Cache Performance of SAT Solvers: A Case Study for Efficient Implementation of Algorithms

Lintao Zhang and Sharad Malik

Department of Electrical Engineering,
Princeton University, Princeton NJ 08544, USA
{lintaoz,sharad}@ee.princeton.edu

Abstract. We experimentally evaluate the cache performance of different SAT solvers as a case study for efficient implementation of SAT algorithms. We evaluate several different BCP mechanisms and show their respective run time and cache performances on selected benchmark instances. From the experiments we conclude that cache friendly data structure is a key element for efficient implementation of SAT solvers. We also show empirical cache miss rates of several modern SAT solvers based on the Davis-Logemann-Loveland algorithm with learning and non-chronological backtracking. We conclude that recently developed SAT solvers are much more cache friendly in data structures and algorithm implementations compared with their predecessors.

1 Introduction

Boolean Satisfiability (SAT) solvers are beginning to grow out of academic curiosity to become a viable industrial strength reasoning and deduction engine for production tools [4]. As the algorithms of SAT solving mature, the implementation issues begin to show their importance as proven by the recent development of solvers such as Chaff [12] and BerkMin [7]. Similar to earlier SAT solvers such as GRASP [11], relsat [1] and sato [21], these solvers are based on the Davis-Logemann-Loveland algorithm [6] and incorporate learning and non-chronological backtracking proposed by [11] and [1]. By combining efficient design and implementation together with careful tuning of the heuristics, these new solvers are often orders of magnitude faster than their predecessors.

Even though it is widely acknowledged that the recent SAT solvers gain considerable speed up due to careful design and implementation, there are few quantitative analysis to support the argument. As SAT solver design becomes more and more sophisticated, more detailed information about the solving process are needed to squeeze the last bit of efficiency out of the SAT solvers because a small speed up of the engine could translate to days of savings for industrial tools in the field. In order to optimize the SAT solvers, it is necessary to have quantitative analysis results on various aspects of the SAT solving process.

In this paper, we try to analyze one aspect of the SAT solvers, namely cache performance of BCP operation, to illustrate the importance of careful design of data structures and algorithms. It is well known that for current computer architecture, memory hierarchy plays extremely important role on the performances of the computer systems. Algorithms designed with cache in mind can gain considerable speedup over non-cache aware implementations as demonstrated by many authors (e.g. [9, 19]). The importance of cache performance for SAT solvers is first discussed by the authors of SAT solver Chaff [12]. However, no empirical data is available to support the discussion. In this paper, we use a cache analysis tool to quantitatively illustrate the gains achieved by careful designed algorithms and data structures.

The paper is organized as follows. In Section 2, we describe several mechanisms to perform Boolean Constraint Propagation and their implications on the performance of the SAT solver. We evaluate the cache performances of these mechanisms under the same solver framework to illustrate the importance of algorithm design. In section 3, we evaluate the cache performances of several different SAT solvers that are based on the same DLL with learning principle. Experiments show that these solvers vary greatly in their cache performance. Finally we draw our conclusion in Section 4.

2 Cache Performance of Different BCP mechanisms

The Davis-Logemann-Loveland procedure [6] for SAT solving is a branch and search algorithm. One of the main operations of such a procedure is Boolean Constraint Propagation (BCP). After a variable is assigned, the solver needs to propagate the effect of the assignment and find the unit and conflicting clauses that may occur as a consequence of the variable assignment. Conflicting clauses make the solver backtrack from current search space, while unit clauses imply free variables. The process of iteratively assigning values to variables implied by unit clauses until no unit clause exists is called the Boolean Constraint Propagation (BCP).

During a typical SAT solving process, BCP usually takes the most significant part of the total run time. Therefore, BCP algorithms usually dictate the data structure of the clause database and the overall organization of the solver. The implementation of BCP is most essential to the efficiency of the overall implementation of the SAT solvers[1]. In this section, we discuss some popular BCP algorithms and empirically show their implications on the overall efficiency of the SAT solver.

