Towards Improving the Resource Usage of SAT solvers

Norbert Manthey* and Ari Saptawijaya

International Center for Computational Logic
TU Dresden, 01062 Dresden, Germany
Norbert.Manthey@mail.inf.tu-dresden.de

Abstract. The paper presents our work on cache analysis of SAT-solving. The aim is to study how resources are utilized by a SAT solver and to use this knowledge to improve the resource usage in SAT-solving. The analysis is performed mainly on our CDCL-based SAT solver and additionally on MiniSAT and PrecoSAT. The measurement is conducted using sample-based profiling on some industrial benchmark from the SAT competition 2009. During the measurement the following hardware events are traced: total cycles, stall cycles, L2 cache hits and L2 cache misses. From the measurement results, our runtime and implementation analysis unveil that several improvements on resource usage can be done, in particular on data structures and memory access. These improvements result in about 60% speedup of runtime performance for our solver.

1 Introduction

The satisfiability problem (SAT) has its importance not only theoretically but also practically. The active development of SAT solvers in recent years has turned SAT solvers into a powerful tool to solve SAT-encoded problems in various fields, from model checking to bioinformatics. Encoding industrial problems often results in large SAT instances with millions of variables and clauses. Therefore it is important to have appropriate data structures and techniques to handle them. Driven by the annual SAT competitions and SAT races, many improvements have been made on SAT solvers. These include improvements on the algorithm and employing various heuristics and cache-aware data structures.

In this work we study how computing resources are utilized by a SAT solver in solving industrial problems. In particular, we observe and analyze the use of cache in solving SAT instances. We aim at an optimal use of cache by a SAT solver as cache allows a faster data access compared to main memory access. In the end, this leads to the improvement of the overall solver performance.

We consider a conflict-driven clause learning (CDCL)-based SAT solver in our work. The solver is based on HydraSAT [3], that is developed in our group. The solver employs components used commonly in state-of-the-art SAT solvers. The

* The author was supported by the European Master's Program in Computational Logic (EMCL).

industrial problems in the study are taken from some benchmark used in the last SAT competition 2009. The measurement is conducted using the HPCToolkit [1] via sample-based profiling. At each sample point, the performance counter is accessed using the PAPI library [2]. Using these performance measurement tools, we observed the following processor events: total cycles, stall cycles, L2 cache hits and L2 cache misses. Additionally, the number of clause read-access and write-access are collected.

The runtime analysis from the measurement suggests some improvements that can be made on data structures. The improvements particularly deal with the clause representation and with prefetching clauses in a watcher list. We propose the use of the slab allocator [5] that, together with several clause representation schemes, improve the runtime performance of the solver up to 23% speedup. Two prefetching schemes are also proposed which give about 12% speedup of runtime performance. Based on the implementation analysis, we also carry out some experiments to improve memory access. The experiments include reusing data structure vector in conflict analysis and compressing some data structures, i.e. literals, boolean arrays and the truth-value assignment. We also consider a scheme to maintain a watcher list lazily, where the "gap" in the list (due to removing a clause from it) is not closed immediately by moving subsequent clauses in the list. Instead, this "gap" is only closed once the unit propagation terminates. This lazy maintenance of watcher list speeds up the solver's runtime by about 24%. The most encouraging result is obtained when several improvements are combined, where the runtime performance of the solver can be improved up to 60% speedup. All the improvements we consider in this work do not change the search-path in finding a solution. Besides our solver, we also measure and evaluate the cache performance of MiniSAT 2.0 [7] and PrecoSAT [4].

The paper is organized as follows. The description of the solver is given in Section 2. Then, in Section 3 we detail how the measurement is conducted. We discuss the measurement results and the analysis in Section 4. The proposed improvements based on the analysis are examined in Section 5. Finally, we conclude in Section 6 by discussing related work and some future work.

2 Description of the Solver

The solver used in this work is based on the conflict-driven clause learning (CDCL) procedure. It is customized from HydraSAT [3], a solver that is implemented in C++. The solver is compiled to a 64-bit binary using the GNU Compiler version 4.1.2 with the highest optimization level -O3.

