About Caching in D4 2.0

Jean-Marie Lagniez[1] and Pierre Marquis[1,2]

[1]CRIL, Université d'Artois & CNRS, France
[2]Institut Universitaire de France, France
{lagniez, marquis}@cril.fr

Abstract

We present a new caching scheme and new cache management strategy that have been implemented in the last release of our compilation-based model counter, D4. The caching scheme consists in storing for each entry (a CNF formula forming a connected component given a current variable assignment, together with its model count) the corresponding set of variables and the corresponding set of clauses, except those clauses of the CNF formula that are satisfied or not shortened when conditioned by the assignment. The cache management strategy includes a cache cleaning strategy, based not only on the ages of the entries but also on the proportion of entries of the same size that led to positive hits. It also includes a cache insertion strategy, that aims to memory saving by avoiding to store in the cache every CNF formula that is encountered during search.

1 Introduction

Caching turns out to be a key ingredient of propositional model counters and compilers targeting the Decision-DNNF language for knowledge compilation.

A caching scheme simply is a mapping making precise how each CNF formula φ considered during search is represented. The quality of a caching scheme can be measured as its ability to determine efficiently whether a given CNF formula φ has an equivalent representation already in the cache (the time efficiency), without missing it if this is the case, while ensuring that the cache remains of a reasonable size (the space efficiency). The two aspects (time and space) are clearly linked and have a direct impact on the (time and space) complexity of the model counter/compiler into which the caching scheme is exploited.

Since the size of the cache can become huge (especially in knowledge compilation scenarios, i.e., when Decision-DNNF representations are stored), it is of the utmost importance from the practical side to limit the memory consumption through the use of a cache cleaning strategy so as to avoid out-of-memory errors. Cache management strategies aim to find out entries that should be removed

from the cache because they are of large size and/or are seldom activated (trade-offs are looked for). They also aim to avoid storing in the cache entries that will not be hit during the rest of the computation. Because caching leads to enhance the performance of model counters and compilers, a cache manager is implemented in many state-of-the-art #SAT solvers, including Cachet[1] [SBB+04] and sharpSAT[2] [Thu06], and top-down compilers, including C2D[3] [Dar01, Dar04], Dsharp[4] [MMBH12], and D4[5] [LM17].

In this abstract, we present the caching scheme that has been implemented in the last release of D4. In this scheme, the clauses of the current CNF formula φ are represented explicitly as in the standard scheme used by Cachet [SBB+04], thus avoiding to recognize some entries that are missed by the hybrid scheme considered in sharpSAT [Thu06]. Furthermore, the size of the cache is minimized by storing for each entry the corresponding set of variables and a restricted set of residual clauses. In this restricted set, not only the satisfied clauses of Σ are omitted, but also the binary clauses of Σ considered at start (as in sharpSAT) and, *more generally, every input clause which has not been shortened by the current variable assignment*. We also present a new cache management strategy that refines the one used in sharpSAT, and takes account for the ages of the entries, with dynamic resets to reflect their activities, and but also for the proportion of entries of the same size that led to positive hits (so as to try and keep in the cache the most promising entries). To save memory, only entries bearing on a number of variables that do not exceed a preset bound are stored, and this bound is dynamically updated. Finally, the results of an empirical evaluation are reported, showing the caching scheme and the management strategy implemented in the last release of D4 as valuable.

2 An Improved Caching Scheme

Formally, a *caching scheme* c is a mapping associating with any $\varphi \in S(\Sigma) = \{\delta' \subseteq \delta \mid \delta \in \Sigma\}$ for some CNF formula Σ a representation $r_c(\varphi)$ of φ (this representation can be any data structure). A *cache* for Σ given a caching scheme c is a mapping associating with representations $r_c(\varphi)$ of CNF formulae $\varphi \in S(\Sigma)$ their number of models $\|\varphi\|$ (or Decision-DNNF representations of those φ). Any $\varphi_1 \in S(\Sigma)$ encountered during the search is considered not to be stored in the cache whenever one can find in the cache an entry $r_c(\varphi_2)$ such that $r_c(\varphi_1) = r_c(\varphi_2)$ holds. A *correct caching scheme* c is a caching scheme such that for any CNF formula Σ, $\varphi_1, \varphi_2 \in S(\Sigma)$, if $r_c(\varphi_1) = r_c(\varphi_2)$ then $\varphi_1 \equiv \varphi_2$. Stated otherwise, when the representations of φ_1 and φ_2 coincide, then the two CNF formulae φ_1 and φ_2 must be equivalent. Of course, the converse implication is not expected (otherwise, either

