Performance of Parallel SAT Solvers

August 30, 2012

Abstract

This is the text of the abstract.

Contents

1 Introduction **1**

2 Preliminaries **3**
2.1 SAT and SAT solvers 3
2.2 Parallel SAT solvers 5
2.3 Symmetric multiprocessing 6
2.3.1 Multicore machines 6
2.3.2 Memory hierarchy 6

3 Plingeling performance **6**
3.1 Modified `plingeling` 6
3.2 `plingeling` scalability 8

4 AzuDICI **9**

5 Cache Performance in AzuDICI **12**
5.1 Same search experiments 12
5.2 Different search experiments 12

6 Related Work **12**

7 Conclusions and Future Work **13**

1 Introduction

In this paper, we study issues of scalability of parallel solvers of the satisfiability (SAT) problem on hierarchical-memory multicore (SMP) systems. We find this topic important for three reasons.

Since at least 2009, parallel SAT solvers (henceforth, pSATs) have been performing at the top of the SAT Competition (in 2011, all three wall-clock time

winners of the competition are parallel solvers). Also in 2011, pSATs and sequential SAT solvers are grouped into a single competition track[1], which signals the widespread interest in pSATs by the research and industrial communities. This appeal stems in part because of the inherent interesting properties of parallel algorithms, but also because of the need of the community to do better in other application domains and be able to handle even larger and more complex CNF formulas in smaller times taking advantage of modern hardware.

Meanwhile, instead of increasing clock performance, chip manufacturers are investing heavily on multicore architectures to improve performance and lower power consumption (AMD released the 8-core Opteron 3260 EE in late 2011, and Intel will do the same with the Xeon E5-2650, and its low power version, the Xeon E5-2650L early this year). As Herb Sutter once put it, "the free lunch is over" [1], and this has effectively meant that software in general will not be getting any faster as years go by simply relying on faster processors, but by relying on how software scales in multicore systems.

Finally, modern memory architectures are not flat Processor↔RAM architectures, but a hierarchy of faster-but-smaller to slow-but-large memories with latencies varying from $0.5ns$ access, $32Kb$ memories such as the L1 cache, to tens of nanoseconds, megabyte-large memories like the L3 cache, to gigabyte, $100ns$ access memory such as main memory. Hierarchical memory architectures have a strong impact on the performance of sequential software (e.g., row scanning arrays in row-major representation, memory transfers may be in the order of the input divided by the size of the cache line, while memory transfers for column scanning is in the order of the square of the input).

Thus, given the three arguments above: how *do* pSATs scale in hierarchical-memory multicore architectures? Our case study is the winner of the 2011 SAT Competition, `plingeling`, a portfolio-approach SAT solver [2]. The first experiment tested a *modified* `plingeling` on a 6-core multicore machine, varying the number of threads. This instance of `plingeling` was modified in order that, for each *worker thread*, the *same search* would be performed (i.e. same strategies, starting parameters and without lemma exchanging). This allows us to measure the impact of running p threads on the same physical CPU. In Figure 1 we can see now how the modified-`plingeling`'s performance tends to decay sharply (up to 30%) when several instances are executed on the same processor, even when they do not, in principle, share any resources other than the common process address space. We would thus expect that all instances would perform similarly, plus or minus a small time fraction. This is in fact what happens when `plingeling` is run in two different cores of two different chips, in different and even in the same machine.

In order to find the reason of the performance decay, we have developed a simple portfolio-approach-based SAT solver (which we have called AzuDICI) that allows us to both replicate the behavior of `plingeling`, and then also experiment with alternative scenarios (such as sharing information among threads) in order to take measures towards the improvement of its performance. It is im-

