Belegarbeit

Memory Hierarchy Utilization of a SAT Solver

Norbert Manthey

March 31, 2010

Technische Universität Dresden
Fakultät Informatik

Betreuende Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig,
Prof. Dr. rer. nat. Steffen Hölldobler

Betreuende Mitarbeiter: Dipl.-Inf. Julian Stecklina,
Dipl.-Inf. Ari Saptawijaya

The project analyzes the hardware utilization of a SAT solver. The analysis is done using statistical profiling and tracing the following processor events: total cycles, resource stall cycles, level 2 cache hits and level 2 cache misses. The HPC Toolkit is used to perform the analysis on top of the PAPI library. The used benchmark is a part of the SAT competition 2009 application benchmark.

The analysis has additionally been done on two well known solver MiniSAT and PrecoSAT and unveiled similar utilization problems as in the project SAT solver. Its result is that the utilization can be increased for example by improving the clause representation, using the prefetch unit of the CPU and maintaining frequently used data structures lazily. The combination of the suggested improvements speed up the project SAT solver by 60%. The runtime improvement is mainly caused by fewer main memory and level 2 cache accesses.

Contents

1 Introduction

The importance of Satisfiability Testing (SAT) increased in recent years. The development of SAT solvers made them a powerful tool for solving problems of various fields very fast. There are even domains where SAT solvers are more powerful than specific problem solver. Some of the fields where SAT solvers can be applied are hardware and software verification, bioinformatics and attacking cryptographic algorithms [3].

Since SAT solver work in the domain of propositional logic, they can only handle a certain form of input, namely a formula containing clauses in conjunctive normal form (CNF). Any problem that can be represented in a propositional formula can be solved by a SAT solver. Thus, it first needs to be converted into the SAT domain and afterwards the gained result from the SAT solver needs to be transformed back in the original domain. Due to the fact that SAT is NP complete [10] any problem that can be solved in NP, can also be solved using a SAT solver.

The size of the encoded problems increased with the performance of the SAT solvers. Today an encoded problem can contain more than ten million variables and over 32 million clauses [3]. These sizes force SAT solvers to handle huge amounts of data. For fast maintenance fast access data structures need to be provided. Handling the huge problem size requires also a good algorithm. Many improvements have been introduced in recent years. Major algorithm improvements are the introduction of the CDCL algorithm [16] and the two-watched literal propagation [17]. These improvements have been compared by annual SAT-competitions and SAT-races [3].

In recent years the hardware changed from single core CPUs to multi-core CPUs and the growth of the CPU frequency almost stalled. Thus, the performance of sequential SAT solvers will not improve due to increasing frequency. The implementation of the solvers need to exploit the features of the underlying hardware to gain the best result. The interaction of modern SAT solvers, such as the winner of the last SAT competitions PrecoSAT [6] (2009) and the well known solver MiniSAT [13], and recent hardware has not yet been analyzed in detail. Modern high performance CPUs offer a huge range to increase the performance of applications like large caches, translation lookaside buffers (TLB), a prefetch unit or branch prediction units [2] [1]. Only the first component has been considered in analysis for SAT solver [9].

This work studies the memory hierarchy utilization of a CDCL-based SAT solver. Thus, only the branch prediction unit is excluded directly from the research. TLBs are only concerned for future work. The solver uses similar data structures as MiniSAT. The measurement is done using sample-based profiling by the HPC Toolkit [22]. During the measurement the following processor events are traced: total cycles, resource stall cycles, level 2 cache misses and level 2 cache hits. The used benchmark consists of 40 instances of the application track of the SAT competition 2009 with an overall runtime

of almost 10 hours. During the measurement read and write accesses to the clauses of the formula are traced to retrieve an access statistic.

Basically adapting the algorithm will gain more runtime improvement than adapting the implementation to the hardware. Still the implementation needs to be suited to modern hardware to achieve a reasonable performance. The major goal of this work is to improve the hardware utilization, especially the usage of the cache and the overall runtime of the solver. Measurements of PrecoSAT and MiniSAT unveiled that both systems solve the benchmark faster although their level 2 cache miss rate is lower and higher than the original implementation of the project solver (compare section 5.1). Thus this value does not necessarily indicate better hardware utilization.

After the analysis of the project solver improvement opportunities are suggested. These improvements include restructuring the clause representation and prefetching the clauses of watch lists and applying a more intelligent watch list maintenance. The listed improvements do not change the processing order of the algorithm. This property is very useful if the SAT solver is applied to new problem instances. The effect of this improvement remains for any input instance. The combination of these improvements improves the runtime of the SAT solver further, because their positive impacts sum up. The combination of the best improvements made the project SAT solver twice as fast.

The remaining chapters are structured as follows. Satisfiability Testing and the project SAT solvers are introduced in chapter 2. In chapter 3 the memory hierarchy of modern CPUs is described. Afterwards the measurement and useable frameworks are explained in chapter 4. The next chapter 5 analyzes the measured data and the implementation of the SAT solver. In chapter 6 improvements for higher hardware utilization and their results are presented. Finally, chapter 7 summarizes and concludes the work and gives an outlook on further improvements.

2 Satisfiability Testing

This chapter introduces the theory around the solver and the algorithm. It also includes implementation details and gives an overview of the components that are used in the project solver.

2.1 Propositional Logic

Satisfiability testing is done in the domain of propositional logic. Since SAT Solvers handle only one specific input form only the necessary terms are introduced.

2.1.1 Syntax

The input formula for a SAT solver is formulated in Conjunctive Normal Form (CNF).

Definition 1. *A propositional variable is a binary variable and is called atom.*

Definition 2. *A literal is either an atom a or a negated atom $\neg a$.*

Definition 3. *The polarity of a literal is negative if the literal is a negated atom. Otherwise it is positive.*

Definition 4. *A clause is a disjunction of literals without duplicates.*

Definition 5. *A formula in Conjunctive Normal Form is a conjunction of clauses.*

The solvers variables are represented by integers. Positive numbers refer to positive literals and negative numbers to negative literals. Clauses are written using square brackets like *C= [¬1,¬2,3].* The conjunction of clauses is notated using diamond brackets like *F=⟨ [2,¬1], [1,3]⟩*. The following formula will be used as an example during this chapter.

$$\text{F}=\langle\ [\neg 1,\ 2],\ [\neg 4,\ 5],\ [\neg 1,\ \neg 4,\ 6],\ [\neg 2,\ \neg 5,\ \neg 6],\ [1,\ 3]\ \rangle$$

The five clauses in the formula will be named according to their position in the formula from C_1 for the first clause to C_5 for the last clause.

2.1.2 Semantics

Solving a formula is the task of finding a mapping for each variable to a truth-value such that the application of this mapping to the formula evaluates to true. This mapping is called assignment.

Definition 6. *An assignment α to a set V of Boolean variables is a mapping $\alpha{:}V\rightarrow\{false, true\}$. It is represented by a sequence of literals. The literals in this sequence evaluate to true.*

Definition 7. *A literal is satisfied if it is an atom that is mapped to true or if it is a negated atom that is mapped to false.*

Definition 8. *A clause is satisfied if one of its literals is satisfied. An empty clause is unsatisfied.*

Definition 9. *A formula is satisfied if all its clauses are satisfied. An empty formula is always satisfied.*

The interpretation of a formula F by an assignment α is written as F$|\alpha$. It is applied using the following rules:

- All clauses that contain a satisfied literal are removed.
- All unsatisfied literals are removed from the remaining clauses.

A clause that contains only a single literal under the current assignment, is called *unit clause* or just *unit*. A *binary clause* is a clause with two literals left under the current assignment.

An assignment for the example formula is $\alpha = \{$ 1, 2, 3, 4, 5, 6$\}$. This assignment satisfies all positive literals of the formula. According to the above rule $C1$, C_2, C_4 and C_5 are removed. In clause C_3 all literals are removed, so that it becomes empty and according to Definition 8 and 9 this assignment does not satisfy the formula.

Definition 10. *If an assignment exists that evaluates the formula to true this formula is satisfiable. If there is no such assignment the formula is unsatisfiable.*

During the search the formula is not fixed. Some clauses are added to it. They are obtained by resolution. A clause that is the result of a resolution step is called *resolvent*. Due to Lemma 1 adding resolvents to the formula does not change the result of the search.

Definition 11. *Resolving two clauses leads to a new clause that contains all literals of both clauses. In case a literal occurs positive in the one clause and negative in the other one, all occurrences of the according variable are removed from the new clause. This removal rule is applied only once in a resolution step.*

Lemma 1. *The satisfiability of a formula does not change, if a resolvent is added. [7, p. 138]*

In the given example C_3 and C_4 can be resolved. The resolvent is [$\neg$1, $\neg$4, 6] $\otimes$ [$\neg$2, $\neg$5, $\neg$6] = [$\neg$1, $\neg$2, $\neg$4, $\neg$5] where $\otimes$ is the resolution operand.

2.2 Satisfiability Problem

The task of a SAT Solver is to show whether a given input formula is satisfiable. Most state of the art solvers also give a satisfying assignment if one exists. The naive approach checks all possible total assignments and stops if a satisfying assignment is found. This schema results in 2^n possible mappings for a formula with n variables. Modern SAT Solvers use partial assignments to avoid the huge arising number of total assignments.

Definition 12. *A partial assignment is an assignment that does not contain all variables of the given formula.*

Definition 13. *A variable that is not assigned by a partial assignment is undefined.*

The extension of a partial assignment α by a literal l will be written αl

2.3 Modern SAT Solving Procedures

Solving the satisfiability problem hes been done using a search tree. Next, the Davis Putnam Logeman Loveland (DPLL) algorithm has been introduced [11]. The Conflict Driven Clause Learning (CDCL) algorithm is an improvement of the DPLL algorithm. Both algorithms can be illustrated using a depth first search in a binary tree. The following sections describe the three algorithms.

2.3.1 Search Tree

The search tree is a binary tree. Each edge is labeled by a literal. The literals on the branch from a node to the root represent a partial assignment. The level of a node is the number of literals in its branch to the root node.

If a node does not have child nodes it can be expanded by a variable that is not on its branch. The one edge to the first child node is labeled with the positive atom and the other edge with the negative one. The most intuitive way of assigning variables is choosing the same variable for the same tree depth.

A branch can be closed if the formula contains a clause whose literals occur negated on this branch. If a branch contains all variables and cannot be closed, the literals on this branch are equal to a satisfying assignment for the formula. An incomplete search tree for the example formula is given in Figure 2.1.

In this search tree the fully expanded branches can be closed by the clauses C_3 and C_4. To illustrate the advantages of partial assignments it is shown that the tree expansion can stopped at a higher level. If the clause C=[¬1, ¬2, ¬4, ¬5] is added to the formula, the expansion of the very left branch is stopped at depth four, because this clause is unsatisfied with this partial assignment. This example shows that clauses cut the tree. A clause with n literals in a formula F with the variables V cuts $2^{|V|-n}$ leafs of the tree.

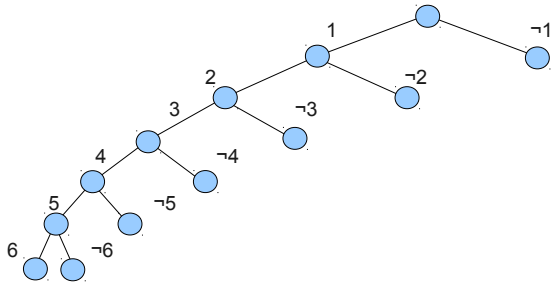


Figure 2.1: Extendable Search Tree.

2.3.2 Davis Putnam Logeman Loveland

The DPLL algorithm is explained in algorithm 1 using a recursive version.

Given a formula F a set of rules is checked. If the formula is empty it is satisfied under the current partial assignment (line 1). If one of the contained clauses is empty the formula is unsatisfied under the assignment (line 4). The *unit rule* (line 7) is the most important rule. It checks whether a clause of the formula is unit and thus its literal must be added to the assignment to satisfy the clause.

Definition 14. *A clause that is applicable for the unit rule is called reason.*

The *pure literal rule* (line 10) checks whether a literal occurs only in one polarity. If this rule applies the literal found can be set to satisfy clauses and to avoid obtaining empty clauses. In modern solvers this rule is not implemented, because its gained result is not worth the time to check whether there are pure literals on common problems. If none of the above rules is applicable a decision (line 13) is made by choosing a literal p and adding it to the partial assignment. This step is called *splitting rule*. If the try fails the variable has to be mapped to the other polarity (line 16). This step is called *chronological backtracking*, because the last decision is undone and the search leaves the current depth of the search tree.

The search tree in Figure 2.2 shows a DPLL search where the splitting rule leads to an unsatisfying assignment. The next step of the procedure is undoing the last decision and proceeding with the branch 1,2,3,¬4. Vertical arrows represent the application of the unit rule.

2.3.3 Conflict Driven Clause Learning

The CDCL algorithm is an extension of the DPLL algorithm. Instead of chronological backtracking, a mechanism called *backjumping* is used. Multiple decisions are undone in one backjumping step. Furthermore the order of the variables on the branches changes.

Algorithm 1 DPLL(F, α)

```
1: if F|α empty then
2:    return SATISFIABLE
3: end if
4: if F|α contains an empty clause then
5:    return UNSATISFIABLE
6: end if
7: if F|α contains an unit clause [p] then
8:    return DPLL(F|αp)
9: end if
10: if F|α contains a pure literal p then
11:    return DPLL(F|αp)
12: end if
13: if DPLL(F|αp) = SATISFIABLE then
14:    return SATISFIABLE
15: else
16:    return DPLL(F|α¬p)
17: end if
```

These two facts make it difficult to give a recursive version of this algorithm. To make the algorithm more comparable to the implementation of a SAT solver it is given in an iterative version. The DPLL algorithm without the pure literal rule can be simulated by the CDCL algorithm (compare subsection 2.4.6). A correctness proof of the presented CDCL algorithm would be very similar to the one given in [14].

The given CDCL algorithm 2 introduce the new variable *current_level* (line 1). It represents the number of branches from the root of the search tree to the current point of the search. For each variable the *level* (line 2) has to be stored. This is done when the assignment α is expanded with the according literal. The variable $conflict$ (line 2) indicates whether there is an unsatisfied clause under the current partial assignment.

Definition 15. *A conflict clause is a clause that is unsatisfied under the current partial assignment.*

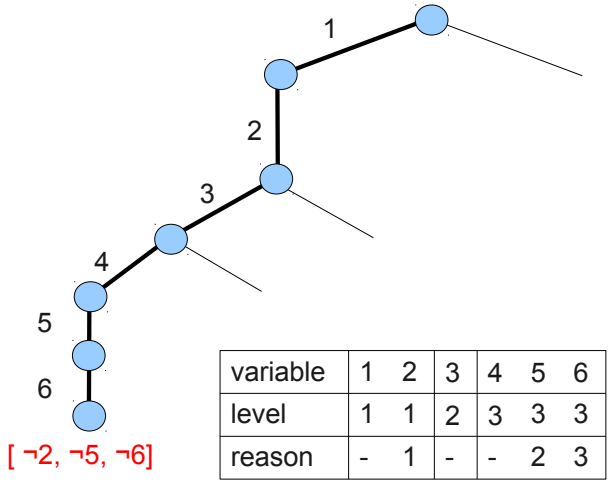
The procedure starts with an empty assignment (line 1). The next steps are repeated until a solution is found (line 3: the current assignment is propagated (line 4). The propagation includes the unit step of the DPLL algorithm. If an unit clause is found the assignment is extended and propagation proceeds.

If the propagation does not lead to a conflict (line 5) a new decision has to be made (line 6). If no decision is possible (line 7), because all variables are assigned and there has been no conflict, the current assignment satisfies the formula (line 8). If the assignment is partial the current node in the search tree can be expanded (line 10) and the variable *decision* contains the literal that extends the assignment (line 11).

If there has been a conflict its aftermath has to be checked (line 13). If the conflict occurs at the root of the search tree the formula is not satisfiable (line 15). An example

Algorithm 2 CDCL(F)

```
1: α ← {}, current_level ← 0;
2: conflict ← 0; decision ←NO_LIT; level[|V|];
3: while true do
4:    conflict ← propagate(F, α);
5:    if conflict = 0 then
6:       decision ← pick_literal();
7:       if no decision possible then
8:          return SATISFIABLE;
9:       end if
10:      current_level ← current_level + 1;
11:      α ← αdecision;
12:      level[decision] ← current_level;
13:   else
14:      if level = 0 then
15:         return UNSATISFIABLE
16:      end if
17:      clause ←analyze(conflict);
18:      literal ← single literal from current level of clause;
19:      current_level ← max{level[x] : x ∈ clause − {literal}};
20:      backtrack(α, current_level);
21:      α ← αliteral;
22:      level[literal] ← current_level;
23:      F ← F ∪ clause;
24:   end if
25: end while
```



variable	1	2	3	4	5	6
level	1	1	2	3	3	3
reason	-	1	-	-	2	3

Figure 2.2: DPLL Search Tree With Conflict.

is the formula $\langle [\neg 1\ ], [\ 1\ ] \rangle$. The unit rule is applied at the root of the tree and results in a conflict. Otherwise the conflict is analyzed and a new clause is obtained. This analysis is described in chapter 2.4.6. Properties of this clause are that it contains only one literal of the current level (line 18) [16]. Due to the fact that it is a resolvent of the formula it can be added to the formula (line 23). An example, which illustrates this property, is given in Figure 2.5. The level where the search continues is the second highest of the obtained clause (line 19) because on that level the obtained *literal* becomes applicable to the unit rule (line 21). Backjumping is done exactly to this level (line 20). Next the assignment is propagated again and the next value of the conflict variable has to be evaluated. An example for such a backtracking step is given in Figure 2.3.

