TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Artificial Intelligence Knowledge Representation and Reasoning

Modern Parallel SAT-Solvers

Steffen Hölldobler, Norbert Manthey, Van Hau Nguyen, Peter Steinke, Julian Stecklina

KRR Report 11-06

Mail to
Technische Universität Dresden
01062 Dresden

Bulk mail to
Technische Universität Dresden
Helmholtzstr. 10
01069 Dresden

Office
Room 2006
Nöthnitzer Straße 46
01187 Dresden

Internet
http://www.wv.inf.tu-dresden.de

Modern Parallel SAT-Solvers

Steffen Hölldobler, Norbert Manthey, Van Hau Nguyen, Peter Steinke, Julian Stecklina
Faculty of Computer Science
Technische Universität Dresden, 01062 Dresden, Germany
sh@iccl.tu-dresden.de

Abstract

This paper surveys modern parallel SAT-solvers. It focusses on recent successful techniques and points out weaknesses that have to be overcome to exploit the full power of modern multi-core processors.

1. Introduction

The Boolean Satisfiability Problem (SAT) is one of most-researched NP-complete problem in Computer Science [13]. Over the last twenty years improvements on all levels, from the logic over calculi, heuristics and data structures to low-level processes, have turned modern sequential SAT-solvers into practical tools in many different application areas like hardware and software verification, planning, scheduling, termination analysis, configuration, or bioinformatics (see [6]).

An analysis of modern sequential SAT-solvers has revealed that the efficiency and speed of these systems hinges – among others – on the appropriate usage of the available hardware and low level processes like access to caches and main memory, the usage of slab allocators, prefetching schemas or lazy maintenance (see e.g. [29], [10] for details).

Of course, the steady improvement of single processor systems in the last decades has also helped to increase the performance of SAT-solvers considerably. However, from an architecture point of view we currently observe a dramatic change from single processor systems to multi-core designs including integrated memory controllers and large caches leading – among others – to non-uniform memory access latencies.

Naturally, modern SAT-solvers should utilize these multi-core design. However, from a theoretical and worst-case analysis point of view, the SAT-problem is inherently sequential. This leads to several serious research problems: How can SAT-problems be solved in a multi-core system effectively? Which calculi support multi-core systems? Which data structures, heuristics, and low level processes make appropriate use of the non-uniform memory access latencies? Which kind of communication shall be employed? How can we compare the performance of parallel SAT-solvers? Can the encoding of applications into SAT-problems be improved in order to better utilize multi-core architectures?

After stating some preliminaries in Section 2, sequential SAT-solvers are characterized in Section 3. The main contribution of this paper is a survey on modern parallel SAT-solvers in Section 4. In Section 5 we briefly review modern multi-core architectures. A second contribution is a first attempt to specify requirements for the implementation of SAT-solvers on multi-core designs in Section 6.

2. Preliminaries

A problem that should be solved by a SAT-solver is usually specified in conjunctive normal form (CNF) of propositional logic. This form restricts propositional logic to variables and the three connectives *negation* ($\neg$), *disjunction* ($\vee$), and *conjunction* ($\wedge$). A *literal* consists of a variable A and is either positive (A) or negated ($\neg A$). A *clause* is a disjunction of literals. Finally, a *(CNF) formula* is a conjunction of clauses. It can be shown that any propositional formula can be transformed into an equivalent formula in CNF. [1]

An *interpretation* is a mapping from formulas to the set of truth values $\{\top, \bot\}$. It is usually represented by a mapping from variables to truth values as the connectives are interpreted in a standard way. SAT-solvers construct interpretations and in this process make use of *partial interpretations*, where only some

1. In this paper we ignore the problem of characterizing an application by a propositional logic formula as well as its transformation into CNF. These processes are quite involved and subject of research. In particular, knowledge encoded in the application might be extremely valuable for heuristic and other decisions made by a SAT-solver.

of the variables are assigned to truth values. Partial interpretations are applied to formulas by replacing the already assigned variables by their truth values[2] and simplifying the obtained formula accordingly.

The Boolean Satisfiability Problem is the question whether there exists an interpretation for a propositional formula such that the formula evaluates to $\top$ with respect to the interpretation.

3. Sequential SAT-Solving

Modern sequential SAT-solvers are still rooted in the *Davis-Putnam-Loveland-Logemann (DPLL)* algorithm [15], [14]. The algorithm consists of rules which are applied to generate and traverse a binary (semantic) search tree. Each branch of the search tree represents a (partial) interpretation. Let F be the given formula and I the interpretation represented by branch B. We distinguish the following cases: (i) If I maps F to $\top$, then a model has been found and F is satisfiable. (ii) If I does not map F to a truth value, then B is expanded by the so-called *split rule*: a currently unassigned variable is assigned a new truth value and a backtrack point is recorded. Afterwards, I is extended by all the implications that can be found with respect to the formula. This is mainly done by the *unit propagation rule*, but additional simplification rules are possible. (iii) If a clause of F is mapped to $\bot$ by I, then this clause is called *conflict (clause)* and B can be closed. Thereafter, naive backtracking is applied to explore most recent alternative branches in the search tree.

The idea to analyze the conflict clause further led to the *conflict driven clause learning (CDCL)* algorithm that was first presented in the SAT-solver GRASP [55]. By applying resolution to the conflict clause and to the clauses which have been used in the implications, new clauses are learned. These learned clauses can be added to the given formula leading to an improved backtracking behavior, where larger parts of the search tree are closed by a single conflict. The most commonly used version of this algorithm is described in [64].

In addition, very special data structures like the *two-watched-literal schema* [47] and an involved memory management (see e.g. [29], [10]) are implemented to minimize the number of accesses to memory as well as the latency of memory access.

Heuristics are applied at many different stages like, for example, to decide which variable is assigned in a split step, which assignment is first explored, which clauses are learned, or which learned clauses are to be removed again (see e.g. [47], [51], [17]).

