Low-latency SAT Solving on Multicore Processors with Priority Scheduling and XOR Partitioning

Stephen M. Plaza*, Igor L. Markov*†, Valeria Bertacco*

*EECS Department, University of Michigan, Ann Arbor, MI

†Synplicity, Inc., Sunnyvale, CA

{splaza, imarkov, valeria}@umich.edu

ABSTRACT

As multicore processors become prevalent, computational methodologies for decision-making, combinatorial optimization, optimal design, and formal verification must adapt to better utilize available CPU resources. We propose new techniques to exploit the power offered by upcoming shared-memory multicore/multi-CPU architectures to boost the performance of solvers for fundamental NP-hard problems, such as Boolean and Pseudo-Boolean SAT. We develop an algorithmic paradigm for parallel solvers centered on 1) a scheduling strategy to reduce the average latency for solving batches of instances of varying complexity, and 2) a novel, balanced decomposition of a SAT instance's search space among multiple threads. These techniques are implemented in a software library that provides parallel-solving services to user applications. Evaluation on an eight-core workstation shows significantly reduced latency for solving multiple SAT instances in parallel, as well as greater CPU utilization.

1. INTRODUCTION

Modern CPUs and SoCs exhibit greater functional complexity, fueling numerous challenges in design and verification. With many related tasks shown NP-hard, even small increases in design size may require a disproportionate increase in computing resources. Managing this fast-growing demand of resources has always been a challenge in the EDA field, and has often set the limit of how complex of a design could be tackled. Hence, the opportunity provided by the recent offering of chip multicore architectures, which can execute multiple threads simultaneously on a shared memory platform, is not one to pass on. However, it currently remains unclear how *leading-edge* algorithms for key problems can be mapped onto such architectures to ensure powerful speed-ups. Our work addresses precisely this issue for the Boolean SATisfiability problem.

Boolean SATisfiability is one of the best-known NP-complete problems with numerous applications in EDA [16]. Recent advances in DPLL-based SAT solvers [16, 17] allow many practical problems to be solved quickly, and have facilitated their widespread deployment in the industry. Fundamental verification techniques, such as equivalence checking [8] and model checking [4], make extensive use of SAT solvers to bypass the often prohibitive memory requirements of BDD-based strategies. However, the performance of state-of-the-art SAT solvers varies widely and unpredictably on problem instances, with some instances being intractable. Also, a slight change in algorithm parameters can sometime affect the runtime of a solver on the same instance by orders of magnitude.

"Intrinsically parallel" tasks, such as multimedia processing, may achieve N times speed-up by using N cores (assuming that sufficient memory bandwidth is available and that cache coherency is not a bottleneck). However, combinatorial optimization and search problems, such as SAT-solving and integer linear programming, are much harder to parallelize. The straightforward solution — to process in parallel different branches of a given decision — often fails miserably in practice because such branches are not independent in leading-edge solvers that rely on recursive learning. The recent "View from Berkeley" project [3] designates branch and bound techniques as one of thirteen core computational categories for which parallel algorithms must be developed. In this work, we propose new techniques to parallelize state-of-the-art SAT solving.

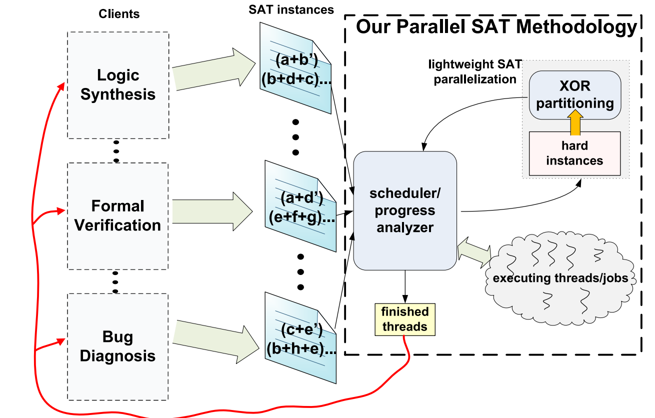

Figure 1: High-level flow of our parallel SAT methodology. We introduce a scheduler for completing a batch of SAT instances of varying complexity and a light-weight parallel strategy for handling hard SAT instances.

In addition to formal verification, state-of-the-art design optimizations require solving multiple SAT instances, such as SAT sweeping in logic synthesis [29, 19], SAT-based technology mapping for FPGAs [13], and logic restructuring [20] require solving multiple SAT instances. These SAT instances are currently found in the bottlenecks of key EDA algorithms, and solving them in parallel offers a chance to speed up a broad range of EDA tools. The observation that these instances exhibit varying complexity is not only curious, but also leads to improved parallel solving techniques. In addition to threads dedicated to solve SAT instances, domain-specific threads may formulate and simplify new SAT instances, as well as interpret collected solutions. Shared-memory systems and multicore CPUs are particularly amenable to such parallelization strategies.

We first introduce a novel architecture for scheduling and solving multiple instances of hard problems, such as those arising in SAT, on shared-memory and multicore CPUs, as illustrated in Figure 1. The first problem we address is the scheduling of M SAT instances on N processors when $M > N$. Take, for example,

the case $N = 1$. If runtimes are known for each instance in advance, then scheduling instances in the increasing order of runtime guarantees the best *batch latency*, which we define as the sum of completion times of all instances from the beginning of the batch. In other words, a long-running job will not delay numerous small jobs. Scheduling for N processors, without *a priori* runtime information, is harder, and our paper is the first to address this problem. Furthermore, many applications generate SAT instances dynamically rather than in batches — the technique we propose handles this case as well.

The need to parallelize a single SAT instance arises primarily when no other instances remain to keep all available cores busy. In a large verification environment, this is most likely to happen with only the hardest SAT instances, which gives us an additional assumption that can be used by parallel SAT algorithms. In our framework, complex jobs can run sequentially for some time before a decision to parallelize them is made.

