Combining Component Caching and Clause Learning for Effective Model Counting

Tian Sang[1], Fahiem Bacchus[2], Paul Beame[1], Henry Kautz[1], and Toniann Pitassi[2]

[1] Computer Science and Engineering, University of Washington, Seattle WA 98195-2350
{sang,beame,kautz}@cs.washington.edu
[2] Dept. Computer Science, University of Toronto, Toronto ON M5S 1A4
{fbacchus,toni}@cs.utoronto.ca

1 Introduction

While there has been very substantial progress in practical algorithms for satisfiability, there are many related logical problems where satisfiability alone is not enough. One particularly useful extension to satisfiability is the associated counting problem, #SAT, which requires computing the number of assignments that satisfy the input formula. #SAT's practical importance stems in part from its very close relationship to the problem of general Bayesian inference.

#SAT seems to be more computationally difficult than SAT since an algorithm for SAT can stop once it has found a single satisfying assignment, whereas #SAT requires finding all such assignments. In fact, #SAT is complete for the class #P which is at least as hard as the polynomial-time hierarchy [10].

Not only is #SAT intrinsically important, it is also an excellent test-bed for algorithmic ideas in propositional reasoning. One of these new ideas is *formula caching* [7, 1, 5] which seems particularly promising when performed in the form called *component caching* [1, 2]. In component caching, disjoint components of the formula, generated dynamically during a DPLL search, are cached so that they only have to be solved once. While formula caching in general may have theoretical value even in SAT solvers [5], component caching seems to hold great promise for the practical improvement of #SAT algorithms (and Bayes inference) where there is more of a chance to reuse cached results. In particular, Bacchus, Dalmao, and Pitassi [1] discuss three different caching schemes: simple caching, component caching, and linear-space caching and show that component caching is theoretically competitive with the best of current methods for Bayesian inference (and substantially better in some instances).

It has not been clear, however, whether component caching can be as competitive in practice as it is theoretically. We provide significant evidence that it can, demonstrating that on many instances it can outperform existing algorithms for #SAT by orders of magnitude. The key to this success is carefully incorporating component caching with *clause learning*, one of the most important ideas used in modern SAT solvers. Although both component caching and clause learning involve recording information collected during search, the nature and use of the recorded information is radically different. In clause learning, a clause that captures the reason for failure is computed from every failed search path. Component caching, on the other hand, stores the result computed when solving a subproblem. When that subproblem is encountered again its value can be retrieved from the cache rather than having to solve it again. It is not immediately obvious how to maintain correctness as well as obtain the best performance from a combination of these techniques. In this paper we show how this combination can be achieved so as to obtain the performance improvements just mentioned.

Our model-counting program is built on the ZChaff SAT solver [8, 11]. ZChaff already implements clause learning, and we have added new modules and modified many others to support #SAT and to integrate component caching with clause learning. Ours is the first implementation we are aware of that is able to benefit from both component caching and clause learning. We have tested our program against the `relsat` [4, 3] system, which also performs component analysis, but does not cache the computed values of these components. In most instances of both random and structured problems our new solver is significantly faster than `relsat`, often by up to several orders of magnitude.[3]

We begin by reviewing DPLL with caching for #SAT [1], and DPLL with learning for SAT. We then outline a basic approach for efficiently integrating component caching and clause learning. With this basic

[3] An alternative approach to #SAT was recently reported by Darwiche [6]. We have not as yet been able to test against his approach.

