Markov Chain Monte Carlo Stimulus Generation for Constrained Random Simulation

Nathan Boyd Kitchen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2010-165
http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-165.html

December 17, 2010

Markov Chain Monte Carlo Stimulus Generation for Constrained Random Simulation

by

Nathan Boyd Kitchen

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Engineering—Electrical Engineering and Computer Sciences

in the

Graduate Division
of the
University of California, Berkeley

Committee in charge:
Professor Andreas Kuehlmann, Chair
Professor Sanjit Seshia
Professor David Aldous

Fall 2010

The dissertation of Nathan Boyd Kitchen is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2010

Markov Chain Monte Carlo Stimulus Generation
for Constrained Random Simulation

Nathan Boyd Kitchen

Abstract

Markov Chain Monte Carlo Stimulus Generation
for Constrained Random Simulation

by

Nathan Boyd Kitchen

Doctor of Philosophy in Engineering—Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Andreas Kuehlmann, Chair

As integrated circuits have grown in size and complexity, the time required for functional verification has become the largest part of total design time. The main workhorse in state-of-the-art practical verification is constrained random simulation. In this approach, a randomized solver generates solutions to declaratively specified input constraints, and the solutions are applied as stimuli to a logic simulator. The efficiency of the overall verification process depends critically on the speed of the solver and the distribution of the generated solutions. Previous methods for stimulus generation achieve speed at the expense of quality of distribution or rely on techniques that do not scale well to large designs.

In this dissertation, we propose a new method for stimulus generation based on Markov chain Monte Carlo (MCMC) methods. We describe the basic principles of MCMC methods and one of the most common of these methods, Metropolis-Hastings sampling. We present our approach, which combines the Metropolis-Hastings algorithm with stochastic local search. We show with experimental results that it surpasses existing stimulus-generation methods in speed, robustness, and quality of distribution.

After presenting our basic algorithm, we describe several refinements and variations of it. These refinements include the addition of control variables to handle dependencies on external data and elimination of variables to increase efficiency and distribution. In addition, we present a parallel version of our algorithm and give theoretical analysis and experimental evidence of the speedup achieved by parallelization.

To Lauren,
who has earned it as much as I have

Contents

List of Figures

List of Tables

Acknowledgments

First I would like to thank Professor Andreas Kuehlmann for his support and collaboration. He has demanded work of high quality, and his encouragement has given me the confidence to produce it. I hope to emulate his example of integrity and hard work in my own career.

I would like to thank Professors Sanjit Seshia and David Aldous for reviewing my dissertation and helping me improve it, and thank them and Professor Bob Brayton for being on my qualifying-exam committee. I would also like to acknowledge Professors Michael Jordan and Alistair Sinclair, whose courses exposed me to MCMC methods just when I needed them and showed me how much I could do with them.

Vitaly Lagoon and Giora Alexandron of Cadence Design Systems helped me tremendously with test cases that challenged me. My algorithm is much more useful in practical settings because of my collaboration with them.

I would like to thank the researchers at Cadence Research Laboratories for their friendship and for many interesting lunch discussions. I am particular grateful to Philip Chong, Carlos Coelho, and Eric Schaefer, who helped me with debugging and experiments. Without their help I would likely still be working on this dissertation for months to come.

I am deeply grateful to Cadence Design Systems for their generosity in funding my studies.

I would like to acknowledge the staff of the UC Berkeley EECS Department for their support. Extra thanks to Ruth Gjerde, who always made me feel welcome and relieved my anxiety many times with her reassurance.

I am grateful to my fellow students for their friendship, their collaboration, and their examples. A few of them deserve special thanks: Abhijit Davare and Bryan Catanzaro, for their warm friendship; Donald Chai, for many thought-provoking conversations; Qi Zhu, for the opportunity to do excellent research together, and Tobias Welp, for working with me on the research for this dissertation and his close friendship.

Finally I would like to thank my family. My parents instilled in me from an early age the importance of education and of hard work. I am thankful that they did, and I am proud to be able to follow my father's example. I am grateful to Karen, Megan, Leslee, and Joanie for stepping in so that I could focus my attention on this work. Most of all, I am grateful to my wife Lauren for her unending patience and encouragement and for her sacrifices. I could not have finished this work without her support.

Chapter 1

Introduction

Modern digital integrated circuits are extremely complex. State-of-the-art chips contain millions or billions of transistors. Designing such a large circuit with the correct behavior is an enormous challenge, and functional verification—checking that a design behaves as required by the specification—is a critical part of the design process. As the size and complexity of circuits has increased, the time required for functional verification has become the majority of total design time; two-thirds of design effort is now spent in verification rather than in implementing the specification [ITR09, Table DESN6]. Thus, the efficiency of functional verification has a significant impact on the speed with which designs can be put into production.

Approaches for functional verification can be grouped into two broad categories: simulation-based techniques and formal methods. These two kinds of approaches differ in their completeness and scalability. Simulation-based methods evaluate the logic of the design for a subset of the possible input values, while formal methods prove behavioral properties for all possible inputs. On the other hand, simulation time scales linearly with the design size and the number of cycles, while the runtime of formal methods can be exponential in the design size. Contemporary verification flows include both kinds of methods, but simulation has been and continues to be the main workhorse in practical verification because it is more scalable and predictable.

The input stimuli for simulation are produced by a *testbench* that is executed simultaneously with the simulator. In order to avoid producing invalid stimuli that might lead to false negative verification results, the testbench must obey input constraints. For example, the implementation of a protocol interface may only be required to be correct for data packets with valid headers, so the input constraints should exclude invalid headers.

The methods used for stimulus production have evolved over time. Testbenches were originally written in imperative programming languages such as C or C++. In their earliest form, they simply applied input assignments that had been chosen manually. The need for greater productivity motivated the development of more complex procedures that generated stimuli algorithmically rather than just enumerating them. They supported nondeterminism through the use of random number generators (e.g., `rand()` in the C standard library); random values could be used as data or for choosing alternative generation paths. These imperatively specified testbenches encoded input constraints implicitly in the generation routines.

As the complexity of input constraints increased, such an unstructured approach became unworkable. First, in the absence of a concise way to express complex constraints, the inputs can easily be under- or over-constrained, resulting in spurious verification failures or un-

covered stimuli. Second, the ad hoc use of random number generators to diversify the stimuli makes it impossible to ensure a "good" distribution over the solution space. This can dramatically decrease the verification coverage and increase the required lengths of simulation runs.

1.1 Constrained Random Simulation

The need to express constraints more concisely led to the use of declarative specifications. Declarative constraints are not directly executable, so an online constraint solver is required to generate the stimuli. Since the testbench can have an internal state and the constraints may depend on it or the state of the design itself (e.g., to vary the data produced during the different phases of a communication protocol), the solver must run in lockstep with the simulator to generate stimuli that correspond to the current state.

This approach to verification, including declarative constraints and a randomized solver as the stimulus generator, is called *constrained random simulation* (CRS). Figure 1.1 depicts the structure of a CRS testbench, along with the design under test (*DUT*). Besides the stimulus generator, the testbench includes a monitor to check the correctness of the design's behavior and a coverage analyzer to measure how much of the behavior has been checked.

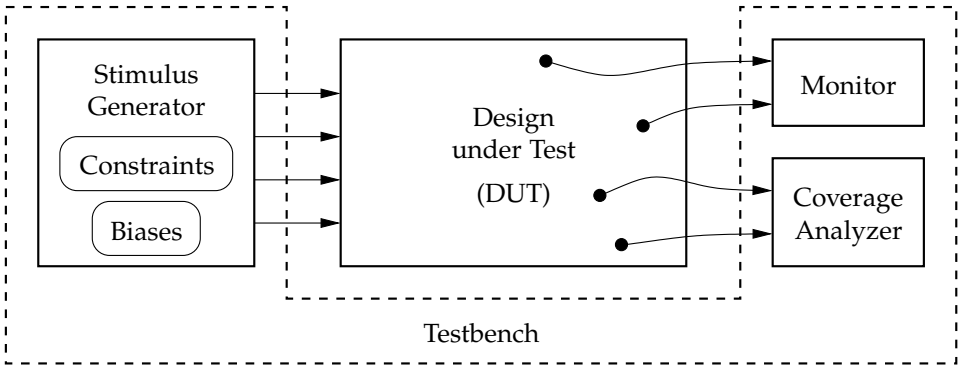
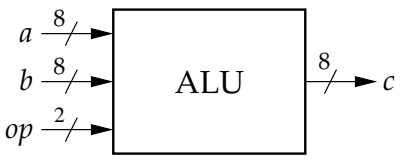


Figure 1.1: Structure of a setup for constrained random simulation.

Specific languages used in practice for specifying testbenches declaratively include SystemC [GLMS02], SystemVerilog [SDF03], and *e* [IJ04]. An example of constraints for an 8-bit ALU, written in SystemVerilog, is shown in Figure 1.2. The constraints specify input values such that the output *c* is computed without overflow or division by zero.




```
rand enum { ADD, SUB, MUL, DIV } op;
rand bit signed [7:0]            a, b;
constraint valid {
  op == ADD -> (-128 <= a + b && a + b <= 127);
  op == SUB -> (-128 <= a - b && a - b <= 127);
  op == MUL -> (-128 <= a * b && a * b <= 127);
  op == DIV -> (b != 0);
}
```

Figure 1.2: An example of a DUT: an 8-bit ALU, along with input constraints in SystemVerilog.

A constraint solver used in stimulus generation must fulfill two requirements. First, the solver must be fast to avoid becoming a performance bottleneck; the overall verification runtime should not be dominated by the time spent solving the input constraints. Second, it must be possible to control the distribution of the generated stimuli in order to achieve the coverage goal as quickly as possible. A highly skewed distribution can decrease the likelihood of stimulating some of the design's behaviors and thus greatly increase the number of simulation cycles required to achieve full coverage. If the relationship between the stimulus distribution and the coverage metric is not known, the best choice for the distribution is the one with maximum entropy—i.e., the uniform distribution—so as to maximize the level of "surprise" and thus the chance of hitting a new coverage point. When many of the coverage points relate to corners of the design's state space, it is better to use a stimulus distribution with increased probability at the corners, such as a piecewise-uniform distribution. This case can be supported by means of user-specified biases on the stimulus distribution.

The key to finding a stimulus-generation approach with good performance and distribution is the fact that input constraints are typically easy to solve. That is, the set of valid inputs is relatively dense within the space of all possible assignments and it does not have complex structure, in contrast with solutions to other decision problems that arise in verification, such as counterexamples in model checking. Therefore, the full deductive power of general-purpose constraint solvers is not needed. This opens the possibility of using methods with less solving capacity in order to gain guarantees on quality of distribution.

1.2 Previous Work

Achieving both efficiency and good distribution in stimulus generation is a demanding challenge. The difficulty of the problem is reflected in the limitations of existing tools for CRS. Some generate stimuli efficiently but from highly skewed distributions. Moreover, the distributions are unstable—they can vary significantly with small changes in specifications, such as changes in variable-declaration order. Other tools generate values uniformly but rely on techniques (e.g., binary decision diagrams) that do not scale well to large designs. In this section, we give a quick overview of various methods for stimulus generation proposed in previous work and note their weaknesses.

Sampling from weighted binary decision diagrams (BDDs) was proposed in [YSP+99, YAPA04]. The idea is to build a BDD from the input constraints and to weight the vertices in such a way that a simple linear-walk procedure from the root to the terminal vertex, with branching probabilities derived from the weights, generates stimuli with a desired distribution. This approach guarantees the correct distribution. However, BDDs often blow up for practical problems. For example, many testbenches use multiplication operations on data variables, but the number of vertices in a BDD for multiplication is exponential in the number of bits [Bry86]. In our experiments we compared a BDD-based sampler with our proposed method; we provide the results in Section 3.6.1.

Another BDD-based approach involves building a circuit whose structure matches that of the constraint BDD [KS00]. This method does not require the variables to be ordered, and so it may avoid the common memory blowups, but its output distribution can be highly skewed.

Interval-propagation sampling is a technique described in [Iye03]. It maintains for each variable a range of possible values, represented as a set of intervals. Each variable is successively assigned a randomly chosen value from its range, and the intervals of the remaining variables

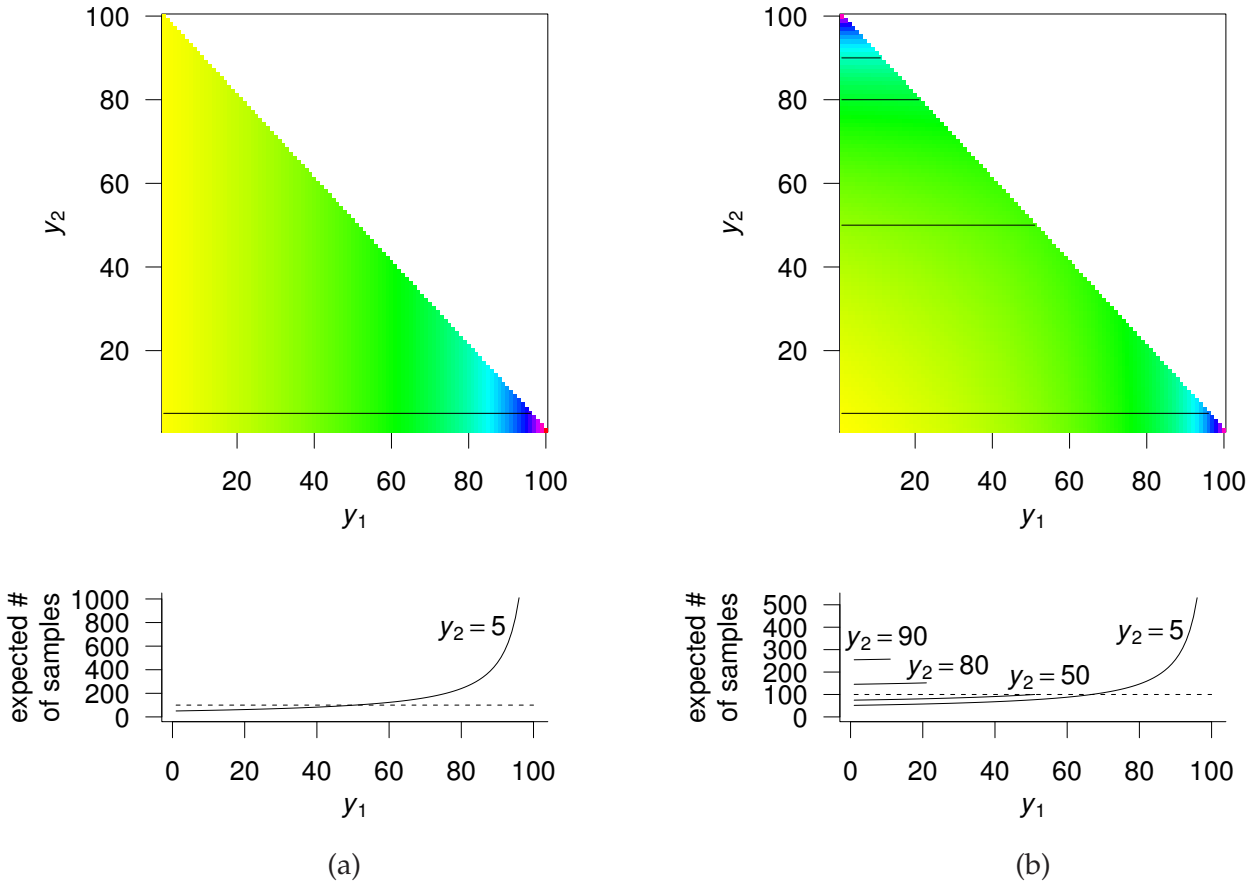


Figure 1.3: Distribution of 505 000 samples when using interval-propagation sampling on the constraints $y_1 + y_2 \leq 101; y_1, y_2 \geq 1$: (a) using fixed sampling order y_1, y_2, (b) using random sampling order. The lower parts show the expected number of samples for each y_1 for fixed values of y_2. The dashed lines show the expected value of 100 for a uniform distribution.

are subsequently refined. When conflicts reduce a variable's range to the empty set, the solver backtracks.

The advantage of the interval-propagation sampling algorithm is its simplicity and relatively high speed. However, the sampling distribution of this algorithm is highly non-uniform. This results in low stimulus entropy, which reduces the overall coverage of verification. To illustrate this problem, Figure 1.3 depicts the skewed sampling distribution given by this method on the solution space defined by the constraints $y_1 + y_2 \leq 101; y_1, y_2 \geq 1$. If the solutions were generated uniformly and the total number of solutions generated were 505 000, the expectation of the number of occurrences of each solution would be 100. In contrast, under interval-propagation sampling, with y_1 sampled before y_2, the solutions $(1, 5)$ and $(101, 5)$ have expected frequencies of 50.5 and 1010, respectively. Varying the order of the variables randomly does not help much: The expected frequencies of the same two solutions become 51.6 and 531.3 (approximately). Moreover, it can be shown that for an n-dimensional simplex defined by $\sum_{i=1}^{n} y_i \leq c$ there is a gap between the expected number of occurrences of $(0, 0, \ldots, 0)$ and any corner $(0, \ldots, c, \ldots, 0)$ that is exponential in n.

Boolean satisfiability solvers, such as those based on the Davis-Putnam-Logemann-

Loveland (DPLL) algorithm [DP60, DLL62, MMZ+01], can be randomized to generate stimuli. To attempt a good distribution for the generated samples, a random pre-assignment is used to guide decisions. When the solver reaches a decision point, it applies the next available value from the pre-assignment. At the first conflict, the solver backtracks as usual, effectively finding a smallest cube that contains both the pre-assignment and a solution. The advantage of DPLL-based sampling is its fairly good scalability for a large set of practical constraints. The problem with it is that, depending on the constraints, the distribution can be highly non-uniform. Furthermore, its runtime can be slow, especially for constraints involving arithmetic. We report results from a comparison of a DPLL-based approach with our method in Section 3.6.1.

In general, all sampling algorithms that require bit-blasting the constraints, including BDD-based sampling, DPLL-based sampling, and the work presented in [KJR+08], lose the word-level structure and its information for efficient sampling.

Various **specialized stimulus generators** [CIJ+95, SD02, SHJ05] have been developed for specific verification domains utilizing particular domain knowledge for constraint specification and solving.

The verification work closest to ours is presented in [SCJI07]. Similar to our approach, the authors propose the use of a **Markov chain Monte Carlo (MCMC) sampler** to generate stimuli, in their case for verifying software rather than hardware. They add constraints in each iteration to guide the generator toward new parts of the input space, in contrast with our work, in which we generate many stimuli satisfying a single constraint formula. We also apply several practical adaptations, such as augmenting the basic MCMC moves with stochastic local search.

For the general problem of solving mixed Boolean and integer constraints, a more efficient alternative to Boolean DPLL is a **SAT Modulo Theory (SMT) solver** combining propositional logic with the theory of integers (see [BSST09] for an overview of SMT). Applying the latest advances of DPLL solvers, SMT algorithms enumerate variable assignments over the Boolean space and dynamically instantiate integer constraint problems which are then solved with a specialized theory solver. Randomness can be introduced into these algorithms at decision points as with DPLL, and SMT-based sampling has the same key weakness as DPLL-based sampling: The distribution is difficult to control.

Random-walk sampling for Boolean constraints was performed in SampleSat [WES04] by a combination of Metropolis sampling [MRR+53] with greedy steps from the local-search SAT solver WalkSat [SKC93]. This method works well for small constraint sets, but its performance deteriorates for larger constraint sets, especially for those that have been generated from arithmetic constraints on integers.

Another line of research [DKBE02, GD06] converts the constraints into **belief networks** and performs approximately uniform sampling based on estimating the size of the solution space. As reported in [GD06], the performance of this method does not match that achieved by SampleSat, despite some improvements adopted from modern constraint solvers.

Among the random-generation approaches described in this section, three types allow effective control of distribution: BDD-based sampling, sampling on belief networks, and MCMC methods. The weaknesses of the former two have already been described in this section. MCMC methods have potential for greater success. In particular, the Metropolis algorithm used in SampleSat works well on dense sampling spaces, in contrast with the Boolean solution spaces on which SampleSat operates. By applying it to word-level constraints instead of Boolean ones, we enable it to work more effectively.

1.3 Contributions of This Dissertation

We have developed a new algorithm for stimulus generation based on a combination of Markov chain Monte Carlo methods. It is a generalization of the algorithm used in SampleSat; where SampleSat simply flips Boolean values, we generate new values for integer variables by constructing and sampling from multiple types of distribution functions. We also reduce the correlation between successive stimuli to make them more useful for practical verification.

This dissertation presents our stimulus-generation algorithm and several extensions. Chapter 2 gives a brief introduction to the Markov chain Monte Carlo methods that are the theoretical foundation of our approach. Chapter 3 describes the primary components of our algorithm and how we combine them and provides results from experiments comparing our method with other approaches. Chapter 4 describes several refinements that sometimes increase our generator's effectiveness. Chapter 5 describes a scheme for combining multiple generators in parallel and gives theoretical and experimental results on the resulting speedup. Chapter 6 summarizes our contribution and reports limitations of our approach.

Chapter 2

Background on Markov Chain Monte Carlo Methods

The problem of sampling from a complex sample space (or, equivalently, sampling from a complex distribution over a sample space) is not unique to constrained random verification. In other fields where this problem occurs, a standard approach to solving it is to use Markov chain Monte Carlo (MCMC) methods. For example, these methods are used for:

- Estimating properties of magnets at different temperatures [NB99]
- Constructing phase diagrams of chemical models [FS01]
- Constructing possible phylogenies (family trees) for species based on similarities between their DNA sequences [MNL99]
- Finding alignments of genetic or protein sequences [HB01, MFWvH01]
- Finding associations of genetic markers for studying diseases [NQXL02]
- Inferring message routes through a network from link traffic [TW98]

Instead of generating each sample independently, an MCMC sampling method takes as samples the successive states visited in a simulation of a Markov chain. If the state transitions are set up appropriately, the distribution of states converges over time to a unique stationary distribution [Bré99]. The value of MCMC methods is that they provide a means to control the stationary distribution: Given a desired target distribution, they specify how to construct the Markov chain so that it converges to that distribution.

In this chapter we discuss properties of Markov chains that guarantee their convergence and give background of the MCMC methods that form the basis of our stimulus generator.

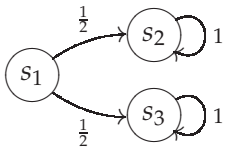
2.1 Markov Chains

For a Markov chain to have a unique stationary distribution and converge to it, two conditions are sufficient: irreducibility and aperiodicity. In this section we describe these two properties. We use M to denote the transition-probability matrix for a Markov chain over states $\{s_i\}$. $M(i,j)$ denotes the conditional probability of moving to state s_j given that the chain is

in state s_i. M^n denotes the nth power of the matrix; i.e., $M^1(i,j) = M(i,j)$ and $M^n(i,j) = \sum_k M^{n-1}(i,k)M(k,j)$ for $n \geq 2$.

Definition 2.1. A Markov chain is *irreducible* if and only if, for all states s_i and s_j, there exists t such that $M^t(i,j) > 0$. That is, every state can be reached from every other state in a finite number of steps.

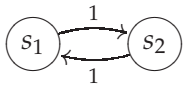
The following Markov chain illustrates the importance of irreducibility for convergence:



This chain has more than one stationary distribution—$(0,1,0)$ and $(0,0,1)$ and all convex combinations of these—and the chain may end up in any of them, so we cannot say that the distribution converges.

Definition 2.2. A Markov chain is *aperiodic* if and only if, for all s_i, $\gcd\{t : M^t(i,i) > 0\} = 1$. That is, given enough initial steps, the chain can visit a state at any time, not just at periodic intervals.

Aperiodicity is needed to "dampen" the influence of the choice of initial state so that the state distribution eventually loses its dependence on it. Consider the periodic Markov chain below:



This chain has stationary distribution $(\frac{1}{2}, \frac{1}{2})$, but these are not the state probabilities at any particular time. For example, if the chain starts in state s_1, it will be in state s_1 with probability 1 after an even number of steps and in state s_2 after an odd number of steps. Neither state probability is ever $\frac{1}{2}$.

2.2 The Metropolis-Hastings Algorithm

One of the most commonly used MCMC methods is Metropolis sampling [MRR+53]. In the Metropolis algorithm, Markov-chain transitions are implemented with two distributions: the target distribution p, which is the stationary distribution, and a proposal distribution q. We execute a transition by first selecting a proposed next state s' with probability $q(s'|s)$. The state s' is then accepted as the actual next state with probability α, where

$$\alpha = \min\left\{1, \frac{p(s')}{p(s)}\right\} \tag{2.1}$$

If s' is not accepted, the current state s is repeated as the next state.

Because the acceptance probability depends on the ratio of the target probabilities of s' and s, not their absolute probabilities, it is not necessary for p to be a normalized probability distribution. Instead, we can use the desired relative frequencies. For example, we can set $p(s_1) = 1$ and $p(s_2) = 2$ if we want s_2 to be generated twice as often as s_1. This property is useful

in the setting of randomized constraint solving where the normalizing factor for p is generally intractable to compute, especially in the typical case where the set of solutions is not known a priori.

The proposal distribution q may be chosen freely as long as it satisfies a few properties that ensure convergence. First, the moves with non-zero probability must connect the entire state space so that the Markov chain is irreducible. Second, the proposed moves must be aperiodic. Aperiodicity is not a simple property of q—it arises from the interaction between p and q—but we can easily achieve it by allowing self-loops: $q(s|s) > 0$. These two properties suffice to guarantee convergence to a stationary distribution. To ensure that the stationary distribution is specifically p, we need a third property: q must be symmetric, i.e., $q(s'|s) = q(s|s')$. The symmetry requirement on q can be relaxed if we incorporate the proposal probabilities into the acceptance rule:

$$\alpha = \min\left\{1, \frac{p(s')}{p(s)}\frac{q(s|s')}{q(s'|s)}\right\} \tag{2.2}$$

This generalization of Metropolis sampling is called the Metropolis-Hastings algorithm [Has70]. With this definition of α, the following equation holds for all pairs of states s and s':

$$p(s)M(s,s') = p(s')M(s',s)$$

This property is called *detailed balance*; it is a sufficient condition for p to be the stationary distribution.

Although the particular choice of proposal distribution does not affect the correctness of the Metropolis-Hastings algorithm (assuming that requirements for convergence are satisfied), it does determine the rate at which the distribution converges and how fast samples can be generated. If the proposed moves are small, then the number of moves needed to cross the state space is large, and the distribution converges slowly. On the other hand, if long-range moves are proposed with probabilities very different from the target distribution, the probability of rejection can be high. A state s with low $p(s)$ may be proposed often but rejected often because α is low. In this case moves to new states are infrequent and convergence is still slow.

To keep the probability of acceptance reasonably high, we should propose moves with distribution similar to p. In general, generating long-range moves with such a distribution is about as difficult as sampling from p directly, which we cannot do efficiently. We must restrict proposals in some way in order to move efficiently, accept frequently, and converge quickly.

2.3 Gibbs Sampling

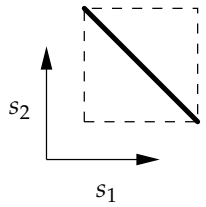
One way to generate long-range moves efficiently while keeping the acceptance rate high is to restrict moves to a single dimension at a time, i.e., to change the value of only one state variable per move. Suppose the state s is a vector $(s_1, \ldots, s_n)$. When a variable s_i is chosen uniformly at random, and the new value of the selected variable is chosen with distribution $p(s_i'|s \setminus s_i)$—the target distribution, conditioned on the current values of all the other variables—a proposed move is always accepted:

$$\alpha = \min\left\{1, \frac{p(s')}{p(s)}\frac{q(s|s')}{q(s'|s)}\right\}$$

$$
\begin{aligned}
&= \min\left\{1, \frac{p(s')}{p(s)} \frac{\frac{1}{n} p(s_i | s' \setminus s'_i)}{\frac{1}{n} p(s'_i | s \setminus s_i)}\right\} \\
&= \min\left\{1, \frac{p(s'_i | s' \setminus s'_i) p(s' \setminus s'_i)}{p(s_i | s \setminus s_i) p(s \setminus s_i)} \frac{\frac{1}{n} p(s_i | s' \setminus s'_i)}{\frac{1}{n} p(s'_i | s \setminus s_i)}\right\} \\
&= \min\left\{1, \frac{p(s'_i | s \setminus s_i) p(s \setminus s_i)}{p(s_i | s \setminus s_i) p(s \setminus s_i)} \frac{\frac{1}{n} p(s_i | s \setminus s_i)}{\frac{1}{n} p(s'_i | s \setminus s_i)}\right\} \\
&= \min\{1, 1\} \\
&= 1
\end{aligned}
$$

This version of the Metropolis-Hastings algorithm is called Gibbs sampling [GG84, GS90].