In this paper we achieve two goals: 1) the minimization of the average latency for solving a collection of SAT problems while ensuring maximum resource utilization and 2) the minimization of runtime for large problem instances by exploiting parallel resources. To reduce the average latency of a collection of SAT instances, we introduce a novel scheduling algorithm that can utilize a predicted distribution of SAT runtimes, and emphasizes synergies between time-slicing and batch scheduling. We achieve a 20% average latency improvement while increasing utilization of parallel resources. To reduce the runtime for single large instances, we consider a novel partitioning scheme based on adding XOR constraints that evenly divides the search-space of SAT instances independent of its underlying structure. We exploit a theoretical result from [21] on randomized polynomial-time algorithms, where adding a limited number of random XOR constraints to a SAT instance can reduce a SAT instance with multiple solutions to a SAT instance with one solution. Our work is the first to apply this result to search-space partitioning in multicore SAT solving, circumventing a major pitfall common in parallel SAT solver algorithm — unbalanced partitioning [24]. We further observe that search-space partitioning is best performed when the random restart frequency is low, which occurs after initially solving the problem sequentially. We validate our work by extensive experiments on an eight-core system and improve resource utilization by 60.5% over prior work that uses solver portfolios. Our contributions are as follows:

1. An approach to scheduling multiple SAT problem instances in a way that minimizes the average latency.
2. A novel partitioning of the SAT search space using XOR constraints which produces sub-problems of similar complexity and evenly partitions the solution space.
3. A lightweight SAT parallelization strategy that can be easily adapted to improve the performance of any state-of-the-art DPLL-based solver.
4. An open-ended parallel SAT-solving methodology that combines our lightweight parallelization strategy with other heuristics to improve the overall resource allocation.

In Section 2, we survey previous work on parallel SAT solving. Section 3 introduces our scheduling algorithm for handling multiple SAT instances of varying complexity in a parallel setting. We discuss the limitations of previous parallel SAT research in Section 4. In Section 5, we propose a partitioning strategy that provides search-space division along with our strategy for parallelization. We analyze the effectiveness of our approach in Section 6 and conclude in Section 7.

2. PRELIMINARIES

For a Boolean formula F in conjunctive normal form (CNF), the SAT problem requires (i) choosing an assignment for a set of variables V that satisfies F, or (ii) confirming that no such assignment exists. The basic approach to solving SAT is a backtracking framework referred to as DPLL [6]. Several innovations such as non-chronological backtracking, conflict-driven learning, and decision heuristics greatly improve upon this approach [16, 17, 25].

First, we outline previous efforts on improving SAT performance in a parallel setting. Then we explain how leading-edge SAT solvers often exhibit characteristic *long-tail* runtime distributions, which we exploit later in our work.

2.1 Previous Approaches to Parallel SAT

Parallel SAT solving strategies have explored coarse-grain or fine-grain parallelization. Fine-grain parallelization strategies target Boolean Constraint Propagation (BCP) which contributes to the largest percentage of runtime for most SAT solvers. In BCP, each variable assignment is checked against all relevant clauses and any implications are propagated. BCP can be parallelized by dividing the clause database among n different solvers so that BCP computation time of each solver is approximately $\frac{1}{n}$ the original. Coarse-grain parallelization strategies typically involve assigning a SAT solver to different parts of the search space.

Fine-grain parallelization. The performance of fine-grain parallelization depends on the partitioning of clauses among the solvers, where an ideal partition ensures an even distribution of BCP costs while minimizing the implications that need to be communicated between each solver. This strategy also requires low-latency inter-solver communication meaning that contention for system locks as implemented on general microprocessors could exacerbate performance. Therefore, fine-grain parallelization has been examined on specialized architectures [27] that can minimize any communication bottlenecks. Also, in [28, 1], significant parallelization was exploited by mapping a SAT instance to an FPGA and allowing BCP to evaluate several clauses simultaneously. The flexibility and scalability of this approach is limited because each instance needs to be compiled to a specific architecture and conflict-driven learning is difficult to effectively implement.

Coarse-grain parallelization. The runtime of an individual problem can also be improved in a parallel setting by using a solver portfolio [10], where multiple SAT heuristics are executed in parallel and the fastest heuristic determines the runtime for the problem. A solver portfolio is also one way of countering the variability that backtrack-style SAT solvers experience on many practical SAT instances [11]. Because one heuristic may perform better than another on certain types of problems, one can reduce the risk of choosing the wrong heuristic by running both. Although parallelization here consists of running multiple versions of the same problem simultaneously, if the runtime difference between these heuristics is significant, a solver portfolio can yield runtime improvements.

However, using a portfolio solver does not guarantee high resource utilization as each heuristic may perform similarly on any given instance or one heuristic may dominate the others. The primary limitation of solver portfolios is that there is no good mechanism to coordinate the efforts of these heuristics and the randomness inherent to them. To better coordinate efforts, other approaches consider analyzing different parts of the search space in parallel [18, 24, 5, 14]. If the parts of the search space are disjoint, the solution to the problem can be determined through the processing of these parts in isolation. However, in practice, the similarities that often exist between different parts of the search space mean that

redundant analysis is done across the different parts. To counter this, the authors in [14] develop an approach to explore disjoint parts of the search space where large shared memory in a multicore system is used to transfer learned information between them. The approach considers dividing the problem instance using different variable assignments called *guiding paths*, as originally described in [24]. One major limitation of this type of search space partitioning is that poor partitions can produce complicated sub-problems with widely varying structure and complexity.

The benefits of learning between solvers working on different parts of the search space in parallel suggest potential super-linear improvement. However, the improvements achieved by current strategies seem more consistent with the inherent variability of solving many real-world SAT problems and the effect of randomization on reducing this variability. Through intelligent randomization strategies, sequential solvers can often avoid complicated parts of the search space and outperform their parallel counterparts.