Table 1 #DPLL Algorithm with component caching

```
#DPLLCache(Φ)
  Φ = RemoveCachedComponents(Φ)
  if Φ = {}, return
  else
    Pick a variable v in some component φ ∈ Φ
    Φ⁻ = ToComponents(φ|_{v=0})
    #DPLLCache((Φ − {φ}) ∪ Φ⁻)
    Φ⁺ = ToComponents(φ|_{v=1})
    #DPLLCache((Φ − {φ}) ∪ Φ⁺)
    AddToCache(φ, GetValue(Φ⁻) × 1/2 + GetValue(Φ⁺) × 1/2)
    // RemoveFromCache(Φ⁻ ∪ Φ⁺) // this line valid is ONLY for linear space
    return
```

Table 2 DPLL Algorithm with learning

```
while(1)
  if (decide_next_branch( ))          // Branching
    while(deduce( )==conflict)        // Unit Propagation
      blevel = analyze_conflicts( );  // Clause Learning
      if (blevel == 0)
        return UNSATISFIABLE;
      else back_track(blevel);        // Backtracking
  else return SATISFIABLE;            // no branch means all variables were assigned.
```

approach, linear-space caching works correctly when combined with clause learning, but experimentally has poor performance. (Simple caching also works but has even worse performance; we do not discuss it further.) On the other hand, component caching with clause learning has good performance but we show, somewhat surprisingly, that it will only give a lower bound rather than an exact count of the number of models. We then show how to refine this basic approach with *sibling pruning*. This allows component caching to work properly without significant additional overhead. The key idea is to prevent the bad interaction between cached components and learned clauses from spreading.

We mention other important implementation issues for component caching such as the use of hash tables and a method, based on stale dating of cached components, that keeps space small but is more efficient than linear-space caching. Finally, we show the results of a set of experiments on both random and structured problems under various combinations of caching and clause learning.

2 Background

#DPLL with Caching: A component of a CNF formula F is a set of clauses ϕ the variables of which are disjoint from the variables in the remaining clauses $F - \phi$. Table 1 shows the #DPLL algorithm for solving #SAT using component caching from [1]. #DPLL takes as input a set of components, each of which shares no variables with any of the other components and whose union of represents the current residual formula (the input formula reduced by the currently assigned variables). The algorithm terminates when the number of satisfying assignments for each component has been computed and stored in the cache.

It first removes all components already in the cache and then instantiates a variable v from one of the remaining components ϕ. The formula Φ^- is obtained by setting $v = 0$ in ϕ, and then breaking the resulting formula up into components (if possible), Φ^+ is defined likewise. The algorithm then recursively solves the original set Φ of components, but with ϕ replaced by Φ^-, and then again with ϕ replaced by Φ^+. Upon return from both recursions, the cache contains the value of all components in Φ^+ and Φ^-, and these values can be combined to obtain the value of the original component ϕ.

The values computed for the components ϕ are maintained as the satisfying probability of ϕ, $\Pr(\phi)$, under a uniformly chosen assignment. The number of satisfying assignments of ϕ on n variables is thus $\#(\phi) = 2^n \cdot \Pr(\phi)$.

Since Φ consists of disjoint component(s), $\Pr(\Phi) = \prod_{\phi \in \Phi} \Pr(\phi)$. Furthermore, during the computation, if a component reappears, duplicate computation is avoided by extracting its value from the cache. These two properties are the keys to the efficiency of this method.

The number of cached components can become extremely large so [1] presents a variant of this algorithm that uses only linear space. The only difference is to add one more line, RemoveFromCache($\Phi^- \cup \Phi^+$), shown in Table 1, which removes all cached values of child components once their parent component's value has been computed.

In [1], the component caching algorithm was proved to have a worst case time complexity of $n^{O(1)}2^{O(w)}$, where n is is the number of variables and w is the underlying branch-width of the instance. The bound shown for the linear-space version was somewhat larger: worst-case time complexity of $2^{O(w \log n)}$ and space complexity $O(n)$.

DPLL with Learning: Our solver is based on the DPLL SAT solver ZChaff which performs clause learning. ZChaff's main control loop is shown in Table 2. This loop expresses DPPL iteratively, and uses the learned clauses rather than explicit returns to guide the backtracking. The procedure is to first choose a branch to descend (a literal to make true), after which unit propagation is performed (deduce). If an empty clause is generated a (conflict) clause explaining that conflict is computed (analyze_conflict), and added to the current set of clauses. From the way the conflict clause is constructed it must be falsified by the current variable assignments, and we can backtrack to a level where enough of the variable assignments have been retracted so that it is no longer falsified. The loop terminates when a solution is found or when the conflict cannot be unfalsified (which forces a backtrack to level 0).