A literal is implemented using a 32-bit unsigned integer. A clause is implemented by storing its activity (32-bit floating point), its size (32-bit unsigned integer) and a pointer to the literals of the clause. In case a clause is used in several solver components, no copy of this clause is made. Instead, only the address of the clause is shared among the components. Finally, a formula is implemented as a vector of pointers to clauses it contains. Auxiliary data structures used in the solver are vectors, stacks, double-ended queues and priority queues. The first

three are adopted from the C++ Standard Template Library. The priority queue is implemented using a binary heap. We consider the following components for our solver.

- *Unit Propagation.* The two watched-literal scheme, which is introduced in Chaff [10] to improve the cache performance, is used for the unit propagation component. As usual, this scheme is realized by maintaining watcher lists. The implementation handles the binary clauses separately from the other longer clauses. Unit propagation is performed firstly on binary clauses and then on longer clauses. Due to the special treatment for binary clauses, watcher lists for literals of binary clauses are introduced. For binary clauses, the watcher list of a literal does not only store the pointer to the clauses, but it stores also the other literal of the clause.
- *Conflict Analysis.* The first UIP scheme [9] is used for the conflict analysis component. Additionally, the learnt clause obtained from the conflict analysis is further minimized using self-subsumption [8].
- *Decision Heuristics.* The decision heuristic used follows the basic principle of VSIDS [10]. Each variable is assigned an activity and the variable with the highest activity is picked as a decision variable. Every 1000 decisions an attempt to pick a decision variable randomly takes place (up to ten attempts). If these attempts fail, a deterministic decision is made using the activity-based heuristic. Decision variables are assigned negative polarity.
- *Restart Event Heuristics.* This component manages the scheduling of restart. The first restart is performed after 100 conflicts. Restart is scheduled according the following scheme:

$$noc(t) = \begin{cases} 0 & \text{, if } t = 0 \\ 100 & \text{, if } t = 1 \\ (noc(t-1) - noc(t-2)) \cdot r + con & \text{, otherwise} \end{cases}$$

  where $noc(t)$ is the number of conflicts needed to restart at time t, r is a constant and con is the current total number of conflicts. In our solver, the value of r is set to 1.5.
- *Removal Heuristics.* Removal of learnt clauses is scheduled immediately after a restart. Hence, there is no need to check whether the clauses to be removed are still actively participated in the current search. In our solver, the heuristic is to remove:
  - learnt clauses with more than six literals, and
  - the oldest 55 % of the remaining learnt clauses with more than two literals.

3 Measurement

The measurement is conducted using the HPCToolkit [1] via sample-based profiling. The solver is run and halted at a specified processor event and the method currently running is analyzed. Such an event is triggered when a performance

counter reaches the maximum of its period. When the program is halted, the performance counters are read using the PAPI Library [2]. The precision of the measurement is assured due to the long run of the solver. This long run ensures the collection of lots of samples during the measurement.

In the measurement, four processor events are traced simultaneously: total cycles, stall cycles, L2 cache hits and L2 cache misses. The sample rate for the total cycles is 10^6, whereas the sample rate for the other three events are set to 10^5. In addition to tracing processor events, the number of clause read-access and write-access are observed. Observing the behavior of clause read-access and write-access allows us to learn which among the two accesses is more frequent. Hence, the more frequent access can be treated differently.

For the measurement, 40 problem instances of the industrial benchmark from the the SAT competition 2009 are used. These are the instances that are solved within 45 minutes timeout using the basic version of our solver, i.e. without any improvement which will be discussed subsequently. Appendix A shows the selected instances together with their solving time and memory usage. The total runtime for the benchmark is 9.5 hours. The measurement is performed on a hardware with AMD Opteron 285 2.66 GHz processor, 1024 KB Level 2 Cache, 2 GB main memory, 64-byte cache line size. In the subsequent sections, the computation of *runtime*, *wait rate*, *L2 cache access*, *L2 cache miss rate* and *work cycles* is as usual:

- $runtime = total\ cycles \times CPU\ Frequency$
- $wait\ rate = stall\ cycles / total\ cycles$
- $L2\ cache\ access = L2\ cache\ misses + L2\ cache\ hits$
- $L2\ cache\ miss\ rate = L2\ cache\ misses / L2\ cache\ access$
- $work\ cycles = total\ cycles - stall\ cycles$

Since L1 cache is not analyzed, in the sequel *memory access* refers also to L2 cache access.

4 Results and Analysis

Based on the measurement results, two analysis are performed: the runtime analysis and the implementation analysis. The former involves the analysis on the processor events and data structure accesses. The latter concerns with the implementation of the solver that suggests some slight improvements.