2.2 Runtime Variability in SAT Solving

While DPLL SAT solvers typically struggle on randomly generated instances, most practical SAT instances possess regular structure and can be solved much faster. However, it has been observed that many practical instances experience exponential runtime variability [11] when using backtrack-style SAT solvers without *random restarting*. In particular, many instances exhibit heavy-tail behavior, meaning that the runtime variance of solving a SAT instance when sampling from existing competitive algorithms is exponential.

DEFINITION 1. *For a random variable X, a heavy-tail probability distribution occurs when $Pr[X > x] \propto x^{-\alpha}$ as $x \rightarrow \infty$ for $0 < \alpha < 2$.*

If the cumulative probability does not converge to 1 quickly enough, the distribution will have a heavy-tail. More specifically, the variance of X is ∞, and when $\alpha < 1$ the mean is also ∞. In analyzing SAT performance, X denotes the number of backtracks (this correlates to the difficulty of solving the problem) required to solve a given instance. Also, since the maximum runtime is exponential, the bounded heavy-tail produces variance that is actually exponential in the number of backtracks.

Effective random restarting strategies, which are now extensively used in DPLL-based solvers and involve a worst-case polynomial number of restarts, can eliminate heavy-tail behavior [11] and also target hard problems which have *fat-tails* ($\alpha > 0$) that are not heavy. Intuitively, random restarts prevent a solver from getting stuck in a difficult part of the search space. Portfolio strategies [10] offer similar benefits because each heuristic explores different parts of the search space. Furthermore, each heuristic can utilize multiple restarting strategies, which in turn can produce more improvement.

Backdoor variables. In [22], it was observed that many common problems possess a small **backdoor set**. A backdoor for a SAT instance is variable set that under some assignment produces a sub-problem solvable in polynomial time. This occurs when the remaining problem can be solved by a linear-time 2-SAT algorithm.

DEFINITION 2. *Given a Boolean formula $F(V)$, variables $B \subseteq V$, and a variable assignment $A_B \in \{0,1\}^{|B|}$, B is a backdoor if $\exists A_B[F_{A_B} \in P \wedge F_{A_B} \neq 0]$*

In other words, if assigning a set of variables results in an instance with a satisfying assignment that can be solved in polynomial time, the set forms a backdoor.

DEFINITION 3. *Given a Boolean formula $F(V)$, a partial variable assignment B is a* **strong backdoor** *if $\forall A_B[F_{A_B} \in P]$.*

For unsatisfiable instances, this would require exploration of $2^{|B|}$ combinations and a total runtime of $2^{|B|}P(F_{A_B})$ where $P(F_{A_B})$ is the runtime of the polynomial algorithm under a given assignment. Empirical evaluation in [22] suggests that many practical problems have $|B| \propto lg(|V|)$ resulting in total runtime of $|V|P(F_{A_B})$ if the backdoor set is known. Although determining this set is not always computationally feasible, decision heuristics like VSIDS which favor variable assignments that decide the problem quickly, implicitly look for such sets. It was also explained in [22] that randomly generated instances have considerably larger backdoors of around 30% of $|V|$. Efficiently determining a backdoor, explicitly or implicitly, is critical for the performance of a SAT solver.

3. SCHEDULING SAT INSTANCES OF VARYING DIFFICULTY

Given M different SAT instances and an N-threaded machine, we desire to solve them in a way that satisfies the following:

$$min(\sum^{M} T_c(m)) \text{ where } \forall_t S_t \leq N \quad (1)$$

where $T_c(m)$ is the completion time for problem m and S_t is the number of instances being solved for a particular time-slice t. Note, when $N = 1$, this formulation considers the case of only a single available thread of execution. Ideally, the completion time T_c for the final m_f for N threads should be N-times smaller than for $N = 1$ to fully utilize the parallel resources.

Optimizing the objective above subject to resource constraints can lead to a schedule that minimizes the total latency for finishing all the SAT instances. Assuming that incoming SAT instances are independent and equally important to solve, minimizing latency is a way to ensure feedback from as many problem instances as quickly as possible. This may unblock the largest number of client threads waiting for results. In the case where the runtime for each m is approximately equal, optimizing the latency objective is trivial as the SAT problems can be solved in any order. However, as shown in Figure 2, several SAT instances can experience a wide variance in runtime. In particular, by analyzing the distribution of runtimes from the SAT 2003 competition [12] which contains several benchmark suites, we observe that most instances either finish in the first 5 minutes or timeout over 64 minutes. An optimal schedule for an N-threaded machine involves scheduling problems in increasing order of complexity on each n thread. Unfortunately, predicting actual runtimes beforehand is not possible; therefore, we will discuss other predictative strategies for handling this limitation.

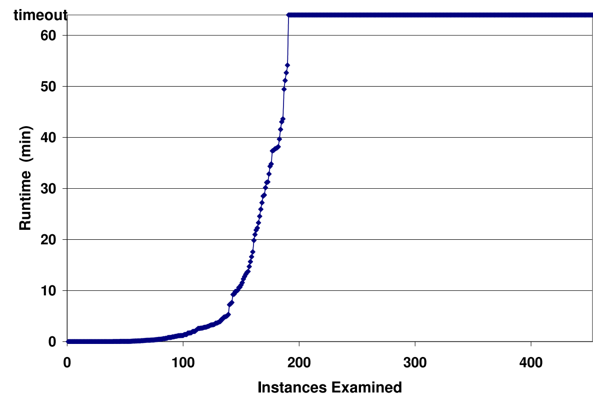

Figure 2: The number of SAT instance solved in a given amount of time. The timeout is 64 minutes.

Because the distribution of runtimes is very uneven, it is possible that in addition to large accumulated latency, random schedul-

ing could result in some threads completing execution well after previous ones, leading to poor resource utilization. To even out execution, we can leverage thread schedulers present in most operation systems, which perform time-slicing. Through time-slicing, problems with small runtimes will still finish pretty early; however, longer instances will be evened out and finish around the same time as each other, thus increasing accumulated latency.