3 Integrating Component Caching and Learning

Bounded Component Analysis In component caching, components are defined relative to the *residual formula* $\Phi = F|_\pi$ where F is the original formula and π is the current partial assignment. Component analysis is performed by detecting components within the residual formula (which, as in ZChaff, is maintained only implicitly).

A key to integrating clause learning with component caching is to notice that clause learning *deduces* new clauses; i.e., all of the new clauses are entailed by the original formula. Hence, if F is the original formula, and G is any set of learned clauses, then an assignment satisfies $F \wedge G$ iff it satisfies F. Furthermore, this one-to-one correspondence between satisfying assignments is preserved under partial assignments. That is, if π is a partial assignment to the variables of F, then the satisfying assignments of $F|_\pi$ are identical to the satisfying assignments of $(F \wedge G)|_\pi$ (note that $(F \wedge G)|_\pi = F|_\pi \wedge G|_\pi$).

This observation provides the basic intuition that to perform component caching it should be sufficient to examine only the formula F, ignoring the learned clauses G. We call this approach *bounded component analysis*. Bounded component analysis is in fact critical to the success of component caching in the presence of clause learning for a number of reasons. First, for a typical formula F, the set of learned clauses G can be orders of magnitude larger than F. Hence, component analysis on $F \wedge G$ would require significantly more overhead. Second, the residual learned clauses in $G|_\pi$ will often span the components of $F|_\pi$. Hence, component analysis on $F \wedge G$ would reduce the savings achievable from decomposition. Finally, the set of learned clauses grows monotonically throughout the search; so, the clauses which lie in a cached component at one stage may very well have been augmented by additional learned clauses the next time that component would have been encountered. Hence, component analysis on $F \wedge G$ would significantly reduce the chance of reusing cached information.

Although the learned clauses G are ignored when detecting and caching dynamically generated components, these clauses are used in unit propagations to prune the search tree. This means that in the subtree below a partial assignment π the search will only encounter satisfying assignments of $F|_\pi \wedge G|_\pi$. As pointed out above, there is no intrinsic problem with this, since every satisfying assignment of $F|_\pi$ also satisfies $G|_\pi$. The difficulty arises, however, from the values that might be computed for components of $F|_\pi$. In particular, if A is a component of $F|_\pi$, the search below π will only encounter satisfying assignments to the variables of A that *do not falsify* $F|_\pi \wedge G|_\pi$. This might not include all satisfying assignments of A!

Lemma 1. *There is a formula $F = A \wedge B$ and clause C such that $F \Rightarrow C$ (and thus C is a potential clause learned from input F) and a partial assignment π such that*
(i) $F|_\pi$ splits into disjoint components $A|_\pi$ and $B|_\pi$,
(ii) $C|_\pi$ is defined entirely on the variables of $A|_\pi$, and
(iii) $\Pr(A|_\pi) \neq \Pr(A|_\pi \wedge C|_\pi)$.

Proof. Let A be the formula $(p_0 \vee \overline{a_1} \vee p_1)(p_0 \vee \overline{p_2} \vee a_2)(a_1 \vee a_2 \vee a_3)$ and let B be $(\overline{p_1} \vee b_1)(\overline{b_1} \vee b_2)(\overline{b_2} \vee p_2)$. It is not hard to check that $C = (p_0 \vee \overline{a_1} \vee a_2)$ is a consequence of $F = A \wedge B$. Let π be $\{p_0 \leftarrow 0,\ p_1 \leftarrow 1,\ p_2 \leftarrow 0\}$.