[1] `http://www.satcompetition.org/`

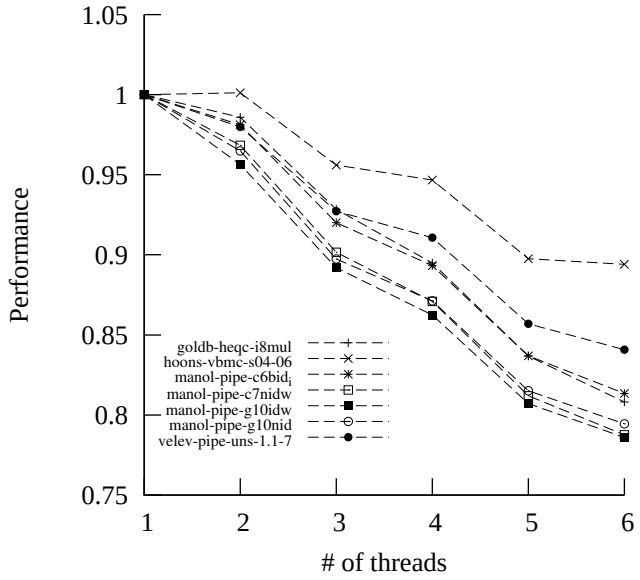


Figure 1: Performance decay of `plingeling` when run over many cores

portant to highlight that our SAT solver does not compare to high-performance current state-of-the-art solvers such as `plingeling` itself. AzuDICI serves as a useful tool to test and analyze the behavior of portfolio-approach-based SAT solvers.

2 Preliminaries

2.1 SAT and SAT solvers

Let $v \in V$ be a *propositional variable* and V a fixed finite set of propositional symbols. If $v \in V$, then v and $\neg v$ are *literals* of V. The *negation* of a literal l, written $\neg l$, denotes $\neg v$ if l is v, and v if l is $\neg v$. A *clause* is a disjunction of literals $l_1 \vee \ldots \vee l_n$. A (CNF) *formula* is a conjunction of one or more clauses $C_1 \wedge \ldots \wedge C_n$. A (partial truth) *assignment* M is a set of literals such that $\{v, \neg v\} \subseteq M$ for no v. A literal l is *true* in M if $l \in M$, is *false* in M if $\neg l \in M$, and is *undefined* in M otherwise. A clause C is true in M if at least one of its literals is true in M. It is false in M if all its literals are false in M, and it is undefined in M otherwise. A formula F is true in M, or *satisfied* by M, denoted $M \models F$, if all its clauses are true in M. In that case, M is a *model* of F. If F has no models then it is *unsatisfiable*.

The problem we are interested in is the *SAT problem*: given a formula F, decide whether there exists a model of F or not. Since there exists a polynomial transformation (see [3]) for any arbitrary formula to an equisatisfiable CNF one,

we will assume w.l.o.g. that F is in CNF.

A software program that solves this problem is called a *SAT solver* The Conflict-Driven-Clause-Learning (CDCL) algorithm, is nowadays at the basis of most state-of-ther-art SAT-solvers[4, 5, 6]. This algorithm has, at its roots, the very simple DPLL algorithm [7]. Thanks to the work done, mainly, in [8, 9, 10, 11, 12, 13] it has evolved to an algorithm that incorpores several conceptual and implementation improvements, making modern SAT-solvers able to handle formulas of millons of variables and clauses. Algorithm 1 sketches the CDCL algorithm.

Algorithm 1: CDCL algorithm

Input : formula $F = \{C_1, \ldots, C_m\}$
Output: SAT OR UNSAT

```
1  status := UNDEF;
2  model := {};
3  dl := 0;
4  while status == UNDEF do
5  |  (conflict, model) := UNIT_PROPAGATE(model, F);
6  |  while conflict ≠ NULL do
7  |  |  if dl == 0 then
8  |  |  |  return UNSAT;
9  |  |  lemma := CONFLICT_ANALYSIS(conflict, model, F);
10 |  |  F := F ∧ lemma;
11 |  |  dl := LARGEST_DL(lemma, model);
12 |  |  model := BACKJUMP_TO_DL(dl, model);
13 |  |  (conflict, model) := UNIT_PROPAGATE(model, F);
14 |  |
15 |  if status == UNDEF then
16 |  |  dec := DECIDE(model, F);
17 |  |  if dec = 0 then status := SAT;
18 |  |  model := model ∪ { dec };
19 |  |
20 return status
```

Basically, the CDCL algorithm is a backumping search algorithm that incrementally builds a partial assignment M over iterations of the *DECIDE* and *UNIT_PROPAGATE* procedures, returning SAT if M becomes a model of F or UNSAT if no such model exists.