The white nodes are the ones that have been accessed before the conflict. The analysis of the conflict led to a backjumping to level 1 where the unit rule became applicable again. The search proceeds at the lowest filled node with the branch 1,2, ¬4.

The presented algorithm represents the basic CDCL search method. A state of the art solver includes more methods. At some point in the search the whole search tree is thrown away and the search starts from scratch, just with the advantage of the learnt clauses. This method is called *restart* and tries to recompense wrong decisions that have been made in low levels of the search tree. Early wrong decisions are very expensive, because the search is a depth-first search and thus leaving the entered part of the search tree needs to process lots of nodes. Restarts help escaping these parts.

Another problem of the presented algorithm is its memory usage due to added learnt clauses. A part of the learned lemmas has to be deleted during search. This approach is called *removal*. The removal is important, because propagation slows down if too many clauses have to be processed. Heuristics for this two strategies are introduced in

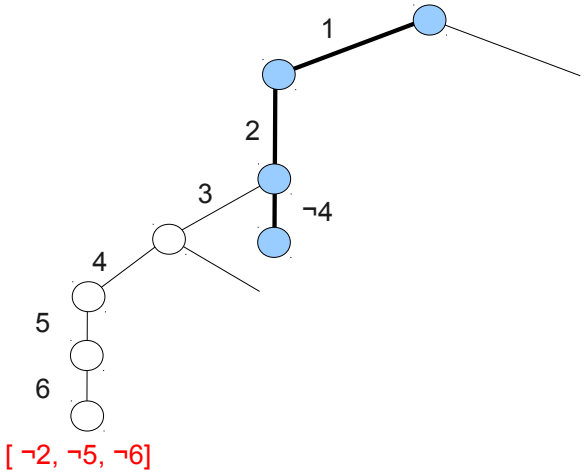


Figure 2.3: CDCL Backjumping After a Conflict.

subsection 2.4.8 and subsection 2.4.9.

2.4 The Project Solver

This section describes the composition of the used SAT solver. The first two subsections 2.4.1 and 2.4.2 describe used data structures and the way they are used during search. The remaining subsections describe the solver and its components.

2.4.1 Data Structures

A clause is represented by an array of literals, its size and an activity, which is a floating-point number. A clause is never copied if it is used in several modules of the solver. Only the address of the clause is shared among the components, because it is unique and can identify the clause.

Arrays and vectors are collections, whose elements can be accessed and changed. The elements are stored in an array. A vector can be enlarged explicitly or by adding elements to it. Removing elements from it can be done by preserving the order of all elements or by swapping the element that should be deleted with the last one and afterwards removing the new last element of the vector. The latter method is advantageous, because only two elements have to be processed. A stack is a special vector. Removing or adding elements is only allowed at its end. The access of its elements is not restricted. The used queue is double ended. All of its elements can be changed and adding and removing elements can be done at the front and the end. The vector, stack, double ended queue and linked list (will be called *list* in the sequel) implementations are taken from the C++ Standard

Template Library [23].

The priority queue is implemented via a binary heap. All elements can be accessed via an index, the element with the highest priority is returned fast and all elements can be inserted according to the order of their reference value. The heap is stored and managed using a vector. All elements of the heap need to be combined with a reference value that is used to order the elements.

In theory the clauses in the formula do not have an order, because the formula is represented as a set of clauses. The order is introduced by implementing this set as a vector. This order influences the search process, because the search iterates systematically over all collections to perform its work. The elements of the formula vector are pointer to clauses.

2.4.2 Elements of the Search

As presented in the algorithm 2 important data structures are the **assignment**, the **formula** and the **current level** of the search. Additionally to the assignment a **trail** is introduced that stores the assigned literals according to the order of their assignment time. It represents the current branch of the search tree. Per variable the **level** of its assignment is stored using an array, which is indexed by the variable. The clauses that are reason for a variable mapping are stored as pointer in an array named **reason**.

The current state of the search is represented by this data. The search object stores this state. The CDCL algorithm is implemented in the search method of the object and connects the components.

2.4.3 Implementation Details

The whole solver is implemented in C++. Most of the implemented data structures and algorithms are taken from HydraSAT [5], which has been ranked in the middle field of SAT competition 2009 [3]. Since HydraSATs implementation is close to MiniSAT [13] the implemented data structures are also similar. The major difference between the two solvers is the implemented removal heuristic (compare 2.4.9).

The project solver is component based. The components, which are described in section 2.4.4, can be replaced without another compilation using command-line parameter and the parameters for the algorithms can be set. This enables an easy exchange of procedures to measure several algorithms for a specific part of the search without implementing another solver. The used data structures can only be chosen at compile time. Choosing them at runtime introduces too much overhead.

The lines of code of the basic version of the solver are almost 3600 lines C++ code and about 800 lines Ansi C code. The basic version includes only the components described in the following subsections and no hardware utilization improvements. The solver is compiled to a 64 bit binary using the GNU Compiler version 4.1.2 with the highest optimization level -O3. Thus all data pointers use 8 bytes. The data types *literal_t* and *variable_t* are represented using unsigned integers of 32 bits. Floating point numbers are stored using the 32 bit single precision float data type.

2.4.4 Solver Components

The search can be split into several components. This modular implementation introduces the ability to exchange routines without much effort, because only implemented objects need to be exchanged.

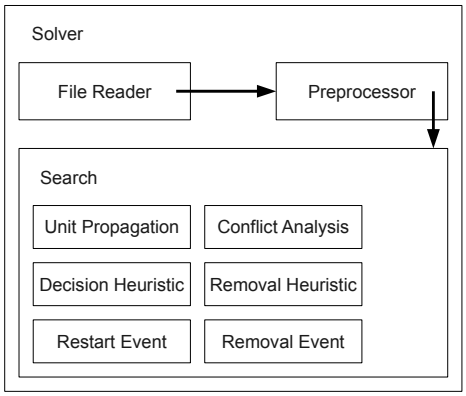


Figure 2.4: Components of the Project Solver.

The controlling object is called solver. It reads the input formula via the *file reader* and tries to simplify it using a *preprocessor* . Afterwards the search, whose task is split into several components, is called.

- The **unit propagation** checks the formula under the current partial assignment for units and conflicts
- The **conflict analysis** returns a resolvent given the current state of the search and a conflict clause.
- The **decision heuristic** picks a new decision literal when the splitting rule is applied.
- The **removal heuristic** keeps track of the added learnt clauses and chooses clauses to be removed again.
- The **restart event heuristic** schedules restarts.
- The **removal event heuristic** schedules removals.

For a straightforward replacement of components all communications between single components have been reduced to communication between the search and the specific component. In Figure 2.4 it is shown how the solver handles the input formula. The formula is read by the *file reader* and then passed to the preprocessor. The preprocessed formula is passed to the search, which applies the CDCL algorithm using its components. There is no direct communication between the search components.

The file reader and the preprocessor do not influence the search process much. The file reader runs only once at the beginning and parses the input file. It stores the formula in main memory. The search can only be controlled by changing the order of the clauses or placing the clauses to a given position in memory. The preprocessor works as the one implemented in MiniSAT 1.4 [12]. This component tries to simplify the input formula by reducing the amount of clauses. The preprocessor influences the search only once before the search. The details of its algorithm are not discussed in this work.

2.4.5 Unit Propagation

The unit propagation (UP) does the major work of solver. Its task is to propagate the current partial assignment through all clauses, checking them for conflicts and applying the unit rule of the DPLL algorithm. If the propagation finds a conflict it stops immediately and returns the conflict clause. It is the only component that changes the current state of the search. Therefore, backtracking is also implemented in this component.

For the propagation only the current level is interesting, because all the previous assignments have been already propagated. Therefore, the propagation object has a *unit queue* of literals that have to be propagated at this level. At the beginning of a propagation this queue contains only the current decision literal.

The propagation through the clauses is not done via visiting all clauses and checking their state. Instead the *two-watched-literal schema*, which has been introduced in the Chaff solver [17], is used to visit only clauses that can become unit or conflict during the propagation of the current literal. Therefore, for each literal set of clauses is stored in a structure, which is called *watch list*. This literal is watched in these clauses.

The watch list contains clauses with the complement literal, because this clause becomes smaller under an assignment that sets the literal to true. The literals that are watched in a clause are called **watched literals**. The clause that contains two watched literals will be called **watched clause** for these two literals, because it is in the set of clauses that will be visited if these literals are propagated. If the assignment is applied the literals are not really removed from the clause.

Given the assignment $\alpha = \{2\}$ and the clause C=[$\neg$**2,** $\neg$**5,** $\neg$6] the literal 2 has to be propagated. The clause is watched by literal 2 and literal 5. The propagation accesses the clause and checks it for a satisfied literal or an unassigned literal ignoring the other watched one. The propagation finds literal $\neg$6, which will be watched now. The clause is moved from the list of literal 2 to the list of literal 6 and now contains the following literals C=[$\neg$2, $\neg$**5,** $\neg$**6**] . If a satisfied literal is found, the clause will be watched by this literal as well as in case the literal is undefined.

Assume the next assignment looks like $\alpha = \{2, 5\}$. The clause is accessed again and there is no other unassigned literal. Therefore, the other watched literal has to be propagated because the clause can only be satisfied by this literal. The assignment will be extended to $\alpha = \{2, 5, \neg 6\}$ and the clause will be stored as reason for the assignment of the literal $\neg$6.

The last remaining case occurs if the other watched literal is also assigned but not yet propagated. This effect is caused by handling the literals of the unit queue sequentially.

Assuming the assignment is $\alpha = \{2, 5, 6\}$ and variable 2 has already been propagated and variable 5 is propagated at the moment. Then extending the assignment with $\neg 6$ fails, because the variable 6 is already assigned. The clause is a conflict clause in this case.

If the unit rule has to be applied the according literal is added to the assignment and to the trail. The literal that has to be set to true is enqueued to the unit queue. After the propagation of one literal finished the next literal is dequeued and propagated. This procedure is repeated until the queue is empty or a conflict is found. Other conflicts can be found and the search would proceed in another part of the search tree. Dequeuing literals from the unit queue is done according to the breadth-first search.

Algorithm 3 propagate(F, α)

```
1: while queue_not_empty() do
2:    lit =queue_dequeue();
3:    conflict =propagate_binary(lit);
4:    if conflict = 0 then
5:       conflict =propagate_long(lit);
6:    end if
7:    if conflict ≠ 0 then
8:       return conflict;
9:    end if
10: end while
11: return 0;
```

The implementation handles binary clauses in a special way, because they are implications and their propagation is easier to execute than the one for long clauses. Algorithm 3 shows how the propagation is split. As long as there are literals to propagate (line 1) the next literal *lit* is dequeued and propagated (line 2). The given procedure handles binary clauses (line 3) before it propagates literal *lit* through the rest of the formula (line 5) but only if no conflict is found before (line 4). There is a discussion whether it is useful to have binary conflict clauses or whether one should look for a long conflict if a short one has been found [6].

The special treatment is implemented as follows. The watch list of a literal for binary clauses does not only store the pointer to the clauses but also the other literal so that the check of the other literal becomes very cheap. The spatial overhead of this method is that every literals has to store a watch list for long clauses and another one for binary clauses.

The maintenance of watch lists can only be done if the UP gets to know the newly learned clauses. For adding and removing clauses some methods are provided. If a learned clause is added this clause is checked whether it is unit under the current assignment. In this case the according literal is enqueued to the unit queue.

The task of the backtracking is undoing all assignments that have been made at a higher level than the current one. All literals with a higher level are removed from the trail, their assignment is set to undefined and their reason and level are reset to

undefined. The undefined variables are passed back to the search just in case some other component wants to know which variables are assigned.

2.4.6 Conflict Analysis

The conflict analysis (AN) analyzes the conflict in the current state of the search. It is sufficient to return a clause of all negated decision literals. This approach results in a DPLL like search with chronological backtracking. To achieve a non chronological back jumping in the search one needs to return a clause that is unit under a part of the current partial assignment.

In the conflict clause all literals are unsatisfied (Definition 15). The literals of this clause are either set by decision or by the unit rule and have a reason (compare section 14). If they have a reason they occur complementary in it. This fact can be used to resolve the conflict clause and the reason clauses for the literals of the current level.

The procedure traverses the trail from its back to the front. The reason of the current literal is resolved with the last resolvent. Initially the conflict clause is this resolvent. The procedure is stopped if the resolvent contains only a single literal with the current level. This approach has been introduced as first UIP learning in [16].

The gained clause is called learnt clause. It contains only unsatisfied literals in the current search state. The learnt clause becomes unit if the partial assignment is backtracked until one literal is undefined again. This literal will be the one from the current level, because it has the highest level. If the level is reduced lower than the second highest level of the literals of the learnt clause, this clause is no unit clause any more. Therefore the second highest level is chosen.

The learnt clause can be minimized further by resolving it with the reasons of its literals. If the new resolvent is shorter than the old one and the number of different levels of the literals is not higher the new clause is kept. Otherwise the minimization is stopped. The backjump level is calculated as before. Experiments showed that minimizing the learnt clause result in less memory consumption and a faster search [21].

Figure 2.5 shows the analysis given the search state of Figure 2.2 and the example formula F the analysis of the conflict including minimization. Only step 1 and 2 belong to the analysis. The result of step 2 satisfies already the criteria that the clause should contain only one literal of the current level. These literals are the bold printed ones in the result column. The other bold printed literals refer to the current literal of the step. The backjump level of this analysis is 1. Due to backtracking the literals 3, 4, 5, 6 are undefined and the clause [$\neg 1$, $\neg \mathbf{4}$] becomes unit under the new partial assignment.

2.4.7 Decision Heuristic

A very important part of SAT solvers is the decision heuristic (DH). It chooses the search path. If it always chooses the right path, SAT problems could be solved in sub exponential complexity. Modern heuristics seem to be close to right choices for real-life problems, because most industrial SAT problems are solved in a tiny part of the

variable	1	2	3	4	5	6
level	1	1	2	3	3	3
reason	-	1	-	-	2	3

step	current literal	current resolvent	reason	result
1	6	[¬2, ¬5, **¬6**]	[¬1, ¬4, **6**]	[¬1, ¬2, **¬4, ¬5**]
2	5	[¬1, ¬2, ¬4, **¬5**]	[¬4, **5**]	[¬1, ¬2, **¬4**]
3	2	[¬1, **¬2**, ¬4]	[¬1,**2**]	[¬1, **¬4**]

Figure 2.5: Conflict Analysis Example.

theoretical worst case execution time, if one uses the number of decisions in the whole search process as a metric.

The used decision heuristic follows the principles of the Variable State Independent Decay Sum (VSIDS). An activity using a single precision floating point number is stored per variable and initialized with 0. This activity is increased by an increase factor, if this variable was involved in the resolution process to obtain a learnt clause. The increase factor increases, if another conflict occurs. Thus, every new conflict increases the importance of all literals that have been used for recent conflict analysis.

If a decision literal should be chosen the unassigned variable v with the highest activity is chosen and the negated variable is returned. The aim is to access recently learnt clauses again and use them to create even smaller learnt clauses at the next conflict. The order of the variables is managed using a priority queue.

After 1000 decisions a random decision is made. The heuristic tries to find an unassigned variable randomly. If this attempt fails 10 times a deterministic choice is done.

If all variables are assigned NO_LIT is returned. It indicates that no more literal can be set to true.

2.4.8 Restart Event Heuristic

Scheduling restarts is done using an event heuristic (RH) working according to a geometric schema. The first event is triggered after 100 conflicts and the increment factor is 1.5. The calculation of the next event point is

$$limit(n) = (limit(n-1) - limit(n-2)) * factor + totally\_made\_conflicts$$

with $limit(0) = 0$ and $limit(1) = 100$. If a limit is reached a restart is only scheduled if no conflict occurred in the recent propagation. Due to this fact the totally_made_-conflicts value occurs in the calculation. If no conflict occurred when reaching the limit its value is exactly the same as $limit(n-1)$. A restart is done by backjumping to level 0 with all its consequences. Thus only variables with reason clauses that are unit are kept.

Restarts are very important in the search, because they can undo early wrong decision. The search process starts initially with almost no information about the problem to solve. After a while the activities of the variables in the decision heuristic are increased. If

the search had started with the these activities it would have done completely different decision. These decision could lead faster to the satisfying assignment. Since the search is a depth first search early decisions are hard to undo without restarts.