Observe that $A|_\pi = (a_1 \vee a_2 \vee a_3)$ and $B|_\pi = (b_1)(\overline{b_1} \vee b_2)(\overline{b_2})$ are disjoint and the learned clause C becomes $C|_\pi = (\overline{a_1} \vee a_2)$ which is entirely defined on the variables of $A|_\pi$. One can easily check that $\Pr(A|_\pi \wedge C|_\pi) = 5/8 < 7/8 = \Pr(A|_\pi)$. ■

Examining this example, we see that if clause learning was to discover C, then the number of different satisfying assignments to $A|_\pi$ the search below π would encounter would only be 5 not 7: two of these satisfying assignments would be pruned because they falsify C. The correct value for the whole residual formula $(A \wedge B)|_\pi$ will be computed, however, because $B|_\pi$ is UNSAT and thus the value overall will be zero. Nevertheless, unless we are careful the incorrect value of $\Pr(A|_\pi)$ could be placed in the cache where it might then corrupt other values computed in the rest of the search. For example, if $A|_\pi$ reappears as a component of another residual formula which happens to be satisfiable, then the value computed for that formula will be corrupted by the incorrect cached value of $A|_\pi$. Although our example did not demonstrate that the clause C would actually be learned, in our experiments we have in fact encountered incorrect cached values arising from this situation.

The fact that $B|_\pi$ is UNSAT in this example is not an accident. The problem of under-counting the satisfying assignments of components of $F|_\pi$ cannot occur if $F|_\pi$ is satisfiable (all of its components must then also be satisfiable).

Theorem 1. *Let π be a partial assignment such that $F|_\pi$ is satisfiable, and let A be a component of $F|_\pi$, and $G|_\pi$ be the set of learned clauses G reduced by π. Then any assignment to the variables of A that satisfies A can be extended to a satisfying assignment of $F|_\pi \wedge G|_\pi$.*

Proof. Let $\langle \rho_A, \rho_{\bar{A}} \rangle$ be a satisfying assignment to $F|_\pi$, where ρ_A is an assignment to the variables in A and $\rho_{\bar{A}}$ is an assignment to the variables outside of A. Let ρ'_A be any assignment to the variables of A that satisfies A. Then $\langle \rho'_A, \rho_{\bar{A}} \rangle$ must be a satisfying assignment of $F|_\pi$, since A is disjoint from the rest of $F|_\pi$. Furthermore $\langle \rho'_A, \rho_{\bar{A}} \rangle$ must also satisfy $F|_\pi \wedge G|_\pi$ since $G|_\pi$ is entailed by $F|_\pi$. Hence $\rho_{\bar{A}}$ is the required extension of ρ'_A.

This theorem means for every satisfying assignment ρ_A of A there must exist at least one path visiting ρ_A that does not falsify the current formula $F|_\pi \wedge G|_\pi$. Hence, in a satisfiable subtree if our algorithm correctly counts up the number of distinct satisfying assignments to A in the subtree, it will compute the correct value for A even if the learned clauses G are being used to prune the search. The only case we must be careful of is in an unsatisfiable subtree. In this case the value computed for the entire residual formula $F|_\pi$, zero, will still be correct, but we cannot necessarily rely on the value of components computed in the unsatisfiable subtree.

Our algorithm puts these ideas together. It performs component analysis along with clause learning, exploring the subtree below a component in order to compute its value. The learned clauses serve to prune the subtree and thus make exploring it more efficient. The computed values are cached and used again to avoid recomputing already known values. Thus clause learning and component caching work together to improve efficiency. The main subtlety of the algorithm is that when computing the value of a component in a subtree, it applies its decomposition scheme recursively, much like the #DPLL algorithm presented in Table 1. That is, component values are computed by further breaking up the components into smaller components.