A simple and intuitive implementation for BCP is to keep counters for each clause. This scheme is attributed to Crawford and Auton [5] by [20]. Similar schemes are subsequently employed in many solvers such as GRASP [11], relsat [1], satz [10] etc. Counter-based BCP has several variations. The simplest counter-based algorithm can be found in solvers such as GRASP [11]. In this scheme, each clause keeps two counters, one for the number of value 1 literals in the clause and one for the number of value 0 literals. Each clause also keeps the

[1] In this paper we are only concerned with implementation issues. Improvements in algorithm are often more important, but that's not what we are discussing here.

count of total number of literals in it. Each variable has two lists that contain all the clauses where the variable appears as a positive and negative literal respectively. When a variable is assigned a value, all of the clauses that contain the literals of this variable will have their two counters updated. The data structure of this scheme is like the following:

```
struct Clause {
    int literal_array [];
    int num_total_literal;
    int num_value_1_literal_counter;
    int num_value_0_literal_counter;
};
struct Variable {
    int value;
    Clause pos_occurrence_array [];
    Clause neg_occurrence_array [];
};
struct ClauseDatabase {
    int num_clauses;
    Clause clause_array [];
    int num_variables;
    Variable variable_array [];
};
```

After the variable assignment, there are three cases for a clause with updated counter: if the value 0 count equals to the total number of literals, then the clause is a conflicting clause. If the value 0 count is one less than the total number of literals in the clause and the value 1 count is 0, then the clause is a unit clause. Otherwise, the clause is neither conflicting nor unit, we do not need to do anything. If a clause is found to be a unit clause, the solver has to traverse the entire array of the literals in that clause to find the free (unassigned) literal to imply. Given a Boolean formula that has m clauses and n variables, assuming on average each clause has l literals, then in the formula each variable on average occurs lm/n times. Using this BCP mechanism, whenever a variable gets assigned, on the average lm/n counters need to be updated. Since in this scheme each clause contains two counters, we will call this BCP scheme the *2-Counter Scheme*.

An alternative implementation can use only one counter for each clause. The counter would be the number of non-zero literals in the clause. The data structure for the clause in this scheme is:

```
struct Clause {
    int num_total_literals;
    int literal_array [];
    int num_value_non_0_literal_counter;
};
```

Notice that the number of total literals in a clause do not change during the search. We can save the storage for it by using some tricks to mark the end of the literal array. For example, we can set the highest bit of the last element in the array to 1 to indicate the end of the array.

In this scheme, when a variable is assigned a value, we only need to update (reduce by 1) all the counters of clauses that contain the value 0 literal of this variable. There is no need to update clauses that contain the value 1 literal of this variable because the counters (number of non-zero literal) do not change. After the assignment, if a clause's non-zero counter is equal to 0, then the clause is conflicting. If it is 1, the solver needs to traverse the literal array to find the one non-zero literal. There are two cases: if the non-zero literal is a free (unassigned) literal, then the clause is a unit clause and the literal is implied; otherwise it must already evaluate to 1 and the clause is satisfied, so we need to do nothing.

This scheme essentially cuts the number of clauses that need to be updated by half. Whenever a variable gets assigned, on the average only $lm/2n$ counters need to be updated. However, it does need to traverse clauses more often than the first scheme. If a clause has one literal evaluating to 1 and all the rest evaluate to 0, the previous BCP scheme will know that the clause is already satisfied and no traversal is necessary. In this scheme, the solver still needs to traverse the clause literals because it does not know whether the literal evaluates to 1 or is unassigned. Because in this scheme each clause contains only one counter, we will call it the *1-Counter Scheme*.

Notice that in the data structure described above, the counters associated with each clause are stored within the clause structure. This is actually not good from cache performance point of view. When a variable is assigned, many counters need to be updated. If the counters are located far from each other in the memory, the updates may cause a lot of cache misses. Therefore, to improve the data locality, the counters should be located as close as possible with each other in the memory. It is possible to allocate a separate continuous memory space to store the counters. In that case, the data structure becomes:

```
struct Clause {
    int literal_array [];
    int num_total_literal;
};
struct ClauseDatabase {
    int num_clauses;
    Clause clause_array [];
    int clause_non_0_counter_array [];
    int num_variables;
    Variable variable_array [];
};
```

In this way, we move the counters for non-zero literals in each clause into a array in the `ClauseDatabase` structure. We will call this alternative data structure to implement the 1-Counter Scheme as the *Compact 1-Counter Scheme*.