Gibbs moves can easily fail to connect the state space, so that the Markov chain is reducible and may not converge. For example, consider the state space consisting of the assignments (s_1, s_2) such that $a \leq s_1, s_2 \leq b$ and $s_1 + s_2 = b$, shown below; from any given state no single-variable move can reach a different state.



Because of its susceptibility to disconnectedness, Gibbs sampling from a set of solutions is not a good approach for randomized constraint solving, at least not by itself. However, it is a key ingredient in our approach to stimulus generation.

2.4 Convergence

Once we have chosen a proposal distribution that seems likely to give fast convergence, we would like to know how many moves are actually needed to converge. A standard metric to quantify the distance of a state distribution $\tilde{p}$ from stationarity is the total variation distance d_{TV} between $\tilde{p}$ and the stationary distribution:

$$d_{\text{TV}}(\tilde{p}, p) = \frac{1}{2} \sum_s |\tilde{p}(s) - p(s)|$$

Theoretical bounds on the total variation distance from p are known, as well as bounds on the number of moves needed to bring the distance below a desired threshold [Fil91, LPW08], but these bounds are intractable to compute in practice (for example, they involve computing eigenvalues of the transition-probability matrix, while we cannot even afford to enumerate the state space).

Because there are no general bounds on time to convergence, we must assess the quality of distribution for each constraint set individually. The measure of quality that ultimately matters is the degree to which the solutions, when applied as input stimuli, exercise the behavior being verified. This can only be determined by measuring the verification coverage empirically.

When MCMC methods are used for statistical estimation, the initial samples are typically discarded, the rationale being that the values generated before convergence may not be

representative of the target distribution. This practice is known as "burn-in". In the setting of constrained random verification, we can use all the solutions without waiting for convergence for two reasons. First, burn-in is often unnecessary, independent of the application—if we start from a state that we do not mind having as a sample, our samples are just as good as if we waited for convergence [Gey92, GeyOL]. In our setting, any solution to the constraints is usually a good sample, so we can start from any solution. Second, even in cases where the target distribution is not uniform and we happen to start from a low-probability state, an initial run of improbable samples does not reduce the quality of results. We are not computing expectations as in a typical MCMC application, so these samples do not introduce error; they stimulate behavior just as any other stimulus does, and applying them in simulation can only increase verification coverage.

Chapter 3

Stimulus Generation Based on Markov Chain Monte Carlo Methods

In this chapter we discuss how we combine the Metropolis-Hastings algorithm with other ingredients in order to solve constraints randomly for stimulus generation. Our approach is inspired by SampleSat [WES04], which generates random solutions to Boolean constraints by interleaving Metropolis moves with iterations of a local-search solver. Our method extends this concept to mixed Boolean and integer constraints.

For MCMC-based methods to be used effectively in practical verification, they must be adapted. Among the challenges presented by these methods is correlation between successive stimuli. This correlation is undesirable in simulation because it typically reduces the diversity of behaviors that are stimulated. Our algorithm includes techniques for reducing correlation.

3.1 Specifying Constraints

Our method operates on constraints specified in clausal form, a generalization of the conjunctive normal form (CNF) commonly used to express Boolean propositional constraints. Our constraint form is sufficiently expressive for many practical testbenches but also lends itself to efficient Metropolis-Hastings sampling. We use a combination of Boolean variables and integer variables of bounded domain (i.e., of fixed bit-width); in our application domain of hardware verification these correspond to the inputs of the DUT. The literals in the clauses can be Boolean, or they can be arithmetic relations on integer variables. Our method can handle linear relations and a small class of nonlinear relations, as described in this section.

The constraints as we describe them in this section do not depend on the state of the DUT, even though support for design-state dependence in constraints is important for practical verification. Our approach requires very little modification to incorporate design state, so we omit it for clarity in this chapter and postpone our discussion of it to Chapter 4.

3.1.1 Normal Form

Formally, let $x = (x_1, \ldots, x_m)$ be a vector of m Boolean variables and $y = (y_1, \ldots, y_n)$; $-2^{B-1} \leq y_i \leq 2^{B-1} - 1$ be a vector of n integer variables, where B is the maximum bit-width

of y_i. The constraints are specified as a formula φ described by the following grammar:

$$\begin{aligned}
\textit{formula} &: \textit{clause} \mid \textit{formula} \wedge \textit{clause} \\
\textit{clause} &: \textit{literal} \mid \textit{clause} \vee \textit{literal} \\
\textit{literal} &: x_i \mid \neg x_i \mid [\, \textit{expression relop Integer} \,] \\
\textit{relop} &: \leq \mid \geq \mid = \mid \neq \\
\textit{expression} &: \textit{term} \mid \textit{expression} + \textit{term} \\
\textit{term} &: \textit{Integer} \mid y_i \mid \textit{term} \cdot y_i
\end{aligned}$$

Because all expressions have integer values, we do not include distinct relation types for strict inequalities $<$ ($>$); these can be expressed by using non-strict inequalities $\leq$ ($\geq$) and decrementing (incrementing) the constant on the right-hand side by one.

As an example of a constraint formula, consider the ALU input constraints given in SystemVerilog in Figure 1.2. We repeat them here for easy reference:

```
rand enum { ADD, SUB, MUL, DIV } op;
rand bit signed [7:0]            a, b;
constraint valid {
  op == ADD -> (-128 <= a + b && a + b <= 127);
  op == SUB -> (-128 <= a - b && a - b <= 127);
  op == MUL -> (-128 <= a * b && a * b <= 127);
  op == DIV -> (b != 0);
}
```

Suppose that the ADD, SUB, MUL, and DIV operations are encoded with $x = (0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$, respectively. We express these constraints with the following formula:

$$\begin{aligned}
&[\, y_1 \geq -128 \,] \wedge \\
&[\, y_1 \leq 127 \,] \wedge \\
&[\, y_2 \geq -128 \,] \wedge \\
&[\, y_2 \leq 127 \,] \wedge \\
(x_1 \vee \;\; x_2 \vee &[\, y_1 + y_2 \geq -128 \,]) \wedge \\
(x_1 \vee \;\; x_2 \vee &[\, y_1 + y_2 \leq 127 \,]) \;\; \wedge \\
(x_1 \vee \neg x_2 \vee &[\, y_1 - y_2 \geq -128 \,]) \wedge \\
(x_1 \vee \neg x_2 \vee &[\, y_1 - y_2 \leq 127 \,]) \;\; \wedge \\
(\neg x_1 \vee \;\; x_2 \vee &[\, y_1 \cdot y_2 \geq -128 \,]) \;\; \wedge \\
(\neg x_1 \vee \;\; x_2 \vee &[\, y_1 \cdot y_2 \leq 127 \,]) \;\; \wedge \\
(\neg x_1 \vee \neg x_2 \vee &[\, y_2 \neq 0 \,])
\end{aligned}$$

This example demonstrates a common idiom for the normal form: When variables have bit-width B' which is less than the maximum B, we constrain their range with explicit clauses $[-y_i \leq 2^{B'-1}]$; $[y_i \leq 2^{B'-1} - 1]$.

Unlike decision procedures for word-level hardware verification, our semantics for arithmetic is based on unbounded integers rather than bit-vectors; that is, results do not overflow. We

could use a bit-vector semantics with reasonable overflow rules for operations on variables of different widths, but this would add complexity without changing the essence of our method. By using arbitrary-precision arithmetic, we gain the simplicity of having a single interpretation for each operator, regardless of the ranges of its operands.

3.1.2 Integer Literals

Our approach for applying the Metropolis-Hastings algorithm would work with arbitrary relations in the integer literals, as long as we chose suitable proposal distributions. However, for better performance we restrict integer relations to forms for which we can efficiently generate single-variable moves, in the matter of Gibbs sampling. This restriction enables faster convergence of the distribution.

Efficient single-variable moves are possible when we can solve explicitly for each y_i in the support of all relations. That is, we require that for each relation we can efficiently compute intervals of satisfying values for each variable while keeping other variables constant. The computation of satisfying intervals is illustrated in Figure 3.1. The points within the curve satisfy the relation $f(y_1, y_2) \leq 0$. For each value of y_1, we can compute an interval I_1 that contains the satisfying values of y_2; i.e., $I_1(y_1) = \{y_2 : f(y_1, y_2) \leq 0\}$. Likewise, for each value of y_2, we can compute one or two intervals containing the satisfying values of y_1.

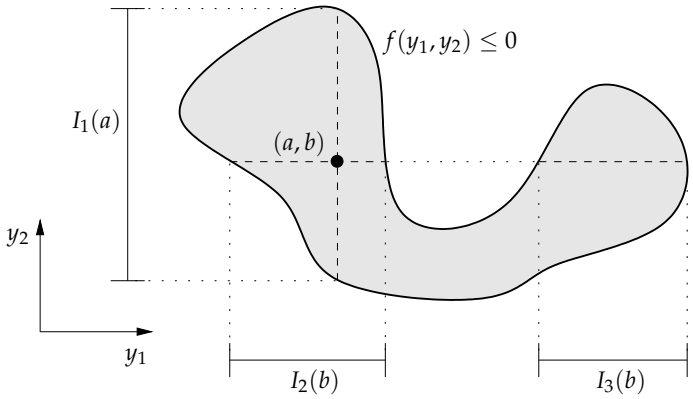


Figure 3.1: Illustration of how satisfying intervals are computed for a relation $f(y_1, y_2) \leq 0$.

One class of integer relations that meets our requirement for efficient solvability is the class of linear inequalities, equations, and inequations (relations with $\neq$). Solving these relations for a variable y_i simply involves moving terms to the right-hand side. For example:

$$y_1 + 2\,y_2 \leq 100 \quad \Longrightarrow \quad \begin{aligned} y_1 &\in \left[-2^{B-1}, 100 - 2\,y_2\right] \\ y_2 &\in \left[-2^{B-1}, \frac{100 - y_1}{2}\right] \end{aligned}$$

Besides linear relations, our approach also works well for many nonlinear constraints that are of practical relevance. In particular, we allow constraints in which the terms multiply several variables but are linear in the individual variables (i.e., every variable that appears in a term has degree 1). We call these relations *multilinear*. Multilinear relations can be solved as

easily as linear ones; the only difference is that the denominators of the interval bounds are not constants. For example:

$$y_1 y_2 \leq 100 \quad \Longrightarrow \quad \begin{cases} y_1 \in \left[-2^{B-1}, \dfrac{100}{y_2}\right] & \text{for } y_2 > 0 \\ y_1 \in \left[\dfrac{100}{y_2}, 2^{B-1}-1\right] & \text{for } y_2 < 0 \\ y_1 \in \left[-2^{B-1}, 2^{B-1}-1\right] & \text{for } y_2 = 0 \end{cases}$$

The fact that the solution of $y_1 y_2 \leq 100$ has three different forms is not a problem, because our moves change the value of only one variable at a time—we can choose the right form based on the current value of y_2.

Multilinear relations can express multiplicative constraints that cannot be encoded as linear relations. Efficient handling of multilinear relations is one significant advantage of our approach over BDD-based methods. Since BDDs for multiplication are exponentially large [Bry86], they cannot express multiplicative constraints in a scalable way.

In addition to multilinear relations, we can handle other nonlinear relations that can be solved. For example, relations whose terms are quadratic in their variables, such as $y_1^2 y_2 + 2y_1 y_3 \leq 12$, can be solved using the quadratic formula.

3.1.3 Translations of Various Constructs to Normal Form

Many constructs that are useful in practical constraints can be expressed in our normal form. In this section we describe transformations from several such constructs to clauses with multilinear relations. In the descriptions we reserve the symbol y_i for variables that appear in the final normal form. For integer variables that are transformed into combinations of several y_is, we use z_i. Term that may be an arbitrary expressions are denoted D or E.

Division operations: We rewrite a quotient ($\div$) or remainder (mod) operation whose dividend is a variable z_i using new variables y_q for the quotient and y_r for the remainder. The divisor, denoted D, can be an expression of any kind.

$$\begin{array}{ccc} z_i \div D & \Longrightarrow & y_q \\ z_i \bmod D & \Longrightarrow & y_r \\ z_i & \Longrightarrow & D \cdot y_q + y_r \end{array}$$

We add constraints to preserve the original range of z_i and to ensure that the result of division is defined and that $|y_r| < |D|$. We also constrain the sign of the remainder to match the semantics of division in C*, Verilog, SystemVerilog, and *e*—in these languages the quotient is rounded toward zero, so the remainder has the same sign as the dividend:

$$\begin{gathered} -2^{B-1} \leq D \cdot y_q + y_r \leq 2^{B-1} - 1 \\ D \neq 0 \\ [D > 0] \rightarrow [-D < y_r < D] \\ [D < 0] \rightarrow [D < y_r < -D] \\ [D \cdot y_q > 0] \rightarrow [y_r \geq 0] \\ [D \cdot y_q < 0] \rightarrow [y_r \leq 0] \end{gathered}$$

*According to the 1999 standard—in the previous standard the sign was undefined.

(The translation of the previous constraints to disjunctive clauses is straightforward; we omit it for clarity.)

Bit shifts: Several useful operations are defined in terms of the binary representations of integers rather than their numerical values. To encode these, we interpret values as being represented in a two's-complement system. Under this interpretation, a left shift («) of an expression E by k bits is equivalent to multiplication by 2^k:

$$E \ll k \qquad \Longrightarrow \qquad 2^k \cdot E$$

A right shift (») is equivalent to a division by 2^k, but we round the quotient down, not toward zero as with other divisions. The k least significant bits that are truncated are the remainder and the $B - k$ bits that are kept are the quotient. As with other divisions, we require that the first operand be a variable z_i.

$$\begin{array}{ccc} z_i \gg k & \Longrightarrow & y_j \\ z_i & \Longrightarrow & 2^k y_j + y_{j+1} \end{array}$$

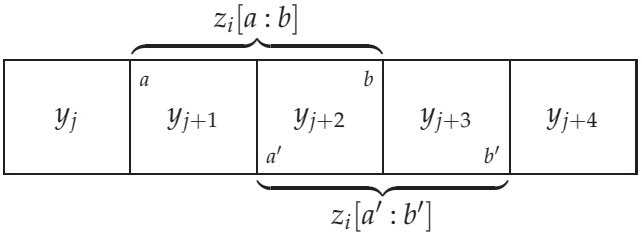
$$\begin{array}{c} -2^{B-k-1} \leq y_j \leq 2^{B-k-1} - 1 \\ 0 \leq y_{j+1} \leq 2^k - 1 \end{array}$$

Bit slices: The result of a bit slice $z_i[a:b]$ with $a > b$, also known as a bit selection (if $a = b$) or a part selection, is the value containing only the bits of z_i's value with indices between a and b, where bit 0 is the least significant bit. We encode bit slices similarly to right shifts, with new variables for the selected slice and the bits to its left and right:

$$\begin{array}{ccc} z_i[a:b] & \Longrightarrow & y_{j+1} \\ z_i & \Longrightarrow & 2^{a+1} y_j + 2^b y_{j+1} + y_{j+2} \end{array}$$

$$\begin{array}{c} -2^{B-a-2} \leq y_j \leq 2^{B-a-2} - 1 \\ 0 \leq y_{j+1} \leq 2^{a-b+1} - 1 \\ 0 \leq y_{j+2} \leq 2^b - 1 \end{array}$$

A variable can be sliced multiple times with different bit indices. If the slices do not overlap, each slice can still be encoded as a single variable. If the slices do overlap (e.g., $a' \geq b$), then each slice is translated into a sum of smaller slices. The following diagram shows the how the bits are partitioned in this case:



For example, if the slices $z_i[15:8]$ and $z_i[11:4]$ are both used in constraints and $B = 32$, z_i is encoded with four 4-bit slices and one 16-bit value:

$$\begin{array}{ccc} z_i[15:8] & \Longrightarrow & 2^4 y_{j+1} + y_{j+2} \\ z_i[11:4] & \Longrightarrow & 2^4 y_{j+2} + y_{j+3} \\ z_i & \Longrightarrow & 2^{16} y_j + 2^{12} y_{j+1} + 2^8 y_{j+2} \end{array}$$

$$\begin{gathered} -2^{15} \le y_j \le 2^{15} - 1 \\ -2^3 \le y_{j+1}, y_{j+2}, y_{j+3}, y_{j+4} \le 2^3 \end{gathered}$$

We interpret all slices as non-negative. This means that any slice containing the most significant bit of a variable (the sign bit) must be treated specially, because the bit's contribution to the variable's value is negative (-2^{B-1} when the bit is 1). If such a slice is used in a constraint, we create a single-bit slice for the sign bit. For example, if $B = 32$, the slices $z_i[31:16]$ and $z_i[15:0]$ are translated as follows:

$$\begin{array}{ccc} z_i[31:16] & \Longrightarrow & 2^{15} y_j + y_{j+1} \\ z_i[15:0] & \Longrightarrow & y_{j+2} \\ z_i & \Longrightarrow & -2^{31} y_j + 2^{16} y_{j+1} + y_{j+2} \end{array}$$

$$\begin{gathered} 0 \le y_j \le 1 \\ 0 \le y_{j+1} \le 2^{15} - 1 \\ 0 \le y_{j+2} \le 2^{16} - 1 \end{gathered}$$

Bitwise logical operators: The result of a bitwise AND operation $z_i \mathbin{\mathbb{\wedge}} z_j$ is obtained by combining corresponding bits of the operands using the $\wedge$ operator, and likewise for bitwise OR ($\mathbb{\vee}$), XOR ($\underline{\mathbb{\vee}}$), and NOT ($\exists$). We encode these operations in terms of single-bit slices $z_i[k:k]$ and $z_j[k:k]$, written as $z_i[k]$ and $z_j[k]$ for simplicity:

$$\begin{array}{ccc} z_i \mathbin{\mathbb{\wedge}} z_j & \Longrightarrow & \displaystyle\sum_{k=0}^{B-1} 2^k z_i[k] z_j[k] \\ z_i \mathbin{\mathbb{\vee}} z_j & \Longrightarrow & \displaystyle\sum_{k=0}^{B-1} 2^k \big(1 - (1 - z_i[k])(1 - z_j[k])\big) \\ z_i \mathbin{\underline{\mathbb{\vee}}} z_j & \Longrightarrow & \displaystyle\sum_{k=0}^{B-1} 2^k \big(z_i[k] + z_j[k] - 2 z_i[k] z_j[k]\big) \\ \exists z_i & \Longrightarrow & \displaystyle\sum_{k=0}^{B-1} 2^k (1 - z_i[k]) \end{array}$$

Although constraints with bitwise logical operators can be expressed in our normal form, they are not well suited for our stimulus-generation method. The solution space of such constraints is complex and sparse relative to the state space, and our ability to handle word-level constraints provides little benefit in this case. BDD-based sampling will usually be a better method for these constraints.

3.1.4 Constraint Partitioning

One aspect of constraint solving that we do not address in this work is constraint partitioning. As described in previous work [YAPA04], variables and constraints can be partitioned into sets which are sampled separately. These techniques are orthogonal to the constraint-solving method, so they can be applied in conjunction with our approach as well. This can provide a major boost to generator efficiency.

3.2 Applying the Metropolis-Hastings Algorithm

The foundation of our method for sampling solutions to mixed Boolean and integer constraints is the Metropolis-Hastings algorithm. To apply the Metropolis-Hastings framework, we must define a state space, a target distribution over the states, and a proposal distribution.

3.2.1 Target Distribution

Our state space is the set of possible assignments to (x, y), i.e., $\mathbf{B}^m \times \mathbf{Z}_B^n$, where $\mathbf{Z}_B = [-2^{B-1} \,..\, 2^{B-1} - 1]$. (The notation $[a \,..\, b]$ denotes an integer interval.) That is, it includes the solutions to the constraint formula φ, called the *satisfying assignments*, as well as the members of the set $\{(x, y) : \varphi(x, y) = 0\}$, called the *unsatisfying assignments*. If any of the clauses give tighter bounds of a simple form, e.g., $a \leq y_i \leq b$, we can narrow the state space to $\cdots \times [a \,..\, b] \times \cdots$, but otherwise all assignments are included, regardless of whether they satisfy the constraints. Our rationale for this is the requirement of irreducibility discussed in Sections 2.1 and 2.2: All states must be reachable from all others. By allowing moves to unsatisfying assignments, we guarantee that the state space is fully connected, even when the set of solutions on its own is not suitable for Gibbs sampling.

Our target distribution is the product of two terms: one for user-specified biases and one to favor solutions over unsatisfying assignments. The bias term is supplied by the user as a function $\tilde{p}(x, y)$; it is the means for specifying a non-uniform distribution on the solutions. Since unsatisfying assignments are not reported to the user, their bias value is not determined by the user. Instead, we define $\tilde{p}(x, y) = 1$ when (x, y) is not a solution. When no bias is desired at all (i.e., the user wants a uniform distribution of solutions), we define $\tilde{p}(x, y) \equiv 1$.

We define the *cost function* $U(x, y)$ to be the number of clauses unsatisfied under the assignment (x, y). Then the target distribution p is

$$p(x, y) = \tilde{p}(x, y) e^{-U(x,y)/T}$$

where T is a parameter called the *temperature*. If we narrow the state space due to static bounds for a variable, as described previously, we effectively assign infinite cost to the values outside the bounds, resulting in zero probability for those values.

The exponential term in $p(x, y)$ is the (unnormalized) Boltzmann distribution used in the original Metropolis algorithm [MRR+53] and in many other MCMC methods, including SampleSat and simulated annealing [KGV83]. The general form of the Boltzmann distribution is

$$p(s) = \frac{1}{Z(T)} e^{-E(s)/T}$$

where $E(s)$ is the energy of state s and $Z(T)$ is a normalizing constant. In our case, the energy corresponds to the number of unsatisfied clauses, so solutions have the lowest energy.

The temperature T provides a means to control the balance between the time spent visiting unsatisfying assignments and the ability to move between separated regions of solutions. When the temperature is high, the distribution is smooth and we move easily between solution regions, but the total probability of non-solution states can be large compared to the total probability of solutions. Conversely, a low temperature biases the distribution toward solutions at the cost of effectively disconnecting solution regions. Becase low temperatures inhibit movement through the state space, they increase the correlation between consecutive solutions, although the correlation-reduction mechanisms described in Section 3.4 mitigate the problem.

3.2.2 Proposal Distributions

We propose only moves that change a single variable's value at a time. This restriction enables efficient computation of proposal distributions and fast convergence as discussed in Sections 2.2 and 2.3. We use different proposal distributions for different types of variables. A proposal for a Boolean variable x_i is simple; we just flip the value of x_i. This is the same way that SampleSat proposes moves.

For an integer variable y_i, we construct a proposal distribution from the ranges of values that satisfy individual relations. Not all of the relations that refer to y_i necessarily contribute to the distribution for the current move; some of them do not constrain y_i under the current assignment (x, y) because their clauses are already satisfied by other literals that do not depend on y_i. We call the relations that do constrain y_i the *active relations*. To identify them, we substitute the current values of $(x, y \setminus y_i)$ into all clauses in the formula φ and remove falsified literals and satisfied clauses. We call the resulting substituted clauses the *active clauses*. Our procedure for collecting the active clauses is shown in Algorithm 3.1. The notation $R|_{x,y\setminus y_i}$ in line 6 denotes the relation on y_i obtained by substituting the current values of $(x, y \setminus y_i)$ into R.

Algorithm 3.1 GetActive

Given: formula φ, variable index i, current assignment (x, y)

```
1:  A := ∅
2:  for each clause C ∈ φ do
3:      C' := ∅
4:      for each literal l ∈ C do
5:          if l is integer relation R and y_i ∈ Support(R) then
6:              C' := C' ∪ {R|_{x,y\y_i}}
7:          else if l is satisfied by (x, y) then
8:              skip to next C
9:      if C' ≠ ∅ then
10:         A := A ∪ {C'}
11: return A
```

The example in Figure 3.2 illustrates how the active clauses are obtained by substitution and simplification of φ.

We construct the proposal distribution $q(y_i|x, y \setminus y_i)$ by combining indicator functions for the active relations. Like the active clauses, the combination of indicators has a two-level structure: For each clause C, we take the pointwise max of the indicators for relations in C to get an indicator for the clause as a whole. Then we conjoin the clause indicators to create a

$$
\begin{array}{l}
[2 \le y_1] \ \wedge \ [y_1 \le 16] \ \wedge \\
[0 \le y_2] \ \wedge \ [0 \le y_3] \ \wedge \\
\big(x_1 \vee [y_1 + y_2 \le 12]\big) \ \wedge \\
\big(\neg x_1 \vee [y_1 + y_2 \le 16] \vee [y_1 - y_2 \ge 0]\big) \ \wedge \\
\big(\neg x_1 \vee [y_1 - y_2 \le 18] \vee [y_2 + 2y_3 \le 20]\big)
\end{array}
\qquad
\begin{array}{l}
x_1 = 1 \\
y_2 = 11 \\
y_3 = 4 \\
\Longrightarrow
\end{array}
\qquad
\begin{array}{l}
[2 \le y_1] \ \wedge \ [y_1 \le 16] \ \wedge \\
\\
\\
\big([y_1 \le 5] \vee [y_1 \ge 11]\big)
\end{array}
$$

Figure 3.2: Example of substitution and simplification to obtain the active clauses.

distribution. By choosing different forms for the indicator functions and conjunctive operations, we obtain different kinds of proposal distributions.

The first kind of proposal distribution resembles Gibbs sampling over the solution space (not over the full state space). However, while true Gibbs proposal distributions would use step functions for the indicators, we use "soft" indicators that place uniform probability on satisfying values and decay exponentially in the unsatisfying intervals. The soft indicator $\sigma_R(y_i)$ for an inequality $R \Leftrightarrow y_i \le c$ $(y_i \ge c)$, with softness parameter r, is given by:

$$\sigma_R(y_i) = \begin{cases} 1 & \text{if } y_i \le c \quad (y_i \ge c) \\ e^{-r|y_i - c|} & \text{otherwise} \end{cases} \tag{3.1}$$

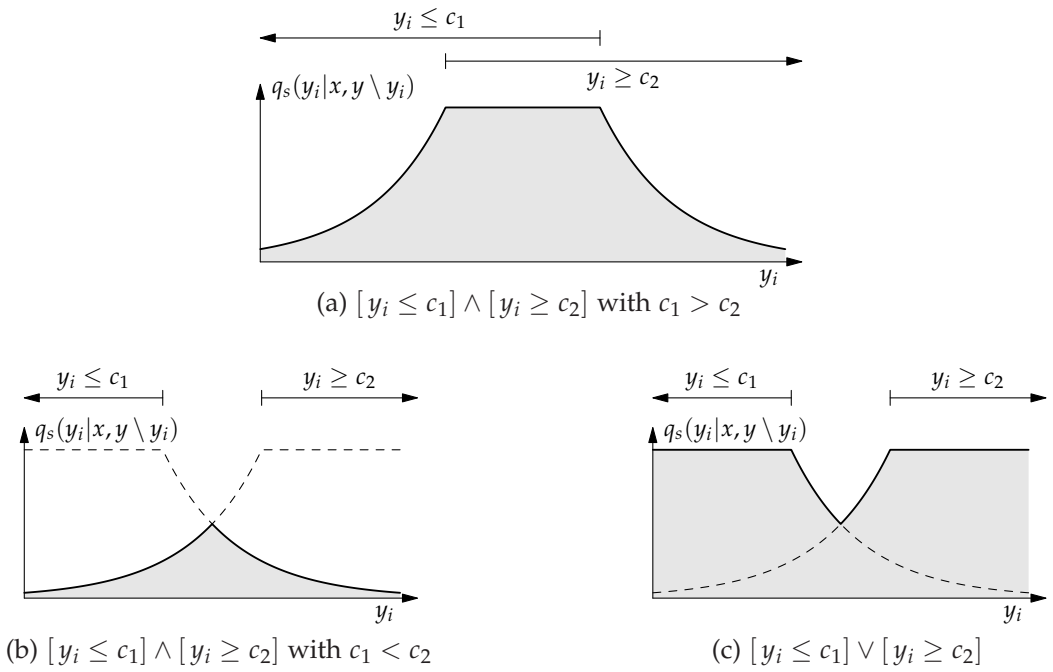
The indicator for relation $y_i = c$ $(y_i \ne c)$ is constructed from the combination $y_i \le c \wedge y_i \ge c$ $(y_i \le c - 1 \vee y_i \ge c + 1)$. We use min for the conjunctive combination. Thus, the overall form of the proposal distribution, which we denote q_s, is

$$q_s(y_i | x, y \setminus y_i) = \tilde{p}(y_i | x, y \setminus y_i) \min_{C \in A} \max_{R \in C} \sigma_R(y_i). \tag{3.2}$$

where A is the set of active clauses. We call q_s a **soft-SAT** proposal distribution because it acts as a relaxed indicator function for the active clauses, taking the value 1 for satisfying values of y_i (in the absence of user-specified bias). Figure 3.3 illustrates the distribution functions for different combinations of relations.

