EasyChair Preprint

№ 7719

Vampire Getting Noisy: Will Random Bits Help Conquer Chaos? (System Description)

Martin Suda

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

April 4, 2022

Vampire Getting Noisy: Will Random Bits Help Conquer Chaos? (System Description)*

Martin Suda[0000−0003−0989−5800]

Czech Technical University in Prague, Czech Republic
martin.suda@cvut.cz

Abstract. Treating a saturation-based automatic theorem prover (ATP) as a Las Vegas randomized algorithm is a way to illuminate the chaotic nature of proof search and make it amenable to study by probabilistic tools. On a series of experiments with the ATP Vampire, the paper showcases some implications of this perspective for prover evaluation.

Keywords: Saturation-based Proving · Evalutation · Randomization.

1 Introduction

Saturation-based proof search is known to be fragile. Even seemingly insignificant changes in the search procedure, such as shuffling the order in which input formulas are presented to the prover, can have a huge impact on the prover's running time and thus on the ability to find a proof within a given time limit.

This *chaotic* aspect of the prover behaviour is relatively poorly understood, yet has obvious consequences for evaluation. A typical experimental evaluation of a new technique T compares the number of problems solved by a baseline run with a run enhanced by T (over an established benchmark and with a fixed timeout). While a higher number of problems solved by the run enhanced by T indicates a benefit of the new technique, it is hard to claim that a certain problem P is getting solved *thanks* to T. It might be that T just helps the prover get lucky on P by a complicated chain of cause and effect not related to the technique T—and the original idea behind it—in any reasonable sense.

We propose to expose and counter the effect of chaotic behaviours by deliberately *injecting randomness* into the prover and observing the results of many independently seeded runs. Although computationally more costly than standard evaluation, such an approach promises to bring new insights. We gain the ability to apply the tools of probability theory and statistics to analyze the results, assign confidences, and single out those problems that *robustly* benefit from the evaluated technique. At the same time, by observing the changes in the corresponding runtime distributions we can even meaningfully establish the effect of the new technique on a single problem in isolation, something that is normally inconclusive due to the threat of chaotic fluctuations.

* This work was supported by the Czech Science Foundation project 20-06390Y and the project RICAIP no. 857306 under the EU-H2020 programme.

In this paper, we report on several experiments with a randomized version of the ATP Vampire [8]. After explaining the method in more detail (Sect. 2), we first demonstrate the extent in which the success of a typical Vampire's proof search strategy can be ascribed to chance (Sect. 3). Next, we use the collected data to highlight the specifics of comparing two strategies probabilistically (Sect. 4). Finally, we focus on a single problem to see a chaotic behaviour smoothened into a distribution with a high variance (Sect. 5). The paper ends with an overview of related work (Sect. 6) and a discussion (Sect. 7).

2 Randomizing Out Chaos

Any developer of a saturation-based prover will confirm that the behaviour of a specific proving strategy on a specific problem is extremely hard to predict, that a typical experimental evaluation of a new technique (such as the one described earlier) invariably leads to both gains and losses in terms of the solved problems, and that a closer look at any of the "lost" problems often reveals just a complicated chain of cause and effect that steers the prover away from the original path (rather than a simple opportunity to improve the technique further).

These observations bring indirect evidence that the prover's behaviour is chaotic: A specific prover run can be likened to a single bead falling down through the pegs of the famous Galton board.[1] The bead follows a deterministic trajectory, but only because the code fixes every single detail of the execution, including many which the programmer did not care about and which were left as they are merely out of coincidence. We put forward here that any such fixed detail (which does not contribute to an officially implemented heuristic) represents a candidate location for randomization, since a different programmer could have fixed the detail differently and we would still call the code essentially the same.

Implementation: We implemented randomization on top of Vampire version 4.6.1; the code is available as a separate git branch.[2] We divided the randomization opportunities into three groups (governed by three new Vampire options).