From a software engineering point of view, the compact 1-Counter Scheme is not a good idea because it destroy the nice object oriented structure of a clause. However, as we will see in the experimental results, this practice improves the overall efficiency of the solver greatly. For large software projects, these kinds of "hack" should be avoided in most cases because of the maintenance issues. But for small kernel engines like a SAT solver, such hacks can often be tolerated to improve performance.

Counter-based BCP mechanisms are easy to understand and implement, but they are not the most efficient ones. Modern SAT solvers usually incorporate learning [11, 1] in the search process. Learned clauses often contain many literals. Therefore, the average clause length l is quite large during the solving process, thus making a counter-based BCP engine relatively slow. Moreover, when backtracking from a conflict, the solver needs to undo the counter assignments for the variables unassigned during the backtracking. Each undo for a variable assignment will update lm/n or $lm/2n$ counters on average depending on the schemes employed.

In [12], the authors proposed a BCP algorithm called *2-literal watching* that tries to address these two problems[2]. For each clause 2-literal watching scheme keeps track of two special literals called watched literals. Each variable has two lists containing pointers to all the watched literals corresponding to it in either phases. We denote the lists for variable v as `pos_watched(v)` and `neg_watched(v)`. Initially the watched literals are free. When a variable v is assigned value 1, for each literal p pointed to by a pointer in the list of `neg_watched(v)`, the solver will search for a literal l in the clause containing p that is not 0. There are four cases that may occur during the search:

1. If there exists such a literal l and it is not the other watched literal, then we remove pointer to p from `neg_watched(v)`, and add pointer to l to the watched list of the variable of l. We refer to this operation as *moving the watched literal*, because in essence one of the watched pointers is moved from its original position to the position of l.
2. If the only such l is the other watched literal and it is free, then the clause is a unit clause, with the other watched literal being the unit literal.
3. If the only such l is the other watched literal and it evaluates to 1, then we need to do nothing since the clause is satisfied.
4. If all literals in the clause is assigned value 0 and no such l exists, then the clause is a conflicting clause.

[2] Another BCP scheme, called head/tail list scheme [20], also claims to have improvements upon counter-based BCP schemes. Unfortunately, there seems to be no *reference implementation* for this scheme. The best solver [7] that employs this scheme is a closed-source solver, therefore, no implementation detail is available. Our in-house developed code that use this scheme cannot validate the claim of it significantly outperforming counter-based schemes. Therefore, we choose not to include the evaluation of this scheme in this paper because our code may not be representative of the best implementations available.

Unlike counter-based BCP mechanisms, undoing a variable assignment during backtrack in 2-literal watching scheme takes constant time [12]. As each clause has only two watched literals, whenever a variable is assigned a value, on the average the state of only m/n clauses needs to be updated assuming watched literals are distributed evenly in either phases. However, each update (i.e. moving the watched literal) is much more expensive than updating a counter as in the counter-based scheme.

To evaluate the BCP mechanisms discussed here. We implemented all of them under a same solver framework based on the SAT solver zchaff [22]. Zchaff already has the data structure optimized for cache performance. Therefore, we can argue that the data presented here is indicative of the characteristics of each algorithms. In the original zchaff code, different BCP mechanisms may lead the solver into different search paths because the variables may be implied in different orders by different implication mechanisms. We modify the zchaff code such that the solver will follow the same search path regardless of the implication method. This modification incur a negligible amount of overhead. Also, because of the modification, the search path are not the same as the default zchaff solver, which is evaluated in the next section.