The purpose of the decaying segments in soft-SAT proposal distributions is to give non-zero probability to unsatisfying values, thus ensuring connectedness of the state space, while still favoring solutions. The exponential form of these segments and the use of min as the conjunctive operator are particularly motivated by the case shown in Figure 3.3b, where the range of satisfying values for y_i is empty. They are chosen to heuristically minimize the effort to get back to a solution from an unsatisfying state, e.g., after a flip of some x_j changes the set of active clauses for y_i. Figure 3.4 illustrates this case: After x_1 is flipped to 1, the state (x, y) is no longer in a satisfying region for y. Because the double-exponential proposal distribution is aligned with the new solution region, the first recovery move is likely to be to a point (x', y') that is only one move away from a solution (x'', y'').

We sample a new value y_i' from a soft-SAT proposal distribution by separating it into uniform and exponential segments, selecting a random segment, and sampling a value within the segment. Each segment has the form of one of the cases in Equation 3.1, multiplied by a scale factor h_j. The scale factor may come from the user-specified bias $\tilde{p}$ in Equation 3.2 or from the intersection of two exponential segments as shown in Figure 3.3b. To select a segment



(a) $[y_i \leq c_1] \wedge [y_i \geq c_2]$ with $c_1 > c_2$

(b) $[y_i \leq c_1] \wedge [y_i \geq c_2]$ with $c_1 < c_2$

(c) $[y_i \leq c_1] \vee [y_i \geq c_2]$

Figure 3.3: Examples of soft-SAT proposal distribution functions.

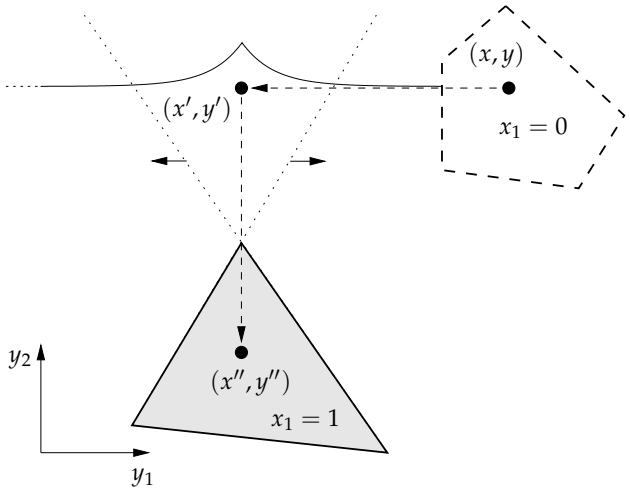


Figure 3.4: Motivating example for the soft-SAT proposal distribution illustrated in Figure 3.3b: A change in the active clauses on y through a flip from $x_1 = 0$ to $x_1 = 1$ results in an empty range for y_1. The soft-SAT distribution enables a quick return to a solution region within a few moves (e.g., the two moves shown).

from a distribution function with segments $q_1(y_i), \ldots, q_k(y_i)$ on intervals $[a_1 \,..\, b_1], \ldots, [a_k \,..\, b_k]$, we compute the unnormalized probability mass (or weight) w_j of each segment:

$$w_j = \begin{cases} \dfrac{h_j\left(1 - e^{-r(b_j - a_j + 1)}\right)}{1 - e^{-r}} & \text{if } q_j(y_i) = h_j e^{-r(y_i - a_j)} \text{ or } q_j(y_i) = h_j e^{-r(b_j - y_i)} \\ h_j(b_j - a_j + 1) & \text{if } q_j(y_k) = h_j \end{cases}$$

and select segment j with probability $w_j / \sum_{j'=1}^{k} w_{j'}$. Sampling the new value y_i' is straightforward if the segment is uniform. If the segment is exponential, we transform a uniform random variate using the cumulative distribution within the segment. Let $F(y_i)$ be the cumulative weight function for a congruent decreasing segment starting at 0:

$$\begin{aligned} F(y_i) &= \sum_{z=0}^{y_i} h_j e^{-rz} \\ &= \frac{h_j(1 - e^{-r(y_i+1)})}{1 - e^{-r}} \end{aligned}$$

We sample $\theta \in (0, w_j]$ uniformly and obtain y_i' from $F^{-1}(\theta)$:

$$\begin{aligned} d &= \left\lceil F^{-1}(\theta) \right\rceil \\ &= \left\lceil -1 - \frac{\ln\left(1 - \theta h_j^{-1}(1 - e^{-r})\right)}{r} \right\rceil \\ y_i' &= \begin{cases} a_j + d & \text{if } q_j(y_i) = h_j e^{-r(y_i - a_j)} \\ b_j - d & \text{if } q_j(y_i) = h_j e^{-r(b_j - y_i)} \end{cases} \end{aligned}$$

The runtime of this sampling procedure is linear in the number of segments because we have closed-form expressions for the weights of the segments and for $F^{-1}(\theta)$.

Soft-SAT proposal distributions tend to minimize the amount of time we spend visiting non-solution states, which helps efficiency, since it keeps the number of moves per solution small. However, they do not work well when solutions are found in multiple regions separated by wide gaps because they put low probability on moves through the gaps.

To move more easily between widely separated regions of solutions, we use a second kind of proposal distribution that is based on the number of unsatisfied clauses rather than the distance from satisfying values. This **cost-based** proposal, denoted q_c, is the Gibbs proposal distribution for our target distribution:

$$\begin{aligned} q_c(y_i | x, y \setminus y_i) &= p(y_i | x, y \setminus y_i) \\ &= \tilde{p}(y_i | x, y \setminus y_i) e^{-U(y_i | x, y \setminus y_i)/T} \end{aligned} \tag{3.3}$$

To construct q_c, we first build up the conditional cost function $U(y_i | x, y \setminus y_i)$ from indicator functions $\chi_R(y_i)$ for the active relations:

$$\chi_R(y_i) = \begin{cases} 1 & \text{if } y_i \text{ satisfies } R \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

$$U(y_i | x, y \setminus y_i) = \sum_{C \in A} \left(1 - \max_{R \in C} \chi_R(y_i)\right) \tag{3.5}$$

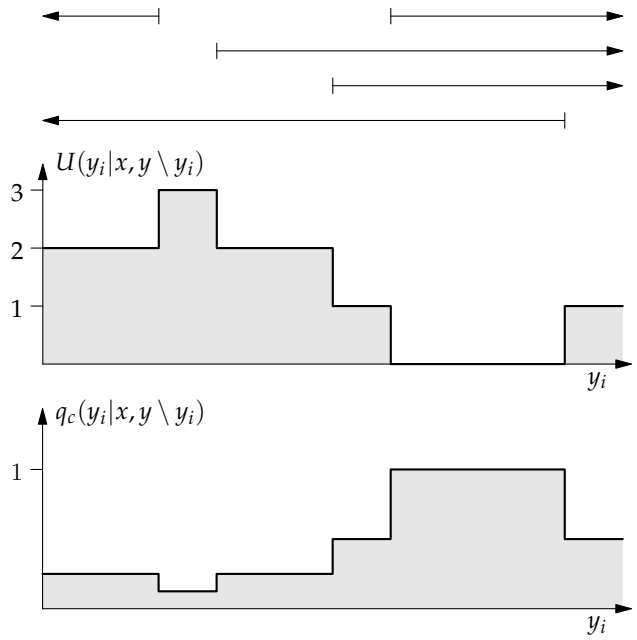


Figure 3.5: Illustration of cost function U and cost-based proposal distribution q_c.

The cost function U and corresponding distribution function q_c are illustrated in Figure 3.5. Strictly speaking, U as computed by Equation 3.5 does not always give the true cost, because it omits unsatisfied clauses which do not depend on y_i. However, the factors for the missing cost cancel each other in the Metropolis-Hastings acceptance ratio in Equation 2.2, so the omission does not change the result.

We sample the new value y_i' from a cost-based proposal distribution using the same procedure as for a soft-SAT proposal. The only difference is that cost-based distribution functions do not have exponential segments; all segments are uniform.

Our proposal distributions meet the requirements for convergence given in Section 2.2 with one exception: When all variables are Boolean and the target distribution is uniform over $\mathbf{B}^m$, the Markov chain implemented with the basic Boolean moves is periodic—it alternates between solutions containing odd and even numbers of 1s. We address this problem by allowing "lazy" moves: We provide a laziness parameter p_z, and with probability p_z we use the current state, unmodified, as the next state.

To confirm that cost-based proposals move more easily between separated solutions than soft-SAT proposals do, we compared them in experiments. The results, given in Section 3.6.2, demonstrate this advantage. Nevertheless, cost-based proposals are not always better, as shown by the following example.

Consider a set of constraints whose solutions lie inside an n -dimensional simplex:

$$0 \leq y_1, \ldots, y_n \leq 10n$$
$$\sum_{i=1}^{n} y_i \leq 10n$$

Suppose that the generator starts in the state where $y_i = 5$ for all i. Without loss of generality, let y_1 be the first variable chosen for a move. The current range of satisfying values for y_1 is $[0\,..\,10n - 5(n-1)] = [0\,..\,5n+5]$. If the value of the temperature parameter T is not too low, the total probability of the unsatisfying values under a cost-based distribution is fairly high, e.g., about 0.23 if $n = 10$ and $T = 1$. Suppose that the new value proposed and accepted for y_1 is, in fact, outside the satisfying range.

For the next move, the variable selected is likely different (assuming $n > 2$). Let this variable be y_2, and let the range of its satisfying values be $[0\,..\,b]$. We have $b = 10n - 5(n-2) - y_1$. Because y_1 took an unsatisfying value in the last move, we know that $y_1 > 5n + 5$, so $b \leq 5$. Thus y_2 is even more likely to take an unsatisfying value than y_1 was. Moreover, for most of the possible values of y_1 the satisfying range for y_2 is empty, e.g., 40 out of 45 values if $n = 10$. In this case the cost is 1 for all values in $[0\,..\,10n]$, so they all have the same probability, and the generator is likely to select a value that leads it farther from solutions.

This example demonstrates the weakness of the cost-based proposal distribution q_c: It has no bias toward the values that are more likely to lead to a solution. The soft-SAT proposal distribution q_s does have this bias, and it tends to help the generator recover when it wanders far from solutions. For this reason, we use both proposal types in our algorithm, selecting one of them at random in each move.

3.3 Increasing Efficiency with Local Search

The Metropolis-Hastings algorithm provides a solid foundation for our stimulus-generation approach, but sometimes it can spend too much time visiting unsatisfying states without reaching a solution. In SampleSat [WES04], this weakness is addressed by the addition of another kind of move. The additional moves are iterations of the Walksat local-search algorithm for Boolean satisfiability [SKC93]. In our approach we take this idea and extend it to integer constraints, thus strengthening the solving power of our stimulus generator and reducing the time between solutions.

To perform a local-search move, we select an unsatisfied clause uniformly at random and attempt to satisfy it by flipping the value of one of its literals. Flipping a Boolean literal is simple; we just complement the value of its variable. To flip an integer literal, we sample a new value for one of the variables in the support of the relation. If the new value does not satisfy the relation, the flip fails and the clause remains unsatisfied (but we do not revert to the previous assignment).

Local-search moves have random and greedy variants. In a random move, we choose the literal to flip uniformly at random. In a greedy move, we flip the literal that maximizes the number of satisfied clauses. Our use of integer constraints gives us an extra degree of freedom compared to Boolean Walksat: We can choose among multiple variables in the support of each integer literal, as well as choosing among the literals. We use this freedom by varying the greediness of literal choice and of variable choice independently; we may choose the best variable for a random literal or select the best literal given random choices of variables.

Once we have chosen an integer literal—i.e., a relation R—and a variable y_i, sampling a new value is very similar to a proposal for a Metropolis-Hastings move. We construct a distribution $q(y_i)$ from the active relations for y_i using the soft-SAT and cost-based proposal distributions as building blocks, as well as indicator functions for R. We give special treatment to any clauses that unconditionally constrain the value of y_i. These clauses define a *static range* for y_i. We

construct an indicator function $\chi_S(y_i)$ for the static range:

$$\chi_S(y_i) = \min_{C \in S} \max_{R' \in C} \chi_{R'}(y_i) \tag{3.6}$$

where S is the set of clauses referring only to y_i and $\chi_{R'}(y_i)$ is an indicator function for relation R', exactly as in Equation 3.4.

The new value for y_i must be in the static range. If none of the values in the static range satisfy R, we use a soft indicator for R to bias the distribution, similarly to a soft-SAT proposal distribution on unsatisfying values:

$$q(y_i) = \chi_S(y_i) \cdot \sigma_R(y_i) \tag{3.7}$$

Figure 3.6a illustrates the construction of $q(y_i)$ for this case.

If some values in the static range do satisfy R, we restrict the distribution $q(y_i)$ to those that satisfy as many other clauses as possible. These are the values with the lowest cost or the greatest target probability. Since the cost-based proposal distribution q_c is the conditional form of the target distribution, we can identify these values by examining q_c. We construct an indicator function $\chi^*(y_i)$ for them as the basis for $q(y_i)$:

$$w^* = \max_{y_i} \left(q_c(y_i | x, y \setminus y_i) \cdot \chi_R(y_i) \cdot \chi_S(y_i) \right) \tag{3.8}$$

$$\chi^*(y_i) = \begin{cases} 1 & \text{if } q_c(y_i | x, y \setminus y_i) \cdot \chi_R(y_i) \cdot \chi_S(y_i) = w^* \\ 0 & \text{otherwise} \end{cases} \tag{3.9}$$

The minimum-cost values for y_i satisfy the static-range constraints and the selected relation R, but they may fail to satisfy other clauses. To help satisfy those clauses in future

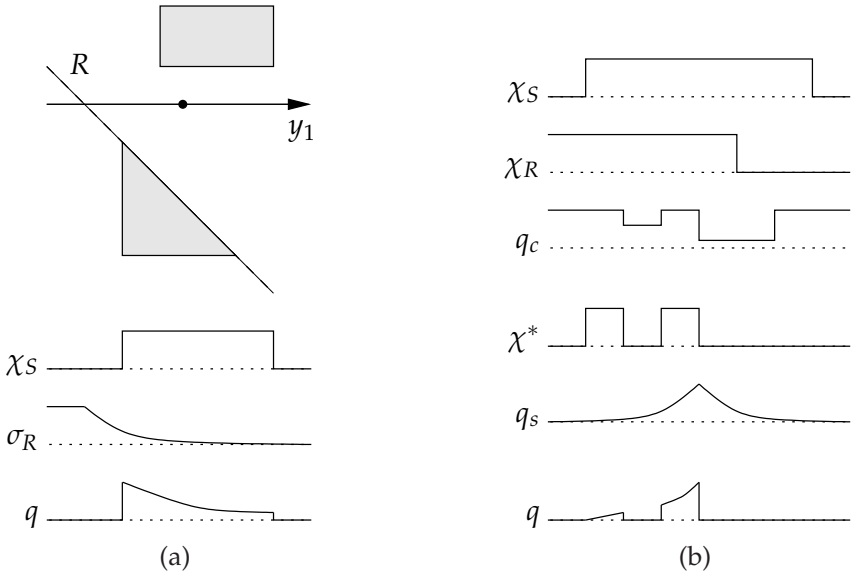


Figure 3.6: Illustrations of the construction of the distribution $q(y_i)$ for local-search moves: (a) using Equation 3.7, (b) using Equations 3.8–3.10.

moves, we include the soft-SAT proposal distribution q_s as a bias in the distribution. The overall distribution for sampling the new value is

$$q(y_i) = \chi^*(y_i) \cdot q_s(y_i | x, y \setminus y_i) \tag{3.10}$$

Figure 3.6b illustrates the construction of $q(y_i)$ according to Equations 3.8–3.10.

Although our local-search moves are based on the same distributions as our Metropolis-Hastings moves, they do not use the acceptance rule, so they distort the stationary distribution away from the target distribution.

3.4 Reducing Correlation

A fundamental property of Markov-chain-based sampling is the high correlation between consecutive states. Correlation decreases the usefulness of stimuli for verification, so the sequence of assignments generated by a MCMC method like ours cannot be applied directly to the DUT. To reduce correlation to a negligible level, we keep a pool of independent states (i.e., states from independent Markov chains) and switch to a new state, selected at random from the pool, each time we find a solution. The state pool is implemented as an array of pointers, so a state switch is carried out inexpensively by a simple pointer swap. Figure 3.7 illustrates the random selection of a state and the writeback of the resulting solution to the same slot in the pool.

We tested the effect of different pool sizes on correlation in experiments. The results are given in Section 3.6.3.

An alternate method for reducing correlation is subsampling or decimation, in which every kth solution is used and the rest are discarded. Decimation requires less memory than our state pool, but it can reduce the throughput by a factor of k, while our pooling technique preserves throughput unchanged. The two techniques are orthogonal; they can be used in combination.

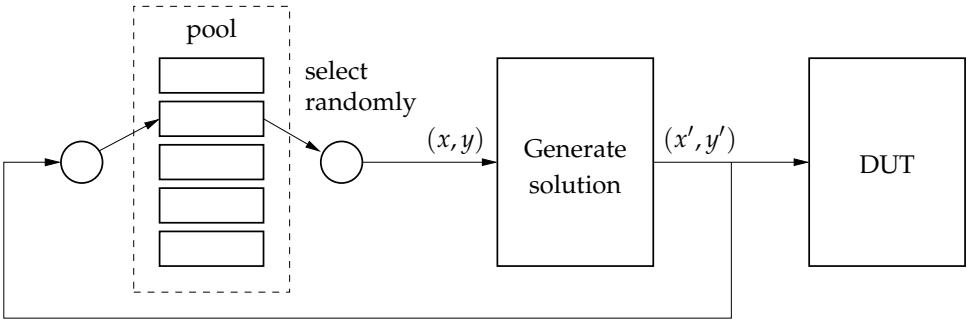


Figure 3.7: Illustration of data flow with the state pool: At the beginning of each solution cycle, a random state (x, y) is selected from the pool. The solution (x', y') derived from this state is applied to the DUT as a stimulus and written back to the pool.

3.5 Overall Generation Algorithm

In this section we give pseudocode for our algorithm for MCMC-based stimulus generation, showing explicitly how we combine the elements described in the previous sections.

Algorithm 3.2 shows the routine for initialization of the state pool. Algorithm 3.3 is the top-level routine for generating a solution. Both techniques for reducing correlation—state pooling and decimation—are implemented in this routine.

Algorithm 3.4 is the GenerateOne routine, which produces a single solution given an initial assignment (x^0, y^0). We begin with a single Metropolis-Hastings move to a new state (x, y). (For simplicity, we omit the -Hastings qualifier in the code, and we use the same shorthand throughout the remainder of this chapter.) If the state is not a solution, we perform a series of *recovery moves* until we satisfy the constraints or exceed a limit L on the number of moves. Each recovery move may be a Metropolis move or a local-search move. The probability of the former is initially p_{MH0} and decreases exponentially as the recovery time increases; our rationale for this is that the urgency of finding a solution increases over time. The move limit bounds the time

Algorithm 3.2 Initialize

Given: state pool $(s^1, \ldots, s^M)$

1: **for** $i := 1$ to M **do**
2: select (x, y) uniformly at random from state space
3: $s^i := (x, y)$

Algorithm 3.3 Generate

Given: state pool $(s^1, \ldots, s^M)$; decimation period D

1: select $i \in [1 \,..\, M]$ uniformly at random
2: $(x, y) := s^i$
3: **for** $t := 1$ to D **do** [*Decimate*]
4: $(x, y) :=$ GenerateOne(x, y)
5: $s^i := (x, y)$
6: **return** (x, y)

Algorithm 3.4 GenerateOne

Given: formula $\varphi(x, y)$; starting state (x^0, y^0); move limit L; move-type parameter p_{MH0}; rate parameter γ

1: $(x, y) :=$ MetropolisMove(x^0, y^0)
2: **for** $t := 1$ to $L - 1$ **do** [*Recovery moves*]
3: **if** $\varphi(x, y)$ **then**
4: **return** (x, y)
5: $p_{\text{MH}} := p_{\text{MH0}} e^{-\gamma(t-1)}$
6: **with** probability p_{MH} **do**
7: $(x, y) :=$ MetropolisMove(x, y)
8: **else**
9: $(x, y) :=$ LocalSearchMove(x, y)
10: Fail("Too many moves")

spent in the recovery phase. When we exceed the limit, we give up and report failure.

Instead of failing after exceeding the move limit, we could fall back on a stronger solution method, such as an SMT solver. However, the implementation that we used for our experiments did not include such a solver.

The routine for Metropolis moves is shown in Algorithm 3.5. For a normal (non-lazy) move, we first choose a variable at random. If the variable is Boolean, we simply flip its value. For an integer variable, we choose a proposal type randomly, construct a proposal distribution q, and sample a new value from it. In both cases, we compute the Metropolis-Hastings acceptance ratio α from the user-specified bias $\tilde{p}$, the proposal distribution, and the change in the number of unsatisfied clauses. Finally we apply the acceptance rule to make the proposed move or keep the current state.

Algorithms 3.6, 3.7, and 3.8 are the routines for local-search moves. Algorithm 3.6 shows the selection of an unsatisfied clause and a literal within the clause. Algorithm 3.7 is the routine for flipping a particular Boolean or integer literal. The greedy or random choice of a variable in the support of an integer relation is similar to the choice of a literal within the selected clause, except that some variables may not be considered for selection. We consider only the variables whose values have changed since the relation was most recently satisfied. This restriction is

Algorithm 3.5 MetropolisMove

Given: assignment (x, y); laziness parameter p_z; proposal parameter p_s; temperature T

with probability p_z **do**
  return (x, y)
select variable z from $\{x_1, \ldots, x_m, y_1, \ldots, y_n\}$ uniformly at random
if z is Boolean x_i **then**
  $x_i' := \neg x_i$
  $(x' \setminus x_i',\ y') := (x \setminus x_i,\ y)$
  $Q := 1$
else [z *is integer* y_i]
  $A :=$ GetActive(i, x, y) (see Algorithm 3.1)
  with probability p_s **do**
    construct soft-SAT proposal distribution $q_s(y_i | x, y \setminus y_i)$ from A (see Equation 3.2)
    $q := q_s$
  else
    construct cost-based proposal distribution $q_c(y_i | x, y \setminus y_i)$ from A (see Equation 3.3)
    $q := q_c$
  sample y_i' with distribution $q(y_i' | x, y \setminus y_i)$
  $(x',\ y' \setminus y_i') := (x,\ y \setminus y_i)$
  $Q := q(y_i | x, y' \setminus y_i')\, q(y_i' | x, y \setminus y_i)^{-1}$
$\tilde{P} := \tilde{p}(x', y')\, \tilde{p}(x, y)^{-1}$
$\Delta U := U(x', y') - U(x, y)$
$\alpha := \min\{1, \tilde{P} Q e^{-\Delta U / T}\}$
with probability α **do**
  return (x', y')
else
  return (x, y)

Algorithm 3.6 LocalSearchMove

Given: formula φ; assignment (x, y); greediness parameter p_g

select unsatisfied clause $C \in \varphi$ uniformly at random
with probability p_g **do** [*Choose literal greedily*]
  for each literal $l_i \in C$ **do**
    $(x^i, y^i) :=$ FlipLiteral(l_i, x, y)
  $i^* := \arg\min_i U(x^i, y^i)$
  return (x^{i^*}, y^{i^*})
else [*Choose literal randomly*]
  select literal $l_i \in C$ uniformly at random
  return FlipLiteral(l_i, x, y)

Algorithm 3.7 FlipLiteral

Given: literal l, assignment (x, y); greediness parameter p_g

if l is Boolean x_i or $\neg x_i$ **then**
  $x_i' := \neg x_i$
  $x' \setminus x_i' := x \setminus x_i$
  return (x', y)
else [l *is integer relation* R]
  $I := \{i : y_i \in$ Support(R) and y_i's value has changed since R was last satisfied$\}$
  with probability p_g **do** [*Choose variable greedily*]
    for each index $i \in I$ **do**
      $y^i :=$ FlipRelation(R, i, x, y)
    $i^* := \arg\min_i U(x, y^i)$
    return (x, y^{i^*})
  else [*Choose variable randomly*]
    select index $i \in I$ uniformly at random
    $y' :=$ FlipRelation$(R, i, x, y))$
    return (x, y')

motivated by the case where a relation refers to a large number of variables. Once the relation becomes unsatisfied by a change in one variable y_i, it may be difficult to satisfy it by changing the value of another variable y_j while still satisfying y_j's static-range constraints. For example, suppose that the relation is $\sum_{i=1}^{100} y_i \leq a$ and all the variables have static constraints $y_i \geq 0$, and that y_1 takes a large value that brings the sum above a. The probability of choosing y_1 again from among all 100 variables is low; to satisfy the relation we would likely have to lower the values of many variables, which would require many moves.

Algorithm 3.8 is the routine for flipping an integer relation once the variable for the flip has been chosen. We construct a distribution function $q(y_i)$ from indicator functions and proposal distributions as described in Section 3.3, then sample the variable's new value from that distribution.

Algorithm 3.8 FLIPRELATION

Given: relation R, variable index i, assignment (x, y)

construct indicator $\chi_S(y_i)$ for static-range clauses (see Equation 3.6)
if $\chi_R(y_i) \cdot \chi_S(y_i) = 0$ for all y_i **then**
    $q(y_i) := \chi_S(y_i) \cdot \sigma_R(y_i)$
else
    $A :=$ GETACTIVE(i, x, y)
    construct cost-based proposal distribution $q_c(y_i|x, y \setminus y_i)$ from A
    $w^* := \max_{y_i}(q_c(y_i|x, y \setminus y_i) \cdot \chi_R(y_i) \cdot \chi_S(y_i))$
    construct indicator $\chi^*(y_i)$ for $\{y_i : q_c(y_i|x, y \setminus y_i) \cdot \chi_R(y_i) \cdot \chi_S(y_i) = w^*\}$ (see Equation 3.9)
    construct soft-SAT proposal distribution $q_s(y_i|x, y \setminus y_i)$ from A
    $q(y_i) := \chi^*(y_i) \cdot q_s(y_i|x, y \setminus y_i)$
sample y'_i with distribution $q(y'_i)$
$y' \setminus y'_i := y \setminus y_i$
return y'

3.6 Experimental Evaluation

We implemented our MCMC-based stimulus-generation algorithm in a program called Ambigen (A mixed Boolean/integer generator). In this section we present experimental results from [KK07] and [KK09] to demonstrate that our algorithm is competitive with other methods and to validate some of our design decisions.

3.6.1 Comparison to Other Stimulus-Generation Methods

For comparison to our method, we implemented the BDD- and DPLL-based sampling algorithms described in Section 1.2. We selected these two approaches because of their complementary characteristics: The BDD-based method is fast and its distribution is uniform, while the DPLL-based method is more robust. In this section we compare the performance, robustness, and distribution of both methods with our algorithm using results from [KK07].

To compare the algorithms, we used each of them to generate solutions to several benchmark constraint sets, giving each solution equal probability. We derived a few of our benchmarks (T1–T3) from simple verification scenarios, such as the ALU in Figure 1.2, and extracted the others (I1–I4) from industrial verification instances. For the BDD- and DPLL-based generators, we converted the mixed Boolean and integer constraints to purely Boolean form by synthesizing circuits that evaluate the constraints. We used OpenAccess Gear [XPC+05] for logic synthesis. The synthesized circuits use finite-precision arithmetic, in contrast with the arbitrary-precision arithmetic used in Ambigen; however, the synthesis tool chooses the bit-widths of intermediate results to avoid overflows, so the difference in precision does not affect the semantics of the constraints.