Our solution involves using the distribution of SAT runtimes to predict a time threshold where the remaining percentage of SAT problems unsolved will likely be of high complexity. However, we will explore other techniques which are not dependent on predictive distributions. From Figure 2, we see that this value is approximately 5 minutes. Before this threshold, we perform time-sliced scheduling over all the problems and after the threshold we increase the thread priority for N instances so that they run in batch mode.

To further reduce the average latency, we can ensure that jobs requiring large memory resources that negatively impact system performance have low priority. Despite the increasing amount of shared memory, large SAT instances could cause a bottleneck due to memory contention. To counteract this, we can assign low priority to the largest X jobs so that the available system memory is greater than $mem(M - X)$ or the remaining required memory. During the time-slice phase of the scheduler, we can temporarily lower the priority of different sets of Y jobs so that each large job from X can be analyzed for a segment of time. In such a way, we ensure that each job receives resources, that contention is minimized, and that the largest number of jobs have the chance of finishing early. If large jobs remain after the time-slicing mode finishes, we can divide the remaining jobs so that batches can execute with total memory consumption within system resources. However, in our experiments, the instances we consider have low memory profiles in general. Also, the growth of a SAT instance due to conflict-driven learning tends to be gradual with respect to the initial size because of efficient memory management found in state-of-the-art SAT solvers.

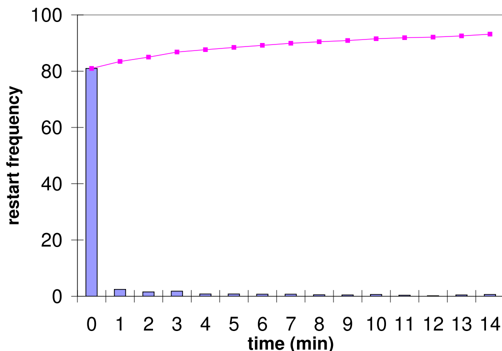

Figure 3: The percentage of total restarts for each minute of execution for a random distribution of SAT instances.

Although not implemented in this paper, scheduling can be based on runtime estimates generated from progress meters found in some SAT solvers [2]. The thread priority for easier instances can be increased in this manner. As a simpler predictive model, we consider random restart frequency or the percentage of restarts done each minute. In Figure 3, we show a distribution of restarts over a random slice of benchmarks. It reveals an exponential decay in frequency, which can be used as a guide to lower thread priority. When few restarts occur, there are less opportunities to quickly arrive at a solution due to a better variable order.

As another extension, we consider the case where jobs occur on the fly. In this situation, new jobs will be given priorities equal to those of the current highest priority jobs (whether in time-slice or batch mode) so they will enter time-slice mode. This ensures that old jobs receive resources while maximizing resources for new jobs, which often complete before the time threshold.

4. CURRENT PARALLEL SAT SOLVERS

Previous efforts at parallelizing algorithms for solving *random* SAT instances have been effective as in [15], but random instances are not common in EDA applications whose problems exhibit structure. For such instances, [14] represents the state of the art where a novel implementation was proposed to exploit shared memory to enable efficient learning between solvers running on different threads. In this section, we describe some pitfalls with this approach, along with some limitations of portfolio solvers.

As previously mentioned, search space partitioning using guiding paths, as in [14], is limited because the division may be unbalanced. This division circumvents the effectiveness of random restarts by prescribing initial assignments to each solver running in parallel. However, addressing this problem by undoing the initial assignments for a thread after each random restart appears to undermine the benefits of partitioning. The partition itself may also produce subproblems that require very different runtimes to search. Furthermore, learning between threads is not always an effective means of boosting performance. As discussed in [26], using *1-UIP* learnts is often more effective at improving the solver's performance than using minimally-sized learnts. This counter-intuitive result suggests that parallel schemes for learning, which often use the size of learnts as a filtering mechanism, may not necessary boost the performance of a particular thread of execution.

Implementing these parallelization strategies requires careful selection of a successful sequential solver. Choosing a poor heuristic for parallelization will still lead to poor performance, especially in a portfolio where it consistently underperforms compared to other heuristics. Furthermore, the underlying heuristics implemented in most successful SAT solvers are finely-tuned which requires careful and time-consuming development of parallel optimizations. The slightest perturbation to the quality of the sequential algorithm caused by parallelization, such as excessive learning between threads, can significantly degrade runtime performance. For example, learning increases the size of the clause database which increases the cost of Boolean constraint propagation. Furthermore, decision heuristics, like VSIDS, are guided by the learning that is performed. Learning can therefore steer the decision heuristic to complicated parts of the search space.

Portfolio solvers are advantageous because there is little implementation overhead, and the risk of performing really poorly on an instance that has high variable runtime is minimal. However, this approach requires that the heuristics have different performance characteristics on different types of instances. As larger computing systems become available, it will be increasingly difficult to find larger collections of different heuristics. Furthermore, even where orders-of-magnitude improvements are possible, some instances may show no improvement, resulting in small overall speed-up.

5. SOLVING INDIVIDUAL HARD INSTANCES IN PARALLEL

In this section, we propose an algorithmic methodology that utilizes available resources to reduce the runtime of hard instances. We overcome the limitations described previously by introducing a novel approach to dividing search-space that allows for more flexible random restarts. Furthermore, our approach can be easily adopted by any state-of-the-art DPLL-based solver.

5.1 Search-space Partitioning using XOR Constraints

We propose our strategy for partitioning the search space evenly. First we elaborate on the theoretical underpinnings of adding XOR constraints to a SAT instance and then reveal its significance for evenly dividing a search space.

