, 253, 257, and 260. Because an 8-bit counter was used, we will actually receive timestamps of 245, 253, 1, and 4, respectively.
- By subtracting the starting time (modulo 256), we can report timestamps of 5, 13, 17, and 20 to software.

The timestamp method requires that each round of BCP does not run for a longer duration than supported by the counter. We use 31 bits, which at 400 MHz provides about 5 seconds. Hardware cannot fit a problem with 1 million variables, hence we require a BCP rate of at least 0.2 million BCP/s. Single-threaded software produces about 5 million BCP/s (and we are faster). In practice, the chance of our system failing from this is extremely small (arguably too small to experimentally measure).

A round of hardware BCP must finish before software can choose the next variable to assign. This is completely synchronized, thus timestamps from different decision levels are independent of each other, so each level starts with a timestamp of 0.

However, after each conflict, we backtrack to a previous decision level and imply a variable *partway through that decision level*. One example of this is illustrated through

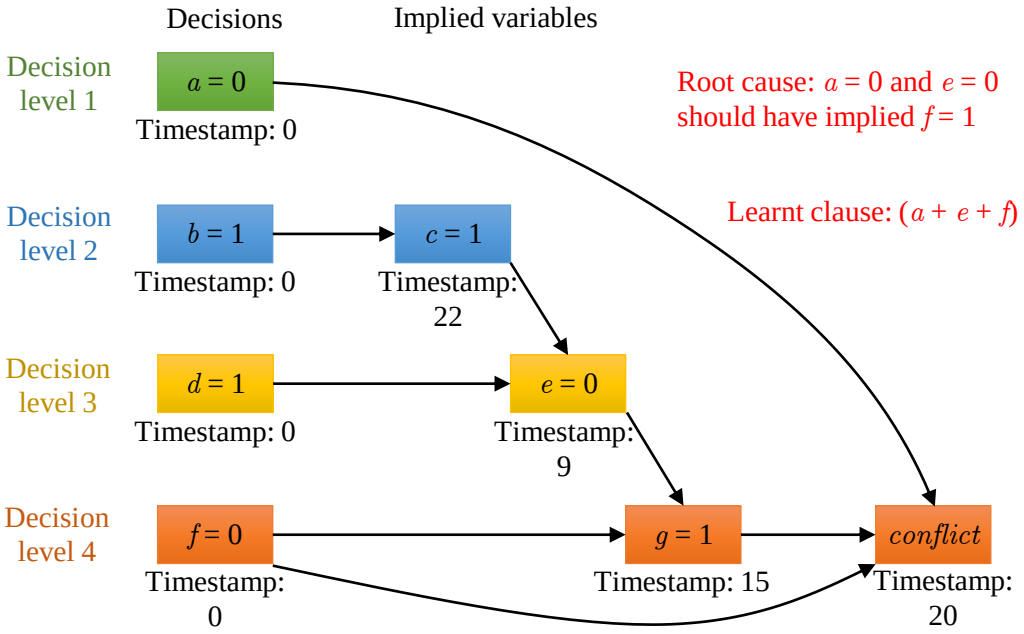


Figure 7.9: Timestamps on different decision levels are independent of each other. This conflict creates a learnt clause, which is then used in Figure 7.10.

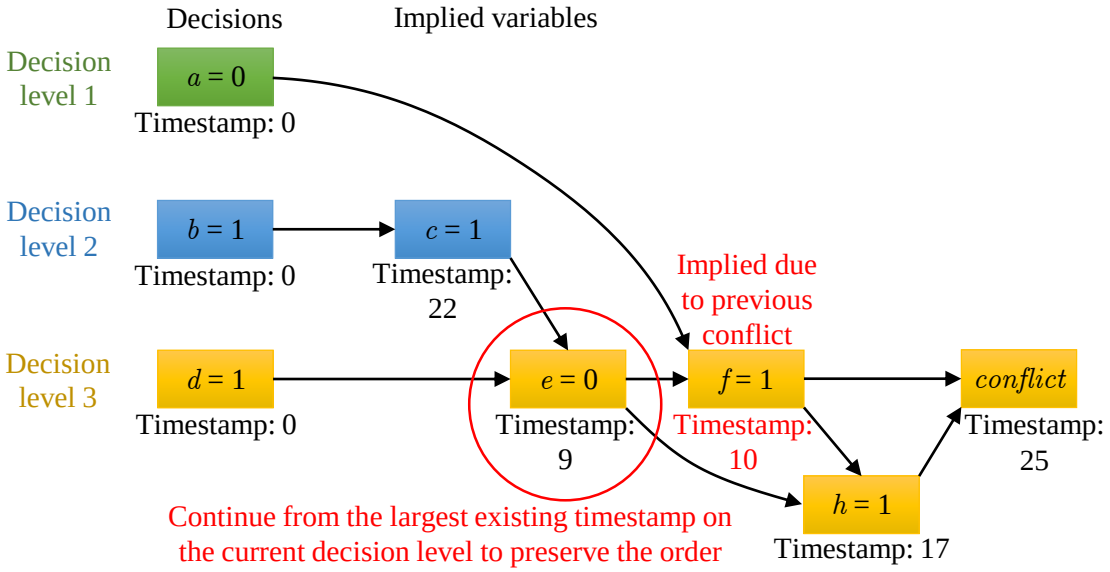


Figure 7.10: Following the conflict from Figure 7.9, we use that learnt clause to imply f. We need to bias the starting timestamp of f in order to ensure the correct ordering for the conflict analysis that will follow as a result of the new conflict.

Figures 7.9 and 7.10. In order to preserve the ordering of timestamps, we need to *bias* the starting timestamp of when $f = 1$ was assigned in Figure 7.10. The timestamp of f should be larger than any timestamp on the same decision level. This information is passed to the BCP central buffer in hardware via a 32-bit PCIe write from software. Each BCP central buffer performs both a bias as well as an offset (demonstrated in the 8-bit counter wrap-around example above).

Finally, we need 33 bits to report each BCP to software excluding the timestamp. Since software operates on pre-defined integer sizes, such as 32 bits or 64 bits, the timestamp simply consumed the remaining space, hence the choice of 31 bits.

7.4.3 Dynamic Ordering During Conflict Analysis

It is computationally wasteful to *always* sort the variables according to the order in which they were assigned. We have restructured conflict analysis to dynamically derive the ordering only as needed.

Recall from section 7.3.1 that we only visit variables on the same decision level as the conflict. Variables on a previous level become part of the learnt clause, thus we do not need to inspect their dependencies and we can store them separately.

For each clause visited by conflict analysis, we use the binary mask to determine which variables in the current clause are the dependencies. For each dependency:

- If it is on the same decision level as the conflict, then this variable is added to the set `VariablesToVisit`.
- Otherwise, it is added to the set `LearntClauseVariables`.

Initially both sets are empty. The next clause to visit is determined as follows:

- When conflict analysis starts, the first clause to visit is the conflict clause.
- Conflict analysis finishes when `VariablesToVisit` contains only 1 item.
- Otherwise, we choose the last variable that was assigned in `VariablesToVisit`. This will be the variable with the largest timestamp.

As an example, the conflict in Figure 7.9 is analyzed as follows:

1. The conflict clause indicates that variables a, f, g caused the conflict. Thus we initialize `VariablesToVisit` $= \{f, g\}$ and `LearntClauseVariables` $= \{a\}$.
2. Next we visit g which has a larger timestamp than f. Variable g depends on f and e. Having visited g, we now have `VariablesToVisit` $= \{f\}$. Also we need to update `LearntClauseVariables` $= \{a, e\}$.

Conflict analysis is now finished since `VariablesToVisit` contains only one variable. The learnt clause is $(a + e + f)$. We non-chronologically backtrack to when only $a = 0$ and $e = 0$ were assigned and imply $f = 1$, as shown in Figure 7.10.

Among the variables to visit from the current decision level, we always select the one with the largest timestamp (which is not necessarily the last one that software received). This is efficiently implemented using a heap data structure.

7.5 Management of Learnt Clauses

Learnt clauses are a byproduct of conflict analysis. Upon identifying the root cause of each conflict, adding a learnt clause prevents this same conflict from happening again (e.g. after the SAT solver restarts, see section 2.5.2.2). For example, if conflict analysis concludes that assigning only $a = 0$, $b = 1$ and $c = 0$ will lead to a conflict,

then we learn the clause $(a + \overline{b} + c)$. Previously when only $a = 0$ and $b = 1$ were assigned, we should have implied $c = 1$.

One important property is that adding a learnt clause to a SAT problem does not change the satisfiability of the problem. We can therefore *choose* whether or not to add each learnt clause, and we can also later remove a learnt clause. Learnt clauses can be preserved even if the SAT solver restarts.

Adding learnt clauses is beneficial because it prevents the revisiting of an earlier conflict, thus augmenting the pruning capabilities of DPLL. However, adding an excessive number of learnt clauses slows down BCP, since for each variable assigned we must inspect more clauses. To balance these competing factors, typically as "more useful" learnt clauses are discovered, they replace "less useful" existing learnt clauses. In general, different SAT solvers have different policies for managing the learnt clause database. Determining the best management of learnt clauses is considered an open research problem [33] and it is not the focus of this thesis.

7.5.1 Adding Learnt Clauses into Hardware Memory

As discussed in section 3.5.1, we can typically find a solution faster by racing several different SAT solvers against each other on the same SAT problem (different solvers are better for different problems). Our implementation uses this strategy.

Each software thread runs a different SAT solver on the same SAT problem. This enables the sharing of clauses in the FPGA's memory over all threads, which greatly reduces the memory usage (compared to not sharing). This enables our system to support larger SAT problems. In software, all threads share the clauses and the translation arrays (to convert between hardware addresses and software variables).

Learnt clauses can be shared by all threads since they do not change the satisfiability of the problem. BCP is computed on the FPGA, so we must update hardware's memory in order to apply BCP to any newly learnt clause. However, as discussed in section 7.1, this update requires the appropriate synchronization. During the memory update, we *must* stop every thread. If BCP is active on any SAT solver, it could desynchronize the variable assignments (as explained in section 7.1).

To amortize the cost of stopping all SAT solvers to add learnt clauses into hardware memory, this update is only performed occasionally. It proceeds as follows:

1. After issuing a stop command, we wait for all software threads to stop. We do not interrupt hardware BCP (typically this finishes quickly anyways).

2. We store original clauses and learnt clauses separately. The learnt clauses from all threads are collected (added to the software database of every learnt clause ever seen). Duplicates are removed.

3. Unit learnt clauses (each containing only one variable) are propagated and thus subsequently eliminated. We must also check for inter-thread conflicts, for example thread 2 indicates $v = 0$ whereas thread 5 indicates $v = 1$.

4. Based on the number of partitions that were previously used (and knowing the capacity of hardware's memory), we estimate how many learnt clauses can be added such that the updated SAT problem will still fit in hardware's memory. An underestimation is used to ensure a fit.

5. Starting from the original clauses in the SAT problem, we add learnt clauses until every learnt clause is added or we meet the quota (estimated in step 4). We start with size 2 learnt clauses, then size 3 learnt clauses, and so on.

6. We run the partitioner, assembler (to generate the binary values for the hardware memory), and finally reload the FPGA memory.

In our current implementation, hardware only accepts learnt clauses if they contain at most 6 variables. We impose this restriction for several reasons:

- This helps to limit the number of added learnt clauses so that BCP does not significantly slow down.
- By ignoring large learnt clauses, software only needs to store smaller and fewer learnt clauses. This saves memory and improves cache usage.
- Our informal experimentation has shown that most of the BCPs produced by learnt clauses are from small learnt clauses, typically 80-90% are from learnt clauses with no more than 6 variables.

From step 4 of the update procedure, our learnt clause management policy is to keep adding learnt clauses until the problem barely fits in hardware. We do this because our results indicate that hardware BCP is sufficiently fast (the remainder of the SAT solver in software is the computation bottleneck). Beware that this may not have been the case if learnt clauses could have had more variables.

We estimate the "usefulness" of each learnt clause based on the number of variables it contains. Intuitively, it is easier to cause a BCP if a clause contains fewer variables. As indicated by step 5 of the update procedure, we add all of the size 2 learnt clauses before adding any size 3 learnt clause. As the SAT solver runs and continues to produce new size 2 learnt clauses, due to the limited amount of hardware memory, the new size 2 learnt clauses can *displace* some size 3 learnt clauses. Likewise, size 3 learnt clauses can displace size 4 learnt clauses, and so on.

As emphasized above, the optimization of the learnt clause database management is an open research problem and beyond the scope of the thesis. There is no *fundamental* limitation to our design that would prevent us from adding larger learnt clauses.

Finally, for ease of implementation, we induce a restart on each SAT solver thread when we update hardware's memory. In order to maintain the completeness of DPLL, the length of the search before the next restart must grow in an unbounded way. We also apply this to the scheduling of our hardware memory update, which operates on longer intervals (typically several restarts occur before each memory update).

7.5.2 Maintaining Learnt Clauses for Conflict Analysis

Large learnt clauses are not added into our hardware BCP accelerator, thus it would appear that software can discard them. However, learnt clauses must be preserved in order for conflict analysis to operate correctly. In particular, if a learnt clause implies a variable that later participates in a conflict, we need that learnt clause to specify the dependencies of that variable during the backtracking process.

One example of this is illustrated through Figures 7.11 and 7.12. The first conflict (Figure 7.11) only involves original clauses. We learn the clause $(a + e + f)$ and backtrack to decision level 3, where $a = 0$ and $e = 0$ are still assigned. Thus we imply $f = 1$ on decision level 3 and then check for any variables that can now be implied. When we encounter the second conflict (Figure 7.12), backtracking first visits variable h, then f, and finally e. When we get to f, the learnt clause $(a + e + f)$ specifies that a and e are responsible for implying f. If this learnt clause had been discarded, backtracking would not have been able to continue properly. Once f is deassigned, if desired we can now safely delete the learnt clause $(a + e + f)$.

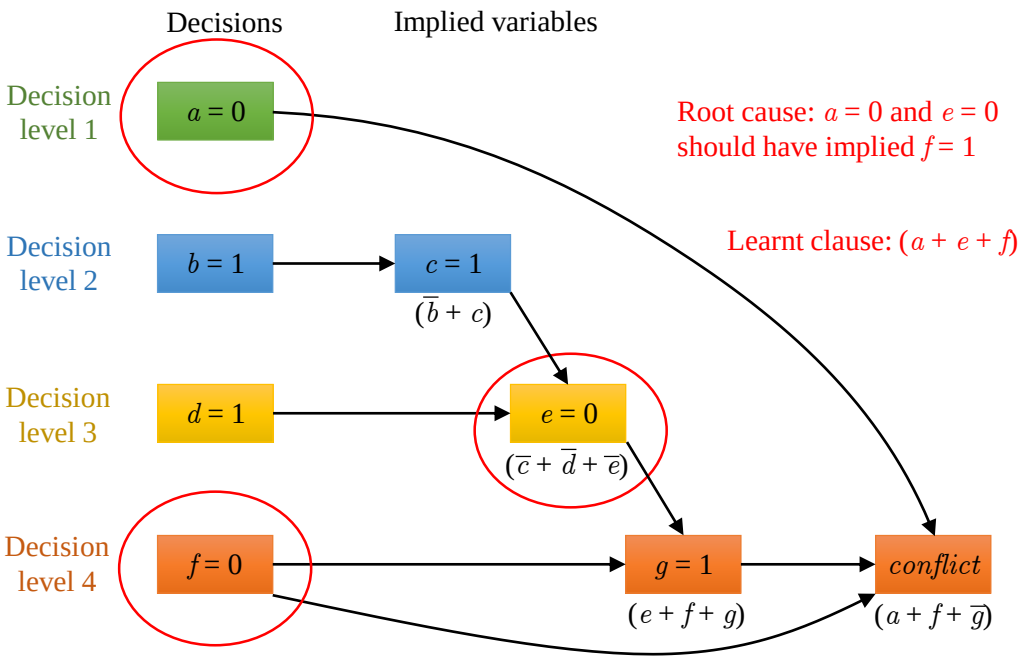


Figure 7.11: This conflict creates a learnt clause, which is then used in Figure 7.12.

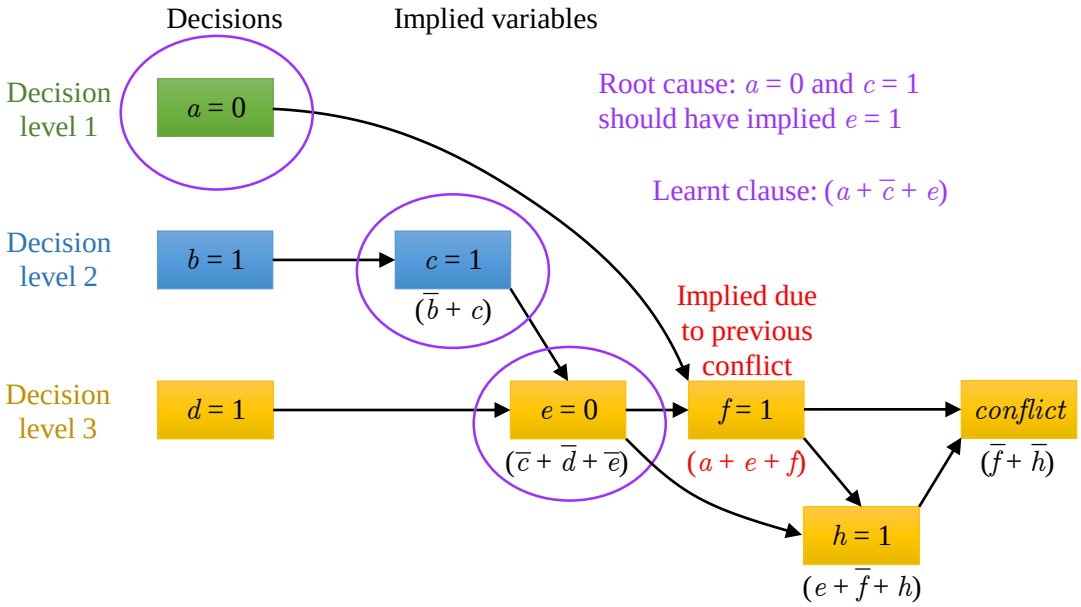


Figure 7.12: Following the conflict from Figure 7.11, we use that learnt clause to imply f. Upon encountering a second conflict, that learnt clause is needed in order to know the dependencies of f during backtracking.

7.6 Software Memory Minimization

Software multithreading can easily exhaust the CPU's shared memory bandwidth. To mitigate this, typically more than half of the transistors in a modern CPU are used for cache [54]. Cache memory is faster but also smaller than the CPU's main memory (DRAM). In order to maximize the use of this faster memory, we must reduce the amount of memory used by our software. We already share "read only" data between threads, such as the original clauses of a SAT problem (which conflict analysis needs).

7.6.1 Bit Manipulations

The setting, testing and clearing of individual bits is arguably more common in embedded computing due to the limited memory capacity. Nonetheless, even in high-performance computing where tens of gigabytes of memory are commonly available, being frugal with the memory allows us to squeeze data into the faster CPU cache.

Each thread in our SAT solver needs the following information for each variable:

1. The assignment of the variable (2 bits).
2. Previous variable polarity, we implemented phase saving from RSat [41] (1 bit).
3. Was the variable implied using a learnt clause (1 bits).
4. Depending on item 3, either which learnt clause was used or where in the DMA buffer did hardware write this variable (19 bits).

We pack all of this information into a single 32-bit integer. In many situations we *atomically* update all pieces of information. Using a single 4-byte write saves memory bandwidth (instead of a read, modify, and write for each individual update).

7.6.2 Variable Activity Heap Restructuring

We discussed the VSIDS heuristic in section 2.5.2.4, which guides the SAT solver in choosing the next variable to assign (to continue the DPLL tree search). In summary:

- Each variable has an "activity".
- If a variable participates in a conflict, its activity is increased.
- Choose the most active unassigned variable to start the next round of BCP.

To maintain independent searches on different software threads each running their own SAT solver, any data structures used by VSIDS must be replicated per thread. With many threads, the total memory usage of VSIDS can easily outweigh the clauses (a variable typically occurs in several clauses, but clauses are shared over all threads).

At the very least, we must store the activity of every variable. Typically an array is used. This is actually sufficient to implement VSIDS (if we were optimizing only for memory usage). However, we would need to sweep through the entire activity array to find the variable with the largest activity, which is very slow.

Many modern software SAT solvers implement VSIDS by using three arrays of size V, where V is the number of variables:

1. An activity array.
2. A max-heap (this provides an efficient way to find the variable with the largest activity).
3. An indexing array (if v participates in a conflict, we must increase the activity of v. The indexing array indicates where v is within the max-heap, after which we repeatedly swap v with its parent until the heap order is restored).

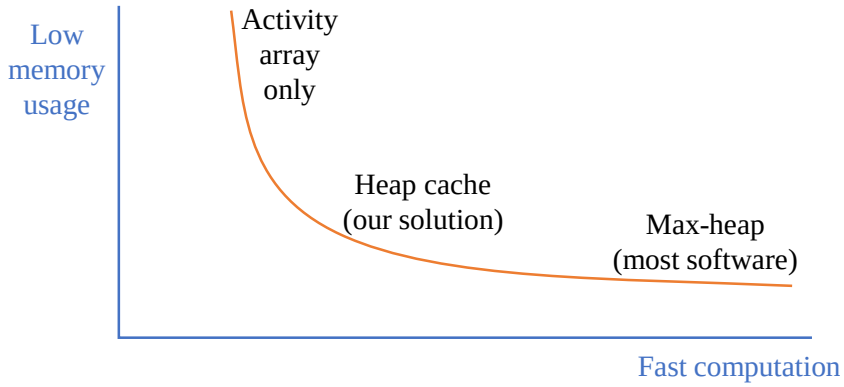


Figure 7.13: Desirable properties such as fast computation and low memory usage can be traded off. However, with numerous software threads thrashing the CPU cache, reducing the memory usage may also increase the computation speed.

Upon assigning the variable with the largest activity, it is removed from the max-heap (to prevent it from being immediately selected again). Later when that variable is deassigned, it is re-inserted into the max-heap. Conceptually, the max-heap should only contain unassigned variables, however this is enforced in a lazy manner. When a variable is implied, we do not go looking for it in the heap. Instead, if it happens to have the largest activity, this is when it will be removed from the max-heap (we would also need to try again to find the next variable to assign).

As shown in Figure 7.13, the two solutions discussed so far represent the extremes on optimizing for low memory usage and optimizing for speed. From an *algorithmic complexity* perspective, these can be traded off. However from a *practical* perspective, with numerous software threads thrashing the CPU cache, reducing the memory usage can also increase the computation speed (by reducing data starvation).

This tradeoff fundamentally arises by using additional memory (in addition to the activity array which is necessary) to store an ordering of the variables. *To reduce the amount of memory, we must reduce the number of variables that are ordered.*

This leads to our so-called "heap cache" solution, where we cache up to M of the most active variables (the cache is ordered, the other variables are unsorted):

- A variable moves from the unsorted group into the cache if its activity is larger than the minimum activity of the cache. This can happen at two points in time:
  1. A variable is deassigned, or
  2. A variable participates in a conflict, thus its activity is increased.
- A variable updates its ranking within the cache if it participated in a conflict.
- A variable moves from the cache into the unsorted group if:
  1. This variable has the highest activity (it will be chosen as the next variable to assign if it is currently unassigned), or
  2. This variable has the lowest activity and we are adding a new variable to an already full cache.

Our cache does not need to be *sorted*, however it needs to be *ordered* in a way that facilitates the easy removal of both the highest activity variable and the lowest activity variable. We have efficiently implemented this by using an interval heap [88], which acts as a double-ended priority queue.

In case the cache is empty (which it will be when the SAT solver starts), we sort the activity of all variables and then fully load the cache with the M most active variables. This is an expensive operation, however with M sufficiently large it is also uncommon, thus the cost is heavily amortized over time. Decent results are obtained with $M = 8192$ whereas our hardware can support SAT problems with hundreds of thousands of variables.

7.7 Summary of Software Integration

A seamless integration between hardware and software is needed in order to fully utilize the BCP acceleration provided by the FPGA. Heterogeneous computing can combine the strengths of various implementation technologies, however the communication latency inherent to such systems must be mitigated. Our multithreaded software is tightly integrated with our PCIe DMA engine in hardware. For instance, hardware can write BCPs into the CPU's memory without any involvement from software. In general, the implementation should ensure that the hardware/software interface does not unnecessarily become the bottleneck.