4.1 Runtime Analysis

Data structure accesses are important to determine the important part of the solver's data. This information cannot be obtained using the HPCToolkit, thus additional runs for measurement have to be performed. Fig. 1 depicts the literal accesses in clauses. The most frequently accessed literals are the literal at index 0 (60% read access, 25% write access) and at index 1 (15% read access, 50% write access) of the literal array. The total number of write accesses is only 17%

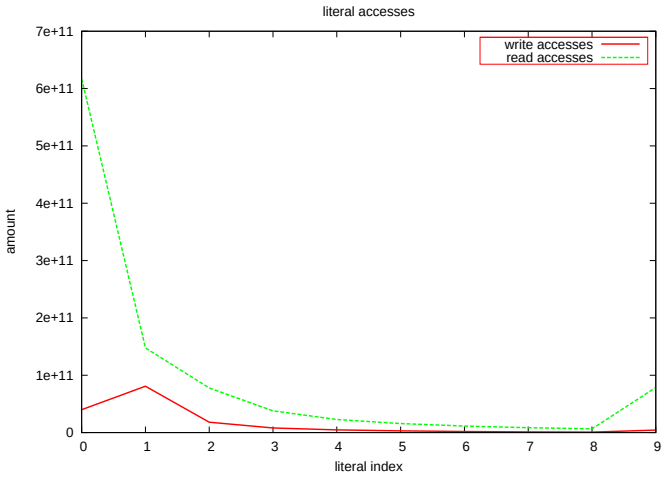

Fig. 1. Literal accesses in clauses.

of read accesses. Note that, literals are only accessed in the unit propagation and in the conflict analysis components.

Table 1 shows the distribution of each processor event amongst the solver components. It can be seen that most of the runtime is spent by the unit propagation. The conflict analysis component consumes only about 6% of the runtime, whereas the remaining components share only 2% of the runtime. Most of the L2 cache misses and hits are produced in the unit propagation as well. Hence, this component needs to be optimized in order to obtain a high impact for the solver's performance.

Based on the measurement result, the wait rate of the solver, i.e. the part of of the execution time where the solver waits for a resource, is 82%. This value indicates that the solver does not use the provided resources well. Computing the L2 cache miss rate, we obtain the value of 40% for our solver.

Analyzing the unit propagation further, we obtain the distribution of each processor event for the propagation on binary clauses and on longer clauses as shown in Table 2. The result indicates that the unit propagation spends most of its runtime in propagating unit on longer clauses than on binary clauses. Furthermore, in propagating on longer clauses most of the runtime is spent in literal read accesses (45.8%) and maintaining the watcher lists (24.26%).

Following the two watched-literal scheme, the watcher list of the literal to propagate is accessed and all the clauses in this list are processed sequentially. Visiting these clauses results in two cache misses, in case the other watched literal is not satisfied. This scheme is considered expensive. The memory scheme of accessing the first literal of a clause in a watcher list during propagation is shown in Fig. 2. The first cache miss occurs when the clause head is visited. The

Component	Total Cycles	Stall Cycles	L2 Misses	L2 Accesses
Decision Heuristics	1.77%	1.59%	3.13%	2.95%
Removal Heuristics	0.31%	0.21%	0.09%	0.21%
Conflict Analysis	5.74%	5.42%	6.27%	7.27%
Restart Event Heuristics	0.00%	1.33%	0.00%	0.00%
Unit Propagation	91.65%	92.62%	90.08%	88.94%

Table 1. Distribution of processor events in solver.

	Total Cycles	Stall Cycles	L2 Misses	L2 Accesses
Propagate on binary clauses	5.71%	5.55%	7.95%	5.64%
Propagate on longer clauses	83.86%	85.30%	78.17%	79.78%
Literal read access	45.8%	54.49%	24.07%	12.57%
Maintain watcher list	24.26%	18.59%	2.19%	36.64%

Table 2. Distribution of processor events for Unit Propagation.

second one results from visiting the first literal afterwards. Note that, the cache misses in looking up the truth value of a literal are negligible (less than 2%). The cache miss due to the extracting the watcher list of a literal occurs only once (while propagating this literal). In the sequel, we refer it as 0^{th} cache miss.