2. Formally, by formulas which are valid or unsatisfiable, resp.

Restarts are another technique to improve the performance. In this case, the search for a satisfying assignment is started from scratch again. Since previously learned clauses are kept, the search tree is usually explored in a very different way (see e.g. [52]).

In certain cases it appears to be promising to *look-ahead* by assuming unit literals. This techniques is valuable for certain formulas, because the look ahead gathers extra information about the formula and can exclude splitting variables early.

The most commonly used SAT-solver that implements most of the mentioned techniques is MINISAT 2.2 ([48], [17]). This solver has been first implemented in 2003 and has been improved each year. It is used as a basis for many sequential and parallel SAT-solvers.

4. Development of Parallel SAT-Solvers

While sequential solvers have been improved iteratively, also parallel systems have been developed. Based on single-core architectures, the first parallelizations were based on network communication. With the occurrence of multi-core CPUs, shared memory was also used for communication. Orthogonally, with the move from DPLL- to CDCL-solvers including restarts, different techniques were developed to parallelize the algorithms for solving SAT-instances.

In 2006, a first overview on parallel SAT-solvers was presented in [56]. However, much of the work on parallel SAT-solvers was done in the last five years and we focus on these recent developments. We further restrict our attention to complete solvers, whose parallelization should reach a superlinear speedup for satisfiable instances.

4.1. DPLL-based parallelizations

The recursive application of the split rule in the DPLL algorithm provides a natural way to parallelize the search. Initially, each computing node receives the given formula. Thereafter, only partial interpretations are communicated such that each computing node receives a different partial interpretation. The first parallel SAT-solvers used single-core CPUs which communicated via a network; see [56] for a survey.

The introduction of shared memory computing systems made it possible to replace the communication via networks by shared memory communication. In [58] two parallelizations of the SAT-solver SATZ were compared: a version using single-core CPUs with network communication based on MPI [21] and a version using a multi-core CPU with shared memory communication based on OpenMP [50].

In a first step, several splitting heuristics were considered: the original SATZ heuristic, picking variables randomly, and picking the variable that occurs maximally in small sized clauses [20]. Load balancing is obtained by picking enough splitting variables, such that sufficiently many subformulas can be created by assigning all possible truth-value combinations to the splitting variables. A first experiment showed that in both parallel versions the best results were obtained by splitting the search tree according to the SATZ splitting heuristics.

In a second step, the two parallel versions were compared. The result of the study showed that the network communication is more efficient than the shared memory communication. This can be explained as follows: running a DPLL solver on a single-core CPU gives this solver full access to all the resources of this computing node. Thus, the solver can exploit the memory hierarchy, especially the small caches, as much as possible to maintain a high performance. For the network configuration, the overhead comes only from the comparable high communication times. If a solver is executed on a multi-core CPU using shard memory communication, then the communication time is much smaller. On the other hand, as the cores share memory, there are frequently more occurrences of cache misses leading to a 15% decrease of the performance of a solver running on a particular core (compare [46]). Because there is only little communication (by communicating partial interpretations), the slowdown based on cache misses on the multi-core architecture is higher than the network communication costs.

4.2. CDCL-based parallelizations

With the success of the first CDCL-based SAT-solvers in 1996 [55], the situation changed dramatically. Due to the addition (and deletion) of learned clauses the search space is traversed much less orderly than in DPLL-based solvers and many new questions arose like: Which learned clauses should be send to other solvers? Which learned clauses should be incorporated into the own search? When shall learned clauses by deleted?

4.2.1. Network communication.

GRIDSAT [9] was the first solver employing a grid. It uses a master-slave approach. A task queue is provided by the master and slaves can be added if a SAT-instance is hard to solve (see [56] for an in-depth discussion). A performance analysis revealed that the speedup can be from sub- to super-linear for both satisfiable and unsatisfiable SAT-instances. Still, the authors claim that the parallel solver is more efficient than a sequential one because (i) by using several CPUs more parts of the search space can be analyzed in the same time, (ii) by splitting and removing redundant parts of subformulas each node can solve smaller formulas, and (iii) by splitting an instance resources can be added whenever they are required. A problem of GRIDSAT appears to be that learned clauses are not kept if a slave finishes to analyze an unsatisfiable subformula. In this case, the slave becomes idle and when solving the next job it cannot use its previously learned clauses.

PMSAT [22] is based on MINISAT 1.14 and MPI. Like GRIDSAT, it uses a master-slave approach, but assigns a fixed number of slaves to a SAT-instance. Thus, new tasks are only assigned if a slave becomes idle. The master creates the assumptions that are used to split an instance. In particular, for k slaves, $3k$ splitting variables are selected such that 2^{3k} jobs have to be handled in the worst case. The master stores the splitting variables and sends a partial interpretation based on these variables to the next free slave. Load balancing is implemented by providing sufficiently many tasks.

After a slave has solved its task and proved unsatisfiability, it can send 50 of its most active learned clauses with a size of less equal 20 literals to the master. The underlying MINISAT is able to solve an instance provided a certain set of assumed literals. The created learned clauses are also valid for all other parts of the search space, because these learned clauses contain the assumed literals if necessary. Thus, the master can forward these learned clauses to the running slaves. Furthermore, the master removes tasks which became unsatisfiable based on the newly received learned clauses from its tasks queue.