Reducing search-space through XOR constraints. To more evenly partition the search space, we extend the work in solution-space reduction using a result described in [21]. It was explained that adding the following XOR constraint to F probabilistically reduced its solution space by approximately $\frac{1}{2}$:

$$F \wedge (x_1 \oplus x_2 \oplus \cdots \oplus x_j \oplus 1) \quad (2)$$

where x_j represent randomly chosen variables in F (the probability of choosing a variable is $\frac{1}{2}$). We define the resulting formula as F_{even} denoting that assignments to the x_j variables must have even polarity to satisfy F_{even}. We can define F_{odd} as:

$$F_{odd} = F \wedge (x_1 \oplus x_2 \oplus \cdots \oplus x_j \oplus 0) \quad (3)$$

$\{F_{even}, F_{odd}\}$ denotes a disjoint partition when the set of x_j variables is the same for F_{even} and F_{odd}. More formally:

DEFINITION 4. *A disjoint partition exists when (1) $F = F_{even} \vee F_{odd}$, (2) $F_{even} \wedge F_{odd} = 0$, and (3) x_j set is the same for F_{even} and F_{odd}.*

This partition generates two sub-problems that can be assigned to different solvers. These partitions can be recursively divided by adding more XOR constraints. As a generalization to the result explained in [21], each XOR constraint probabilistically divides the number of assignment combinations for V variables roughly in half to $2^{|V|-1}$. This means that the constraint probabilistically divides the search space in half with the of hope balancing the workload between different solvers.

However, in practice, simply adding large XOR-constraints is inadequate for reducing the search space because it will not cause a conflict until all of the x_j variables are assigned, which is approximately $\frac{|V|}{2}$ variables. In other words, this large constraint divides the search space evenly, but it is ineffective at restricting the search until after nearly all assignments have been made. We now explain how smaller XOR constraints can be derived that still achieve the same theoretical guarantees.

Connection between backdoors and randomized reductions. As an example of how we can add smaller constraints, consider a combinational circuit D with m inputs. This circuit can be converted to a SAT instance D^C with V variables where the set of solutions determined by the assignments to the primary inputs is M where $|M| = 2^m$. Therefore, the set of solutions $S_{DC} \in 2^{|V|} \leftrightarrow M$. In other words, any assignment $A_M \in 2^m$ results in exactly one solution. According to Definition 3, M is a strong backdoor for D^C. By restricting the set of variables x_j to variables in M, we can construct a partition that gives the same probabilistic guarantees as the original formulation, but produces a smaller XOR constraint while generating conflicts earlier if these variables are decided on first. Namely, 1-XOR constraint roughly divides the solution space relative to M which correspondingly divides S_{DC} roughly in half due to the bijective relationship.

5.2 Light-weight Parallel SAT

For a general SAT instance, we can restrict XOR-constraints to involve only backdoor variables, which typically restricts the variables to a much smaller subset. By adding an XOR constraint involving these backdoor variables B, we can cut the search space roughly in half to $2^{|B|-1}$.

Evenly partitioning the search space in our multi-threaded SAT framework. Because computing the smallest backdoor set explicitly is not always feasible, we use, as an approximation, highly ranked variables determined by selection heuristics in modern DPLL-based solvers like VSIDS. Since [22] observed that many backdoor sets have cardinality $\log(|V|)$, we choose x_j from the top $\log(|V|)$ variables when producing XOR constraints. To generate variable rankings, we run a SAT solver for a certain amount of time before generating these XOR constraints.

Algorithm. In Figure 4, we introduce our algorithm for using XOR partitioning to improve the performance of SAT in a parallel environment. `psat_solve` is called like a normal SAT solver and takes as a parameter the number of random restarts performed before partitioning the problem. This allows very simple instances to be completed sequentially and *trains* the SAT solver so that good variables are chosen for partitioning. When partitioning is required, we add an XOR constraint involving the top $lg(|N|)$ variables through `add_xor_constraints`. Because the XOR constraint is typically small, we don't require a specialized XOR constraint representation as in [9]. We then spawn two threads and wait for their results. Notice that the threaded mode uses the same infrastructure as the sequential mode with only a few minor changes. To maintain an even division of work between the two threads, we ensure that the partition variables `part_vars` are ranked high by increasing their rank after restarting. Because multiple variables are used to drive the partitioning constraint, there is more flexibility in the search procedure than having an exact guiding path. Finally, in the `DPLL_search` function, we share learnts between threads when conflicts occur in a manner similar to [14] to facilitate quick search-space pruning. We expect that our partitioning, however, will produce sub-problems with similar characteristics, thereby making our inter-thread learning more powerful. If one thread finishes unsatisfiable, we do not repartition the problem. We have observed that constant repartitioning hinders the effectiveness of the underlying sequential algorithm to solve instances. In practice, we observe that the even partitioning results in threads that compute for a similar amount of time.

Even Division of Solution Space. We can also exploit the theoretical evenness of our division and note that the number of solutions to the SAT instance are evenly distributed. Therefore, if one sub-problem is found to be unsatisfiable, we can estimate that the other sub-problems have none or very few solutions. This result could be used to guide the selection of a portfolio of solvers on-the-fly.

Parallel SAT Methodology. The previous algorithm for parallelizing SAT is an effective and lightweight strategy for improving the runtime of a sequential solver. Since the sequential solver could perform poorly on certain classes of benchmarks, we run our parallel solver in a portfolio to minimize this risk. Also, in accordance with our priority-based scheduling, we do not partition the instance until the batch-mode time threshold is reached. Similarly, we can partition the instance when the restart frequency is low. In this way, we reserve parallel computation for only the hard problems and avoid deterministically partitioning the search space when the variable rankings change more frequently. Therefore, because the top variables will be fairly consistent between random restarts after several minutes of sequential solving, we simplify our procedure in Figure 4 to not increase the ranking of variables chosen for the partitioning. Furthermore, we also can choose fewer partition variables so smaller XOR constraints are produced.

```
bool psat_solve(CNF cnf, int passes, Mode mod=seq, Lit assump){
  static Var part_vars;
  initialize_assumps(assumps);
  while( not_done() && (passes−− || mod!=seq)) {
    if(mod == parallel) increase_rank(part_vars);
    random_restart();
    result = DPLL_search(mod);
  }
  if(mod == seq && not_done(){
   part_vars = top_vars();
   add_xor_constraints(cnf, part_vars);
   thread(cnf,,parallel,neg);
   thread(cnf,,parallel,pos);
   while(wait){
   if(SAT) return SAT;
   else if(num_threads−−) return UNSAT;
  }
  return result;
}

bool search(Mode mod) {
  while (true) {
   propagate();
   if(conflict) {
    analyze_conflict();
    if(top_level_conflict) return UNSAT;
    backtrack();
    if(mod == parallel) parallel_learn_backtrack();
   }
   else if(satisfied) return SAT;
   else decide();
}}
```

Figure 4: Parallel SAT Algorithm.