Analysis on MiniSAT 2.0 Our solver implements similar data structures as MiniSAT. This leads to similar behavior as well. The analysis can only be done on 39 out of 40 problem instances because MiniSAT fails to solve one instance within the timeout. MiniSAT needs only 80% of the memory accesses compared to our solver. Due to a similar number of L2 cache misses, the L2 cache miss rate of MiniSAT is 50%. On the other hand, the work cycles of MiniSAT are comparable to the work cycles of our solver, indicating that the two algorithms are suited for the benchmark equally well. The wait rate of MiniSAT is equal to our solver.

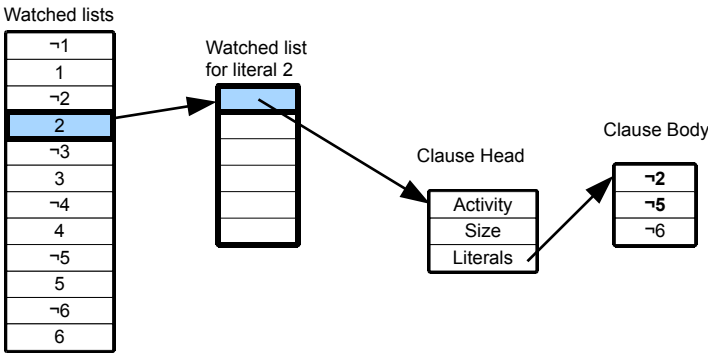

Fig. 2. Accessing the first literal of a clause using the two watched-literal scheme.

Analysis on PrecoSAT PrecoSAT is much faster on the given benchmark than the other two solvers. Similarly to MiniSAT, PrecoSAT solves 39 instances but it uses only 33% of our solver's runtime. The 37% work cycles indicate that PrecoSAT implements a better algorithm, but the solver does not seem to utilize hardware much better than MiniSAT or our solver. PrecoSAT's L2 cache miss rate is 36% and it spends 77% of the execution time waiting for resources.

4.2 Implementation Analysis

We review the implementation of the solver and we observe three flaws in the implementation.

The amount of memory accessed during solving instances can be reduced by compressing data, especially boolean arrays and the truth-value assignment of the solver. Applying the assignment compression [6] saves 75% compared to the original size, because four ternary truth values (positive, negative, undefined) can be stored in a byte. A similar approach can be applied to boolean arrays, which saves about 88% of its original size. Nevertheless, the compression of boolean arrays has a cost as it requires additional instructions that are executed every time the array is accessed.

To enable the phase-saving heuristic in choosing the polarity of a decision variable [11], the truth-value assignment also stores for every variable the backup polarity (i.e. the polarity previously assigned but erased due to backtracking). This polarity is stored next to the current polarity. Thus, reading only an assignment loads every second byte (that stores the backup polarity) unnecessarily into the cache. In fact, this byte that stores the backup polarity is used only when the variable is assigned undefined.

Memory accesses can be avoided in the implementation by reusing data structures. A newly created vector without specifying a size results in a vector with no allocated storage capacity. Enlarging a vector allocates a new piece of memory and copies the content of the old piece of memory to the new one. Afterwards, the old piece of memory is freed. On the other hand, clearing a vector keeps its allocated capacity. Thus, copying memory can be avoided by clearing and reusing a vector, instead of deleting the vector and creating a new one.

5 Improvements

The analysis from Section 4 suggests some improvements with the goal to reduce cache misses and improving data locality. The main reason for cache misses is the separation of the clause head and the clause body. We discuss several improvements that can be done and their evaluation, in this section.

5.1 Clause Access Improvements

The first idea is to move literals from the clause body to the clause head. This improvement is called *cache clause*. By moving four literals, we obtain up to 19%

of runtime speedup. This result is similar to clause packing improvement in [6]. The number of L2 cache misses is reduced by 32%. Choosing to access the local stored literals or the clause body increases the work cycles by 3%.

With the cache clause improvement, the size of the clause head becomes 32-byte, so that two clause heads fit exactly on one cache line. Since all clauses are allocated using the memory allocator `malloc`, 8-byte additional storage (for system information) is added to *every* allocation. This storage prevents the system to place two clause heads compactly on a single cache line. Using the *slab allocator* this additional storage can be avoided [5], since it stores the 8-byte system information only *once*, allowing the clause heads to be stored compactly. We implement our own slab allocator for our purpose. In order to handle clauses with variable sizes, one needs a separate allocator for each size (due to the fixed slab size). Multiple slab allocators of different sizes are then combined in a wrapper. When an allocation should be done for a clause, an allocator for the corresponding size (of the clause) is then chosen from the wrapper. The slab allocator alone does not affect the runtime and main memory accesses, but combining it with the cache clause leads to a better improvement. Combining the two improvements results in 23% speedup of the runtime and the number of L2 cache misses is reduced by 26% compared to the basic version of the solver.