The DECIDE procedure correspond to a branching step of the search and applies when the unit propagation procedure can not set true any further literal (see below). When no inference can be done about which literals should be true in M, a literal l^{dl} is chosen (guessed) and added to M in order to continue the search. Each time a new decision literal l^{dl} is added to M the *decision level* dl of the search is increased and we say that all the literals in M after l^{dl} and

before l^{dl+1} belong to decision level dl. For further reading about the DECIDE procedure, we refer to [9, 14, 15].

The UNIT_PROPAGATE (UP) procedure applies when certain assignment M falsifies all literals but one, of a clause C. If l is the undefined literal in C, in order for M to become a model of F, l must be added to M so as to satisfy C. This procedure ends when there is no literal left to add to M or when it finds that all literals of certain clause C are false. If it happens the first case, UP returns an updated model containing all such propagations and the search continues. In the second case it returns the falsified clause which we call a *conflicting clause*. For details on how modern SAT Solvers implement this procedure and literature regarding Cache performance (mainly related) with UP, we refer to [9, 16]

If unit propagation finds a conflicting clause, then two possibilities apply. If the decision level at which such conflict is found is zero (i.e. no decisions have been made) then the CDCL procedure returns UNSAT. If this is not the case, a *CONFLICT ANALISIS* procedure is called. This procedure analizes the cause of such conflict (i.e. determines which decisions have driven to this conflict) and returns a new clause (which we call a *lemma*) that is entailed by the original formula. Then, the algorithm backjumps to an earlier decision level dl' that corresponds to the highest $dl' < dl$ in the lemma, and propagates with it. CONFLICT ANALISIS works in such a way that, when backjumping and propagating with the lemma, the original conflict is avoided. The lemmas learned at CONFLICT ANALISIS time are usually added to the formula in order to avoid similar conflicts and can also be deleted (when the formula is too big and they are no longer needed). For details of this procedure, we refer to [10, 17] and for lemma deletion heuristics to [18, 19, 4].

For a complete review of this algorithm as well as proofs over its termination and soundness we refer to [20]

2.2 Parallel SAT solvers

Parallel SAT solvers are not as mature as sequential ones and it is still not clear which path to follow when designing and implementing such new solvers. We mainly classify parallel SAT solvers into two categories: Portfolio approach solvers and divide-and-conquer ones.

The main idea behind portfolio approach solvers is the fact that different strategies/parameters of CDCL sequential solvers will perform better for different families of SAT problems. In sequential CDCL SAT-solving, there exist several parameters/strategies related with the algorithm's heuristics in, for example, restarting, deciding or cleaning the clause database. Taking this into consideration, the portfolio approach is very straightforward: Run a group of sequential solvers in different threads, each with different parameters and/or different strategies. The time the portfolio-approach based solver will take to solve the problem will be the time of the fastest thread in the group of solvers running in parallel. Differences between this kind of solvers lie in whether the clause database should be shared [21], or otherwise if each thread should have its own

database. If so, it is possible to implement the threads in order to interchange lemmas both: aggressively [22] or selectively [5] or not have communications between threads at all [23].

Divide-and-conquer solvers do not try to run different solvers in parallel, they run one solving instance, but try to parallelize the search and divide it between the different threads. A common strategy to divide the search space is to use guiding paths [24]. A guiding path is a partial assignment M in F, which restricts the search space of the SAT problem. A solver that divides its search space with guiding paths will assign threads to solve F with the given M from the guiding path the thread was assigned to. Once a thread finishes searching a guiding path with no success, it can request another to keep searching (we refer to [25] for further explanations).

Both parallelization strategies (portfolio approach and divide-and-conquer approach) were and are currently being applied to shared memory parallel computers (e.g. [5]) as well as distributed memory ones (e.g. [25]).