Shuffling the input (`-si on`) randomly reorders the input formulas and, recursively, sub-formulas under commutative logical operations. This is done several times throughout the preprocessing pipeline, at the end of which a finished clause normal form is produced. Randomizing traversals (`-rtra on`) happens during saturation and consists of several randomized reorderings including: reordering literals in a newly generated clause and in each given clause before activation, and shuffling the order in which generated clauses are put into the passive set. It also (partially) randomizes term ids, which are used as tiebreakers in various term indexing operations and determine the default orientation of equational literals in the term sharing structure. Finally, "randomized age-weight ratio" (`-rawr on`) swaps the default, deterministic mechanism for choosing the

[1] https://en.wikipedia.org/wiki/Galton_board
[2] https://github.com/vprover/vampire/tree/randire

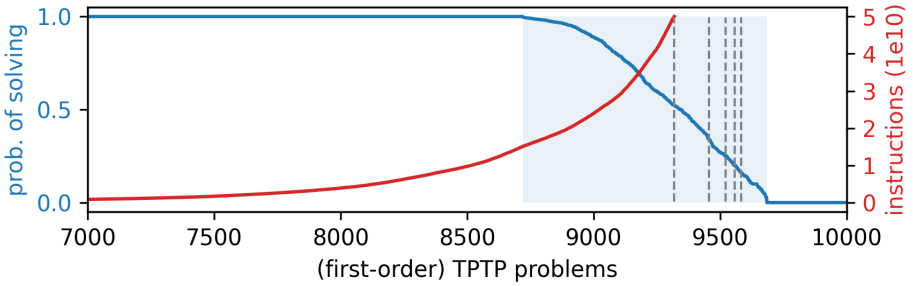

Fig. 1. Blue: first-order TPTP problems ordered by the decreasing probability of being solved by the `dis10` strategy within 50 billion instruction limit. Red: a cactus plot for the same strategy, showing the dependence between a given instruction budget (y-axis) and the number of problems on average solved within that budget (x-axis).

next queue to select the given clause from [12] for a randomized one (which only respects the age-weight ratio probabilistically).

All the three options were active by default during our experiments.

3 Experiment 1: A Single-Strategy View

First, we set out to establish to what degree the performance of a Vampire strategy can be affected by randomization. We chose the default setting of the prover except for the saturation algorithm, which we set to Discount, and the age-weight ratio, set to 1:10 (calling the strategy `dis10`). We ran our experiment on the first-order problems from the TPTP library [13] version 7.5.0.[3]

To collect our data, we repeatedly (with different seeds) ran the prover on the problems, performing full randomization. We measured the executed instructions[4] needed to successfully solve a problem and used a limit of 50 billion instructions (which roughly corresponds to 15 s of running time on our machine[5]) after which a run was declared unsuccessful. We ran the prover 10 times on each problem and additionally as many times as required to observe the instruction count average (over both successful and unsuccessful runs) stabilize within 1 % from any of its 10 previously recorded values.[6]

A summary view of the experiment is given by Fig. 1. The most important to notice is the shaded region there, which spans 965 problems that were solved by `dis10` at least once but not by every run. In other words, these problems have probability p of being solved between $0 < p < 1$. This is a relatively large number

[3] Materials accompanying the experiments can be found at https://bit.ly/3JDCwea.

[4] Please see Appendix A for why we chose limiting our prover runs by the number of executed instructions as opposed to the more usual wall-clock time.

[5] A server with Intel(R) Xeon(R) Gold 6140 CPUs @ 2.3 GHz and 500 GB RAM.

[6] Utilizing all the 72 cores of our machine, such data collection took roughly 12 hours.