Another approach is to store the clause in an array and to combine the clause head and body [7]. This improvement is dubbed *flattened clause*. With this scheme, accessing the size and the activity of a clause is less flexible, but no additional instruction is needed to determine whether to access the locally stored literals or the clause body. Fig. 3 shows the implementation and the memory scheme of the flattened clause improvement. Together with the `malloc` allocator, this scheme yields 21% runtime speedup. The L2 cache miss rate is decreased by 20% and 24% of the L2 cache accesses in the basic version of the solver are caught by the L1 cache. Combining the flattened clause with the slab allocator improves the runtime by 22%.

The improvements in clause implementation avoid the second cache miss occurring in visiting the clauses in a watcher list as discussed in Section 4.1. In

```
typedef literal_t* cls;
cls literals;

literal_t literal(cls literals,uint32_t index){
   return literals[index];
}
uint32_t size(cls literals){
   return ((uint32_t*)literals)[-1];
}
float activity(cls literals){
   return ((float*)(&(((uint32_t*)literals)[-1])))[-1];
}
```

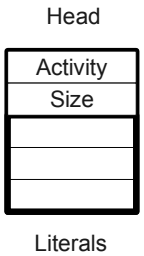

Fig. 3. The implementation and the memory scheme of the flattened clause.

[6], some literals of the clauses in a watcher list are stored directly in the watch list itself in order to avoid the first cache miss. Unfortunately the search path may change by this improvement.

The first cache miss can also be avoided if the prefetch unit is used to store the clauses of a watcher list in the cache. The GNU Compiler provides an instruction that tells the prefetcher the address to fetch into the cache. Since the watcher list is traversed linearly, the clauses it contains can be prefetched.

To avoid 0^{th} cache miss, the corresponding watcher list is prefetched as soon as another literal is added to the propagation queue. Prefetching the clauses (effectively the clause heads) can be done in two ways: either all clauses in the currently visited watcher list are prefetched (*first prefetching scheme*) or the clauses from the watcher lists of the first *depth* literals in the propagation queue are prefetched (*second prefetching scheme*). Note that, *depth* is a parameter and it refers to the number of consecutive literals in the propagation queue.

The following results are obtained without improving the implementation of clause. The first prefetching scheme delivers 12% speedup and the stall cycles are reduced to 84%. The results of the second prefetching scheme depend on the parameter *depth*. For $depth = 10$, we gain 4% speedup of the runtime. This parameter can be tuned further, but this is not considered in the current work. In both schemes the number of work cycles, cache hits and cache misses increase because unnecessary clauses are prefetched due to a conflict which stops the unit propagation. The prefetch unit does not introduce any negative impact on the runtime.

5.2 Reducing Memory Accesses

As shown in Table 2, the maintenance of the watcher lists, i.e. removing elements from the watcher list, needs almost 25% of runtime. Removing a clause from a watcher list pushes all subsequent clauses one position forward. As a result, lots of memory accesses are performed. Using a linked-list instead of a vector for a watcher list reduces the memory accesses by 13%, but it takes longer runtime (increase by 20%). The negative impact is caused by the high miss rate of L2 cache, i.e. 71%, which results from the non-linear read access of the list elements.

In the maintenance of a watcher list, in fact there is no need to push all subsequent clauses immediately to fill in the "gap" (that occurs due to removing a clause from it). The maintenance of the watcher list can be done lazily. This can be illustrated as follows. Suppose that the first clause is removed from the list, the pointer of this clause is kept and marked as a gap. This means, the subsequent clauses are not pushed immediately forward. Suppose that the second clause has also to be removed, then we leave a wider gap (of two clauses) in the list. If the third clause is not removed from the list then this clause can be pushed forward to the top of the list, making the gap in the list smaller. Note that, only this clause is pushed forward, as the following clauses could be potentially removed as well (thus, leaving a new gap). In the end, when the propagation stops (e.g. due to a conflict) then all the gaps can be removed at once from the watcher list. This lazy maintenance of watcher list results in 23% speedup of the solver's

runtime, decreases the memory accesses by 35% and thus increases the L2 miss rate by 54%. This scheme also decreases the work cycles by 52%. These results indicate that the process of maintaining a watcher list is buffered completely in the L2 cache. We refer this improvement scheme as the *lazy maintenance*.