PMSAT implements two approaches for selecting the splitting variables and for applying the selected variables to create subtasks. Either the most frequent variables or variables that occur most frequently in large clauses are selected. The motivation for the latter is to reduce the size of the formula as much as possible. The selected variables are used to generate subtasks by either creating a simply binary search tree based on the possible assignments of these variables or by *scattering* [34]: Let $v_1, \ldots, v_n$ be the selected variables. In a first step, scattering creates a binary split by assigning the unit clauses $[v_1], \ldots, [v_n]$ to the given formula on the one hand, and by assigning the clause $[\neg v_1, \ldots, \neg v_n]$ on the other hand. Thereafter,

this schema is applied recursively to the latter, possibly with new variables, until sufficiently many splits are created. Based on the possible four configurations in PMSAT, experiments have been carried out by comparing the performance of the best and the worst configuration. For the best configuration and unsatisfiable SAT-instances an efficiency[3] upper bound of two has been reported. For satisfiable instances almost always a super-linear speedup could be reached. However, for the worst configuration and satisfiable instances only sometimes a super-linear speedup could be found and unsatisfiable instances have most often an efficiency below 0.5. Additionally, there is the open question of how to determine the best configuration given a SAT-instance. Furthermore, measuring the runtime of the solver is a bit strange. The parallel runtime is the sum of the runtime of the master and the most occupied slave. However, the runtime of all the other running slaves has not been taken into account.

Another method to solve SAT-instances is by applying different search strategies in parallel and independently. In [32] different restart strategies have been implemented into MINISAT 1.14 to be executed in a grid. In particular, the restart schemas based on the Luby series [43] and the exponential series $2^{1.2X}$, where X is the number of the next restart, are investigated. Two restart parallelization schemas are analyzed: the *straightforward* and the *faithful* schema. Let $N \in \mathbb{N}$. The straightforward schema divides the restart schedule into partitions and executes N parallel tasks per partition as follows: The first partition consists of the first N restarts of the chosen schedule; these restarts are used for the first N parallel tasks. The second partition consists of the restarts number $N+1$ to $2N$; these restarts are used for the second N parallel tasks. This partitioning is repeated until the desired number of tasks is created. The faithful schema uses the same partition as the straightforward schema, but only one task per partition is created. One should observe that there is no communication between the solvers in the grid as they solve their tasks independently.

The results of this study show that the efficiency of this problem reaches from 0.5 to close to 1 if there are no delays in the grid. When the task submission delay is included into the runtime, no super-linear speedups are reported. Furthermore, the study shows that the more parallel solvers are executed, the less important is the used restart strategy. Additionally, it is reported that already a small number of parallel solvers is sufficient to solve a SAT-instance fast. This result indicates that the approach is unlikely to be highly scalable. However, since grids provide a huge parallel computing capacity, the authors consider their approach suited for solving a set of instances fast.

Running a solver in parallel on a grid has been improved in [31] by collecting learned clauses of unterminated jobs and share them with the other parallel solvers that are started after the termination. The study shows that sharing learned clauses increases the performance, but no superior heuristic has been found. Furthermore, it is shown that adding learned clauses most of the time increases the solver performance. In detail, hard instances that have not been solved by sequential solvers can be solved by the clause sharing grid approach.

The parallel solver C-SAT [49] combines two ways of parallelism, viz. cooperative parallelism by splitting the search space and competitive parallelism by executing different solver configurations on the same search space. The solver is based on MINISAT 1.14 and MPI, and is implemented for computing clusters. Two different configurations of MINISAT are obtained by using (i) the VSIDS heuristics [47] and (ii) a heuristics, where the activity is stored per literal, for the selection of variables. Search space splitting is implemented by selecting split variables from the current path in the search tree.

C-SAT is organized in three layers. In the first layer a grand-master connects to several masters as its slaves. The grand-master distributes the input formula to the masters. Furthermore, it receives learned clauses from the masters and distributes them among all masters. Additionally, all the learned clauses are checked for redundancy and are simplified if possible. All masters work on the same input formula. Each master maintains a group of slaves that work on subtasks. Selected learned clauses are sent to the master every 100 conflicts.

The highest performance of C-SAT has been achieved by using both decision heuristics in parallel as this allows the solver to learn more different conflict clauses. In particular, the efficiency of this approach is super-linear for satisfiable SAT-instances and more than 0.6 for unsatisfiable SAT-instances.[4] The experiments further revealed that by running more slaves the

3. The efficiency is the ratio of the sequential time T_s and the product of the parallel time T_p and the number of processors p: $eff = \frac{T_s}{T_p \times p}$.

4. One should observe that the means reported for C-SAT are geometric means, whereas all the other reported means are arithmetic means. Usually, the value of the arithmetic mean is higher than for the geometric mean.

chance of learning redundant clauses increases. Finally, the stability of the solver with respect to runtime increases, if more processing units are used to solve a SAT-instance.

In [33] MINISAT 1.14 is used to analyze search tree partitioning techniques. In contrast to all previously mentioned solvers that split the search space, the presented solver does not only split the input formula and solves the subformulas, but also solves the original SAT-instance in parallel. By doing so, the solver is able to solve a given input instance at least as fast as the sequential solver. In particular, [33] discusses three tree partitioning techniques. The first technique is called simple splitting and simply solves the input instance by executing several solvers in parallel. The second technique splits the tree according to the DPLL procedure and additionally applies a look ahead before the next split. This way, branches that result in a conflict at the next level are closed immediately and a better splitting variable is selected instead. The third techniques splits the search tree based on scattering. Because the grid environment limits the task execution time, subtasks might time out, in which case the subtask is split further. For experiments, splitting a formula into subtasks is limited to 5 minutes. Tasks may be solved within 90 minutes. The comparison of the partition techniques shows that the DPLL look ahead partitioning can solve instances faster than scattering. However, the latter is able to solve more instances of the used benchmark. A major disadvantage of the approach is that learned clauses are completely lost when a certain subtask is solved or times out.