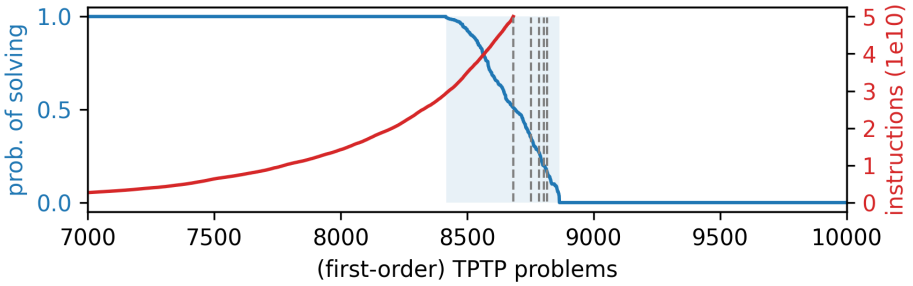

Fig. 2. The effect of turning AVATAR off in the `dis10` strategy (cf. Fig. 1).

and can be compared to the 8720 "easy" problems solved by every run. The collected data implies that 9319.1 problems are being solved on average (marked by the left-most dashed line in Fig. 1) with a standard deviation $\sigma = 11.7$. The latter should be an interesting indicator for prover developers: beating a baseline by only 12 TPTP problems can easily be ascribed just to chance.

Fig. 1 also contains the obligatory "cactus plot" (explained in the caption), which—thanks to the collected data—can be constructed with the "on average" qualifier. By definition, the plot reaches the left-most dashed line for the full instruction budged of 50 billion. The subsequent dashed lines mark the number of problems we would on average expect to solve by running the prover (independently) on each problem twice, three, four and five times. This is an information relevant for strategy scheduling: e.g., one can expect to solve whole additional 137 problems by running randomized `dis10` for a second time.

Not every strategy exhibits the same degree of variability under randomization. Observe Fig. 2 with a plot analogous to Fig. 1, but for `dis10` in which the AVATAR [14] has been turned off. The shaded area there is now much smaller (and only spans 448 problems). The powerful AVATAR architecture is getting convicted of making proof search more fragile and the prover less robust.[7]

Remark Randomization incurs a small but measurable computational overhead. On a single run of `dis10` over the first-order TPTP (filtering out cases that took less than 1 s to finish, to prevent distortion by rounding errors) the observed median relative time spent randomizing on a single problem was 0.47 %, the average 0.59 %, and the worse[8] 13.86 %. Without randomization, the `dis10` strategy solved 9335 TPTP problems under the 50 billion instruction limit, i.e., 16 problems more than the average reported above. Such is the price we pay for turning our prover into a Las Vegas randomized algorithm.

[7] Another example of a strong but fragile heuristic is the lookahead literal selection [4], which selects literals in a clause based on the current content of the active set: `dis10` enhanced with lookahead solves 9512.4 (±13.8) TPTP problems on average, 8672 problems with $p = 1$ and additional 1382 (!) problems with $0 < p < 1$.

[8] On the hard-to-parse, trivial-to-solve `HWV094-1` with 361 199 clauses.