Data structures compression may reduce memory accesses further. This compression includes the compression of the truth-value assignment and boolean arrays. These improvements are called *compressed assignment* and *compressed boolean arrays*, respectively. Our experiment shows that both compressions do not lead to any impact on the runtime performance. Some speedup gained from less L2 cache accesses and misses has to compensate the compression and decompression operations.

The assignment can be stored more compactly by storing the backup polarities from the currently used ones separately, rather than storing both polarities next to each other (cf. Section 4.2). The assignment is partitioned in two halves, albeit stored in a single array (it can alternatively be realized using two separate arrays). The half partition of backup polarities is identified by indexing the assignment with the negative variable. The number of memory accesses is reduced by 1% using this scheme and the runtime is slightly better than the runtime of the above compressing schemes. This improvement is called the *negative index assignment*.

Compressing the literals as it is done in siege [12] is also analyzed in this work. The compression is able to store three literals in a 64-bit integer and reduces the storage needed by 33% in the best case. The maximum number of variables in the formula is reduced to 2^{20}, because the representation of one compressed literal is stored in 21-bit. The number of L2 cache misses reduces by 3%, but the number of work cycles increases by 17%. As a result, the runtime does not change. The number of memory accesses decreased by almost 1%.

The implementation of the conflict analysis needs three vectors. The first one stores the literals of the learnt clause. The second vector stores a backup of the first one during minimizing the learnt clause. The last vector stores temporary literals that have to be processed. Clearing and reusing these vectors lead to 4% runtime improvement and the number of memory accesses decreases by 3%. This improvement is called the *vector reuse*.

5.3 Combination of Improvements

Most of the improvements described previously can be combined. Table 3 gives the results of the six combinations with respect to the total cycles, L2 accesses, L2 miss rate and wait rate. Note that, the values for total cycles and L2 accesses of each combination are relative to those of the basic version, whereas the values for the L2 miss rate and wait rate are absolute. The following acronyms are used in defining the combinations: CC, slab, VR, P1, NA, CBA, CA and LM refer to the cache clause (with four local literals), slab allocator, vector reuse, the first prefetching scheme, the negative index assignment, the compressed boolean arrays, the compressed assignment and the lazy maintenance improvement, respectively.

Configuration	Total Cycles	L2 Accesses	L2 Miss Rate	Wait rate
Basic Version	100.0%	100.0%	40.94%	81.12%
Combination 1	40.93%	56.3%	47.68%	75.56%
Combination 2	39.91%	56.72%	48.7%	75.88%
Combination 3	41.01%	56.01%	48.05%	75.82%
Combination 4	40.9%	56.51%	48.86%	76.14%
Combination 5	40.69%	54.56%	48.25%	74.86%
Combination 6	39.7%	51.71%	49.3%	72.21%

Table 3. Results of improvement combinations. In this table, Basic refers to the basic version of the solver, Combination 1 = CC + slab + VR + P1 + LM, Combination 2 = FC + slab + VR + P1 + LM, Combination 3 = Combination 1 + NA, Combination 4 = Combination 2 + NA, Combination 5 = Combination 3 + CBA and Combination 6 = Combination 1 + CA + CBA.

	Total Cycles	Improvement total cycles	Work Cycles	Improvement work cycles
Combination 6	100%	60.31%	100%	42.62%
Decision Heuristic	4.28%	0.07%	4.52%	-0.02%
Removal Heuristic	0.68%	0.04%	2.33%	-0.58%
Conflict Analysis	13%	0.58%	15.97%	-1.99%
Restart Event Heuristic	0%	1.33%	0.01%	1.33%
Unit Propagation	80.87%	59.55%	74.48%	44.56%
Propagate on binary clauses	14.42%	-0.01%	13.07%	-1.08%
Propagate on longer clauses	61.47%	59.46%	54.14%	46.32%
Literal read access	9.17%	16.04%	8.34%	-3.87%
Maintain watched list	0.22%	24.18%	0.34%	49.51%
Prefetch memory	23.14%	-9.18%	3.46%	-1.98%

Table 4. Comparing cycles distribution of basic version and combined improvements.