This disadvantage has been tackled in further research [35]. Based on MINISAT 2.2 a grid is used again, but now selected learned clauses are submitted back to the master if a slave finishes its task. Two approaches are considered: In the *assumption tagging* approach the master adds the list of assumptions to the learned clause before sending it to its slaves. Because MINISAT treats these assumptions as decisions, all learned clauses are logical consequences of the input formula. However, the more assumptions there are, the longer the clauses might get. Very long clauses may be generated, which do not prune the search space much. As a result, the overall performance of the approach decreases. In the *flag-based tagging* approach the assumptions are not treated as decision but as clauses of the formula. In this way, the input formula can be simplified before the search is started. Now, a clause that depends on the assumptions is tagged. Whenever a new learned clause is derived, it is also tagged if a tagged clause participated in the derivation. This way, the untagged learned clauses, i.e., the clauses which are logical consequences of the input formula, are underestimated, but they are much shorter than in the first approach. Treating assumptions as part of the formula speeds up the solving process twice: Firstly, the formula in each task can be simplified and, secondly, the returned learned clauses are much smaller. Experimental results show that the approach slows down solving simple formulas but improves if more difficult SAT-instances have to be solved.

4.2.2. Shared-memory communication. When multi-core architecture became available, parallel SAT-solvers have been developed that use the shared memory as communication basis. As discussed in Section 4.1 running DPLL-based solvers in parallel on shared-memory communication slowed each individual solver down by about 15%. Because using two cores often leads to shorter runtimes than than using a single core, the performance of the SAT-solvers increased by exploiting more cores, although the achieved speedup might not be the best. Furthermore, the architecture changed. The first multi-core system were built by combining two single-core CPUs, where each CPU had its own memory bus and main memory. Thus, non uniform memory accesses slowed the performance down. This effect was decreased by combining several cores into a single CPU. Thus, memory access became uniform again and the access times of sequential and parallel solver became more similar again.

The first shared memory solver based on CDCL is PASAT [59]. Each running solver picks the next decision literal by choosing a literal from short clauses to prune the search tree by receiving unit clauses. The parallel execution is based on search space partitioning. To this end, a threat splits a part of the current path in the search tree, sets up another solver, and marks the corresponding branch as closed. The newly set-up solver starts analyzing the obtained subtask in parallel. Learned clause of at most size 5 are shared among the solvers by pushing them into a global storage. Each thread incorporates all shared clauses into its database by copying them into its private physical copy of the formula. One should observe that a received learned clause can close a task. The experiments showed that the effect of sharing learned clauses is complex. Depending on the given SAT-instance, the effect can increase or decrease the performance of the solver. Furthermore, the global storage for the learned clauses can become a bottleneck for the performance, because

only one client can write to it simultaneously. Without exchanging learned clause superlinear speedup can be reached for satisfiable instances, whereas the efficiency is close to 1 for unsatisfiable instances. When learned clauses are exchanged, the performance for satisfiable instances increased whereas it differs only slightly for unsatisfiable instances.

By combining several multi-core computing systems to a network, the performance of parallel solvers can be increased further. This approach has been applied to PASAT in [7]. Specifically, PASAT has been distributed and different decision heuristics are applied. Sharing of learned clauses is also extended to the network. For each PASAT instance, a mobile agent collects learned clauses from the other running PASAT solvers. The authors notice that the runtime distribution of the parallel solver heavily depends on clause learning and sharing, because different parts of the search space might be analyzed in different runs resulting in very different run times.

The solver YSAT has been used to study the efficiency of shared memory solvers in [18]. For the parallel implementation the authors chose to split the search tree based on a variable that is close to the root of the tree. The splits are stored in a global work queue. All the learned clauses are also stored in a globally accessible list. Both these data structures have to be read and written by all threads. The formula is also physically shared. The authors report a write blocking overhead of up to 10 % for their parallel solver. Furthermore, the standard deviation of the runtime of the YSAT on certain instances ranges from 20 % to 30 %.

The scalability analysis performed in [18] is based on several architectures and an input formula of the size 1.5 MB. It was noticed that by using n cores (where n is limited to 4) in parallel approximately n times more learned clauses are produced than in the sequential case in the same time. Consequently, the necessary storage for the learned clauses also increases n-fold. By using a single thread, the clocks per instruction (CPI) ratio is 1.4. This value is not 1, because sometimes the CPU has to wait for longer memory accesses. Still, this number is quite low, because the instance and most of the learned clauses fit well into the cache. When 4 threads are used, the ratio increases to 3.7 and the number of stall cycles increases [29]. This effect can be explained by the increased number of created learned clauses, whereas the size of the cache is fixed. It can be assumed that the measured effect is much smaller, if the input formula would be much larger. In this case, the size of the learned clauses is negligible and the CPI ratio would stay almost the same for both the single core and quad-core configuration. However, [18] concludes that using multi-core parallelization cannot be done efficiently.

The solver MIRAXT [41] proved to be a counter example. It parallelizes the solver MIRA [42] by splitting the formula based on the current path of the search tree. Before solving an instance, the preprocessor SatELite [16] is applied. The authors claim that by doing so *bad* splitting variables are already removed from the formula. As in YSAT, all clauses are shared physically. To implement the two-watched-literal unit propagation [47], each thread has to introduce an additional literal reference to store the currently watched literals. The main difference is the way how to share learned clauses. As in YSAT, all learned clauses are shared, but each thread can decide which clause it wants to incorporate. In particular, only clauses that appear to be *useful* are incorporated, viz. unsatisfied clauses which contain (after reduction) at most 10 literals. In contrast to PASAT, this approach also provides a communication without delays because all threads can access the shared clauses immediately. The performance analysis showed, that the two-core solver is more powerful than the single core solver, although the efficiency seems to stay below 1.

By extending MIRAXT to networks [54], its performance could be increased further. The parallel incarnation divide the search space by requesting splits from loaded clients. Furthermore, learned clauses are shared among the master and the client, where the master implements a receive filter. Although the efficiency of PAMIRAXT [54] is only about 0.25 for 8 cores, it is still able to solve many instances faster than sequential reference solvers as MINISAT 2.