2.3 Symmetric multiprocessing

2.3.1 Multicore machines

2.3.2 Memory hierarchy

For replication purposes, it is important to thoroughly characterize the machine we are using to run the tests. All experiments were carried out on a dual-processor 6-core Intel Xeon CPU (E5645) running at 2.40GHz, for a total of 12 physical cores at 28.8GHz. Hyperthreading was disabled. Figure **??** shows the memory hierarchy of the testing computer. To save space, we only show one of the chips (Socket P#0), the other is identical.

Each core has one processor unit (PU), with separate L1d (32KB) and L2 (256KB) caches. (The strange numbers of Cores and PUs have to do with how the Linux Kernel sees them logically.) They share a 12MB L3 cache. Main memory is 6GB. The computer runs Linux 3.0.0-15-server, in 64-bit mode. The details of the caches are as follows:

```
L1 Data:        32K 8-way with 64 byte lines
L2 Unified:     256K 8-way with 64 byte lines
L3 Unified:     12288K 16-way with 64 byte lines
```

3 Plingeling performance

`plingeling` is a portfolio-based SAT solver... [[Short explanation of plingeling]]

3.1 Modified `plingeling`

One of the advantages we assume of parallel computing is that the more cores we add, the better performance we will obtain. This should be also true for `plingeling`, since the only difference of adding more threads (assuming we have

one thread per core) is that we will have a greater variety of solver strategies trying to solve the same problem, and also some logical clause sharing among threads. These are all valid assumptions in theory, but empirical results on multicore shared memory computers also show us that increasing the number of threads also carries a considerable decrease in performance for portfolio solvers like `plingeling`.

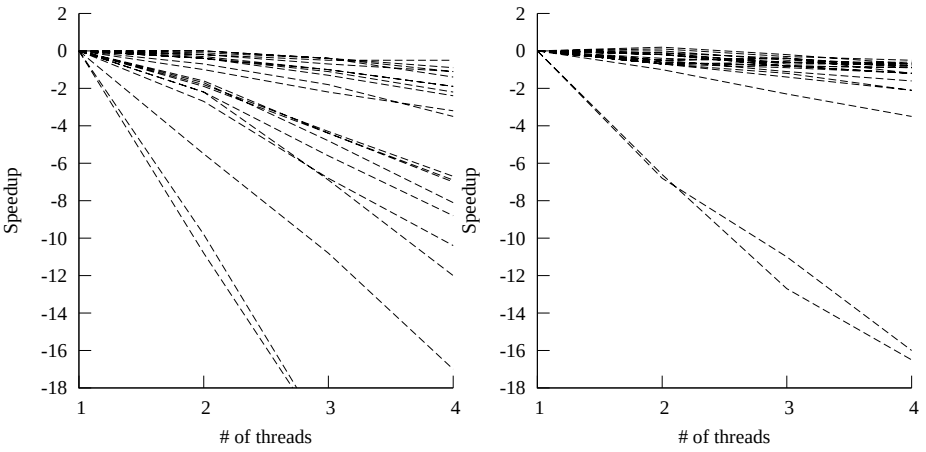
Multicore shared memory systems have their cores sharing the same last level cache (LLC) memory. The last level cache size in modern machines has a few megabytes and is usually not enough to hold all the data required by a SAT instance. Therefore, there will inevitably be some communication between the LLC and the main memory. The time cost of communications between the CPU and the LLC cache are much faster than having to get the data from main memory, so we would like to keep data transfers from main memory to a minimum.

Portfolio SAT solvers which only share clauses logically have to keep a complete database of clauses for each thread's use. So as we add more threads, the solver has greater needs of memory, but because all cores share the same LLC, all threads will have a lower chance of finding their data in the LLC as we add more threads. In this scenario, what we would expect is to observe, as we have in our experiments, a considerable decrease in performance when adding threads, simply because we will incur in more LLC cache misses when the amount of data to be manipulated by different threads increases. We don't usually appreciate this negative performance impact in these type of solvers, because different threads implement different SAT solving strategies, so the solving time will mostly depend on the fastest solving thread, shadowing the negative performance impact of copying the clause database in each thread.