The Basic Algorithm for #DPLL with Caching and Learning We present our algorithm #DPLL with caching and learning in Table 3. For simplicity of presentation assume that the input has no unit clauses and contains only one component. (Our implementation is not restricted to this case.) The algorithm starts with the input component on the branchable_component_stack. At each iteration it pops a component ψ from the stack, chooses a literal ℓ from the component and branches on that literal. Its aim is to compute $\Pr(\psi)$ by first computing $\Pr(\psi|_\ell)$ then $\Pr(\psi|_{\bar{\ell}})$ and summing these values to obtain $\Pr(\psi)$. If all components have been solved, then we backtrack to the nearest unflipped decision literal and flip that literal. This can generate a new set of components to solve.

After ℓ is instantiated unit propagation is performed (deduce), and as in ZChaff if a conflict is detected a clause is learned and we backtrack to a level where the clause is no longer falsified. If ZChaff backtracks

Table 3 #DPLL Algorithm with caching and learning

```
while(1)
  if (!branchable_component_stack.empty( ))
    ψ = branchable_component_stack.pop( );
    choose a literal of ψ as the decision                   // Branching
  else back_track to proper level, or return if back_level == 0;
  while(deduce( )==conflict)                                // Unit Propagation
    analyze_conflicts( );                                   // Learning
    percolate_up(last_branched_component, 0);
    back_track to proper level, or return if back_level == 0;   // Backtracking
  num_new_component = extract_new_component( );             // Detecting components
  if (num_new_component == 0)
    percolate_up(last_branched_component, 1);               // Percolating and caching
  else for each newly generated component φ
    if (in_cache(φ))                                        // Checking if in cache
      sat_prob = get_cached_value(φ));
      percolate_up(φ, sat_prob);
      if (sat_prob == 0)
        back_track to proper level, or return if back_level == 0;
    else branchable_component_stack.push(φ);
```

Table 4 Routine remove_siblings

```
if component φ has value 0
  remove all its cached siblings and their descendants
  if φ is the last branched component of its parent
    add_to_cache(φ, 0)
  else remove all the cached descendants of φ
```

to a node whose left hand branch is already closed, it can resolve the clause labeling the left hand branch with the newly learned clause to backtrack even further. This is not always possible in #SAT, since the left hand branch need not have been UNSAT.

After $\psi|_\ell$ has been reduced by unit prop, it is broken up into components by extract_new_component. If there are no components, i.e., $\psi|_\ell$ has become empty, then this means that its satisfying probability is 1. This is recorded and percolated up by the percolate_up routine as part of the value of $\Pr(\psi)$. Otherwise each new component is pushed onto the branchable_component_stack after removing and percolating up the value of all components already in the cache.

If one of the new components has value zero we know that $\psi|_\pi$ is UNSAT and we can backtrack. As values of components are percolated up, the values of parent components eventually become known and can be added to the cache. This allows an easy implementation of linear space caching: we simply remove all children components from the cache once the parent's value has been completed. When the algorithm returns, the original formula with its satisfying probability is in cache.

To facilitate immediate backtracking upon the creation of zero-valued components we cache zero-valued components. It should be noted that caching zero-valued components is not the same as learning conflict clauses. Learning conflict clauses is equivalent to caching partial assignments (the partial assignment that falsifies the clause) that reduce the input formula to an UNSAT formula. The reduced UNSAT formula might in fact be UNSAT because it contains a particular UNSAT component. This UNSAT component might reappear in the theory under different partial assignments. Hence caching UNSAT components can detect some deadends that would be undetected by the learned clauses. On the other hand, the learned clauses can be resolved together to generate more powerful clauses. There is no easily implementable analogous way of combining UNSAT components into larger UNSAT components. Hence, the benefits of clause learning and caching of zero-valued components are orthogonal and it is useful to do both.

Correctness To use full component caching we must insure that the cache is never polluted. This can be accomplished by removing the value of all components that have UNSAT siblings using the following routine within percolate_up.