6. EMPIRICAL VALIDATION

We consider benchmarks from the SAT 2003 Competition [12] from the `handmade` and `industrial` categories which feature several different suites. The runtime of each benchmark is profiled using MiniSAT 2 [7] on a 4 processor dual-core Opteron system clocked at 1GHz with 16 GB of memory running the Fedora 8 SMP OS. We set a timeout for each benchmark at 64 minutes and created a distribution of runtimes over the entire suite. Our results indicate that most benchmarks finish in either less than 1 minute or over 1 hour. This highlights the wide variance in runtime performance motivating our proposed methodology. Statistics for the benchmarks as well as the runtime distributions can be found in Table 1 and Figure 2 respectively.

SAT suite	#SAT	#UNSAT	#TimeOut $> 64min$	#total	time (min)
handmade	48	90	215	353	13779
industrial	19	33	48	100	3160

Table 1: MiniSAT 2 results on the SAT 2003 benchmark suite.

6.1 Effective Scheduling of SAT Instances

We first consider the upper-bound of resource utilization by executing several problems concurrently in the ideal case where each benchmark is roughly of the same complexity. Here, we consider only small benchmarks from the suite previously analyzed and we show how a multi-threaded machine can effectively be used so that n threads result in approximately n-times speed-up. The performance increase is ideal because each thread has an independent sub-problem thus being perfectly parallelizable. However, this analysis, shown in Table 2, is vital in showing that if n independent problems are derivable, a corresponding speed-up is possible. Perfect parallelization is not seen in this table due to the slight variation in runtime complexity for the different instances. In the following paragraphs, we show our results for solving a set instances with a potential wide variance in runtime.

#threads	runtime(min)	speed-up
1	67	1
2	34	1.98
4	18	3.72
8	10	6.70

Table 2: Running MiniSAT on a set of benchmarks using different numbers of threads.

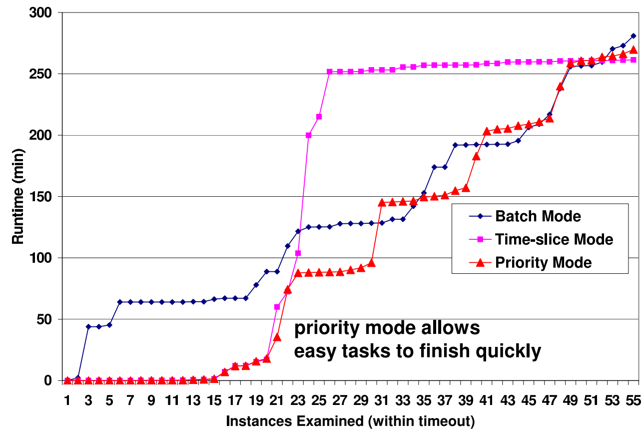

Figure 5: The number of SAT instances solved (up to the timeout) in a given amount of time by considering three different scheduling schemes for an 8-threaded machine. Our priority scheme gives the best average latency, which is 20% better than `batch mode` **and 29% better than** `time-slice mode`**.**

Scheduling SAT problems with varying complexity. To test a parallel methodology under a realistic distribution of runtimes, we randomly select a subset of benchmarks, with total runtime of $\sim$ 32 hours, that follows the distribution seen in Figure 2. We show the performance of a non-ideal methodology denoted by the line `batch mode` in Figure 5 that schedules the SAT problems as a batch of jobs to our 8-threaded machine. Although the total runtime for all the problems is around 4 hours, we note that several easily executed problems are not scheduled until much later. In particular, we notice that the latency for each job is not minimized especially for smaller instances. Using the operating-system to schedule threads results in the `time-slice mode` line. Notice that although several easy instances finish early, the latency for harder instances increase over batch mode. In our `priority mode`, we transition to batch-mode by adjusting thread priorities after a time threshold is reached. Notice that the area below this line is smaller, indicating better latency, according to our criterion. Here, we achieve a 20% improvement in average latency over `batch mode` and 29% improvement over `time-slice mode`. Figure 5 shows wall-clock time; however, we have observed that the system time is insignificant for each strategy ($<$ 2 minutes). This is due, in part, to the efficiency of the OS scheduler along with the relatively small memory profile required for the random slice of 55 instance considered.

6.2 Solving Individual Hard Problems

Ultimately, fast verification turn-around may require faster solution of individual hard SAT instances. Solvers such as SatZilla [23] try to exploit the fact that some solvers perform better on certain classes of SAT problems than others. By carefully assigning different solvers to each instance, one can improve runtime compared to using any one solver. In the parallel setting, the choice can be simplified by running until one of them completes. However, unlike the single-threaded portfolio variant, it is desirable that the improved runtime is comparable to the extra computing

resources that are used. Although super-linear runtime improvement over the baseline of MiniSAT is possible due to the high variability of performance of different approaches on a given problem instance, it is important that consistent improvements are achieved that actually exploit available computational resources. In our following analysis, we consider SAT instances from the `handmade` and `industrial` categories to more realistically reflect the types of instances seen in practice. We choose a subset of instances where MiniSAT requires significant computation (~ 1 hr).