In our experiments, we modified `plingeling`to replicate the exact same search in each of its threads. What we would expect, theoretically, is that adding more threads would have no impact in the solving time, because all cores would be making the exact same search with their own data. However, in practice, we found that the performance decrease of having six threads in six cores to be of around 15-40% of the total time one thread would take (Figure 1). The reasons for this behavior may be due to several factors in modern SMP architectures. However, sharing resources (such as the caches, communication and/or synchronization, main memory) could be seen as the main suspects. To find out what was happening in the machine, we ran another experiment, where a `plingeling` instance performing the same search with four threads was executed on different CPU chips, and, to compare, we ran the same experiment on four cores of the same CPU chip.

[[EXPLAIN MODIFICATIONS]]

As can be seen, executing the solver on *different* CPU chips does not impact performance, while executing it on the *same* CPU chip incurs in a significant performance decay. According to our experiment above, the only shared resource that could impact performance when run in one chip is the LLC or Last Level Cache. To effectively measure the involvement of the LLC, we used the



(a) on same chip (b) on different chips

Figure 2: Modified `plingeling` performance decay

`perf` tool[2] [[TALK ABOUT PERF]]

Figure 3: LLC statistics for modified `plingeling`

As the LLC-load-misses (number of times main memory data had to be retrieved) figure (Figure 2a) strongly suggests, this performance decrease is due to the six thread solver wasting much more time retrieving data from main memory than its single threaded version did. As we would also expect, our results show that the L1 cache behaves equally for each core, because L1 caches are not shared as opposed to the LLC cache.

3.2 `plingeling` scalability

In this section, we provide an overview of how `plingeling` behaves at a larger scale. So far, because of the experiment above, we know that the more threads we add, the more the cache impacts negatively in performance. However, we also know that adding threads also adds new workers with new (and possibly successful) strategies. This results in a trade-off between cache contention versus portfolio-approach benefits. To test this, we ran the original `plingeling` over 207 standard benchmarks taken from past SAT races and competitions (see link above), varying the number of threads from one to ten on a single chip with 10 physical cores.

[2] `xxxxxx`

Threads	# Problems solved	Total time
1	113	101399
2	121	95745
3	119	93854
4	122	90412
5	124	87953
6	124	89506
7	127	87416
8	124	88434
9	124	88931
10	125	89003

Table 1: Scalability

Figure 4 and Table 1 show that up until the fifth thread, scalability is good, but from then on, the number of solved problems and total time reaches a plateau. This means that `plingeling` cannot scale up on the number of cores sharing an LLC.

4 AzuDICI

AzuDICI[3] is a standard CDCL solver based on `plingeling`, `barcelogic` and `miraXT`, built with the aim of finding out whether or not sharing information among threads (particularly the clauses database) helps avoid the performance degradation that can be seen in Figure 1.

In particular, AzuDICI implements binary implication lists for the propagation with binary clauses, and the two-watched literals scheme for unit propagation [] with clauses of more than two literals. AzuDICI also implements the 1-UIP algorithm for conflict analysis **??**, the lemma simplification algorithm used in `PicoSAT`, Luby restarts [], a policy for lemma cleaning that keeps only binary and ternary lemmas, and more than four-literal lemmas that have participated in a conflict since the last cleanup. Finally, AzuDICI also incorporates the EVSIDS heuristic for branching literal decisions [].