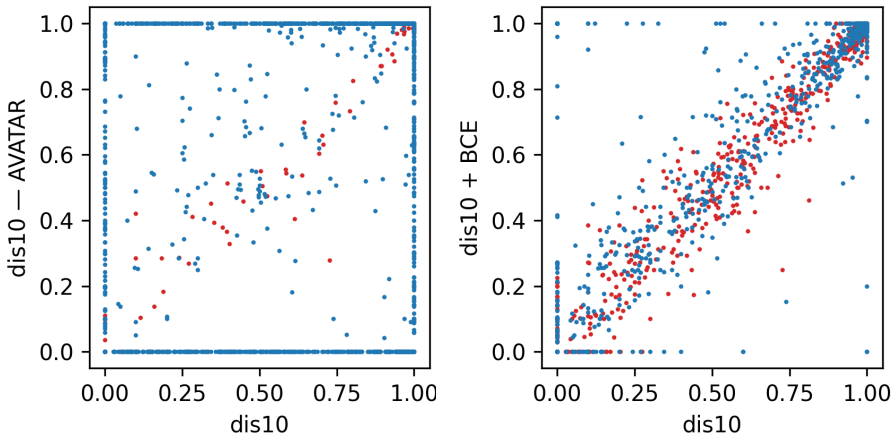

Fig. 3. Scatter plots comparing probabilities of solving a TPTP problem by the baseline `dis10` strategy and 1) `dis10` with AVATAR turned off (left), and 2) `dis10` with blocked clause elimination turned on (right). On problems marked red the respective technique could not be applied (no splittable clauses derived / no blocked clauses eliminated).

4 Experiment 2: Comparing Two Strategies

Once randomized performance profiles of multiple strategies are collected, it is interesting to look at two at a time. Fig. 3 shows two very different scatter plots, each comparing our baseline `dis10` to its modified version in terms of the probabilities of solving individual problems.

On the left we see the effect of turning AVATAR off. The technique affects the proving landscape quite a lot and most problems have their mark along the edges of the plot where at least one of the two probabilities has the extreme value of either 0 or 1. What the plot does not show well, is how many marks end up at the extreme corners. These are: 7896 problems easy for both, 661 easy for AVATAR and hard without, 135 hard for AVATAR and easy without.

Such "purified", one-sided gains and losses constitute a new interesting indicator of the impact of a given technique. They should be the first to look at, e.g., during debugging, as they represent the most extreme but robust examples of how the new technique changes the capabilities of the prover.

The right plot is an analogous view, but now at the effect of turning on *blocked clause elimination* (BCE). This is a preprocessing technique coming from the context of propositional satisfiability [6] extended to first-order logic [7]. We see that here most of the visible problems show up as marks along the plot's main diagonal, suggesting a (mostly) negligible effect of the technique. The extreme corners hide: 8648 problems easy for both, 17 easy with BCE (11 satisfiable and 6 unsatisfiable), and 2 easy without BCE (1 satisfiable and 1 unsatisfiable).

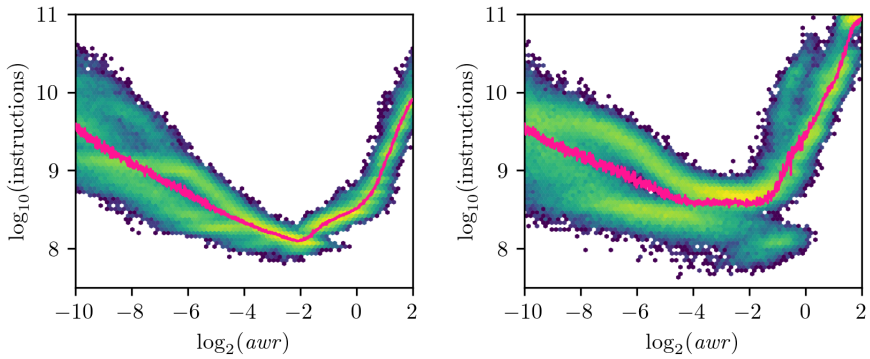

Fig. 4. 2D-histograms for the relative frequencies (color-scale) of how often, given a specific *awr* (x-axis), solving `PRO017+2` required the shown number of instructions (y-axis). The curves in pink highlight the mean y-value for every x. The performance of `dis10` (left) and the same strategy enhanced by a goal-directed heuristic (right).

5 Experiment 3: Looking at One Problem at a Time

In their paper on age/weight shapes [12, Fig. 2], Rawson and Reger plot the number of given-clause loops required by Vampire to solve the TPTP problem `PRO017+2` as a function of age/weight ratio (*awr*), a ratio specifying how often the prover selects the next clause to activate from its age-ordered and weight-ordered queues, respectively. The curve they obtain is quite "jiggly", indicating a fragile (discontinuous) dependence. Randomization allows us to smoothen the picture and reveal new, until now hidden, (probabilistic) patterns.