heuristic type & num. instances heuristic solves first							
Solver Portfolio		MiniSAT Vers.		w/MiraXT		w/pMiniSAT	
MiniSAT	6	m1	3	MiniSAT	6	**pMiniSAT**	5
Mira1T	0	m2	2	MiraXT	1	Mira1T	1
HaifaSat	1	m3	1	-	-	-	-
Jerusat1.3	1	m4	1	Jerusat1.3	0	Jerusat1.3	1
march_ks	0	m5	1	march_ks	0	march_ks	0
picosat	2	m6	2	picosat	2	picosat	2
rsat	0	m7	1	rsat	2	rsat	1
zchaff	2	m8	1	zchaff	2	zchaff	2
time(min)	321		326		335		**200**
speed-up	1.67		1.65		1.60		**2.69**
%util	20.9		20.6		20.0		**33.6**

Table 3: Using 8 threads of computation with a portfolio of solvers to handle hard SAT instances.

heuristic & # solved			
MiniSAT	7	**pMiniSat**	8
picosat	2	picosat	2
zchaff	2	zchaff	2
Jerusat1.3	1	-	-
time(min)	359		**218**
speed-up	1.50		**2.46**
%util	37.4		**61.6**

Table 4: Using 4 threads of computation with a portfolio of solvers to handle hard SAT instances.

Table 3 shows the speed-up achieved by running multiple heuristics simultaneously where we consider different solver portfolios. In the last two columns, we highlight the improvement to CPU utilization achieved by incorporating our lightweight parallel algorithm. The total runtime without parallelization for MiniSAT is 537 min. The first, third, fifth, and seventh columns list different heuristics organized in a portfolio. We report the number of hard instances that a particular heuristic solves the fastest. The first column shows a collection of state-of-the-art SAT solvers. Notice that even though MiniSAT is not the best in all cases, the speed-up on 8 cores is pretty small at 1.7 meaning that only 20.9% of the 8-times ideal speed-up is realized. The third column shows a portfolio of different versions of MiniSAT given by `MiniSAT Vers.` produced by adjusting several tunable knobs such as: restart frequency, variable decay rate, and decision heuristic. These results reveal similarly poor utilization where neither randomness nor different heuristics achieve high utilization. We then tried running MiraXT [14] with two threads but did not see additional speed-up in the portfolio (one heuristic is removed from the original portfolio to account for the extra thread required by MiraXT) . Because its performance was dominated by MiniSAT, parallelizing this solver is ineffective at increasing utilization. Furthermore, the results reported in [14] consider only two threads with speed-up much smaller than 2. Additionally, we have observed that their heavyweight approach to partitioning and learning has diminishing returns when considering more threads.

By incorporating our parallel version of MiniSAT, `pMiniSAT`, described in Figure 4 in the solver portfolio, we are able to achieve significant speed-up and higher utilization of 60.5% with respect to the 8 threads of execution compared to the best solver portfolio. Furthermore, in Table 4 we show that our utilization numbers are better when considering only 4 threads. This indicates the limitation of large solver portfolios, illustrating that our lightweight approach to parallelization can be beneficial for achieving greater utilization by applying it across multiple, different heuristics.

6.3 Partitioning Strategies

We compared our XOR-based partitioning to partitioning with a single guiding variable, which is a special case of guiding paths. In Figure 6, we reveal the effectiveness of using XOR constraints for achieving balanced workloads which evenly distributes solutions between the threads. Figure 6a gives the percentage of satisfiable problem instances, out of 16 instances examined, where the first thread that completes has at least one solution. We compare the single variable partitioning with XOR constraints of size $2-4$ and consider parallelization using 2, 4, and 8 threads concurrently. Notice, in the 2-thread case, 100% of the threads that finish first is satisfiable using XORs of size 4, compared to only 75% using one variable. In general. this experiment reveals that our partitioning is more effective at distributing solutions, which can be exploited in adjusting the portfolio of heuristics considered dynamically. We expect even better performance in application domains where the number of solutions is much greater than the number of available threads of computation.

Figure 6b shows the runtime balance between 2, 4, and 8 threads. We examined different partitioning strategies on a set of 29 unsatisfiable problem instances and calculated the normalized standard deviation, which is the standard deviation of thread runtime divided by the average runtime. We disable learning for this experiment to more accurately analyze how the search space is partitioned. For the single variable partitioning for two threads, the normalized standard deviation is 0.35, compared to a much smaller 0.22 for XOR-based partitioning with 4 variables. In general, we notice close to a 2-time improvement in the runtime deviation between the single variable strategy and the 4 variable XOR when considering different numbers of threads.

7. CONCLUSIONS

The ubiquity of SAT-solving in EDA and its growing adoption for new applications necessitates methodologies that can exploit parallel resources. However, the inherent variability and complexity of SAT instances challenge state-of-the-art strategies to efficiently utilize these resources.

In this work we have empirically evaluated portfolios of SAT solvers – perhaps the most promising approach so far for parallel SAT on up to four cores. The results hint that portfolios are unlikely to scale to large numbers of processors because their performance on each particular benchmark type tends to be dominated by only one or two solvers, rendering the remaining solvers essentially useless. Indeed, when 100 cores are available, it is unlikely that there will be 100 very different but equally competitive SAT solvers available.

To better address workload balance in parallel SAT, we proposed a two-part strategy for utilizing parallel processing more effectively. First, we introduced a scheduling algorithm that incorporates previous runtime distributions over a set of SAT instances to minimize average latency over batch scheduling by 20%. Since several instances require prohibitive amounts of runtime, we proposed a lightweight parallel SAT algorithm that effectively partitions the search space after initially running the solver in sequential mode. We observe that our partitioning results in $\sim 50\%$ better run-time balance than simply choosing one splitting variable. Our partition-