2.4.9 Removal Heuristic

Keeping the number of clauses reasonable is very important for the performance of the solver. Therefore, at some point some learned clauses have to be removed again. Every conflict creates another clause and after a large number of conflicts UP is much slower. The solver will run out of memory or will propagate slowly.

The aim of the removal is to keep clauses that speed up the search process and cut off big parts of the search tree and throw away useless ones. Due to the fact that a clause with n literals removes $2^{|V|-n}$ total assignments short clauses are kept and long ones are thrown away. A removal is scheduled immediately after a restart by the **removal heuristic (RM)**. The solver removes

- all clauses with more than 6 literals.
- the oldest 55% of the remaining clauses with more than two literals.

For managing these lists the component needs to be notified, if clauses are added.

3 Memory Hierarchy

Since SAT solvers access lots of data the features of the underlying hardware needs to be utilized as well as possible to keep the runtime of the solver small. This chapter introduces the memory hierarchy of modern computers. The focus of the work is to analyze the utilization of this hierarchy.

3.1 Memory Performance

Figure 3.1 shows the latency for working with various amounts of data. The access time of memory increases with the size of the data that is processed.

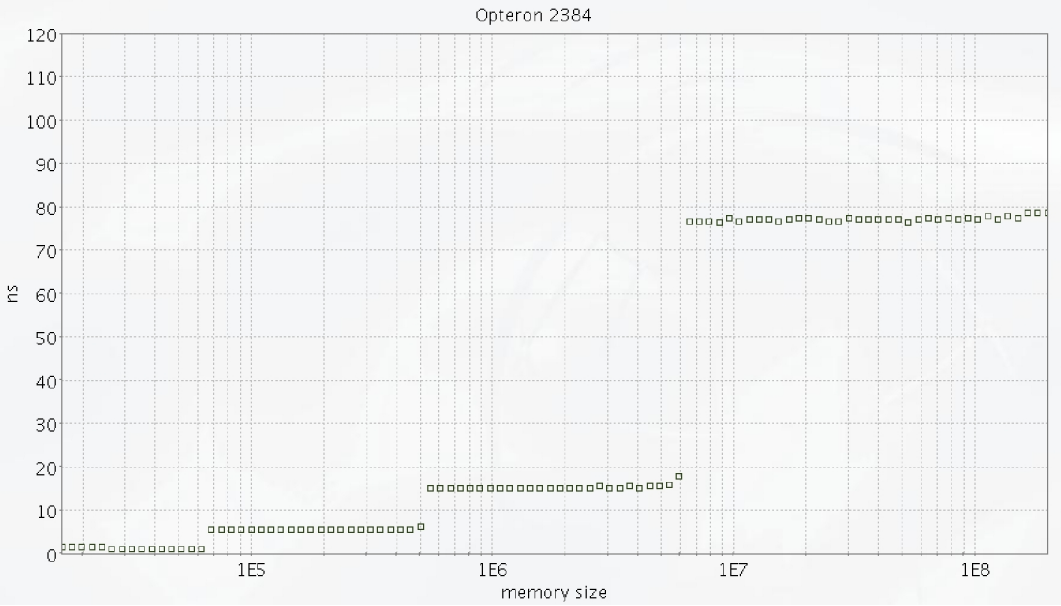


Figure 3.1: Memory Latency for AMD Opteron 2384.

The steps of the curve are the result of the memory hierarchy. Small data set can be hold in caches that are accessed fast. The processor AMD Opteron 2384 that has been used for this measurement has three cache levels. The lower three levels of the curve refer to cache accesses and the last level shows accesses in main memory. The latency of the access increases with the level in the hierarchy.

Unfortunately the memory latency does not keep pace with improvements in raw computing power as shown in Figure 3.2 [15, p. 374]. The absolute time to get data from main memory and the time for a CPU cycle has been equal about 1980. The

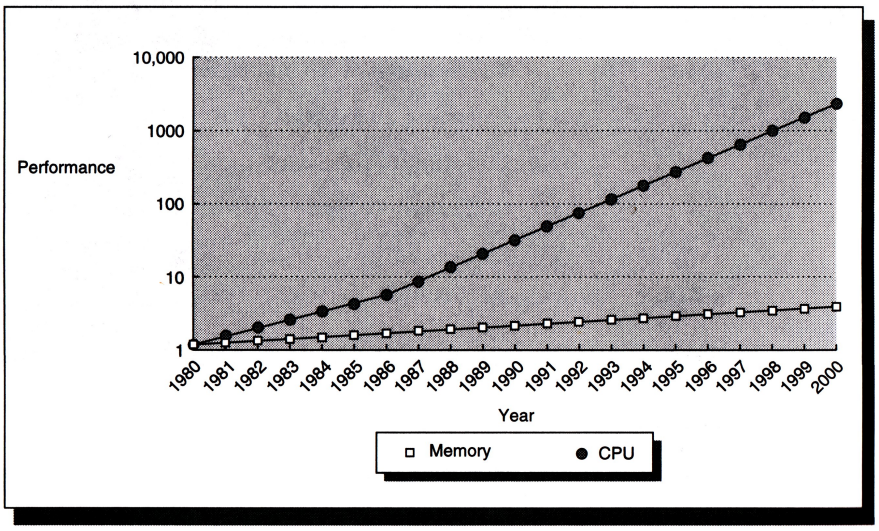


Figure 3.2: Comparison of CPU and RAM Latency.

improvement factor of the two developments has been and is still different. Thus the time to access data decreases only by nine percent every year whereas next year's CPU can execute 160% of the operations the current one is able to execute in a certain amount of time. The arising gap between the two hardware components increases by 50% per year [15].

Definition 16. *The memory footprint of a program is the size of memory that is touched during the whole execution of a program.*

Comparing the memory footprint in Figure 3.3 to the memory performance in Figure 3.1 the solver seems to work at the lowest memory performance. This behavior is controlled by the memory access pattern. Thus the memory footprint of an application alone does not determine the hardware utilization. The memory access pattern control in which layer of the memory hierarchy an application buffers its currently accessed data.

Memory	Average	Minimum	Maximum
MB	226.25	25.38	626.04

Figure 3.3: Memory Usage of Project Solver.

3.2 Introducing Caches

The fact that RAM is much slower than the CPU led hardware vendors introducing caches. A **cache** is a small and fast storage that buffers accesses to main memory. Figure 3.1 shows the dependencies between memory size and memory access time. It shows that the number of cycles to access a hierarchy level increases with the size that

the according level can store. Table 3.4 shows a possibility of adding caches between the CPU and main memory. The CPU that has been used for the SAT solver analysis implements two cache level. The small level 1 (L1) cache is separated into a storage for instructions and a storage for data. The level 2 (L2) cache stores both instructions and data. Accessing main memory takes 15 times longer than accessing L2 cache.

Memory	Size	Access Cycles
Main Memory	2 GB	~240
L2 Cache	1 MB	~14
L1 Cache	64 KB + 64 KB	3
Register	16 * 8 B	1

Table 3.1: Properties of Memory Hierarchy for AMD Opteron 285.

3.3 Cache Foundations

Foundations of the cache architecture are the assumptions of temporal and spatial locality [15, p. 41]. Temporal locality means that currently accessed data will be accessed again in near future with a high probability. Holding this data in fast memory is likely to improve the programs performance. Spatial locality means that data, which is stored next to previous accessed data, will be accessed with a high probability. Therefore, this data should be stored in the cache as well. To achieve this goal more than a single word is buffered per memory access.

Buffering data that is likely to be accessed is the task of the *prefetcher*, which is an unit of the CPU. The prefetcher stores data with high access probability in the cache, for example by recognizing linear memory accesses. It works while the CPU proceeds executing the program. The goal of the prefetcher is to reduce the latency of the programs memory accesses.

3.4 Cache Implementation

Caches are implemented as associative memory. They are content addressable with respect to the addresses. If the CPU needs some data from a certain address it checks the caches for this data in parallel to getting the data from main memory as it is shown in Figure 3.4. The first response is used and the CPU continues executing the program. Thus, if the data is fetched from main memory in both L1 and L2 cache this access resulted in a cache miss. Any fetch from a hierarchy level is caused by misses on the lower levels. The other way around a hit in a certain hierarchy level does not result in an event of the higher levels.

Caches are split in *tag memory* and *data memory*. The tag memory stores the address of the according cache line of the data memory. If a certain address is accessed the tag is

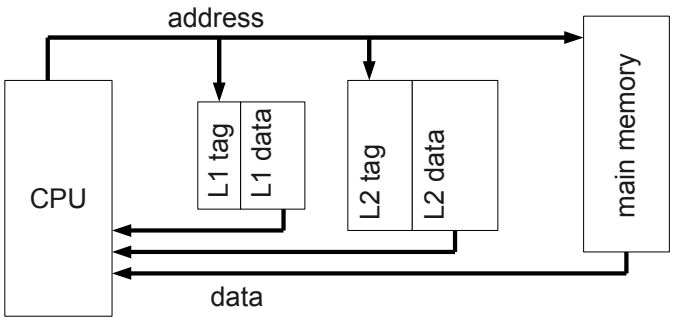


Figure 3.4: Accessing Data in the Memory Hierarchy.

compared to it. If the comparison succeeds the desired data is buffered in the according cache line.

There are several parameters for the implementation of caches. They are explained in subsection 3.4.1. The remaining subsections describe how caches can be organized.

3.4.1 Cache Parameter

The following set of parameters describes the cache implementation. The *cache size* gives the amount of memory that can be stored in the cache. This memory is organized in *cache lines*. If a piece of data is cached a whole line is stored in the cache. The *cache line size* is the number of bytes that are handled as one block. The number of cache lines can be determined by dividing the cache size by the line size.

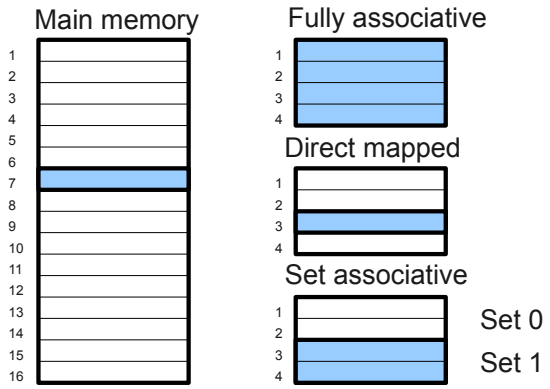


Figure 3.5: Data Organization in Caches.

There are three ways of organizing memory lines in the cache as shown in Figure 3.5. The marked line 7 in main memory is stored in the marked lines in the caches according

to the given schema. A fully associative cache stores the data in any line, a direct mapped cache stores a memory line to one fixed line. Set associative caches determine a set of cache lines by calculating memory line number modulo number of cache lines sets [15, p. 376].

3.4.2 Direct Mapped Cache

There is only one fixed cache line per memory line. This location is determined by calculating the modulo of its line number to the number of cache lines. In this case only one tag needs to be compared. On the other hand the old cache line needs to be evicted from cache. The line to evict is strictly determined.

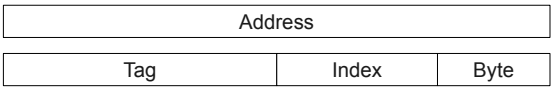


Figure 3.6: Partition of Addresses for Direct Mapped and N-way Set Associative Caches.

The address is split into a *tag*, an *index* and a *byte* part as shown in Figure 3.6. The index determines the line to choose. It is equal for all memory lines that are stored in the same cache line. Therefore, there is no need to store it in the cache. The number of index bits is the logarithm to the basis 2 of the number of cache lines. The index is the result of the application of number of memory line to store modulo number of available cache lines [15, p. 376]. The byte part is used to select the desired byte of the cache line. The number of used bits is the logarithm to the basis 2 of the cache line size. All remaining bits are called the tag. This tag is compared to the tag memory of the matching cache line. It is the only part of the address that needs to be stored in the tag memory.

It is not guaranteed that the whole cache is used, because some lines are maybe never used due to the memory layout of the running application, which maybe uses only specific parts of main memory.

3.4.3 Fully Associative Cache

This problem of being forced to evict a cache line is solved by fully associative caches. A memory line can be stored in any of the cache lines. The line that needs to be replaced is chosen from the whole cache.

The address is only split into a tag and a byte part as shown in Figure 3.7. The index is missing, because there is no restriction to use a certain line. A negative aspect of this schema is that the architecture needs to compare all tags to the address of the desired data. The number of bits to compare for searching a cache line is the largest among these schemes.

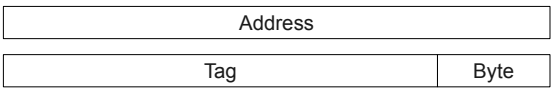


Figure 3.7: Partition of Memory Address for Fully Associative Caches.

3.4.4 N-Way Set Associative Cache

To reduce the number of tag comparisons and keep the replacement strategy the N-way set associative cache divides the cache into N sets. The number of memory lines to store determines the set of cache lines where it is buffered. The number of the set is calculated by the module of the number of the memory line to the number of cache line sets. The lines in each set are chosen as in the fully associative schema.

The address is split into tag, index and byte, which is also the address in Figure 3.6. Tag and byte part have the same function as in the fully associative schema. The index part does not choose single lines like but sets of lines. Consequently the number of bits is the logarithm to the basis 2 of the number of sets. This schema is a trade of between cache usage and accessing speed.

3.5 Cache Misses and Improvements

Every program has to access memory. A **cache hit** is a data access that accesses data that is stored in the cache. A **cache miss** is the complement. As previously shown in Figure 3.1 the performance of the program depends on the number of cache misses and cache hits. Thus, the higher levels in the figure are results to cache misses in the lower levels. Hitting the lowest level results in the smallest latency.

The following subsections describe the influence of caches. The three types of cache misses are explained in detail and some solutions to avoid them are given.

3.5.1 Performance Influence of Caches

The CPU time of a program without memory accesses can be calculated using the formula

$$CPUtime = IC \cdot CPI \cdot CycleTime$$

where IC is the number of executed instructions, CPI is the number of cycles that a single instruction needs in average to be executed. $CycleTime$ is the time that is needed for a single clock cycle. If memory is taken into account the memory accesses have to be calculated as well. The given formula [15, p. 386] considers only one cache level. This approach is not a problem for analyzing a SAT solver, because the penalty for the last hierarchy level is orders of magnitude higher than the one of lower levels. As shown in the analysis chapter 5 the number of cache misses is almost as high as the number of

cache hits. The number of cache hits does not influence the runtime much, because their penalty is much lower.

$$CPUtime = (IC \cdot CPI + MemoryAccesses \cdot MissRate \cdot MissPenalty) \cdot CycleTime$$

$MemoryAccesses$ is the number of total memory accesses of the execution. The $MissRate$ is the ratio of cache misses compared to the overall memory accesses and the $MissPenalty$ gives the number of cycles the CPU has to wait until the data is read from main memory.

Concerning the high latency of main memory the numbers of memory accesses and the miss rate have to be optimized to improve the runtime of the program.

Reducing the miss rate works differently for the three existing types of cache misses. They are explained in the following subsections.

3.5.2 Compulsory Cache Miss

The very first access to a block can never be in the cache except the prefetch unit buffers the data in the cache before. Thus this kind of miss can only be reduced by touching less lines of main memory or using the prefetch unit.

3.5.3 Capacity Cache Miss

Caches are usually significantly smaller than main memory. Accessing memory that has been evicted from the cache, because the cache was not able to buffer all the data, results in a capacity miss. To reduce these misses it has to be ensured that only the frequently accessed data is stored in the cache and is not removed.

The used data could be separated into an often used part and the remaining data. Since accessing one element loads another element of the same cache line into the cache frequently used data has to be stored compactly. Accessing seldom used data would result in a cache miss and thus important data is removed from the cache. Since these accesses are rare it should be beneficial to differentiate between frequently used and other data. For the decision which data is frequently used some additional instructions have to be executed.

3.5.4 Conflict Cache Miss

In case of set associative or direct mapped caches the replacement of a cache line is called a conflict miss, if its reasons is that too many lines need to be placed in a certain set of cache lines.

An improvement can be achieved by using certain areas of main memory to force an equally mapping of cache lines to main memory. This approach is called *cache coloring*.

Assuming a restriction of 2 GB main memory the solver would quickly run into memory capacity issues. In case of an N way set associative cache one needs to reserve the Nth part of main memory for a single data structure. Most of the used data structures are much smaller.

3.6 Non Data Caches

Recent CPUs also implement other caches. The main type is the **Translation Look aside buffer (TLB)** [15, p. 445]. Due to virtual memory [15, p. 439] the used virtual addresses in the program have to be translated to physical ones. The translation is done using a hierarchy of page tables [15, p. 451]. The process results in additional memory accesses. Thus its latency is comparable to the one of L2 cache misses [15, p. 447]. To avoid redoing the same translation again TLBs store the result of last translations to reuse them and avoiding redoing the same translation. This work does not focus on TLB utilization. In section 7.2 the impact of TLB misses is briefly discussed.