Software needs to understand our hardware optimizations, for example to perform conflict analysis on enhanced BCP clauses. Software also resolves many of the ensuing complications that arise in hardware, such as the BCP reordering problem.

Since we designed both the hardware and software, we can jointly optimize them. Our timestamp-based conflict analysis is faster than purely software dependency-based reordering, yet it consumes substantially fewer logic resources than ensuring the correct ordering in hardware. Generally, a thorough understanding of the application as well as the computational structure is needed to make such broad optimizations.

Hardware must be stopped in order to safely add learnt clauses. Because of dynamic task creation, this is the only way to ensure a synchronized update of data shared over all threads. In other applications that require intrusive updates, updating only occasionally can amortize this overhead.

Chapter 8

SAT Partitioning and Preprocessing in Software

With the SAT solver aspects of our software established in the previous chapter, this chapter discusses the remainder of our software. Our hardware BCP is distributed across numerous processors, thus we created an algorithm to partition the clauses of a SAT problem. We also discuss the compaction techniques used by our preprocessor. This enables larger SAT problems to fit within hardware memory, thus extending the practicality of our system.

8.1 Clause Partitioning Algorithm

Our use of lossless compression from section 4.2.3 inherently partitions the clauses of a SAT problem. Fewer bits per address can only specify a smaller (and thus localized) region of memory. In this section, we present our software algorithm for partitioning clauses as well as many optimizations to reduce the run time.

Due to the limited amount of on-chip SRAM, our goal is to minimize the number of memory locations needed. Supporting larger SAT problems extends the practicality of our system. There are two independent aspects to memory minimization:

1. Minimize the number of global links. For each variable v, we should minimize the number of partitions that v occurs in.

2. Maximize clause packing. Each macro clause (section 6.2.6) requires N memory locations for N threads of variable assignments regardless of how many variables it contains. Ideally we should use all 12 variables.

We first address the problem of assigning a partition to each clause. Clause packing is discussed later in section 8.1.4. Our preprocessing techniques (section 8.2) also reduce the memory usage, however these are *orthogonal* to compaction from partitioning.

In section 8.1.5, we also demonstrate how to facilitate load balancing between link list traversal engines in hardware. For small SAT problems, excessive compaction (e.g. placing every clause in the same partition) reduces the amount of parallelism, thus decreasing the overall BCP performance.

8.1.1 Motivating the Use of a Bottom-Up Heuristic

Due to the combinatorial nature of assigning a partition to each clause, we think that this problem *could be* NP-hard. Furthermore, this problem would be larger than the original SAT problem since we would have to add clauses to describe the partitioning. It is therefore impractical to use an exact solution. Partitioning is also an overhead for the SAT solver which we should minimize. From a practical perspective, the SAT problem just needs to fit in hardware memory. Conclusively, we use a heuristic.

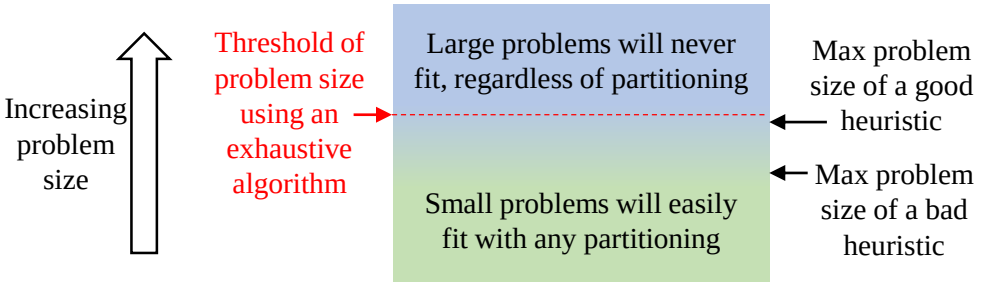


Figure 8.1: Improving the quality of partitioning allows larger SAT problems to fit in hardware memory.

As shown in Figure 8.1, improving the quality of partitioning allows larger SAT problems to fit, yet there is still an upper bound. With a good heuristic, in practice it becomes unlikely that we cannot find a partitioning for a SAT problem that does fit.

One approach is to use a top-down heuristic, where each clause starts in its own partition and these are iteratively merged. We cannot merge partitions if the result no longer fits within the memory of one hardware link list traversal engine. However, one immediately challenge is that it is difficult to *fully* fill each partition given larger blocks. For example, we could have two partitions each requiring 501 memory locations which cannot merge if each hardware engine only contains 1000 locations. Disassembling an earlier grouping may re-introduce global links for variables that were previously contained only within that one partition. To avoid such complications, we have chosen to use a bottom-up approach.

8.1.2 Description of Our Bottom-Up Heuristic

We add one clause at a time to the current partition until it becomes full, at which point we start a new partition. Each partition must fit in the memory of one hardware link list traversal engine. Once a clause is assigned to a partition, it cannot be undone. The algorithm finishes when every clause has been assigned to a partition.

The actual challenge of designing this heuristic is establishing a decent way to *choose which clause to add* to the current partition on each iteration.

Heuristic Insight 1. *To minimize the number of global links, we should "reward" a clause that contains variables already in the partition and "penalize" a clause that contains new variables.*

As clauses are added to a partition, we update which variables already occur within this partition. This set of variables is used for ranking new clauses yet to be added.

As an example, suppose the current partition contains variables a, b, and c. When we consider adding the clause $(c + d)$, variable c is beneficial because there is already a clause in this partition that contains c, hence that clause can use a local link for c. Variable d is unfavorable because d may occur in another partition, in which case a global link would be needed to introduce d into the current partition.

A clause may contain both beneficial and unfavorable *components*. We compute the "net benefit" of each clause using:

$$Net\_benefit(C) = \sum_{i} Individual\_benefit(C_i)$$

where C_i is the i^{th} variable in clause C. This allows us to determine the best clause to add to the current partition on each iteration:

$$C_{best} = \arg\max_{C \in \mathcal{R}} (Net\_benefit(C))$$

where $\mathcal{R}$ is the set of *remaining* clauses. When the algorithm starts, $\mathcal{R}$ = all clauses. If and only if C_{best} can fit into the current partition, then we remove C_{best} from $\mathcal{R}$. If C_{best} does not fit, we start a new partition. The algorithm ends when $\mathcal{R}$ is empty.

The individual benefit that each variable C_i contributes to its clause is:

$$Individual\_benefit(C_i) = \begin{cases} Reward(C_i) & \text{if } C_i \in \mathcal{V} \\ Penalty(C_i) & \text{otherwise} \end{cases}$$

where $\mathcal{V}$ is the set of all variables that the current partition already contains. $\mathcal{V}$ is empty at the start of each partition, and $\mathcal{V}$ is updated after each clause is added. The reward function always returns a positive value whereas the penalty function always returns a negative value or zero.

Heuristic Insight 2. *The "reward" should be larger for existing variables that have fewer remaining occurrences.*

For example, suppose that variable x occurs in 5 clauses. If we have already added 4 clauses that contain x in the current partition, then adding the fifth and final clause will mean that no global links will be needed for x. Conversely, if we had already added only 1 clause that contains x, adding a second clause that contains x still provides a reward, but not as much as previously. As the number of *remaining* occurrences of a variable decreases, we get closer to not needing a global link.

To favor "finishing off" a variable (assigning a partition to every clause that contain this variable), we use:

$$Reward(C_i) = 8e^{-remaining\_occurrences(C_i)}$$

where e is Euler's number ($e \approx 2.718$) and 8 is a scaling factor which we have experimentally determined produces decent results (and likewise for the base of the exponent). The reward function is illustrated in Figure 8.2.

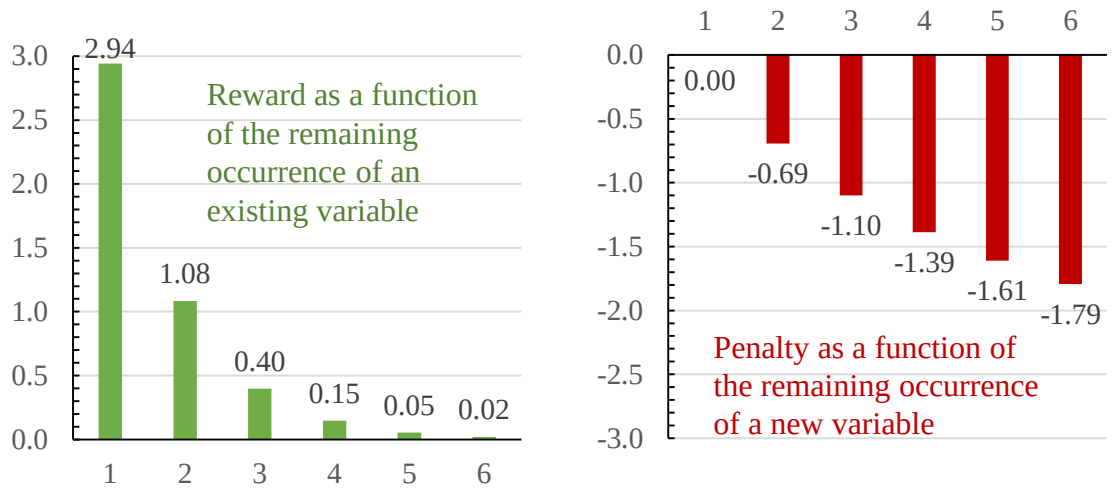


Figure 8.2: Visualization of the reward and penalty functions.

Heuristic Insight 3. *The "penalty" should be larger for new variables that have more remaining occurrences.*

This is similar to the previous insight. For example, it is easier to finish off x if it only occurs in 2 remaining clauses rather than 9 remaining clauses. By finishing off a variable, we prevent it from occurring in a future partition, thus saving a global link.

As illustrated in Figure 8.2, the penalty function has a different shape:

$$Penalty(C_i) = -\ln(remaining\_occurrences(C_i))$$

We use the natural logarithm since $\ln(1) = 0$. If the remaining variable occurrence is only 1, we have to pay for a global link *somewhere* (if we add this clause now then we pay for it in the current partition, otherwise it will be in a future partition). In all other cases, since $ln(x) > 0$ for $x \geq 2$, the penalty function produces a negative value, which decreases the net benefit.

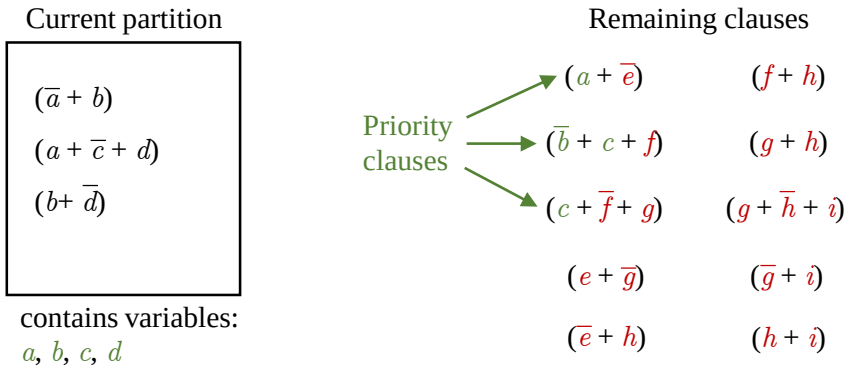


Figure 8.3: Priority clauses have at least one common variable with the current partition. We favor these because they have at least one reward (green). Other clauses only have penalty (red).

8.1.3 Speeding Up the Heuristic

Our greedy heuristic must compute the net benefit for every remaining clause on each iteration. One way to speed up the heuristic is to narrow the selection of which clauses may be added on each iteration. Among the remaining clauses, a clause C is a "priority clause" if C has a variable that is already in the current partition (see Figure 8.3). We favor clauses that have at least one variable in common with the variables that are already in the partition since these clauses will have *at least one reward*.

We label the set of all priority clauses as $\mathcal{P}$. We can now update the criteria for selecting the best clause C_{best} to add to the current partition:

$$C_{best} = \begin{cases} \arg\max_{C \in \mathcal{P}} (Net\_benefit(C)) & \text{if } \mathcal{P} \neq \emptyset \\ \text{any } C \in \mathcal{R} & \text{otherwise} \end{cases}$$

The set of remaining clauses $\mathcal{R}$ is never empty, unless the partitioning algorithm is finished. Conversely, $\mathcal{P}$ will be empty if the current partition is empty. When we start each partition, any remaining clause can be chosen (the net benefit is not computed).

Narrowing the selection of clauses from $\mathcal{R}$ to $\mathcal{P}$ on each iteration results in significantly fewer computations of net benefit. However, one partition can only fit a few hundred clauses, yet we have noticed that $\mathcal{P}$ can grow to thousands of clauses. We further decrease the run time of the partitioning algorithm by not considering substantially more candidates than we can actually fit in one hardware partition.

We isolate from $\mathcal{P}$ a subset of "active" priority clauses, and the remainder of $\mathcal{P}$ is considered "inactive":

- Only a clause from the active set can be added to the current partition.
- When new clauses are added to $\mathcal{P}$ (as a result of a new variable being introduced into the current partition), these clauses are added to the active set.
- When the active set contains more than 20 items, we sort the clauses in the active set by net benefit and only the top 10 are kept (the others are moved to the inactive set).
- When the active set contains fewer than 5 items, the entire inactive set is moved into the active set. This may induce the above step.

The above thresholds were empirically determined and can be easily modified by the user of our software. Larger thresholds typically produce incrementally better results at the expense of significantly more run time.

Additional pruning can be achieved by filtering away clauses that are extremely unlikely to have the largest net benefit. Suppose we introduce a new variable v into the current partition. If there are new clauses that contain v, we must update $\mathcal{P}$. Suppose a clause C is one of these newly added clauses. Notice that C is *only allowed into* $\mathcal{P}$ because of the newly introduced variable v. All of the other variables in C are

not in the current partition (or else we would have earlier added C to $\mathcal{P}$). The net benefit of C is composed of:

- One reward from v, and
- Several penalties from all of the other variables in C.

In the case where v occurs in many remaining clauses, the reward from v will be very small. Given all of the other penalties, C is highly unlikely to have the largest net benefit. Therefore we may as well prevent C from being added to $\mathcal{P}$ in order to reduce the number of net benefits that we need to compute.

This pruning mechanism only applies if v occurs in *many* remaining clauses. As we later add more clauses that contain v, the remaining occurrence of v will decrease. When the remaining occurrence eventually reaches 3, this is when we allow C to be added to $\mathcal{P}$. This threshold was empirically determined and is easily modifiable.

Finally, the reward and penalty values are pre-computed (stored in arrays) to speed up the computation of each net benefit.

8.1.4 Clause Packing

To minimize the number of memory locations used for storing variable assignments, the original clauses must be packed as tightly as possible into macro clauses (from section 6.2.6). We support clauses with up to 12 variables, and this is regulated during SAT preprocessing.

We greedily group clauses from largest to smallest. In other words, we pack all size 12 clauses before even considering how to pack the size 11 clauses. By the time we try to pack size N clauses, there will be no more clauses of size $N + 1$ or larger.

1. Clauses with 12, 11, and 10 variables are too large to pack with anything else.
2. Clauses with 9, 8, and 7 variables may double pack with clauses of size 3, 3-4, and 3-5, respectively.
3. Size 6 clauses can double pack. If there is one leftover, we look for the best remaining grouping (two size 3 clauses, one size 5, one size 4, or one size 3).
4. Size 5 clause pack best with one size 3 and one size 4 clause. Leftovers are then double packed. A single leftover is grouped in the best way remaining.
5. Size 4 clauses can triple pack. Size 3 clauses can quadruple pack. All possible leftovers will pack without wasted space (as exhaustively verified).

Recall from section 6.2.5 that most size 2 clauses can be eliminated, and the few remaining ones are treated as size 3 with minimal wastage.

In our C++ implementation, we save computation with a simpler overestimation of the space required by the variable assignments:

$$var\_assign\_space = (\lceil s3/4 \rceil + \lceil s4/3 \rceil + \lceil (s5 + s6)/2 \rceil + sLarge) \times num\_threads$$

$$sLarge = s7 + s8 + s9 + s10 + s11 + s12$$

where $s3$ is the number of size 3 clauses, likewise for the others. $sLarge$ is the number of clauses that are too large to double pack with another clause of the same size.

Only if the estimation indicates that adding a clause to the current partition will not fit, then we use an exact check. Once we start using the exact check, typically only a few more clauses can be added until the partition is actually full (at which point we will start a new empty partition). This reduces the total amount of computation.

8.1.5 Partition Size Regulation

If a small SAT problem consumes one quarter of our on-chip memory, we could:

- Fully fill one quarter of all partitions (3 out of 4 partitions would be unused),
- Fill every partition to one quarter capacity, or
- Fill half of the partitions to half capacity.

Increasing the number of partitions used enables more parallel processing in hardware, however it also increases the amount of on-chip network traffic, thereby adding latency to a link list traversal. The full characterization of this tradeoff is extremely complex and beyond the scope of the thesis.

From a practical perspective, our experimentation *suggests* that it is better to use more partitions. Suppose each hardware clause traversal engine contains X memory locations (we used $X = 8192$). If we estimate that a given SAT problem only needs one quarter of the memory, we could artificially tell the partitioning algorithm that each partition only contains $X/4$ (e.g. 2048) memory locations. This causes approximately four times as many partitions to be used (compared to using X memory locations), thereby spreading the SAT problem over most of the FPGA's embedded memories.

The estimate of the problem size must be an overestimation. If we over constrain the partition size limit (make it too small), the partitioning algorithm may not find a solution to a problem that actually does fit in hardware. To prevent this mishap, our estimation assumes:

- Each occurrence of a variable will use a local link and a global link.
- Clauses are loosely packed. We reuse the over approximation of var_assign_space from section 8.1.4.

8.2 Preprocessing for SAT Problem Compaction

With the establishment of optimizations for the software in our SAT solver, we now focus on our software SAT preprocessor. In this section, we present several techniques that preserve the satisfiability of an arbitrary SAT problem while also compacting it. This enables us to support larger SAT problems, thus extending the practicality of our system.

8.2.1 Description of Our Five Preprocessor Stages

Our preprocessor consists of five stages, which are discussed in the following five subsections. For Boolean SAT problems, all five stages are run sequentially (the output of the first stage is the input for the second stage, and so on). We intentionally split the entire preprocessor into distinct stages, as discussed in section 8.2.2.

8.2.1.1 Boolean Propagation

This first stage of the preprocessor is generic, as these optimizations will help *any* SAT solver. We alternate between two processes until no more updates are found:

1. Propagation of unit clauses.

2. Size two clause pairings.

If a clause only has one variable, we apply BCP and subsequently eliminate this clause and its variable. This may produce more unit clauses, which are also propagated. Clauses may also shrink in size, e.g. assigning $a = 0$ to $(a + b + c)$ results in $(b + c)$.

Among only the clauses with two variables, we look for two clauses that contain the same variables. Duplicate clauses are removed. One of two cases will happen:

1. One variable has the same sign. For example, if we find $(a + b)(a + \overline{b})$, we must assign $a = 1$, thus we will later run the propagation of unit clauses again.

2. Both variables have opposite signs. For example, if we find $(a + \overline{b})(\overline{a} + b)$, we know that $a = b$, thus we can replace all occurrences of b with a.

We alternate between the two processes because one can produce work for the other. Finally, the first stage of our preprocessor ends by compacting the variable range. For example, we convert $(x_3 + \overline{x}_5)(x_2 + \overline{x}_3 + \overline{x}_6 + x_9)$ into $(y_2 + \overline{y}_3)(y_1 + \overline{y}_2 + \overline{y}_4 + y_5)$ since there are only five unique variables. Most modern SAT solvers store the assignment of each variable in an array, thus compacting the variable range results in better memory utilization. Array-based data structures in general will benefit from this.

8.2.1.2 Gate Extraction

In the second stage, we essentially apply pattern matching to the Boolean clauses to identify and subsequently extract XOR gates, NOR gates, and 2:1 multiplexers. For example, if we find:

$$(a + b + c)(\overline{a} + \overline{b})(\overline{a} + \overline{c})$$

we can replace these Boolean clauses with one enhanced BCP clause: $a = b$ NOR c.

Some of the problems in the SAT competition 2013 benchmarks [33] encoded more elaborate logic structures rather than primitive gates. One example is illustrated in Figure 8.4, where signal a is a function of 4 inputs. Notice that the Boolean encoding does not include variable f. We introduced this dummy variable so that this logic will conform to our supported enhanced BCP clause types. We are able to identify some but not all of these complex logic structures. This is further discussed in section 8.2.2.

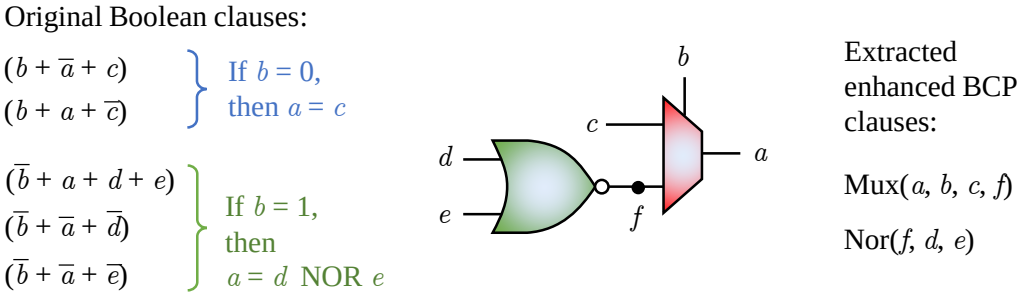


Figure 8.4: We are able to extract some Boolean SAT encodings of complex logic structures. To conform to our supported enhanced BCP clause types, extraction may introduce a dummy variable, such as f.

8.2.1.3 Gate Combining

The third stage of our preprocessor attempts to further compact the enhanced BCP clauses that were extracted in the previous stage:

- We search for several 2-input NOR gates that combine to specify:
  - One 2:1 multiplexer (3 NOR gates, see Figure 8.5),
  - One 2-input XOR gate (3 NOR gates), or
  - One 3-input majority function, which is the carry logic in a 1-bit full adder (4 NOR gates).
- We search for 2-input NOR gates that pair with 2-input XOR gates to form a half adder.

Some SAT problems from hardware verification applications directly encode the netlist into SAT. By extracting more complex logic, such as a multiplexer, we obtain a higher ratio of lossless compression with enhanced BCP. Furthermore, if the outputs of the NOR gates used to specify the multiplexer are not used anywhere else, we can eliminate these variables, as shown in Figure 8.5.

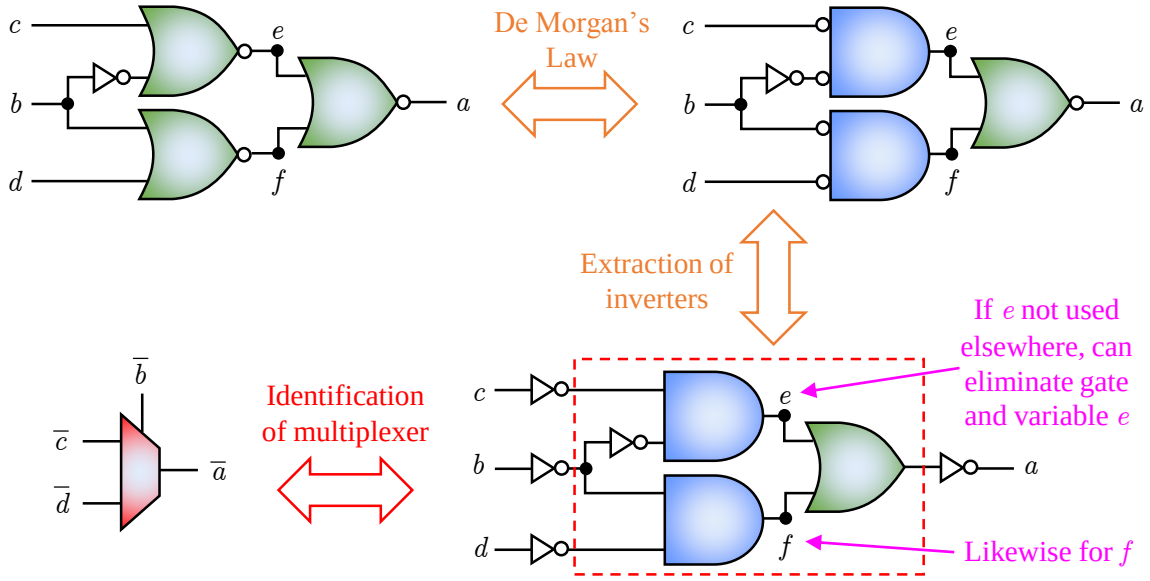


Figure 8.5: In addition to Boolean clauses, multiplexers can also be extracted from NOR gates.

