Caching in Model Counters: A Journey through Space and Time

Jeroen G. Rook[1]*, Anna L. D. Latour[1], Holger H. Hoos[1,2], and Siegfried Nijssen[3]

[1] Leiden University, Leiden, The Netherlands
[2] University of British Columbia (UBC), Vancouver, Canada
[3] UCLouvain, Louvain-la-Neuve, Belgium

1 Motivation

DPLL-style exact model counters have become increasingly fast in solving #SAT problems, due to new branching heuristics [7, 3], clause learning [6], and component caching [1, 2]. In this work, we aim to contribute to this last line of work on model counters.

A DPLL-based model counter induces a depth-first search tree by assigning truth values to variables in a propositional input formula (typically in CNF). At any node in the search tree, the truth assignments to variables may yield a residual formula that can be decomposed into disjoint subformulae (*components*) that do not have any variables in common. The model counts of these components are therefore independent of their sibling components. Caching the model counts of components helps a DPLL-style model counter to avoid recomputing the model count of subformulae that are encountered at multiple points in the search tree.

We observe that memory is typically a limited resource, that the number of components of a formula can be exponential in the number of variables in the input CNF, and thus that the cache will eventually be exhausted for sufficiently large problems. Consequently, model counters usually clean their cache regularly, to make room for storing new component counts. Intuitively, we would assume that the more memory we make available to the counter to cache these component model counts, the faster it will compute the total model count of the input formula, since a larger cache allows for more counts stored at the same time, and thus a higher chance that a component's count is stored there and a cache hit occurs.

Interestingly, in an empirical evaluation of state-of-the-art DPLL-based model counter GANAK [9], it was found that the running time performance of GANAK does not vary much when we limit it to a very small cache, compared to giving it access to a much larger cache. In this work we investigate the reasons for this; specifically, we aim to answer the following questions:

Q1 How does the running time of GANAK relate to the cache size limit?

Q2 How do branching heuristics and cache management schemes influence the counter's interaction with the cache?

Q3 To what extent can we predict the *relevance* of a component's stored count, and use that knowledge to craft new cache management schemes?

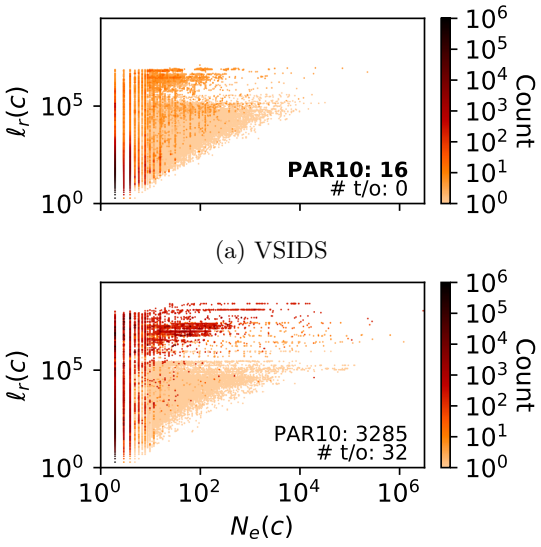
2 Approach and results

For our analysis, we used two synthetic benchmark sets, DQMR and GRID [8], as well as one industrial benchmark set, PA [5].

In order to answer our research questions, we built tracking functionality into GANAK that allows us to follow components as they are encountered at different times during the search. We used this information to carefully map the cache usage of GANAK under different circumstances.

We traced cache usage both for the default settings of GANAK, and for other *branching heuristics*, which have a great influence on the size and shape of the search tree and therefore can be suspected to also impact the way the counter uses the cache, and other *cache management schemes*, which determine how the

*Corresponding author j.g.rook@umail.leidenuniv.nl.



(a) VSIDS

(b) AUPC

Figure 1: Relevance window length ($\ell_r(c)$) as a function of number of encounters ($N_e(c)$) for components sampled from GRID instances.

cache is cleaned when it is full.

In order to conduct a fair comparison, we made GANAK highly configurable by parameterising magic constants and implementing additional branching heuristics and cache management schemes that were either known from the literature [7, 3, 11, 2] or crafted by us. We then used *automated algorithm configuration* (AAC) [4] to find benchmark-specific, optimised configurations for the parameters that were not fixed in our experiments.

We used this configurable version of GANAK to run a simple first experiment, in which we only fixed the maximum allowed cache size, and used AAC to find optimised parameter values. Answering **Q1**, we found that with the default parameter settings for GANAK as well as with the optimised ones, the influence of the cache size on running time performance, measured in terms of PAR10[1], was minimal. Doubling the cache size yielded at most a 28% speed up, but on average this was only 1.5%.

[1]Penalised average running time with penalty factor 10.