4 Measurements

There are several tools for obtaining various data of a running program, for example the number of cache misses, the runtime of a function, the number of calls of a function or the memory accesses of a single instruction. In the following sections 4.1 some of these tools are discussed. In the remaining sections the used benchmark is described and its results are discussed.

4.1 Performance Measurement Tools

The most common framework to do runtime analysis is Valgrind [19]. For cache simulation Callgrind [4] can be chosen, which is an extension for this framework. Callgrind is described in subsection 4.1.1. In the next subsection 4.1.2 the PAPI [18] library is introduced that can trace processor events and thus does not need to simulate the cache. For the current work the HPC Tool Kit [22] has been chosen. The tool kit is described in subsection 4.1.3.

4.1.1 Callgrind

Callgrind is a tool of the Valgrind framework [19] [4]. Valgrind is a framework for creating dynamic analysis tools. It analyzes the execution via binary translation. The framework also provides a range of tools for detailed performance profiling. One of the dynamic analysis tools is Callgrind that analyzes the calls of functions and can simulate the memory hierarchy. Callgrind is capable of counting the number of instruction and memory accesses as well as cache misses per source code line.

An advantage of Callgrind is that its results are not influenced by other processes of the system, because it simulates the entire CPU and only runs the binary to analyze on this CPU. Thus the results are very precise.

The main disadvantage is its very high overhead. The analysis of a program can take up to 100 times of the normal execution time. Another issue is that the accuracy of the measured data depends on the compilation of the program. For precise results the program cannot be fully optimized by the compiler, resulting in an even longer runtime of the analysis. The prohibitive time overhead makes Callgrind inapplicable to compute-intense applications.

4.1.2 PAPI Library

Most of the current high performance processors implement performance counters [2] [1]. These counters are hardware registers that count processor events. The PAPI library provides a *Performance Application Programming Interface* that can read the counters of the processor [18]. The provided interfaces are an abstraction layer to the varying implementation of the performance counter among the different CPUs. There is a low level interface that manages processor events, and a high level interface that starts, stops and reads these counter. The accuracy of the hardware counter depends on the length of the instruction block that is instrumented, because there is some overhead of the counter interface.

The PAPI library can be used for manual instrumenting the SAT solver. Therefore, the solver has to be extended to store the performance data and output it in a readable format. An advantage is the provided control. The user can choose the traced counter and measure only specific parts of the solver. The library has not been chosen for the analysis, because implementing the measurement for every single function call would be more difficult than using an existing framework.

4.1.3 HPCToolKit

The High Performance Tool Kit consists of several components for measuring the performance of fully optimized executables [22]. The analysis is done via sample based profiling, which is also known as statistical profiling. The program is interrupted at a sample point and the currently running method is detected to update its event counter. Such a sample point 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 [18].

The measurement is not as precise as using Valgrind because it is statistical. The measured result is only an approximation of the real values and thus is not exact, but the precision of the measurement increases with the number of samples. The benchmark just needs to fit the criteria of a long runtime, such that the accuracy of the measurement stays reasonable. The traced events can be any supported combination of all the PAPI events. The HPC Toolkit can only trace 4 events simultaneously.

The overhead of the analysis is negligible, thus this tool usable for long running benchmarks. The total runtime with and without measurement differ only 4%. Another advantage is that the binary can be optimized because the analysis detects in-lined functions and restores the original program structure, if the compiler adds some debugging information during compilation.

4.2 Measured Data

Important data for finding weaknesses of the implementation are the number of processor events. Nevertheless, for improving the usage of the cache the accesses to data structures and their sizes are interesting as well. The following processor events have been traced:

1. Total Cycles
2. Resource Stall Cycles
3. L2 Cache Miss
4. L2 Cache Hit

Total cycles are the CPU cycles the program needs to execute the program. Resource stall cycles are cycles that are spend waiting for resources. The sample period for the total cycles is 10^6. The sample periods for the other processor events are 10^5.

With the number of the given events additional data can be calculated. The difference between total cycles and resource stall cycles is called **work cycles**, because during these cycles the CPU proceeds with the algorithm and does not wait for resources. The number of **L2 accesses** is the sum of L2 hits and L2 misses. Since L1 accesses are not traced, this number is also equal to the total recordable **memory accesses**, because the sum of main memory accesses and L2 cache hits are the total accesses to any memory in the hierarchy. In the sequel memory access will be used if the focus is on the overall algorithm and L2 access will be used for focusing on processor events. The number of **main memory accesses** is equal to the L2 cache misses, because there is no other cache level between main memory and L2 cache in the used system. The **cache miss rate** is the ratio of L2 misses to L2 accesses. The **wait rate** is the ratio of resource stall cycles to total cycles.

For the data structure access the following values are stored:

1. Clause read access
2. Clause write access

Since not only the hardware counters are measured but also accesses to data structures multiple measurements had to be done. Processor events have only been measured when tracing data structures was disabled, because of the overhead of the access analysis. Additional to the measured and calculated metrics above the implementation of the used data structures and the implementation of the algorithm have been analyzed, because some of their weaknesses cannot be found by the run time analysis due to their small impact on the runtime.

4.3 Benchmark

The benchmark is a subset of the application benchmark of the SAT Competition 2009 [3]. Problem instances that are solved between 2 and 45 minutes by the basic version of the solver have been selected. The files with their solve time and the used memory are given in Table 4.1. The overall runtime of the algorithm for solving the algorithm is almost 10 hours.

Instance	**Solve time**(in s)	**Memory**(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

Table 4.1: Analysis Benchmark Details.

The number of clauses and variables is recorded after preprocessing the formula, because only the search is analyzed. The runtime of the preprocessor is negligible compared to the overall runtime as shown in Table 5.1.

The number of specific clause sizes is interesting for dividing the data. Figure 4.1 shows this ratio for the given benchmark after preprocessing. A first look emphasizes the special treatment of short clauses.

4.4 Benchmark System

The AMD Opteron 285 CPU with a clock frequency of 2.66GHz has been used for analyzing the SAT solver. During the analysis no other application has been allowed to use the second core of the dual core CPU. The cache is a 4-way-associative cache and

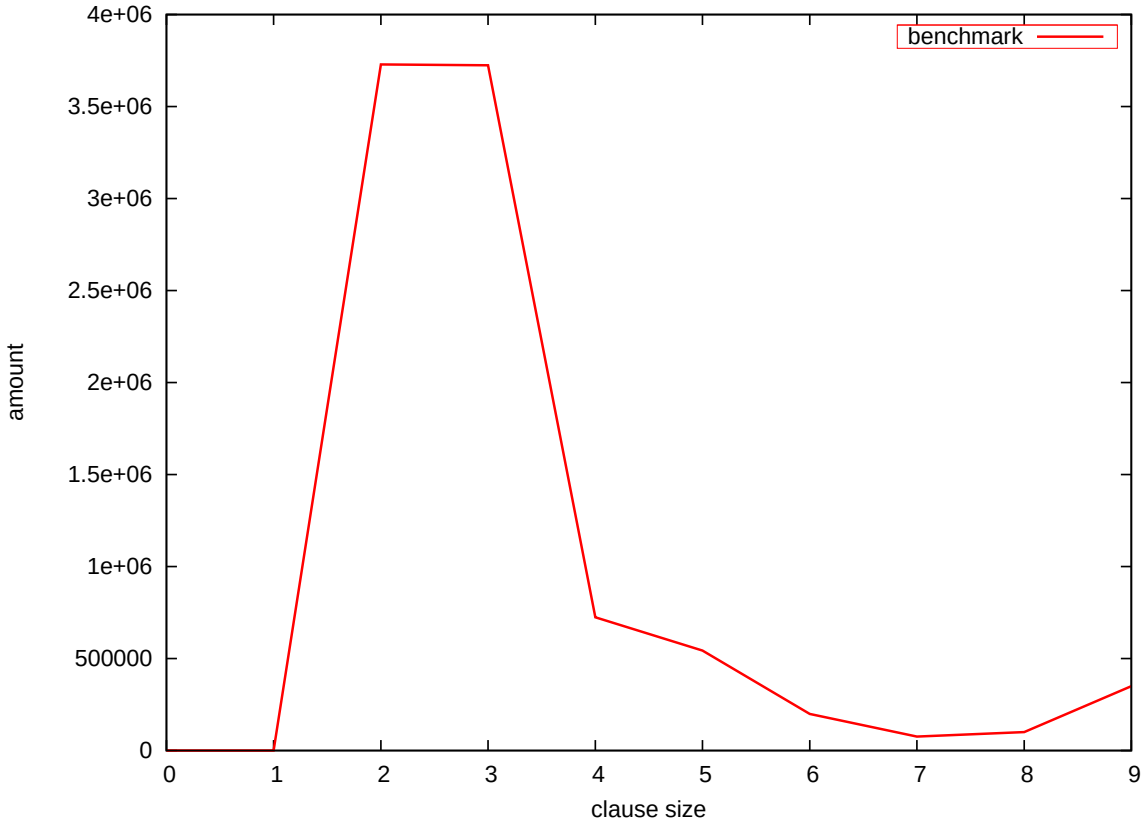


Figure 4.1: Clause Size Distribution after Preprocessing.

can store 1MB. The system has 2GB main memory. The level 2 TLB has 512 entries for data and 512 for instructions, so that 2MB data can be accessed without a TLB miss. The used page size is 4 KB.

5 Analysis

This chapter presents the analysis of the SAT solver. The analysis can only give hints where these optimization opportunities are located in the algorithm and its implementation. Furthermore, the implementation of the algorithm is analyzed.

An overview about the memory hierarchy utilization of leading SAT solvers is given in section 5.1. Section 5.2 discusses the data of the analysis. In the following sections the found improvement opportunities are examined.

5.1 Analysis of Leading SAT Solvers

MiniSAT 2.0 and PrecoSAT 246 have been analyzed using the same benchmark. Both solvers solved 39 out of the 40 instances in the given timeout. The cache miss rate of MiniSAT is 50%. The runtime for the 39 instances is 92% of the project solver runtime although MiniSAT accesses L2 cache and main memory less often. The number of work cycles is also comparable. The similar behavior of MiniSAT and the project solver is caused by the usage of similar data structures.

PrecoSAT solves the given benchmark much faster than the other two solvers. It needs only 33% of the basic version runtime. PrecoSAT processes the 39 instances with 37% of the project solvers work cycles. PrecoSATs L2 cache miss rate is 36%, and thus it has the lowest cache miss rate among the analyzed solvers.

Figure 5.1 shows the comparison of the three SAT solvers and gives an outlook to the final improved version of the project solver (Combination 6). Among the analyzed solvers, PrecoSAT has the best values, but the improved project solver comes close in the total runtime. The low number of work cycles of PrecoSAT indicate that this SAT solver implements a smarter algorithm for the benchmark.

5.2 Runtime Analysis

Based on the runtime analysis of section 5.2.1 the remaining sub sections discuss the result. The part of the algorithm where most of the runtime is spent is discussed and the resulting memory access schemas are introduced.

5.2.1 Runtime Distribution

Table 5.1 shows the distribution of the runtime spent among the components. The distribution of resource stall cycles and cache hits and misses are also given. In the first

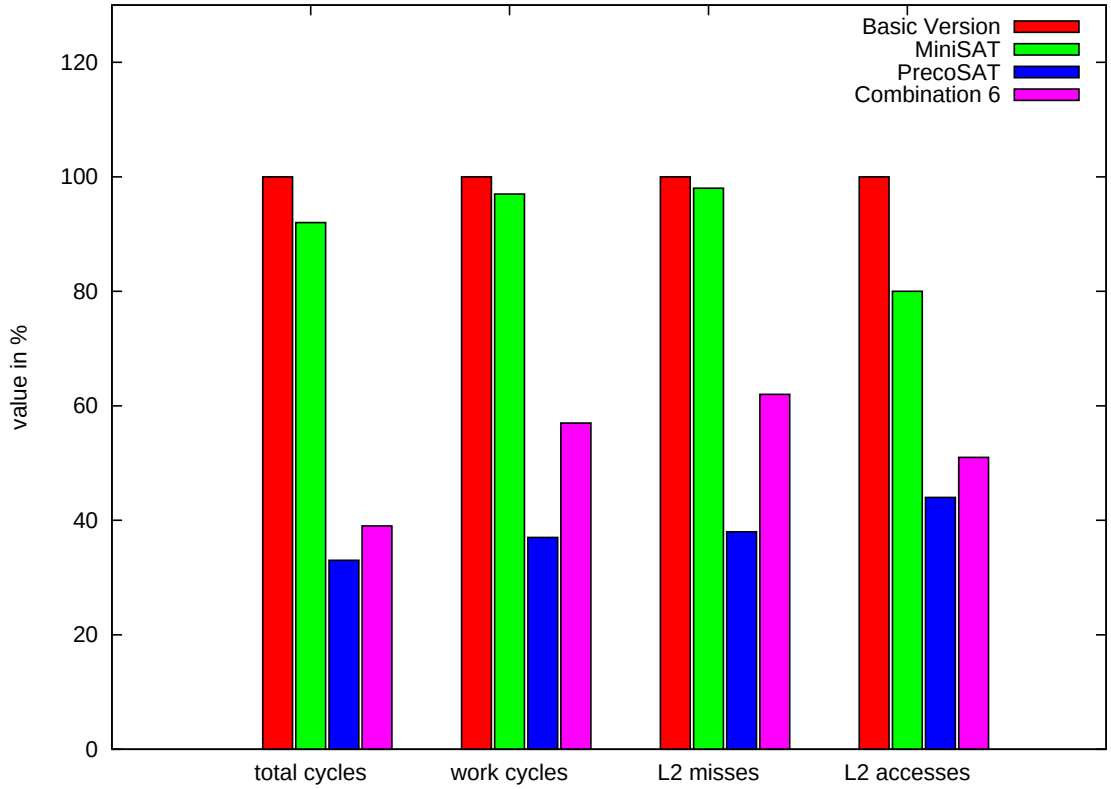


Figure 5.1: Comparison of SAT Solvers.

row, the absolute numbers for the benchmark are provided. The remaining rows present the percentage that is needed by the specified part of the solver (first row).

Improving the overall performance can mainly be done in the propagation component. Reducing the number of cycles of any other component can only save about eight percent of the total runtime. Therefore, the improvements focus on this component. A more detailed analysis shows that most of the runtime is spent in the propagate_long() method. Even more specific, there are two function calls in this function that use a huge part of the solving time.

Table 5.2 shows that most of the solving time is spent in propagating long clauses. Most cache accesses occur in this method. The runtime of the function propagate_long() is mainly consumed by the following three functions: Clause.literal(0) and Clause.literal(i) are clause read accesses (Listing 5.2) with the only difference that the index of the literal is known at compile time for the first function. The Vector.erase() procedure executes almost a quarter of the runtime, although it has a low L2 cache miss rate. The reason for the large number of stall cycles is the number of cycles that are needed to access L2 cache. The Vector.erase() procedure is called, if a clause is watched by another literal. The pointer has to be removed from the current watch list and the gap in the vector has to be closed by pushing all following elements one position forward.

Component	Cycles	Stall Cycles	L2 Misses	L2 Accesses
Program	$9.222 \cdot 10^{13}$	$7.535 \cdot 10^{13}$	$2.9566 \cdot 10^{11}$	$7.2225 \cdot 10^{11}$
Program	100.0%	100.0%	100.0%	100.0%
Search	99.41%	99.68%	99.50%	99.27%
Propagation	91.65%	92.62%	90.08%	88.94%
Decision Heuristic	1.77%	1.59%	3.13%	2.95%
Removal Heuristic	0.31%	0.21%	0.09%	0.21%
Analysis	5.74%	5.42%	6.27%	7.27%
Event Heuristic	0.00%	1.33%	0.00%	0.00%

Table 5.1: Processor Event Distribution Among the Solver Components.

Component	Cycles	Stall Cycles	L2 Misses	L2 Accesses
Program	$9.222\ 10^{13}$	$7.535\ 10^{13}$	$2.9566\ 10^{11}$	$7.2225\ 10^{11}$
propagate()	90.34%	91.25%	87.00%	86.06%
propagate_binary()	5.71%	5.55%	7.95%	5.64%
propagate_long()	83.86%	85.30%	78.17%	79.78%
Clause.literal(0)	19.68%	23.86%	19.84%	8.95%
Clause.literal(i)	26.12%	30.63%	4.23%	3.62%
Vector.erase()	24.26%	18.59%	2.19%	36.64%

Table 5.2: Processor Event Distribution in Unit Propagation.

5.2.2 Propagate_long Implementation

An outline of the propagate_long() implementation is given in Algorithm 5.1. The three major functions of Figure 5.2 are the following. Line 5 and line 8 refer to Clause.literal(0). In line 13 the method Clause.literal(i) is called with an index. In line 20 Vector.erase() is called.