Theorem 2. *Using bounded component analysis, #DPLL with component caching and clause learning in Table 3, plus the routine remove_siblings, computes the correct satisfying probability.*

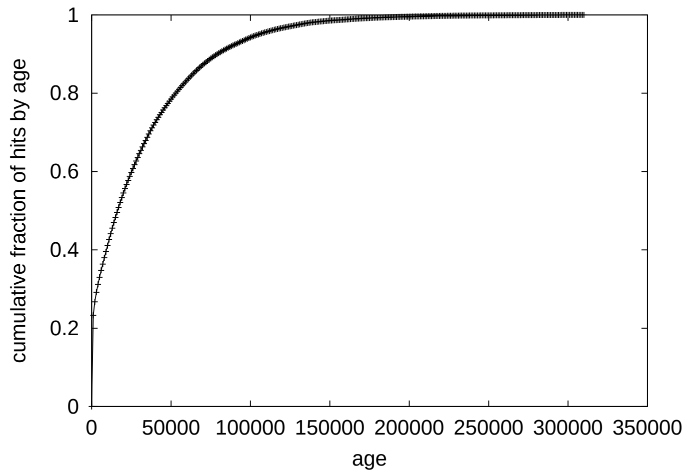

Fig. 1. Cumulative fraction of cache hits by cache age on 50-variable random 3-CNF formulas, clause/variable ratio=1.6, 100 instances

4 Implementation

We implement the dynamic component detection required by the algorithm using a simple depth-first search on the component ψ in which the decision literal is chosen; this is repeated with unit propagation. While it may seem that this is expensive, as seen by the results in the next section, the speed-ups that dynamic component detection provides are typically worth the effort. Code profiling on our examples shows that on hard random examples, roughly 38% of the total runtime is due to the cost of component detection, while on structured problems the cost of component detection varied from less than 10% to nearly 46% of the total run-time.

A component is represented as a set of unsatisfied clauses with falsified literals removed. Known components and their value are stored in a component cache implemented as a hash table with separate chaining. This table is checked when a new component is created to see if its value is already in the hash table.

With the number of components encountered during the execution of the algorithm, space complexity can become a serious bottleneck if entries in the cache are never flushed; this also holds despite the routine remove_siblings, because remove_siblings is not triggered when a residual formula is satisfiable. For example, to solve a 75 variable random 3-CNF formula with a hard clause-variable ratio (for #SAT this is near 1.8 rather than 4.2 because of the large number of satisfying assignments that need to be examined), we saw more than 9 million components, while 2GB of physical memory could only accommodate about 2.5 million components. However, as shown in Figure 1 (using somewhat smaller formulas so that we could run the experiment) the utility of the cached components typically declines dramatically with age.

Therefore, we simply give each cached component a sequence number and eliminate those components that are too old. This guarantees an upper bound on the size of the cache; we allow the age limit as an input parameter. For efficiency reasons, age checking is not done frequently; when a new component is cached, we only perform age checking on the chain that contains the newly cached component.

5 Experimental Results

As described in the last section, in our experiments we implemented the cache using a bounded hash table and a scheme for lazily pruning old entries—this entails occasionally having to recompute the value of some components. We also chose the heuristic of always branching on the literal of the current component appearing in the largest number of clauses. Furthermore, we had to ignore unit propagations generated by learned clauses that are outside the current component due to the difficulty of integrating them with the ZChaff design. Hence, the full power of the learned clauses was not available to us.

We have conducted experiments on random 3-CNF formulas and structured problems from three domains. For random 3-CNF formulas as shown in Figure 2, component caching+learning has a many orders

Ratio	1.0	1.2	1.4	1.6	1.8	2.0
relsat	3809	11719	9663	7060	7665	4043
CC+L	2	18	21	35	82	74
relsat	5997	12695	9895	20224	11173	7259
CC+L	5	25	32	41	184	150
relsat	5806	13740	17822	11155	13620	13510
CC+L	5	24	75	61	131	179
relsat	9213	23976	24066	12944	15344	18504
CC+L	6	12	70	90	133	273
relsat	31660	31175	24606	15047	12422	32998
CC+L	20	38	52	101	170	547

Fig. 2. Comparison of `relsat` and component caching+learning (CC+L) on 75 variable random 3-CNF formulas, 5 samples per clause/variable ratio. The number of solutions in these examples ranges from a low of 2.59×10^{13} at ratio 2.0 to a high of 1.96×10^{18} at ratio 1.0.