We apply common subexpression elimination to NOR gates to reduce the total number of variable occurrences. As an analogy with AND gates, suppose we have:

$$\overline{x} = \overline{a}\ \&\ b\ \&\ c\ \&\ \overline{d}\ \&\ e$$
$$y = \overline{a}\ \&\ b\ \&\ c\ \&\ \overline{d}\ \&\ \overline{f}\ \&\ g$$
$$z = \overline{a}\ \&\ b\ \&\ c\ \&\ h.$$

It is better to instead factor out the common terms:

$$t = \overline{a}\ \&\ b\ \&\ c.$$

Although we have added 4 variable occurrences, we will lose 6 variable occurrences (by removing a net of two occurrences per AND gate) once we make the following substitution:

$$\overline{x} = t \;\&\; \overline{d} \;\&\; e$$
$$y = t \;\&\; \overline{d} \;\&\; \overline{f} \;\&\; g$$
$$z = t \;\&\; h.$$

We could also factor out $(t \;\&\; \overline{d})$, however it would not be beneficial in this case.

Finally, gate combining tries to eliminate variables by combining primitive logic gates into larger ones. For example, suppose we have:

$$a = b \text{ XOR } c$$
$$c = d \text{ XOR } e.$$

If c does not occur anywhere else, we can safely eliminate c to obtain:

$$a = b \text{ XOR } d \text{ XOR } e.$$

8.2.1.4 Boolean Compaction

In the fourth stage of our preprocessor, we attempt to compact the remaining Boolean clauses that could not be template matched to enhanced BCP clauses. The same common subexpression elimination technique as above is used.

Common subexpression elimination is essentially the reverse of variable elimination (from section 2.6.1). SAT solvers that are purely Boolean and purely software typically favor many large clauses with fewer variables because:

- Fewer variables results in a smaller DPLL tree search.
- As the size of a Boolean clause increases, it becomes less likely to make it all the way to only one unassigned variable without being satisfied.

Conversely, if we are given a large SAT problem, we actually favor the opposite because we must ensure that the problem actually fits into the FPGA's memory. This is our motivation for reducing the total number of variable occurrences at the expense of introducing dummy variables.

Finally, we extract partial NOR clauses (see Table 6.1), which have the form:

$$(a + b)(a + c)(a + d)(a + e) \ldots$$

Partial NOR clauses are essentially a grouping of size 2 clauses that share a common variable. Enhanced BCP (section 6.1) is a form of *lossless compression* and therefore does not necessarily need to be a logical construct.

8.2.1.5 Regulation

The fifth and final stage of our preprocessor ensures that the SAT problem conforms to the limitations of our hardware:

- Clauses may only have up to K variables (our implementation uses $K = 12$). In general, K can be as small as 3 if different hardware optimizations were desired, for example. Boolean clause splitting was demonstrated in section 2.6.3. Splitting an enhanced BCP clause typically involves logic gate decomposition, except for Partial NOR which decomposes easily.

- For each variable with high occurrence, we may decide to split it into several equivalent variables in order to avoid long link lists in hardware (as further discussed in section 4.4.2).

8.2.2 Customizing the Preprocessor Flow

A purely Boolean SAT problem is typically represented in DIMACS format [33]. The example below illustrates the basic idea of the format:

```
c one-line comments start with a 'c' character
c below we specify that the problem has 3 variables and 2 clauses
p cnf 3 2
c with 3 variables, they must be labeled as 1, 2, or 3
c each clause then ends with a 0
1 -2 3 0
-3 1 0
```

The above example shows how we specify the SAT problem $(x_1 + \overline{x}_2 + x_3)(\overline{x}_3 + x_1)$.

All five preprocessor stages use a common format to represent the enhanced BCP SAT problem, which extends the DIMACS format:

```
p cnf 3 4
bool 1 -2 3 0
-3 1 0
c if no label is provided, the clause type is assumed to be boolean
xor 2 -3 0
nor -2 -1 3 0
```

Any SAT problem in DIMACS format also conforms to our format. A common "superset of DIMACS" format allows any stage of our preprocessor to be bypassed.

As shown in Figure 8.6, we could skip stages 2 and 3 if we wanted to test hardware without enhanced BCP (e.g. to measure the compression that enhanced BCP provides).

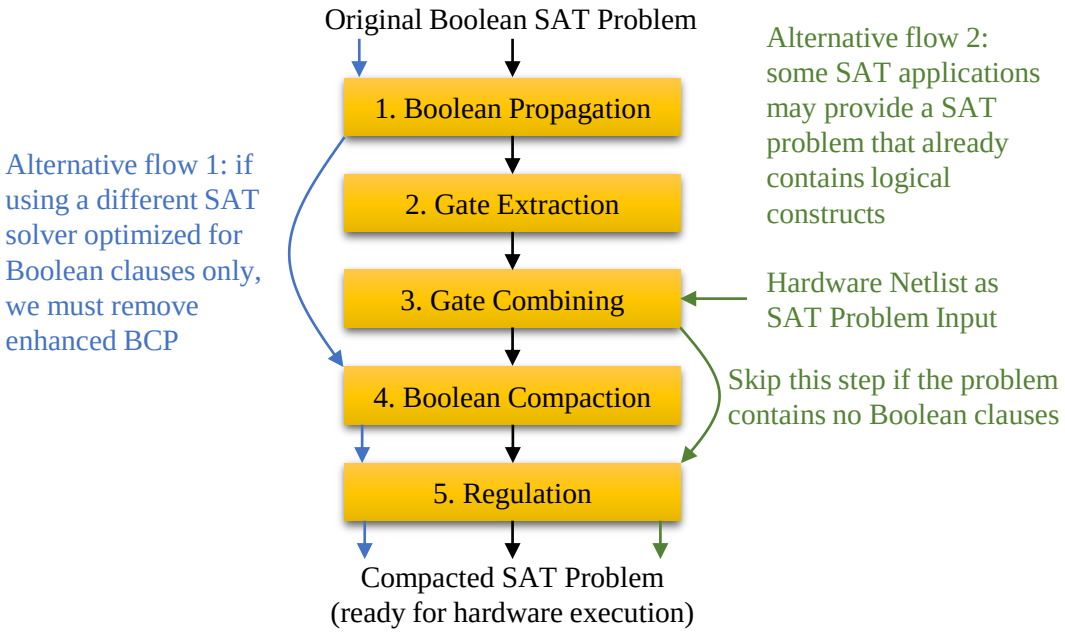


Figure 8.6: The flow of our SAT preprocessor can be customized. The default flow runs all five stages sequentially. Stages can be skipped based on the intended usage.

Alternatively, if a SAT problem has few logical constructs, we could use a different software SAT solver which is optimized for Boolean clauses only.

Many applications of SAT (such as software verification, hardware optimization, and cryptography) inherently involve logical constructs. Ideally we should be given the problem in this format (e.g. a hardware netlist) instead of having to extract logical constructs from Boolean clauses. As illustrated in Figure 8.6, we would start the preprocessor at stage 3 and possibly skip stage 4 if there are no Boolean clauses at all.

The Boolean encodings of logical constructs are not always obvious, as exhibited in Figure 8.4. We cannot always infer what the entire structure is, hence we are missing opportunities where enhanced BCP could have been used. This problem can be completely avoided by having a tighter integration between the SAT solver and the application.

8.3 Summary of Software Compaction

In this chapter, we presented software techniques for compacting the SAT problem. These are complementary to the advanced hardware optimizations from chapter 6. SAT problem compaction has two primary advantages:

1. We can support larger SAT problems in hardware memory. This extends the practicality of our system.

2. Compacting a SAT problem to use less software memory facilitates better utilization of CPU cache for the non-BCP parts of the SAT solver in software.

Compaction is an overhead for software, as the computation needed to perform it is separate from the SAT solver itself. However, it is of questionable practical value to accelerate easy SAT problems, which are quickly solvable with pure software. As the difficulty of a SAT problem increases, we expect the overhead of performing compaction to decrease relative to the amount of computation required by the SAT solver. Our experimental results in chapter 9 demonstrate that this overhead is small in practice.

Chapter 9

Experimental Results

A massive implementation effort was required to develop a fully operational SAT solver which accelerates BCP in a real FPGA (not simulation) with tight hardware/software integration. With such a complex system, debug and test were major challenges, and we begin this chapter by briefly discussing our testing methodology.

Unlike many prior works which relied on simulation, our real-life implementation allows us to *accurately observe system-level limitations*, such as exhausting the CPU's memory bandwidth with many concurrent software threads. This may also have a negative effect on the hardware/software communication latency. Furthermore, simulation is several orders of magnitude slower than the real system, thus limiting the scale of testing for many prior works. Our overall system performance is measured in absolute time (wall-clock time in software).

Due to our extensive use of lossless compression, we are able to support significantly larger SAT problems than all prior works that accelerate DPLL-based SAT in hardware (excluding DRAM-based approaches, which would likely perform slower than software, see section 3.3.2). Some prior works benchmark with SAT problems that would fit

within a single link list traversal engine in our system, yet there is little practical value in accelerating small SAT problems which take pure software only milliseconds to solve. In addition to compression, our use of multithreading increases the performance. We can benchmark SAT problems in uncharted territory for FPGA-accelerated SAT.

9.1 Testing Methodology

Although this chapter focuses on performance, the *correctness* of the system cannot be overlooked. In this section, we briefly discuss our strategies for testing our system.

By accelerating only BCP in hardware (instead of the entire SAT solver), we only need to support one type of processing. This simplifies both the design and test since:

1. We can replicate the same processor throughout the entire hardware design. Less effort is required to design and test one processor rather than several different types of processors.

2. One type of processing means that everything else must be communication. This facilitates a separate development for processors and for communication.

Due to the repetitive structure of our on-chip networks, we found that isolated testing by simulation only was sufficient. Using parameterizable traffic generators and consumers, in simulation we can easily widen the data paths to add tagging information to each network packet. Since we have full visibility of the generators and consumers, we can verify that every network packet was successfully delivered. Varying amounts of randomized backpressure were also tested.

For our hardware link list traversal engines, we have used progressively less detailed tests in order to facilitate larger amounts of randomized testing:

1. Debug almost always begins as a manual process, e.g. using a behavioral simulation to check the value of specific registers on every clock cycle.

2. We then moved to a memory-based strategy. After each round of BCP, we would compare every value in the hardware memory to a golden reference. This was done both in simulation as well as with a real FPGA (we would read each memory location via PCIe and do the comparison in software).

3. Finally we moved to BCP-based strategy. For each variable that our software SAT solver chooses to assign, we would also perform BCP in software in order to compare with the results from hardware BCP. Also, if our SAT solver reports that a problem is satisfiable, we check that the variable assignments actually satisfy the original clauses. Checking hardware against software is as close as *practically* possible to a full system verification.

We coded our own purely software SAT solver to *exactly* match the behavior of Minisat [37], which is an established SAT solver. For many SAT problems, we verified that we always chose the same variable to assign, implied the same variables, observed the same conflicts, and produced the same learnt clauses. This served as the basis for the non-BCP parts of our SAT solver which would remain in software. We coded our own software SAT solver in order to customize the data structures for easy integration with our own PCIe driver as well as to support enhanced BCP and clause packing.

Finally, we have designed specific regression tests to exercise certain corner cases. Using hand-crafted SAT problems that deliberately induce an excessive number of BCPs, we can test the offload and reload networks (all the way to offloading into the CPU's memory). This also tests the cached sliding window of the BCP central buffers.

9.2 System Specification

In all of our benchmarks, our host computer used the Intel Core i7-980 CPU (6 cores, 3.33 GHz, 12 MB L3 cache). It has three DRAM channels each with 8 GB DDR3-1067 with 7-7-7-20 latencies (24 GB total RAM). Our software environment (for the non-BCP parts of the SAT solver) used Ubuntu Linux 12.04 LTS 64-bit [89]. All C and C++ software was compiled with gcc and g++ 4.8.1 [90] respectively using maximum optimization. We used pthreads [91] for multithreading.

We used the Altera Stratix V DSP Development Kit FPGA [59]. Each BCP uses 8 bytes, and the third generation PCIe interface with 8 lanes has a theoretical maximum of 8 GB/s bandwidth. Our hardware processors and communication infrastructure all operate at 400 MHz. The clock changes to 250 MHz only at the PCIe interface.

Our design features 200 clause traversal engines, each with 8192 memory locations (20 bits per location). We have 12 BCP central buffers to support up to 12 threads. Our memory layout enables the same hardware to be used with fewer threads.

9.2.1 FPGA Resource Utilization

As summarized in Table 9.1, our design is limited by the amount of on-chip SRAM. We could not use up all M20K embedded memories since these were not uniformly distributed physically across the FPGA. Dense regions of memory lacked sufficiently local amounts of logic resources to implement extra processing engines.

Adaptive lookup tables (ALUTs) can be used to implement combinational logic as well as LUT-based memory. This is useful for small queues in which using an entire M20K (20 kilobits) instead would be wasteful. 20 ALUTs and 20 registers are grouped into one logic array block (LAB). Different parts of the design use different ratios of

Table 9.1: Resource utilization of our entire FPGA design in relation to device capacity.

Total number of ALUTs	199,318 / 345,200 = 58%
Total number of registers	167,992 / 345,200 = 49%
Total number of LABs	16,181 / 17,260 = 94%
Total number of M20K memories	1,891 / 2,014 = 94%

Table 9.2: Top level distribution of resources. Each category includes the corresponding networks (e.g. regional overflow encompasses the global distributed shift registers).

	ALUTs	Registers	M20Ks
PCIe	4,311 (2%)	4,247 (3%)	15 (1%)
Central overflow buffer	2,188 (1%)	2,285 (1%)	8 (<1%)
All 18 regional overflow buffers	3,692 (2%)	8,967 (5%)	72 (4%)
All 12 central BCP buffers	16,475 (8%)	17,701 (11%)	96 (5%)
All 100 processors	172,652 (87%)	134,792 (80%)	1,700 (90%)

ALUTs to registers, thus we typically cannot utilize all of the resources within each LAB. From Table 9.1 it may appear that many more ALUTs and registers can be used, yet the FPGA is actually nearly full based on the number of LABs. This is also confirmed by the floorplan image of our implementation in Figure 5.23.

Utilization is also restricted by routing. Each LAB has a limited number of inputs and outputs, which is substantially smaller than the total number of inputs and outputs from all ALUTs and registers. The total amount of interconnect was only 27% of the device capacity. As shown in Figure 5.24, routing tends to be concentrated within each processor. The pipeline registers within our on-chip networks facilitate localism, which reduces the amount of routing between processors.

Table 9.2 shows the distribution of resources to each major functional group. Each category includes all networks that physically reside in that area. For example, the BCP central buffers also include a reload network (to receive BCPs), an offload network (to send to PCIe), insertion into the random access network (for deassignments), and

Table 9.3: Average resource utilization for one processor. The contents of one processor was defined in Figure 5.21.

	ALUTs	Registers	M20Ks
Total	1,723	1,331	17
Both clause traversal engines	553 (32%)	436 (33%)	16 (94%)
Constraint propagation	284 (16%)	181 (14%)	1 (6%)
All 5 queues	457 (27%)	232 (17%)	0
All 3 networks	233 (14%)	284 (21%)	0
Other (mostly queue arbitration)	196 (11%)	198 (15%)	0

the vertical trunk of the synchronization network (see Figure 5.13). PCIe includes transceiver configuration, the DMA engine, and clock domain crossing queues.

The average distribution of resources for one processor is presented in Table 9.3. The distribution is an *estimation*. Optimizations within the CAD tool (Quartus 13.1 in our case) may restructure combinational logic such that logic from one source file is implemented in a different part of the design to jointly optimize the entire design.

We have 100 processors, yet the total in Table 9.3 multiplied by 100 is slightly less than the amount in Table 9.2. Our on-chip network is a full 2D grid of 12 rows by 9 columns. Of the 108 network interfaces, 8 of them have no processing engines and no queues (network traffic can flow through, insertion and extraction are not allowed).

Each clause traversal engine uses 8 M20Ks. Constraint propagation time-shares one M20K for "clause information", which indicates the packing and enhanced BCP types (see section 6.2). All five queues are implemented with LUT-based memory.

Compared to the entire processor, relatively few resources are used by all three networks (distributed shift register, synchronization, and random access). Much of the resources are left for the processing itself, since communication can be regarded as an overhead which performs no computation yet consumes resources.

9.3 Performance Metric

As motivated in section 3.4, we accelerate only BCP in hardware. The remainder of the SAT solver is implemented in software to facilitate an easy integration of new SAT innovations as they continue to be developed. Our hardware is compatible with *any* DPLL-based SAT solver, which almost all modern SAT solvers are [33].

Improvement of techniques such as how to choose the next variable to assign attempt to reduce the amount of computation needed in order to solve a SAT problem. This is *orthogonal* to the acceleration of BCP, which performs the same amount of computation in less time.

For this reason, our performance metric is the *number of BCPs computed per second*. This is an abstraction of the ultimate metric, which is arguably the number of SAT problems that can be solved within a given amount of time. Our chosen metric is as close as possible to this ultimate metric given that our contributions are *independent of the continually evolving non-BCP aspects of the SAT solver*.

9.4 Comparison with Prior Hardware SAT Work

Throughout this chapter, our comparisons will be mostly against software. Almost all of the prior works on hardware SAT suffer from limited practicality because:

- Local search algorithms cannot prove that a SAT problem is unsatisfiable due to their incompleteness (section 2.3).
- The FPGA design uses DRAM memory, which is typically slower than the CPU's DRAM (used by software) and drastically slower than CPU cache (section 3.3.2).

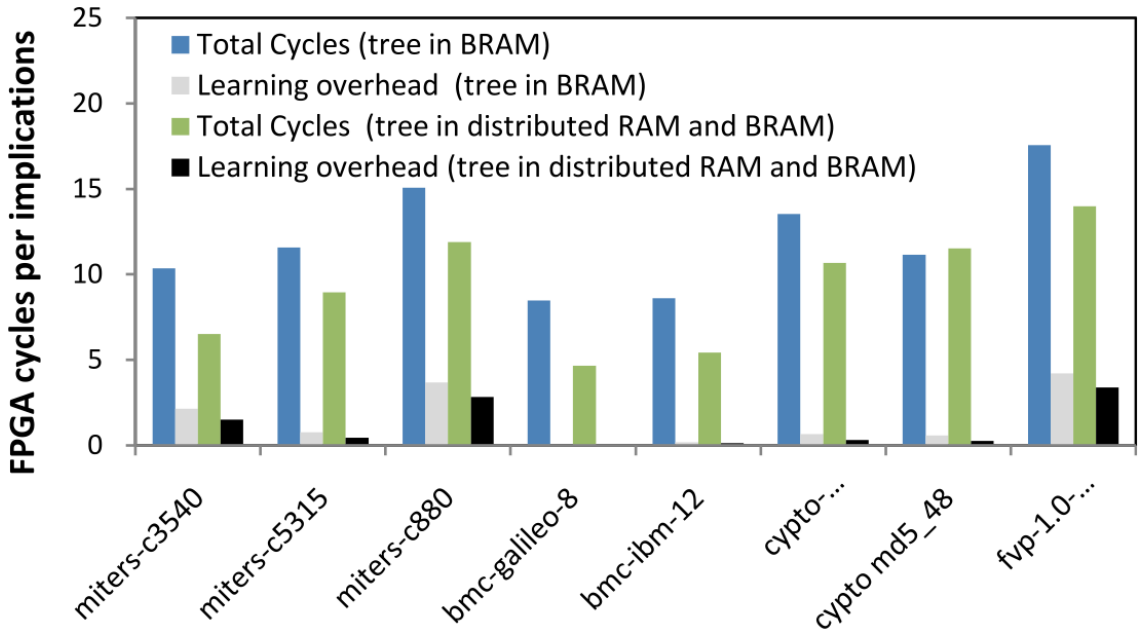


Figure 9.1: The results from Davis *et al.* (Figure 9 reproduced from [81]).

- The hardware only supports small SAT problem sizes due to the lengthy compile time of an instance-specific design (section 3.2.1) or needing to share the limited hardware resources with non-BCP tasks (section 3.4.1).

The *only* prior work that overcomes all of the above limitations is from Davis *et al.* [80]. Learnt clauses were added in [81]. Like our design, they only accelerate BCP using SRAM in hardware. However, as discussed in section 3.5.3, their design inherently cannot support multithreading on the FPGA, although in fairness it would not have been practical at the time.

A limited amount of results were presented in [81], as shown in Figure 9.1 (we have reproduced Figure 9 from [81]). Results were generated using *only simulation* and it was *assumed* that the clock frequency is 200 MHz. Our experience suggests that this clock speed would have been difficult to actually implement 6 years ago, however it is

Table 9.4: Sizes of the SAT problems in Figure 9.1 (Table 1 reproduced from [81]).

SAT Problem	Variables	Clauses
miters-c3540	3451	9327
miters-c5315	5400	15025
miters-c880	958	2591
bmc-galileo-8	58075	294822
bmc-ibm-12	39599	194661
crypto-md4_wang5	53229	221185
crypto md5_48	66893	279265
fvp-1.0-1dlx_c_mc_ex_bp_f	777	3726

Table 9.5: Millions of BCPs per second for the problems that were actually available from Table 9.4. Our speedup relative to [81] is shown in brackets.

SAT Problem	Davis [81]	Our design (12 threads)	Our design (3 threads)	Our design (1 thread)
bmc-galileo-8	40	102 (2.6×)	40 (1.0×)	15 (0.4×)
bmc-ibm-12	33	150 (4.5×)	43 (1.3×)	20 (0.6×)
crypto md5_48	18	83 (4.6×)	22 (1.2×)	7 (0.4×)

a reasonable expectation today with faster FPGAs and better CAD tools. We will therefore compare results in terms of absolute time per BCP.

A summary of the SAT problem sizes is shown in Table 9.4 (this is Table 1 reproduced from [81]). Some of these problems originated over one decade ago and are no longer accessible. For the three problems that we were able to find, Table 9.5 presents the performance in terms of millions of BCPs per second. The results for [81] were extracted from Figure 9.1 using their assumed 5 ns clock. Our own results were measured in absolute time in software. When we use all 12 threads, we are several times faster.

It is difficult to obtain a fair comparison with [81]. If we *intentionally handicap* our system to one thread, then our results become worse because:

- Our measurement of BCP performance includes the remainder of the SAT solver (e.g. conflict analysis) which acts as an overhead since [81] only considers BCP.
- Our hardware/software latency is higher. We used PCIe whereas [81] only *simulated* a HyperTransport interface (which has lower latency than PCIe).

It is also extremely unlikely that their simulation would account for system-wide limitations, such as a PCIe packet being delayed because the CPU's memory system must first read data from DRAM in case of a cache miss. We mitigate this latency with multithreading, and with only 3 threads we are already faster than the best possible performance from [81] since their design cannot support multithreading.

9.5 Randomized Boolean 3-SAT Benchmark

In this section, we benchmark our system using SAT problems where every clause has *exactly three* variables. Variables were chosen randomly using a uniform distribution. This is one of commonly used benchmarks [33]. Due to the symmetry in each problem, the SAT solver is forced to learn the relationship between variables instead of simply exploiting the problem structure. Only Boolean clauses were used (enhanced BCP was not used, yet our hardware and software still support it). These SAT problems do not contain any structures our preprocessor can simplify, so we bypassed preprocessing.