The 2D-histogram in Fig. 4 (left) was obtained from 100 independently seeded runs for each of 1200 distinct values of *awr* from between 1:1024 = 2^{-10} and 4:1 = 2^2. We can confirm Rawson and Reger's observation of the best *awr* for `PRO017+2` lying at around 1:2. However, we can now also attempt to explain the "jiggly-ness" of their curve: With a fragile proof search, even a slight change in *awr* effectively corresponds to an independent sample from the prover's execution resource[9] distribution, which—although changing continuously with *awr*—is of a high variance for our problem (note the log-scale of the y-axis).[10]

The distribution has another interesting property: At least for certain values of *awr* it is distinctly multi-modal. As if the prover can either find a proof quickly (after a lucky event?) or only after much harder effort later and almost nothing in between. Shedding more light on this phenomenon is left for further research.

It is also very interesting to observe the change of such a 2D-histogram when we modify the proof search strategy. Fig. 4 (right) shows the effect of turning

[9] Rawson and Reger [12] counted given-clause loops, we measure instructions.

[10] Even with 100 samples for each value of *awr*, the mean instruction count (rendered in pink in Fig. 4) looks jiggly towards the weight-heavy end of the plot.

on SInE-level split queues [2], a goal directed clause selection heuristic (Vampire option `-slsq on`). We can see that the mean instruction count gets worse (for every tried *awr* value) and also the variance of the distribution distinctly increases. A curious effect of this is that we observe the shortest successful runs with `-slsq on`, while we still could not recommend (in the case of `PRO017+2`) this heuristic to the user. The probabilistic view makes us realize that there are competing criteria of prover performance for which one might want to optimize.

6 Related Work

The idea of randomizing a theorem prover is not new. Ertel [1] studied the speedup potential of running independently seeded instances of the connection prover SETHEO [9]. The dashed lines in our Figs. 1 and 2 capture an analogous notion in terms of "additional problems covered" for levels of parallelism $1 - 5$. randoCoP [11] is a randomized version of another connection prover, leanCoP 2.0 [10]: especially in its incomplete setup, several restarts with different seeds helped randoCoP improve over leanCoP in terms of the number of solved problems. Gomes et al. [3] notice that randomized complete backtracking algorithms for propositional satisfiability lead to heavy-tailed runtime distributions on satisfiable instances. While we have not yet analyzed the runtime distributions coming from saturation-based first-order proof search in detail, we definitely observed high variance also for unsatisfiable problems. An interesting view on the trade-offs between expected performance of a randomized solver and the risk associated with waiting for an especially long run to finish is given by Huberman et al. [5]. This is related to the last remark of the previous section.

7 Discussion

As we have seen, the behaviour of a state-of-the-art saturation-based theorem prover is to a considerable degree chaotic and on many problems a mere perturbation of seemingly unimportant execution details decides about the success or the failure of the corresponding run. While this may be seen as a sign of our as-of-yet imperfect grasp of the technology, the author believes that an equally plausible view is that some form of chaos is inherent and originates from the complexity of the theorem proving task itself. (A higher-order logic proof search is expected to exhibit an even higher degree of fragility.)

This paper has proposed randomization as a key ingredient to a prover evaluation method that takes the chaotic nature of proof search into account. The extra cost required by the repeated runs, in itself not unreasonable to pay on contemporary parallel hardware, seems more than compensated by the new insights coming from the probabilistic picture that emerges. Moreover, other uses of randomization are easy to imagine, such as data augmentation for machine learning approaches or the construction of more robust strategy schedules. It feels that we only scratched the surface of the opened-up possibilities. More research will be needed to fully harness the potential of this new perspective.