For the BDD generator, we built BDDs for the output functions of the constraint evaluation circuits. In the variable order of each BDD, we put the control variables first and interleaved the bits of the data variables, with the least significant bits first. We used CUDD [Som05] as our BDD package.

For the DPLL generator, we converted the circuits, with their outputs asserted, to Boolean CNF. We adapted the solver MiniSat [ES03] to use the method described in Chapter 1:

	Boolean/integer CNF					BDD		Boolean CNF	
bench-mark	Boolean vars	integer vars	clauses	literals	multi-linear	vars	nodes	vars	clauses
T1	2	2	11	26	2	34	*	2 634	7 855
T2	10	4	33	60	0	46	6 073	330	961
T3	5	67	146	162	0	539	84 530	6 640	18 347
I1	0	6	30	30	0	116	143 848	879	2 444
I2	1	9	28	34	6	127	*	29 290	87 684
I3	0	30	214	455	0	232	*	1 080	2 954
I4	0	213	497	514	0	244	565 977	7 428	21 827

Table 3.1: Characteristics of benchmarks in different forms: our Boolean/integer normal form (used by Ambigen), binary decision diagrams, and Boolean CNF (used by DPLL-based generator). Memory blowups are indicated by *.

We selected a random pre-assignment and variable order, and each time the solver reached a decision point we used the next available pre-assigned value. When the solver found a conflict, we allowed it to backtrack and find a solution using its usual decision heuristic.

The characteristics of each version of the benchmarks are given in Table 3.1. For the benchmarks in our mixed Boolean and integer normal form, the table lists the numbers of Boolean variables, integer variables, clauses, literals (both Boolean and integer), and multilinear relations (i.e., multiplicative, not linear ones). For the constraints in Boolean CNF, the table lists the numbers of variables and clauses. For the constraint BDDs, the table lists the number of variables and the number of nodes. Each entry with a * indicates that the BDD could not be built with a memory limit of 1 GB.

We used each generation method to generate 1 million solutions for each benchmark, with a time limit of 10 000 seconds. The results of these experiments are shown in Table 3.2. For Ambigen, the table lists the runtime, the number of unique solutions, and the number of moves per solution (including both Metropolis and local-search moves). For the BDD-based generator, the table lists the total runtime and the time needed to build the BDD. Entries with asterisks in columns 5 and 6 indicate that no solutions were generated because of memory blowup. For the DPLL-based generator, the table lists the runtime and the total number of samples generated. The number of unique samples for the BDD- and DPLL-based generators is not reported because almost all the solutions generated were unique; there were duplicate solutions in only three cases, and at most 100 duplicates in each of these.

A comparison of the runtimes shows that Ambigen has similar performance to the BDD-based generator, and that both of these are faster than the DPLL-based algorithm by 1–3 orders of magnitude. However, Ambigen is more robust than the BDD-based algorithm, because it does not suffer from memory blowup. Note that the memory blowup is not limited to benchmarks with multiplicative constraints. In preliminary experiments, we found this was true even when we built the BDDs with other variable orders.

Column 4 of Table 3.2 lists the average number of moves per solution. We computed this value as the ratio of the total number of moves to the total number of solutions, including those that were decimated (thus the denominator is $10^6 D$). This value is an important measure of Ambigen's ability to move through the solution space. For each benchmark, the number of

	MCMC (Ambigen)			BDDs		DPLL	
bench-mark	runtime (sec)	unique solutions	avg moves per solution	runtime (sec)	build time (sec)	runtime (sec)	solutions
T1	35	975 627	1.11	*	*	2 091	1 000 000
T2	49	858 590	1.56	41	< 1	423	1 000 000
T3	193	999 986	1.05	382	70	10 000	16 845
I1	112	992 391	1.40	111	1	4 409	1 000 000
I2	59	999 995	1.00	*	*	10 000	252 495
I3	179	999 979	1.19	*	*	10 000	854 484
I4	586	920 922	1.58	453	20	10 000	726 645

Table 3.2: Results of generating 1 000 000 random solutions with a time limit of 10 000 sec and a memory limit of 1 GB. Parameters for Ambigen: state-pool size $M = 1$, decimation period $D = 5$, move limit $L = \infty$, move-type probability $p_{\text{MH0}} = 0.5$, rate $\gamma = 0$, laziness $p_{\text{z}} = 0$, proposal-type probability $p_{\text{s}} = 1$, temperature $T = 1$, softness $r = 1$, greediness $p_{\text{g}} = 0.5$.

moves per solution is close to 1 (at most 1.58), indicating that the generator is most often able to find a solution in the initial Metropolis move; recovery moves are usually not needed.

The experiments described above are not well suited for comparing the output distributions of the algorithms. Each of the benchmarks has a solution space many orders of magnitude larger than the number of solutions generated in our experiments, so each solution could not be expected to occur more than once in a uniformly generated sample. Ambigen's distribution appears to be skewed because it generates some duplicates. However, some duplication should be expected, given the local-walk nature of the algorithm. This does not contradict the theory on the convergence to the target distribution, since the time for convergence can be very long for a large state space.

Decimation is one of the easiest ways to reduce the number of duplicate solutions. To investigate the effectiveness of decimation, we swept the decimation period D from 1 to 10 for the benchmark T2, which had the fewest unique solutions according to the results in Table 3.2. We kept other parameters, including the number of solutions generated, the same as in the previous experiments. The results are shown in Figure 3.8, including the percentage of generated solutions that were unique and the runtime spent on generation. (A large fraction of the runtime was spent printing the solutions; the times in the plot do not include this I/O time.) Increasing D from 1 to 5 raises the number of unique solutions from 32% to 81% of the total number of solutions, while the generation time increases by 2.5x. Continued increases of D bring diminishing returns—increasing D to 9 yields an additional 9% unique samples at the cost of almost 4x the original runtime.

Given appropriate choices for the parameters and enough time to converge, the output distribution of our algorithm can be brought arbitrarily close to the target distribution. This is not the case, in general, for the DPLL-based method. Suppose that the target distribution is uniform. If the number of solutions is not a power of 2, the backtracking strategy of DPLL cannot map the full Boolean space—from which the pre-assignments are uniformly selected—into the solution space in a uniform manner. Depending on the particular constraints, the mapping may not even be close to uniform. For example, we generated solutions to a small set of constraints similar to

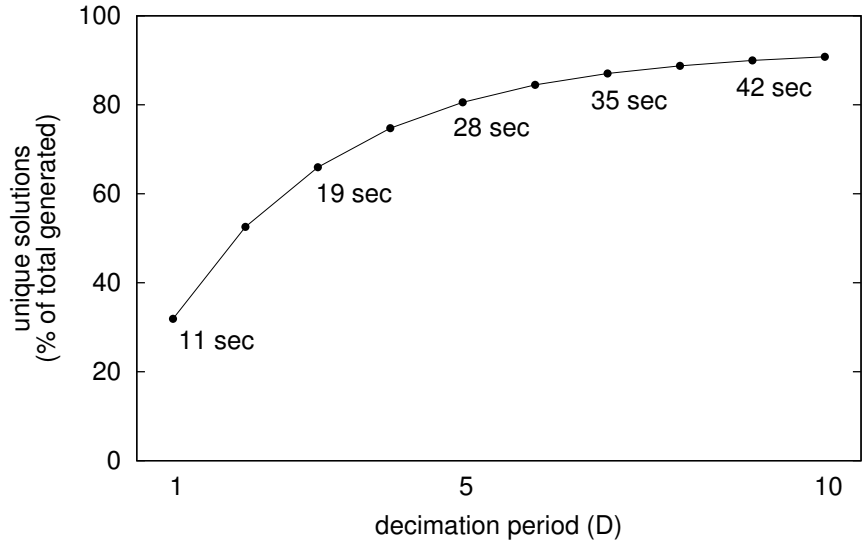


Figure 3.8: Decimation periods and percentages of unique solutions (relative to total solutions) generated by Ambigen for benchmark T2, along with the time spent in generation for selected periods (not including I/O time).

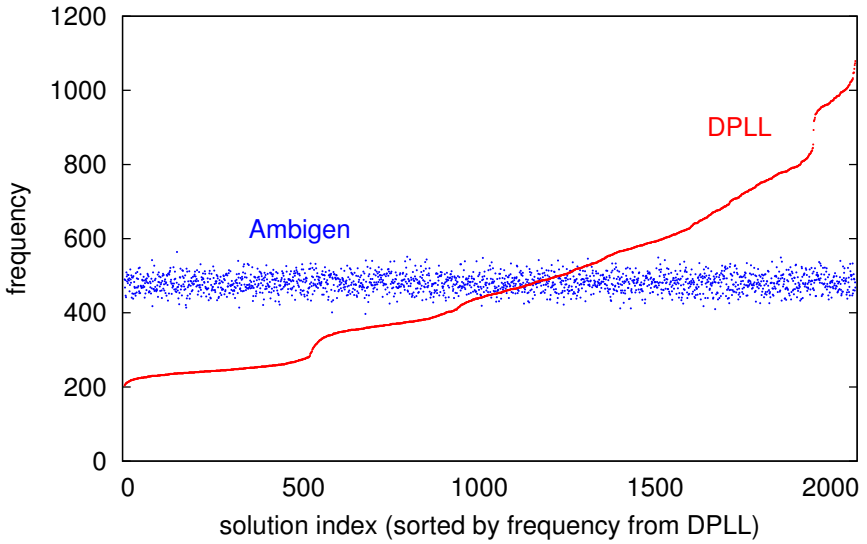


Figure 3.9: Absolute frequencies of all solutions to constraints $[y_1, y_2 \geq 0, y_1 + y_2 \leq 63]$ from Ambigen and DPLL-based generator. The total number of samples from each generator is 1 000 000.

the example in Figure 1.3. The resulting output distributions for Ambigen and the DPLL-based generator are shown in Figure 3.9. The ratio of the greatest frequency from DPLL to the least is more than 5, while the frequencies from Ambigen are limited to a narrow band.

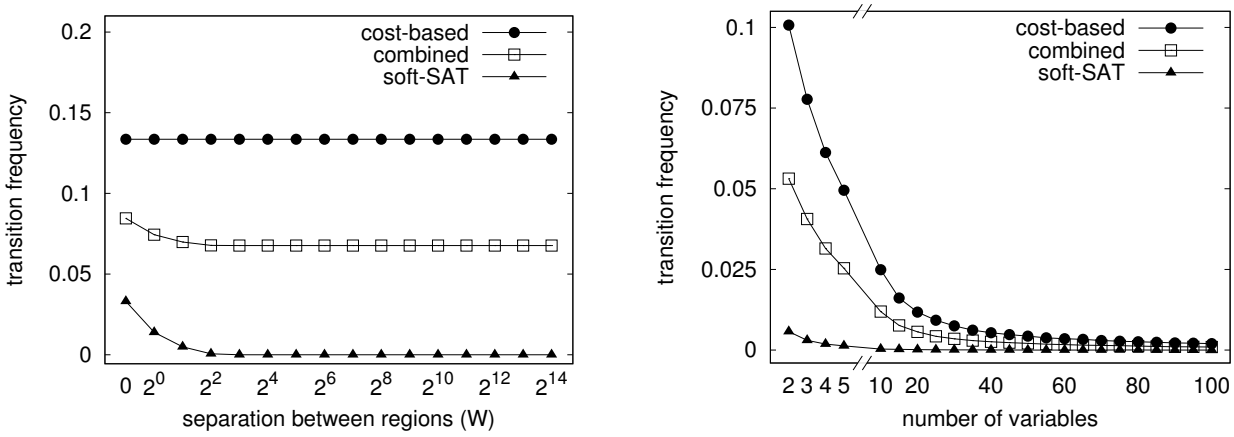
3.6.2 Comparison of Proposal Types

As we stated in Section 3.2.2, soft-SAT proposal distributions are not effective for moving between regions of solution spaces that are separated by wide multidimensional gaps (i.e., gaps that cannot be crossed in a single move). For this reason we developed cost-based proposal distributions. In this section we describe experiments that we performed to test the effectiveness of the cost-based distributions in moving between separated solution regions.

We generated two groups of constraint sets. The solution sets for the benchmarks in the first group consist of two squares of width 8 separated by distance W in each dimension:

$$0 \leq y_1, y_2 < 16 + W$$
$$(y_1 < 8 \wedge y_2 < 8) \vee (y_1 \geq 8 + W \wedge y_2 \geq 8 + W)$$

We generated 100 000 solutions for each of $W = 0, 1, 2, 4, 8, \ldots, 2^{14}$ and for each type of proposal distribution. We disabled local-search moves to avoid distortion of the basic Metropolis-Hastings distribution. Figure 3.10a shows the relative frequencies of transitions between the regions that we observed, i.e., the frequencies with which consecutive solutions were from different regions, relative to the total number of solutions. The transition frequencies are constant with increasing distance for cost-based proposal distributions and decrease quickly to zero for soft-SAT proposals. Using both proposal types with equal probability gives intermediate transition frequencies.



(a) Separated squares

(b) Simplex corners

Figure 3.10: Relative frequencies of transitions between regions for different types of proposal distributions. Parameters: $M = 1$, $D = 1$, $L = \infty$, $p_z = 0$, $T = 1$, $r = 1$; for (a) $p_{\text{MH0}} = 1$ and $\gamma = 1$ (i.e., no local-search moves were used); for (b) $p_{\text{MH0}} = 0.5$ and $\gamma = 1$ (local-search moves had constant probability 0.5).

The second group of benchmarks has solution regions at the corners of n-dimensional simplices for $n = 2, 3, 4, 5, 10, 15, 20, \ldots, 100$. The constraints have the form:

$$0 \leq a_i y_i \leq b$$

$$\sum_{i=1}^{n} a_i y_i \leq b$$
$$\bigvee_{i=1}^{n} (y_i \geq c_i)$$

For each value of n, we generated 100 benchmark instances with random values of the coefficients, constraining them so that the solution regions would not be connected. For each benchmark instance we generated 100 000 solutions for each type of proposal distribution. Figure 3.10b shows the relative frequencies of transitions between regions. The generator moves between regions much more often when using cost-based proposal distributions than when using soft-SAT proposals. Even for the cost-based proposals, the transition frequency decreases rapidly with the number of variables; this fits with the fact that the average probability of an unsatisfying assignment decreases exponentially as the average cost increases.

The results from the experiments on these two groups of benchmarks demonstrate that cost-based proposal distributions are more effective for moving between disconnected regions than soft-SAT proposals, especially when the regions are widely separated.

3.6.3 Evaluation of State Pooling for Decorrelation

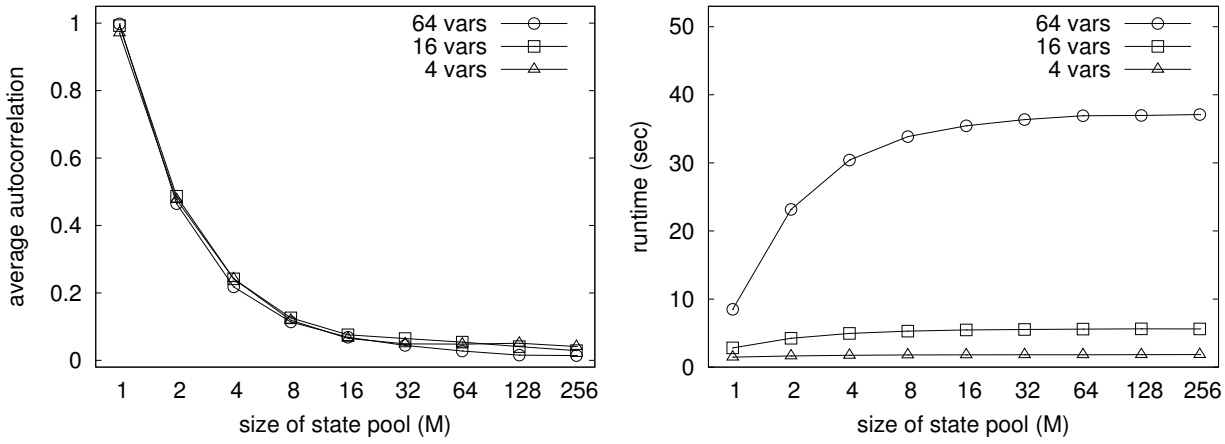
To test the effectiveness of state pooling at reducing serial correlation, we generated benchmarks with solution spaces consisting of points "sandwiched" between oblique hyperplanes:

$$-c \leq y_i \leq c$$
$$-b \leq \sum_{i=1}^{n} a_i y_i \leq b$$
$$b = \frac{c}{10}$$

Because the bounding hyperplanes are oblique—not orthogonal to any coordinate axis—and close together, the range of any single move is much less than the width of the solution space. As a result, successive solutions are highly correlated.

We generated 100 benchmark instances with random coefficients for each of $n = 4$, 16, and 64 and sampled 10 000 solutions for each instance with state-pool sizes $1, 2, 4, \ldots, 256$. Figure 3.11a shows the autocorrelation of the solutions, computed as an average of the autocorrelation for each variable. The correlation decreases quickly as the size of the state pool increases, showing the effectiveness of our technique.

Figure 3.11b shows the average runtimes of the instances for each state-pool size M. For $M = 1$, the runtime increases linearly with the number of variables, and this is expected, since each benchmark has constraints with all the variables in their support and these constraints are evaluated for every move. However, the trend of the runtimes for the 64-variable benchmarks requires further explanation. The rapid rise for small values of M and the disproportionately long runtimes in general are likely due to cache misses. This suggests a strategy for improving runtime: Subsample with a longer period and use a smaller state pool. Subsampling on the same position in the pool gives higher cache-hit rates than state pooling but reduces serial correlation similarly.



(a) (b)

Figure 3.11: Effect of state pooling on (a) average autocorrelation and (b) average runtime in seconds. Parameters: $D = 1$, $L = \infty$, $p_z = 0$, $T = 1$, $r = 1$, $p_{\text{MH0}} = 0.5$, $\gamma = 1$ (local-search moves had constant probability 0.5).

3.7 Summary

In this chapter we described how we apply the Metropolis-Hastings algorithm and how we adapt it to make it practical for stimulus generation. We use two types of proposal distributions for easy traversal of the state space and avoid wandering far from solutions. For quick recovery from unsatisfying states, we use additional, more directed moves that integrate the proposal distributions into iterations of a local-search SAT solver. We keep serial correlation low by switching among a pool of independent states. Our experiments demonstrate that our approach surpasses other methods in speed, robustness, and quality of distribution.

In the next chapter we describe several refinements that increase the capability of our algorithm and improve its performance.

Chapter 4

Refinements to Markov Chain Monte Carlo Stimulus Generator

In the previous chapter we presented the essential components of our approach for stimulus generation. After implementing it as described there, we worked with developers of an industrial testbench-automation tool to integrate our generator into their software. Our experience from the integration work motivated further research into refinements to make our generator more effective. In this chapter we describe two successful refinements: generation with externally controlled values and static variable elimination. We also present a dynamic form of variable elimination that handles arbitrary combinations of equality constraints and show results that demonstrate that our basic, unmodified algorithm works just as effectively on such constraints.

4.1 Generation of Stimuli Dependent on External Values

Chapter 3 presents our generation approach as having a one-way flow of information from the stimulus generator to the logic simulator. However, in practice the generator often uses information from other components of the testbench. Two scenarios in particular require this: First, input constraints for constrained random simulation often refer to the current state of the design under test (DUT). Second, the verification of complex protocols often requires sequential constraints, which specify conditions on sequence of stimuli, e.g., using PSL [Acc04] or SystemVerilog [SDF03]. Sequential constraints can be implemented by storing state in the testbench. We support these both scenarios by allowing *control variables* whose values are not generated randomly but set by the testbench to communicate the state of the DUT or the testbench itself.

Formally, let $u = (u_1, \ldots, u_{m'})$ be a vector of Boolean control variables and $v = (v_1, \ldots, v_{n'})$; $-2^{B-1} \leq v_i \leq 2^{B-1} - 1$ be a vector of integer control variables. We extend the state space to include assignments to the control variables in addition to the randomly generated vectors x and y; that is, a state in the extended state space is a tuple (u, v, x, y). For a given assignment to (u, v), we denote by $\Phi(u, v)$ the set of solutions for the constraint formula φ, i.e., $\Phi(u, v) = \{(x, y) : \varphi(u, v, x, y) = 1\}$. When sampling solutions to φ for a stream of control values $((u^1, v^1), \ldots, (u^t, v^t))$, we want a sequence of solutions $((x^1, y^1), \ldots, (x^t, y^t))$ with $(x^i, y^i) \in \Phi(u^i, v^i)$ whose distribution matches the desired user bias function $\tilde{p}(x, y|u, v)$. That is, each sample in the sequence is chosen independently of the other samples and with probability

proportional to its bias value:

$$\forall(x^i, y^i) : \Pr((x^i, y^i) = (x, y)) \;=\; \frac{\tilde{p}(x, y|u^i, v^i)}{\sum_{(x,y)\in\Phi(u^i,v^i)} \tilde{p}(x, y|u^i, v^i)}$$

where $\Pr(A)$ denotes the probability of event A.

The use of control variables effectively creates a changing sequence of constraints for the generator. Each new assignment to (u, v) selects a new constraint formula, namely, $\varphi|_{(u,v)=(u^i,v^i)}$. The previous values of (x, y) do not necessarily satisfy the new formula; in fact, they may be very far from a solution.

Our algorithm can handle control variables with very little modification. Within the context of an individual move, the state (u, v, x, y) with new control values looks just like any other state; it could have been reached via a Metropolis move instead of a new control assignment. Our generator already handles recovery from unsatisfying assignments; the only modification needed in the GenerateSolution procedure is to exclude u and v when a variable is chosen for a move.

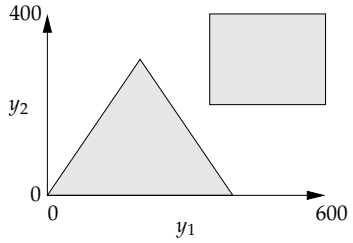
Our observation is that dependence on external values usually does not trigger a large amount of jumping between disjoint solution spaces in practice. That is, the solution sets $\Phi(u^i, v^i)$ and $\Phi(u^{i+1}, v^{i+1})$ tend to overlap significantly. In cases where the control variables do move frequently between values with substantially different solution sets, the distribution can be distorted significantly. Solutions that are closest (in number of moves) to the previous solution set are usually sampled more often.

4.2 Static Variable Elimination

Our generation algorithm tends to perform best on dense solution spaces—i.e, when a large fraction of the possible assignments are solutions—but constraints are not always specified in the densest possible way. This is not merely a hypothetical problem; when our generator was used in the industrial tool we observed that the translation of constraints into our normal form often used more variables than necessary, increasing the size of the state space. We can improve the performance of our algorithm by preprocessing such constraints to eliminate the extra variables. Because this transformation is carried out before any values are generated, we refer to it as *static variable elimination*.

Both Boolean and integer variables may be added in the process of translating constraints to clausal form. The translator may introduce a variable for each Boolean subexpression in order to ensure that arbitrary constraints can be expressed with linear complexity. Similarly, integer variables are sometimes introduced for integer subexpressions. For example, consider the following SystemVerilog constraints with solutions in two regions, as illustrated on the right:

```
rand bit  [8:0] y1;
rand bit  [9:0] y2;
constraint c {
  3*y1 >= 2*y2;
  3*y1 + 2*y2 <= 1200 || (y1 inside {[350:600]} &&
                          y2 inside {[200:400]});
}
```



A straightforward way to translate these constraints into clauses is to introduce new variables x_1, x_2, and x_3, constraining them to have the same values as the Boolean subexpressions, and add variable y_3 for the expression $3y_1 + 2y_2$:

$$[y_1 \geq 0] \quad (4.1)$$
$$[y_2 \geq 0] \quad (4.2)$$
$$[y_3 \geq 0] \quad (4.3)$$
$$[y_1 \leq 1023] \quad (4.4)$$
$$[y_2 \leq 511] \quad (4.5)$$
$$[y_3 \leq 3579] \quad (4.6)$$
$$[3y_1 \geq 2y_2] \quad (4.7)$$
$$[y_3 = 3y_1 + 2y_2] \quad (4.8)$$
$$[y_3 \leq 1200] \vee x_1 \quad (4.9)$$
$$\neg x_1 \vee x_2 \quad (4.10)$$
$$\neg x_1 \vee x_3 \quad (4.11)$$
$$x_1 \vee \neg x_2 \vee \neg x_3 \quad (4.12)$$
$$\neg x_2 \vee [y_1 \geq 350] \quad (4.13)$$
$$\neg x_2 \vee [y_1 \leq 600] \quad (4.14)$$
$$x_2 \vee [y_1 < 350] \vee [y_1 > 600] \quad (4.15)$$
$$\neg x_3 \vee [y_2 \geq 200] \quad (4.16)$$
$$\neg x_3 \vee [y_2 \leq 400] \quad (4.17)$$
$$x_3 \vee [y_2 < 200] \vee [y_2 > 400] \quad (4.18)$$

In theory, all variables added during translation to clausal form could be identified and eliminated. In practice, this is not a good idea. Eliminating Boolean variables can increase the number of clauses exponentially. For example, the constraint $\bigvee_{i=1}^{n} (x_i \wedge x_{i+n})$ can be expressed in $3n + 1$ clauses using n additional variables; without the extra variables it requires 2^n clauses.

We can eliminate some Boolean variables without increasing the number of clauses, while still preserving conjunctive normal form (CNF), by introducing new types of relations. In particular, when a variable x_i is constrained to be equivalent to a conjunction or disjunction of integer relations, we can replace occurrences of it with a relation that combines the argument relations. For example, if the clauses specify that $x_i \Leftrightarrow (y_j \geq a \wedge y_j \leq b)$, each literal x_i can be replaced by the relation $[a \leq y_j \leq b]$. The replacement relation is a single literal, not a pair of literals $[a \leq y_j]$ and $[y_j \leq b]$ whose conjunction must be distributed in order to maintain CNF. The two-sided inequality is not in our normal form, but it works just as well within our algorithm as one-sided inequalities, because we can efficiently construct the interval of satisfying values for it. We simply take the intersection of the intervals that satisfy the constituent inequalities $a \leq y_j$ and $y_j \leq b$. Likewise, for a disjunction $(y_j \leq a \vee b \leq y_j)$, we can construct the satisfying intervals as the union $[-2^{B-1} \,..\, a] \cup [b \,..\, 2^{B-1} - 1]$.

The composite relations resulting from eliminating Boolean variables can themselves be combined in further elimination steps. However, as relations become more complex, the efficiency of moves decreases, potientially offsetting the benefit of elimination. In preliminary experiments, we observed that our algorithm performed best when we combined only relations with the same variable support. For example, we do not eliminate x_i if $x_i \Leftrightarrow (y_j \leq a \wedge y_{j+1} \leq b)$.

We perform static elimination of Boolean variables by identifying patterns of clauses that specify equivalence between variables and combinations of integer relations. We also identify simple equivalence with Boolean or integer literals. Table 4.1 lists the patterns we search for and the equivalences they specify. When we find a pattern, we replace each occurrence of the variable x_i with its equivalent literal (or the complement, for occurrences of $\neg x_i$).

Elimination of integer variables does not increase the number of clauses nor the number of relations, so we do not restrict it based on variable support as we do with Boolean variables. We identify all clauses of the form $[y_i = f(y \setminus y_i)]$ and eliminate each such y_i.