All combinations result in a runtime improvement of almost 60%. Combinations with slab, VR, P1, LM together with a clause improvement (CC or FC), as in combination 1 and 2, serve as the core of optimizations. The performance drops significantly when only slab, VR, P1 and LM are considered (without any clause improvement), where we obtain 64.49% total cycles, 73.92% L2 accesses, 55.42% L2 miss rate and 84.45% wait rate. The gained improvement does not interfere with compressing data structures much. We only further analyze the combination with the best runtime improvement, viz. Combination 6. Table 4 shows the distribution of its runtime and work cycles. It also lists the amount of improvement obtained by comparing its absolute runtime (and work cycles) to the runtime (and work cycles, respectively) of the basic version. Compared to the basic version of the solver (Table 1), the distribution of the total cycles moves from the unit propagation to other components. The conflict analysis now needs 13% of the runtime. The work cycles change mainly for the unit propagation.

The major improvement is achieved in propagating longer clauses. The runtime improvement of this part is caused by the improvement of the literal read access (16%) and the lazy watcher list maintenance (24%). The newly introduced prefetching scheme consumes about 9% of this improvement.

The unit propagation still remains at the heart of the solver and is the part where further improvements concerning the resource usage should be applied. The usage of the prefetch function seems to offer some space for the optimization of the solver, since it requires only about 4% of the work cycles but consumes about 23% of the total runtime.

6 Conclusion and Future Work

In this paper we have described our study on how computing resources are utilized by a modern SAT solver in solving several industrial problems. The aim is to improve the resource usage in SAT-solving and the overall performance of SAT solvers. We perform cache analysis of our CDCL-based SAT solver via sample-based profiling measurements on some industrial benchmark from the SAT competition 2009. The analysis of the measurements suggests several improvements, which include efficient representations of clauses, the use of the slab allocator and clause-prefetching schemes in two watched-literal propagation. Additionally, compression schemes of several data structures and lazy maintenance of watcher list are also considered. By combining several improvements, the runtime performance of the solver can be improved up to 60% speedup.

The idea of cache clause improvement and compressed assignment is also considered in [6], to improve MiniSAT. MiniSAT also enjoys lazy maintenance of watcher lists, similar to what we describe here. There is also some similarity between the idea of the flattened clause and the clause representation described in [13], where the clause head and body are combined and stored in an array. Differently from [13], the additional clause offsets array is not needed, since we do not store the whole clause database in a single array. We also record in our clause representation the activity of clause instead of the number of watched literals. Cache performance of SAT solvers has also been studied in [14]. Compared to [14], we do not study the cache performance on various unit propagation schemes. Instead, we consider only the two watched-literal propagation, which is commonly used in recent solvers, and study some improvements that can be done on it further. We also examine the cache performance of some recent solvers, viz. MiniSAT and PrecoSAT (as well as our own solver), using measurement tools different from [14]. In addition to that, our measurements are conducted on more SAT instances, taken from the recent benchmark of SAT competition. This is important in order to validate our analysis better. Our work is an extension of the above mentioned previous work, as we also examine several other improvements. These improvements include the slab memory allocator and the clause-prefetching schemes in two watched-literal propagation, that combined with some other improvements, increase the solver's performance significantly.

As a future work, we would like to study further improvement through Translation Lookaside Buffer (TLB). A preliminary study shows that using 2 MBytes page size (instead of 4 KBytes page size) our cache-optimized solver obtains up to 10% performance speedup. For this result, the benchmark is run on an Intel Core i7 860 CPU with 8 MBytes L2 cache and a clock frequency of 2.80 GHz. Using the 2 MByte pages decreased the runtime of both the basic and the improved version of the solver. This kind of improvement requires that the underlying machine supports huge page size. It is also interesting to study the impact of the slab allocator and various prefetching schemes in some other solvers, e.g. MiniSAT or PrecoSAT. Another direction for the future work is to study improvements that affect the search path. In this line of work, a metric to compare search paths has to be defined in order to evaluate the results of the measurement correctly.

Acknowledgment The authors would like to thank Julian Stecklina, Hermann Härtig and Steffen Hölldobler, for many fruitful discussions during this work.