References

1. Ertel, W.: OR-parallel theorem proving with random competition. In: Voronkov, A. (ed.) LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings. LNCS, vol. 624, pp. 226–237. Springer (1992)
2. Gleiss, B., Suda, M.: Layered clause selection for saturation-based theorem proving. In: Fontaine, P., Korovin, K., Kotsireas, I.S., Rümmer, P., Tourret, S. (eds.) PAAR 7, Paris, France, June-July, 2020. CEUR Workshop Proceedings, vol. 2752, pp. 34–52. CEUR-WS.org (2020), http://ceur-ws.org/Vol-2752/paper3.pdf
3. Gomes, C.P., Selman, B., Crato, N., Kautz, H.A.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. J. Autom. Reason. **24**(1/2), 67–100 (2000)
4. Hoder, K., Reger, G., Suda, M., Voronkov, A.: Selecting the selection. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings. LNCS, vol. 9706, pp. 313–329. Springer (2016)
5. Huberman, B., Lukose, R., Hogg, T.: An economics approach to hard computational problems. Science (New York, N.Y.) **275**, 51–4 (02 1997)
6. Järvisalo, M., Biere, A., Heule, M.: Blocked clause elimination. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings. LNCS, vol. 6015, pp. 129–144. Springer (2010)
7. Kiesl, B., Suda, M., Seidl, M., Tompits, H., Biere, A.: Blocked clauses in first-order logic. In: Eiter, T., Sands, D. (eds.) LPAR-21, Maun, Botswana, May 7-12, 2017. EPiC Series in Computing, vol. 46, pp. 31–48. EasyChair (2017)
8. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. LNCS, vol. 8044, pp. 1–35. Springer (2013)
9. Letz, R., Schumann, J., Bayerl, S., Bibel, W.: SETHEO: A high-performance theorem prover. J. Autom. Reason. **8**(2), 183–212 (1992)
10. Otten, J.: leanCoP 2.0 and ileanCoP 1.2: High performance lean theorem proving in classical and intuitionistic logic (system descriptions). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings. LNCS, vol. 5195, pp. 283–291. Springer (2008)
11. Raths, T., Otten, J.: randoCoP: Randomizing the proof search order in the connection calculus. In: Konev, B., Schmidt, R.A., Schulz, S. (eds.) PAAR 1, Sydney, Australia, August 10-11, 2008. CEUR Workshop Proceedings, vol. 373. CEUR-WS.org (2008), http://ceur-ws.org/Vol-373/paper-08.pdf
12. Rawson, M., Reger, G.: Old or heavy? Decaying gracefully with age/weight shapes. In: Fontaine, P. (ed.) CADE 27, Natal, Brazil, August 27-30, 2019, Proceedings. LNCS, vol. 11716, pp. 462–476. Springer (2019)
13. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. Journal of Automated Reasoning **59**(4), 483–502 (2017)
14. Voronkov, A.: AVATAR: the architecture for first-order theorem provers. In: Biere, A., Bloem, R. (eds.) CAV 2014, Vienna, Austria, July 18-22, 2014. Proceedings. LNCS, vol. 8559, pp. 696–710. Springer (2014)

A Limiting Instructions

In this paper, we limit the prover runs by the number of executed instructions (as measured by the `perf` tool or with the help of the `__NR_perf_event_open` system call) instead of the typical limiting by the wall-clock time.

The local servers that we use for our experiments have 72 (hyper-threaded) cores and 500 GB RAM. While parallelisation is desirable to save time (and the memory is sufficient for many short-lived concurrent prover runs), we observed that the performance of the prover measured by the number of problems solved within a given time limit significantly depends on the degree of parallelization we use. This is probably caused by a competition of the concurrent processes for memory accesses (and for the cache use) and, as the following experiment shows, can be quite severe. That is why we decided to switch to instruction count limiting, which appears to be more stable (although it is not exactly commensurate with time).