9.5.1 Limitations of Accelerating Easy SAT Problems

In [80], random 3-SAT instances were benchmarked. With PCIe, they claimed a speedup of 6.7× over software using 200 variables and 860 clauses. We ran this experiment using 100 of our own random instances, however our system solved every

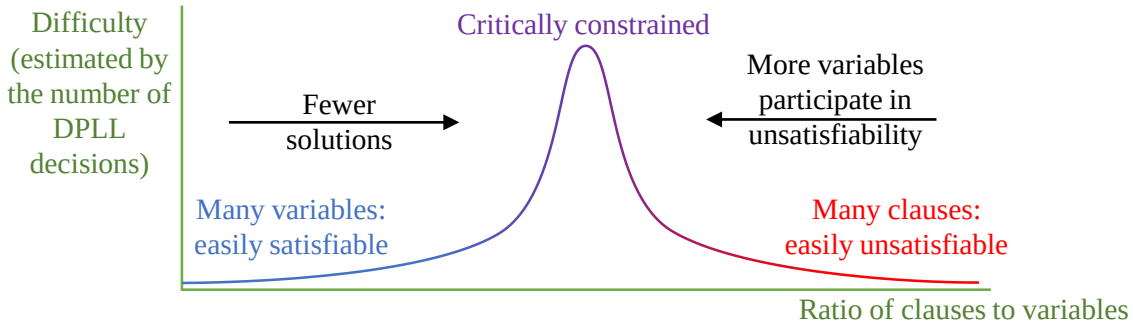


Figure 9.2: The ratio of clauses to variables affects the difficulty of SAT problems. This is a *qualitative* illustration and is not drawn to scale. Exact values are provided throughout [92] for problems of various characteristics.

instance so quickly that learnt clauses never got the chance to be added to hardware (we only occasionally add them to amortize the cost, see section 7.1). The average run time was under 1 second. With 12 threads we produced 45 million BCPs per second, although the measurement accuracy is questionable given the short time duration. Minisat [37] in software only produced 5.4 million BCPs/s on the same 100 problems, yet it required slightly less total time than our system to solve every problem. Learnt clauses are immediately usable in software. *For easy SAT problems, accelerating BCP in an FPGA is of limited practical value due to the overhead of hardware/software interfacing* (and our in case, also the overhead of clause partitioning).

Many real-life SAT problems are non-trivial, hence acceleration does provide a practical benefit. To emulate this with randomized SAT problems, we need to use the appropriate ratio of clauses to variables. For 3-SAT problems, [92] demonstrated that a ratio of 4.3 (e.g. 860 clauses to 200 variables) resulted in the most difficult SAT problems. Intuitively, too many variables (a low ratio) will under constrain the problem, thus making it easily satisfiable. For satisfiable problems, generally fewer solutions remain as this ratio increases, thus making it more difficult to find a solution. Conversely, too many clauses (a high ratio) will over constrain the problem and make

Table 9.6: Detailed performance of our system on the Boolean 3-SAT problems. A timeout of 60 seconds was used. At each problem size, we average over 48 random instances. Results are left blank when the problems are too large to fit into hardware.

Problem specification		Millions of BCPs per second			
Clauses	Variables	4 threads	6 threads	8 threads	12 threads
20,000	4,651	30.87	42.06	49.17	60.84
40,000	9,302	40.57	57.44	67.05	82.40
60,000	13,953	44.73	64.16	74.60	90.99
80,000	18,605	46.50	66.97	77.81	94.35
100,000	23,256	47.09	68.10	78.88	93.66
120,000	27,907	47.40	68.55	79.27	91.84
140,000	32,558	47.57	68.53	78.95	89.28
160,000	37,209	47.52	68.40	78.32	86.41
180,000	41,860	47.28	68.05	77.64	84.06
200,000	46,512	47.08	67.49	76.72	
220,000	51,163	46.94	67.12	76.03	
240,000	55,814	46.66			

it easily unsatisfiable. It is typically easier to prove unsatisfiability if fewer variables participate in the contradiction. As shown in Figure 9.2, there is a certain ratio at which SAT problems become critically constrained. The number of DPLL decisions needed to solve such problems becomes extremely large.

9.5.2 Performance Analysis of Our System

Table 9.6 provides detailed results of our system performance on the randomized Boolean 3-SAT benchmark. Although enhanced BCP (section 6.1) could not be used, we still used clause packing (section 6.2). For practicality reasons, we could not let our SAT solver run until a solution is finally found, hence the timeout of 1 minute. At each problem size, the results were averaged over 48 random instances. The same SAT problems were used as we vary the number of threads. Being critically constrained

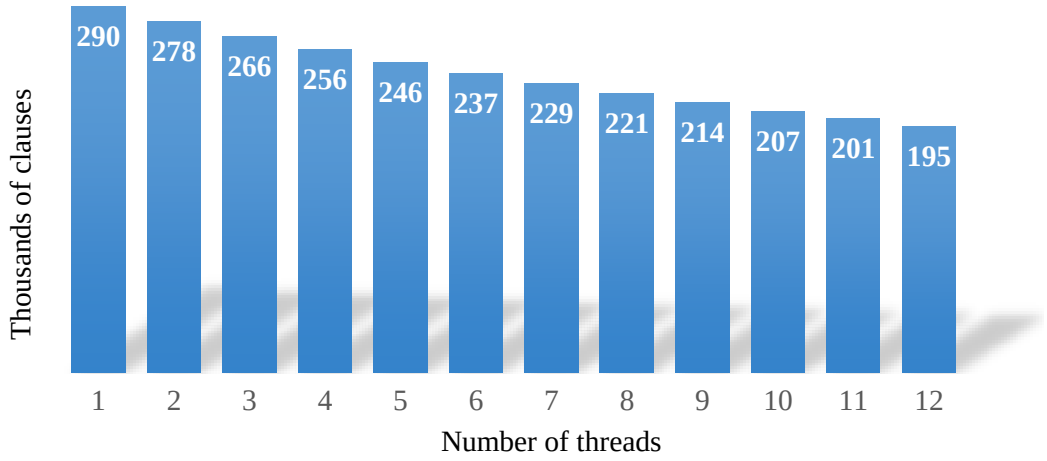


Figure 9.3: Maximum problem size that our hardware supports for Boolean 3-SAT problems with a clause to variable ratio of 4.3.

SAT problems, none were solved within the timeout period (the software we compare against in section 9.5.3 could not solve any of these problems either).

As the number of threads increases, more space in the hardware memory is needed for variable assignments. For large problems, the number of threads must be reduced in order for the problem to still fit in hardware. Eventually even one thread will no longer fit. For Boolean 3-SAT problems with a clause to variable ratio of 4.3, the maximum problem size supported by our hardware is summarized in Figure 9.3. As we continue to add one more thread, the *incremental* penalty to the capacity decreases.

For different numbers of threads, Figure 9.4 illustrates our BCP performance as a function of the SAT problem size. In small SAT problems, there are fewer constraints and fewer variables to imply. As problem size increases, Figure 9.5 shows that more BCPs are produced per round. One round of BCP is started by software by either assigning a new variable or by reversing one variable assignment after a conflict.

The ratio of clauses to variables remains constant (4.3 in our case) as we increase the problem size, yet the interdependencies between variables expands. A problem

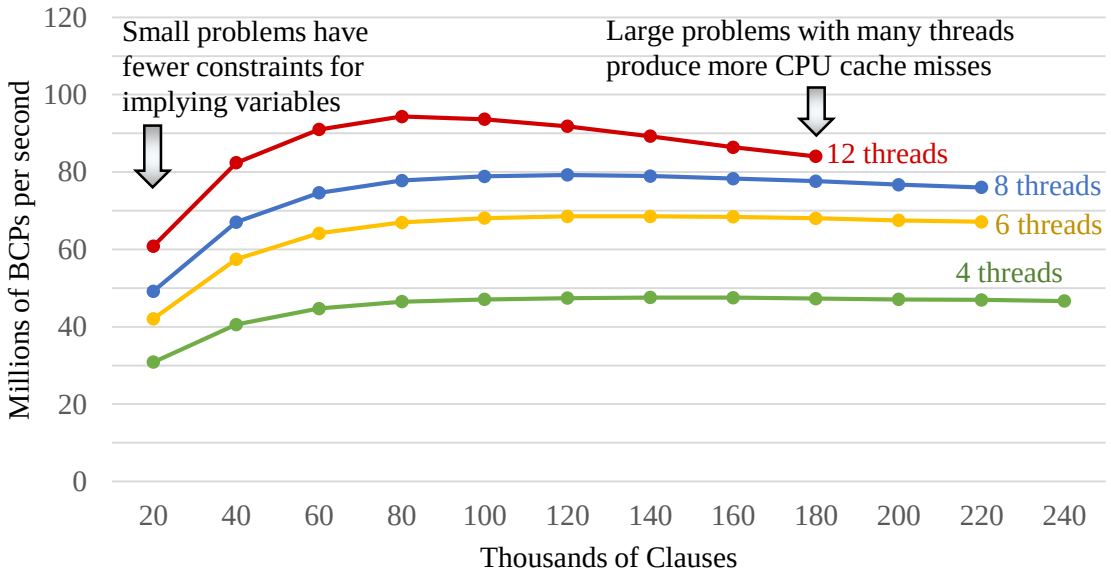


Figure 9.4: BCP performance versus problem size for a varying number of threads. For exact values, refer to Table 9.6.

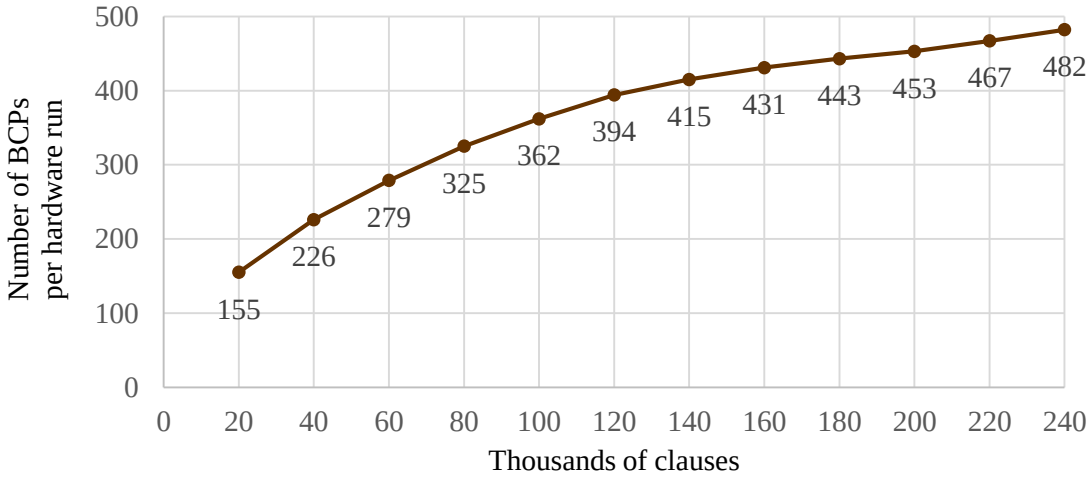


Figure 9.5: Average number of BCPs discovered on each round. One round of hardware BCP starts from one DPLL decision or from reversing one variable assignment after a conflict. As this increases, the communication latency between software and the FPGA becomes better amortized. These results are from 4 threads, however different numbers of threads produce nearly identical results.

with 10 variables contains 43 clauses. Each clause contains exactly 3 variables, thus there are 129 total occurrences. On average each variable occurs in approximately 13 different clauses and can be influenced by 26 other variables. With increasing problem size, these 26 variables become more *diverse* (influenced by a larger set of variables).

Returning to Figure 9.4, this explains the lower performance for smaller problem sizes. With fewer BCPs per round, the hardware/software communication latency is more expensive per BCP due to less amortization.

Increasing the problem size requires more software memory and thus induces more cache misses (the CPU cache size is constant). Software becomes slower as the CPU accesses more data in DRAM. Eventually this outweighs the benefit from better amortization of the hardware/software communication latency. In Figure 9.4, this trend is easily observed with 12 threads, somewhat noticeable with 8 threads, and practically non-existent at 4 threads. Fewer software threads are less capable of thrashing the CPU cache. Also, our hardware cannot fit a large enough SAT problem to induce substantial amounts of CPU cache misses with only 4 threads.

In Figure 9.6, we illustrate the timing profile of our software. Using 12 threads, we vary the problem sizes. We show the average of the absolute times over all 48 random problems at each size. As consistent with Figure 9.5, as the problem size increases:

- More BCPs per round leads to a decrease in the average software polling time (waiting for hardware BCP to finish). The hardware/software communication latency is amortized over more BCPs.

- More time is needed to translate the increased number of BCPs from hardware. Conflict analysis will need to visit more variables on the same decision level (see section 7.3.1) and more variables would be deassigned following a conflict.

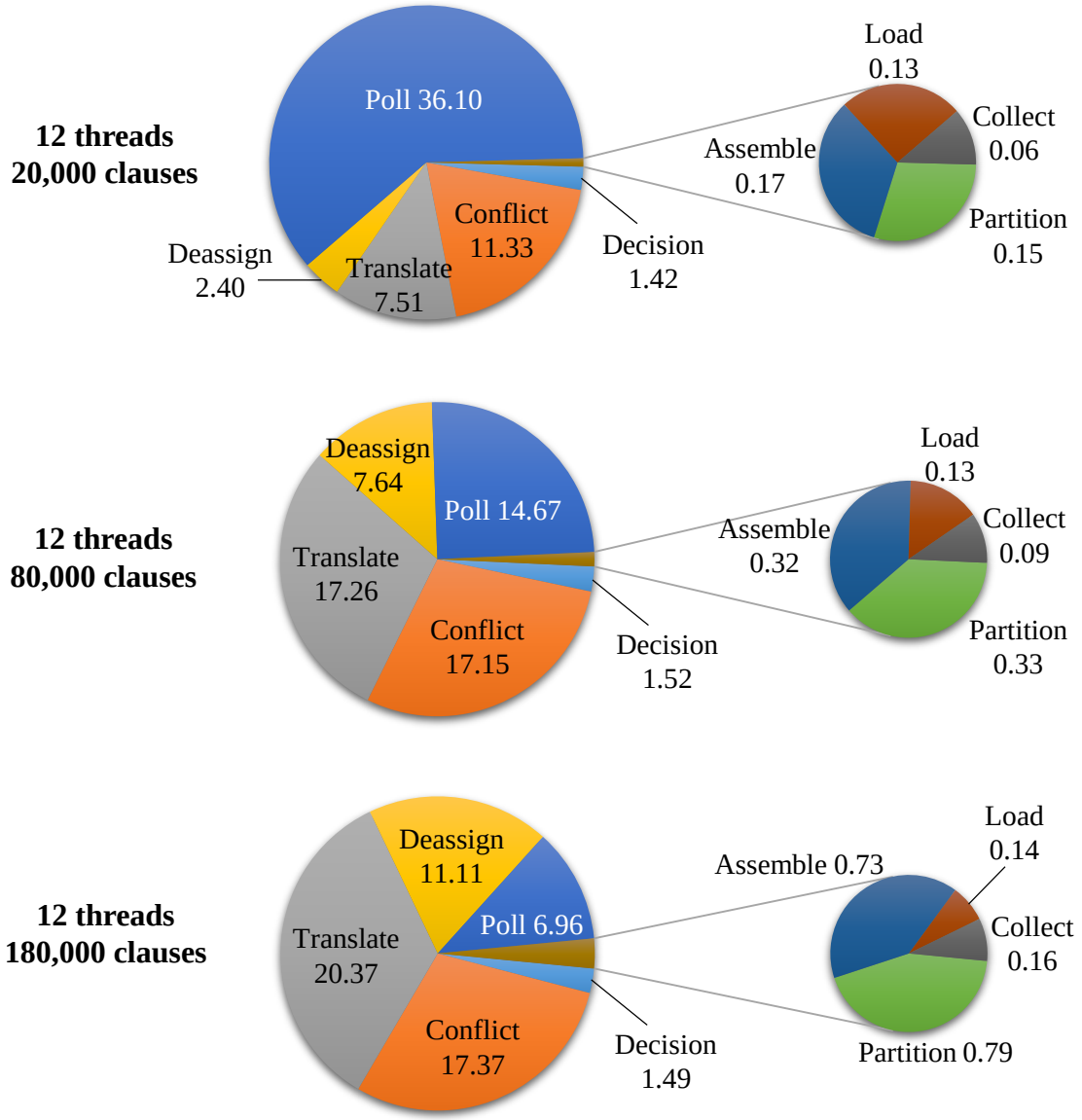


Figure 9.6: Absolute time in seconds is shown for each task in the software timing profiles for different Boolean 3-SAT problem sizes using 12 threads. The SAT solver must make a *decision* on the next variable to assign, perform *conflict* analysis, *translate* BCPs from hardware to update software variable assignments, *deassign* software variable assignments, and *poll* to wait for hardware BCP to finish. We introduce a small overhead to the SAT solver since we must *partition* the clauses, *assemble* this into the hardware memory format, *load* the FPGA, and *collect* learnt clauses between threads.

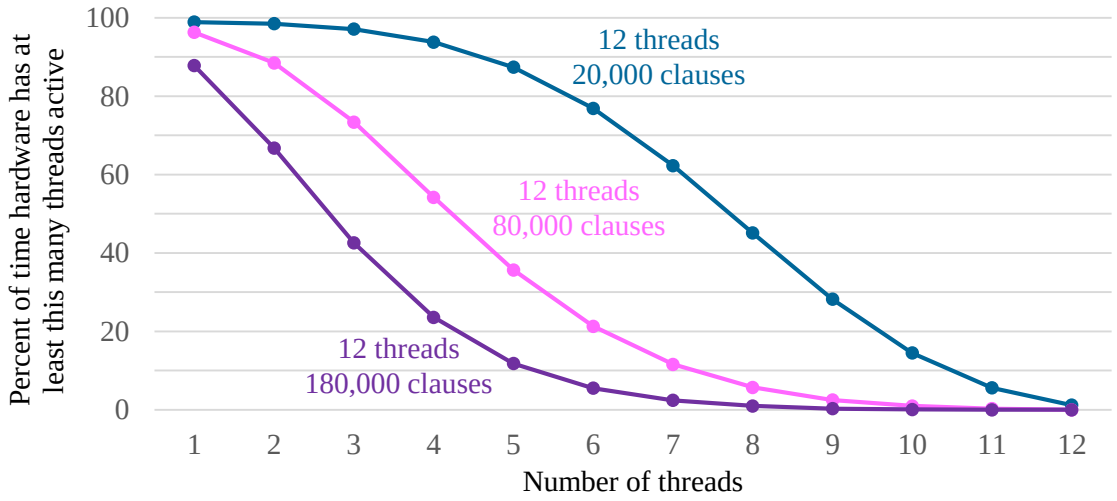


Figure 9.7: Thread occupancy of hardware with 12 threads. This Figure illustrates that for Boolean 3-SAT problems with 80,000 clauses, on 73% of all clock cycles in hardware, at least 3 out of the 12 threads are performing BCP.

A small software overhead of 1-3% is introduced to the SAT solver since we must partition clauses, generate the hardware memory, load the FPGA, and collect learnt clauses between threads. This happens 12 times within 60 seconds.

Using performance counters in hardware, Figure 9.7 illustrates the average thread occupancy for 12 threads. Figure 9.8 shows the average hardware utilization for various problem sizes and number of threads, which provides a broader perspective of the distribution of computation between hardware and software. For small problems, few variables are implied and thus software is relatively fast. By starting the next round of BCP quickly, hardware is kept busy. From Figure 9.7, for 77% of all clock cycles in hardware, BCP is active on at least half of all threads (6 or more out of 12).

As the problem size increases, more memory is needed (causing more CPU cache misses) and more BCPs are produced. Software slows down relative to hardware, thus the utilization in hardware decreases, as shown in Figures 9.7 and 9.8. This is also consistent with the decrease in software polling time in Figure 9.6. Amdahl's Law [93]

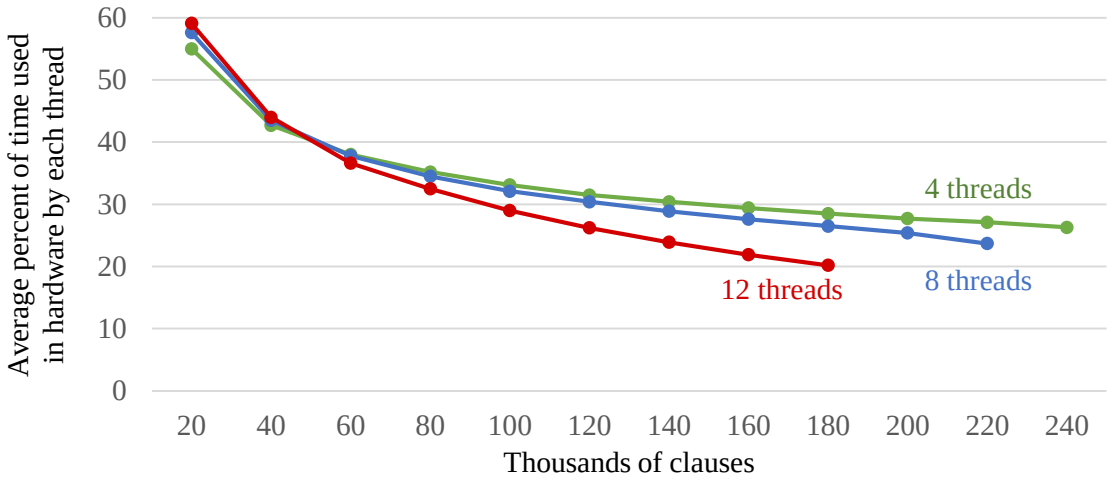


Figure 9.8: Average amount of time that each thread uses in hardware (in percent of all clock cycles, averaged over all threads). For ease of presentation, 6 threads is not shown since it fits in between 4 and 8.

indicates that as hardware further accelerates BCP, software eventually becomes the computation bottleneck, thus starving the FPGA of data. Nonetheless, even at the largest problem size supported by 12 threads, at least one hardware thread is active 88% of the time.

As demonstrated in section 7.2.3, we can perform some computation in hardware and software *concurrently on the same thread*. For example, with 12 threads at 80,000 clauses, Figure 9.8 indicates an average hardware utilization of 33% yet Figure 9.6 shows that software is polling (waiting on hardware BCP to finish) only 25% of the time. Overlapping computation further helps to mitigate the hardware/software communication latency.

Software generally slows down as the number of threads increases. In addition to more CPU cache misses due to needing more memory, the CPU clock frequency decreases as more CPU cores are used in order to limit the total power consumption. We are *already* experiencing dim silicon, as further explained in section 3.1.1.