The parallel solver PMINISAT [11] is based on MINISAT 2.0. It splits SAT-instances on the top-most decision variables of the current search path to create a task queue. Shared learned clauses are incorporated into the running solvers after reducing them with respect to the current partial interpretation. A novel idea of this solver is to keep learned clauses if they reduce to a unit clause under the partial interpretation of some task in the task queue. Thus, whenever an idle solver is assigned the next task, it can propagate the corresponding units by incorporating these shared clauses.

The way of splitting an instance has been analyzed in [46]. In addition, SAT4J// implements *parallel portfolio solving*. It first splits the instance and if the

runtime for the splitting mechanism reaches a certain limit, the solver switches to the portfolio approach (see Section 4.2.3) and tries to solve the instance by using different heuristics. The novel idea for splitting is to execute several CDCL solvers with the VSIDS decision heuristic in parallel. After a certain time, the activities of the variables are cumulated and, afterwards, the variables with the highest activities are chosen as splitting variables. In this way, the overall information about the variables is higher than if a single CDCL solver would be used to determine the splitting variables. The results of the work show, that the performance of the hybrid approach is higher than using either splitting or portfolio solving purely. Furthermore, the analysis shows that running four times SAT4J [40] on a quad-core CPU in parallel slows down each solver by 25 %.

4.2.3. Pure Portfolio solvers. After it became clear that the frequent use of restarts improved the performance of sequential SAT-solvers significantly, many researchers moved from cooperative parallelism by splitting the search space to competitive parallelism where all parallel solvers try to find a solution for the same SAT-instance. The latter are often called *portfolio solvers*.

With MANYSAT [27] portfolio solvers became popular. MANYSAT is based on MINISAT 2.0 and applies several restart, decision and learning heuristics to its four parallel instances. Learned clauses with at most eight[5] literals are shared among the instances. Each MINISAT instance has its private copy of all the clauses. The restart heuristics are the geometric, nested geometric, arithmetic, and Luby series [43]. For the decision heuristic the VSIDS heuristic with different ratios for random decision, namely 2% and 3%, is used. Finally, for two MiniSAT instances, learning is extended by the algorithms mentioned in [2]. The obtained efficiency for the chosen combination of the heuristics is reported with 1.5 when compared to MINISAT 2.1, the best sequential solver of the SAT Race 2008.

Further experiments on the threshold for sharing learned clauses in [26] showed that the performance can be increased if a dynamic limit is used instead of a static predefined limit. It has been shown that the average size of learned clauses increases over time, so that after a certain runtime no clauses would be shared anymore. Thus, the threshold has to be adopted during search. The authors suggest two measurements. The first one is based on the number of the exchanged clauses between two solvers. If this number drops below a threshold, the length limit is increased. Otherwise, it is decreased. The second criterion is based on the *quality* of the clauses, where the quality is measured by the size of the clause after reduction. As in the first measurement, the quality limit is loosened, if the number of qualitative exchanged clauses drops below a threshold. The performance of *ManySAT* was improved by using the size based control. By also adding the quality measure, 6 more instances out of the 201 instances of the SAT Race 2008 could be solved.

The overall performance of portfolio solvers can be further improved by exchanging additional information between the solvers. In [24], the solvers are divided into masters and slaves and additional information is sent from a master to its slaves. This information may include the last decision literals, the asserting literals of the last conflict analyses, or the literals that have been used to derive the very last conflict clause. With the additional information, the slaves can either search in the same search space of their master, can make better use of the learned clauses of their master, or search around the same conflict as their master, respectively. Experiments showed that solving around the same conflict results in the best performance. For the parallel running solvers, several topologies were tested. It turned out that using two masters and two slaves resulted in the best performance of the solver, both in number of solved instances and average runtime. In comparison to the original version of MANYSAT, this configuration is able to solve nine more instances on the industrial benchmark of the SAT Competition 2009, namely 221 out of 292 instances.

The latest development on MANYSAT is to avoid non-determinism introduced by the parallel execution of the CDCL algorithm and the sharing of learned clauses [25]. By introducing so-called *barriers*, the solvers execution becomes repeatable. Barriers are introduced at restarts and during the exchange of learned clauses. Because learned clauses are shared during a restart, all solvers wait at a restart until each solver reaches a predefined barrier. Thereafter, the learned clauses are shared deterministically. It was shown experimentally that if the barrier is set after a static period, the average waiting time among all threads is quite high; If sharing is carried out after 10000 conflicts, the waiting time still consumes 17 % of the overall runtime; For sharing at each conflict, the waiting ratio is 40.9 %; It is reduced to 26.3 % if the sharing of learned clauses is carried out every 100 conflicts. In order to minimize the waiting time, for each solver the number of learned clauses is measured and the periods are dynamically calculated. With this

5. The number was experimentally determined.

technique, the deterministic parallel solver performed as least as high as the non-deterministic version of MANYSAT.

A different portfolio approach has been implemented in the parallel solver SARTAGNAN [39]. This solver supports up to eight cores, where only six of them execute the CDCL algorithm. The seventh core uses *decision making with reference points* [23] to solve the instance. The last core tries to simplify the formula. All reported simplifications are incorporated into the other running solvers because the given SAT-instance and all learned clauses are shared. Furthermore, updating a clause or sharing learned clauses has been implemented without using locks. Clauses are incorporated into a solver, if their *LBD activity* [3] is good enough, or if a certain percentage of the clauses literals have an activity that is higher than half the maximum variable activity.

Other recent successful parallel portfolio solvers are PLINGELING [5] and a new version of CRYPTOMINISAT [60]. Both solvers execute the same solver configuration in parallel and differ only in the used random seed. Usually, only very short clauses are shared. In particular, PLINGELING shares only unit clauses. Both solvers have been successful in the SAT Competition 2011. A different portfolio approach has been also shown to perform very well. By simply running very different solvers in parallel, the solver PPFOLIO [53] was ranked high in most of the tracks of the SAT Competition 2011. The different solvers have been executed without any communication. Still, by combining a stochastic local search solver [30], a look ahead solver and several specialized CDCL solvers the overall performance has been well enough to achieve a good ranking.