Table 4.2 illustrates static variable elimination for the example given earlier. The second column shows how Clauses 4.9–4.12 are transformed as each variable is replaced by its equivalent

Pattern	Equivalence	Replacement for $\neg x_i$
$x_i \vee \neg l$ $\neg x_i \vee l$	$x_i \Leftrightarrow l$	$\neg l$
$x_i \vee \neg R_1 \vee \neg R_2$ $\neg x_i \vee R_1$ $\neg x_i \vee R_2$	$x_i \Leftrightarrow [R_1 \wedge R_2]$	$[\neg R_1 \vee \neg R_2]$
$\neg x_i \vee R_1 \vee R_2$ $x_i \vee \neg R_1$ $x_i \vee \neg R_2$	$x_i \Leftrightarrow [R_1 \vee R_2]$	$[\neg R_1 \wedge \neg R_2]$

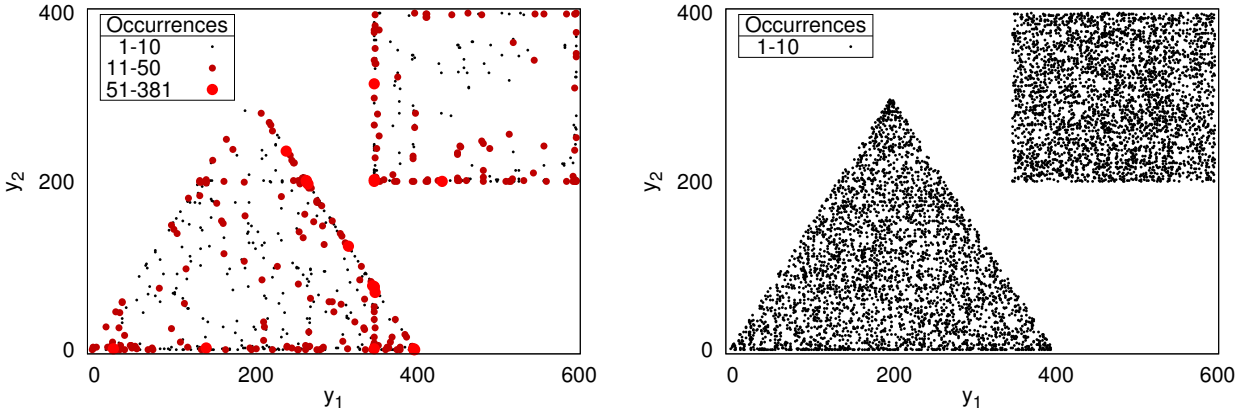
Table 4.1: Patterns of clauses that we identify for eliminating Boolean variables. The integer relations that replace x_i and $\neg x_i$ for the second and third patterns are single literals, not combinations of two literals.

expression. Clause 4.8 enables us to eliminate y_3. Clauses 4.13–4.15 and 4.16–4.18 match the second equivalence pattern in Table 4.1, so we can eliminate x_2 and x_3.

Eliminating k Boolean variables from a constraint formula decreases the size of the state space by a factor of 2^k without changing the number of solutions. Elimination of integer variables provides an even greater reduction. As a result, the generator spends less time visiting unsatisfying states. For the example in this section, after y_3, x_2 and x_3 are eliminated, the fraction of states that are solutions rises from 7.4×10^{-6} to 0.11. The increase in the density of the solution space lessens the need for local-search moves and reduces their distortion of the distribution. To demonstrate this benefit, we generated 10 000 solutions for Clauses 4.1–4.15, both with static variable elimination and without it. With elimination, the fraction of moves made with local search decreased from 41% to 18%. The corresponding improvement in the distribution is illustrated in Figure 4.1. The plots show the values of y_1 and y_2, as well as the number of occurrences of

Variable eliminated	Transformed clauses (replacements underlined)
y_3	$[\underline{3y_1 + 2y_2} \leq 1200] \vee x_1$ $\neg x_1 \vee x_2$ $\neg x_1 \vee x_3$ $x_1 \vee \neg x_2 \vee \neg x_3$
x_2	$[3y_1 + 2y_2 \leq 1200] \vee x_1$ $\neg x_1 \vee \underline{[350 \leq y_1 \leq 600]}$ $\neg x_1 \vee x_3$ $x_1 \vee \underline{[350 > y_1 \vee y_1 > 600]} \vee \neg x_3$
x_3	$[3y_1 + 2y_2 \leq 1200] \vee x_1$ $\neg x_1 \vee [350 \leq y_1 \leq 600]$ $\neg x_1 \vee \underline{[200 \leq y_2 \leq 400]}$ $x_1 \vee [350 > y_1 \vee y_1 > 600] \vee \underline{[200 > y_2 \vee y_2 > 400]}$

Table 4.2: Elimination of variables y_3, x_2, and x_3 from Clauses 4.9–4.12.



(a) Without variable elimination

(b) After eliminating y_3, x_2, and x_3

Figure 4.1: Distributions of (y_1, y_2) in 10 000 solutions to Clauses 4.1–4.15.

each distinct solution. Figure 4.1a shows that the bias of the local-search moves pushes most of the probability mass to the edges and corners of the solution space. In contrast, the solutions in Figure 4.1b are evenly distributed. The improvement in distribution came without a significant difference in runtime—the runtime for generation with elimination was about 3% greater than the runtime without elimination.

4.3 Dynamic Variable Elimination

Sometimes clauses do not fit the patterns we use to eliminate variables statically, but during generation, when we have specific values for the variables, the active clauses do fit the patterns. In such a case, we can increase the time spent visiting solutions by eliminating variables temporarily, for the duration of a single move. For example, consider the following clauses:

$$\neg x_1 \vee [y_1 = 2y_2] \tag{4.19}$$

$$x_1 \vee [y_2 \leq 3y_3] \tag{4.20}$$

We cannot eliminate y_1 statically. However, when y_2 is chosen as the variable for a Metropolis move and $x_1 = 1$, the equation $y_1 = 2y_2$ is active and y_1 can be eliminated during the move. The elimination leaves y_2 without any active constraints, so it can take any value without falsifying a clause; in contrast, the only satisfying value before the elimination was $\frac{y_1}{2}$. After a new value for y_2 is sampled, it can be substituted into the equation to get a new value for y_1. We refer to this technique of temporary elimination and resubstitution as *dynamic variable elimination*.

Figure 4.2 illustrates how dynamic variable elimination operates on the state space for constraints similar to Clauses 4.19 and 4.20. Figure 4.2a shows moves without dynamic elimination: It requires multiple moves to go between distinct solutions in the front plane (where $x_1 = 1$), because an equality constraint is active. Figure 4.2b shows a single move between the same solutions using dynamic elimination. Either y_1 or y_2 can be eliminated temporarily. Suppose that y_2 is selected as the variable for a move. The elimination of y_1 projects solutions in the front plane onto the subspace $\mathbf{Z}_B$; consequently, more than one of them can be reached with

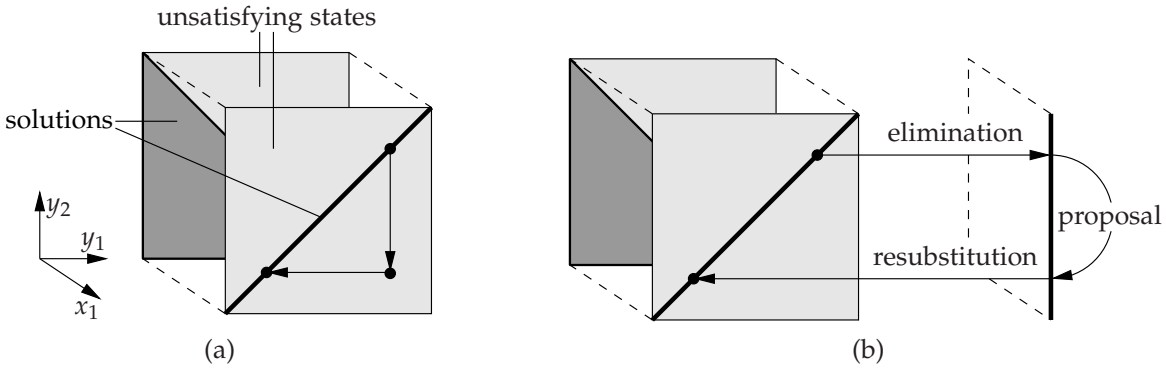


Figure 4.2: Effect of dynamic variable elimination on moves: (a) Without elimination, reaching a new solution requires multiple moves. (b) Elimination of y_1 projects solutions in the $x_1 = 1$ plane onto a single line, making multiple solutions reachable in a single move.

only one move. After a solution is proposed, resubstitution projects the new state back onto the diagonal plane that satisfies the active equation.

The patterns that clauses must match for elimination of Boolean variables are more complex than the clause pattern for eliminating integer variables (a single equation), and the number of states we avoid visiting is smaller, so the effort of matching patterns is less likely to result in a net benefit. Therefore, we restrict dynamic variable elimination to integer variables.

The mechanism by which we implement dynamic variable elimination (i.e., the projection of states onto a subspace) is simultaneous solution of the active equations and inequalities. With this approach we can easily handle a wide variety of equations, not only those of the form $y_i = f(y \setminus y_i)$ that we use in static elimination. To ensure that we can solve the equations efficiently, we perform elimination only when all the active relations are linear.

4.3.1 Basic Approach

For simplicity, we first describe the process of dynamic variable elimination for the restricted case where each equation is the only integer literal in its clause. Suppose that we have selected variable y_i as the primary variable for the move (i.e., in line 3 of Algorithm 3.5). If the values of $y \setminus y_i$ are held fixed, each equation referring to y_i has only one solution (at most). When the current state is a solution (which is ideally the common case for our algorithm), it is the unique solution for y_i, and the generator must move to an unsatisfying state or repeat the current state. In order to reach other solutions without recovery moves, we must allow a simultaneous change to at least one other variable in each active equation. In general, to ensure that this is possible, we must select at least as many additional variables for the move as there are active equations. We call them *dependent variables*. These are the variables that we effectively eliminate for the duration of the move.

Each dependent variable may be constrained by other equations that do not have y_i in their support. To have multiple solutions available as destinations for the move, we must consider each of these equations as active, too, and choose an additional dependent variable for each of them, and so on. Different choices of the initial dependent variables may lead to different sets of active equations to solve. In our approach we select between possible dependent

variables randomly; if the resulting system of equations does not have a solution we fall back to a single-variable move (i.e., a move with no variables eliminated).

Algorithm 4.1 outlines our procedure for selecting dependent variables and collecting active equations (still assuming that equations do not share clauses with other relations). In each iteration, we take an equation E_j from a priority queue, select an available dependent variable y_k in its support, and push onto the queue any equations referring to y_k that were not already taken as active. The queue is ordered by increasing size of variable support in order to reduce the likelihood that all the variables in an equation's support have already been selected by the time it is popped from the queue, in which case we would not have enough dependent variables.

Algorithm 4.1 SELECTION OF DEPENDENT VARIABLES AND ACTIVE EQUATIONS

Given: primary variable y_i
Local data: priority queue Q of equations, sorted by increasing $|\textsc{Support}(E_i)|$

```
1:  initialize Q with all active equations E_j for y_i
2:  Y := {}  [dependent variables]
3:  A := {}  [active equations]
4:  loop
5:      if |Q| = 0 then
6:          return (Y, A)
7:      pop E_j from Q
8:      V := SUPPORT(E_j) \ {y_i} \ Y  [candidate variables]
9:      if |V| > 0 then
10:         select new y_k from V uniformly at random
11:         Y := Y ∪ {y_k}
12:         A := A ∪ {E_j}
13:         for each E_l such that y_k ∈ SUPPORT(E_l) do
14:             if E_l ∉ A ∪ Q and E_l's clause is not satisfied by x then
15:                 push E_l onto Q
```

As an illustration of the selection of dependent variables, consider the following clauses:

$$[y_1 \leq 100] \qquad [y_3 \leq 500] \vee [y_3 \geq 600]$$
$$[y_1 + y_2 = 200] \qquad [y_3 - y_4 \geq 700]$$
$$[y_2 \leq 300] \qquad [y_4 + y_5 = 800]$$
$$[y_2 + y_3 - y_4 = 400]$$

Suppose that y_1 is selected as the primary (i.e., non-dependent) variable for a move. In the first iteration of the loop in Algorithm 4.1, the only equation in the queue is $y_1 + y_2 = 200$, and y_2 is selected as a dependent variable. The equation $y_2 + y_3 - y_4 = 400$ is pushed onto the queue and popped in the second iteration of the loop. At this point either y_3 or y_4 may be chosen as the next dependent variable. Suppose that y_3 is chosen. Then no more equations are pushed onto the queue, and the procedure terminates.

After selecting the dependent variables, we collect the active clauses; they include all clauses that refer to the primary variable or any dependent variable, except those currently satisfied by Boolean literals and relations that do not depend on the primary or dependent variables. For the previous example, all the clauses are active except $[y_4 + y_5 = 800]$.

We take each of the active inequalities in turn and solve it simultaneously with the active equations to get a satisfying interval for the primary variable y_i. We combine the resulting intervals to construct the proposal distribution for y_i just as described in Section 3.2.2 for ordinary single-variable moves. We sample a new value for y_i; if it is a satisfying value, we substitute it into the active equations and solve them to obtain new values for the dependent variables. We then proceed with the Metropolis acceptance step as usual.

The following example illustrates the solution of the active inequalities with the equations. A two-dimensional problem is established by the following clauses:

$$[\, y_1 - y_2 \geq -2 \,] \tag{4.21}$$
$$[\, y_1 + y_2 \leq 32 \,] \tag{4.22}$$
$$[\, y_1 + y_2 \geq 17 \,] \tag{4.23}$$
$$[\, y_1 - 4y_2 \leq 1 \,] \tag{4.24}$$
$$[\, 2y_1 - y_2 = 16 \,] \tag{4.25}$$

The constraints in the clauses and the solutions are shown in Figure 4.3a. Let $y = (13, 10)$ be the current state and assume that we select y_1 as the primary variable; y_2 is the only possible dependent variable. In an ordinary move without dynamic variable elimination, the sampling range for y_1 would include 13 with cost 0 and $8 \ldots 12, 14 \ldots 22$ with cost 1. The soft-SAT proposal distribution for such a move is shown in Figure 4.3b.

We combine each inequality with the equation in Clause 4.25 to create a linear system and solve it using Gaussian elimination to obtain a transformed bound for y_1. To determine the direction of a bound c (i.e., upper or lower), we substitute $c - 1$ for y_1 in the equation, solve for y_2, and check whether the inequality is satisfied. For example, solving the linear system

$$\begin{bmatrix} 1 & -1 & -2 \\ 2 & -1 & 16 \end{bmatrix}$$

gives us $y_1 = 18$. Substituting $y_1 = 17$ into the equation, we get $y_2 = 18$. Since $17 - 18 \geq -2$, we conclude that 18 is an upper bound on y_1.

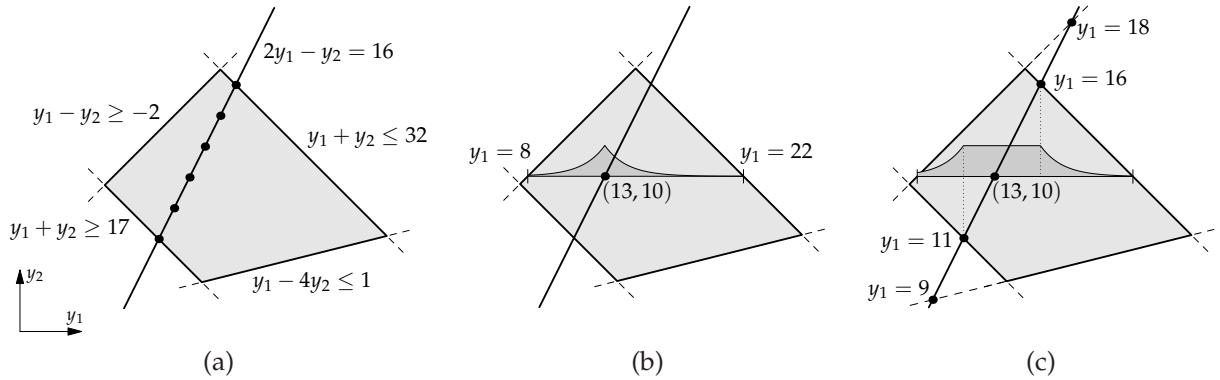


Figure 4.3: Illustration of dynamic variable elimination for Clauses 4.21–4.25: (a) constraints and solutions, (b) soft-SAT proposal distribution for y_1 without variable elimination, (c) proposal distribution with dynamic elimination of y_2.

Solving all the inequalities in Clauses 4.21–4.24 in this way gives the following transformed bounds for y_1:

$$[y_1 \leq 18] \ \wedge \ [y_1 \leq 16] \ \wedge \ [y_1 \geq 11] \ \wedge \ [y_1 \geq 9]$$

We combine the intervals satisfying these bounds to get a proposal distribution with maximum probability on all values in the range $11 \leq y_1 \leq 16$, as shown in Figure 4.3c. Suppose that we sample 15 as the new value y_1'. We substitute it into the equation and get $y_2' = 14$.

If y_2 had been selected as the primary variable for the move and an odd value had been sampled for it, no integer solution would exist for y_1. In this case we would take the nearest integer value. Since the resulting state does not satisfy the constraints, it may be rejected when the Metropolis acceptance rule is applied or be handled by recovery moves.

A drawback of dynamic variable elimination is that it may distort the distribution of solutions because it breaks the theoretical guarantee of the Metropolis-Hastings algorithm. Moves with elimination avoid proposing states that do not satisfy equations, so they do not have the detailed-balance property described in Section 2.2. For example, consider the following constraints:

$$0 \leq y_1, y_2 \leq 100$$
$$x_1 \vee [y_1 = y_2]$$

The state $s = (0, 10, 20)$ may be reached by flipping x_1. From this state, a move with dynamic elimination may arrive at $s' = (0, 30, 30)$. The reverse move from s' to s is never proposed, neither with elimination nor without it. Therefore, the Markov chain does not exhibit detailed balance:

$$p(s)M(s, s') > 0$$
$$p(s')M(s', s) = 0$$
$$p(s)M(s, s') \neq p(s')M(s', s)$$

4.3.2 Handling Disjunctions of Equations

When a clause contains an equation and another relation that shares variable support with it, the dependencies between variables may not be defined in a way that allows us to identify the active equations unambiguously. For example, suppose that one of the clauses is $[y_1 + y_2 = 200] \vee [y_1 - y_3 \geq 250]$ and the primary variable for a move is y_1. If $y_1 + y_2 = 200$ is active, then y_2 is a dependent variable, but if $y_1 - y_3 \geq 250$ is satisfied, then the equation need not be true, so no dependent variable is required. The dependency between y_1 and y_2 holds only for possible values of y_1 that do not satisfy $y_1 - y_3 \geq 250$. However, our approach for constructing the proposal distribution assumes that the active constraints are known before we try to select values for the variables.

We handle such cases by transforming the constraints to forms whose active relations can be determined unambiguously: We apply the distributive law to create multiple CNF subformulas, each satisfying the simplifying assumption used in the previous section that each equation is the only integer literal in its clause. Formally, let E be an equation and R be another relation (equation or inequality). If the constraint formula φ has the form $(E \vee R) \wedge \varphi'$, where φ' is any subformula, then we apply the transformation:

$$(E \vee R) \wedge \varphi' \qquad \Longrightarrow \qquad (E \wedge \varphi') \vee (R \wedge \varphi')$$

Boolean literals do not cause ambiguity in the selection of active constraints, so we do not create additional subformulas for them, only for the integer literals. For example:

$$(x_i \vee E \vee R) \wedge \varphi' \qquad \Longrightarrow \qquad \big((x_i \vee E) \wedge \varphi'\big) \vee \big((x_i \vee R) \wedge \varphi'\big)$$

We apply dynamic variable elimination to each subformula separately to obtain proposal distributions, using the procedure described in the previous section, then take the pointwise maximum of them to create a distribution for the entire formula. (That is, we apply the same disjunctive operator between subformulas as we do for literals in clauses as described in Section 3.2.2.) The new value y_i' that we sample from the combined distribution may be part of a solution for more than one subformula. From among all the subformulas whose distribution functions are maximized at y_i', we select one at random and solve its linear system to obtain values for the dependent variables.

The number of subformulas after distribution may be exponential in the number of clauses, so we impose a bound on the number of them that we solve. If the total number after distribution exceeds the bound, we select a random subset of them from which to construct the proposal distribution for the move.

As an illustration of how we handle disjunctions of equations, consider the following variation on Clauses 4.21–4.24:

$$[\,y_1 - y_3 \geq -2\,] \tag{4.26}$$
$$[\,y_1 + y_2 \leq 32\,] \tag{4.27}$$
$$[\,y_1 + y_2 \geq 17\,] \tag{4.28}$$
$$[\,y_1 - 4y_3 \leq 1\,] \tag{4.29}$$
$$[\,2y_1 - y_2 = 16\,] \vee [\,2y_1 + 3y_3 = 69\,] \tag{4.30}$$

Let $y = (13, 10, 6)$ be the current state and y_1 be the primary variable selected for the current move. Distributing over Clause 4.30 produces two subformulas, each containing one of the equations in that clause. The dependent variable for the first subformula is y_2. We substitute the current value of y_3 and solve the relations in Clauses 4.26–4.29 with $2y_1 - y_2 = 16$ to get the following bounds:

$$[y_1 \geq 4] \;\wedge\; [y_1 \geq 11] \;\wedge\; [y_1 \leq 16] \;\wedge\; [y_1 \leq 25]$$

For the second subformula, the dependent variable is y_3. Substituting the current value of y_2 and solving the inequalities with $2y_1 + 3y_3 = 60$ gives the following bounds:

$$[y_1 \geq 7] \;\wedge\; [y_1 \geq 12.6] \;\wedge\; [y_1 \leq 22] \;\wedge\; [y_1 \leq 25.\overline{36}]$$

Suppose that the selected type of proposal distribution is cost-based. The number of bounds violated in each subformula ranges between 0 and 2. We compute the distribution functions for both sets of bounds and take their pointwise maximum to get the composite distribution function:

$$q(y_1) = \begin{cases} e^{-2/T} & \text{for } y_1 \in [-2^{B-1} \,..\, 3] \\ e^{-1/T} & \text{for } y_1 \in [4 \,..\, 10] \\ 1 & \text{for } y_1 \in [11 \,..\, 22] \\ e^{-1/T} & \text{for } y_1 \in [23 \,..\, 25] \\ e^{-2/T} & \text{for } y_1 \in [26 \,..\, 2^{B-1} - 1] \end{cases}$$

Suppose that we sample $y'_1 = 20$ from the distribution function. This value has cost 1 for the first subformula and cost 0 for the second, so we use the second to obtain values for the dependent variable. Substituting into $2y'_1 + 3y'_3 = 69$ gives $y'_3 = 9.\overline{6}$; since this is not an integer, we take $y'_3 = 10$ instead. The new state is not a solution; recovery moves will follow.

4.3.3 Experimental Evaluation

To evaluate the effectiveness of dynamic variable elimination, we generated a set of 100 random benchmarks. Each benchmark has 40 integer variables and 10 clauses. No benchmark has Boolean variables. Each relation has the form:

$$\sum_{i \in I} a_i y_i \leq b \qquad \text{or} \qquad \sum_{i \in I} a_i y_i = b \tag{4.31}$$

where I is a set of indices. The clause sizes, index sets, coefficients, and constant terms were all chosen uniformly at random; Table 4.3 shows the ranges of these values. We accepted only satisfiable sets of constraints. When we could not generate the full number of desired solutions to a benchmark, we discarded it and generated a new one to replace it.

Value	Symbol in Formula 4.31	Range
Relations per clause		$[1\,..\,2]$
Linear terms per inequality	$\lvert I \rvert$	$[1\,..\,4]$
Linear terms per equation	$\lvert I \rvert$	$[2\,..\,4]$
Coefficient of variable	a_i	$[-4\,..\,-1, 1\,..\,4]$
Constant term	b	$[-1000\,..\,1000]$

Table 4.3: Ranges of randomly generated values in benchmarks for dynamic variable elimination.

We generated 10 000 solutions to each benchmark using our Ambigen program with dynamic variable elimination, and the same number of solutions without elimination. We expected the moves without elimination to be faster because of their simplicity, but they are also more likely to repeat solutions. To test whether the moves with elimination might still be more efficient at generating useful solutions, we rejected repeated solutions. That is, we counted each solution only when it differed from the preceding one.

The results of this experiment are shown in Figures 4.4 and 4.5. The plot in Figure 4.4 shows for each benchmark the ratio T_e/T_0, where T_e is the runtime with elimination and T_0 is the runtime without it; this ratio is the slowdown caused by dynamic variable elimination. The benchmarks are sorted in order of increasing slowdown. As the plot shows, dynamic elimination does not enable our generator to find distinct solutions faster; on the contrary, the runtime is 7–53x slower, with an average slowdown of about 23x (the arithmetic mean of the ratios is 22.8 and the ratio of the total runtimes—i.e., $\sum T_e / \sum T_0$—is 22.6).

Figure 4.5a shows the number of moves made in each mode; for 58 of the 100 benchmarks more moves were required when dynamic variable elimination was used. At most elimination reduced the number of moves by 42%. The reduction was not enough to compensate for the extra time taken by elimination. Figure 4.5b shows the relationship between the move

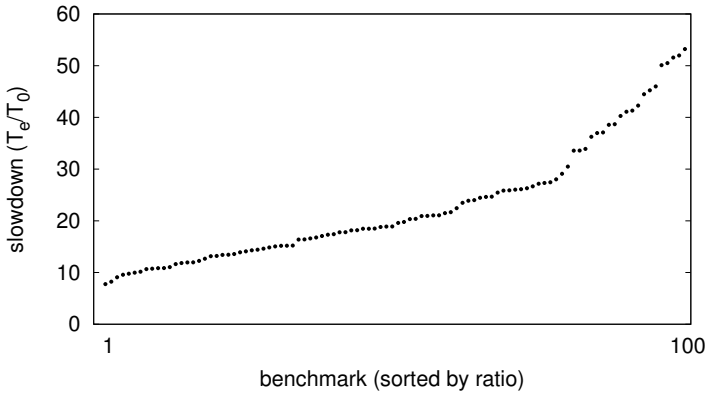


Figure 4.4: Ratios of runtimes with dynamic variable elimination (T_e) to runtimes without it (T_0).

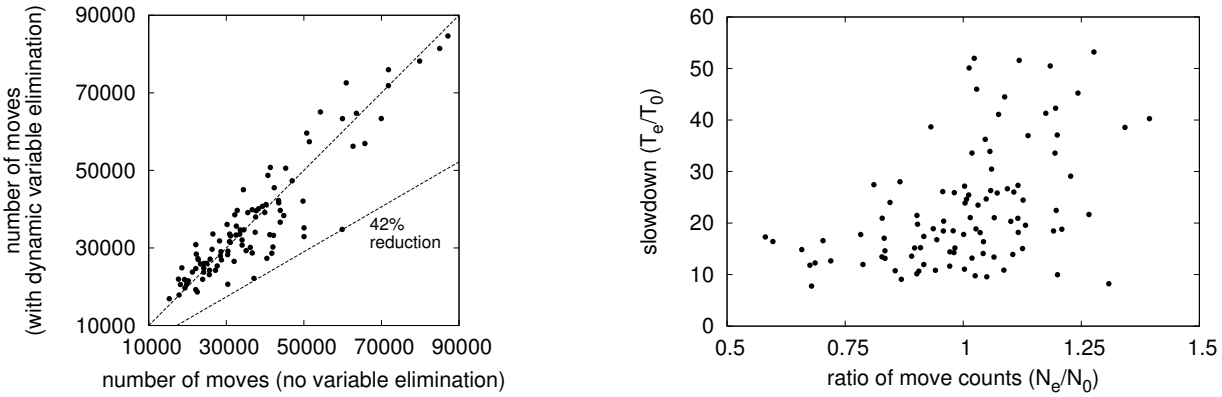


Figure 4.5: (a) move counts with and without dynamic variable elimination, (b) ratios of move counts (N_e = with elimination, N_0 = without it) and runtimes.

counts and the slowdowns: the runtime ratios are plotted against the ratios N_e/N_0, where N_e is the number of moves made with dynamic elimination and N_0 is the number of moves without it. The slowdowns were least when elimination reduced the number of moves most, but otherwise the relationship is weak.

These results show that for constraints like our generated benchmarks dynamic variable elimination, unlike static variable elimination, does not provide enough benefit to offset its cost. Our benchmarks may not be representative of all constraints where it could be applied, but the magnitude of the slowdowns we observed suggest that it is too slow in general.

4.4 Summary

In this chapter we presented several refinements to our basic generation algorithm. The first refinement is slight: We can handle constraints that depend on the current state of the DUT or testbench by including control variables in the constraints. The procedures for generating

moves support this with almost no change. The second refinement is static variable elimination, a preprocessing step that increases the density of the solution space and improves both efficiency and distribution. The third refinement, dynamic variable elimination, applies a similar idea during generation, but it has high overhead which outweighs its benefit.