In the experiment, we ran Vampire 5 times over TPTP with different degrees of parallelization, namely using 5, 15, 35, 70, and 100 parallel processes. Note that 70 makes the server almost "full" and 100 makes it "overfull" (and context switching has to be used to satisfy all the running instances of the prover). The runs were limited using time (with a timeout of 10 s) and instructions were measured with the help of the `perf` tool invoked as in:

```
perf stat -e instructions:u ./vampire -t 10 <other arguments>
```

If we first look at the number of problems solved by the individual (otherwise identical) invocations we get the following picture:

parallelization	5	15	35	70	100
solved	9437	9370	9229	9090	8918

Even is we discard the result of the "overfull" run, the difference between parallelization 70 and 5 is 347 problems, a number that would make a theorem prover developer very happy if it would correspond to a performance boost by a new clever heuristic (evaluated under reproducible conditions).

In Fig. 5, we find a scatter plot from this experiment, showing successful Vampire runs (groups with the same level of parallelism share a color) with the point coordinates corresponding to the measured time and instructions. We can see that each color group defines a distinct "slope" of how fast the instructions can be used up in the slowest possible rate (the behaviour at higher rates is much more noisy). Quite surprising is that the distinction between parallelization 15 and parallelization 5 is quite substantial (even a bit stronger, it seems, than the distinction between parallelization 70 and parallelization 35). Memory bandwidth competition seems to be a plausible explanation for this.

The outliers above the "main slope" of each group (i.e. the examples of "higher rates of instruction burn") could be explained by imagining the prover running

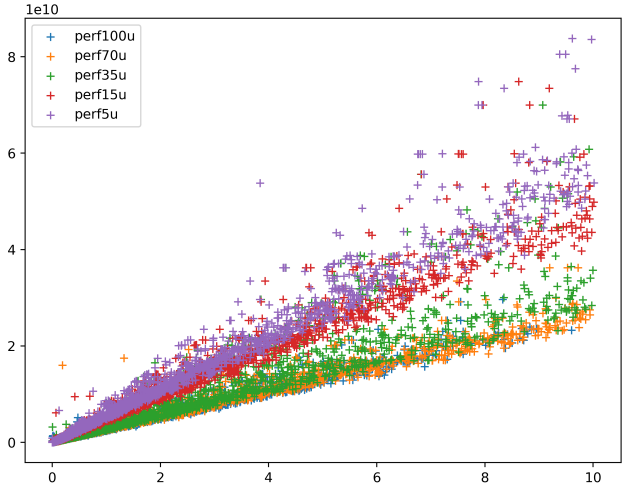

Fig. 5. Five runs of Vampire (`dis10`) on the TPTP library under different levels of parallelization (distinguished by different colors). Each mark corresponds to a successful prover run, the x-coordinate denoting its termination time (s) and the y-coordinate the number of measured (user) instructions (in tens of billions).

on some atypical problem where a lot of computation (instruction burning) can be suddenly done without many memory accesses or only localized ones (which a cache could satisfy).

Finally, while there is a lot of variation in the figure, when grouping the successful runs by the problems, the measured instruction counts were always "almost the same": the *coefficient of variation*[11] was in the order of 10^{-3}. (The worse, i.e. that largest σ relative to the mean, was 0.0014 for measured values of 36 393 304, 36 263 483, 36 238 127, 36 292 870, 36 298 029.)

Possible limitations of instruction limiting: Since wall-clock time is the ultimate metric the impatient user is interested in optimizing for (although instructions could more reliably correspond to power consumption), by switching to instruction limiting we risk developing a prover which does not make our users as happy as we otherwise could. If the hypothesis about memory bandwidth as the bottleneck is correct, we should not, for instance, use instruction limiting when trying to tune a new version of a prover for better cache usage. (In such a context, the improved prover would finish faster after burning the same number of instructions). On the other hand, the development of smarter heuristics, where we typically perform additional computations in the hope of making better decisions, should be unaffected by this.

[11] https://en.wikipedia.org/wiki/Coefficient_of_variation