In order to allow for comparison, we made several versions of AzuDICI. AzuDICI-shared-all is a version which shares all clauses between threads. It has a shared clause database which includes unitary clauses, binary clauses and n-ary clauses (clauses with more than two literals). Figure 5 shows how threads share the clause database. Each thread has access to the same physical clauses and they do not require synchronization with each other. Because binary implication lists are simple data structures that do not keep track of watched literals, it is easy to share binary clauses between threads without hindering the performance of propagation or any other task each sequential worker thread

[3]You can find the latest implementation of AzuDICI at `https://github.com/leoferres/AzuDICI`

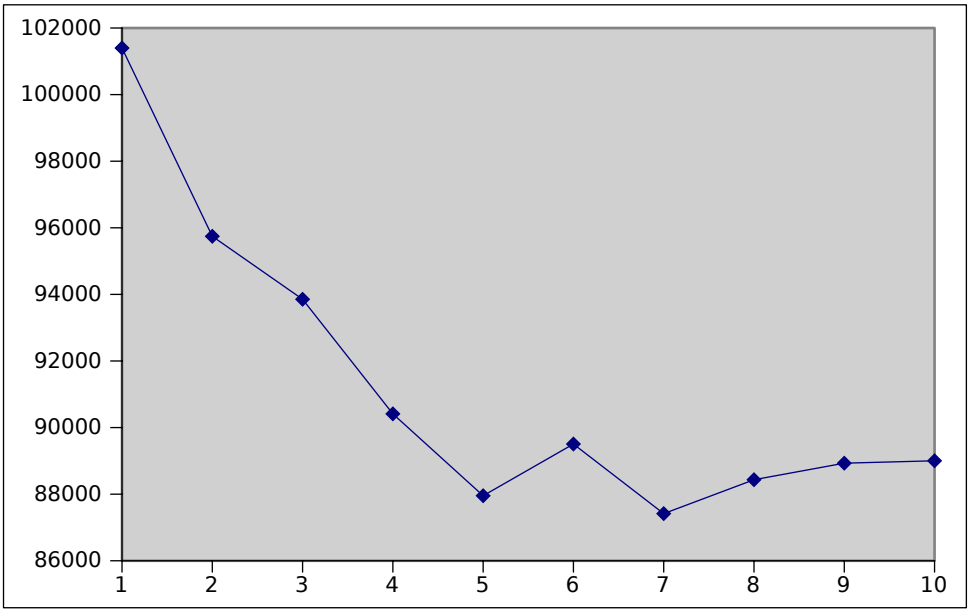


Figure 4: Total solving time per number of threads.

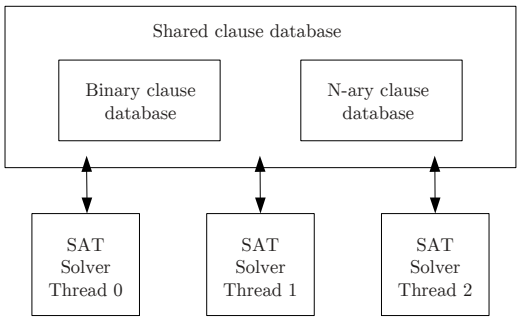


Figure 5: Shared clause database among threads.