Chapter 5

Parallel Stimulus Generation

Parallelism is becoming increasingly important as a means to make programs faster. During the last decade, manufacturers of high-performance microprocessors have shifted their efforts away from making individual processors faster; instead, they now make processors with increasing numbers of cores. As this trend continues, applications that cannot exploit parallelism for speedup will waste increasingly larger fractions of available computational power.

In this chapter we describe how our generation method can be parallelized for faster production of stimuli. Our parallelization approach does not replace the sequential algorithm given in the previous chapters but uses multiple sequential generators in combination. In the absence of control variables, we could use a division-of-labor scheme in which each generator would produce a subset of the stimuli needed. In the best case, such a scheme would yield linear speedup (neglecting synchronization overhead): With P generators running in parallel, the total runtime would be reduced by a factor of P. However, this approach does not work in the general case; when the constraints depend on the current state of the DUT or testbench, stimuli must be generated on demand, in sequence. Therefore, we use a "race" parallelization scheme instead, in which the generators search using the same control values and only the first solution found is used as a stimulus. This scheme seems more wasteful than division of labor because the generators that lose the race make many moves whose results are not directly used. However, it has the potential to yield greater speedup, as we show in this chapter.

The organization of this chapter is as follows: Before we describe our parallel algorithm in detail, we show how variability in generation time creates potential for speedup. To illustrate this point, we introduce a set of benchmarks with constraints more difficult than those used in previous chapters; we refer back to them throughout the chapter. We project speedup bounds for the benchmarks based on data from sequential generation, and we show through theoretical analysis that there are conditions under which speedup can be arbitrarily high. We demonstrate with results of experiments using our benchmarks that this theoretical potential for speedup is difficult to realize in practice. The speedup we observed in our experiments falls short of our projections, and we examine our results in detail to determine why. Among the reasons for the discrepancies are insufficient parallelism in hardware, correlation in the generators' performance, and delayed application of control values to the generators. Our investigation demonstrates that achieving high speedup from parallelization, even with a straightforward algorithm, is challenging.

5.1 Benchmarks

The potential for speedup in our parallel generation approach comes from variability in the time needed for a single generator to produce a solution. When one generator takes a long time to solve the constraints, another generator, starting from a different state and sampling different random values, may reach a solution in much less time.

For many constraint sets, the average number of moves per solution made by our algorithm is close to 1. Because the variability in solution time for such constraints is low, they have little potential for speedup from parallel generation. Most of the benchmarks in previous chapters are in this category, so they are not interesting test cases for parallelization. To explore parallel generation where it may actually provide a significant benefit, we generated benchmarks with constraints that are more difficult to solve. The generated constraints are a generalization of random Boolean 3-CNF using integer variables, including control variables. A Boolean 3-CNF clause, e.g., $x_1 \vee x_2 \vee \neg x_3$, can be expressed with integer constraints in our normal form in two ways—as a clause with three literals:

$$[y_1 \geq 1] \vee [y_2 \geq 1] \vee [y_3 < 1]$$

or as an integer literal with three terms:

$$[y_1 + y_2 + (1 - y_3) \geq 1]$$
$$0 \leq y_1, y_2, y_3 \leq 1$$

In our benchmarks we included clauses generalized from both of these forms, for example:

$$[y_1 \geq v_1] \vee [y_2 \geq v_2] \vee [y_3 < v_3]$$
$$[y_1 + y_2 + (1000 - y_3) \geq 1000]$$
$$0 \leq y_i \leq 1000$$

We departed from the 3-CNF pattern slightly by using equality literals such as $[y_1 = y_2]$ in place of inequalities with probability 0.1. The full grammar for the clauses in the benchmarks is the following:

$$\begin{aligned} \mathit{clause} &: \mathit{literal} \vee \mathit{literal} \vee \mathit{literal} \mid [\, \mathit{term} + \mathit{term} + \mathit{term} \geq 1000 \,] \\ \mathit{literal} &: [\, y_i \geq v_i \,] \mid [\, y_i < v_i \,] \mid [\, y_i = y_j \,] \mid [\, y_i = 1000 - y_j \,] \\ \mathit{term} &: y_i \mid 1000 - y_i \end{aligned}$$

We generated 100 random benchmarks, each with 20 random integer variables y_i, 20 integer control variables v_i, and 100 random clauses (not including the clauses for $0 \leq y_i \leq 1000$). To check the potential for speedup from parallelism, we generated 10 000 solutions to each benchmark, using our sequential algorithm described in the previous chapters, with control values assigned randomly from the range $[0\,..\,1000]$, and recorded the elapsed (wall-clock) time used to generate each solution. Figure 5.1 shows, for each benchmark, a box-and-whisker plot of the summary statistics of the solution times, including the minimum, the maximum, and the quartiles. The benchmarks are sorted by median solution time. The wide variability in the times suggests that these benchmarks have good potential for speedup from parallelization.

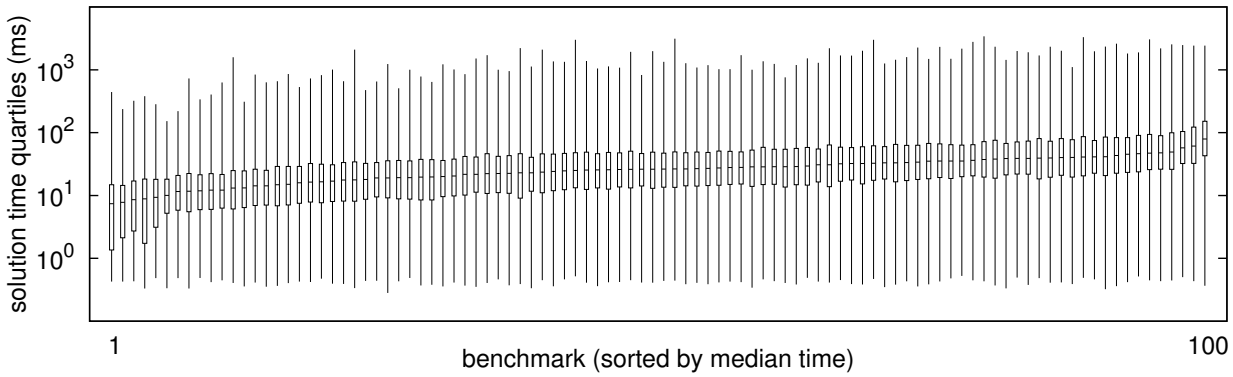


Figure 5.1: Quartiles of time per solution for our generalized 3-CNF benchmarks, sorted by median time. For each benchmark, the box is between the first and third quartiles; the line across the box is at the median. The whiskers extend from the first and third quartiles to the maximum and minimum times.

5.2 Bounds on Expected Speedup

The expected speedup from parallelization on a given set of constraints is intractable to compute because the solution time may depend on the control values. However, we can compute an upper bound on the expected speedup, based on the empirical distribution of solution time for a single generator. If we assume that the solution times of multiple generators are independent of each other and of the control values and identically distributed (i.i.d.) with cumulative distribution $F^{(1)}(t)$, that is, $F^{(1)}(t) = \Pr(\text{solution time} \leq t)$, then the solution time of P generators, starting simultaneously and working in parallel, has distribution:

$$F^{(P)}(t) = 1 - (1 - F^{(1)}(t))^P \tag{5.1}$$

This formula holds for both continuous and discrete distributions. We apply this formula to the discrete empirical distribution $\hat{F}^{(1)}(t)$ to get a projected distribution $\tilde{F}^{(P)}(t)$, from which we compute the projected density $\tilde{f}^{(P)}(t)$, mean $\tilde{\mu}^{(P)}$, and speedup $\tilde{S}^{(P)}$:

$$\begin{aligned}
\tilde{f}^{(P)}(t) &= \tilde{F}^{(P)}(t) - \tilde{F}^{(P)}(t-1) \\
\tilde{\mu}^{(P)} &= \sum_{t=0}^{\infty} t\tilde{f}^{(P)}(t) \\
\tilde{S}^{(P)} &= \frac{\tilde{\mu}^{(1)}}{\tilde{\mu}^{(P)}}
\end{aligned}$$

The projected speedup is an upper bound on expected speedup, not an estimate of it, for two reasons. First, it does not take into account the overhead of synchronization. Second, the assumption of i.i.d. solution times often does not hold in practice. The constraints may be harder to solve for some control values than for others, and all generators with the same control values will be subject to the increased difficulty, so that the times are positively correlated. Positive correlation lowers the speedup of parallelization relative to the i.i.d. case, as Figure 5.2 illustrates. The figure shows simple discrete joint distributions $f(t_1, t_2)$ of solution times t_1 and t_2 for two

$f(t_1, t_2)$

	$t_2 = 1$	$t_2 = 2$
$t_1 = 1$	0.36	0.24
$t_1 = 2$	0.24	0.16

$\text{corr}(t_1, t_2) = 0$

$\text{E}\left[S^{(2)}\right] = 1.207$

$f(t_1, t_2)$

	$t_2 = 1$	$t_2 = 2$
$t_1 = 1$	0.45	0.15
$t_1 = 2$	0.15	0.25

$\text{corr}(t_1, t_2) = 0.375$

$\text{E}\left[S^{(2)}\right] = 1.120$

$f(t_1, t_2)$

	$t_2 = 1$	$t_2 = 2$
$t_1 = 1$	0.25	0.35
$t_1 = 2$	0.35	0.05

$\text{corr}(t_1, t_2) = -0.4583$

$\text{E}\left[S^{(2)}\right] = 1.333$

Figure 5.2: Joint distributions of solution times for two generators in parallel and resulting correlations and expected speedup. For each distribution, $\Pr(t_i = 1) = 0.6$ and $\Pr(t_i = 2) = 0.4$.

generators in parallel, along with the correlation of the times and the expected speedup. All three cases have the same distribution for a single generator's solution time.

In addition to the effect of positive correlation, the figure also shows that a distribution with negative correlation has greater speedup than the uncorrelated distribution. As the example implies, the i.i.d. speedup is not an upper bound on the speedup for an individual run; the empirical solution times may happen to be negatively correlated. However, the *expected* correlation is positive. All the generators receive the same external input (the control values); there is no mechanism for this information to consistently influence the solution times of different generators in opposite directions.

We computed the projected speedup $\tilde{S}^{(P)}$ for each of our 100 benchmarks with $P = 2$, 3, 4, ..., 16 from the empirical solution-time distributions summarized in Figure 5.1. The resulting distributions of $\tilde{S}^{(P)}$ across the benchmarks are shown in Figure 5.3 as quartiles. For $P < 4$, more

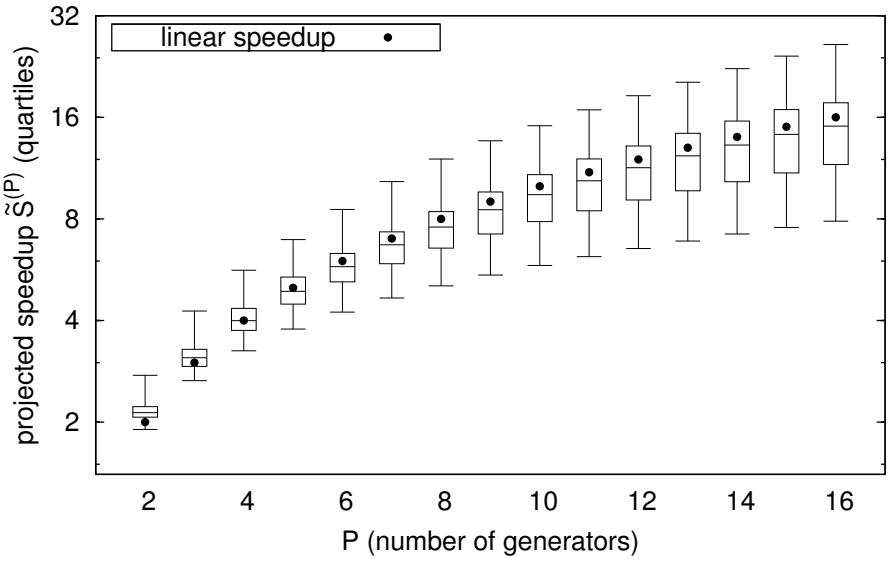


Figure 5.3: Distributions of projected speedup from parallel generation on generalized 3-CNF benchmarks.

than 50% of the benchmarks had $\tilde{S}^{(P)} \geq P$, i.e., superlinear speedup. For larger P, the median speedup was sublinear, but in each case a significant fraction of benchmarks still had superlinear projected speedup.

It may seem counterintuitive that parallelization can give a speedup greater than the number of generators working in parallel. However, we can show it analytically for a certain type of distribution of solution times: a mixture of exponential distributions. We do not claim that this distribution is a good model for the solution times of our generation algorithm, but the derivation shows that solution-time distributions with superlinear speedup exist and thus supports the plausibility of our projections.

Suppose that the solution time of a single generator is distributed as a mixture of two exponential variates: $F^{(1)}(t) = \theta_1(1 - e^{-\lambda_1 t}) + \theta_2(1 - e^{-\lambda_2 t})$, with $\lambda_1 \neq \lambda_2$, $0 < \theta_1, \theta_2 < 1$, and $\theta_1 + \theta_2 = 1$. That is, with probability θ_1 the time is exponentially distributed with mean λ_1^{-1}, and likewise for θ_2 and λ_2^{-1}. The mean μ_1 of the mixture is $\theta_1\lambda_1^{-1} + \theta_2\lambda_2^{-1}$. We apply Equation 5.1 to obtain the distribution for two generators in parallel:

$$\begin{aligned}
F^{(2)}(t) &= 1 - \Big(1 - \big(\theta_1(1 - e^{-\lambda_1 t}) + \theta_2(1 - e^{-\lambda_2 t})\big)\Big)^2 \\
&= 1 - \big(\theta_1 e^{-\lambda_1 t} + \theta_2 e^{-\lambda_2 t}\big)^2 \\
&= 1 - \theta_1^2 e^{-2\lambda_1 t} - 2\theta_1\theta_2 e^{-(\lambda_1+\lambda_2)t} - \theta_2^2 e^{-2\lambda_2 t} \\
&= (\theta_1 + \theta_2)^2 - \theta_1^2 e^{-2\lambda_1 t} - 2\theta_1\theta_2 e^{-(\lambda_1+\lambda_2)t} - \theta_2^2 e^{-2\lambda_2 t} \\
&= \theta_1^2(1 - e^{-2\lambda_1 t}) + 2\theta_1\theta_2(1 - e^{-(\lambda_1+\lambda_2)t}) + \theta_2^2(1 - e^{-2\lambda_2 t})
\end{aligned}$$

The result is a mixture of three exponential distributions with mean:

$$\mu^{(2)} = \theta_1^2(2\lambda_1)^{-1} + 2\theta_1\theta_2(\lambda_1 + \lambda_2)^{-1} + \theta_2^2(2\lambda_2)^{-1}$$

Next we use Jensen's inequality, which states that for a strictly convex function $g(x)$ and $x_1 \neq x_2$, $g\big(\frac{1}{2}(x_1 + x_2)\big) < \frac{1}{2}\big(g(x_1) + g(x_2)\big)$. We apply it to the second term of $\mu^{(2)}$ with $g(x) = x^{-1}$, which is strictly convex for $x > 0$, to get:

$$\begin{aligned}
\mu^{(2)} &< \theta_1^2(2\lambda_1)^{-1} + \frac{1}{2}\theta_1\theta_2(\lambda_1^{-1} + \lambda_2^{-1}) + \theta_2^2(2\lambda_2)^{-1} \\
&= (\theta_1 + \theta_2)\theta_1(2\lambda_1)^{-1} + (\theta_1 + \theta_2)\theta_2(2\lambda_2)^{-1} \\
&= \frac{1}{2}\big(\theta_1\lambda_1^{-1} + \theta_2\lambda_2^{-1}\big) \\
&= \frac{\mu^{(1)}}{2}
\end{aligned}$$

Thus the expected speedup $\mathrm{E}[S^{(2)}]$ is strictly greater than 2.

There is no upper bound on $\mathrm{E}[S^{(2)}]$ for a mixture of exponential distributions. Assume, without loss of generality, that $\lambda_1 > \lambda_2$. For fixed θ_1, θ_2, and λ_2, the derivative $d\mu^{(2)}/d\lambda_1$ is always negative, so the bounds of $\mu^{(2)}$ can only be found at the extremes of the range of λ_1. One extreme gives the speedup bound of 2 that we derived above: $\lim_{\lambda_1 \to \lambda_2} \mu^{(2)} = \frac{1}{2}\mu^{(1)}$. We find the other bound as we let λ_1 grow indefinitely:

$$\lim_{\lambda_1 \to \infty} \mu^{(2)} = \theta_2^2(2\lambda_2)^{-1}$$

$$\lim_{\lambda_1 \to \infty} \mathrm{E}\left[S^{(2)}\right] = \frac{\theta_2 \lambda_2^{-1}}{\theta_2^2 (2\lambda_2)^{-1}} = \frac{2}{\theta_2}$$

By choosing a sufficiently small θ_2 and large λ_1, we can make the speedup arbitrarily large.

Similar analytical bounds apply for $P > 2$. Although we do not show the derivations, the arguments given in this section can be generalized to prove that $\mathrm{E}[S^{(P)}] > P$ and $\lim_{\lambda_1 \to \infty} \mathrm{E}[S^{(P)}] = P\theta_2^{-(P-1)}$ for exponential mixtures. These results show that our speedup values projected from empirical data are plausible; in fact, they are modest compared to the values that theory allows.

5.3 Parallel Generation Algorithm

In the introduction of this chapter, we outlined briefly our general approach for parallel stimulus generation: We run multiple sequential generators simultaneously; in each iteration we take the solution from the first generator that finds one. In this section we describe in detail how we implement this approach as multi-threaded code.

Each sequential generator runs in a separate *worker* thread. An additional component, the *master*, runs in the same thread as the rest of the testbench and provides the interface between it and the workers. The master receives the control values from the testbench and supplies them to the workers, and it selects between available solutions.

The master and the workers store their own copies of the control values and generated values. The master's values are denoted $(u, v)^0$ and $(x, y)^0$, and the values in worker i are denoted $(u, v)^i$ and $(x, y)^i$. As shorthand for $((u, v)^i, (x, y)^i)$ we use $(u, v, x, y)^i$. Each worker also maintains output buffers $(\vec{u}, \vec{v})^i$ and $(\vec{x}, \vec{y})^i$. These buffers provide stable values for the master to read while the worker continues generation. To avoid data races when variables are accessed by multiple threads, we use mutex locks such as those provided by the Pthreads library [NBF96]. The output buffers in worker i are protected by lock L_i, and the lock L_0 protects access to $(u, v)^0$.

Besides the state of the generator and the output buffers, each worker stores an additional piece of state: a blocking count β_i. This count is used to prevent a single worker's solutions from being selected too often. When the master takes a solution from worker i, it resets the blocking count to the value of the blocking parameter β_0. Adjustment of β_0 enables a tradeoff between throughput and correlation of solutions.

Algorithm 5.1 is the top-level routine executed in each worker thread. The worker does not necessarily load new control values from the master as soon as they become available but rather loads them with probability p_{ld}. Consequently, when a generation cycle begins with control values that were used for a previous solution, there is some chance that one of the workers still has these values, along with a solution for them. This caching behavior can make a solution available more quickly. On the other hand, if cache "hits" are infrequent, a low value of p_{ld} will increase the average time to find a solution because the number of workers with the current control values will increase only gradually.

After (possibly) loading the control values $(u, v)^0$, the worker makes a single Metropolis or local-search move, altering its private state $(x, y)^i$. Because this state is separate from the output buffers and not accessed directly by the master thread, no locks are held during this period where the most intensive computation is performed. The MOVE routine, shown in Algorithm 5.2,

Algorithm 5.1 WORKER THREAD

Given: worker index i; load probability p_{ld}; formula $\varphi(u, v, x, y)$

initialize $(u, v, x, y)^i$ randomly
loop
  with probability p_{ld} **do**
    locking L_0 **do**
      $(u, v)^i := (u, v)^0$
      $t_i := 0$
  $(x, y)^i := \text{MOVE}((u, v, x, y)^i, t_i)$
  $t_i := t_i + 1$
  locking L_i **do**
    if $\beta_i > 0$ **then**
      $\beta_i := \beta_i - 1$
  if $\varphi((u, v, x, y)^i)$ **then**
    $t_i := 0$
    locking L_i **do**
      $(\vec{u}, \vec{v}, \vec{x}, \vec{y})^i := (u, v, x, y)^i$

Algorithm 5.2 MOVE

Given: formula $\varphi(u, v, x, y)$; state (u, v, x, y); move count t; move-type parameter p_{MH0}; rate parameter γ

if $t = 0$ or $\varphi(u, v, x, y)$ **then**
  $(x, y) := \text{METROPOLISMOVE}(u, v, x, y)$
else
  $p_{\text{MH}} := p_{\text{MH0}} e^{-\gamma(t-1)}$
  with probability p_{MH} **do**
    $(x, y) := \text{METROPOLISMOVE}(u, v, x, y)$
  else
    $(x, y) := \text{LOCALSEARCHMOVE}(u, v, x, y)$
return (x, y)

is a refactored version of the GENERATEONE routine given in Algorithm 3.4. The reason we make only one move, rather than continuing until a solution is found, as in GENERATEONE, is to allow earlier preemption of the worker: It may begin working with new control values before it finds a solution with the old ones, a solution which would likely be discarded anyway.

After making a move, the worker increments its move count and decrements its blocking count. The decrement must be synchronized using the lock L_i, because the master also reads and writes β_i. If the state resulting from the move is a solution, the worker copies it to the output buffer, along with its control values, and resets the move count (thus resetting the bias toward local-search moves described on page 28).

Algorithm 5.3 is the routine invoked by the testbench in the master thread. It copies the control values from the testbench into its buffer, then checks the output buffers of the workers repeatedly until it finds a solution for those control values. We use two techniques to avoid taking solutions from any worker too often: We check the workers in random order, and we avoid recently chosen workers using the blocking count β_i as described earlier in this section.

Algorithm 5.3 GENERATE (in master thread)

Given: control assignment (u, v); blocking parameter β_0

```
 1: locking L_0 do
 2:     (u, v)^0 := (u, v)
 3: loop
 4:     I := (1, ..., P)
 5:     randomize order of I
 6:     for each i ∈ I do
 7:         locking L_i do
 8:             if β_i = 0 and (u⃗, v⃗)^i = (u, v)^0 then
 9:                 (x, y)^0 := (x⃗, y⃗)^i
10:                 β_i := β_0
11:                 goto line 12 [unlocking L_i]
12: return (x, y)^0
```

The master is likely to repeat the main loop in GENERATE many times before a solution becomes available. Other mechanisms for synchronization, such as condition variables, would allow the master thread to be suspended rather than consuming CPU time in busy waiting. However, the time needed to find a solution is often much less than the length of a time slice in the scheduler (e.g., the default time slice in Linux 2.4 is 10 ms). A context switch would add significant delay to the master. For this reason we keep the master polling for solutions instead of suspending it.

Figure 5.4 shows the flow of data between the master and workers. The diagram illustrates key features of our parallel algorithm: the master as the sole interface between the DUT and the sequential generators, and the locks associated with all the variables accessed by more than one thread.

Programs with a high degree of parallelism are typically written with a different approach from ours, e.g., using a message-passing model such as MPI [Mes94] instead of the shared-memory model that we use. Our approach could be implemented using message passing; the communication between the master and workers would be significantly slower because sending messages over the network takes much longer than synchronization with locks. As the number of cores per processor increases, greater parallelism will be possible without incurring the overhead of message passing.

5.4 Experimental Evaluation

To evaluate our approach for parallelizing our stimulus-generation algorithm, we generated solutions to the benchmarks described in Section 5.1. In this section we present the results of our experiments.

A significant challenge in our evaluation of parallel stimulus generation was the availability of parallel hardware. Although we expect processors with more cores to become more common in the future, at the time we performed this research we had ready access only to computers with four cores or fewer. Because our master thread consumes CPU cycles while waiting, we could run at most three workers with true simultaneity. Later in this section we describe how we worked around this limitation.

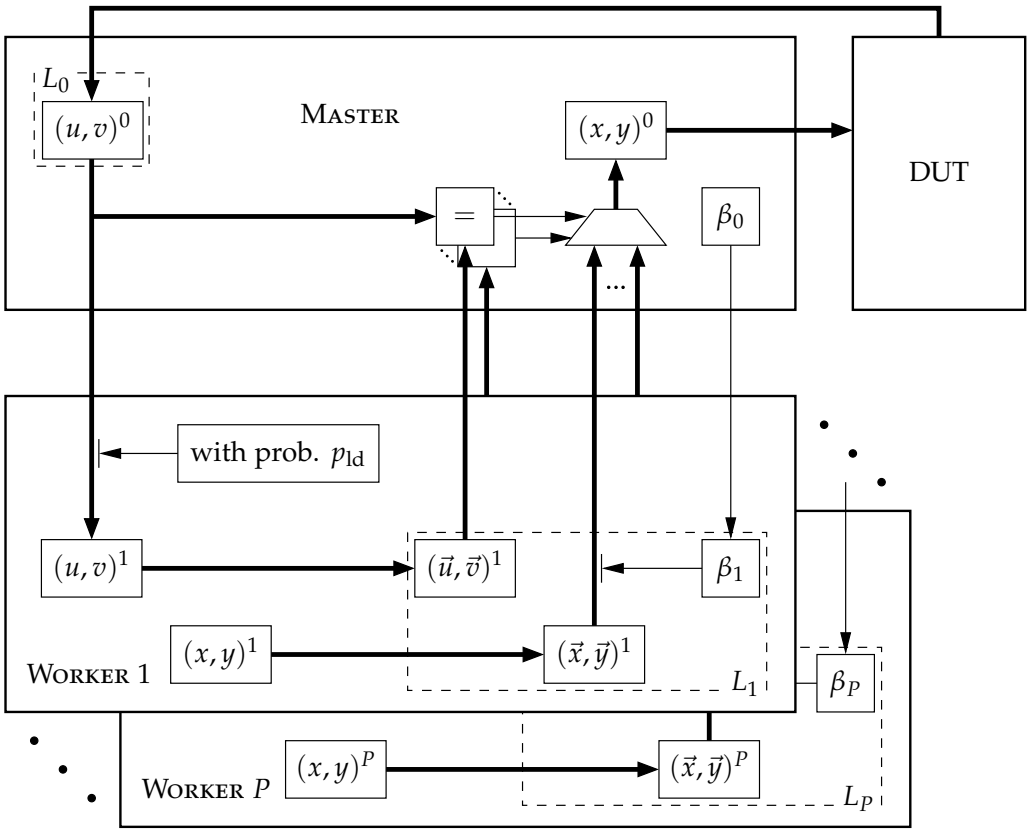


Figure 5.4: Data flow in parallel stimulus generation.

5.4.1 Generation using True Parallelism

We generated 10 000 solutions to each of our benchmarks with $P = 1, 2, 3$ (i.e., using 1, 2, and 3 workers). For the load probability parameter we used $p_{\text{ld}} = 1$, because the number of possible control assignments for the benchmarks is so large that we expected the hit rate on cached control values to be negligibly small. For the blocking parameter we used $\beta_0 = 1$.

The primary metric of performance is elapsed time; because it tends to be unstable under scheduling competition with other, unrelated processes in the system, we recorded it as the median of the elapsed times from five runs with the same random seed. We computed the resulting speedup ratio $S^{(P)}$ (the ratio of the median time with P workers to the time with 1 worker) for each benchmark. The distributions of $S^{(P)}$ are shown in Figure 5.5 as quartiles. In every case, generation with multiple workers was faster than generation with a single worker. However, the best speedup is still sublinear. On average, $S^{(3)}$ is only slightly greater than $S^{(2)}$. The range of $S^{(3)}$ is greater than the range of $S^{(2)}$, but the distribution of $S^{(3)}$ is more tightly concentrated around the median. The tighter concentration means that the speedup becomes more predictable, on average, as the number of workers increases.