The propagation object knows the current assignment *assignment*. It also manages the watch lists in the structure *watchlists*. The procedure is called and the literal to propagate is passed to it as a parameter (line 1). For a better overview, the variable *watch* is introduced. It represents the watch list for the literal (line 2). Next all clauses in this list are processed (line 3). Giving a better overview the current clause is referenced by the variable *clause* (line 4). The next step is to ensure the following Invariant (line 5-7).

Invariant 1. *The watched literals are stored at the first two positions of the clause.*

If the other watched literal already satisfies the clause, the next clause is processed (line 8-10). Otherwise, the remaining literals have to be examined. They are accessed in ascending order of their index (line 12). The variable *literal* represents the current literal

```
1 clause propagate_long(literal_t literal)
2 {
3   watch = watchlists[ literal ];
4   for( index = 0; index < watch.size(); ++ index ){
5     clause = watch[ index ];
6     if( clause.literal(0) == invert(literal) ){
7         clause.swap( 0, 1);
8     }
9     if( assignment.literal_satisfied( clause.literal(0) )){
10      continue;
11    }
12    unsigned int i = 2;
13    for(; i < clause.size(); ++i ){
14      literal_t literal = clause.literal(i);
15      if(
16        assignment.variable_undefined(variable(literal)) ||
17        assignment.literal_satisfied( literal )
18      ){
19        clause.swap( 1, i );
20        watch_clause( clause, literal );
21        watch.erase( index );
22        index --;
23        continue;
24      }
25    }
26    if( i == clause.size() ){
27      if( enqueue_literal( clause.literal(0), clause ) == false ){
28        return clause;
29      }
30    }
31  }
32  return 0;
33 }
```

Listing 5.1: Implementation of the propagate_long() Method.

(line 13). Its state is checked next (line 14-17). If this literal is satisfied or undefined, it can be watched (line 18-24).

Invariant 2. *The first two literals in the clause are the inverse of the literals that watch the clause.*

Invariant 2 is ensured (line 18), the watch list of the literal is updated (line 19) and the clause is removed from the current watch list (line 20). Because the clause became no unit, the next one can be processed (line 21-22). If no satisfied or undefined literal has been found (line 25) the clause is either unit or a conflict clause. These cases are checked in the *enqueue_literal*() method (line 26). Its task is to enqueue the unit to the unit queue. The reason, level and the assignment for the literals are set in this method as well. If the literal cannot be enqueued, because it is unsatisfied, the clause

is returned as conflict clause, because all of its literals are unsatisfied under the current assignment (line 27). If all clauses had been processed successfully, a 0 is returned (line 31) to indicate that no conflict has been found. In this case, the unit propagation goes ahead with propagating the next literal from the unit queue as outlined in algorithm 3.

5.2.3 Data Structure Implementation

The unit propagation does not access clauses directly. As stated in section 2.4.5 a list of clauses is stored per literal. These lists are implemented using a vector. The reference to the according watch list for a literal is also implemented using a vector, which is indexed by the literal. To propagate the current literal in the first clause the according watch list has to be accessed. This procedure is illustrated in Figure 5.2.

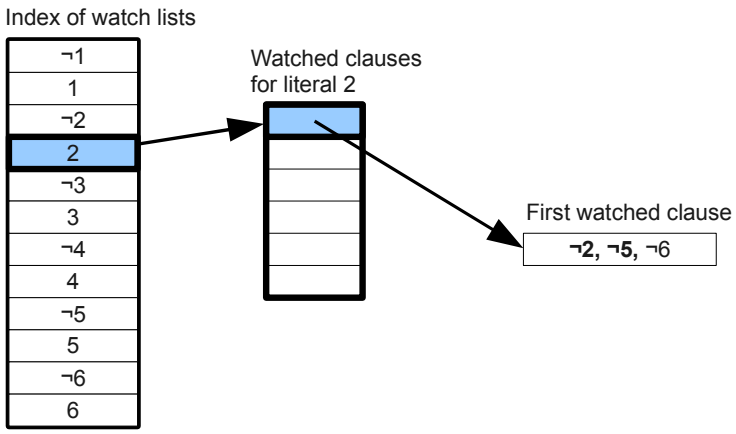


Figure 5.2: First Clause Access in a Watched List.

Accessing a clause results in two memory accesses. The first access will be called *memory hop*, because this access only gets the address of the data and the next access fetches the data in memory. Two accesses need two cache lines to be fetched from main memory in the worst case. These loads are reasons why the literal read procedure of the clause object consume such a big part of the runtime.

Another memory hop arises due to the implementation of the clause. The literals are not stored in the clause itself but in an external array, which is called clause body. The clause head stores only the address of this array and its size. Additionally the clause stores an activity value that is used by some removal component variants to decide whether the clause has to be removed. Listing 5.2 shows the implementation of the clause and the appropriate memory schema.

The size of the clause head is 16 bytes. This size can be calculated by the sum of the stored data structures (compare subsection 2.4.3). The size of the external array depends on the number of literals that belong to the clause. Accessing one of the literals always results in a memory hop, because only knowing the location of the clause head

```
1 class CppClause
2 {
3   private:
4     float activity;
5     uint32_t size;
6     literal_t* literals;
7   public:
8     literal_t literal(uint32_t index)
9     {
10      return literals[index];
11    }
12
13    void literal(uint32_t index, literal_t lit)
14    {
15      literals[index] = lit;
16    }
17    // constructor and other methods
18 };
```

Listing 5.2: Clause Implementation

is not sufficient to know the location of the literals. Their address is created at runtime by allocating memory dynamically.

5.2.4 Literal Accesses

The trace of literal accesses in a clause is shown in Figure 5.3. Almost 60% of read accesses are on the first literal in the clause. Only 15 percent of the read accesses are spent at the second literal. The higher the index of the literal the fewer is the number of accesses. The total number of literal read accesses is approximately $1.024\ 10^{12}$.

The number of write accesses is much lower. The $1.63\ 10^{11}$ writes are only 16% of the number of read accesses. The accesses to the first two positions are different compared to the read accesses. The second literal is written more often than the first one, namely 50% compared to 25%. This effect is caused by Invariant 1. If Invariant 2 does not hold the two literals are simply swapped and it is satisfied again. If another literal has to be watched in the clause, this literal is swapped to index 1.

The descending number of accesses to a literal with ascending index can be explained by the algorithm. The literals of a clause are accessed according to their position until a condition is satisfied (Listing 5.1, line 12). The remaining literals are not touched in this iteration. Together with the clause size distribution in Figure 4.1 the many accesses on the first two literals can be explained.

5.2.5 Measurement Errors

Since the overhead of the HPC Toolkit is not known the measurement error has been determined. Therefore, the basic version of the solver has been analyzed fife times on

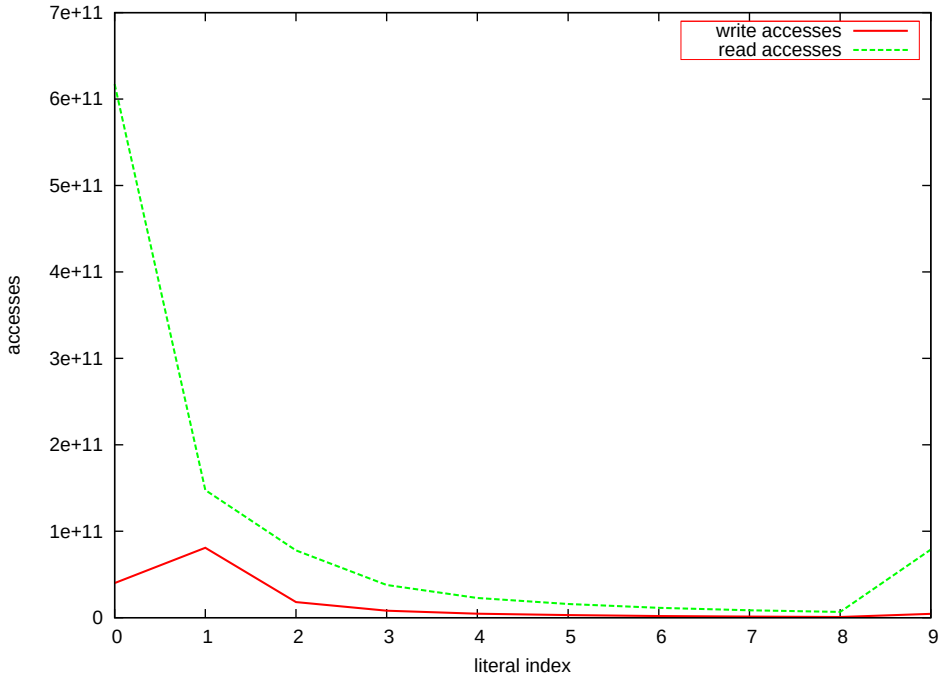


Figure 5.3: Distribution of Literal Accesses in Clauses.

Processor Event	Relative Standard Deviation
Total Cycles	3.16%
Resource Stall Cycles	3.95%
Level 2 Cache Misses	8.18%
Level 2 Cache Hits	7.54%

Table 5.3: Measurement Standard Deviation Relative to Arithmetic Mean.

the benchmark. Afterwards the arithmetic mean of each processor event measurement has been determined and the standard deviation of the fife runs has been calculated. This procedure results in the values given in table 5.3.

The most precise measurement is performed on the total cycles. The relative error of the level 2 misses is close to 10%. Therefore, the results of the analysis will mainly be judged by their overall runtime. The other processor events will be compared as well, but only differences larger than 10% can really be considered to be caused by the applied changes.

5.3 Implementation Analysis

The implementation of the used data structures does not really fit to the statements in section 3.5 The memory footprint can be minimized to increase the hardware utilization.

5.3.1 Assignment

The assignment holds ternary values because it is partial and needs to know whether a variable is set to $true$, $false$ or whether it is $undefined$. Each ternary element is stored in a byte although one byte can hold 256 different values. It is possible to store 5 ternary values in 256 elements, because $3^5 = 243 \leq 256$ holds. Since the assignment stores one polarity per variable the size of the assignment can be reduced from 500KB to 120KB assuming the biggest instance of the benchmark with 500.000 variables (section 4.3).

5.3.2 Boolean Array

The Boolean array use 8 bit per element where only one is needed. Most of these arrays are used to store a flag for a variable. The biggest number of variables is 600.000. If the Boolean array is compressed the used size is reduced from 600 KB to 75KB. For accessing a single bit in a byte some additional instructions have to be executed.

5.3.3 Dynamic Allocated Objects

Objects for temporary usage with an unknown size are usually allocated and freed at the runtime of the program. Receiving this memory executes some instructions of the operating system or a system library. The touched and freed memory increases for the memory footprint and should be reduced. Especially C++ vectors have the following bad property. Every time their current capacity is reached and another element should be stored, their whole data has to be copied to another place in memory. The conflict analysis is done in one function that creates three vectors. They are used for temporal data during the analysis and are thrown away afterwards again. At the next conflict, memory is allocated again. The reallocation can be avoided by reusing the old vector storage by not throwing it away, but clearing and reusing it.

6 Improvement

There are two ways of improving the execution of a program. **Micro-optimization** tries to minimize the executed cycles by improving the implementation of the algorithm. **Macro-optimization** improves the complexity of the algorithm. Usually the latter approach has a bigger impact on the runtime. Examples are the comparison of the search algorithms linear and binary search or the sort algorithms bubble-sort and merge-sort. Each latter algorithm has a better worst case runtime complexity and thus runs faster, if the number of processed elements increases. Micro optimization can improve the runtime only by a constant factor.

Nevertheless, the work will concentrate on micro optimization. Section 6.1 will repeat the conflicts of modern hardware and the solver implementation. Afterwards the arising problems of macro optimization are explained in section 6.2. In the remaining sections the improvements are introduced, the effect of their combination is discussed and a final version of the solver configuration is presented.

6.1 Conflicts of Hardware and Implementation

In section 3.5 the main statements have been:

- keep number of memory accesses small
- reduce the cache miss penalty
- increase the cache hit rate
- keep memory footprint small
- keep frequently used data in cache

The implementation presented in Algorithm 5.1 (page 40) and the runtime distribution in Figure 5.2 (page 39) show that the project solver does not meet these criteria well.

The memory footprint of the algorithm cannot be reduced to the memory size of the parsed input formula since clause learning is a main part of the algorithm. The number of accessed clauses during propagation is small. According to the watched literal schema in section 2.4.5, the number of processed clauses is reduced compared to the naive schema that processes all clauses in every propagation step. A disadvantage of the two watched literal schema is its almost random access of clauses. The clauses where the same literal is watched are not necessarily stored close to each other or in a specific order. This fact complicates the work of the prefetcher.

6.2 Comparison of Different Runs

As mentioned above macro optimization is the better choice in general. However, in case of this project it is difficult to apply it, because it is not possible to determine the reason of an improvement if the algorithm has been changed. The question whether the runtime decreased because of a better cache hit rate or a smaller search tree remains open. The following sections explain the arising problem.

6.2.1 Same Search Path

Comparing two runs of the same instance using the same solver configuration is easy, because the execution is deterministic and thus repeatable. The order of made decisions, found conflicts, learnt clauses and clauses in the watch lists remain. Even the schedules of restarts and removals remain, because they depend on the number of conflicts (section 2.4.8 and 2.4.9). The whole search path is invariant for a single configuration. The difference in runtime can be explained only by the different system behavior and the resulting measurement errors (section 5.2.5).

Thus, the comparison of two runs with an equal runtime is easy, if the search path is the same. Assuming the same hardware is used, every change in runtime is caused only by the differences of the two implementations. One can determine the effect of this kind of changes by regarding the runtime first. If the runtime of the new version is worse, this version can be ignored for the combination of the best versions. Otherwise, it is analyzed further to find the benefits of the differences, so that it can be combined with other configurations.

6.2.2 Different Search Path

If the search path of two configurations is not equal, it is difficult to determine the influence of the differences and hardware effects. Runtime changes can be caused by the algorithmic change, by improved resource usage or even by a combination of both. The impacts of the different reasons cannot be separated.

The runtime of two different algorithms can be the same, although the first one may visit less elements of the search tree than the other one. This effect can be caused by a better cache miss rate because of the compact storage of the propagated clauses. On the other hand accessing fewer elements could also result in a longer runtime because of a spread storage of the clauses. There is no way of extracting the impact on the cache from the runtime and the hardware events that can be watched without knowing more details about the storage of the clauses and the number of processed elements of the search tree.

It would be nice to have a metric that compares the search path of two runs and returns a runtime ratio. There are only approximations for this metric. For example MiniSAT 2.0 [13] shows the number of propagations per second. From a theoretical point of view, a more precise metric seems to be the number of processed clauses during search and even more accurate should be the number of processed literals.

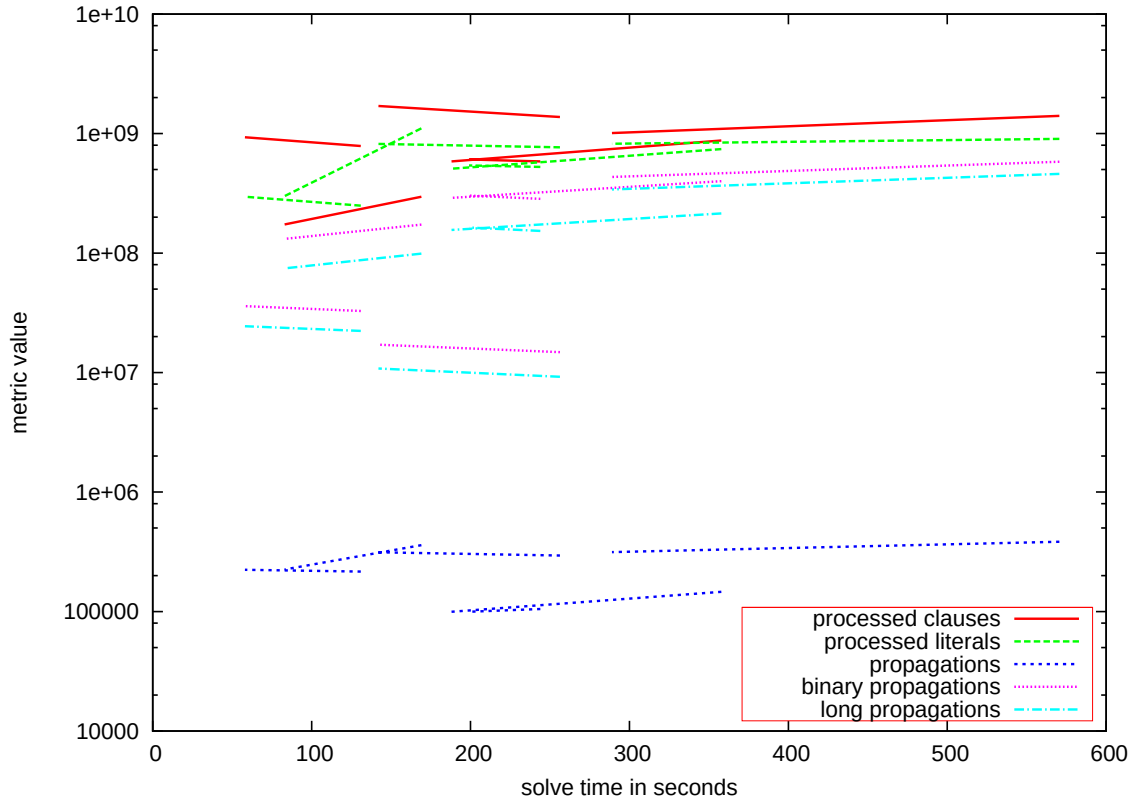


Figure 6.1: Metrics to Compare Search Path.