[1] www.cs.rochester.edu/~kautz/Cachet/index.htm
[2] sites.google.com/site/marcthurley/sharpsat
[3] reasoning.cs.ucla.edu/c2d/
[4] www.haz.ca/research/dsharp/
[5] www.cril.fr/kc/

the cache size or the look-up operation would be prohibitive).

Because of the syntactic nature of the approximate equivalence test, the performance of a caching scheme depends only on two key factors: the cumulated sizes of the entries and the quality of the approximation of equivalence that is achieved. In the standard caching scheme s[6]), every CNF formula φ encountered during search is represented as a string gathering the identifiers (integers) of the literals of the clauses in it, separating the clauses by zeroes. The identifier of a positive literal x_i is its index i in the enumeration $x_1 < x_2 < \ldots < x_{n-1} < x_n$, and the identifier of a negative literal $\neg x_i$ is $-i$. In the hybrid caching scheme h considered in sharpSAT, the clauses of the input CNF formula Σ are indexed in an arbitrary way (the index of the first clause of Σ is 1, the index of the second clause of Σ is 2, etc.). Every CNF formula φ encountered during search is represented as a pair consisting of the set of indexes of the variables occurring in φ (again, the identifier of x_i is i) in ascending order, and the set of indexes of the clauses of Σ that are "still alive" (non satisfied) and containing at least two unassigned literals), in ascending order. A variant of h, called o, consists in omitting in the set of indexes of the clauses of φ every index of a binary clause. Indeed, as shown in [Thu06], this set is redundant since it can be reconstructed knowing the indexes of the literals occurring in it. This leads to smaller sizes of the codes, hence to a more efficient cache management. For the sake of code conciseness, indexes are typically compressed, i.e., represented using as few bits as possible (and not using a fixed format for integers). The o scheme with such a code compression is referred to as the p scheme in [Thu06] ("p" stands for "packing").

Clearly enough, representing clauses by their indexes in Σ (as with the h scheme, the o scheme or the p scheme) is more efficient in term of encoding sizes than representing them explicitly (as with the s scheme). However, explicitly representing the clauses leads to a finer equivalence relation between the clause sets.

Let us present the new caching scheme i, which has been implemented in the last release of D4. In order to describe it in a smooth way, we start with a basic caching scheme b, which is such that for any CNF formula Σ and any $\varphi \in S(\Sigma)$, $r_b(\varphi) = (V, C)$ where V is the set of indexes of variables of $Var(\varphi)$ ordered in ascending way and C is the set of residual clauses of φ given Σ (the clauses which are not satisfied by the assignment corresponding to φ). With b, C is sorted by increasing clause size first, and within each cluster of clause representations having the same size, the indexes of the literals of each clause are listed in increasing order w.r.t. the lexicographic ordering induced by the literal ordering $x_1 < \neg x_1 < x_2 < \neg x_2 < \ldots < x_{n-1} < \neg x_{n-1} < x_n < \neg x_n$ when the variable ordering is such that $x_1 < x_2 < \ldots < x_{n-1} < x_n$. Multi-occurrent clauses are removed (only one occurrence per clause is kept). When the basic caching scheme b is used, the residual clauses are represented in an explicit way, as in the s scheme and not using indexes (unlike the schemes h, o, p presented above), but they are ordered in such a way that two sets of residual clauses that differ only as to the ordering of the clauses

[6] According to [Thu06], it is the one used in Cachet ([SBB+04] do not make it precise).

and/or as to the ordering of literals within clauses will have the same representation C (this ordering is maintained dynamically). The indexes used for representing clauses in C also use as few bits as possible, as in the p scheme. A variant of b, noted 2, is obtained by avoiding to store the binary clauses as with o. What makes i different of b also is the notion of residual clause under consideration. With the i caching scheme, not only the binary clauses of Σ are not stored, but every clause of φ that is not shortened by the corresponding variable assignment is not represented as well. Finally, with the variant i' of i, one considers the same set of residual clauses as in i, but represents those clauses using indexes as in the h scheme. Interestingly, refraining from storing the clauses of Σ that have not been shortened does not question the correctness of the approach, whatever clauses are represented explicitly (by the literals in them) or implicitly (by their indexes): the caching schemes i and i' are correct.