Besides elapsed time, another metric of generation performance is the number of moves used. Comparing the two metrics from our experiments helps explain the results in Figure 5.5. For each benchmark, we took the move count from the run with the median elapsed time. Each move count includes only the moves used to generate the selected solutions. That is, each worker

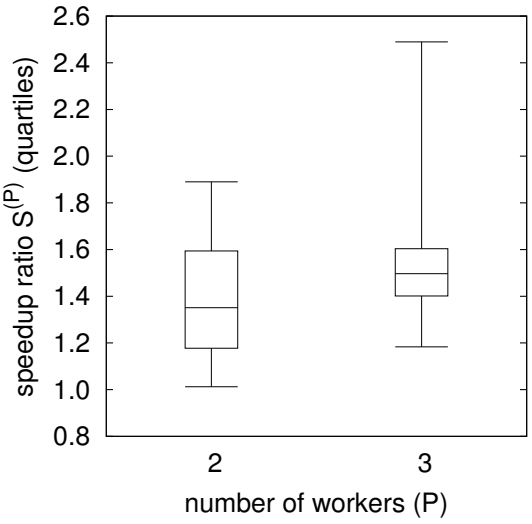


Figure 5.5: Distributions of speedup from parallel generation.

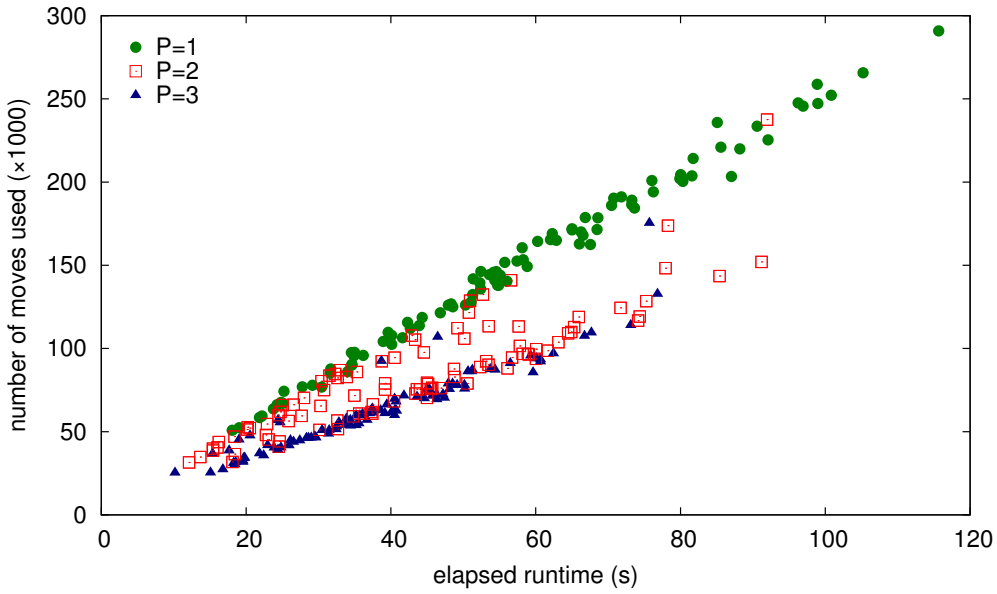
counted the moves it made since it had last loaded the control values, and each time the master selected a worker and took the solution from it, we recorded that worker's move count. We did not record moves made by other workers to produce solutions that were not used.

The times and move counts are compared in Figure 5.6a. With $P = 1$, the time and move count are well correlated. The same is true with $P = 3$, but with fewer moves per unit time than with $P = 1$. The points for $P = 2$ follow neither of these trends consistently but are distributed between them instead. The distributions of points correspond to the different degrees of dispersion in Figure 5.5: As the points for $P = 3$ follow a more consistent trend than those for $P = 2$, the speedup for $P = 3$ is less dispersed.

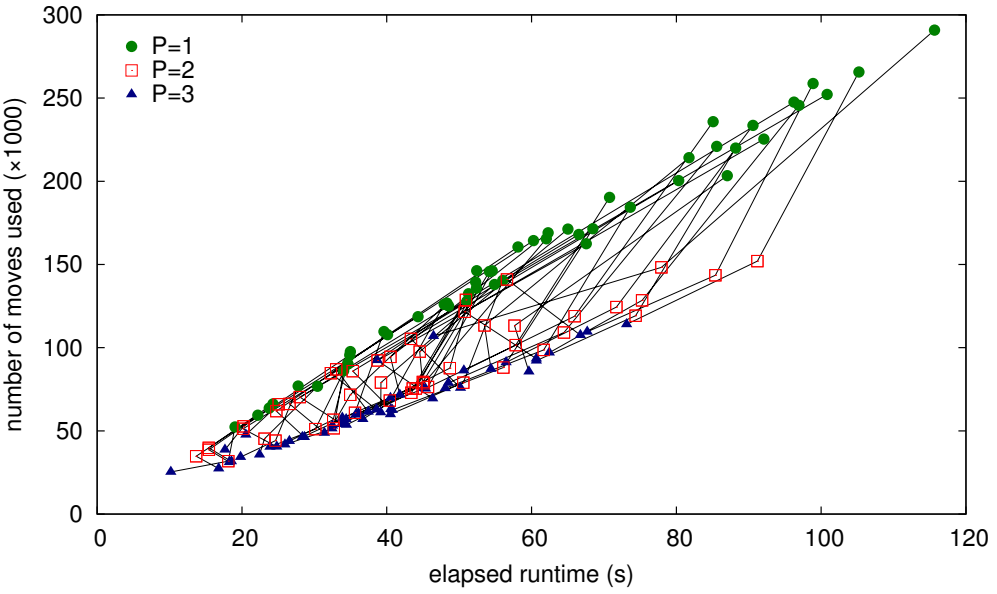
Overall, the number of moves used per unit time decreases with the number of workers. The plot in Figure 5.6b reveals the patterns in these results in greater detail. It shows the same data with the addition of lines connecting the three data points for each benchmark. The lines make clear that generation with 2 workers consistently takes less time and fewer moves than generation with 1 worker, and the number of moves always decreases as P increases. However, for about 40% of the benchmarks, generation with 3 workers is slower than with 2, despite the lower move counts.

We cannot explain the decreasing move rate of parallel generation with certainty, but we have a hypothesis as to its cause: We believe it is an artifact of scheduling conflicts. Although we ran our experiments on a processor with sufficient cores for the master and worker threads, our program was not the only process running on the system. At least one other process was occasionally scheduled: the daemon that periodically flushes data from the disk cache. The total CPU load from this process is negligible, but our algorithm is highly sensitive to any interruption in a thread. We conjecture that context switches delay the master's recognition of available solutions, with greater delay when our threads are using all the cores.

The total move count can serve as a proxy metric for the elapsed runtime, and it provides a significant advantage over the latter: It can be decoupled from scheduling. That is, the workers' moves can be scheduled in such a way that the total number of moves is unaffected by context switches. Consequently, we can run many more workers than there are cores available,



(a)



(b)

Figure 5.6: Results of parallel generation of 10 000 solutions for the benchmarks described in Section 5.1: (a) elapsed runtime and number of moves, (b) a subset of the same data, with links between the three data points for each benchmark.

circumventing the problem of insufficiently parallel hardware. To justify this, we first establish that the move count is a sufficiently faithful proxy for elapsed time, or rather, that the speedup computed as a ratio of move counts accurately represents the speedup as a ratio of elapsed times in the absence of scheduling conflicts.

We denote the speedup ratio derived from elapsed times $S_t^{(P)}$ and the speedup derived

from move counts $S_m^{(P)}$. Given that our algorithm has a consistently lower move rate with 3 workers than with 1 worker, $S_t^{(3)}$ and $S_m^{(3)}$ can not be expected to match. However, for $P = 2$ a significant fraction of the data points are close to the trend for $P = 1$. For these, $S_m^{(2)}$ should be a good predictor of $S_t^{(2)}$. To check this relationship, we computed least-squares linear fits to the data points for $P = 1$ and $P = 3$ and classified the data points for $P = 2$ into three groups depending on their distances to the two fit lines. Figure 5.7 shows the data with the fit lines and different markers for the three groups of points. Figure 5.8 shows the speedup ratios, both time- and move-based, with the same markers for the groups. The plot illustrates that the two kinds of speedup ratios do indeed match closely for the benchmarks whose times follow the trend line for $P = 1$; i.e., $S_m^{(P)}$ is a good proxy for $S_t^{(P)}$ in the absence of artifacts due to scheduling.

5.4.2 Generation using Simulated Parallelism

Having justified the use of the move count as a performance metric, we ran the remainder of our experiments with simulated parallelism: We modified our parallel generation algorithm to have the workers make their moves in a round-robin fashion rather than simultaneously, as shown in Algorithm 5.4. The master still takes the first solution available, but it is the solution generated with the fewest moves, regardless of the time taken.

We generated 10 000 solutions for each benchmark using Algorithm 5.4 with $P = 1$, 2, 3, 4, 5, 6, 8, 12, 16, 24, 32, 48, and 64. We kept all other parameter values the same. Figure 5.9 compares the resulting move counts for each benchmark for $P = 1, 2$, and 3 to the move counts for the same benchmarks run with workers in parallel. The strong correlation shows that our use of simulated parallelism does not significantly impact the move counts.

Figure 5.10 shows the distributions of speedup $S_m^{(P)}$ across benchmarks as quartiles. The median and maximum speedup increases with the number of workers, but it is sublinear in

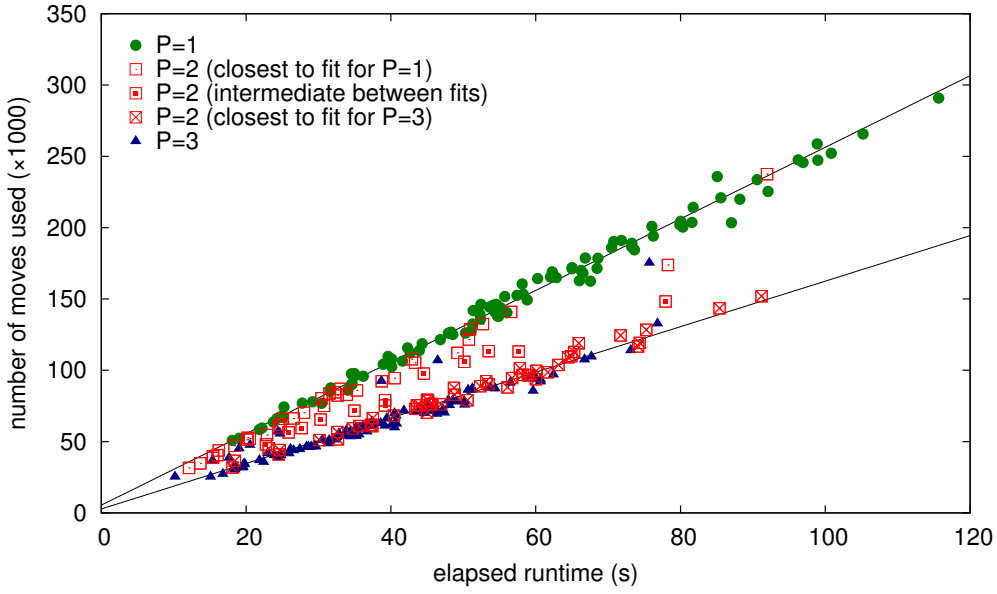


Figure 5.7: Results of parallel generation, using the same data as in Figure 5.6, with linear fits for $P = 1$ and $P = 3$ and the points for $P = 2$ classified by distance to the fit lines.

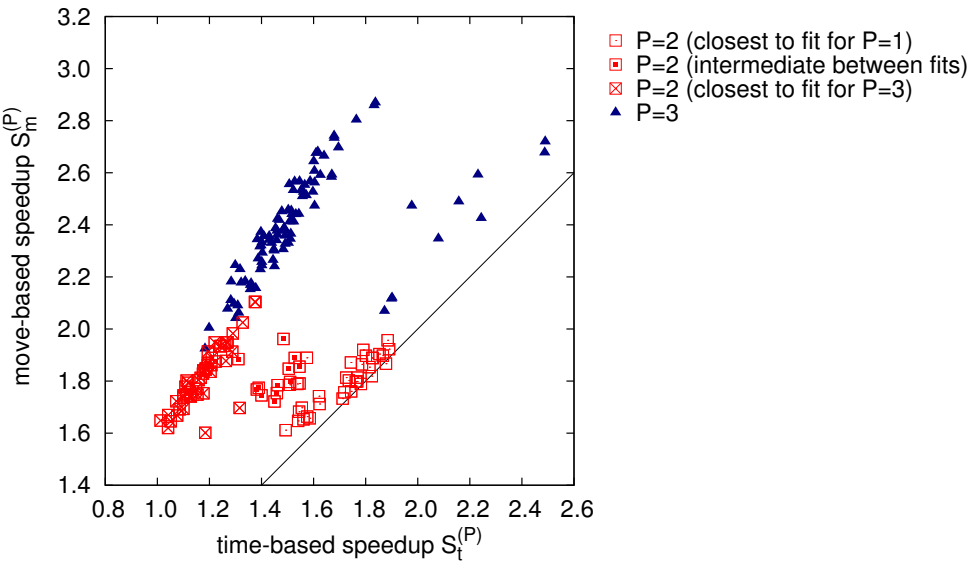


Figure 5.8: Speedup ratios derived from elapsed times and move counts in Figure 5.6, with the same group markers for $P = 2$ as in Figure 5.7.

almost every case. The only exceptions are a few benchmarks run with two workers that had speedup slightly greater than 2. In contrast with the results in Figure 5.5, here the speedup becomes less predictable as the number of workers increases.

Figure 5.11 shows comparisons of the projected speedup $\tilde{S}_m^{(P)}$ and actual speedup $S_m^{(P)}$ for each benchmark and value of P. In every case the actual speedup is less than the projection, consistent with the description of the projection as an upper bound in Section 5.2. For low values of P, the discrepancy is relatively small, but it grows with P, until the ratio $S_m^{(P)}/\tilde{S}_m^{(P)}$

Algorithm 5.4 GENERATE (with simulated parallelism)

Given: control assignment (u, v); load probability p_{ld}; formula $\varphi(u, v, x, y)$

```
1:  for i := 1 to P do
2:      with probability p_ld do
3:          (u,v)^i := (u,v)
4:          t_i := 0
5:  loop
6:      I := (1,...,P)
7:      randomize order of I
8:      for each i ∈ I do
9:          (x,y)^i := MOVE((u,v,x,y)^i, t_i)
10:         t_i := t_i + 1
11:         if β_i > 0 then
12:             β_i := β_i − 1
13:         if φ((u,v,x,y)^i) and β_i = 0 and (u,v)^i = (u,v) then
14:             β_i := β_0
15:             return (x,y)^i
```

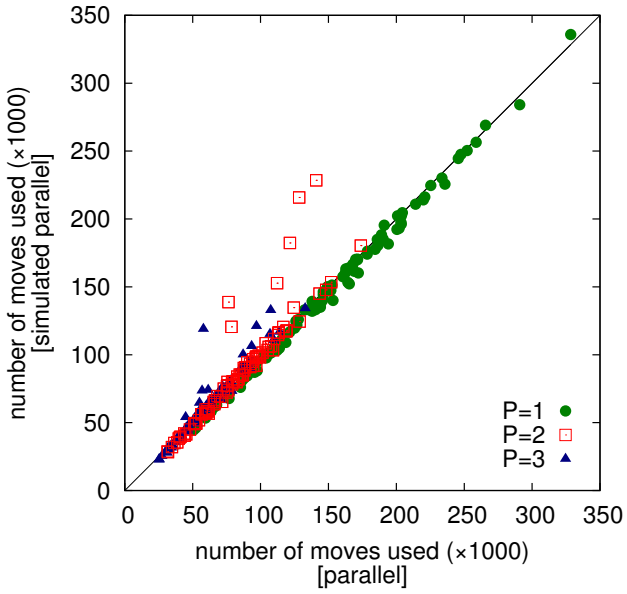


Figure 5.9: Move counts for generation with true and simulated parallelism.

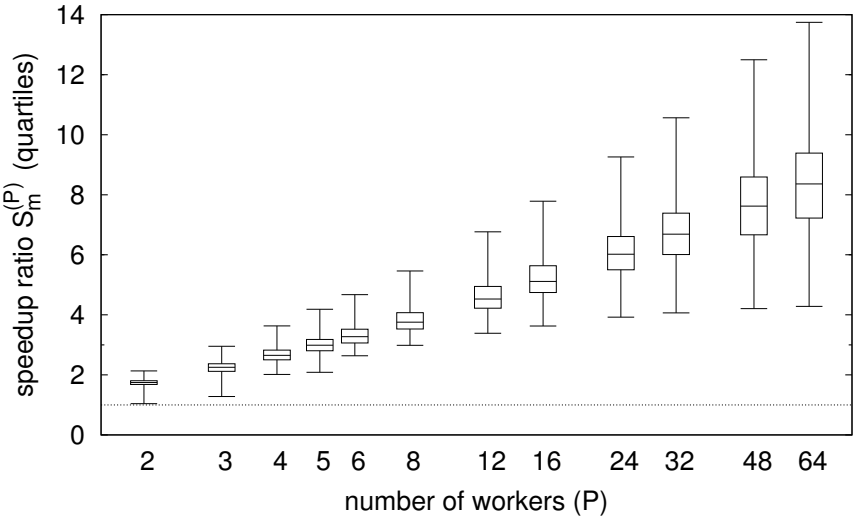


Figure 5.10: Distributions of move-based speedup from simulated parallel generation.

approaches 0.5.

The increasing discrepancy between the projected and observed speedup is likely due to the fact that the projection does not take into account correlation between workers. Some combinations of constraints and control values are more difficult to solve than others, so that they require more moves, on average. In our experiments we applied the same control values to all workers at the same time, so the workers were simultaneously subject to the increased difficulty. We expect positive correlation of moves per solution as a consequence of this. The correlation results in reduced speedup relative to the projection, as described in Section 5.2.

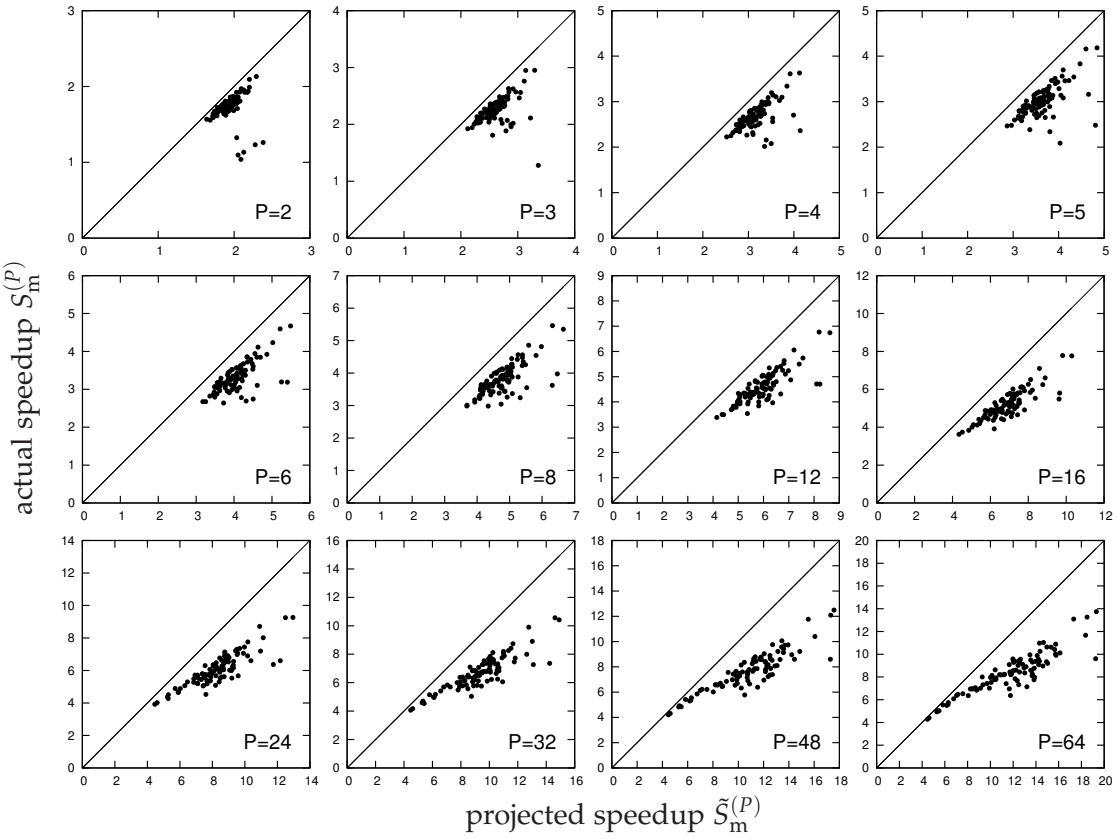


Figure 5.11: Projected and measured speedup ratios from simulated parallel generation.

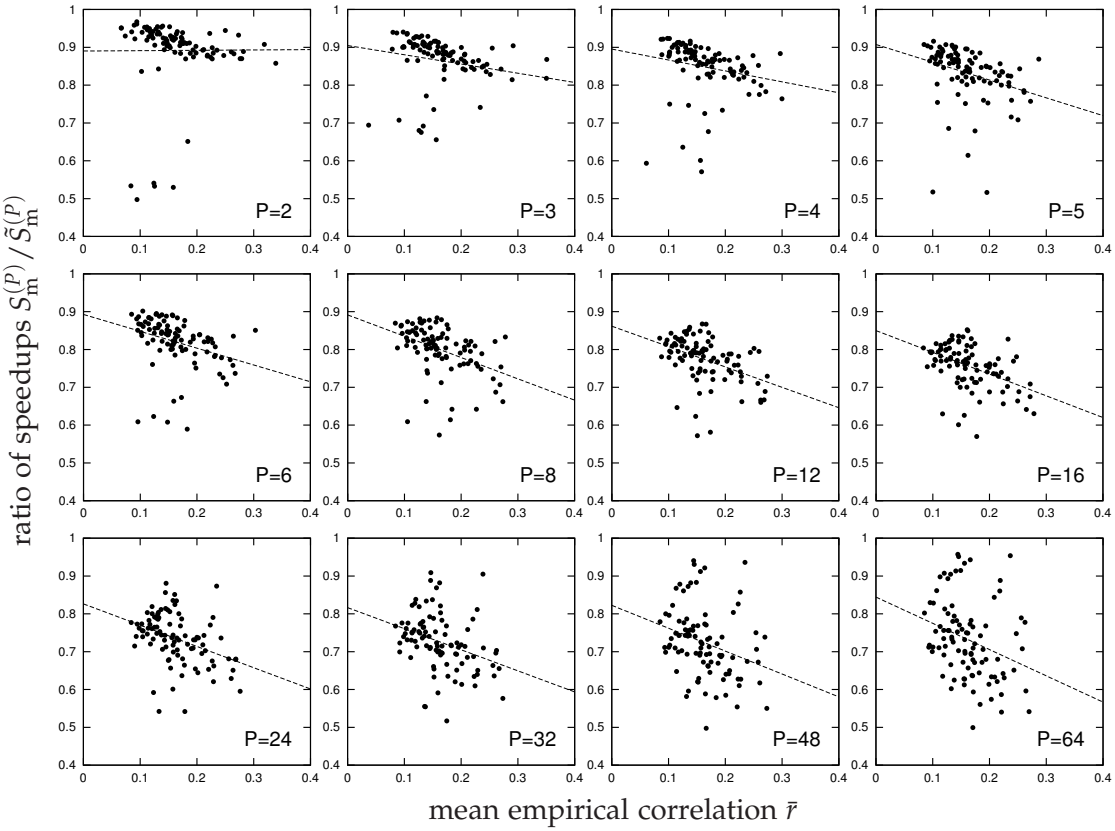


Figure 5.12: Mean empirical correlation of moves per solution per worker versus ratio of actual to projected speedup.

To check our hypothesis about the effect of correlation on speedup, we compare the mean empirical correlation in the moves per solution made by each worker with the ratio $S_m^{(P)} / \tilde{S}_m^{(P)}$ of actual to projected speedup (how much the observed results fell sort of the projection). Collecting data to compute the correlation required a slight modification to Algorithm 5.4: Each worker was allowed to make as many moves as necessary to reach a solution, even after the first solution was found. Figure 5.12 shows the comparison, along with least-squares linear fits for the data (including outliers). The data supports our hypothesis: The ratio of speedups—in other words, the performance relative to projections—falls as the correlation increases, and the decline appears steepest for the largest values of P.

Correcting the projections to account for correlation is difficult. Even if we assume that the distribution of moves per solution is the same for all workers, there are many possible joint distributions with the same mean correlation, and we would need much more data to approximate one of them than our experiments produced. However, there is one joint distribution that we can easily construct with a given move-count correlation, and it serves for a crude check on our results: Suppose that the empirical distribution of move counts for a single worker working alone is $\hat{F}^{(1)}(t)$, and that the empirical mean correlation for P workers is $\bar{r}$, where $\bar{r} \geq 0$. Then we use the following mixture distribution for the projected move counts of the P workers:

$$\tilde{F}^{(P)}(t) = \bar{r}\hat{F}^{(1)}(t) + (1 - \bar{r})\left(1 - (1 - \hat{F}^{(1)}(t))^P\right) \tag{5.2}$$

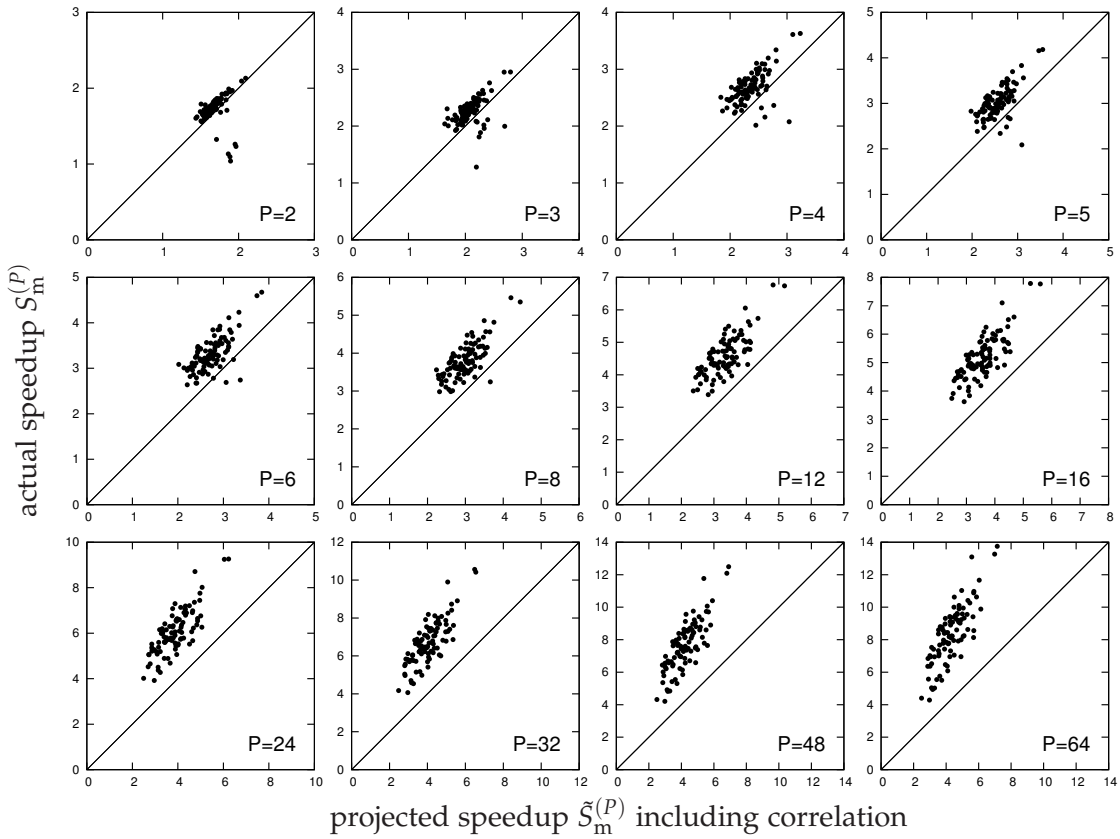


Figure 5.13: Projected and measured speedup ratios from simulated parallel generation, including empirical correlations in projections.

That is, with probability $\bar{r}$ the workers are perfectly correlated, all making the same number of moves, and otherwise their move counts are i.i.d. We expect the projected move counts to be pessimistic, because perfectly correlated move counts give the worst possible speedup—none—while the actual distribution has some variation that allows for positive speedup.