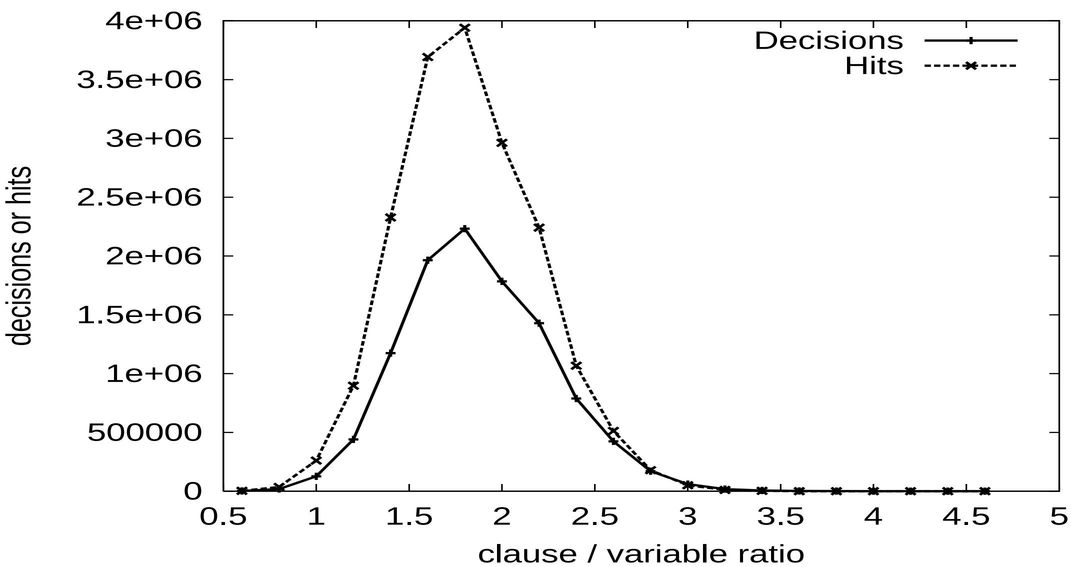

Fig. 3. Comparison of the median number of decisions and cache hits for 75 variable random 3-SAT formulas, 100 instances per ratio

of magnitude advantage over `relsat`. Figure 3 shows that a significant portion of that advantage is due to the cache hits; there are roughly twice as many cache hits as decisions.

For structured problems, in order to compare the performance of different caching and learning schemes, as well as component caching+learning, we also used several other variants of our implementation: caching only, learning only and linear space caching with learning[4]. The formulas are of three types: satisfiable layered grid-pebbling formulas discussed in [9] (where we also add alphabetical tie-breaking for the variable selection rule); circuit problems from SATLIB; and logistics problems produced by Blackbox. Component caching+learning is the best over almost all problems, frequently by a large margin.

6 Conclusions and Future Work

We have presented work that makes substantial progress on combining component caching and clause learning to create an effective procedure for #SAT that does extremely well when compared with either technique alone. The experimental results on both random 3-SAT and structural problems indicate that our program with caching+learning is frequently substantially better than rel-sat and other schemes without

[4] A naive direct implementation of explicit counting on top of ZChaff times out on virtually all of the problems so we do not include it