Figure 1 shows an example of a typical result from our experiments to answer **Q2**. Here, we limited the cache size to a maximum of 64 MB, fixed the branching heuristic to either VSIDS [7] or AUPC [7], used AAC to find optimised parameter settings for the other parameters and then measured for each component c how often GANAK encounters it in the search tree ($N_e(c)$) and how long it is 'relevant', *i.e.*, how much time passed between the first and last encounter, measured in the number of decision nodes created in that time ($\ell_r(c)$). Analysing the data, we find that for the best performing branching heuristic, VSIDS, there are more component encounters and that the components are relevant for shorter time periods, compared to the worst performing branching heuristic, AUPC.

Answering just one aspect of **Q2**, we found that, regardless of branching heuristic, the distributions of $N_e(c)$ follow a power law; this means that most components are only encountered once by GANAK. Herein lies an opportunity to improve cache management heuristics, because the counts of those components are stored in the cache, taking up space that could have been used to store model counts of components that could lead to cache hits.

Interestingly, we find that the branching heuristics that yield the smallest PAR10 values are the ones for which the solver finds relatively more components that are only relevant for very short periods of time (and thus for very small parts of the search tree). We take this as indication that such branching heuristics yield small search trees. It also explains why fairly simple cache management schemes, such as *first-in-first-out*, are likely to perform quite well. While this may be somewhat unsurprising, we believe that a study of a solver's interaction with the cache may yield interesting insights towards developing branching heuristics and caching schemes that form successful duos in reducing the running time of the solver.

Finally, inspired by Soos et al. [10], we took a machine learning approach to answering **Q3**. We used the aforementioned tracking functionality to construct a database of feature vec-

tors that capture for each tracked component information about its structure (*e.g.*, number of variables and number of Horn clauses) and the search state at the first time that it was encountered by the solver (*e.g.*, depth in the search tree and polarity of decisions). We combined these feature vectors with labels that each capture a proxy of a component's *relevance*, and use the resulting database to train binary classifiers to determine whether or not we should evict a stored component from the cache.

We evaluated the quality of these classifiers using the area under the precision-recall curves on the validation partition of our database. We found that all classification approaches improve over the baseline, which labelled all components as relevant, by at least a factor two. For the tasks where the threshold for being considered as relevant is the highest, the improvements even go up to an order of magnitude. Looking at the importance of the different features, we found that simple features that capture, for instance, the size of the component, have the most predictive power in this task.

3 Conclusion

We performed an extensive study of a state-of-the-art DPLL-style model counter, focusing on the role that component caching plays in reducing the running time of such model counters. Our results provide leads for future research aiming to develop novel cache management schemes designed to maximise the use of the cache, and thus to be not only faster, but also less reliant on limited memory resources. Specifically, we suggest that efforts to improve cache management should focus on reducing the number of single-encounter components whose counts are stored in the cache.

So far, we have considered components individually, divorced from their role as super- or sub-components of other components. Taking these relationships into account in the development of new cache management schemes is an interesting direction for future research.

References

[1] Bacchus, F., Dalmao, S., Pitassi, T.: DPLL with caching: A new algorithm for #SAT and Bayesian inference. Electron. Colloquium Comput. Complex **10**(003) (2003)

[2] Beame, P., Impagliazzo, R., Pitassi, T., Segerlind, N.: Memoization and DPLL: formula caching proof systems. In: Computational Complexity Conference. pp. 248–259. IEEE Computer Society (2003)

[3] Bliem, B., Järvisalo, M.: Centrality heuristics for exact model counting. In: ICTAI. pp. 59–63. IEEE (2019)

[4] Hoos, H.H.: Automated algorithm configuration and parameter tuning. In: Autonomous Search, pp. 37–71. Springer (2012)

[5] Möhle, S., Ge, C., Biere, A.: Program analysis benchmarks submitted to the model counting competition mc 2020 (2020), www.mccompetition.org, retrieved Aug. 2020

[6] Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: presented at SAT. p. 9 (2004), http://www.satisfiability.org/SAT04/programme/21.pdf

[7] Sang, T., Beame, P., Kautz, H.A.: Heuristics for fast exact model counting. In: SAT. Lecture Notes in Computer Science, vol. 3569, pp. 226–240. Springer (2005)

[8] Sang, T., Beame, P., Kautz, H.A.: Performing Bayesian inference by weighted model counting. In: AAAI. pp. 475–482. AAAI Press / The MIT Press (2005)

[9] Sharma, S., Roy, S., Soos, M., Meel, K.S.: GANAK: A scalable probabilistic exact model counter. In: IJCAI. pp. 1169–1176. ijcai.org (2019)

[10] Soos, M., Kulkarni, R., Meel, K.S.: Crystalball: Gazing in the black box of SAT solving. In: SAT. Lecture Notes in Computer Science, vol. 11628, pp. 371–387. Springer (2019)

[11] Thurley, M.: sharpSAT - counting models with advanced component caching and implicit BCP. In: SAT. Lecture Notes in Computer Science, vol. 4121, pp. 424–429. Springer (2006)