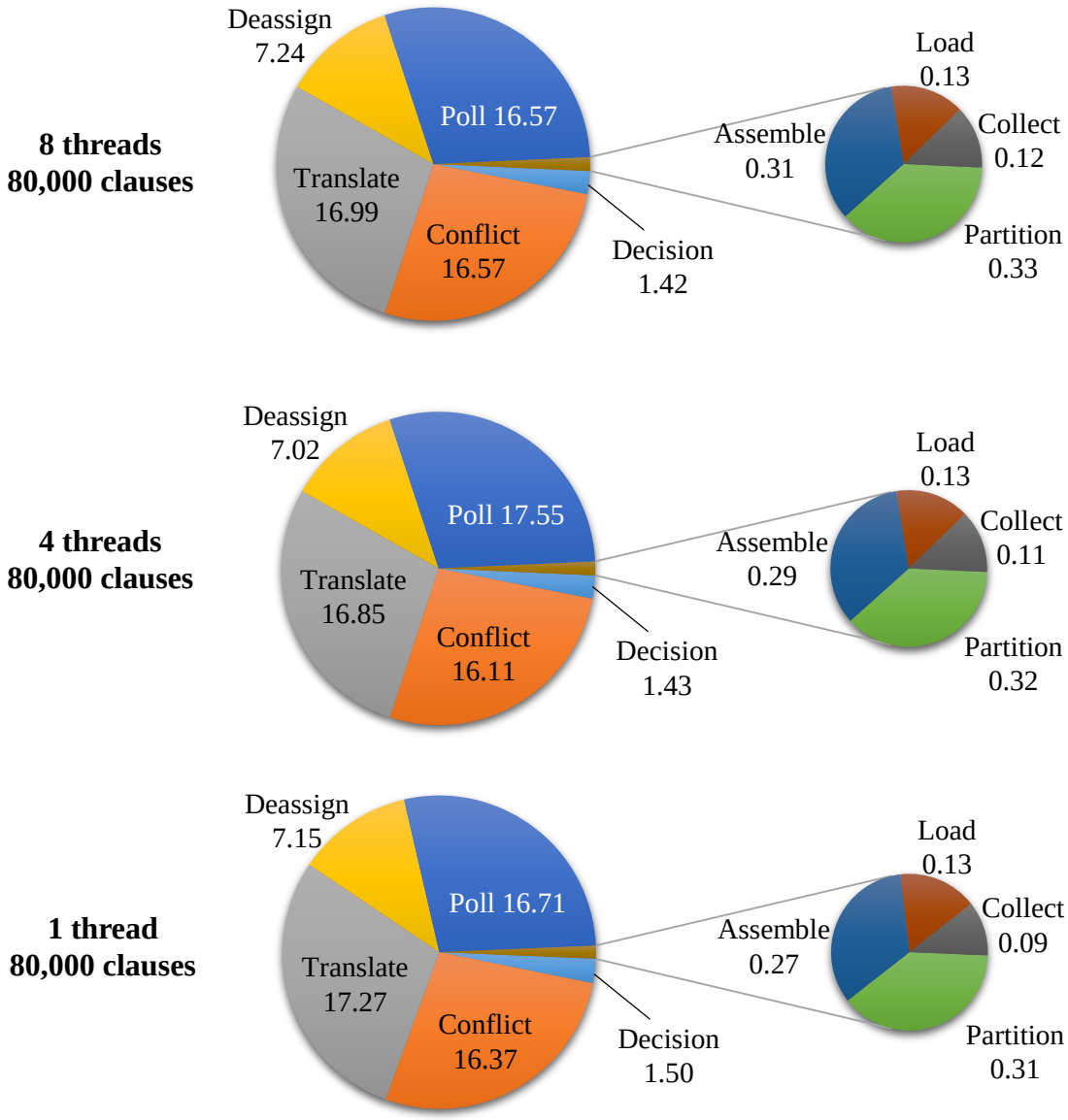


Figure 9.9: Software timing profiles for the same SAT problems with different numbers of threads. The same problem with 12 threads is shown in the middle row of Figure 9.6. The number of threads has practically no effect on the *distribution* of computation.

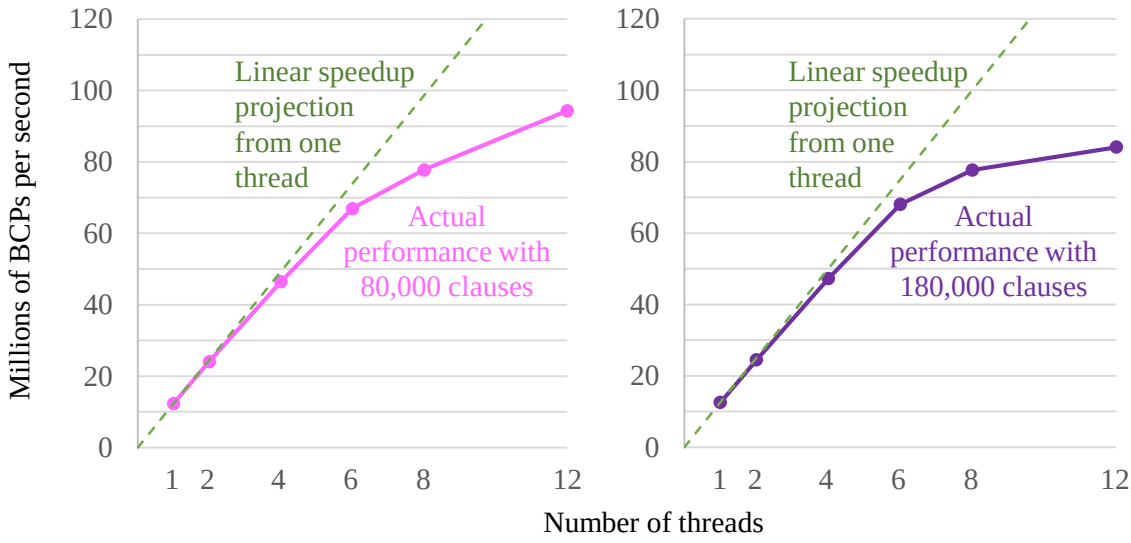


Figure 9.10: BCP rate versus the number of threads on the same SAT problems.

Changing the number of threads has practically no effect on the software timing profile, as illustrated in Figure 9.9. Intuitively, using more cores on a CPU should not affect the distribution of computation, it should only affect the speed.

Figure 9.10 illustrates our BCP rate on the same SAT problems as the number of threads increase. Each thread may become slower, yet collectively more threads produce a higher total computational throughput. With up to 6 threads, the speedup is nearly linear. Power limitations force the CPU cores to slow down with more software threads, thus the speedup is strictly less than linear. For the same SAT problems, more software threads induce more CPU cache misses, hence the larger gap between the projected and actual performance on the right side of Figure 9.10.

9.5.3 Comparison Against Multithreaded Software

We benchmark against Plingeling [94] and Minisat [37]. Plingeling is the parallel (multithreaded) version of Lingeling [39], and Lingeling is the successor of Picosat [38].

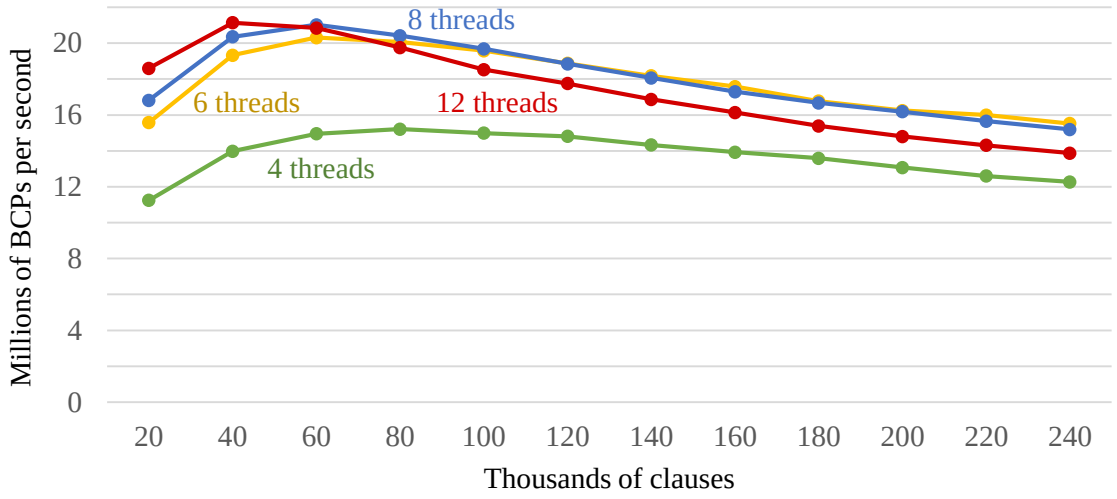


Figure 9.11: BCP performance of Plingeling.

Plingeling and its related variants won several categories in the SAT 2013 competition [33]. Minisat won the applications category of the SAT 2005 competition [33]. Minisat is such a well established SAT solver that many recent SAT solvers are built on top of Minisat. One example is Glucose [95], which won various categories in the SAT 2013 competition. Glucose shares *exactly the same BCP implementation* as Minisat since its improvement comes from better management of the learnt clauses database.

Minisat is a single-threaded solver, so we approximate its multithreaded performance by running several instances of it at the same time. This is exactly the behavior of a multithreaded portfolio-based SAT solver, which races 4 individual SAT solvers against each other on the same SAT problem given 4 CPU cores, for example.

Using the same random SAT problems, the BCP performance of Plingeling and Minisat are shown in Figures 9.11 and 9.12 respectively. They have the same trend of lower performance on larger problem sizes (same reasons as for our hardware BCP).

Plingeling has data structures specialized for size 3 clauses whereas Minisat does not. With more efficient BCP, Plingeling is faster and also exhibits lower BCP performance

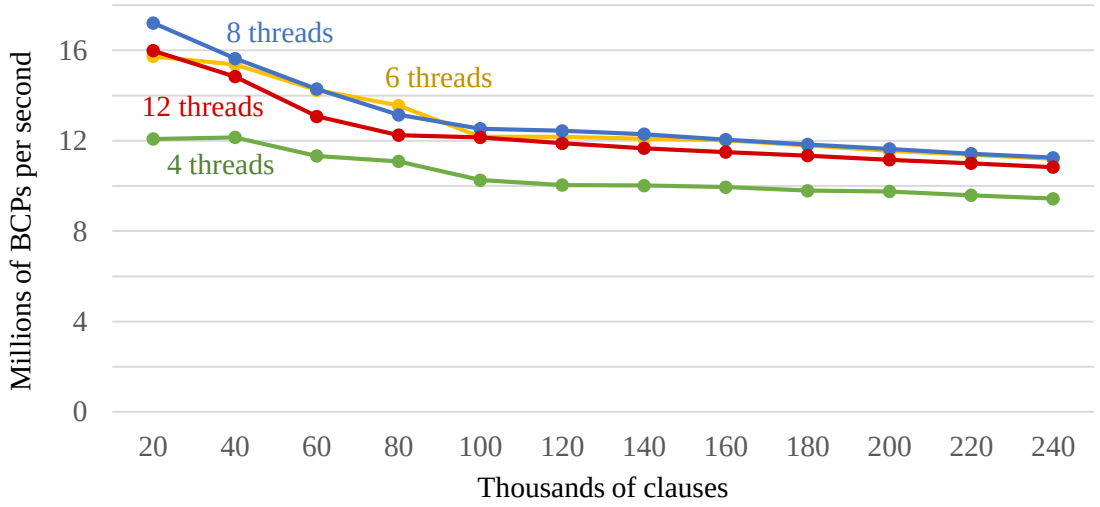


Figure 9.12: BCP performance of Minisat. Note that Minisat is a single-threaded solver, and we estimate the multithreaded performance by running several instances concurrently.

for small problems due to the lower number of BCPs per round. Conversely, Minisat seems to be limited by memory bandwidth even at small problems, so we only observe the trend of the lower performance on larger problems.

Interestingly, the performance degrades when too many threads are used. The performance is typically best with 6 to 8 threads (neither outperforms the other across all problem sizes). This is most likely due to the limited bandwidth of the CPU's shared memory. Once the DRAM bandwidth is saturated, adding more threads will only further thrash the CPU cache, thus decreasing the overall performance. Our hardware BCP performance improved with 12 threads, which suggests that increasing the number of threads outweighs any further decrease in the CPU clock frequency due to total power limitations.

Software BCP is memory intensive, thus offloading BCP to hardware raises the threshold of the maximum number of threads that can be used before the CPU encounters the "memory wall". It also delays the onset of lower performance for larger

Table 9.7: BCP speedup over the best performance of multithreaded software for a varying number of hardware threads. Each is averaged over all problem sizes.

	2 threads	4 threads	6 threads	8 threads	12 threads
Plingeling	1.3×	2.5×	3.5×	4.0×	4.5×
Minisat	1.8×	3.5×	4.9×	5.6×	6.4×

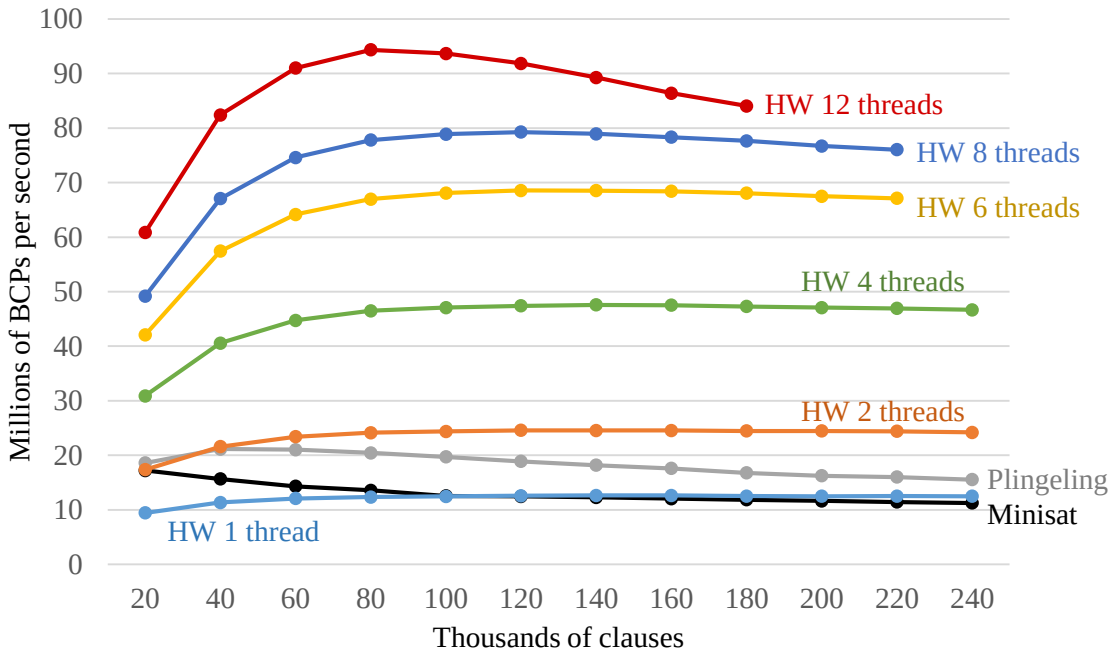


Figure 9.13: Our hardware BCP performance versus multithreaded software. The best performance using 4, 6, 8, or 12 threads is shown for Plingeling and Minisat.

problems. Our hardware BCP performance peaks at 80,000 clauses whereas Plingeling peaks at 40,000 clauses.

Figure 9.13 shows the difference in BCP performance between our heterogeneous compute system versus multithreaded software (CPU only). For Plingeling and Minisat, we show the best performance obtained using 4, 6, 8, or 12 threads. For some problem sizes, 6 threads maximizes the BCP performance, whereas for other problem sizes, 8 threads may produce the best result, or sometimes even 12 threads.

Table 9.7 summarizes our speedup over software. We are nearly at the limit of Amdahl's Law [93]. Software BCP typically takes 80-90% of the CPU time, so in order to further improve the speedup beyond about 5×, we would need to accelerate more than just BCP in hardware.

From an alternative perspective of energy efficiency, notice that with only 1 thread our BCP performance is comparable to the best performance of Minisat. With 2 threads, we outperform both software approaches. Typically FPGAs operate at 20-30 watts whereas a CPU consumes around 100 watts at a full work load. By significantly reducing the CPU work load, we may *potentially* decrease its power consumption by more than that of the FPGA, thus resulting in a net decrease in total power. Unfortunately we do not have power measurement equipment to validate this.

9.6 Randomized Enhanced BCP Benchmark

9.6.1 Benchmark Description

In this section, we explore how the presence of many logical constructs within the SAT problems affect BCP performance. To be consistent with the previous 3-SAT benchmark, we also used 48 random instances for each problem size and a 1 minute timeout. Except for Boolean and XOR clauses, all other enhanced BCP clause types may imply several variables per clause. For example, assigning the output of a NOR gate to true implies all inputs as false. Logical constructs typically produce more BCPs than Boolean clauses for the same problem size, as confirmed by Figure 9.14.

Due to the extensive use of lossless compression, the enhanced BCP benchmark uses significantly larger Boolean SAT problems than the 3-SAT benchmark. Figure

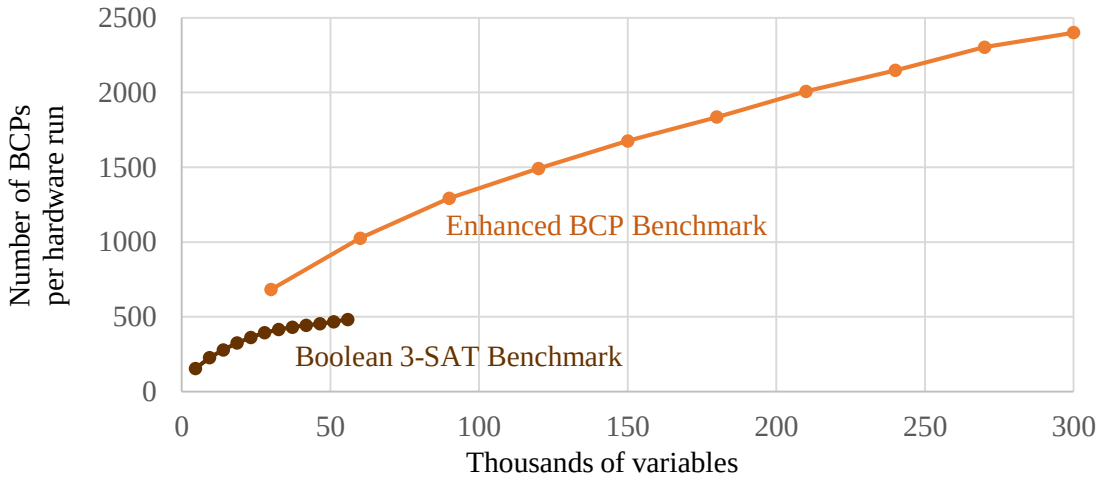


Figure 9.14: With enhanced BCP, we benchmarked significantly larger SAT problems. Logical constructs also tend to cause more constraint propagation, as many clause types are capable of implying multiple variables.

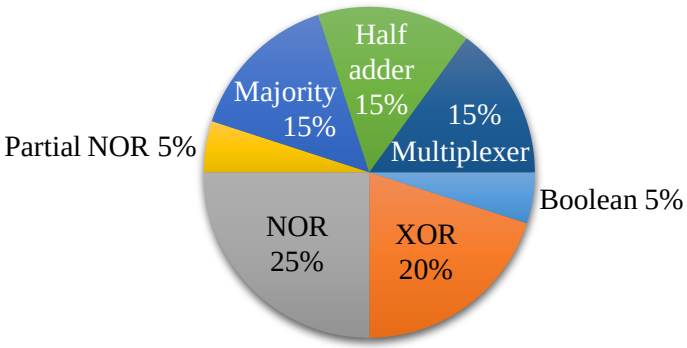


Figure 9.15: Distribution of the enhanced BCP clause types used in our benchmark.

Table 9.8: Distribution of the enhanced BCP clause sizes used in our benchmark.

Clause size	Occurrence based on enhanced BCP clause type		
	Boolean, XOR	NOR, Partial NOR	Others
2	5%		
3	45%	50%	
4	30%	30%	100%
5	10%	10%	
6	4%	4%	
7 to 12	1% each	1% each	

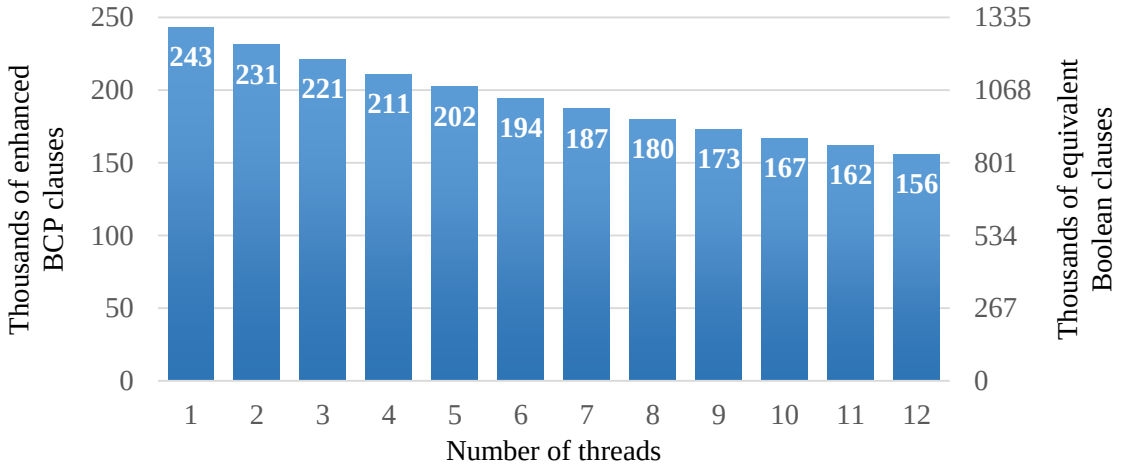


Figure 9.16: Maximum problem size that our hardware supports for our chosen distributions of enhanced BCP clause types and sizes.

9.14 also illustrates this. We used the number of variables to estimate the problem size instead of trying to compensate for clause compression due to enhanced BCP.

Figure 9.15 and Table 9.8 summarize the distribution of clause types and clause sizes respectively. With these distributions, each enhanced BCP clause is equivalent to about 5.34 Boolean clauses (Tseitin encodings were illustrated in section 2.2.1). Variables within each clause were randomly chosen using a uniform distribution.

Few Boolean and Partial NOR clauses are used since these are essentially leftover constraints that did not belong to a logical construct. Real-life SAT problems that contain logical constructs typically have small clauses. For instance, a variable v may depend on 8 other variables, however unless all of those 8 variables affect v in exactly the same way, we cannot combine them into e.g. a single 8-input NOR gate.

Figure 9.16 reports the maximum problem size supported by our hardware. We fit fewer absolute number of clauses than the 3-SAT benchmark since the average clause size has increased to around 4. However, enhanced BCP clauses are a form of lossless compression, so the equivalent number of Boolean clauses is significantly larger.

Table 9.9: Number of SAT problems with 20,000 clauses solved within 1 minute. Results are shown using the best number of threads for each system.

	Our system (12 threads)	Plingeling (8 threads)	Minisat (1 thread)
Satisfiable problems solved	10	7	5
Unsatisfiable problems solved	6	6	6
Total problems solved	16	13	11

We attempted to create critically constrained SAT problems (see Figure 9.2) since there is little practical value in accelerating easy SAT problems. This was discussed in section 9.5.1. Given the above distributions, we experimentally found that a clause to variable ratio of 2/3 (e.g. 100 enhanced BCP clauses with 150 variables) produces a similar number of satisfiable and unsatisfiable problems for small problem sizes. However, unlike the previous benchmark which only used Boolean clauses with exactly 3 variables each, the enhanced BCP benchmark considers several different clause types and sizes. Due to this distribution, it is unlikely that every randomly generated problem will be critically constrained.

To demonstrate this, at our smallest problem size of 20,000 enhanced BCP clauses, both we and software were able to solve some of the problems within the 1 minute time limit, as summarized in Table 9.9. Conversely, no SAT problems were solved in the Boolean 3-SAT benchmark. As BCP gets faster we expect to solve more problems within the same amount of time, as our system has demonstrated.

All 3 solvers solved the same 6 unsatisfiable problems, which requires pruning the entire DPLL tree search. For satisfiable problems however, finding any satisfying assignment is sufficient. The incremental difficulty to solve the next hardest problem happened to be lower for satisfiable problems in our randomly generated benchmark. In general, the shape of Figure 9.2 is likely not symmetrical as qualitatively illustrated.

Table 9.10: Detailed performance of our system on the enhanced BCP benchmark.

Problem specification		Millions of BCPs per second			
Clauses	Variables	4 threads	6 threads	8 threads	12 threads
20,000	30,000	77.76	110.49	123.93	137.60
40,000	60,000	81.30	115.79	123.07	116.47
60,000	90,000	81.34	113.14	114.97	103.17
80,000	120,000	79.65	106.19	104.67	82.77
100,000	150,000	76.69	98.65	95.97	76.91
120,000	180,000	71.33	92.03	87.47	66.98
140,000	210,000	71.15	83.94	80.25	61.11
160,000	240,000	66.66	76.60	73.17	
180,000	270,000	63.50	70.81	67.28	
200,000	300,000	58.47			

9.6.2 Performance Analysis of Enhanced BCP

Detailed performance using enhanced BCP is shown in Table 9.10. The results are also illustrated later in Figure 9.24. We used 48 random instances per problem size each with a 1 minute timeout. With many threads, our results suggest that we have already passed the threshold of problem size where more BCPs per round will improve the performance. Enhanced BCP uses larger SAT problems than in the 3-SAT benchmark, and we only observe the trend of lower performance on larger problems due to more CPU cache misses. Furthermore, once the CPU memory bandwidth is saturated, using more threads will thrash the CPU cache which decreases the overall BCP performance. Notice that typically 8 threads performs better than 12 threads.