The instances we use for the evaluation are chosen from various application domains with various sizes and run times to avoid biasing the result[3]. They include instances from microprocessor verification [18] (2dlx), bounded model checking (bmc [17], barrel6 [3]), SAT planning [8] (bw_large.d), combinational equivalence checking(c5315), DIMACS benchmarks (hanoi4), FPGA routing [14] (too_large) as well as a random 3-SAT instance with clause-variable ration of 4.2 (rand)[4]. Table 1 shows the statistics about the instances and the solving process common to all of the deduction mechanisms. These statistics include the number of variables, the number of original and learned clauses and literals as well as the ratio of literals to clauses. We also show the number of implications (i.e. variable assignment) needed to solve the problem. All of the experiments are carried out on a Dell PowerEdge 1500sc computer with 1.13Ghz PIII CPU and 1G main memory. The PIII CPU has a separated level-1 cache of 16K data cache and 16K instruction cache. The level-2 cache is a unified 512K cache. Both L1 and L2 caches have 32byte cache line with 4-way set associativity.

We use `valgrind` [15], an open source cache simulation and memory debugging tool, to perform the cache simulation. The simulated CPU is the same as the CPU we run the SAT solver on. The simulation results are shown in Table 2. In the table, we show the run time (not simulation time) to solve each instance, number of instructions executed (in billions), number of data access (in billions)

[3] Since the topic of this paper is implementation, which is of more interest to real world applications than to algorithm researches, therefore, the benchmarks we choose are mainly from real world instead of randomly generated. Due to the same reason we use zchaff instead of a regular (no learning) DLL algorithm because most real world applications use solvers similar to the algorithms used by zchaff.

[4] The random 3-SAT instance has a clause to literal ratio less than 3 because duplicated literals of the same variable in a clause are removed.

Instance Name	Num. Vars	Orig Cls	Orig Lits(k)	Orig Lits/Cls	Lrned Cls	Lrned Lits(k)	Lrned Lits/Cls	Num. Impl(k)
2dlx_cc_mc_ex_bp_f	4583	41704	118	2.83	13756	1180	85.78	3041
barrel6	2306	8931	25	2.76	34416	2489	72.32	12385
bmc-galileo-9	63624	326999	833	2.55	2372	80	33.94	4629
bw_large.d	5886	122412	273	2.23	14899	399	26.75	11110
c5315	5399	15024	35	2.30	83002	7061	85.07	30174
hanoi4	718	4934	12	2.47	4578	259	56.47	364
rand	200	830	2	2.98	25326	500	19.73	1671
too_largefs3w8v262	2946	50416	271	5.38	71772	2847	39.67	9466

Table 1. Statistics of the Instances Used to Evaluate BCP Mechanisms

and cache miss rates for data accesses. The cache misses for instructions are usually negligible for SAT solvers and therefore are not shown in the table.

From the table we find that different implementations of the BCP mechanism have significant impact on the runtime of the SAT solver. Compare the best performing mechanism, i.e. the 2-literal watching scheme, with the worst scheme, i.e. the 2-Counter scheme, we see that for some benchmarks there is a speed difference of almost 20x. Comparing the 2-Counter Scheme and the 1-Counter Scheme, we find that the 1-Counter scheme reduces memory access by a small amount but the cache miss rates for the 1-Counter scheme is significantly smaller than that of the 2-Counter scheme. This validate our suspicion that the counter updates are the main sources of cache misses. By reducing the counter updates by half, we reduce the cache miss rate by almost a half. Because the 1-Counter Scheme needs more checks for unit clauses as compared with the 2-Counter scheme, the total memory access is not reduced by as much. However, these memory accesses do not cause many cache misses. Comparing the Compact 1-Counter Scheme with the regular 1-Counter scheme, we find that even though the number of the instructions executed and the data accesses are almost the same, the locality of counter accesses improves the cache miss rates of the Compact 1-Counter Scheme significantly. Therefore, the actual run time is also improved by as much as 2x. This again validate our previous conjecture, i.e. counter updates cause most of the cache misses. The 2-Literal Watching scheme performs best among these schemes, with significant smaller number of instructions executed as well as the lowest number of data accesses. Moreover, it has the lowest cache miss rates.