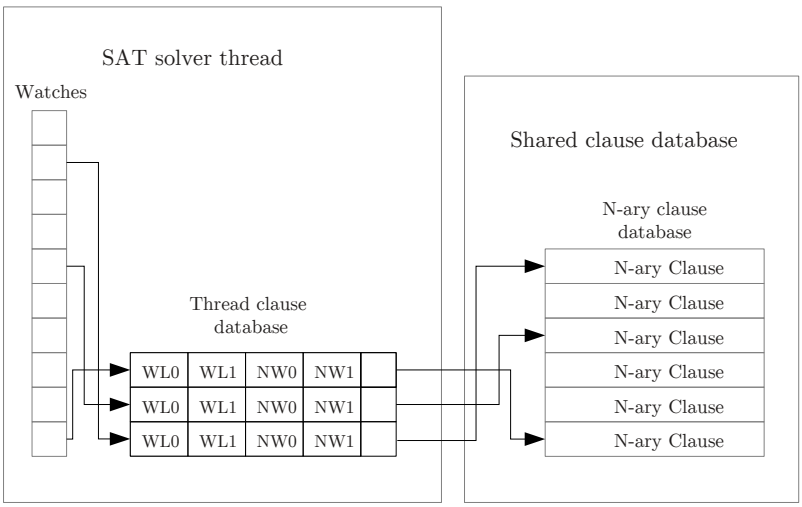


Figure 6: The thread clause database and n-ary clause database

performs. This is not the case for the n-ary clause database. The two-watched-literal scheme requires to make changes to the clause in order to identify the literals being watched at one given instant (for example, place both literals first in the clause). Since many threads will be accessing the same clauses, these changes to the clause are not feasible to share n-ary clauses. Instead, we have used a similar approach as used in `miraXT`, where each thread keeps track of the literals being watched in each clause. Figure 6 is a schematization of how each thread worker relates with the n-ary clause database. Each SAT solver thread has a vector of pointers to thread clauses called *watches*, and each literal present in the SAT problem has a position associated in this watches vector. A thread clause has two watched literals (WL0 and WL1), two pointers to another thread clause (NW0 and NW1) and a pointer to an actual n-ary clause in the n-ary clause database. W0 and W1 keep track of the literals being watched by the thread for a given n-ary clause. NW0 and NW1 point to the next thread clauses that are also watching WL0 and WL1 respectively.

As sharing n-ary clauses requires to make some changes to the implementation of the propagation scheme of the two-watched-literals, we also made the AzuDICI-shared-binaries version, which only shares the binary clause database. With this version we aim to find out if the drawbacks of sharing n-ary clauses, because we have to keep track of each watched literal in each thread, surpass the benefits. The AzuDICI-shared-none version doesn't share any clause and it's just a group of independent solvers running in parallel, each with its own clause database.

It is important to note that in order to make this comparison of different solver versions objective, we have forced them all to make the exact same search, so that the differences in performance we observe will not be due to the search

path each version takes. For the same reason, all threads in each version of the solver will also make the same search. We would not want an instance of a solver performing better just because an additional thread found a better search path.

Regarding thread safety of the solver versions which share clauses, we added some locks to ensure the correctness of the information being shared. The locks are only used to avoid two different threads inserting clauses to a same clause database at the same time, because the chance that this event occurs is so low, in practice there is no significant contention between threads.

5 Cache Performance in AzuDICI

To find out whether sharing the clause database was beneficial to the same-chip portfolio-approach solver, we ran AzuDICI in three different versions (sharing all the clauses, sharing none of the clauses and sharing only binaru clauses) on eight representative problems (XXXNAMESXXX) using one to six threads. Each run was repeated five times to clean up potential system noise. XXXKeiraXXX

5.1 Same search experiments

For this experiment, AzuDICI was modified to carry out the *same* search in each thread (i.e., there is no lemma sharing among threads). For each AzuDICI version, we measured the time needed to solve each benchmark, and the percentage of LLC misses. The results for this are shown in Table **??** below.

[[[[TABLE WITH RESULTS]]]]

[[[[GRAPH AGGREGATING EXEC TIME PER THREAD]]]]

[[[[GRAPH PERCENNTAGE LLC MISSES PER THREAD]]]]

[[[[ANALYSIS]]]]

As is evident, our implementation shows that physically sharing the clause database is beneficial to avoid cache contention.

5.2 Different search experiments

Our previous experiment validates our implementation of the solver. Nonetheless, the results are not generalizable to a full-featured SAT solver. It may be the case that while threads are carrying out the same search, it is more likely that they will access the same data. Whereas different search threads are clearly not necessarily accessing the same data at the same time.

To find out how a real portfolio-approach SAT solver implementing different levels of physical clause sharing behave, we ran AzuDICI in the same three different versions as before (sharing-all, share-none and share-bin) on eight different (harder) problems (XXXNAMESXXX) using one to six threads and with a 5-minute timout. Each run was repeated five times to clean up potential system noise. XXXKeiraXXX

[[[[TABLE WITH RESULTS]]]]

[[[[GRAPH AGGREGATING EXEC TIME PER THREAD]]]]
[[[[GRAPH PERCENNTAGE LLC MISSES PER THREAD]]]]
[[[[ANALYSIS]]]]

From these data, we may conclude that physically sharing the whole clause database does not improve the cache performance of the portfolio-approach-based SAT solvers

6 Related Work

Talk about miraXT and the other paper found by Leo.