3 An Improved Cache Management Strategy

A *cache management strategy* must make precise the entries that have to be stored in the cache (i.e., a cache insertion policy), and those that must be removed from the cache and an agenda for the operations (especially, the cleaning operations can be achieved periodically, or be triggered by some events like the number of entries or the fact that the quantity of memory that is available falls down a preset threshold).

In `Cachet` [SBB$^+$04] and in `sharpSAT` [Thu06], every `CNF` formula encountered during the search is a candidate for being inserted into the cache (thus, the cache insertion policy is in some sense trivial). Observing that the utility of the cached components typically declines dramatically with age, in `Cachet` each cached component is given a sequence number and those components that are too old are removed from the cache (the age limit is considered as an input parameter). For efficiency reasons, age checking is not done frequently; when a new component is cached, age checking is achieved on the chain that contains the newly cached component. The cache is cleared whenever the number of entries exceeds $2^{21} \times 10$ and only 25% of the entries are kept whenever a cleaning operation takes place. In `sharpSAT`, the entries to be cleaned up do not depend only on their ages (i.e., the first time they are encountered), but also on their activity levels and on their sizes. A score $score(e)$ is associated with each entry e. At start, $score(e)$ is given by the age of the entry but it is reset during the search each time a positive hit corresponding to the entry is obtained. All scores are divided periodically so as to penalize the oldest entries. The cache is cleared only if its size exceeds a fixed fraction of the maximal allowed size (another input parameter). Entries are considered by increasing scores and removed from the cache until a sufficient amount of space (e.g., half of the maximal allowed size) has been recovered.

In the cache management strategy that has been implemented in the last release of `D4`, it is not always the case that every `CNF` formula φ encountered during the search is inserted into the cache (when it is not already in). To the goal of memory

saving, only those φ such that $|\mathit{Var}(\varphi)| \leq t$, where t is a given threshold, are put into the cache. t is first set to an initial value t_{init}, and the value of t is dynamically updated on a regular basis. Basically, for every $n \in [t]$, the number of positive hits for any φ such that $|\mathit{Var}(\varphi)| = n$ is stored in a table T. Periodically, the largest $m \in [t]$ such that $T[m] \neq 0$ is computed and t is replaced by $p \times m$, where p is a fixed value. Depending on m, this update operation leads to increase or to decrease the value of t. An aging mechanism is implemented (each value in T is divided by 2 once the value of t has been updated).

With each entry e in the cache we also associate an integer value $\mathit{score}(e)$ as in `sharpSAT`, together with a Boolean value $\mathit{flag}(e)$ that is set to true once the entry hits positively. Each time a cache hit occurs on an entry e, $\mathit{score}(e)$ is reset and $\mathit{flag}(e)$ is set to true. Moreover, we store information about how many entries $\mathit{nbTotal}[s]$ of any given size s the cache data structure contains, as well as how many entries $\mathit{nbHit}[s]$ of size s have their *flag* variable set to true. The size of an entry is measured as the number of variables in it.

Each time tc new entries have been added to the cache (where tc is a parameter of the strategy), some cleaning is performed. All the entries of the cache are visited in order to decide the ones that must be removed. Contrary to `sharpSAT`, our cleaning strategy does not remove half of the entries in a systematic way, and the cleaning operations are not triggered by how filled the cache is. For each entry e, the ratio

$$r(e) = \frac{\mathit{nbHit}[|\mathit{Var}(e)|]}{\mathit{nbTotal}[|\mathit{Var}(e)|]}$$