3 Cache Performance of Different SAT Solvers

In the previous section, we evaluated several different BCP mechanisms and showed their respective cache performances. In this section, we evaluate the cache behavior of several existing SAT solvers. In the past, SAT solvers were often designed as a validation for various algorithms proposed by academic researchers.

Instance Name	**RunTime (s)**	10^9 **Instr. Executed**	10^9 **Data Accesses**	**L1 Data Miss Rate**	**L2 Data Miss Rate**
2dlx_cc_mc_ex_bp_f	20.09	10.24	4.78	13.54%	16.02%
barrel6	102.15	48.04	22.49	15.09%	16.61%
bmc-galileo-9	16.60	6.83	3.26	5.69%	56.87%
bw_large.d	118.71	28.63	12.82	14.44%	47.16%
c5315	427.47	169.53	81.21	16.48%	23.39%
hanoi4	1.79	1.63	0.78	12.50%	1.16%
rand_1	96.28	39.20	18.36	19.76%	19.09%
too_largefs3w8v262	730.49	171.81	79.58	20.34%	36.73%

(a) 2-Counter Scheme

2dlx_cc_mc_ex_bp_f	14.47	8.43	4.26	8.76%	15.20%
barrel6	67.61	37.78	19.52	9.08%	15.98%
bmc-galileo-9	12.25	6.43	3.17	3.49%	56.57%
bw_large.d	49.09	22.53	11.06	7.22%	28.87%
c5315	268.18	123.82	66.55	11.91%	21.28%
hanoi4	1.47	1.31	0.68	8.32%	1.59%
rand_1	46.73	23.70	13.20	13.22%	13.00%
too_largefs3w8v262	329.31	103.99	57.29	13.45%	32.25%

(b) 1-Counter Scheme

2dlx_cc_mc_ex_bp_f	12.74	8.12	4.31	5.86%	17.48%
barrel6	52.88	36.18	19.77	6.07%	13.69%
bmc-galileo-9	11.55	6.36	3.20	3.22%	55.21%
bw_large.d	39.84	21.97	11.21	5.23%	27.32%
c5315	172.82	116.22	66.92	7.13%	18.10%
hanoi4	1.35	1.25	0.69	4.13%	2.42%
rand_1	22.67	21.81	13.22	6.33%	7.95%
too_largefs3w8v262	141.19	96.29	57.48	8.29%	18.93%

(c) Compact 1-Counter Scheme

2dlx_cc_mc_ex_bp_f	7.26	5.48	2.13	3.83%	14.52%
barrel6	30.35	21.83	8.31	3.79%	12.90%
bmc-galileo-9	8.12	5.90	2.68	2.16%	54.37%
bw_large.d	25.36	17.10	6.97	3.44%	19.82%
c5315	66.70	45.20	17.78	4.27%	14.44%
hanoi4	0.87	0.79	0.31	3.25%	2.75%
rand_1	5.35	3.78	1.58	4.17%	7.66%
too_largefs3w8v262	41.62	25.99	10.28	3.85%	22.83%

(d) 2 Literal Watching Scheme

Table 2. Memory Behavior of Different BCP Mechanisms

Therefore, little emphasis was placed on implementation issues. Recently, as SAT solvers become more and more widely used as a viable deduction engine, more and more emphasis has been placed on implementing them efficiently. Here, we evaluate several SAT solvers that are based on the same principle proposed by [11, 1].

The solvers being evaluated includes some of the most widely used SAT solvers such as GRASP [11], relsat [1] and SATO [21]. We also included three winners of the most recent SAT solver competition [16] on structured instances: zchaff [22], BerkMin [7] and limmat [2]. JeruSAT [13] is a SAT solver that was developed most recently. These solvers are very similar in the algorithm employed. For example, they all use unit clause implication in deduction and perform non-chronological backtracking and learning. The major algorithmic difference of them are decision strategies and clause deletion polices. Still, we want to point out that cache performance is not the only indication of the overall quality of the implementation because different algorithms employed may dictate different memory behavior.