Figure 6.1 shows the ratio between the runtime and the metric value. The two versions that are compared are the basic version and a version using the phase saving heuristic. The benchmark consists of six instances of the benchmark.Every end of a line represents a run of one of these versions. The point in the diagram represents the relation between runtime and metric value. The slope of the line should be positive because the value of the metric should increase with the runtime. All the given metrics fail this criteria. For each of them there are lines with a negative slope on the left side of the picture.

6.3 Improving Data Structures

The analysis pointed out that clauses are the critical data structure for the runtime. The literal access of clauses needs most of the solving time. One reason is the number of memory hops to access literals. Some improvements to tackle this weakness are presented in subsection 6.3.1. The following subsection 6.3.2 tries to solve the other time consuming part of the runtime, maintaining the watch list.

6.3.1 Clause Implementation Variants

Figure 5.3 shows the access of a single literal in the clause. The head of the clause and its literals do not have to be stored separately. Storing both together safes one memory hop per access and thus one less memory line to be accessed when accessing a literal, if

the clause can be stored on a single cache line. This schema is called **flatten clause**.

```
1  typedef literal_t* cls;
2  cls literals;
3
4  literal_t literal(cls literals,
5                    uint32_t index)
6  {
7    return literals[index];
8  }
9
10 uint32_t size(cls literals)
11 {
12   return ((uint32_t*)literals) [-1];
13 }
14
15 float activity(cls literals)
16 {
17   return
18      ((float*)(&(((uint32_t*)cls)[-1])))[-1];
19 }
20
21 // constructor and other methods
```

Listing 6.1: Member Access in Flatten Clause

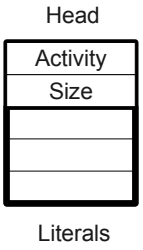


Figure 6.2: Flatten Clause.

Figure 6.2 shows the combination of clause head and body. The clause is implemented in an array. The first two elements of the array are no literals but the size and the activity. The literals are accessed as in the previous implementation (line 4-8). Accessing the size cannot be done using a simple getter method. The element before the stored literals has to be read (line 10-13). Due to the fact that literals are read more often than the size of the clause, their access is kept simple and accessing the size involves pointer arithmetic. Accessing the activity is implemented similar to accessing the size with the difference that the activity is stored before the size (line 15-19).

The flatten clause approach improves the runtime of the solver by 21.18%. Because the second memory hop is not needed, the number of memory accesses is reduced by 24.46% and the cache miss rate is only 32.52%. The distribution of the runtime among the components varies only slightly compared to the basic version. The unit propagation needs 90.65% instead of 91.65%. The runtime of the propagate_long() function decreases to 81.2%. The watch list management uses a larger part of 30.67% for erasing elements, because it is not improved.

Having in mind the distribution of the clause sizes from Figure 4.1 and the special treatment of frequently used data the presented idea can be extended. Not all literals are added to the clause head. Instead, a number of literals is determined. This schema will be called **cache clause** and has been already introduced in [9] as clause packing. The literals that are stored in the head will be called *local literals* and the other part will be referred to as *external literals*. The number of local literals should be set such that

multiple causes fit exactly in a cache line. The fact that the storage of the C runtime to manage the memory is also on the cache line will be discussed in section 6.4.3.

```
1  class CppCacheClause
2  {
3  private:
4    float activity;
5    uint32_t  size;
6    literal_t *literals;
7    literal_t local_lits[ELEMENTS];
8  public:
9    literal_t literal(uint32_t index)
10   {
11     if(index < ELEMENTS){
12       return local_lits[index]
13     } else {
14       return
15           literals[index-ELEMENTS];
16     }
17   }
18   // constructor and other methods
19 }
```

Listing 6.2: Cache Clause Implementation

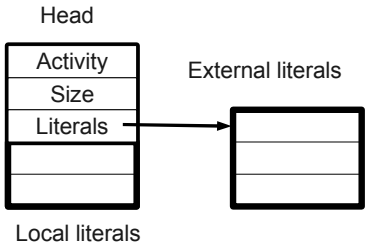


Figure 6.3: Cache Clause.

The cache clause schema (Figure 6.3) and the implementation of the literal access are presented in Listing 6.2. The head of the clause contains an additional array of local literals (line 7). Their access does not introduce a memory hop. In the given example, the number of local stored literals is two. In the implementation the constant $ELEMENTS$ determines the number of local stored literals. The saved memory hop has to recompense the additional instructions that are executed to determine where the desired literal is stored (line 9-17).

The number of local literals for the experiment has been chosen to be 4 such that the size of the clause became 32 bytes and thus two clauses fit exactly on one cache line if the system memory overhead is ignored. The runtime decreased to 80.83% of the basic versions runtime. As in the flatten clause approach the number of memory accesses is reduced, but only for the first four literals of the clause. Thus, the cache miss rate decreased to 33.06% and the number of memory accesses decreases to 85.38% compared to the base version. The distributions of the runtime among the components and in the propagate() method differ only half a percent to the ones from the flatten clause approach.

The two above schemas reduce the average access time of the literals in the clause. The analysis in Figure 5.2 shows that the first access needs about 20% of the total runtime. This time is the result of two memory accesses. The address of the watched clauses are stored in the watch list. Accessing the first literal needs to access the clause first to obtain the address. The big percentage of the total runtime is spent at this

access, because also a big part of the total cache misses occurs during these accesses. The miss penalty for accessing the clause can be reduced by **prefetching** the watched clauses before propagating the current literal.

When a literal is propagated, the very first memory access retrieves the according watch list. This access can also result in a cache miss. Therefore, the list is fetched immediately when a literal is added to the propagation unit queue. Two variants are tested with different results.

The first approach **prefetch method 1** prefetchs the watched clauses immediately before they are used in the propagation. This method achieves 12.86% runtime gain. Of this runtime the propagate_long() method that is the only affected method still needs 81.26% of the programs runtime. Comparing the total time of this method to the basic version of the solver the runtime of this method decreased from 83.36% to 70.81%. Prefetching the clauses takes just a little time so that the achieved improvement comes only from the propagate_long() method.

The second approach **prefetch method 2** introduces a parameter *depth* that controls when the clauses of a watched list have to be prefetched. Only watch lists of a certain number of literals at the beginning of the unit queue are fetched. For example the unit queue $q = \{2, 5\}$ and the value of the parameter is $depth = 2$ and the literal 1 is propagated at the moment. Assuming literal $\neg 6$ is implied by literal 1 and thus added to the queue. The resulting queue is $q = \{2, 5, \neg 6\}$. The watch list of $\neg 6$ is prefetched but not the watched clauses. Just when the next literal 2 is dequeued, the watched clauses of literal $\neg 6$ are prefetched.

The given results are obtained with the parameter $depth = 10$. At first glance the gained effect for the propagate_long() method looks even better than in the previous example. A closer look explains why the overall runtime is only 96.4% of the basic versions runtime. The distribution among the components does not change. The nearly 4% gain belong to the unit propagation. The propagate_long() method needs only 76.6% of the runtime, because the clause heads of the current watch list are already in cache. The prefetching is done by the enqueue() method that enqueues literals to the unit queue of the propagation. The part of runtime that is used by this function increased from almost 0% to 5.3% so that the overall costs of propagating a literal through the long clauses is not different to the one in the previous approach. The improvement of this approach can be increased by tuning the *deptch* parameter. If a value had been found, it is not ensured that this value is also the best for other instances. Thus, this optimization is not performed in the current work.

6.3.2 Watch List Improvements

Another weakness of unit propagation in Figure 5.2 (page 39) is the part that is spent in the method Vector.erase(). Removing a value in the middle of the vector is done by moving all the following pointers one position forward. Thus, time is spent on copying data in memory. Erasing elements from a list is less expensive. Only the preceding and the successive element have to be notified that the element in the middle does not exist anymore. The part of the runtime that is spent on reading elements of the watch list

cannot be extracted. Reading an element in a vector is cheaper than one from a list, because the address of the element is known in advance. Processing a list is done by iterating over all its elements, as it is done with the watch list during propagation as well. Thus, it should be no major disadvantage using a list instead of a vector for implementing the watched list. The implementation using a list for the watch list implementation is called **list propagation**.

It turned out that using a vector is the better choice. Using a list to avoid the erase overhead resulted in 20% more runtime. The number of total cycles increased by 18% although the work cycles are only 67.2% compared to the standard version. The number of total memory accesses decreases by 13.4% but the number of cache misses increased dramatically by 50.3%. This effect is caused by the comparison of reads in a vector and a list. Iterating over a vector results in linear memory accesses so that the prefetch unit is able to prefetch the address of the next clause. Since the size of an address is only 8 bytes, eight clauses can be processed before another cache line has to be loaded. Iterating over a list often loads a new cache line for every element, because the elements of the list are not stored close to each other. The advantage of the removal time disappears because of the iteration cache misses.

Removing elements from the vector can also be done lazily. There are two kinds of clauses that stay in the current watch list while propagating a literal. Clauses of the first kind are clauses whose other watched literal is satisfied (Algorithm 5.1, line 9-11). Clauses of second kind are clauses that become reason for a literal. Clauses that where another literal is watched are removed (line 15-24). If a conflict is found in the watch list the propagation stops. The watch list contains the kept clauses, the conflict clause and all clauses that are stored behind this conflict clause.

If a clause is removed, the pointer can be kept in the watch list and be marked as a gap. If the propagation stops, all these gaps need to be closed. All the kept clauses can be stored compact at the beginning of the vector. The order of clauses in the vector is as follows:

1. kept clauses
2. gap
3. clauses to be processed

This removal of clauses from the watch list is called **lazy removal**. It is illustrated in Figure 6.4. The picture shows four states of a watch list during propagation. Propagating a literal starts with visiting the first clause in state 1. The kind of the following clauses it not known at the current state.

The first clause should be removed from the current vector. Instead of pushing all following elements one position forward a gap is created. The propagation proceeds with the second clause. This clause should also be removed. Again, the rest of the watch list is not touched, but the gap is enlarged. This state is illustrated in state 2.

The next clause to propagate has to be kept in the watch list. Its final position in the watch list will be index 0. Thus, it is moved to this position. Consequently, its old

position is added to the gap. Processing the next clause that shall be removed enlarges the gap again. The last two steps are repeated for the next two clauses. The resulting situation is illustrated in state 3. The two clauses that have to be kept are stored at the first positions of the watch list. The gap is behind these kept clauses and behind the gap the remaining clause of the watch list are stored.

Since the next clause to propagate is a conflict clause propagation stops. The gap has to be removed from the watch list again. This step is done by moving all clauses that are stored behind the gap close to the already kept clauses. Finally removing the gap from the watch list results in state 4. This state is the same as if all the clauses to remove are removed immediately. In contrast to the old implementation, the lazy removal moves every clause only once. Thus, the number memory accesses is reduced.

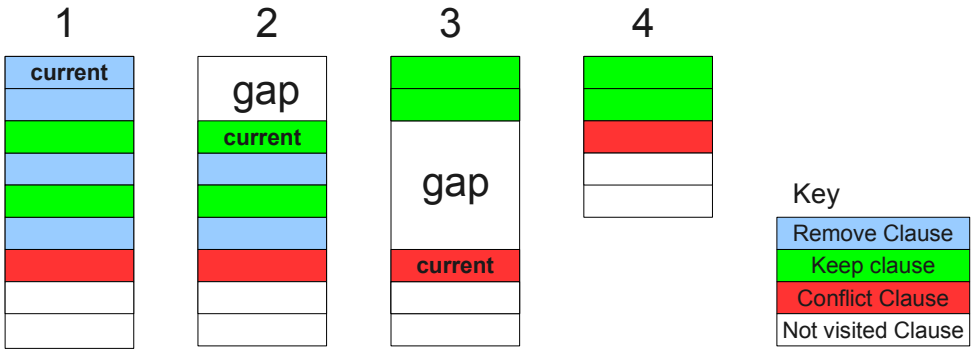


Figure 6.4: States of a Watch List with Lazy Removal.

The new removal reduces the runtime of the watch list maintenance. This reduction results in 23.78% speedup of the solver. The number of L2 misses does not change, but the number of memory accesses decreased by 35.34%. Thus, the L2 miss rate increases to 54.42%. These results indicate that maintaining the watch list is buffered in the L2 cache completely. Moving elements multiple times in the same lists meets the temporal locality criteria. Due to the lazy moving of clauses in the watch list the number of work cycles decreases by 52.05%. Consequently, the basic version of the solver seems to spend 50% of its work cycles on maintaining the watch lists. The according memory accesses are buffered in L2 cache and thus the cache miss rate is lower. Removing these easy to predict memory accesses increases the L2 cache miss rate, because the part of almost random accesses among all accesses increases. The L2 cache miss rate does not seem to be sufficient to indicate the hardware utilization of a SAT solver well.

Figure 6.5 summarizes the explained changes to the clause and watch list usage. The major improvements come from saving memory hops in the flatten clause and the cache clause and the lazy removal in the watch list. Both improvements reduce the runtime by almost 25%. The number of work cycles is only improved by the latter one. On the other hand, the other improvements reduce the number of L2 cache misses significantly.

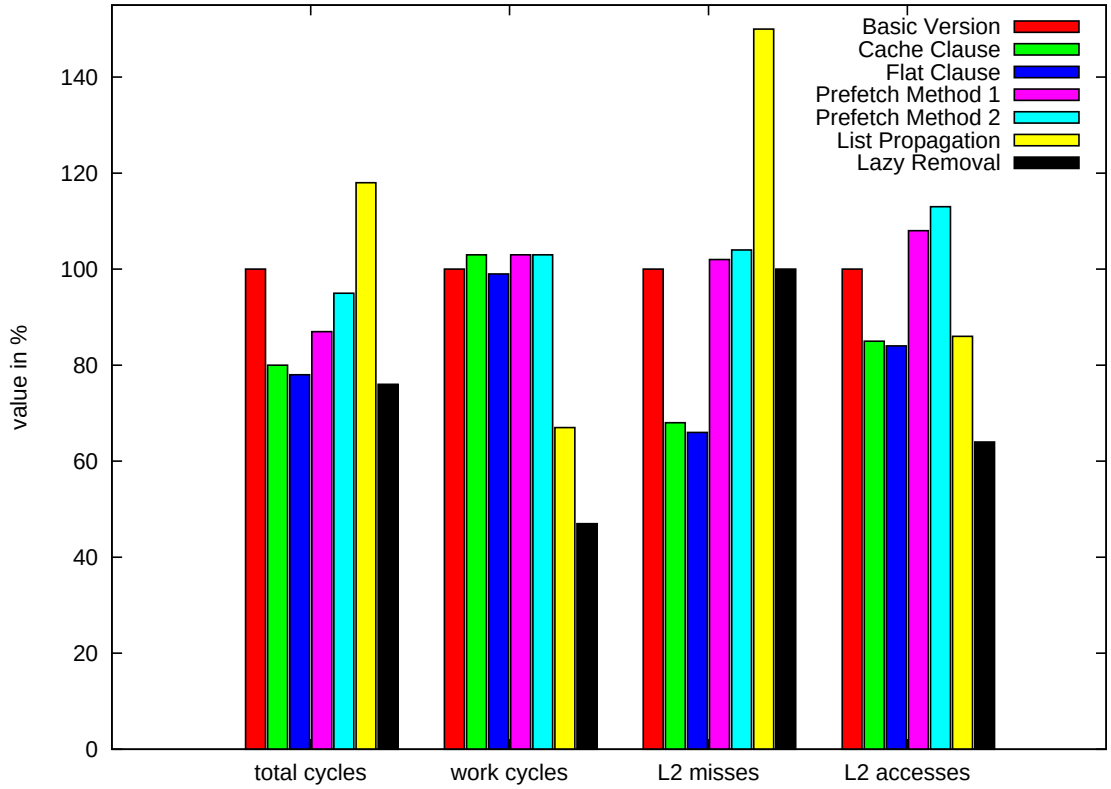


Figure 6.5: Comparison of Data Structure Improvements.

The diagram also shows the higher number of L2 cache misses of the list propagation approach.

6.4 Improving Memory Accesses

The goal of improving memory accesses can be achieved in three ways: compress data, store data compact, reuse already allocated memory. Subsections 6.4.5 presents a comparison of different compiler optimizations.

6.4.1 Compression of Data Structures

Data representation can be compressed, if it uses more memory than needed. For example storing a Boolean value can be done using a single bit instead of a byte as it is done in the implementation. The representation of a Boolean array can be compressed. The configuration with a compressed Boolean array is referred to as **compressed Boolean array**.