Grid-pebbing Formulas								
layers	vars	clauses	solns	relsat	caching+learning	caching only	learning only	linear-space
7	56	92	7.79E+10	1	0.01	0.04	0.27	0.07
8	72	121	4.46E+14	49	0.02	0.18	4	0.04
9	90	154	5.94E+23	1438	0.06	0.50	62	6
10	110	191	6.95E+18	X	0.06	0.92	6961	0.67
15	240	436	3.01E+54	X	0.53	106	X	465
20	420	781	5.06E+95	X	3	X	X	X
25	650	1226	1.81E+151	X	35	X	X	X
30	930	1771	1.54E+218	X	37	X	X	X
Circuit Problems								
Problem	vars	clauses	solns	relsat	caching+learning	caching only	learning only	linear-space
ra	1236	11416	1.87E+286	18	8	9	9	8
rb	1854	11324	5.39E+371	80	16	17	22	22
2bitcomp_6	150	370	9.41E+20	272	201	109	746	424
2bitadd_10	590	1422	0	667	475	X	509	505
rand1	304	578	1.86E+54	1731	31	186	1331	1128
rc	2472	17942	7.71E+393	2260	1485	3435	1327	1747
Logistics Problems								
Problem	vars	clauses	solns	relsat	caching+learning	caching only	learning only	linear-space
prob001	939	3785	5.64E+20	< 1	0.57	588	0.75	102
prob002	1337	24777	3.23E+10	4	65	6432	66	245
prob003	1413	29487	2.80E+11	4	119	5545	118	261
prob004	2303	20963	2.34E+28	200	239	X	3766	2279
prob005	2701	29534	7.24E+38	4957	1507	X	X	X
prob0012	2324	31857	8.29E+36	12323	950	X	33082	16162

Fig. 4. Comparisons of `relsat` and different caching and learning schemes on structured problems. (X denotes that the run exceeded the 12 hour time limit.)

caching or learning and is rarely worse. Bounded component analysis was important for the approach, where only original clauses contribute to component analysis and learned clauses only contribute to unit propagations.

However, our current procedure is still very much a work in progress. We have not yet accommodated cross-component implications. More importantly, since the branching order is critical to the search space, we need to have good heuristics on both what component to branch on and what variable in a component to branch on. So far, we simply use DFS for the former and the greedy largest degree heuristic for the latter, so certainly there is plenty of room for improvement.

References

1. F. Bacchus, S. Dalmao, and T. Pitassi. DPLL with Caching: A new algorithm for #SAT and Bayesian inference. In *Proceedings 44th Annual Symposium on Foundations of Computer Science*, Boston, MA, October 2003. IEEE.
2. F. Bacchus, S. Dalmao, and T. Pitassi. Value elimination: Bayesian inference via backtracking search. In *Uncertainty in Artificial Intelligence (UAI-2003)*, pages 20–28, 2003.
3. Roberto J. Bayardo Jr. and Joseph D. Pehoushek. Counting models using connected components. In *Proceedings, AAAI-00: 17th National Conference on Artificial Intelligence*, pages 157–162, 2000.
4. Roberto J. Bayardo Jr. and Robert C. Schrag. Using CST look-back techniques to solve real-world SAT instances. In *Proceedings, AAAI-97: 14th National Conference on Artificial Intelligence*, pages 203–208, 1997.
5. Paul Beame, Russell Impagliazzo, Toniann Pitassi, and Nathan Segerlind. Memoization and DPLL: Formula Caching proof systems. In *Proceedings Eighteenth Annual IEEE Conference on Computational Complexity*, pages 225–236, Aarhus, Denmark, July 2003.
6. Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In *Proceedings, AAAI-02: 18th National Conference on Artificial Intelligence*, pages 627–634, 2002.
7. S. M. Majercik and M. L. Littman. Using caching to solve larger probabilistic planning problems. In *Proceedings of the 14th AAAI*, pages 954–959, 1998.

8. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001. ACM/IEEE.
9. Ashish Sabharwal, Paul Beame, and Henry Kautz. Using problem structure for efficient clause learning. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, 2003.
10. Seinosuke Toda. PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 20(5):865–877, October 1991.
11. Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer Aided Design*, pages 279–285, San Jose, CA, November 2001. ACM/IEEE.