Figure 5.13 compares the projected speedup using the correlated mixture distribution with the actual speedup. The actual speedup is greater than the projection except for a few benchmarks at low values of P, and the pessimism of the projection increases with P. The plots lend further support to our correlation hypothesis.

Correlation may explain the discrepancy between the projected and observed speedup shown in Figure 5.11, but a second discrepancy remains that still needs explanation: The projections shown in that figure, although optimistic, are significantly lower than the projected speedup ratios derived from the distributions of time per solution, as shown in Figure 5.3. The likely reason for it is greater dispersion in the solution-time distributions than in the move-count distributions, due to variability in time per move. Greater dispersion yields greater speedup from parallelization because the minimum value (e.g., solution time) is farther from the other values. To check this explanation, we computed the *coefficient of variation* (CV), a standard measure of dispersion, of the time per solution and the number of moves per solution. The CV is defined as $\frac{\sigma}{\mu}$, the ratio of the standard deviation to the mean. The solution-time data was from our sequential algorithm; this is the same data set used for Figure 5.1. The move-count data is from our

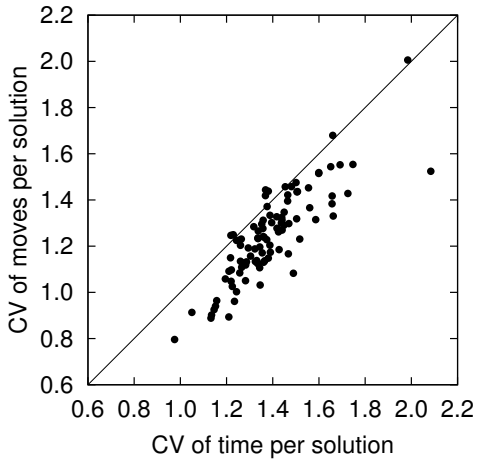


Figure 5.14: Coefficients of variation (CVs) for time and move count per solution.

simulated parallel algorithm with one worker; this data was also used for Figure 5.10.

Figure 5.14 compares the CVs for all 100 benchmarks. In almost every case, the CV of the move count is less than the CV of the solution time. That is, the solution time is indeed more dispersed than the move count. This result supports our explanation for the discrepancy between the time-based and move-based projected speedup. It also suggests that our parallel generation algorithm, if run with adequate hardware parallelism, would provide greater speedup in runtime than the values reported in this section.

5.4.3 Evaluation of Control-Value Caching

When the number of possible assignments to control variables is small, and we set the parameter $p_{\text{ld}} < 1$ so that workers do not load new control values immediately, the control values in the workers act as a cache: The control values from the testbench may match the values loaded by a worker in a previous assignment, and that worker's state may already be a solution.

To evaluate this caching behavior in our algorithm, we generated 40 benchmarks with a small number of control assignments. These benchmarks are similar to the ones described in Section 5.1, but they do not have integer control variables v_i nor constraints of the form $y_i \geq v_i$. Instead, they have constants a_i and 8 Boolean control variables $u_1, \ldots, u_8$, and each clause (except for the static-range clauses) includes one control variable. For example, the following are possible clauses in the benchmarks:

$$u_1 \vee [y_1 \geq a_1] \vee [y_2 \geq a_2] \vee [y_3 < a_3]$$
$$\neg u_2 \vee [y_1 + y_2 + (1000 - y_3) \geq 1000]$$

Each benchmark has 100 random clauses in addition to static-range clauses for $y_1, \ldots, y_{20}$.

We generated 10 000 solutions for each benchmark using Algorithm 5.4 with $P = 1$, 2, 3, 4, 5, 6, 8, 12, 16, 24, 32, 48, and 64 and $p_{\text{ld}} = P^{-1}$. For each benchmark and value of P we varied the effective number of control bits from 1 to 8 by taking $m = 1, \ldots, 8$ and assigning variables $u_1, \ldots, u_m$ randomly and keeping $u_{m+1}, \ldots, u_8$ fixed. For the blocking parameter, we used $\beta_0 = 1$.

The results are shown in Figures 5.15 and 5.16. Figure 5.15 shows the mean speedup $S_{\text{m}}^{(P)}$ across benchmarks for each value of P and m as a 3-dimensional surface. Figure 5.16a and

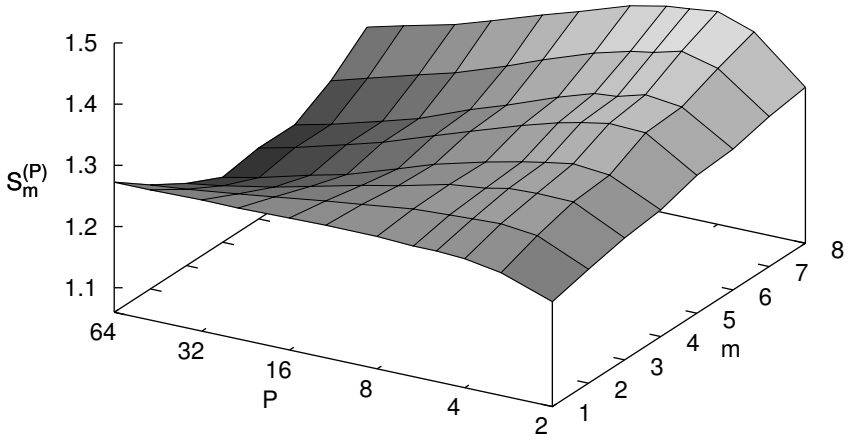


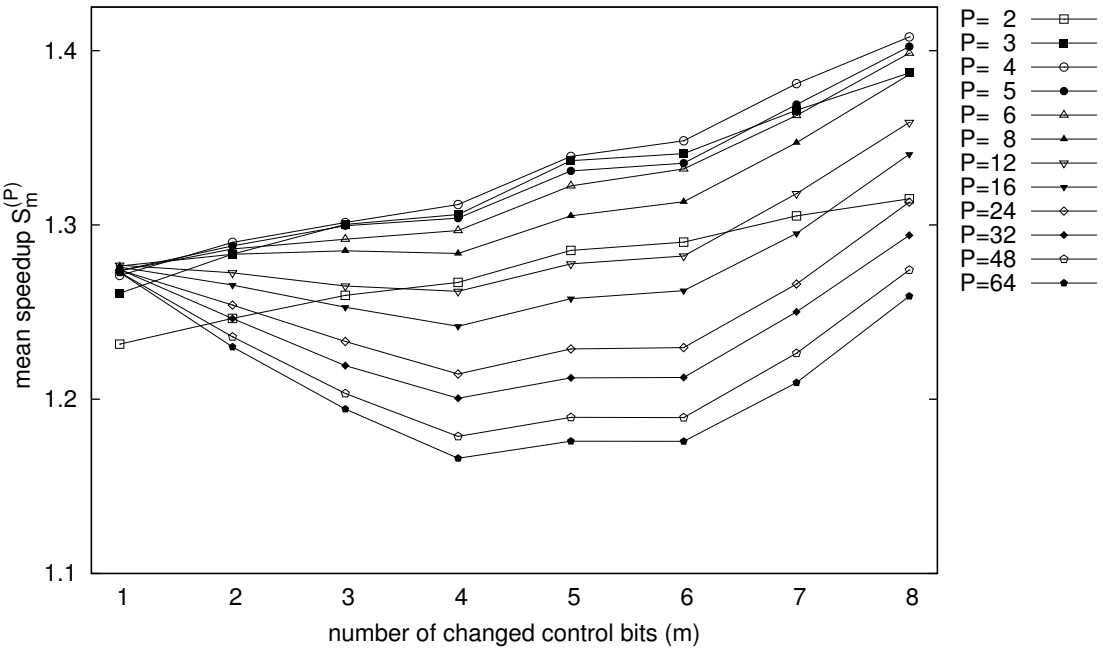
Figure 5.15: Mean speedup $S_m^{(P)}$ across benchmarks for number of workers P and number of assigned control bits m.

Figure 5.16b show different views of this same data. The speedup does not follow a simple trend of decreasing with m (due to caching) and increasing with P. Instead, there are several unexpected patterns in the data: For low P, the speedup increases with m; for high P, it is non-monotonic with the minimum at $m = 4$. Along the other axis, the speedup increases monotonically with P for low m, but for high m, it peaks at $P = 4$ and decreases thereafter.

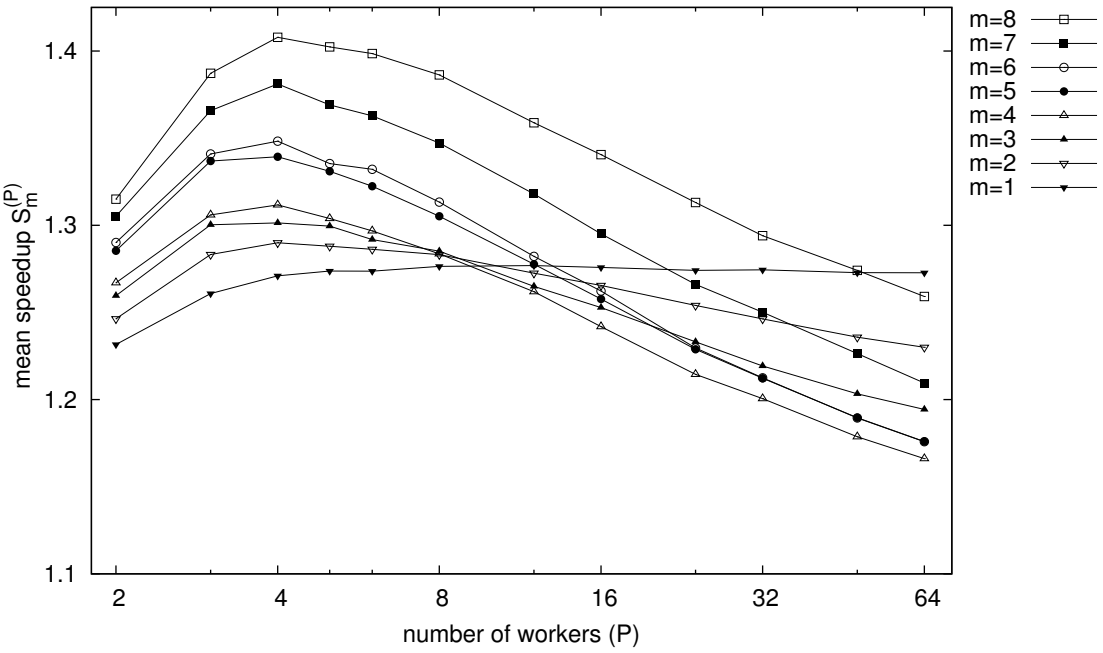
We can show that some of the unexpected trends in these results are consistent with a simplified theoretical projection from empirical data. The projection is based on a distribution similar to the one described by Equation 5.2; it additionally accounts for the fact that the workers load the control values gradually, not all at once. Let $\hat{F}^{(1)}(t)$ be the empirical distribution of moves per solution for a single worker working alone on a benchmark with m control bits changing. Let $\bar{r} > 0$ be the mean empirical correlation of move counts for generation with P workers on the same benchmark with the same value of m. Then we construct $\tilde{F}^{(1)}_{/P}(t)$, the projected distribution of moves per solution for one worker among a set of P workers, and $\tilde{F}^{(P)}(t)$, the projected distribution of move counts for the solutions taken by the master:

$$\begin{aligned}
\tilde{F}^{(1)}_{/P}(t) &= \sum_{\tau=0}^{t-1} \Pr(\text{worker loads } u \text{ after } \tau \text{ moves})\, \hat{F}^{(1)}(t-\tau) \\
&= \sum_{\tau=0}^{t-1} (1-p_{\text{ld}})^{\tau} p_{\text{ld}}\, \hat{F}^{(1)}(t-\tau) \\
&= \sum_{\tau=0}^{t-1} \Big(1-\frac{1}{P}\Big)^{\tau} \frac{1}{P}\, \hat{F}^{(1)}(t-\tau) \\
\tilde{F}^{(P)}(t) &= \bar{r}\hat{F}^{(1)}_{/P}(t) + (1-\bar{r})\big(1-(1-\hat{F}^{(1)}_{/P}(t))^{P}\big)
\end{aligned}$$

Note that this projection does not model caching of control values. Unlike the projected distributions used in previous sections, $\tilde{F}^{(P)}(t)$ as defined here has non-zero value for arbitrarily large t, not only the values of t in the support of the empirical distribution $\hat{F}^{(1)}$. To bound the number of terms when computing projected speedup, we limit t to values such that $\tilde{F}^{(P)}(t) < 1 - 10^{-3}$. That is, we stop when the constructed distribution has probability mass at least 0.999.



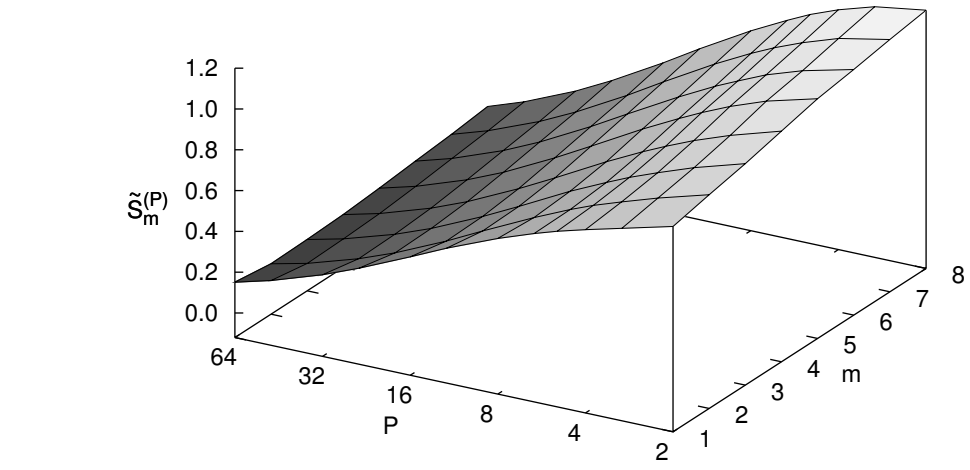
(a)



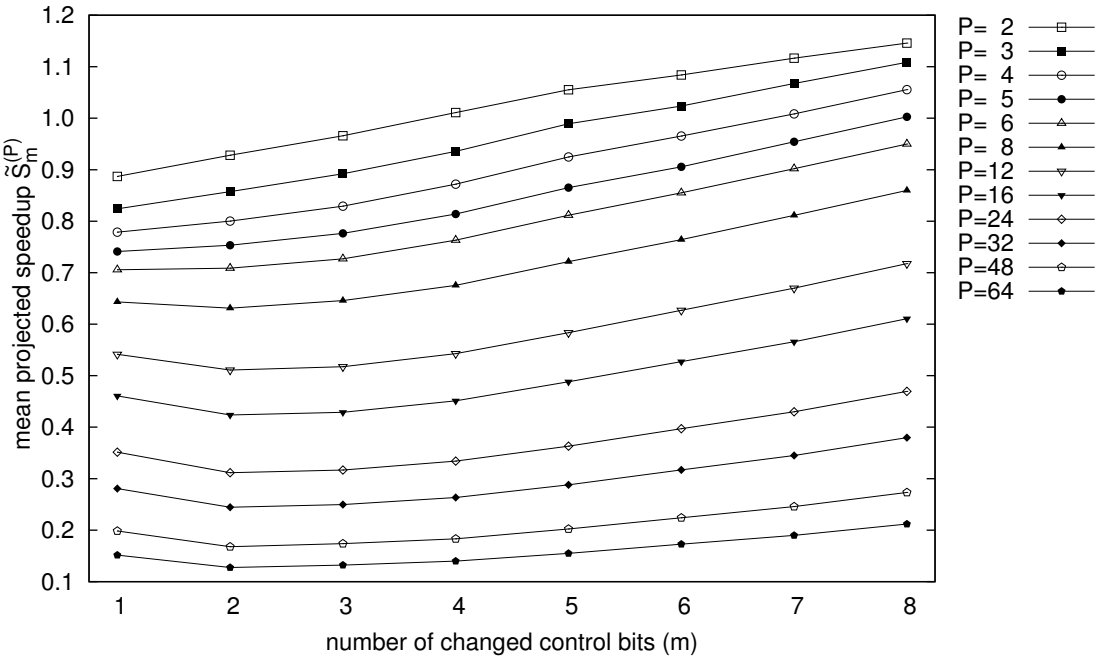
(b)

Figure 5.16: Different views of the same data shown in Figure 5.15.

We computed the projected speedup from this distribution for all benchmarks and values of P and m. The results are shown in Figure 5.17. The range of speedup extends lower than in the actual experimental data, but the data exhibit several of the same trends: The speedup decreases with P for large m and even shows non-monotonicity for high P, albeit more subtly. These projections do not fully explain the experimental data, but they give it credibility.



(a)



(b)

Figure 5.17: Mean projected speedup $S_m^{(P)}$ across benchmarks for number of workers P and number of assigned control bits m, accounting for gradual loading of control values. Both plots show the same data.

The region of the parameter space where the trends in projected speedup differ most from those in the actual speedup is where P is high and m is low. This is also the only region where the speedup seems to show the effect of control caching: It decreases as m increases. This result makes sense: These are the conditions under which the caching hit rate is expected to be highest—the probability of a hit for a single worker is $\frac{1}{2^m}$, so the expected number of hits is $\frac{P}{2^m}$. In general, though, the effects of caching seem to be negligible compared to other effects, including the effect of gradual loading of control values. Therefore, in practice the best speedup is likely to be achieved with immediate loading ($p_{\text{ld}} = 1$).

5.5 Summary

In this chapter we described how we parallelized our stimulus-generation algorithm for faster generation. The potential for speedup in our parallelization scheme arises from variability in generation time per solution. We introduced benchmarks that exhibit this variability. We computed upper bounds on speedup for the benchmarks using theoretical projections from empirical data and derived speedup values analytically to support the plausibility of our projections. We presented experimental results for our benchmarks from two implementations of our parallel algorithm: one using multiple threads and a variant implementation using simulated parallelism; the latter enabled us to evaluate generation for numbers of workers exceeding the number of cores in available hardware. Our results showed that parallelization made generation faster, but the amount of speedup observed was smaller than we projected. We explained the discrepancy as a consequence of correlation between generators. We also presented results from experiments on caching of control values and showed that it did not provide significant benefit. Our investigation of our experimental results illustrates that understanding the performance of parallel algorithms may require analysis deeper than simple runtime measurement.

Chapter 6

Conclusions

We began this dissertation by describing the importance of efficient functional verification in the production of digital integrated circuits. Constrained random simulation has become the dominant approach in state-of-the-art verification because of its scalability, predictability, and ability to handle complex input constraints. For high productivity, the constraint solver that supplies random stimuli for simulation must solve the constraints quickly and produce values that are well distributed over the input space.

As the main contribution of this work, we proposed a generation approach using Markov chain Monte Carlo (MCMC) methods that meets the requirements for productive simulation. In Chapter 2 we introduced the basic principles of MCMC methods. In Chapter 3 we presented our approach, which combines the widely used Metropolis-Hastings algorithm with a local-search satisfiability solver and correlation-reduction techniques. It surpasses existing stimulus-generation methods in speed, robustness, and quality of distribution. The value of our approach is attested by the fact that it has been adopted by developers of industrial verification software for integration into their testbench tool.

The integration work motivated several refinements that we described in Chapter 4. We explained how our algorithm handles dependencies on external data with the addition of control variables. We described how elimination of variables increases the density of solutions in the state space; we applied it both statically as preprocessing and dynamically during generation and showed that static elimination improves the distribution of solutions but dynamic elimination imposes too much overhead.

In Chapter 5 we explored the effects of parallelizing our algorithm. We computed bounds on achievable speedup using theoretical analysis and empirical data, and we showed that that parallelism enables faster generation. We also investigated conditions that limit the speedup achieved.

Our stimulus-generation algorithm is able to run efficiently and produce well-distributed stimuli because we assume a restriction on the input constraints: We expect them to be relatively easy to solve and to have a dense solution space. When this assumption does not hold (e.g., as when the constraints involve mostly Boolean variables or the solution space is highly fragmented), the efficiency and quality of distribution are likely to be low. Nevertheless, for many constraints used in practice, our MCMC-based approach is a good way to generate stimuli for constrained random simulation.

A direction of future work that could bring further improvements in verification efficiency is implementation of our approach in hardware rather than software. Designs are some-

times run in hardware emulators or field-programmable gate arrays instead of logic simulators because they are much faster than software. The communication between a software testbench and a design in hardware is a major bottleneck for constrained random simulation, and an implementation of our algorithm as digital logic would remove this bottleneck.

Bibliography

[Acc04] Accellera Organization. *Property Specification Language, Reference Manual, Version 1.1*, June 2004.

[Bré99] Pierre Brémaud. *Markov Chains: Gibbs fields, Monte Carlo Simulation, and Queues*, volume 31 of *Texts in Applied Mathematics*. Springer-Verlag, New York, 1999.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, 35:677–691, Aug 1986.

[BSST09] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[CIJ+95] A. Chandra, V. Iyengar, D. Jameson, R. Jawalekar, I. Nair, B. Rosen, M. Mullen, J. Yoon, R. Armoni, D. Geist, and Y. Wolfsthal. AVPGEN-a test generator for architecture verification. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(2):188–200, 1995.

[DKBE02] Rina Dechter, Kalev Kask, Eyal Bin, and Roy Emek. Generating random solutions for constraint satisfaction problems. In *Eighteenth national conference on Artificial intelligence*, pages 15–21, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Comms. ACM*, 5:394–397, July 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J.˜ACM*, 7:201–215, 1960.

[ES03] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proc. 6th Int'l Conf. Theory & Appl. Satisfiability Testing (SAT)*, pages 502–518, May 2003.

[Fil91] James A. Fill. Eigenvalue bounds on convergence to stationarity for nonreversible Markov chains, with an application to the exclusion process. *The Annals of Applied Probability*, 1(1):62–87, February 1991.

[FS01] Daan Frenkel and Berend Smit. *Understanding Molecular Simulation: From Algorithms to Applications*, volume 1 of *Computational Science Series*. Academic Press, San Diego, second edition, November 2001.

[GD06] Vibhav Gogate and Rina Dechter. A new algorithm for sampling CSP solutions uniformly at random. Technical report, School of Information and Computer Science, University of California, Irvine, May 2006.

[GeyOL] Charles Geyer. Burn-in is unnecessary. Available: http://www.stat.umn.edu/~charlie/mcmc/burn.html. Loaded 6 March 2010.

[Gey92] Charles J. Geyer. [Practical Markov Chain Monte Carlo]: Rejoinder. *Statistical Science*, 7(4):502–503, 1992.

[GG84] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 6(6):721–741, November 1984.

[GLMS02] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic, Boston, 2002.

[GS90] Alan E. Gelfand and Adrian F. M. Smith. Sampling-based approaches to calculating marginal densities. *J. Amer. Statist. Assoc.*, 85(410):398–409, 1990.

[Has70] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57:97–109, 1970.

[HB01] Ian Holmes and William J. Bruno. Evolutionary HMMs: a Bayesian approach to multiple alignment. *Bioinformatics*, 17(9):803–820, September 2001.

[IJ04] Sasan Iman and Sunita Joshi. *The e Hardware Verification Language*. Kluwer Academic, Norwell, MA, USA, 2004.

[ITR09] International Technology Roadmap for Semiconductors 2009 edition. Available: http://www.itrs.net.

[Iye03] Mahesh A. Iyer. RACE: A word-level ATPG-based constraints solver system for smart random simulation. In *IEEE International Test Conference (ITC)*, pages 299–308, Charlotte, NC, United States, September 2003.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[KJR+08] Hyondeuk Kim, Hoonsang Jin, Kavita Ravi, Petr Spacek, John Pierce, Bob Kurshan, and Fabio Somenzi. Application of formal word-level analysis to constrained random simulation. In *Computer Aided Verification*, pages 487–490, Princeton, NJ, July 2008. Springer-Verlag.

[KK07] Nathan Kitchen and Andreas Kuehlmann. Stimulus generation for constrained random simulation. In *IEEE/ACM Int'l Conf. on CAD*, pages 258–265, Nov 2007.

[KK09] Nathan Kitchen and Andreas Kuehlmann. A Markov chain Monte Carlo sampler for mixed Boolean/integer constraints. In *Computer Aided Verification (CAV'09)*, pages 446–461, Grenoble, France, July 2009. Springer-Verlag.

[KS00] James H. Kukula and Thomas R. Shiple. Building circuits from relations. In *Proc. 12th Int'l Conf. Computer-Aided Verif. (CAV)*, pages 113–123. Springer-Verlag, 2000.

[LPW08] David A. Levin, Yuval Peres, and Elizabeth L. Wilmer. *Markov Chains and Mixing Times*. American Mathematical Society, 2008.

[Mes94] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennessee, 1994.

[MFWvH01] D. Metzler, R. Fleissner, A. Wakolbinger, and A. von Haeseler. Assessing variability by joint sampling of alignments and mutation rates. *J Mol Evol*, 53(6):660–669, December 2001.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Linto Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proc. 38th ACM/IEEE Design Automation Conf.*, pages 530–535, June 2001.

[MNL99] Bob Mau, Michael A. Newton, and Bret Larget. Bayesian phylogenetic inference via Markov chain Monte Carlo methods. *Biometrics*, 55(1):1–12, March 1999.

[MRR+53] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equations of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1087–1092, June 1953.

[NB99] M. E. J. Newman and G. T. Barkema. *Monte Carlo Methods in Statistical Physics*. Oxford University Press, New York, 1999.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline P. Farrel. *Pthreads Programming*. O'Reilly and Associates, Inc., 1996.

[NQXL02] T. Niu, Z. S. Qin, X. Xu, and J. S. Liu. Bayesian haplotype inference for multiple linked single-nucleotide polymorphisms. *American Journal of Human Genetics*, 70(1):157–169, January 2002.

[SCJI07] Sriram Sankaranarayanan, Richard M. Chang, Guofei Jiang, and Franjo Ivancic. State space exploration using feedback constraint generation and Monte-Carlo sampling. In *Proc. ESEC-FSE '07: – 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 321–330, New York, NY, USA, 2007. ACM.

[SD02] Kanna Shimizu and David L. Dill. Deriving a simulation input generator and a coverage metric from a formal specification. In *Proc. 39th Design Automation Conf.*, pages 801–806, New Orleans, LA, United States, Jun 2002.

[SDF03] Stuart Sutherland, Simon Davidmann, and Peter Flake. *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling*. Kluwer Academic, Norwell, MA, USA, 2003.

[SHJ05] Che-Hua Shih, Juinn-Dar Huang, and Jing-Yang Jou. Stimulus generation for interface protocol verification using the nondeterministic extended finite state machine model. In *Proceedings of the Tenth IEEE International High-Level Design Validation and Test Workshop*, pages 87–93. IEEE Computer Society, 2005.

[SKC93] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In Michael Trick and David S. Johnson, editors, *Proc. 2nd DIMACS Challenge on Cliques, Coloring, and Satisfiability*, Providence RI, 1993.

[Som05] Fabio Somenzi. CUDD: CU decision diagram package, release 2.4.0, 2005.

[TW98] Claudia Tebaldi and Mike West. Bayesian inference on network traffic using link count data. *Journal of the American Statistical Association*, 93(442):557–573, 1998.

[WES04] Wei Wei, Jordan Erenrich, and Bart Selman. Towards efficient sampling: Exploiting random walk strategies. In *Proc. Nat'l Conf. Artificial Intelligence*, pages 670–676, Jul 2004.

[XPC+05] Zhong Xiu, David A. Papa, Philip Chong, Christoph Albrecht, Andreas Kuehlmann, Rob A. Rutenbar, and Igor L. Markov. Early research experience with OpenAccess Gear: An open source development environment for physical design. In *Proc. ACM Int'l Symp. Phys. Design (ISPD)*, pages 94–100, 2005.

[YAPA04] Jun Yuan, Adnan Aziz, Carl Pixley, and Ken Albin. Simplifying Boolean constraint solving for random simulation-vector generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(3):412–420, March 2004.

[YSP+99] Jun Yuan, Kurt Shultz, Carl Pixley, Hillel Miller, and Adnan Aziz. Modeling design constraints and biasing in simulation using BDDs. In *Digest Tech. Papers IEEE/ACM Int'l Conf. Computer-Aided Design*, pages 584–589, Nov 1999.