4.3. Other parallelization approaches

In NAGSAT [19] the DPLL search has been parallelized by *nagging* [61]. One master is executing a DPLL search. Additional slaves are added that perform nagging by picking the first r decisions of the master in the same polarity, where r is chosen randomly. The chosen split is called *nagpoint*. Afterwards, the slaves perturbate the order of the remaining decision variables and solve the same subspace as their master. Four cases can occur: (i) a nagging slave finds a solution before the master, (ii) a slave proves unsatisfiability before the master, (iii) the master backtracks over the nagpoint and, thus, proves unsatisfiability before the slaves, or (iv) the master finds a solution before the slaves. Experiments on NAGSAT showed that for two computing nodes the efficiency for both satisfiable and unsatisfiable instances is higher than 2.3. However, the approach does not scale well up to 64 processors, because the measured efficiency for satisfiable and unsatisfiable instances is sub-linear. For satisfiable instances, an efficiency of 0.65 can be reached, whereas the efficiency for unsatisfiable instances is only 0.11.

An alternative splitting approach is to divide the variables into two partitions and try to find a model for each partition that can be extended so that the other partition can be satisfied as well [57]. The splitting works as follows. First, the set of variables V is divided into two sets $V = V_1 \cup V_2$. The intersection $V_{js} = V_1 \cap V_2$ of this two sets should be as small as possible because this set determines the number of possible partial interpretations that have to be extended. According to the variable sets, the clauses are partitioned into three sets: C_1 is the set of all clauses that contain only variables that do not occur in V_2, C_2 is the set of all clauses that contain only that do not occur in V_1, and C_3 is the set which contains all remaining clauses. Solving is done by creating a satisfying assignment for C_1. If this assignment cannot be satisfied with respect to C_3 it is rejected. Otherwise, an attempt is made to extend it with respect to C_2 and C_3. Obviously, finding satisfying assignments for the sets C_1 and C_2 can be done in parallel. Furthermore, in JACKSAT [57], the splitting is applied recursively and the models for the clause sets are created by using an all model finding SAT-solver such as RELSAT [4]. The approach has one weakness, namely finding a good splitting that reduces the size of V_{js}. The results in the publication are created on instances that do have less than 100 variables and at most 500 clauses. Still, the runtime of the parallel solver is much slower than the runtime of an sequential algorithm. An explanation can be as follows: Because not only a single model has to be found, but several models have to be created, checked and extended, much additional work has to be done compared to running a single tree search on the original input formula.

Finally, in contrast to the statement that an optimized search with already implemented shortcuts is hard to parallelize, the solver PRISS [45] can run a parallel two-watched-literal unit propagation [44]. Although it has been shown in [37] that unit propagation itself is P-complete, this publication shows that on real world instances the performance of the solver can still be increased. A reason for choosing to parallelize the unit propagation is that this part of the algorithm uses 80%

of the overall runtime of a modern SAT-solver [29], [10]. The parallel unit propagation separates the input formula and learned clauses into partitions. Each thread is assigned a private partition and it is the only thread with access to these clauses during unit propagation. Thus, each thread has to propagate the current decision and its implied literals on its private clause partition. Furthermore, found implied literals have to be shared with the other threads to keep completeness. This part of the algorithm introduces overhead. The results of this study show that in average an efficiency of 0.65 for two threads can be reached. Furthermore it is shown, that the approach does not scale beyond two threads. The performance of the parallel unit propagation also suffers from the shared memory bus as reported in [46].

5. Modern Architecture

Single processor systems have for a long time dominated the computing landscape. The steady performance improvements per processor generation left little incentive to parallelize applications.

Improvements to uniprocessor performance started to decline about ten years ago due to power dissipation problems, almost fully utilized instruction-level parallelism, and missing improvements in memory latency [28]. Parallelism instead of single core clock frequency has started to dominate performance of a given computer system. At present, every major chip manufacturer has switched to multi-core designs.

Another development that is indirectly caused by increased parallelism is the move away from front side bus (FSB) architectures, in which every processor in a SMP system accesses main memory via a shared bus. Because memory accesses from different processors compete for bus and memory controller resources, the front side bus can become a performance bottleneck [12].

Current systems avoid this bottleneck by integrating memory controllers inside the processor die [38], [65]. On system with multiple dies, memory accesses are distributed across multiple controllers. this creates non-uniform memory access latencies depending on which memory controller has to serve the request. This property is exaggerated by large caches that are meant to hide comparatively long memory access delays [8]. To fully utilize the available processing power, an application needs to be aware of which memory is cheap to access.

While the trend toward commercial many-core architectures is not as fast as predicted by Asanovic et al [1], chip manufacturers are experimenting with designs that incorporate an order of magnitude more cores on a chip than commercially available today, such as Intel's 80-core prototype [63] and its *Single-Chip Cloud Computer* (SCC) [36].

The SCC combines 48 standard, but comparatively weak, cores on a single die. Two cores form a node in a mesh network. Access to main memory is provided by four memory controllers sitting on the edge of the mesh. Communication between cores is facilitated by dedicated message passing buffers. While this processor is not meant for production, similar network-on-a-chip processors are expected to be commercially available in 2012 [62]. Serial applications cannot exploit such systems.

It is never easy to give a reliable outlook into the future, but regarding future processors one can formulate solid assumptions: (1) Multi- and many-core systems will be the norm. A single processor system will be the exception. (2) Single core performance improvements will further decline. (3) Memory access latency and bandwidth will be increasingly non-uniform.

While the future may be foggy, it is clear that current software needs to change to adapt to hardware developments.