With fewer threads, the CPU's shared memory is not as stressed, hence larger problems are required before we finally observe a decrease in performance. In the 3-SAT benchmark, we could not fit a large enough problem in hardware to exercise this with 4 threads (only about 55,000 variables would fit). With enhanced BCP, Table 9.10 shows that the performance degrades after 90,000 variables with 4 threads.

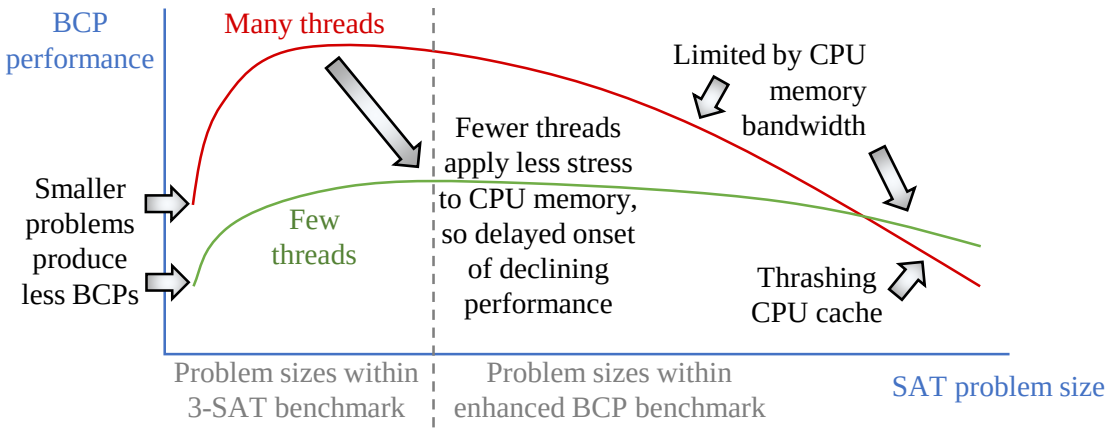


Figure 9.17: Amalgamation of trends between the Boolean 3-SAT and enhanced BCP benchmarks. More BCPs are produced as the problem size increases, which eventually limits the BCP performance due to the CPU's memory bandwidth. This onset is delayed with fewer threads. This is a *qualitative* illustration and is not drawn to scale.

Enhanced BCP somewhat provides a preview of what *may* happen if we could fit larger Boolean 3-SAT problems in the FPGA. This concept is sketched in Figure 9.17.

With larger problems, hardware is active less often and software spends even less time polling for hardware BCP to finish. Figure 9.18 illustrates the average time utilization in hardware. This is significantly lower than in Figure 9.8 for Boolean 3-SAT. Figure 9.19 shows the average percent of time that at least one thread is active in hardware. Low values indicate that software is the computation bottleneck and thus it cannot supply work quickly enough for the FPGA. This is consistent with the software timing profiles in Figure 9.20, which indicate software is spending very little time waiting for hardware BCP to finish (only 3% of the time for 180,000 clauses with 8 threads). The software timing trends in Figure 9.20 (for enhanced BCP) are similar to Figure 9.6 (for Boolean 3-SAT), where larger SAT problems cause more BCPs per round. This better amortizes the software/hardware communication latency, yet more CPU time is needed for translating hardware BCPs and for deassignment.

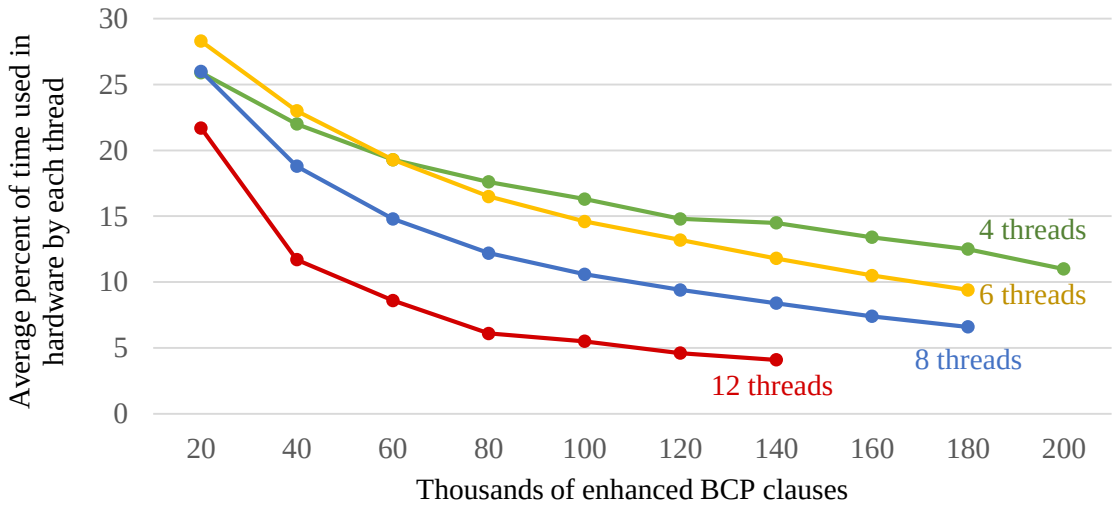


Figure 9.18: Average amount of time that *each* thread uses in hardware (in percent of all clock cycles, averaged over all threads). This largely depends on how fast software can supply work to the FPGA. Software slows down with more threads and larger problem sizes.

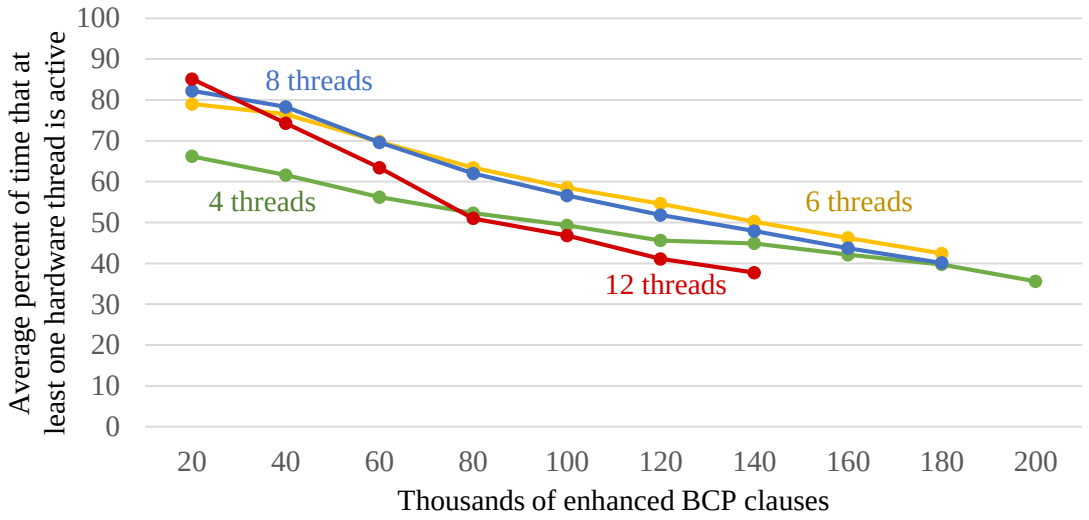


Figure 9.19: Average amount of time that *at least one* hardware thread is active (in percent of all clock cycles). Increasing the number of threads provides a higher chance of *some* thread being used, however too many threads will slow down software and thus starve the FPGA of work. 100% minus each value indicates the amount of time that hardware is idle (absolutely no processing).

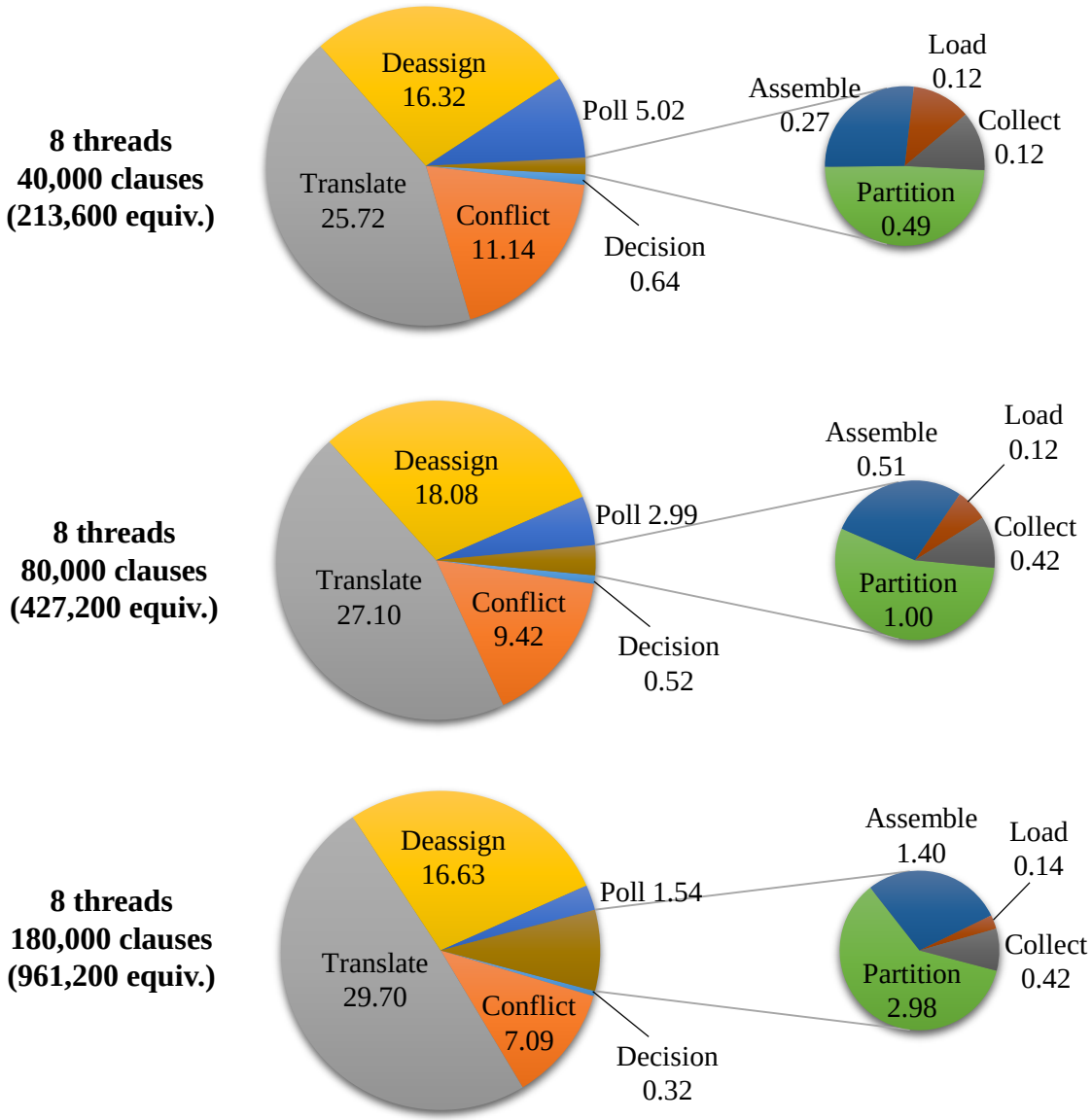


Figure 9.20: Software timing profiles for enhanced BCP with 8 threads. Each enhanced BCP clause is equivalent to about 5.34 Boolean clauses. Larger problems produce more BCPs, hence more CPU time is needed to translate BCPs from hardware and update software variable assignments (both assigning and deassigning).

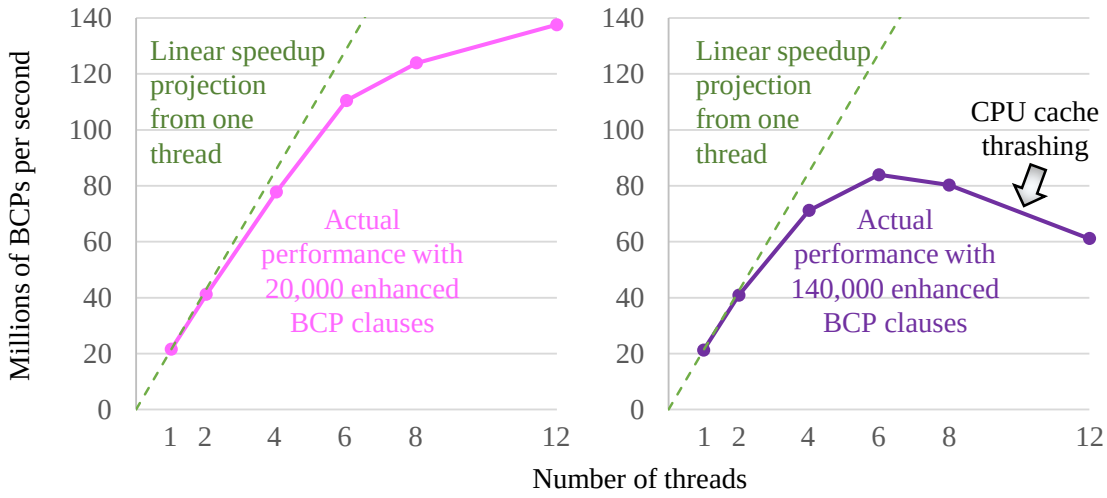


Figure 9.21: BCP rate on the same problems with varying number of threads. Enhanced BCP enables us to use large enough problems to observe a decrease in performance with more threads (partly due to thrashing the CPU cache).

The enhanced BCP benchmark uses more CPU time for partitioning compared to the 3-SAT benchmark for a similar absolute number of clauses. Enhanced BCP clauses have more variables per clause, and the variables are more diverse (the total number of variables in each SAT problem is significantly larger). Recall from section 8.1.3 that this would create a larger set of "priority clauses" to choose from, thus more "net benefit" values must be computed. The overhead of all non-SAT solver tasks (the smaller pie charts on the right side of Figure 9.20) has risen to as high as 8%.

Although not shown, the distribution of software time spent on each task does not change significantly as the number of threads is varied (on the same SAT problems). This trend is consistent with the Boolean 3-SAT benchmark (see Figure 9.9).

Finally, Figure 9.21 shows the performance for the same SAT problems as the number of threads is varied. For small problems, the trend is similar to Figure 9.10 from the 3-SAT benchmark. However for large problems, eventually more threads results in lower BCP performance due to thrashing of the CPU cache.

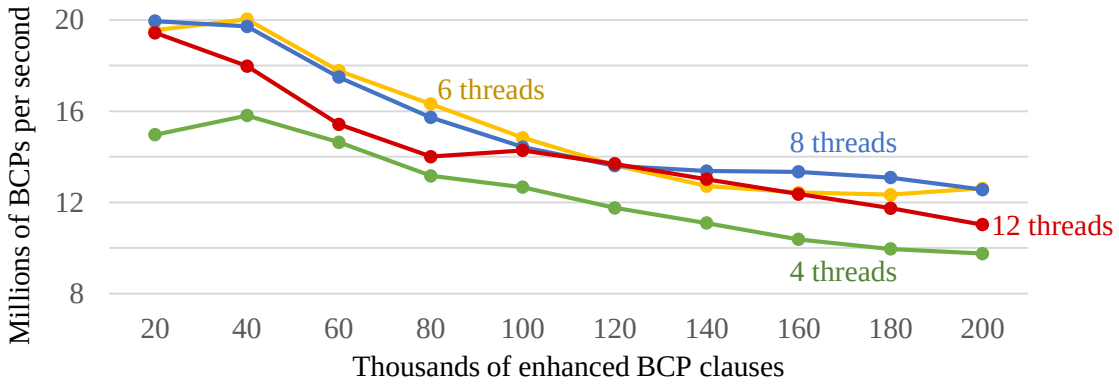


Figure 9.22: BCP performance of Plingeling. The range of the BCP rate has been compressed to highlight the difference between using different numbers of threads.

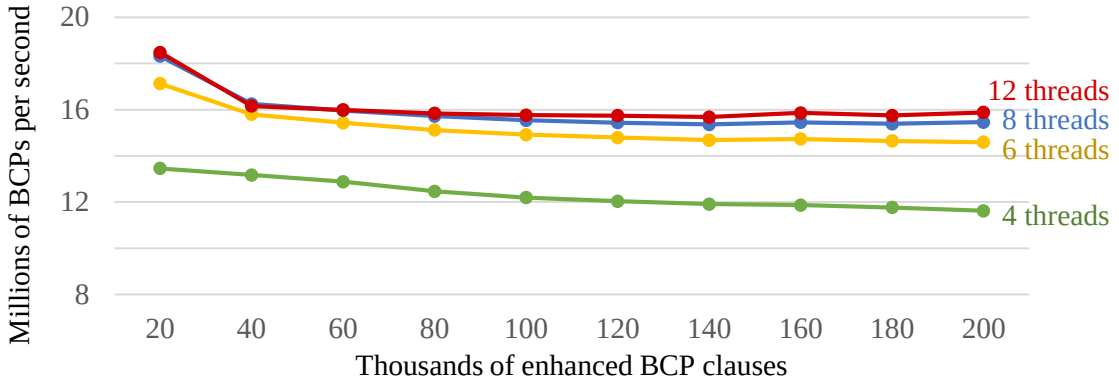


Figure 9.23: BCP performance of Minisat. The range is identical to Figure 9.22.

9.6.3 Comparison With Multithreaded Software

We converted the same random enhanced BCP problems into Boolean SAT problems for a comparison against multithreaded software. The BCP performance of Plingeling and Minisat are shown in Figures 9.22 and 9.23 respectively. Plingeling has specialized data structures for size 2 and size 3 clauses whereas Minisat does not, hence the peak BCP rate of Plingeling is larger. However, Plingeling also shares learnt clauses between threads whereas Minisat does not. With no thread synchronization overhead, Minisat eventually performs BCP faster than Plingeling as the problem size increases.

Table 9.11: BCP speedup over the best performance of multithreaded software for a varying number of hardware threads. Each is averaged over all problem sizes.

	1 threads	2 threads	4 threads	Ideal (4, 6, 8, or 12 threads)
Plingeling	1.4×	2.6×	4.7×	6.2×
Minisat	1.3×	2.5×	4.5×	6.0×

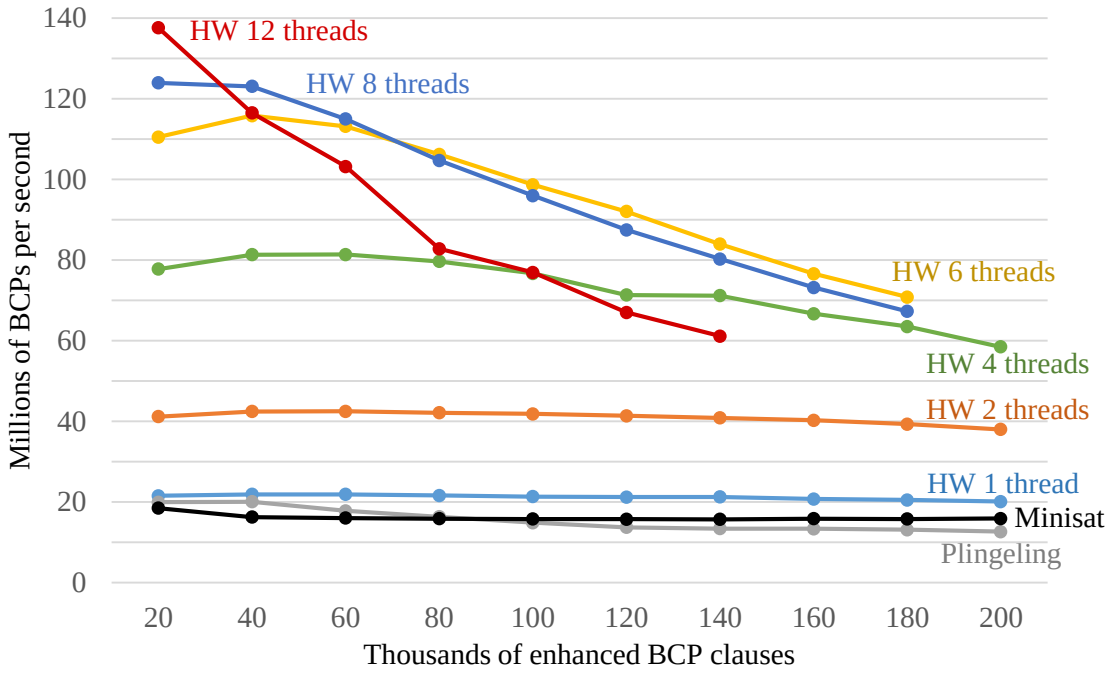


Figure 9.24: Our hardware BCP performance versus multithreaded software. For Plingeling and Minisat, we show the best performance using 4, 6, 8, or 12 threads.

Figure 9.24 shows our BCP performance versus the best performance of pure software. Unlike the 3-SAT benchmark, using 12 hardware threads does not always produce the best performance. With sufficient profiling, we could estimate the best number of threads to use for a given SAT problem. Using this approach, we obtain a 6× speedup over the best performance of software. Table 9.11 also illustrates the speedup using fewer hardware threads. Since we are faster with only 1 thread, this also opens an opportunity for energy efficiency, as discussed at the end of section 9.5.3.

9.7 SAT Competition 2013 Applications Benchmark

To demonstrate the practicality of our system, our final benchmark uses SAT problems derived from real-life applications. The SAT research community annually hosts an international competition [33] to establish the performance of the state-of-the-art software SAT solvers. SAT problems are categorized into the following benchmarks:

1. Industrial: SAT problems derived from real applications.
2. Hand-crafted: SAT problems with properties that make them difficult for SAT solvers to solve, typically these are combinatorial-based problems.
3. Random: this benchmark is typically the most critically constrained.

Our previous two benchmarks used artificial SAT problems to examine the trends of how problem size independently affects many factors, including the BCP rate. These types of problems are also useful for empirically examining the strengths and weaknesses of different SAT heuristics (e.g. how to choose the next variable to assign).

In this benchmark, the SAT problems come from various applications which have *diverse characteristics* (e.g. ratio of variables to clauses, average clause size). Having already established many trends from previous benchmarks, we now focus on the overall performance and our ability to support practical SAT problems in hardware.

9.7.1 Fitting into Hardware Memory

As demonstrated in section 3.3.2, we have to use SRAM in order to be significantly faster than software. Higher density memory, such as DRAM, has poor performance on random access. FPGAs have limited amounts of on-chip SRAM, thus limiting the maximum problem size that we can support.

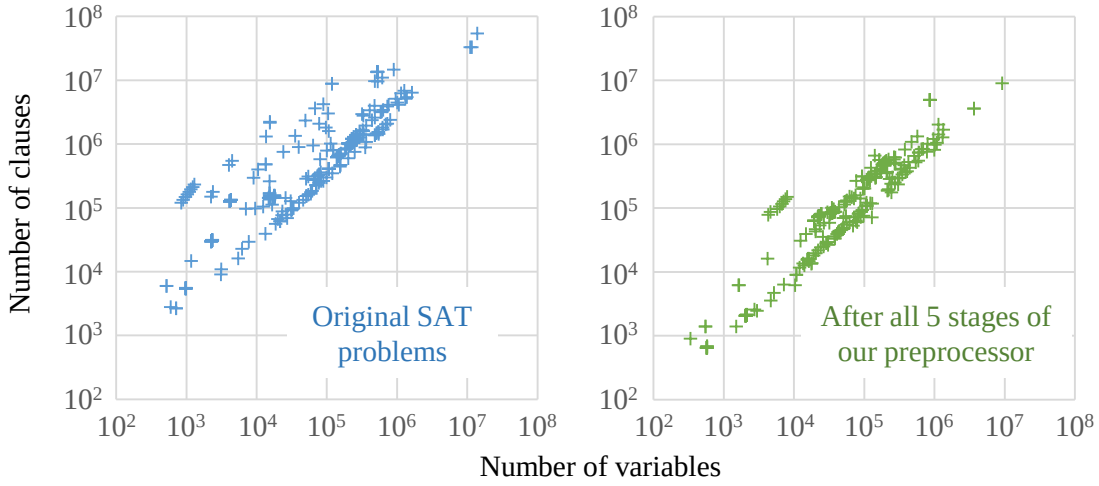


Figure 9.25: The sizes of the 300 SAT problems in the SAT Competition 2013 applications benchmark. Our preprocessor applies many compaction techniques, including the extraction of enhanced BCP clause types.