Figure 6.6 gives an example for this schema. The compression and decompression of the data requires more instructions than just returning the value of a byte. The implemented array is based on 32 bit words. First, the according word has to be accessed (line 6-8). Next, the matching bit of this word has to be extracted. Therefore, its offset is read (line 10) and a bit mask is created (line 12). Using this mask and word, the

```
1
2 bool get( boolarray_t boolarray,
3           const uint32_t index )
4 {
5   // 32 elements per word
6   uint32_t wordPos = index >> 5;
7   uint32_t word =
8     ((uint32_t*)boolarray)[wordPos];
9   // last 5 bits are offset
10  uint32_t offset = (index & 31);
11  // create mask
12  uint32_t element = 1 << offset;
13  // get bit from word
14  element = word & element;
15  // move back to lowest bits
16  return (bool)(element >> offset);
17 }
```

Listing 6.3: Uncompression of Boolean Array

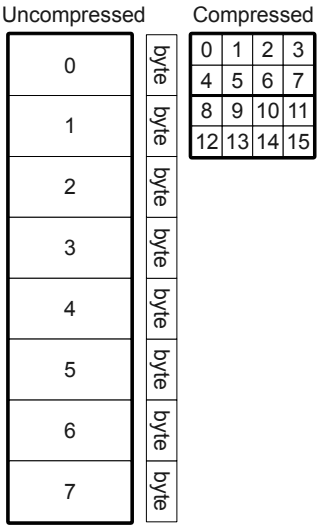


Figure 6.6: Compressing Boolean Arrays.

desired information that is stored in the accessed bit is returned (line 16). The runtime improved by 0.9%. This improvement indicates that the compression of a small array and the 1.3% less L2 cache misses are not worth the additional instructions that are executed every time the array is accessed.

The assignment can be compressed in a similar way with the only difference that a ternary value has to be stored. In the basic implementation again a byte is used per element although this byte can store 256 different values instead of 3. In the extreme case, five ternary values can be stored in a byte, because all possible combinations result in $3^5 = 243$. Accessing this compact compression is only possible by using the slow modulo instruction. Due to the fact that the compression of factor eight of the Boolean array does not result in a high improvement, only four ternary values are stored in a byte. In this approach the values are accessible using fast bit shift operations similar as in Listing 6.3. This schema is called **compressed assignment** and results in an insignificant improvement of 0.4%. As in the case of the compressed Boolean array the gained 2.3% reduction of L2 cache misses are not worth the additional 6.3% work cycles. This result has been also obtained in [9].

The implemented assignment supports backing up the last set polarity of a variable. This backup polarity is used by some decision heuristics to assign the polarity again to the variable and to propagate the variable again.

As shown in Figure 6.7 every even value of the assignment array corresponds to the current polarity and every odd field corresponds to the backup polarity. Reading the current polarity during propagation always loads the backup polarities in the cache as well. To avoid this effect the backup polarities are stored at a negative index. The left assignment is implemented in the basic version. Its pointer points at the very first element. The new approach on the right side divides the assignment into two parts and

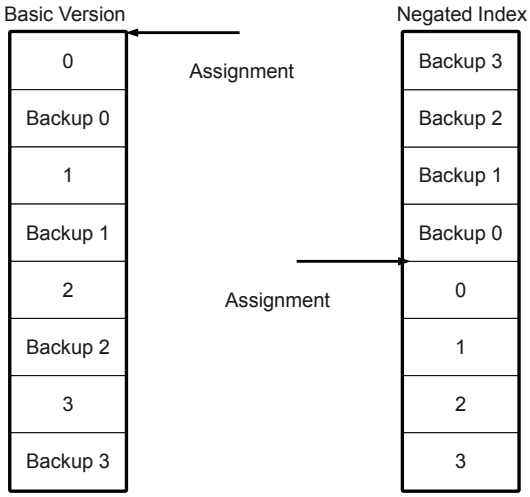


Figure 6.7: Storing Backup Assignment Using Negative Indexes.

begins indexing in the middle of the array. Thus, the variable can be used as index for the assignment and not its double as it is done in the basic case. This approach, which is called **negative index assignment**, achieves 0.75% runtime improvement. Again, the saved 1.2% L2 cache misses are not worth the 1.2% additional instructions.

6.4.2 Compression of Literals

Compressing literals is easier to implement than compressing a binary or ternary value, because the programming language provides the necessary elements. As introduced in Chaff [17] it is possible to store 3 literals in a 64 bit double word. In Chaff each literal is represented with 20 bit such that maximum number of variables is 2^{19}, because the literal has to store the polarity. Since a literal can also be represented using 21 bit and three of them still fit into a 64 bit double word the implementation has been done in this way and is called **compressed clause**. The size of 21 bit allows to process formulas with 2^{20}. This limit is more practical than the previous one but compared to using 32 bit literals and 2^{31} possible variables it remains a limit. The used benchmark can be handled with the Chaff approach, because the instance with the highest number of variables contains only 507145 variables.

The compression of the literals saves almost a third of the necessary storage for literals of clauses having a size larger than 2. The compression factor depends on the number of literals in a clause as shown in Figure 6.8. The picture on shows to literal lists. The left one is uncompressed and stores eight literals using 32 bytes, because every literal needs 4 bytes memory. The right list uses 24 bytes for storing three literal triple of 8 bytes. The compression factor of 33% is reached, if the number of literals to compress is a multiple of three.

```
1
2  class CppCompressClause
3  {
4  private:
5    struct lits_t{
6      unsigned lit1 : 21;
7      unsigned lit2 : 21;
8      unsigned lit3 : 21;
9      unsigned pad : 1;
10   }__attribute__((packed));
11   float activity;
12   uint32_t size;
13   lits_t* literals;
14 public:
15 literal_t literal(uint32_t index)
16 {
17   assert(index < size);
18   lits_t tmp = literals[index/3];
19   uint8_t mod = index%3;
20   if(mod==0){
21     return (literal_t)tmp.lit1;
22   } else if( mod == 1){
23     return (literal_t)tmp.lit2;
24   } else {
25     return (literal_t)tmp.lit3;
26   }
27 }
28 // other methods
29 };
```

Listing 6.4: Uncompression of Literals in Clause

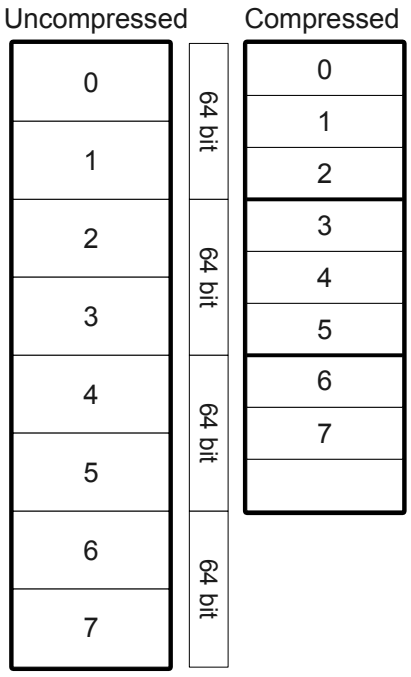


Figure 6.8: Compression of Literals in Clauses.

The implementation of the clause in Listing 6.4 provides a structure *lits_t* that stores the 21 bit values in a 64 bit double word (line 5-10). The elements of the clause remain the same except the literal array that became a pointer to an array of literal triples. Accessing a literal needs to extract the right triple (line 18) and to select the matching 21 bit value (line 19). The modulo operation is used although it is slow, because there is no faster way to determine the according element out of a triple. The matching 21 bit value is returned (line 20-26). The order of the if statements is important, because of the clause access pattern (Figure 5.3). Most of the time the very first element is accessed so this case is handled first and thus the average runtime of the routine is optimal.

The obtained result is not satisfying, because the runtime does not improve at all. The cache miss rate decreases slightly to 40.09% but the work cycles increase by 17.55% such that the interference of both effects discharges.

With respect to the fact that the size of a clause can be represented by the same amount of bits as literals the next approach, which is named **size compressed clause**, compresses also the size and stores the first literal triple in the clause head, similar to

the cache clause approach. Therefore, the result of this run cannot only be compared to the basic version or the compress clause approach, because a memory hop is saved, if one of the two first literals is accessed. The size of the clause head is not optimal for 64 bit systems in this schema because it is no factor of the cache line size. Using a 32 bit system the size of the pointer to the literal triple array has a size of 4 bytes and the size of the clause head would be 16 bytes. This size would be optimal concerning the cache line size. On the tested 64 bit system the runtime improved by 9.87%. The number of L2 cache accesses decreases to 89.78% and the miss rate is 33.33%. The distribution of the runtime among the components changes such that the unit propagation looses 2%. These 2% are shared among the remaining components. Compared to the cache clause approach the compress size clause approach is not the better choice.

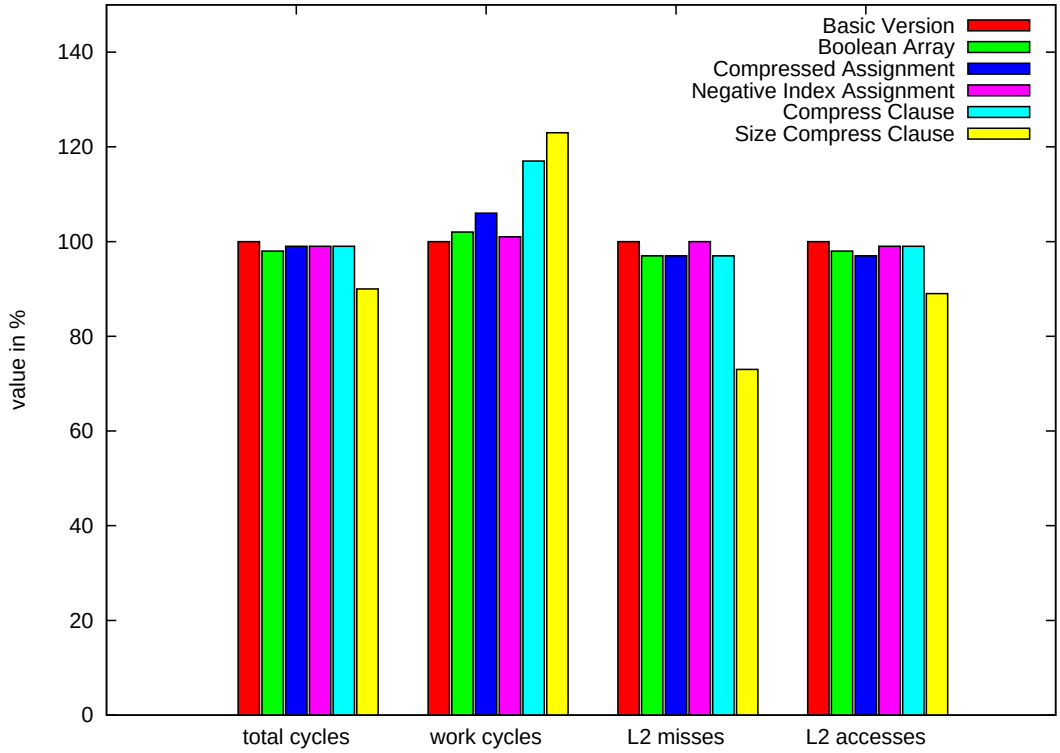


Figure 6.9: Comparison of Data Compression Improvements.

Figure 6.9 summarizes improvements due to data compression. It shows that compression leads to a small positive impact on the runtime due to less memory accesses and a better cache miss rate. The compression of data always leads to additional work cycles. The gap between saving a memory hop in case of the size compress clause and only compressing the data is nicely shown.

6.4.3 Slab Memory

Storing important data compact can be done by using an own memory allocator instead of malloc that is provided by the operating system. The slab allocator, which has been

introduced in [8], can be used for the allocation of memory of the same size. It organizes the single allocated objects in slabs of a fixed size. The size of the clause head is known at compile time, thus the slab allocator is used for them in an approach called **slab clause**. An advantage of this allocator is, that it does not introduce additional storage per element as it is done by the system allocator malloc(). On the test machine malloc() stores 8 bytes before every allocated memory block.

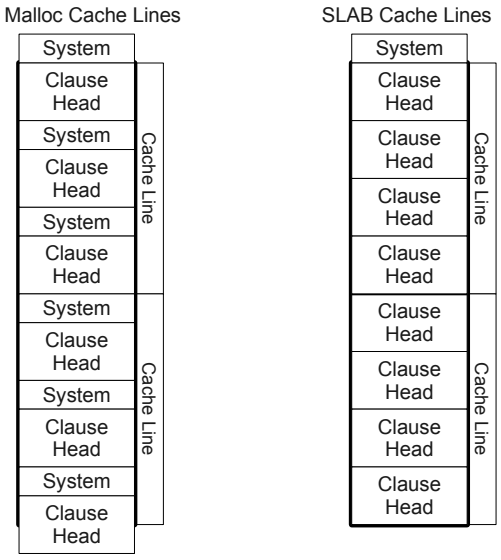


Figure 6.10: Comparison of Cache Lines with System and Slab Allocator.

Figure 6.10 shows two cache lines, each filled with clauses by a different allocator. The slab allocator stores the 8 bytes system information only once for a larger memory block and thus the clause heads are stored compact. If the block is large enough all cache lines contain only user information and no system overhead. The system allocator cache line contains up to three times 8 bytes system overhead and thus wastes a quarter of the cache line.

The processor events of this run are not different compared to the runtime of the basic version. Accessing different clauses of the same cache line does not happen often enough to gain some improvement on the cache misses and thus the overall runtime.

The slab allocator can also be applied to any other structure where the size is known and thus the allocator with the according slab size can be chosen. The literals of the clause are allocated using the slab allocator in the **slab literals** schema. Since the allocator reserves some memory in advance, an upper bound of slab sizes is introduced. Memory allocation above this bound are passed to the system allocator.

The result of the last approach is the same as for the first one. As shown in Figure 5.3 most of the time the literals in a clause are not accessed one after another. Only the first literals are accessed often. Thus, placing literals closer together does not lead to an improvement. Most of the time the next memory access touches another clause. Additional instructions are not introduced because the slab allocator is used instead of malloc.

6.4.4 Reuse Structures

In the conflict analysis the clause resolution is implemented using a vector that stores the literals that the learnt clause contains. For the minimization this vector is copied twice. One copy holds the literals that are resolved in the current step. The next one contains the literals of the previous run, if the last resolution process has to be undone. Thus, these vectors are created and destroyed for every conflict analysis. Their lifetime is very small. Since the extension of a vector involves memory allocation and copying this process is expensive. In case of a vector it is easy because instead of destroying the old vector and creating a new one the old vector just has to be cleaned. Cleaning a vector keeps the capacity of the vector so that the vector does not need to be extended again. The approach is called **reused vector**.

The overall runtime decreases by 4.53%. In the basic version the analysis needs 5.74% of the total runtime. Surprisingly in the improved version it still needs 5.95%. Compared to the runtime of the analysis in the basic version the analysis consumes 5.67% of the runtime. The change does not affect the analysis at all. A side effect of keeping the vectors and not extending, another vector is the reduction of memory fragmentation. Less memory fragmentation results in faster memory allocation, because the allocator finds suitable free parts of main memory faster and data is stored more compact so that the chance of hitting the same cache line twice increases and the prefetch unit prefetchs some user data instead of unused memory.

6.4.5 Compiler Options

The runtime of -O1 is 2.27% slower than -O3. This result can be explained by the differences of the optimization that is done. Except aligning loops the -O3 level includes all optimizations that are done by -O1. It further includes minor changes that mainly improve the execution of conditional execution branches. Thus, the -O1 run has 9.91% more work cycles and a slightly better cache miss rate of 40.88%, which interfere with each other.

The reduction of memory overhead led to a slight increase of the runtime as shown in Figure 6.11. Only reusing the vector in the conflict analysis reduces the number of memory accesses, the cache miss rate and thus the overall runtime.

6.5 Search Path Changing Improvements

Some improvements that change the search path turned out during analyzing the solver. Accessing literals during propagation is expensive, because of the two memory hops. Their avoidance is discussed in the subsection 6.5.1. The second subsection 6.5.2 discusses the reduction of literal accesses and subsection 6.5.3 explains another decision heuristic that could lead to an improved cache utilization. All statements comparing the two solver configurations can be applied only to the used benchmark. Using other instances may result in different results.