6. Conclusion

Until now, modern CPUs contain only few cores. Recent parallel SAT-solvers for such shared-memory architectures are mainly based on the techniques developed for sequential SAT-solvers. In most cases, they either run few sequential solvers independently with different random seeds and heuristics or they share selected learned clauses. However, as soon as the individual solvers running on different cores and share memory, the efficiency of the individual solvers decreases. None of these parallel solvers seem to scale well if hundreds or even thousands of cores become available. None of these parallel solvers seem to be aware and make use of the specific features of modern computer architectures like non-uniform memory access latencies. At least we are not aware of reports that mention such techniques.

A lesson learned from the development of sequential SAT-solvers is that these solvers are only fast if they take the specific features of the underlying hardware into account. We expect that this holds for parallel SAT-solvers as well. Efficient parallel SAT-solvers running on many-core systems should be based on the following principles: (1) The solver implementation should be aware of the underlying hardware to tackle bottlenecks like the non-uniform memory access as well as possible. (2) The exchange of learned clauses

and other data between the processes running on different cores has to be carefully balanced on all levels from the calculus and heuristics level up to the data structure, implementation, and hardware level.

Designing a parallel SAT-solver respecting these principles seems to be difficult on the basis of current sequential state-of-the-art SAT-solvers. To create a parallel solver for many-core systems, the complete implementation of the sequential solvers needs to be reviewed and redesigned. For example, techniques like the two-watched-literal propagation might not be the best way to yield a good compromise between streaming memory accesses and fewer random memory accesses. Also, the splitting approach, which was neglected in recent years, might be reconsidered as a candidate to solve huge industrial formulas by dividing them into many partitions. It remains an open question, how to design the architecture of a future SAT-solvers to achieve a reasonable efficiency and a high scalability to exploit the parallel architectures, which will be installed in all future computing systems.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Sais. A generalized framework for conflict analysis. In *Proceedings of the 11th international conference on Theory and applications of satisfiability testing*, SAT'08, pages 21–27, Berlin, Heidelberg, 2008. Springer.

[3] G. Audemard and L. Simon. Predicting Learnt Clauses Quality in Modern SAT Solver. In *Twenty-first International Joint Conference on Artificial Intelligence(IJCAI'09)*, pages 399–404, jul 2009.

[4] R. J. Bayardo, Jr., and J. D. Pehoushek. Counting Models using Connected Components. In *In AAAI*, pages 157–162, 2000.

[5] A. Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical report, Technical Report 10/1, Institute for Formal Models and Verification, Johannes Kepler University, 2009.

[6] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

[7] W. Blochinger, C. Sinz, and W. Küchlin. A Universal Parallel SAT Checking Kernel. In H. R. Arabnia and Y. Mun, editors, *Proc. of the Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*, volume 4, pages 1720–1725, Las Vegas, NV, June 2003. CSREA Press.

[8] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[9] W. Chrabakh and R. Wolski. GridSAT: A Chaff-based Distributed SAT Solver for the Grid. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, New York, NY, USA, 2003. ACM.

[10] G. Chu, A. Harwood, and P. J. Stuckey. Cache conscious data structures for boolean satisfiability solvers. *JSAT*, 6:99–120, 2009.

[11] Chu, Geoffrey and Stuckey, Peter J. and Harwood, Aaron. PMiniSat - A parallelization of MiniSat 2.0. http://baldur.iti.uka.de/sat-race-2008/descriptions/solver_32.pdf, 2008.

[12] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *Micro, IEEE*, 27(2):10–21, March-April 2007.

[13] S. 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 Press.

[14] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[15] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

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

[17] N. Eén and N. Sörensson. An Extensible SAT-solver. In *Proc. 6th SAT, LNCS 2919*, 2004.

[18] Y. Feldman, N. Dershowitz, and Z. Hanna. Parallel Multithreaded Satisfiability Solver: Design and Implementation. In *Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification (PDMC 2004)*, volume 128 of *Electronic Notes in Theoretical Computer Science*, pages 75–90, 2005.

[19] S. L. Foremane and A. M. Segre. NAGSAT: A Randomized, Complete, Parallel Solver for 3-SAT. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 236–243. Springer, 2002.

[20] J. W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Philadelphia, PA, USA, 1995. UMI Order No. GAX95-32175.

[21] A. Geist, A. G. Ornl, W. S. Nas, and T. Skjellum. MPI-2: Extending the Message-Passing Interface, 1996.

[22] L. Gil, P. Flores, and L. M. Silveira. PMSat: a parallel version of MiniSAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:71–98, 2008.

[23] E. Goldberg. A decision-making procedure for resolution-based SAT-solvers. In *Proceedings of the 11th international conference on Theory and applications of satisfiability testing*, SAT'08, pages 119–132, Berlin, Heidelberg, 2008. Springer.

[24] L. Guo, Y. Hamadi, S. Jabbour, and L. Sais. Diversification and intensification in parallel SAT solving. In *Proceedings of the 16th international conference on Principles and practice of constraint programming*, CP'10, pages 252–265, Berlin, Heidelberg, 2010. Springer.

[25] Y. Hamadi, S. Jabbour, C. Piette, and L. Saïs. Deterministic Parallel DPLL: System Description. In *Pragmatics of SAT(POS'11)*, jun 2011.

[26] Y. Hamadi, S. Jabbour, and L. Sais. Control-based clause sharing in parallel SAT solving. In *Proceedings of the 21st international jont conference on Artifical intelligence*, pages 499–504, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[27] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a Parallel SAT Solver. *JSAT*, 6(4):245–262, 2009.

[28] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2007.

[29] S. Hölldobler, N. Manthey, and A. Saptawijaya. Improving resource-unaware SAT solvers. In C. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6397 of *Lecture Notes in Computer Science*, pages 357–371. Springer Berlin / Heidelberg, 2010.

[30] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations & Applications*. Elsevier / Morgan Kaufmann, 2004.

[31] A. E. Hyvärinen, T. Junttila, and I. Niemelä. Incorporating Learning in Grid-Based Randomized SAT Solving. In *Proceedings of the 13th international conference on Artificial Intelligence: Methodology, Systems, and Applications*, AIMSA '08, pages 247–261, Berlin, Heidelberg, 2008. Springer-Verlag.