The SAT Competition 2013 applications benchmark contains 300 SAT problems. Figure 9.25 shows that the problem sizes vary greatly. SAT is an NP-complete problem, thus even with only 1000 variables, the search space of 2^{1000} is extremely large. Even so, large problems are solvable in practice, especially if they are not critically constrained (see Figure 9.2). To find a satisfying solution within a reasonable amount of time, chances are the problem contains a sufficiently large number of satisfying assignments. To prove unsatisfiability within a reasonable amount of time, chances are a sufficiently small subset of the variables actually participate in the contradiction.

Table 9.12 shows the number of SAT problems that fit in hardware. With more threads, the largest problems can no longer fit into our FPGA due to thread-replicated variable assignments in our memory layout. Preprocessing, enhanced BCP, and macro clauses each provide different forms of lossless compression. Our FPGA has about 40 Mbits of on-chip SRAM, or 2 million locations with 20 bits each, yet the largest problem that we could fit contained over 3.6 million clauses before preprocessing.

Table 9.12: Statistics for the problems from the SAT Competition 2013 applications benchmark that we were able to fit into hardware memory. Largest problem sizes show the number of original clauses and variables (before preprocessing).

Threads	Number of SAT problems that fit	Largest problem by number of clauses	Largest problem by number of variables
1	227 (76%)	3,601,247 clauses 67,300 variables	2,384,932 clauses 795,369 variables
2	223 (74%)	same as above	same as above
4	216 (72%)	same as above	same as above
6	206 (69%)	same as above	same as above
8	198 (66%)	same as above	same as above
12	189 (63%)	same as above	1,424,299 clauses 542,367 variables

Table 9.13: Number of partitions used for all 300 of the SAT Competition 2013 applications problems. The average is skewed by a few very large problems, hence we also show the median. Each partition contains 8192 memory locations.

	Average		Median	
	4 threads	12 threads	4 threads	12 threads
All optimizations	283.2	406.7	65	93
Naïve partitioning	336.8	460.9	75	104
No macro clauses	407.2	779.0	95	181
No enhanced BCP	539.7	767.8	101	148
No optimizations	845.7	1463.3	173	322

9.7.2 Estimation of Memory Compaction Performance

We *intentionally handicapped* our system by removing enhanced BCP, macro clauses, and our partitioning algorithm in order to measure how much compaction each of these provide. Since some partitioning is required, we compare against a naïve approach where clauses are added to the partitions in the order that they appear in the SAT problem. This is still a bottom-up approach like our algorithm (section 8.1), however our algorithm uses the "net benefit" to select which clause to add on each iteration.

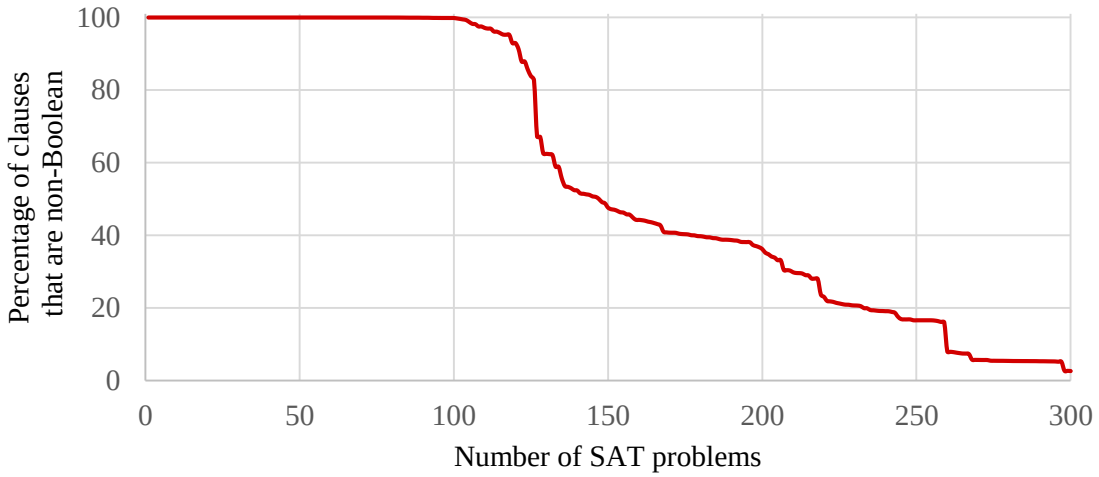


Figure 9.26: Utilization of enhanced BCP. We sorted each problem in the SAT Competition 2013 applications benchmark based on the percentage of logical constructs it contains. Half of the problems each contain at least 47% non-Boolean clauses.

Using all 300 problems from the SAT Competition 2013 applications benchmark, the average number of partitions is shown in Table 9.13. The average is skewed by a few large SAT problems, thus we also show the median. By using naïve partitioning, typically more global links are used, which results in 10-20% more memory needed overall. The number of global links is left to chance, as fewer will be used if a SAT problem happens to often contain the same variables within adjacent clauses.

Macro clauses (section 6.2.6) pack several clauses together to recover wasted space in the variable assignments from small clauses (see Figure 6.3). As expected, this has a larger impact with more threads. From Table 9.13, disabling this with 12 threads results in a 95% overhead compared to only a 45% overhead for 4 threads.

Enhanced BCP (section 6.1) uses only one clause to represent a set of Boolean clauses that form a logical construct. The amount of compression depends on how many logical constructs are in the SAT problems, which is illustrated in Figure 9.26. From Table 9.13, enhanced BCP overall provides about a 55% reduction in memory.

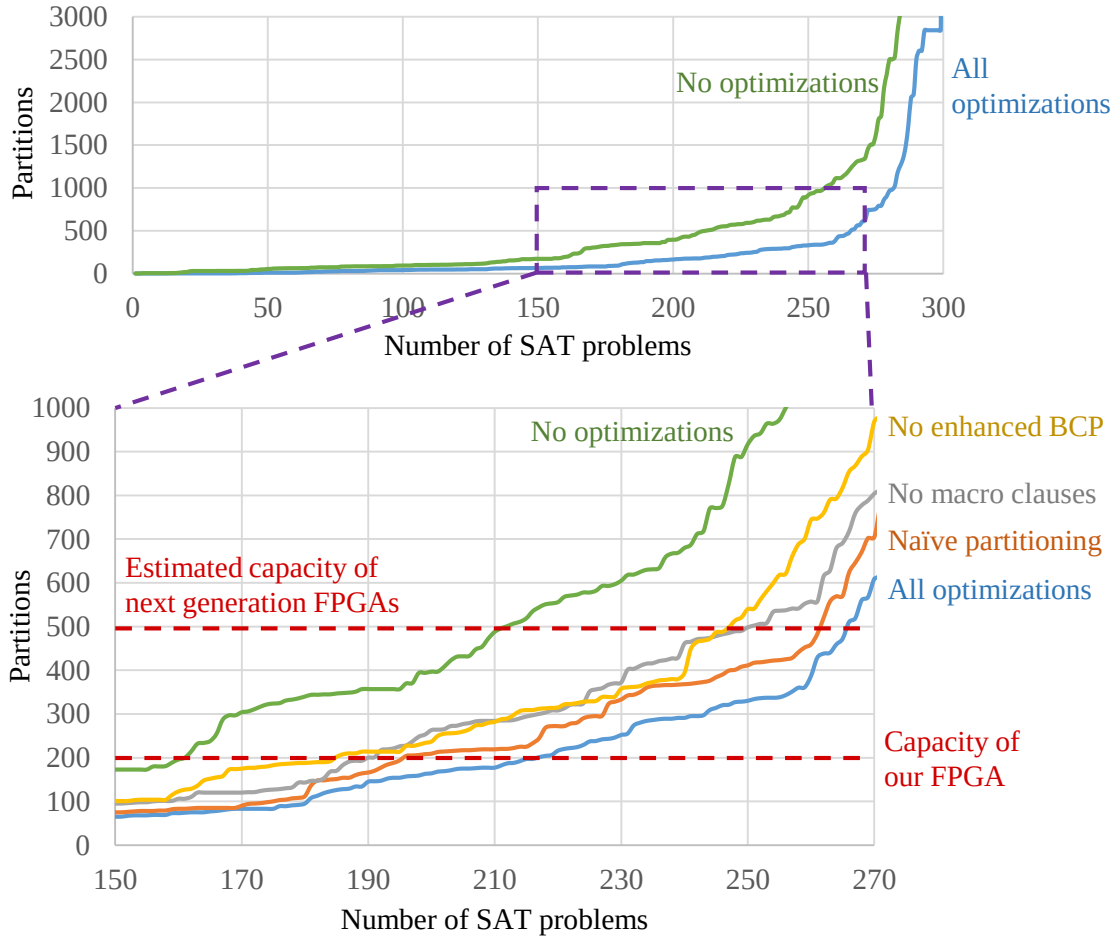


Figure 9.27: We sorted each problem in the SAT Competition 2013 applications benchmark based on the number of partitions needed for 4 hardware threads. With each handicap, fewer SAT problems fit within the same number of partitions.

Figures 9.27 and 9.28 illustrate the severity of disabling each optimization for 4 threads and 12 threads respectively. Given a fixed number of partitions in hardware memory, fewer SAT problems will fit if any of our optimizations are removed. Naïve partitioning induces some extra global links, however not nearly as much as the wasted variable assignment space that can be recovered by packing small clauses together. Since the variable assignments are replicated per thread, this wasted space eventually outweighs the effect of enhanced BCP as the number of threads increases.

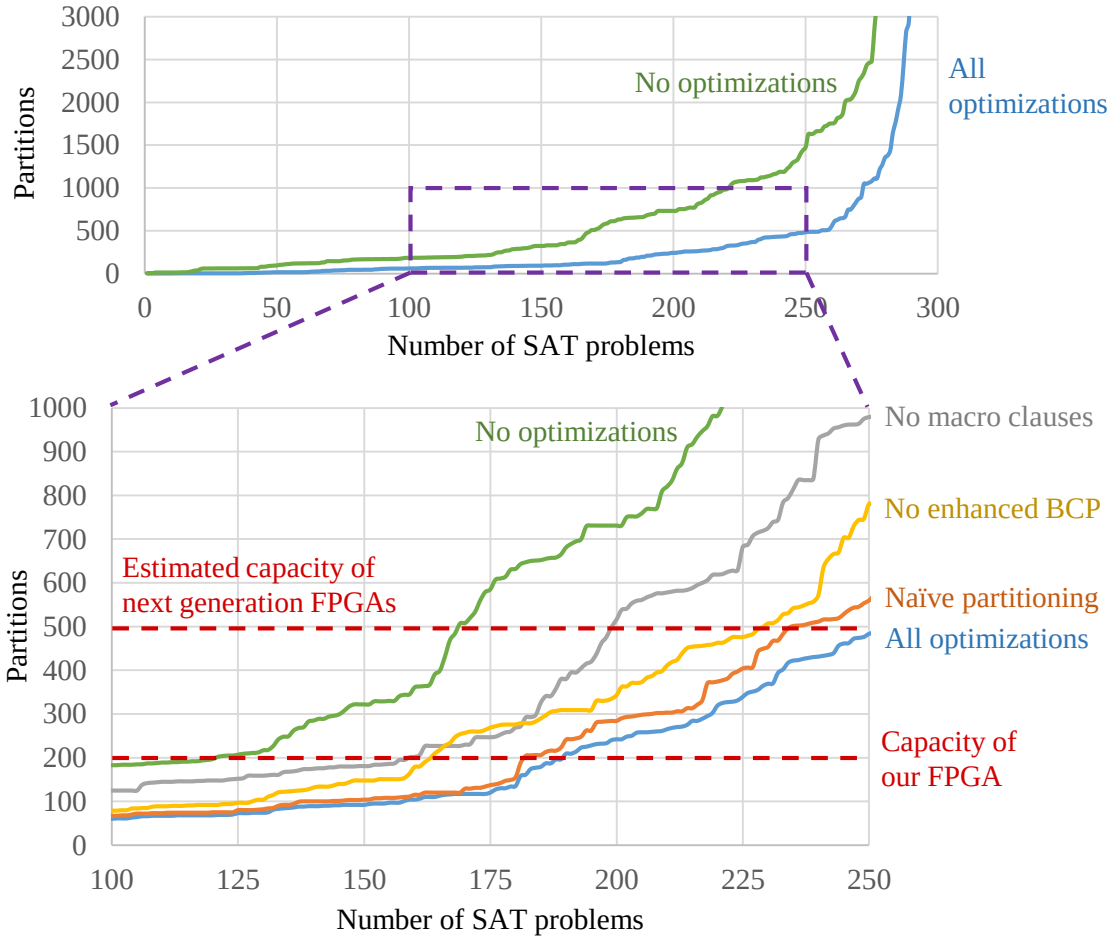


Figure 9.28: Number of partitions needed for different handicaps. Similar to Figure 9.27 but now with 12 threads. Note that the zoomed-in window is in a different place.

Table 9.14: Average number of clauses per partition. Each partition contains 8192 memory locations. Clauses are already preprocessed but not yet multiply packed. Enhanced BCP reduces the number of preprocessed clauses by about 65%, thus we also show a normalization where appropriate for a fairer comparison.

	4 threads	12 threads
All optimizations	$1363 \times 1.65 = 2249$	$945 \times 1.65 = 1559$
Naïve partitioning	$1175 \times 1.65 = 1939$	$848 \times 1.65 = 1399$
No macro clauses	$914 \times 1.65 = 1508$	$475 \times 1.65 = 784$
No enhanced BCP	1402	976
No optimizations	828	450

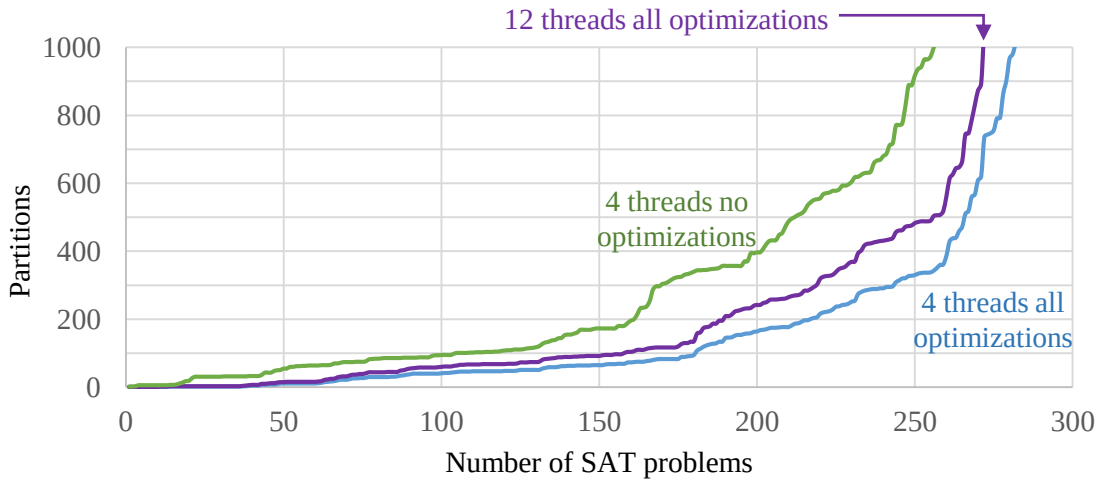


Figure 9.29: Number of partitions needed for 4 threads versus 12 threads.

Variable assignments are replicated per thread, yet as shown in Figure 9.29, the extra space in hardware memory required by increasing the number of threads is only a fraction of the space that we can recover with our optimizations.

Another way to measure the severity of disabling each optimization is by the average number of preprocessed clauses per partition, as summarized in Table 9.14. The trends are consistent with our above explanations.

Finally, we estimate the overall compaction of our entire system. We averaged the number of original clauses per partition over all 300 SAT problems. The average was 4877 clauses per partition using 12 threads with all optimizations on. Given an average clause size of 2.739, we fit 13358 original variable occurrences per partition on average, which is more than the 8192 memory locations per partition.

In conclusion, our use of implicit representations and lossless compression from section 4.2 already make our hardware memory more compact than most prior works. By adding enhanced BCP, macro clauses, and preprocessing compaction, we are able to benchmark in uncharted territory for FPGA-accelerated SAT.

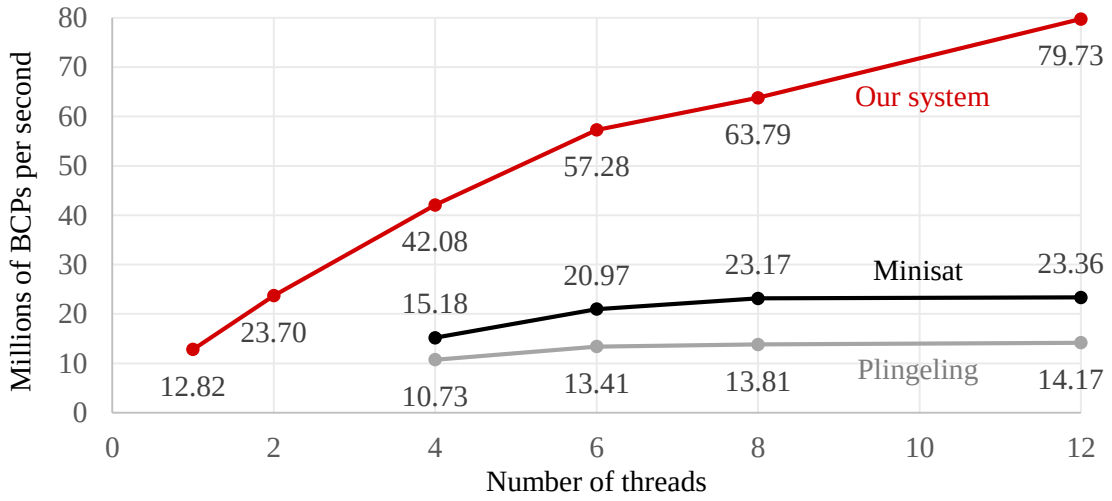


Figure 9.30: Average BCP performance on the SAT Competition 2013 applications benchmark.

9.7.3 BCP Performance Versus Multithreaded Software

For each number of threads, we averaged our hardware BCP performance over all SAT problems that could fit into hardware memory. With more threads, the largest problems no longer fit, thus a slightly different set of SAT problems was used for benchmarking each number of threads.

For software (Plingeling and Minisat), we computed the average BCP rate using all 300 SAT problems as well as the average BCP rate using only the problems that fit in hardware for the same number of threads. The higher of the two BCP rates is shown for each result in Figure 9.30. Excluding the largest problems may benefit software performance (fewer CPU cache misses) or may penalize software performance (larger problems tend to produce more BCPs). Choosing the higher of the two rates for software ensures we have not biased our comparison against software in our favor.

With 12 threads, our BCP is 5.6× and 3.4× faster than Plingeling and Minisat respectively. With only 2 threads we outperform both software approaches, thus

Table 9.15: Statistics from all SAT Competition 2013 applications problems that fit in hardware at each number of threads.

Hardware threads	Average number of BCPs per round	Average time used by each thread	Average time at least one thread is active
1	779	28.8%	28.8%
2	770	30.1%	45.1%
4	763	31.7%	57.9%
6	769	31.7%	59.7%
8	755	29.2%	58.4%
12	728	27.2%	57.8%

potentially offering more energy efficient computation than pure software for the same BCP performance, as discussed at the end of section 9.5.3.

As shown in Table 9.15, the average number of BCPs per round was in the 700s. This is higher than in the Boolean 3-SAT benchmark but lower than in the enhanced BCP benchmark (see Figure 9.14). This is expected, as Figure 9.26 shows that some SAT problems contain mostly logical constructs whereas other problems have few, thus this benchmark is like a mix of the previous two. On practical SAT problems, the hardware/software communication latency is still highly amortized over many BCPs.

As we increase the number of threads, the largest problems are excluded and thus the number of BCPs decreases. When software spends less time translating hardware BCP and deassigning variables, it can supply more work to the FPGA, hence the hardware utilization increases. However, as we continue to increase the number threads, eventually this is outweighed by the increase in CPU cache misses.

With 4 or more threads, hardware is active for more than half of all clock cycles. This is about twice as much compared to only using a single thread. Multithreading mitigates the hardware/software communication latency by allowing multiple rounds of BCP (from different threads) to be active at the same time.

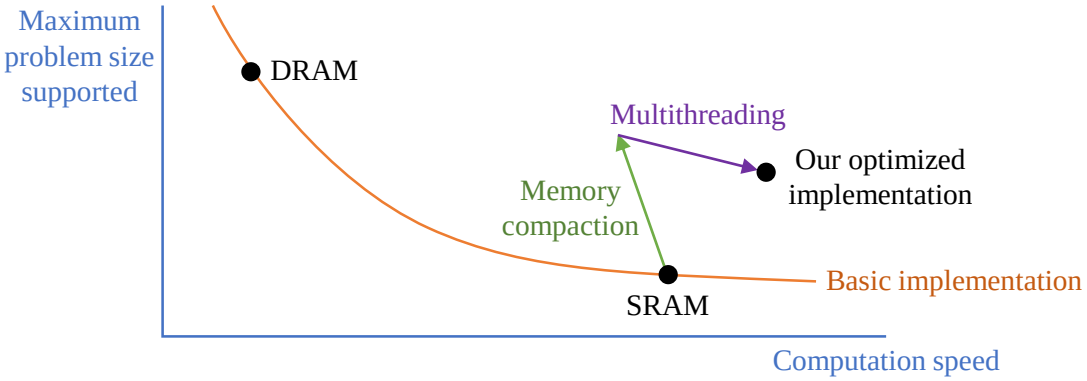


Figure 9.31: The type of memory determines the tradeoff between speed and size. We then optimize this with memory compaction and multithreading.

9.8 Iterative Refinement with Future Benchmarks

Benchmarking with various parameter settings facilitates profiling optimization (e.g. how to choose the best number of threads to run given any arbitrary SAT problem). It may also indicate which specific design aspects offer the most room for improvement. However, large scale benchmarking is required before any empirical refinement is possible. Due to the massive implementation effort needed for this, we are now finally able to propose the first step in refinement.

Figure 9.31 shows that the underlying physical characteristics of memory determines the tradeoff between computation speed and problem size support. To be faster than software, we require SRAM, as discussed in section 3.3.2. This can be optimized with memory compaction and multithreading. Implicit representations, lossless compression, and other memory compaction optimizations significantly increase the supportable problem size at a small expense to the computation speed. Conversely, multithreading significantly increases the total computational throughput, but replicated variable assignments slightly decreases the maximum problem size. Both features together result in a net improvement in both speed and size, as illustrated in Figure 9.31.

Table 9.16: Millions of BCPs per second in the SAT Competition 2013 applications benchmark.

Threads	Enhanced BCP (always used)	Mostly Boolean clauses (use enhanced BCP only when needed to fit in FPGA memory)	Speedup
1	12.82	13.34	4%
2	23.70	24.68	4%
4	42.08	44.21	5%
6	57.28	61.07	7%
8	63.79	70.49	11%
12	79.73	88.05	10%

Our existing optimizations have mostly focused on memory compaction, since the effects on the memory layout can be *analyzed*. Measuring the compaction from enhanced BCP and macro clauses does not require any hardware implementation.

Having finally measured the computation speed on large benchmarks for the first time, one aspect we have noticed is that using enhanced BCP is *sometimes* slower than using the equivalent Boolean-only SAT problem. When we converted the enhanced BCP benchmarks from section 9.6 to the equivalent Boolean problems, nearly identical BCP performance was measured (for the smaller problems that still fit in hardware memory). Conversely, on the SAT Competition 2013 applications benchmark from section 9.7, we observed a small improvement in performance by using enhanced BCP only when absolutely necessary to fit the problem in hardware memory. The results are presented in Table 9.16.

One potential slowness of enhanced BCP is caused by the revisiting of a clause to imply more than one variable. For example, given the clause $a = b$ NOR c, assigning $a = 1$ will imply $b = 0$ and $c = 0$. As explained in section 6.1, $b = 0$ is implied now, and when we later visit all clauses that contain b, we imply $c = 0$ when we revisit the

$a = b$ NOR c clause. There may be other tasks already in the input queue which must be processed before this revisit, thus delaying when $c = 0$ is implied.