Due to different branching heuristics used, these solvers behave drastically different on different benchmark instances. Therefore, it is unfair to compare their actual performance by solving a small number of SAT instances. In order to concentrate on the cache behavior of the solvers, we choose some difficult SAT instances and let each solver run under `valgrind` for 1000 seconds[5]. We make sure none of the solvers can solve any of the test instances during the run time in order to focus on cache behaviors independent of the total run time. Except for relsat, all other solvers use default parameters. We use option `-p0` on relsat to disable the time consuming preprocessing. The benchmarks used includes SAT instances generated from bounded model checking [3] (barrel, longmult), microprocessor verification [18] (9vliw, 5pipe), random 3-SAT (rand2) and equivalence checking of xor chains (x1_40). The statistics of the benchmarks we use are shown in the first two columns of Table 3.

The results of cache miss rates for different solvers are listed in Table 3. From the table we find that earlier SAT solvers such as GRASP, relsat and SATO performs poorly in memory behavior. The solvers would be somewhat around 3x slower than the new contenders such as Chaff and BerkMin solely due to the high cache misses. One of the reasons for the high cache miss rates of GRASP and relsat is due to their use of counter-based BCP mechanism. But compare their cache miss rates with the best that can be achieved by counter-based BCP mechanisms shown in previous section, we find that they are still under par even considering their BCP mechanisms[6]. SATO use the same BCP mechanism as BerkMin, but BerkMin seems to be much better optimized in cache performance. One benchmark SATO performs well is the xor chain verification instance x1_40. The reason is because this is a very small instance and SATO has

[5] Programs running under valgrind are around 30 times slower than on the native machine.

[6] The benchmarks used are not exactly the same for these two experiments, but the numbers should be representative of their respective behavior.

Instance Name	Num. Variables	Num. Clauses	GRASP		SATO		relsat	
			D L1%	D L2%	D L1%	D L2%	D L1%	D L2%
x1_40	118	314	23.72	75.22	1.72	0.00	1.73	0.01
rand_2	400	1680	25.05	60.38	36.79	0.57	11.14	0.12
longmult12	5974	18645	20.86	57.69	19.32	41.68	13.24	19.78
barrel9	8903	36606	21.22	51.73	21.62	55.55	15.73	19.14
9vliw_bp_mc	20093	179492	22.43	88.58	40.49	42.85	14.58	55.34
5pipe	9471	195452	31.14	81.74	36.22	4.22	15.38	33.39

Instance Name	zchaff		BerkMin		Limmat		JeruSAT	
	D L1%	D L2%	D L1%	D L2%	D L1%	D L2%	D L1%	D L2%
x1_40	3.32	26.32	7.17	17.14	6.05	64.45	4.31	2.35
rand_2	6.58	23.46	7.63	11.25	5.83	44.83	5.66	14.90
longmult12	6.46	18.17	7.87	20.80	5.77	40.31	5.61	61.47
barrel9	6.62	33.43	10.98	32.04	7.58	34.39	5.55	54.22
9vliw_bp_mc	9.48	54.37	3.69	9.44	5.68	53.98	7.04	51.95
5pipe	6.14	21.54	5.31	18.51	6.45	46.12	6.49	42.66

Table 3. Memory Behavior of Different SAT Solvers

Instance Name	Num. Variables	Num. Clauses	zchaff		BerkMin		Limmat	
			D L1%	D L2%	D L1%	D L2%	D L1%	D L2%
2dlx	224920	3596385	11.47	91.22	7.38	69.12	6.00	74.01
pipe	64262	2291120	10.49	54.37	8.42	61.95	5.60	51.12

Table 4. Cache Performance of the Best SAT Solvers on Challenging Benchmarks

a very aggressive clause deletion heuristic. Therefore, SATO can fit the clause database into the level-1 cache while other solvers cannot due to their more tolerant clause deletion policies. The same reason explains the low cache miss rates for relsat on x1_40 and both SATO and relsat's low L2 miss rates for rand_2. The newer SAT solvers such as zchaff, BerkMin, limmat and JeruSAT all have relatively good cache behavior. Part of the reason for this is due to their better BCP mechanisms, but better design and implementation also contribute to their low cache miss rates.