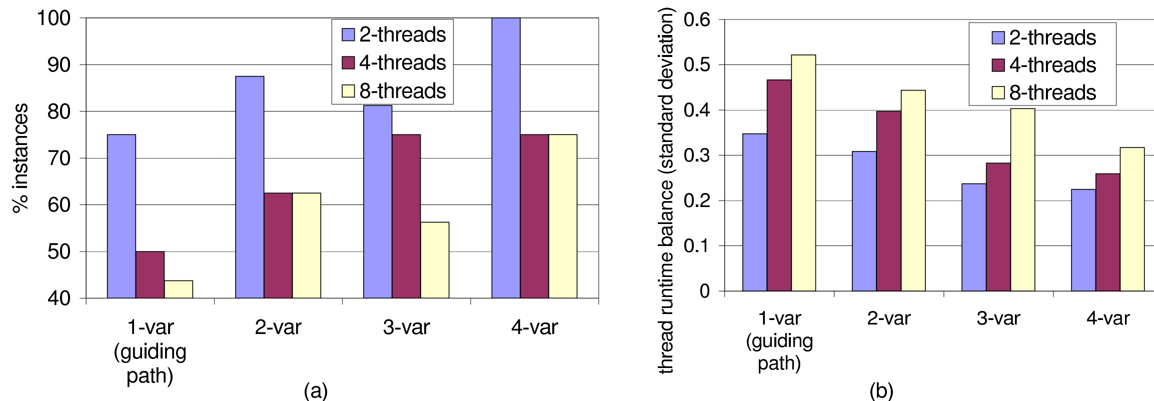

Figure 6: a) The percentage of satisfiable instances where the first thread that completes finds a satisfying assignment. b) The standard deviation of runtime between threads. Using XOR constraints as opposed to splitting one variable can significantly improve load balance and more evenly distribute solutions among threads.

ing strategy enables us to improve resource utilization over solver portfolios by 60.5%. By incorporating our parallel solving strategies on several different SAT solvers, solver portfolios can be further improved because the randomness vital for efficiently solving SAT instances is better coordinated.

In addition to sheer necessity for parallel SAT, successful techniques opens a new avenue in the design of future parallel EDA tools. Rather than custom-developing parallel techniques for various existing optimizations, one can look for reductions to (parallel) SAT. Even in the cases where reductions to SAT were not justified on a single processor, a highly-optimized generic parallel-SAT library may grow increasingly competitive as the number of cores on a chip increases beyond several dozen.

8. REFERENCES

[1] M. Abramovici, J. DeSousa, and D. Saab, "A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware", *DAC*, pp. 684-690, 1999.

[2] F. Aloul, B. Sierawski, and K. Sakallah, "Satometer: how much have we searched?", *TCAD*, pp. 995-1004, 2003.

[3] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick "The landscape of parallel computing research: a view from Berkeley", *ERL Technical Report*, Berkeley.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. "Symbolic model checking without BDDs", *TACAS*, pp. 193-207, 1999.

[5] W. Chrabakh and R. Wolski, "GraDSAT: a parallel SAT solver for the grid", *UCSB Comp. Sci. TR*, 2003.

[6] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving", *Comm. of ACM*, pp. 394-397, 1962.

[7] N. Een and N. Sorensson, "An extensible SAT-solver", *SAT '03*, (http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/).

[8] E. Goldberg, M. Prasad, and R. Brayton, "Using SAT for combinational equivalence checking", *DATE*, pp. 114-121, 2001.

[9] C. Gomes, W. Hoeve, A. Sabharwal, and B. Selman, "Counting CSP solutions using generalized XOR constraints", *AAAI*, pp. 204-209, 2007.

[10] C. Gomes and B. Selman, "Algorithm portfolios, *AI*, pp. 43-62, 2001.

[11] C. Gomes, B. Selman, K. McAloon, and C. Tretkoff, "Randomization in backtrack search: exploiting heavy-tailed profiles for solving hard scheduling problems", *AIPS*, 1998.

[12] H. Hoos and T. Sttzl, "SALIB: an online resource for research on SAT", *SAT*, pp. 283-292, 2000.

[13] Y. Hu, V. Shih, R. Majumdar, and L. He, "Exploiting symmetry in SAT-based Boolean matching for heterogeneous FPGA technology mapping", *ICCAD*, pp. 350-353, 2007.

[14] M. Lewis, T. Schubert, and B. Becker, "Multithreaded SAT solving", *ASP-DAC*, pp. 926-932, 2007.

[15] P. Manolios and Y. Zhang, "Implementing survey propogation on graphics processing units," *SAT*, pp. 311-324, 2006.

[16] J. Marques-Silva and K. Sakallah, "GRASP: A search algorithm for propositional satisfiability", *IEEE Trans. Comp*, pp. 506-521, 1999.

[17] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver", *DAC*, pp. 530-535, 2001.

[18] F. Okushi, "Parallel cooperative propositional theorem proving", *Annals of Mathematics and AI*, pp. 59-85, 1999.

[19] S. Plaza, K.-H Chang, I. Markov, and V. Bertacco, "Node mergers in the presence of don't cares", *ASP-DAC '06*, pp. 414-419.

[20] S. Plaza, I. Markov, and V. Bertacco, "Optimizing non-monotonic interconnect using functional simulation and logic restructuring", *to appear in ISPD'08*.

[21] L. Valiant and V. Vazirani. "NP is as easy as detecting unique solutions", *Theor. Comput. Sci.*, pp. 85-93, 1986.

[22] R. Williams, C. Gomes, B. Selman, "Backdoors to typical case complexity", *IJCAI*, 2003.

[23] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown, "SATzilla-07: the design and analysis of an algorithm portfolio for SAT", *CP*, 2007.

[24] H. Zhang, M.P. Bonacina, and J. Hsiang, "PSATO: a distributed propositional prover and its application to quasigroup problems", *Jrnl of Symb Comp*, pp. 1-18, 1996.

[25] H. Zhang, "SATO: an efficient propositional prover", *CADE*, pp. 272-275, 1997.

[26] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, "Efficient conflict driven learning in Boolean satisfiability", *ICCAD*, pp. 279-285, 2001.

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

[28] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Using configurable computing to accelerate Boolean satisfiability", *TCAD*, pp. 861-868, 1999.

[29] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT sweeping with local observability don't cares", *DAC '06*, pp. 229-234.