References

1. HPCToolkit. http://hpctoolkit.org/.
2. PAPI library. http://icl.cs.utk.edu/papi/.
3. C. Baldow, F. Gräter, S. Hölldobler, N. Manthey, M. Seelemann, P. Steinke, C. Wernhard, K. Winkler, and E. Zenker. HydraSAT 2009.3 solver description. SAT 2009 Competitive Event Booklet, http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf.
4. A. Biere. PrecoSAT system description. http://fmv.jku.at/precosat/preicosat-sc09.pdf.
5. J. Bonwick. The slab allocator: an object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference*, 1994.
6. G. Chu, A. Harwood, and P. J. Stuckey. Cache conscious data structures for boolean satisfiability solvers. *JSAT*, 6:99–120, 2009.
7. N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. 6th SAT, LNCS 2919*, 2004.
8. N. Eén and N. Sörensson. Minisat - a SAT solver with conflict-clause minimization. Poster - 8th SAT, 2005.
9. J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, 1996.
10. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *Design Automation Conference*, pages 530–535, 2001.
11. K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proc. 10th SAT, LNCS 4501*, 2007.
12. L. O. Ryan. Efficient algorithms for clause learning SAT solvers. Master's thesis, Simon Fraser University, Canada, 2004.
13. A. van Gelder. Generalizations of watched literals for backtracking search. In *Seventh International Symposium on AI and Mathematics*, 2002.
14. L. Zhang and S. Malik. Cache performance of SAT solvers: a case study for efficient implementation of algorithms. In *Proc. 6th SAT, LNCS 2919*, 2003.

A Problem Instances Used in the Measurement

Instances	**Solving Time** (seconds)	**Memory Usage** (KB)	**Satisfiable?**
ACG-10-5p0.cnf	169.062565	170968	no
AProVE09-20.cnf	1756.697786	203888	yes
UCG-15-5p0.cnf	476.773796	321968	no
UCG-20-5p1.cnf	1226.080625	474164	yes
UR-15-5p0.cnf	574.231887	338256	no
UTI-10-10p0.cnf	607.533968	388956	no
UTI-15-10p0.cnf	1027.736229	601984	no
blocks-4-ipc5-h22-unknown.cnf	570.543656	269496	no
cmu-bmc-longmult15.cnf	130.956184	26744	no
countbitswegner064.cnf	2585.413578	266988	no
eq.atree.braun.8.unsat.cnf	256.244014	30292	no
gss-16-s100.cnf	243.911243	38484	yes
gss-17-s100.cnf	357.822362	40828	yes
gss-20-s100.cnf	705.240074	51040	yes
gus-md5-07.cnf	121.45559	98880	no
gus-md5-09.cnf	820.299265	102548	no
manol-pipe-c10nidw_s.cnf	820.53928	625660	no
manol-pipe-c6bidw_i.cnf	257.124069	175516	no
manol-pipe-c6nidw_i.cnf	273.521094	181600	no
manol-pipe-g10id.cnf	812.70279	339084	no
manol-pipe-g10nid.cnf	2334.201878	570644	no
mizh-md5-47-3.cnf	814.090877	275084	yes
mizh-md5-47-4.cnf	668.657788	246784	yes
mizh-sha0-35-3.cnf	219.293705	153244	yes
ndhf_xits_20_SAT.cnf	393.776609	252364	yes
post-c32s-gcdm16-22.cnf	998.362393	257068	yes
q_query_3_L60_coli.sat.cnf	336.085004	240176	yes
q_query_3_L70_coli.sat.cnf	561.367083	285092	yes
q_query_3_l44_lambda.cnf	2024.402517	135872	no
q_query_3_l45_lambda.cnf	1767.374454	133816	no
q_query_3_l48_lambda.cnf	2068.153251	136716	no
rbcl_xits_06_UNSAT.cnf	415.165946	32620	no
schup-l2s-abp4-1-k31.cnf	448.384022	69608	no
schup-l2s-guid-1-k56.cnf	2439.468457	306456	no
schup-l2s-motst-2-k315.cnf	344.433525	561832	yes
simon-s02b-dp11u10.cnf	1189.330328	80564	no
uts-l05-ipc5-h27-unknown.cnf	353.102067	163212	no
uts-l06-ipc5-h31-unknown.cnf	1186.990182	284108	no
vmpc_24.cnf	659.017186	73148	yes
vmpc_26.cnf	539.513717	84628	yes