As a final experiment we show the cache performance of zchaff, BerkMin and Limmat: three state-of-the-art SAT solver's cache performance on two very large and challenging benchmarks. The benchmarks are obtained from Miroslav Velev's microprocessor verification problem [18]. They come from fvp-unsat-3.0 (pipe_64_16) and SSS-Liveness-SAT-1.1 (2dlx_cc_ex_bp_f_bug3) suite respectively. These represent the cache performance of most challenging SAT instances on the best SAT solvers available. We run each instance under `valgrind` for 10000 seconds. The results are shown in Table 4.

From Table 4 we find that as the formulas become very large, both level-1 and level-2 cache miss rates for the solvers increase considerably. Among these three solvers, zchaff is the oldest one while limmat is the newest. From these limited experimental results it seems that the cache performances of the SAT solvers are still improving incrementally on carefully designed SAT solvers, whether on purpose or coincidentally. In current (and future) generation of microprocessors, the speed difference of main memory (on the mother board) and L2 cache (on die) tends to be large. Therefore, L2 miss penalty is quite high. From these results we find that there are still a lot of space for improvements for future efficient implementation of SAT solvers.

4 Conclusion

In this paper, we investigate the memory behavior of several different BCP mechanisms for Boolean Satisfiability Solvers. We find that different implementation of BCP can significantly affect the memory behavior and overall efficiency of the SAT solvers. From the experiments we conclude that cache friendly data structure is a key element for efficient implementation of SAT solvers. We also show empirical cache miss rate data of several modern SAT solvers based on the Davis-Logemann-Loveland algorithm with learning. We find that recently developed SAT solvers are much more cache friendly compared with their predecessors. We use this as a case study to illustrate the importance of algorithm implementation. Efficient implementation (not necessarily limited to cache behavior) is key to the success of a modern SAT solver and should not be overlooked.

References

1. Roberto J. Jr. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, 1997.
2. Armin Biere. Limmat sat solver. http://www.inf.ethz.ch/personal/biere/projects/limmat/, 2002.
3. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), number 1579 in LNCS*, 1999.
4. Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an alpha microprocessor using satisfiability solvers. In *Proceedings of 13th Conference on Computer Aided Verification(CAV'01)*, 2001.
5. J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI'93)*, pages 21–27, 1993.
6. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
7. E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, March 2002.
8. H. A. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

9. A. LaMarca and R.E. Ladner. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithmics*, 1, 1996.
10. Chu-Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 366–371, Nagoya, Japan, August 23–29 1997.
11. João P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996.
12. 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 (DAC'01)*, June 2001.
13. Alexander Nadel. Jerusat sat solver. http://www.geocities.com/alikn78/, 2002.
14. G. Nam, K. A. Sakallah, and R. A. Rutenbar. Satisfiability-based layout revisited: Routing complex fpgas via search-based boolean sat. In *Proceedings of International Symposium on FPGAs*, Feburary 1999.
15. Julian Seward. Valgrind memory deubugger and cache simulator. http://developer.kde.org/ sewardj/, 2002.
16. Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. Sat 2002 solver competition report. Available at http://www.satlive.org/SATCompetition/onlinereport.pdf, 2002.
17. Ofer Strichman. Tuning sat checkers for bounded model-checking. In *Proceedings of Computer Aided Verification, 2000 (CAV'00)*, 2000.
18. M.N. Velev and R.E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, pages 226–231, June 2001.
19. L. Xiao, X. Zhang, and S. A. Kubricht. 'improving memory performance of sorting algorithms". *ACM Journal of Experimental Algorithmics*, 5:1–23, 2000.
20. H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale (Florida USA), 1996.
21. Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.
22. Lintao Zhang. Zchaff sat solver. http://www.ee.princeton.edu/ chaff/zchaff.php, 2000.