Conversely, using the equivalent Boolean clauses of $(\overline{a} + \overline{b})(\overline{a} + \overline{c})(a + b + c)$, two separate clauses each imply one variable, thus there is no revisit. However, if $(\overline{a} + \overline{b})$ and $(\overline{a} + \overline{c})$ reside in different processors, implying $c = 0$ can still be delayed by the backpressure in the random access network and by other tasks already in the input queue at the destination processor. In practice, this likely has less impact than revisiting because our clause partitioning algorithm tries to place $(\overline{a} + \overline{b})$ and $(\overline{a} + \overline{c})$ in the same partition to minimize the number of global links.

Lossless compression is only required if the given SAT problem will not otherwise fit in hardware memory, so one simplistic way to improve the computation speed is to disable memory compaction optimizations based on problem size. Alternatively, we could restructure the queues in hardware to accept more than one BCP per clock cycle. This is not a trivial modification. Additional control will add latency to the processing, which may outweigh the new benefits.

Ultimately we will need additional benchmarks (possibly with hand-crafted SAT problems) to *accurately* determine the effect of enhanced BCP on the computation speed. Several rounds of refinement and benchmarking will likely be needed.

For practicality reasons, we have not been able to perform such extensive benchmarking in the limited amount of time. These future optimizations are beyond the scope of this thesis, in which the primary objective was to demonstrate that customized heterogeneous computing can significantly accelerate SAT compared to pure software.

Chapter 10

Conclusion

This chapter discusses how our FPGA design is compatible with alternative and future technologies and can likely be extended to support generalized decision-based problems. We conclude by summarizing and generalizing our contributions.

10.1 Compatibility with Alternative Technology

10.1.1 Xilinx FPGAs

Xilinx [60] and Altera [59] are the two primary FPGA vendors. We prototyped on an Altera FPGA, yet all components in our hardware design are compatible with Xilinx FPGAs. Our design contains about 10,000 lines of register transfer level code (in SystemVerilog) as well as intellectual property (IP) blocks, which are only used for:

1. The PCIe interface, which includes hardwired high-speed logic.
2. PLLs (phase-locked loops) for generating clocks.
3. On-chip SRAM, both LUT-based memory and embedded memory.

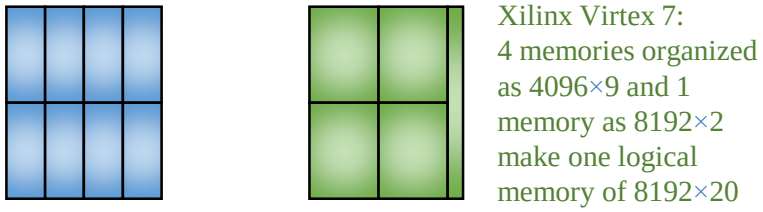


Figure 10.1: Different physical memories can construct the same logical memory.

Xilinx Virtex 7 FPGAs support PCIe generation 3 with 8 lanes, which is what we used. We have actually implemented a simpler PCIe interface on Virtex 5, and the signals are very similar to Altera's IP block. Modern FPGAs contain many PLLs to support multiple clock domains. Our processing and communication operate on only one clock, and the frequency is independent of the second clock for the PCIe interface.

The usage of memory in Altera Stratix V and Xilinx Virtex 7 are very similar. Writing is identical, and so is the read latency. Our design by construction will never write to an address with one port while reading that same address from the other port on the same clock cycle, as this may produce different behavior on different FPGAs.

For embedded memory, both support dual port mode (two independent accesses per clock cycle). Both support using two different clocks, one on each port of the memory (we wrote our own clock domain crossing queues). We use LUT-based memory for queues, and the features we need are supported by both FPGA vendors.

With respect to our design, the *only* considerable difference is the size of the embedded memories. Stratix V uses M20K memories (each 20 kilobits) whereas BRAM memories in Virtex 7 are 36 kilobits. Stratix V and Virtex 7 FPGAs have a similar capacity of logic elements and embedded memory, so we expect our design on Xilinx to use several BRAMs per clause traversal engine. As shown in Figure 10.1, we used 8 physical M20K memories to form one logical memory of 8192 addresses with 20 bits per location. This can be created with 5 BRAMs (using techniques from [96]).

10.1.2 3D Silicon Die Stacking

As transistor sizes approach the atomic scale and lithography becomes increasingly challenging, one emerging approach to prolonging Moore's Law is to stack several 2-dimensional (2D) silicon dies to form one 3-dimensional (3D) integrated circuit [61].

Die stacking allows for an easier integration of heterogeneous computing. For example, an FPGA can be stacked on top of a CPU using through-silicon vias. This would significantly reduce the hardware/software communication latency within our system, leading to higher overall performance. If 2D FPGAs are stacked to form one 3D circuit, the 2D on-chip networks in our design could easily add another level to the existing hierarchy to adapt to the added physical dimension.

Memory chips can also be stacked on an FPGA or CPU. For CPUs, this could introduce another level in the cache hierarchy for the now larger last layer cache. Through-silicon vias consume significantly less physical space than a pin, thus mitigating the "memory wall" which causes data starvation.

Our FPGA design is limited by the amount of SRAM within the FPGA itself. This could easily be alleviated by stacking one or more SRAM chips on the FPGA. The SRAM chips could potentially use a different manufacturing process than the FPGA, thus facilitating additional optimizations (e.g. using 4 transistors per bit instead of 6).

One major challenge for 3D integrated circuits is power and heat dissipation. With more transistors and a limited power budget, we must become more energy efficient per transistor. This heavily favors customized computing, in which we perform the equivalent of several CPU instructions within a single clock cycle.

In conclusion, the technology trends in the near future will likely provide more benefit for accelerating SAT on an FPGA compared to pure software on CPUs.

10.2 Extensibility of Our Design

Any NP-complete decision problem can be solved via Boolean SAT, yet preserving the high-level semantics can improve the practicality of the solver. For example, when using SAT for verification, we must encode the behavior of the digital system, which often involves integers. Once a problem is "bit blasted" into Boolean SAT (as demonstrated in section 2.2.3), it is arguably more difficult for a SAT solver to discover obvious relationships among the variables, such as $x + y = y + x$ where x and y are integers each composed of several equivalent Boolean variables.

Our hardware architecture *could be* capable of supporting Satisfiability Modulo Theories (SMT), which is a generalization of SAT. However, the extensions from SAT to SMT are complex. We refer the reader to [23, 24] for a comprehensive analysis of SMT. Examining the numerous intricacies is orthogonal to and beyond the scope of this thesis, hence we do not claim that our design will easily support it.

With variable assignments still coming from software, our constraint propagation engine could be replaced by a "theory solver". For example, suppose we had the constraint: $2x - y + 3z \leq 10$, where x, y, and z are non-negative integers. If software decides to assign $x \geq 1$ and $y = 0$, then hardware could propagate $z \leq 2$. We have partially taken a step in this direction with enhanced BCP. Operating with integers would facilitate the use of the hardwired multipliers which are commonly available in FPGAs. However, an *arbitrary* theory solver would be extremely complex and thus significantly fewer processors would likely fit within the same FPGA.

The clause traversal engines would likely operate similarly, except that each variable assignment would be multi-valued, thus consuming more memory. Even so, SMT problems typically have significantly fewer variables than Boolean SAT problems.

10.3 Summary of Our Contributions

The class of discrete or combinatorial problems are typically difficult to accelerate. This is due to the complex control flow inherent to the decision-based approach used to solve such problems. Our acceleration of SAT addressed this with a customized computing architecture.

Our contributions arguably begin in chapter 3, as we motivate our design decisions based on the suitability of each technology for our chosen application. In general, to efficiently compute any application involving mostly random access to memory, one requires a memory density no more dense than SRAM. Heterogeneous computing offers the potential to maximize the strength of each underlying technology, however the inherent communication latency between these technologies is a non-trivial overhead. In the case of fundamentally serial computation such as BCP, data dependencies can be broken at the algorithm level (e.g. racing SAT solvers against each other) to avoid starving processors of work.

Our hardware design was presented throughout chapters 4, 5 and 6. To obtain acceleration, computation and data structures must be *fundamentally restructured*. Simply porting software to hardware is disadvantageous since FPGAs have a significantly slower clock than CPUs. To compensate, FPGAs use customized control and data paths to perform the equivalent of numerous CPU instructions within a single FPGA clock cycle.

Chapter 4 demonstrated many ways to minimize memory while also facilitating faster computation. Implicit representations and lossless compression are powerful tools, and these will be useful in other memory bound problems. In chapter 5, different classes of on-chip communication were isolated in order to exploit the properties of

each. In general, applications may benefit from separating synchronization networks from data. Chapter 6 presented techniques for further compaction. In memory bound problems, it is often advantageous to trade more computation for less memory.

Integration with software was discussed in chapters 7 and 8. We resolved many of the ensuing complications of hardware accelerated BCP, such as conflict analysis on enhanced BCP, BCP reordering, integration of learnt clauses into hardware, and clause partitioning. In general, computation effort in software and hardware design complexity can be jointly optimized. For example, timestamp-based conflict analysis is faster than purely software dependency-based reordering and it also consumes significantly fewer logic resources than enforcing the correct ordering in hardware.

Finally, our results can be regarded as a contribution since we have benchmarked in unchartered territory for FPGA-accelerated SAT. Using real-life SAT problems, we are substantially faster than software. Furthermore, our implementation enables us to accurately observe system limitations such as thrashing the shared cache of the CPU.

10.4 Concluding Remarks

Ultimately this thesis provides some new insights on heterogeneous computing. CPUs provide large amounts of functionality per transistor, yet customized computing typically trades this off for higher compute throughput per transistor. Optimizing this balance between devices (heterogeneous computing) and within a device (our FPGA design) requires a comprehensive understanding of the application.

We have thoroughly investigated the acceleration of decision-based problems. For the Boolean satisfiability problem, we presented key architectural features for fast BCP in an FPGA as well as a seamless integration with multithreaded software.

Bibliography

[1] S. A. Cook, "The complexity of theorem-proving procedures," in *ACM Symposium on Theory of Computing*. ACM, 1971, pp. 151–158.

[2] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[3] R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*. Plenum Press, 1972, pp. 85–103.

[4] 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.

[5] J. Marques-Silva, "Practical applications of boolean satisfiability," in *International Workshop on Discrete Event Systems*. IEEE Press, 2008, pp. 74–80.

[6] E. Goldberg, "Equivalence checking of circuits with parameterized specifications," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer-Verlag, 2005, pp. 107–121.

[7] S. Reda and A. Salem, "Combinational equivalence checking using boolean satisfiability and binary decision diagrams," in *Conference and Exhibition on Design, Automation and Test in Europe*. IEEE Press, 2001, pp. 122–126.

[8] A. Gupta, M. K. Ganai, and C. Wang, "SAT-based verification methods and application," in *International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems*. Springer-Verlag, 2006, pp. 108–143.

[9] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based bounded model checking for software verification," *Theoretical Computer Science*, vol. 404, no. 3, pp. 256–274, Sep. 2008.

[10] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, Jul. 2001.

[11] N. Amla, X. Du, A. Kuehlmann, R. P. Kurshan, and K. L. McMillan, "An analysis of SAT-based model checking techniques in an industrial environment," in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 2005, pp. 254–268.

[12] A. C. Ling, D. P. Singh, and S. D. Brown, "FPGA logic synthesis using quantified boolean satisfiability," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer-Verlag, 2005, pp. 444–450.

[13] G.-J. Nam, K. A. Sakallah, and R. A. Rutenbar, "Satisfiability-based layout revisited: Detailed routing of complex FPGAs via search-based boolean SAT," in *International Symposium on Field Programmable Gate Arrays*. ACM, 1999, pp. 167–175.

[14] N. Ryzhenko and S. Burns, "Standard cell routing via boolean satisfiability," in *Design Automation Conference*. ACM, June 2012, pp. 603–612.

[15] T. Larrabee, "Test pattern generation using boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 1, pp. 4–15, Jan 1992.

[16] S. Eggersglüß, "Robust algorithms for high quality test pattern generation using boolean satisfiability," Ph.D. dissertation, University of Bremen, 2010.

[17] I. Mironov and L. Zhang, "Applications of SAT solvers to cryptanalysis of hash functions," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer-Verlag, 2006, pp. 102–115.

[18] H. Zhang, D. Li, and H. Shen, "A SAT based scheduler for tournament schedules." in *International Conference on Theory and Applications of Satisfiability Testing*. Springer-Verlag, 2004, pp. 191–196.

[19] J. Rintanen, "Engineering efficient planners with SAT," in *European Conference on Artificial Intelligence*. IOS Press, 2012, pp. 684–689.

[20] I. Lynce and J. Marques-Silva, "SAT in bioinformatics: Making the case with haplotype inference," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer-Verlag, 2006, pp. 136–141.

[21] N. Ollikainen, E. Sentovich, C. Coelho, A. Kuehlmann, and T. Kortemme, "SAT-based protein design," in *International Conference on Computer-Aided Design*. IEEE Press, 2009, pp. 128–135.

[22] G. S. Tseitin, "On the complexity of derivation in the propositional calculus," *Zapiski Nauchnykh Seminarov LOMI*, vol. 8, pp. 234–259, 1968, from *Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics* (translated from Russian).

[23] R. Sebastiani, "Lazy satisfiability modulo theories," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 3, pp. 141–224, Dec. 2007.

[24] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)," *Journal of the ACM*, vol. 53, no. 6, pp. 937–977, Nov. 2006.

[25] B. Selman, H. Levesque, and D. Mitchell, "A new method for solving hard satisfiability problems," in *National Conference on Artificial Intelligence*. AAAI Press, 1992, pp. 440–446.

[26] B. Selman, H. Kautz, and B. Cohen, "Noise strategies for improving local search," in *National Conference on Artificial Intelligence*. MIT Press, 1994, pp. 337 – 343.

[27] A. S. Fukunaga, "Efficient implementations of SAT local search," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer-Verlag, 2004, pp. 287–292.

[28] N. Kitchen and A. Kuehlmann, "Stimulus generation for constrained random simulation," in *International Conference on Computer-Aided Design*. IEEE Press, 2007, pp. 258–265.

[29] E. Goldberg, "Testing satisfiability of CNF formulas by computing a stable set of points," *Annals of Mathematics and Artificial Intelligence*, vol. 43, no. 1-4, pp. 65–89, Jan. 2005.

[30] E. Goldberg, "Solving satisfiability problem by computing stable sets of points in clusters," Cadence Research, Technical report CDNL-TR-2005-1001, 2005.

[31] J. P. Marques-Silva and K. A. Sakallah, "GRASP: a new search algorithm for satisfiability," in *International Conference on Computer-Aided Design*. IEEE Computer Society, 1996, pp. 220–227.

[32] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver," in *International Conference on Computer-Aided Design*. IEEE Press, 2001, pp. 279–285.

[33] International SAT Competition. `http://www.satcompetition.org`. 2014.

[34] C. P. Gomes, B. Selman, and H. Kautz, "Boosting combinatorial search through randomization," in *Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*. AAAI Press, 1998, pp. 431–437.

[35] H. Zhang, "SATO: An efficient propositional prover," in *International Conference on Automated Deduction*. Springer-Verlag, 1997, pp. 272–275.

[36] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Design Automation Conference*. ACM, 2001, pp. 530–535.

[37] N. Eén and N. Sörensson, "An extensible SAT-solver," in *International Conference on Theory and Applications of Satisfiability Testing*, vol. 2919. Springer-Verlag, 2003, pp. 502–518.

[38] A. Biere, "PicoSAT essentials," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 4, no. 2-4, pp. 75–97, May 2008.

[39] A. Biere, "Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010," Institute for Formal Models and Verification, Johannes Kepler University, Technical report, 2010.

[40] L. Ryan, "Efficient algorithms for clause learning SAT solvers," Master's Thesis, Simon Fraser University, 2004.

[41] K. Pipatsrisawat and A. Darwiche, "A lightweight component caching scheme for satisfiability solvers," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer-Verlag, 2007, pp. 294–299.

[42] M. Luby, A. Sinclair, and D. Zuckerman, "Optimal speedup of Las Vegas algorithms," *Information Processing Letters*, vol. 47, no. 4, pp. 173–180, Sep. 1993.

[43] J. Huang, "The effect of restarts on the efficiency of clause learning," in *International Conference on Artificial Intelligence*. Morgan Kaufmann Publishers, 2007, pp. 2318–2323.

[44] N. Een and N. Sörensson, "Minisat v1.13 - a SAT solver with conflict-clause minimization," Chalmers University of Technology, Sweden, Technical report, 2005.

[45] J. Chen, "Building a hybrid SAT solver via conflict-driven, look-ahead and XOR reasoning techniques," in *International Conference on Theory and Applications of Satisfiability Testing*, 2009, pp. 298–311.

[46] K. N. Mate Soos and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer-Verlag, 2009, pp. 244–257.

[47] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer-Verlag, 2005, pp. 61–75.

[48] S. Subbarayan and D. Pradhan, "NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances," in *Theory and Applications of Satisfiability Testing*. Springer Berlin Heidelberg, 2005, vol. 3542, pp. 276–291.

[49] F. Bacchus and J. Winter, "Effective preprocessing with hyper-resolution and equality reduction," in *Theory and Applications of Satisfiability Testing*. Springer Berlin Heidelberg, 2004, vol. 2919, pp. 341–355.

[50] M. Järvisalo, A. Biere, and M. Heule, "Blocked clause elimination," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, J. Esparza and R. Majumdar, Eds., vol. 6015. Springer, 2010, pp. 129–144.

[51] N. Manthey, "Coprocessor - a standalone SAT preprocessor," in *Applications of Declarative Programming and Knowledge Management*. Springer Berlin Heidelberg, 2013, pp. 297–304.

[52] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, Apr. 1965.

[53] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct 1974.

[54] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, "CPU DB: Recording microprocessor history," *Queue*, vol. 10, no. 4, pp. 10:10–10:27, Apr. 2012.

[55] M. B. Taylor, "Is dark silicon useful?: Harnessing the four horsemen of the coming dark silicon apocalypse," in *Design Automation Conference*. ACM, 2012, pp. 1131–1136.

[56] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2011.

[57] Joint Electron Device Engineering Council. `http://www.jedec.org`. 2014.

[58] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Power limitations and dark silicon challenge the future of multicore," *ACM Transactions on Computer Systems*, vol. 30, no. 3, pp. 11:1–11:27, Aug. 2012.

[59] Altera Corporation. `http://www.altera.com`. 2014.

[60] Xilinx Corporation. `http://www.xilinx.com`. 2014.

[61] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. Loh, D. McCauley, P. Morrow, D. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, "Die stacking (3D) microarchitecture," in *International Symposium on Microarchitecture*. IEEE Computer Society, 2006, pp. 469–479.

[62] O. Mencer and M. Plazner, "Dynamic circuit generation for boolean satisfiability in an object-oriented design environment," in *International Conference on Systems Sciences*. IEEE Computer Society, 1999, pp. 1–8.

[63] T. Suyama, M. Yokoo, H. Sawada, and A. Nagoya, "Solving satisfiability problems using reconfigurable computing," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 9, no. 1, pp. 109–116, Feb. 2001.

[64] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Using configurable computing to accelerate boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 6, pp. 861–868, Jun. 1999.

[65] I. Skliarova and A. B. Ferrari, "A software/reconfigurable hardware SAT solver," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 12, no. 4, pp. 408–419, Apr. 2004.

[66] I. Skliarova and A. de Brito Ferrari, "Reconfigurable hardware SAT solvers: a survey of systems," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1449–1461, Nov. 2004.

[67] Terasic Corporation. `http://www.terasic.com.tw/en/`. 2014.

[68] L. Haller and S. Singh, "Relieving capacity limits on FPGA-based SAT-solvers," in *Conference on Formal Methods in Computer-Aided Design*. IEEE, 2010, pp. 217–220.

[69] 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, May 2008.

[70] H. Katebi, K. A. Sakallah, and J. a. P. Marques-Silva, "Empirical study of the anatomy of modern SAT solvers," in *International Conference on Theory and Application of Satisfiability Testing*. Springer-Verlag, 2011, pp. 343–356.

[71] M. Safar, M. El-Kharashi, and A. Salem, "FPGA-based SAT solver," in *Canadian Conference on Electrical and Computer Engineering*. IEEE, 2006, pp. 1901–1904.

[72] M. Safar, M. El-Kharashi, M. Shalan, and A. Salem, "A reconfigurable, pipelined, conflict directed jumping search SAT solver," in *Design, Automation, and Test in Europe Conference*. IEEE, 2011, pp. 1–6.

[73] W. Chrabakh and R. Wolski, "GridSAT: A Chaff-based distributed SAT solver for the grid," in *Supercomputing*. ACM, 2003, pp. 37–49.

[74] W. Chrabakh, "GridSAT: A distributed large scale satisfiability solver for the computational grid," Ph.D. dissertation, University of California at Santa Barbara, 2006.

[75] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: Portfolio-based algorithm selection for SAT," *Journal of Artificial Intelligence Research*, vol. 32, no. 1, pp. 565–606, Jun. 2008.

[76] Y. Hamadi, S. Jabbour, and L. Sais, "ManySAT: a parallel SAT solver." *Journal of Satisfiability*, vol. 6, no. 4, pp. 245–262, Jun. 2009.

[77] S. Holldobler, N. Manthey, V. H. Nguyen, J. Stecklina, and P. Steinke, "A short overview on modern parallel SAT-solvers," in *International Conference on Advanced Computer Science and Information System*. IEEE, 2011, pp. 201–206.

[78] K. Kanazawa and T. Maruyama, "An FPGA solver for SAT-encoded formal verification problems," in *International Conference on Field Programmable Logic and Applications*. IEEE Computer Society, 2011, pp. 38–43.

[79] M. Suzuki and T. Maruyama, "Variable and clause elimination in SAT problems using an FPGA," in *International Conference on Field-Programmable Technology*. IEEE, 2011, pp. 1–8.

[80] J. D. Davis, Z. Tan, F. Yu, and L. Zhang, "A practical reconfigurable hardware accelerator for boolean satisfiability solvers," in *Design Automation Conference*. ACM, 2008, pp. 780–785.

[81] J. D. Davis, Z. Tan, F. Yu, and L. Zhang, "Designing an efficient hardware implication accelerator for SAT solving," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer-Verlag, 2008, pp. 48–62.

[82] J. Thong and N. Nicolici, "FPGA acceleration of enhanced boolean constraint propagation for SAT solvers," in *International Conference on Computer-Aided Design*. IEEE Press, 2013, pp. 234–241.

[83] PCI Special Interest Group. `http://www.pcisig.com/home`. 2014.

[84] HyperTransport Consortium. `http://www.hypertransport.org`. 2014.

[85] Networks-on-Chip Consortium. `http://www.nocsymposium.org`. 2014.

[86] Z. Fu and S. Malik, "Extracting logic circuit structure from conjunctive normal form descriptions," in *International Conference on VLSI Design*. IEEE Computer Society, 2007, pp. 37–42.

[87] Linux. `http://www.linux.com`. 2014.

[88] J. van Leeuwen and D. Wood, "Interval heaps." *Computer Journal*, vol. 36, no. 3, pp. 209–216, Jun. 1993.

[89] Ubuntu Linux. `http://www.ubuntu.com`. 2014.

[90] The GNU Compiler Collection. `http://gcc.gnu.org`. 2014.

[91] POSIX.1-2008 Base Specification.
`http://pubs.opengroup.org/onlinepubs/9699919799/`. 2014.

[92] B. Selman and S. Kirkpatrick, "Critical behavior in the computational cost of satisfiability testing." *Artificial Intelligence*, vol. 81, no. 1-2, pp. 273–295, Mar. 1996.

[93] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Spring Joint Computer Conference*. ACM, 1967, pp. 483–485.

[94] A. Biere, "Lingeling, Plingeling and Treengeling entering the SAT competition 2013," Institute for Formal Models and Verification, Johannes Kepler University, Technical report, 2013.

[95] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *International Conference on Artificial Intelligence*. Morgan Kaufmann Publishers, 2009, pp. 399–404.

[96] K. Elizeh and N. Nicolici, "Embedded memory binding in FPGAs," in *Design Automation Conference*. ACM, 2010, pp. 457–462.

Index