[32] A. E. Hyvärinen, T. Junttila, and I. Niemelä. Strategies for Solving SAT in Grids by Randomized Search. In *Proceedings of the 9th AISC international conference, the 15th Calculemas symposium, and the 7th international MKM conference on Intelligent Computer Mathematics*, pages 125–140, Berlin, Heidelberg, 2008. Springer-Verlag.

[33] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä. Partitioning SAT instances for distributed solving. In *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 372–386, Berlin, Heidelberg, 2010. Springer.

[34] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä. A distribution method for solving SAT in grids. In *In SAT 2006, 4121 of LNCS*, pages 430–435. Springer, 2006.

[35] A. E. J. Hyvärinen, T. Junttila, and I. Niemelä. Grid-Based SAT Solving with Iterative Partitioning and Clause Learning. In *In CP 2011*, 2011.

[36] Intel. *SCC External Architecture Specification*, 2010.

[37] S. Kasif. On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. *AI*, 45(3):275–286, 1990.

[38] C. Keltcher, K. McGrath, A. Ahmed, and P. Conway. The AMD Opteron processor for multiprocessor servers. *Micro, IEEE*, 23(2):66–76, March-April 2003.

[39] S. Kottler and M. Kaufmann. SArTagnan - A parallel portfolio SAT solver with lockless physical clause sharing. In *Pragmatics of SAT*, 2011.

[40] D. Le Berre. Sat4j: a reasoning engine in Java based on the SATisfiability problem (SAT). http://www.sat4j.org.

[41] M. D. T. Lewis, T. Schubert, and B. Becker. Multithreaded SAT Solving. In *ASP-DAC*, pages 926–931. IEEE, 2007.

[42] M. D. T. Lewis, T. Schubert, and B. W. Becker. Early Conflict Detection Based BCP for SAT Solving. In *The International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[43] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Inf. Process. Lett.*, 47:173–180, 1993.

[44] N. Manthey. Parallel SAT Solving - Using More Cores. In *Pragmatics of SAT(POS'11)*, 2011.

[45] N. Manthey. Solver Submission of riss 1.0 to the SAT Competition 2011. Technical Report 2011-1, Knowledge Representation and Reasoning Group, Technische Universität Dresden, 01062 Dresden, Germany, 2011.

[46] R. Martins, V. Manquinho, and I. Lynce. Improving Search Space Splitting for Parallel SAT Solving. In *Proceedings of the 2010 22nd IEEE International Conference on Tools with Artificial Intelligence - Volume 01*, ICTAI '10, pages 336–343, Washington, DC, USA, 2010. IEEE Computer Society.

[47] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. *Design Automation Conference*, pages 530–535, 2001.

[48] Niklas Sörensson. MiniSAT 2.2 and MiniSAT++ 1.1. http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_25+26.pdf, 2010.

[49] K. Ohmura and K. Ueda. c-sat: A Parallel SAT Solver for Clusters. In O. Kullmann, editor, *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 524–537. Springer, 2009.

[50] OpenMP Architecture Review Board. OpenMP Application Program Interface. Specification, 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[51] K. Pipatsrisawat and A. Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *Proceedings of 10th International Conference on Theory and Applications of Satisfiability Testing(SAT)*, pages 294–299, 2007.

[52] A. Ramos, P. van der Tak, and M. Heule. Between Restarts and Backjumps. In K. Sakallah and L. Simon, editors, *SAT 2011*, Lecture Notes in Computer Science. Springer, 2011.

[53] O. Roussel. ppfolio solver. http://www.cril.univ-artois.fr/~roussel/ppfolio/, 2011.

[54] T. Schubert, M. Lewis, and B. Becker. PaMiraXT: Parallel SAT solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:203–222, 2009.

[55] J. P. M. Silva and K. A. Sakallah. GRASP: A New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[56] D. Singer. *Parallel Resolution of the Satisfiability Problem: A Survey*. Wiley Interscience, Oct. 2006.

[57] D. Singer and A. Monnet. JaCk-SAT: a new parallel scheme to solve the satisfiability problem (SAT) based on join-and-check. In *Proceedings of the 7th international conference on Parallel processing and applied mathematics*, PPAM'07, pages 249–258, Berlin, Heidelberg, 2008. Springer.

[58] D. Singer and A. Vagner. Parallel Resolution of the Satisfiability Problem (SAT) with OpenMP and MPI. In *Parallel Processing and Applied Mathematics (PPAM 2005)*, volume 3911/2006 of *Lecture Notes in Computer Science*, pages 380–388, 2006.

[59] C. Sinz, W. Blochinger, and W. Küchlin. PaSAT - parallel SAT-checking with lemma exchange: Implementation and applications. In H. Kautz and B. Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, volume 9 of *Electronic Notes in Discrete Mathematics*, Boston, MA, June 2001. Elsevier Science Publishers.

[60] M. Soos. CryptoMiniSat 2.5.0. In *SAT Race competitive event booklet*, July 2010.

[61] D. Sturgill and A. Segre. A novel asynchronous parallelism scheme for first-order logic. In A. Bundy, editor, *Automated Deduction — CADE-12*, volume 814 of *Lecture Notes in Computer Science*, pages 484–498. Springer Berlin / Heidelberg, 1994.

[62] Tilera. TILE-Gx 3000 Series Overview, 2011.

[63] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *Solid-State Circuits, IEEE Journal of*, 43(1):29 –41, jan. 2008.

[64] L. Zhang, C. F. Madigan, and M. H. Moskewicz. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *ICCAD*, pages 279–285, 2001.

[65] D. Ziakas, A. Baum, R. Maddox, and R. Safranek. Intel QuickPath Interconnect Architectural Features Supporting Scalable System Architectures. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 1–6, August 2010.