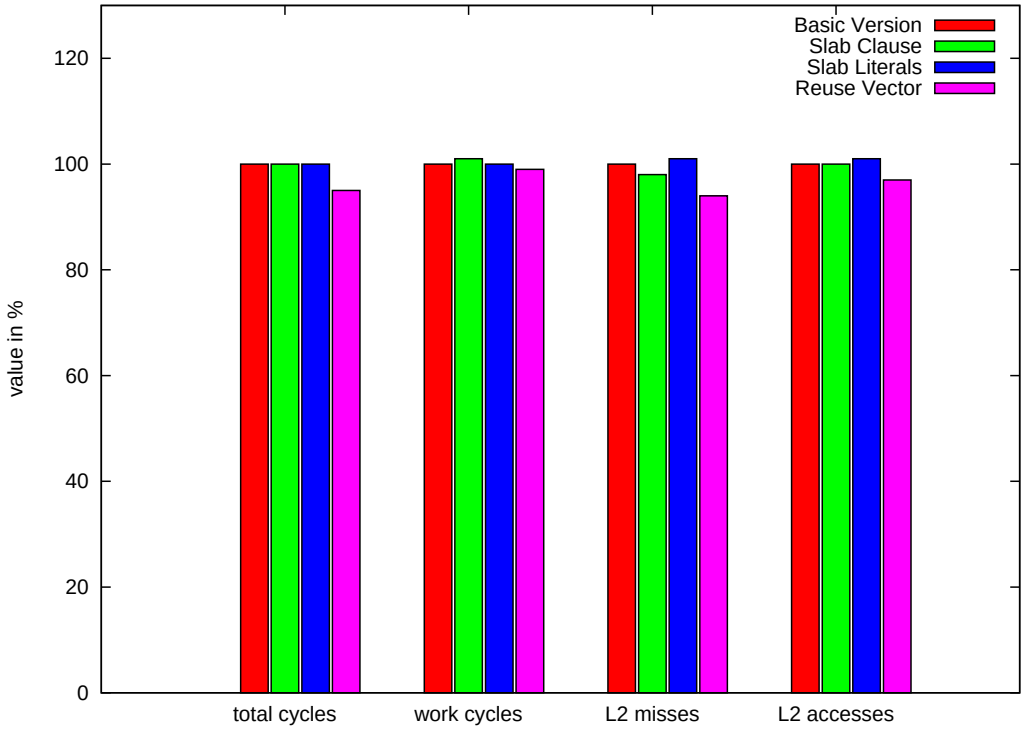


Figure 6.11: Comparison of Overhead Improvements.

6.5.1 Watch List Literals

Figure 5.2 (page 41) shows the two memory hops to access a literal in a clause. A first approach to improve the accesses has been introduced in section 6.3.1 (page 47) where the literals are stored in the clause head. Another approach is to store some of the literals next to the address of the clause in the watch list as in [9]. The access of these literals does not introduce another memory hop. If such a literal is satisfied the matching clause must not be accessed at all. Thus, they are checked before the clause is processed. There are several ways to manage the literals in the watch list. When the clause is added the first two literals of the clause are stored in the watched list. The approach is called **cache watch list**.

The work cycles of this approach almost three folded (279.12%), but the total time of this schema only increased by 57.24%. The number of memory accesses three folded as well (316.63%), since the solver processes clauses most of the time. The difference between the rises of the different cycle types can be explained by the cache miss rate that decreased to 10.25% as expected because of the skipped clause accesses.

Another weakness of the watch list implementation is the removal of items when a clause is watched by a new literal. According to Figure 5.2 (page 39) this process needs about a quarter of the whole search time. Applying a removal that does not keep the order of the elements leads to less memory copying and thus less memory accesses. The new removal swaps the element that should be erased with the last element of the vector.

Afterwards the size of the vector is reduced so that the new last element, which is the element to remove, is erased from the vector. The name of this schema is **unordered propagation**. Its result is negative in all aspects. Although the number of work cycles stayed almost the same, the total runtime increases by 77.05%.

The runtime distribution stayed the same, except the part of the Vector.erase() method. The original runtime of the watch list maintenance almost disappears completely. That this better maintenance does not lead to an overall improvement of the propagate function is caused by processing on other parts of the search tree and thus visiting more clauses and accessing memory with a higher L2 cache miss rate (68.3%).

A third approach is to reversely process the clauses in the watch list. Since removing an element from the list pushes the following elements one position forward the number of clauses to push is reduced, because some of the initial following clauses have already been removed. The overall runtime of this schema, which is called **reverse propagation**, increases by 88.42%. The number of work cycles increased by 41.86% and indicate that the search path of this approach seems to be worse than the one of the basic version. The part of the runtime that is needed by the Vector.erase() method is reduced to 10.81%.

6.5.2 Reducing Literal Access

The unnecessary swapping of the watched literals in the clauses is removed in the next approach, which is called **no swapped literals**. Instead of swapping the watched literals (line 6-8, Listing 5.1) in the clause when it is processed the other watched literal and position of the current watched literals are remembered. The consequences of this schema are less accesses to L1 cache and another search path, because of a different order of literals in the clauses. They result in a worse number of work cycles (217.75%) and a larger runtime (186.47%). The number of memory accesses increases due to the longer work process.

6.5.3 Change Decisions

The decision heuristic is the component controlling the next part of the search tree that is accessed. There are approaches that seem to access the same part of the search tree more often. One example is the phase-saving decision heuristic [20]. Its main goal is to pick the polarity of the decision literals such that is the same as it had been previously. Therefore, the old polarity of all assigned variables needs to be backed up when the variables are undefined. Applying this schema satisfies the same set of clauses that has been satisfied before resetting the variable. These clauses have not been accessed during the last propagation and thus they are probably not stored in the cache. In the current propagation they are not accessed again, because they are satisfied again. The schema does not access clauses that are known to be not stored in the cache. Due to the two watched literal propagation some of the satisfied clauses could still be accessed, because the satisfying literal is not necessarily the watched literal. Thus, propagating another literal of a satisfied clause still touches it and loads it into the cache.

6.6 Combination of Improvements

Several improvements can be combined. The only restriction is that not all clause improvements can be combined. The most promising combinations have been chosen to find the best improvement.

A remaining weakness of the cache clause approach is that the clause head does fit on the cache line twice but the system allocator memory overhead prevents this. Thus, the slab allocator and the cache clause are combined to store two clauses on a single cache line. This combination results in a speedup of 23.7% and thus gain another 4% compared to the separate improvements. The cache miss rate decreased to 31.07%. This effect is caused by the 5% reduction of the L2 cache misses. The other processor events behaved as in the cache clause approach.

The slab allocator has also been combined with the flatten clause. Due to the fact that the flatten clause is not separated in clause head and body the effect of storing two clauses in one cache line is only gained for clauses with two or four literals. Thus, the impact on the runtime is not as big as in the above approach. Compared to the flatten clause approach the introduced combination gains 2% in the L2 cache miss rate such that the overall runtime decreased to 77.38% of the basic versions runtime. The other processor events changed only slightly compared to the flatten clause schema.

Table 6.1 gives the results of some combinations of improvements with respect to total cycles, wait rate, L2 cache accesses and miss rate. The search path changing improvements have not been considered for the combination, because their impact may be too different on another benchmark. The six combinations combine the following improvements. The cache clause is used with four literals and uses the acronym CC. The flatten clause improvement uses the acronym FC. LR is the acronym for the lazy removal improvement. VR refers to the vector reuse improvement and slab refers to the slab allocator. P1 is the acronym for the prefetch method 1 , NA refers to the negative index assignment improvement and CBA is the acronym to the compressed Boolean array approach. Finally, CA refers to the compressed assignment schema. The first two columns of the table are relative values with respect to the basic version. The values of the remaining columns are absolute numbers, because they already represent ratios.

The following combinations have been chosen.

- Combination 1 = CC + slab + LR + VR + P1
- Combination 2 = FC + slab + LR + VR + P1
- Combination 3 = CC + slab + LR + VR + P1 + NA
- Combination 4 = FC + slab + LR + VR + P1 + NA
- Combination 5 = CC + slab + LR + VR + P1 + NA + CBA
- Combination 6 = CC + slab + LR + VR + P1 + CA + CBA

Configuration	Total Cycles	L2 Accesses	L2 Miss Rate	Wait rate
Basic Version	$9.222 \cdot 10^{13}$	$7.2225 \cdot 10^{11}$	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 6.1: Results of Improvement Combinations.

Table 6.1 shows that the chosen combinations lead to almost 60% runtime improvement. The number of L2 accesses differs only about 5%. The compression of data structures leads to less memory accesses. Less accesses also result in a lower wait rate. The compression advantages do not seem to interfere much with the runtime. The L2 cache miss rate differs only 2%. Combination 6 is the fastest combination and has the lowest wait rate. Its L2 cache miss rate is the highest, because it is also the combination with the least memory accesses.

6.7 Final Version

Since the runtime difference among the presented combinations is not significant only Combination 6 is analyzed further. Table 6.2 shows the distribution of total cycles and work cycles among the solver parts. It also gives the improvements compared to the basic version of the solver.

Compared to the basic version of the solver, the distribution of the total cycles moves slightly from the unit propagation to the other components. The conflict analysis needs 13% of the runtime. The runtime of the propagate() function is reduced by 60% compared to the basic run. This improvement is caused by the positive impact on the clause read access of 16%. The newly introduced prefetch instruction consumes 10% of this improvement. Another part of the positive effect is the decreased number of cycles for the Vector.erase() method. The methods task is spread in the propagate_long() method, such that it is not possible to trace it and assign it again to the watch list maintenance.

Combination 6 and the basic version of the solver differ only in 232 lines of code of the activated components. Most of these lines are needed for compressing the assignment (93 lines) and implementing the slab allocator (80 lines). The other changes can be implemented with less than 25 lines. The effect of the few additional lines of code is significant for the overall performance of the solver.

Unit propagation remains 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 more optimization opportunities since it executes only 4% of the work cycles but consumes 24% of the total runtime. Improving the decision heuristic

	Total Cycles	Improvement of total cycles	Work Cycles	Improvement of work cycles
Basic Version	$9.222 \cdot 10^{13}$		$7.535 \cdot 10^{13}$	
Combination 6	$3.66 \cdot 10^{13}$	$5.562 \cdot 10^{13}$	$9.64 \cdot 10^{12}$	$6.571 \cdot 10^{13}$
Combination 6	100%	60.31%	100%	42.62%
Search	98.63%	60.26%	97.19%	42.42%
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%
Event Heuristic	0%	1.33%	0.01%	1.33%
Unit Propagation	80.87%	59.55%	74.48%	44.56%
propagate()	77.86%	59.43%	72.09%	44.89%
propagate_binary()	14.42%	-0.01%	13.07%	-1.08%
propagate_long()	61.47%	59.46%	54.14%	46.32%
Clause.literal(i)	9.17%	16.04%	8.34%	-3.87%
Vector.erase()	0.22%	24.18%	0.34%	49.51%
prefetch()	23.14%	-9.18%	3.46%	-1.98%

Table 6.2: Comparing Cycle Distribution of Basic and Final Version.

may lead to a much better search path, such that applying the decision heuristic and conflict analysis may take as long as unit propagation does.

7 Summary

This chapter summarizes the work. In section 7.1 implementation rules that lead to the major runtime improvement are presented. Section 7.2 presents results of related experiments. Finally section 7.3 concludes the work.

7.1 Implementation Hints

The following rules led to major runtime improvements.

1. Avoid memory hops.
2. Separate frequently accessed and other data.
3. Fit data structures to the cache line size
4. Store frequently used data compact.
5. Prefetch data if the memory accesses are not linear.
6. Do not touch data unnecessarily.

Avoiding memory hops always results in less cache misses and thus in a better cache miss rate. Less cache misses result in a better work rate and thus the CPU does not spend that much time on waiting. Both the flatten and cache clause approach are a prove that this rule is very effective if it is applied alone.

The separation of frequently used data from the other data increases the chance of accessing the same cache line twice, because this line contains frequently used data. Since the prediction which cache line will be used again is difficult this approach can only be followed by a statistic approach. The cache clause approach showed the positive impact of this rule.

Fitting the size of data structures to the cache line size is important to avoid loading another line from main memory. The positive impact of this schema is shown in combination with the first rule in the combination of cache clause and slab allocator. Two clause head including four literals can be stored on a cache line and thus if the other clause has to be accessed, it is already load into cache.

The combination slab allocator and cache clause showed also that it is important to store other data between frequently used data. The application of the slab allocator alone did not lead to any effect. It removed the system memory overhead but the compact stored data has not been the frequently used one. Adding the slab allocator

approach to the cache clause approach led to another improvement, because the cache clause approach already determined the frequently used part of the data.

The fifth rule is very important, especially if the other rules cannot be applied. It enables the programmer to give the hardware hints about the structure of the memory accesses of the program. Thus the miss penalty for cache misses reduces, because the memory hierarchy has been notified about the data access before that access is really executed. Both prefetching methods prove that this improvement is important. The prefetch method 2 also introduces another parameter to suite the solver even more to a certain benchmark.

Ignoring the last rule has a dramatic impact on the runtime of the solver. Avoiding multiple unnecessary accesses of the clauses in watch lists decreases the number of memory accesses a lot, and due to the miss penalty for higher memory hierarchy layers also the runtime decreased dramatically. Every memory access can result in a main memory access and thus introduce many stall cycles. Thus keeping the number of accesses as small as possible is crucial for the runtime of the solver.

7.2 Further Work

Another hardware component that tries to reduce wait cycles is the TLB [15, p. 445]. It avoids repeating memory address translation, which are very expensive. An analysis using another computer system (Intel Core i7 860, 8MB L2 cache, clock frequency of 2.80GHz) showed the influence of the TLB. The benchmark has been run using different page sizes and thus the amount of memory for which the address translation is buffered in the TLB is changed. Enabling 2MB pages instead of the usual 4KB pages improved the runtime of both the basic and final version of the solver by 10%. This result shows that the TLB behavior needs to be analyzed to further increase the hardware utilization of the solver.

Recent CPUs implement a branch prediction unit. At the moment is not known how often this unit miss-predicts the right branch and thus can be optimized. In addition, the possible impact of this improvement is unknown.

Finding a metric for comparing different search path is of major interest. The work cycles meet the monotony criteria for a metric, but they also depend on the overhead an implementation spends on improving the hardware usage. The work cycles may be considered as a metric if the used data structures of the two versions are the same and thus do not introduce additional work cycles. The suitability of this datum as a comparing metric need to be analyzed further. A disadvantage is that it is a metric that is hard to calculate without executing the algorithm and measuring its value.

A subsequent research of this work could be the analysis of the suggested improvements on a parallel version of the solver using a multi-core computer platform with a shared cache. The current work provides a basis for further cache analysis of SAT solver. Since current CPUs become more parallel and implement shared caches, the further analysis may be of future interest.

7.3 Conclusion

The report presents the analysis of a modern SAT solver with respect to its hardware utilization on industrial problems. The research analyzed weaknesses of the solver concerning the memory hierarchy. These weaknesses have been reduced and thus the hardware utilization and the overall runtime improved. The major improvements include efficient clause representation, prefetching and lazy watch list maintenance, since the clause and the watch list are the most used data structures in a SAT solver. Combining several micro optimization improvements led to 60% speedup of the algorithm.

This speedup is only of minor interest for researchers, because the complexity of the algorithm remains the same. The linear speedup reduces only the runtime of further analysis. For users of SAT solvers this speedup is very important, since they are interested in smallest possible runtime of the solver. Industrial users of SAT solver do not care where the runtime improvement comes from. Every second they can save speeds up their overall process and reduces their costs. These people do not care where the speedup comes from. Thus saving more than the half of the runtime with adding only 250 lines of code is a very positive result of this work.

List of Figures

Bibliography

[1] AMD64 Architecture Programmer's Manual, Volume 2: System Programming. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf.

[2] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide. http://www.intel.com/Assets/PDF/manual/253668.pdf.

[3] The International SAT Competition Webpage. http://www.satcompetition.org/.

[4] Valgrind: Tool Suite. http://valgrind.org/info/tools.html, 2009.

[5] 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.

[6] A. Biere. P{re,i}coSAT@SC'09. SAT 2009 Competitive Event Booklet, http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf.

[7] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185. IOS Press, Amsterdam, 2009.

[8] J. Bonwick. The slab allocator: an object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference*, 1994.

[9] G. Chu, A. Harwood, and P. J. Stuckey. Cache conscious data structures for boolean satisfiability solvers. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:99–120, 2009.

[10] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[11] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[12] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *In proc. SAT'05, volume 3569 of LNCS*, pages 61–75. Springer, 2005.

[13] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[14] Yuri Gurevich and David G. Mitchell. A sat solver primer. *Bulletin of the EATCS*, 85:112–133, 2005.

[15] John Hennessy and David Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 1996.

[16] J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, 1996.

[17] Matthew W. Moskewicz and Conor F. Madigan. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535. ACM, 2001.

[18] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[19] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 89–100, San Diego, California, USA, June 2007.

[20] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In João Marques-Silva and Karem A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[21] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In *In proc. SAT'09, volume 5584 of LNCS*, pages 237–243. Springer, 2009.

[22] Nathan R. Tallent, John M. Mellor-Crummey, Laksono Adhianto, Michael W. Fagan, and Mark Krentel. Diagnosing performance bottlenecks in emerging petascale applications. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[23] G. Bowden Wise. An overview of the standard template library. *SIGPLAN Notices*, 31(4):4–10, 1996.