is computed. For each entry size, during the search, this ratio evaluates the proportion of entries of this size having led to a positive hit. In our cache cleaning strategy, entries with a high ratio $r(e)$ are promoted. Indeed, we have observed empirically that along the search, entries e with a high ratio $r(e)$ are often the most promising ones, i.e., when entries of the same size led to many positive hits, it is likely that e will also lead to some positive hits (this observation can be explained by the presence of variables playing symmetric roles in the instances). Thus, as a rule of thumb, it makes sense to give a bonus to such entries. Accordingly, in our cache cleaning strategy, every entry e that has a ratio $r(e)$ less that some fixed threshold rm (another parameter), with a $\mathit{score}(e)$ equals to zero and a $\mathit{flag}(e)$ set to false is flushed. The other entries e are kept and their $\mathit{score}(e)$ are divided by two. Whenever $\mathit{score}(e)$ falls to zero, $\mathit{flag}(e)$ is set to false.

4 Some Empirical Results

In order to evaluate the performance of the caching schemes and of the cache management strategies, we performed a number of experiments. We have considered 400 `CNF` instances, which are precisely the benchmarks used for evaluating the performances of the (possibly weighted) model counters during the First International Competition on Model Counting that was held in 2020 (see

https://mccompetition.org/). Those instances are those used for Track 1 of the competition (model counting - 200 instances) plus those used for Track 2 (weighted model counting - 200 instances).

Name	E/I	All	Not b	Not s
n	-	-	-	-
b	E	✓		
2	E		✓	
i	E			✓
h'	I	✓		
p	I		✓	
i'	I			✓

Table 1: Seven caching schemes considered in the paper.

In our experiments, `D4` has been used in the model counter mode: when invoked with option `-mc`, `D4` explores the same search space as the one considered when it is used as a Decision-`DNNF` compiler, but stores in its cache model counts instead of Decision-`DNNF` representations [LM17]. The branching heuristic that has been exploited is the one based on dynamic decomposition (DECOMP) used by default in `D4`.

We have considered the caching schemes n, b, 2, i, as well as h', the variant of h where indexes are compressed, p, and i'. The features characterizing those schemes are summarized in Table 1. "E(xplicit)/I(mplicit)" makes precise whether residual clauses are represented explicitly (by their literals) in the scheme, or implicitly (by their indexes). When "All (residual clauses)" is activated, every residual clause is stored. When "Not b(inary)" is activated, the binary clauses of Σ are not stored. Finally, when "Not s(hortened)" is activated, the binary clauses of Σ and the clauses of Σ that have not been shortened are not stored. The hash function used in the implementation of the cache was `MurMurHash2` (see https://en.wikipedia.org/wiki/MurmurHash).

We have also considered several cache management strategies: no cache cleaning, the strategy used by `Cachet`, the one used by `sharpSAT`, and our own cleaning strategy, together with two cache insertion modes (*all*: every `CNF` formula φ encountered during the search is inserted into the cache, and *some*: only those `CNF` formulae φ encountered during the search and such that $|Var(\varphi)| \leq t$ are considered as candidates for being put into the cache). The parameters used in the strategy implemented in `Cachet` and the ones used by `sharpSAT` were set to their default values. t_{init} was set to 500 and the value of t was updated every 10^5 recursive calls. p was set to $\frac{3}{2}$, tc to 10^5 and rm to 0.3.

All the experiments have been conducted on a cluster equipped with quadcore bi-processors Intel XEON E5-5637 v4 (3.5 GHz) and 128 GiB of memory. The kernel used was CentOS 7, Linux version 3.10.0-514.16.1.el7.x86_64. The compiler used was gcc version 5.3.1. A time-out (TO) of 1h and a memory-out (MO) of 7.6 GiB has been considered for each instance.

For each combination caching scheme/cache management strategy and each benchmark, the performance of `D4` has been assessed by measuring the model counting time, and the memory consumed by `D4`. To get a baseline, we have also run `D4` on the whole dataset while disabling the cache. In this case, only 160 instances have been solved in due time, showing the significance of the caching ingredient for solving instances from the dataset under consideration.

Table 2 makes precise for each combination caching scheme / cache management strategy, some empirical data of the form $x(y)$ where x is the number of instances solved (out of 400) and y the number of MOs that occurred. "E/I", "All", 'Not b", "Not s" characterize the caching scheme used and have precisely the same meaning as in Table 1. "Cleaning" indicates the cleaning strategy used (the one of `Cachet`, the one of `sharpSAT`, or our own one - "ours"). Finally, "Insert" gives the cache insertion mode (*all* or *some*).