7 Conclusions and Future Work

Let's hope to say that implementation efforts for programming a portfolio approach-based sat-solver with shared clauseDB between thread pays off when leading with cache problems.

Future work is to continue with the development of azudici, incorporating in it further enhancements like formula preprocessing, variable elimination, etc, etc, etc.

Acknowledgments

Acknowledgements to Barcelogic group for sharing the code of a past Barcelogic SAT-solver and their support.

Intel MTL

Armin Biere for opensourcing his `plingeling` code.

References

[1] Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobbs Journal **30** (2005) 202–210

[2] Biere, A.: Lingeling and friends at the SAT Competition 2011. Technical Report 11-1, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria (2011)

[3] Tseitin, G.S.: On the Complexity of Derivation in the Propositional Calculus. Zapiski nauchnykh seminarov LOMI **8** (1968) 234–259

[4] Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In Boutilier, C., ed.: IJCAI. (2009) 399–404

[5] Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Technical report, Institute for Formal Models and Verification, Johannes Kepler University (2010)

[6] Soos, M., Nohl, K., Castelluccia, C.: Extending sat solvers to cryptographic problems. In Kullmann, O., ed.: SAT. Volume 5584 of Lecture Notes in Computer Science., Springer (2009) 244–257

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

[8] Jr., R.J.B., Schrag, R.: Using csp look-back techniques to solve real-world sat instances. In: AAAI/IAAI. (1997) 203–208

[9] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: 38th Design Automation Conference, DAC'01, ACM Press (2001) 530–535

[10] Marques-Silva, J., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Trans. Comput. **48** (1999) 506–521

[11] Zhang, H., Stickel, M.E.: An efficient algorithm for unit propagation. In: Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96), Fort Lauderdale (Florida USA) (1996)

[12] Eén, N., Sörensson, N.: An Extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: 6th International Conference on Theory and Applications of Satisfiability Testing, SAT'03. Volume 2919 of Lecture Notes in Computer Science., Springer (2004) 502–518

[13] Biere, A.: Picosat essentials. JSAT **4** (2008) 75–97

[14] Eén, N., Sörensson, N.: An Extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: 6th International Conference on Theory and Applications of Satisfiability Testing, SAT'03. Volume 2919 of Lecture Notes in Computer Science., Springer (2004) 502–518

[15] Pipatsrisawat, K., Darwiche, A.: Rsat 2.0: Sat solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA (2007)

[16] Zhang, L., Malik, S.: Cache Performance of SAT Solvers: A Case Study for Efficient Implementation of Algorithms. In Giunchiglia, E., Tacchella, A., eds.: 6th International Conference on Theory and Applications of Satisfiability Testing, SAT'03. Volume 2919 of Lecture Notes in Computer Science., Springer (2004) 287–298

[17] Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: Int. Conf. on Computer-Aided Design (ICCAD'01). (2001) 279–285

[18] Bayardo, R.J.J., Schrag, R.C.: Using CSP look-back techniques to solve real-world SAT instances. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97), Providence, Rhode Island (1997) 203–208

[19] Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT-solver. In: Design, Automation, and Test in Europe (DATE '02). (2002) 142–149

[20] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). Journal of the ACM, JACM **53** (2006) 937–977

[21] Kottler, S., Kaufmann, M.: Sartagnan - a parallel portfolio sat solver with lockless physical clause sharing. In: Pragmatics of SAT. (2011)

[22] Hamadi, Y., Jabbour, S., Sais, L.: Manysat: a parallel sat solver. JSAT **6** (2009) 245–262

[23] Roussel, O.: Description of ppfolio. Technical report, CRIL, Centre de Recherche en Informatique de Lens (2011)

[24] Zhang, H., Bonacina, M.P., Hsiang, J.: Psato: a distributed propositional prover and its application to quasigroup problems. J. Symb. Comput. **21** (1996) 543–560

[25] Schubert, T., Lewis, M.D.T., Becker, B.: Pamiraxt: Parallel sat solving with threads and message passing. JSAT **6** (2009) 203–222