E/I	Cleaning	Insert	All	Not b	Not s
E	none	all	244(155)	251(148)	276(119)
E	none	some	258(132)	264(125)	281(107)
E	`Cachet`	all	243(156)	261(133)	288(101)
E	`Cachet`	some	258(132)	274(110)	293(89)
E	`sharpSAT`	all	262(134)	266(127)	285(87)
E	`sharpSAT`	some	277(107)	275(108)	291(76)
E	ours	all	280(77)	283(69)	299(23)
E	ours	some	**294**(54)	**296**(47)	**305**(12)
I	none	all	254(145)	261(138)	271(122)
I	none	some	263(124)	269(118)	273(110)
I	`Cachet`	all	256(143)	271(109)	282(90)
I	`Cachet`	some	265(122)	279(89)	285(78)
I	`sharpSAT`	all	273(106)	274(102)	280(85)
I	`sharpSAT`	some	278(86)	282(85)	282(72)
I	ours	all	279(48)	283(35)	290(23)
I	ours	some	**288**(26)	**292**(16)	**292**(10)

Table 2: Number of instances solved by `D4` (# MO reached) for several combinations of caching scheme / cache management strategy.

In Table 2, the largest numbers of instances solved for each caching scheme when the cache management strategy (made precise by the values of "Cleaning" and "Insert") varies are boldfaced, and the largest numbers of instances solved for each cache management strategy when the caching scheme (made precise by the flags "E/I", "All", 'Not b", "Not s") are reported in lilac cells. One can observe that, on the one hand, the caching scheme i (characterized by "E" and "Not s") performs better than any other scheme whatever the cache management strategy used; on the other hand, that the new cache management strategy proposed in the paper and characterized by "ours" and "some" performs better than any of the three other cache management strategies that have been considered. The underlying principle (trying to learn what are the most promising entries) looks quite useful here. Espe-

cially, it can be observed that the cache cleaning strategy "ours" performs not bad in term of the numbers of MOs that are obtained (in particular when it is coupled to i or i'). This is somewhat surprising given that the amount of available memory is not exploited in this strategy, while cleaning operations are triggered by the number of entries / amount of available memory in `Cachet` / `sharpSAT`. Finally, the cache insertion strategy "some" where all the `CNF` formulae encountered during the search are not systematically added to the cache, also appears as the best insertion strategy since it leads to more instances solved than its challenger "all" whatever the caching scheme and the cache cleaning strategy that are used.

Empirically, with 305 instances solved, the best combination turns out to be i, together with the new cleaning strategy ("ours") and the new insertion strategy ("some"). It leads to solve a significant amount (more than 10%) of additional instances compared to the current version of `D4` (which is based on i as well, but does not take advantage of any cache management strategy, i.e., it corresponds to the combination "none" for Cleaning and "all" for Insert).

Acknowledgments

This work has been partly supported by the PING/ACK project (ANR-18-CE40-0011) from the French National Agency for Research (ANR).

References

[Dar01] A. Darwiche. Decomposable negation normal form. *Journal of the ACM*, 48(4):608–647, 2001.

[Dar04] A. Darwiche. New advances in compiling CNF into decomposable negation normal form. In *Proc. of ECAI'04*, pages 328–332, 2004.

[LM17] J.-M. Lagniez and P. Marquis. An Improved Decision-DNNF Compiler. In *Proc. of IJCAI'17*, pages 667–673, 2017.

[MMBH12] Ch.J. Muise, Sh.A. McIlraith, J.Ch. Beck, and E.I. Hsu. Dsharp: Fast d-DNNF compilation with sharpSAT. In *Proc. of AI'12*, pages 356–361, 2012.

[SBB+04] T. Sang, F. Bacchus, P. Beame, H.A. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *Proc. of SAT'04*, 2004.

[Thu06] M. Thurley. sharpSAT - counting models with advanced component caching and implicit BCP. In *Proc. of SAT'06*, pages 424–429, 2006.