

Impact- and Cost-Oriented Propagator Scheduling for Faster Constraint Propagation

Georg Ringwelski¹ and Matthias Hoche²

¹ 4C, University College Cork, Ireland g.ringwelski@4c.ucc.ie

² Fraunhofer FIRST, Kekuléstr.7, 12489 Berlin, Germany mathoc@first.fhg.de

Abstract. Constraint Propagation can be speeded up significantly by choosing a good execution order for propagators. A propagator is an implicit representation of a constraint which is widely used in today's powerful constraint solvers. In this paper we evaluate different ways to find good execution orders automatically during runtime. We extend previous work in this area by two new techniques: fair-scheduling and impact-oriented prioritization of propagators.

1 Introduction

Constraint Propagation is one of the key techniques of Constraint Programming. In finite integer domains Propagation is often based on efficient filtering algorithms of constraints in compiled form. These algorithms implement so-called *propagators*. Thus, they represent the semantics of their associated constraint in an implicit way and detect inconsistencies by inferring wipe-outs of variable domains. Propagators are the basis of constraint propagation in the currently fastest finite domain constraint solvers which include SICStus Prolog, ECLⁱPS^e, ILOG Solver, CHIP, Mozart, GNU Prolog and others.

The propagators are used by the constraint solver to compute a fixed point of constraint propagation as a preprocessing step or during search. This fixed point can be a bounds-consistent CSP but there are no general requirements on the properties of that fixed point. However, given a set of propagators the fixed point is uniquely determined. To compute this determined fixed point, the propagators have to be re-executed until no further domain restrictions are inferable. Naturally this can be done in arbitrary ways and there are many different orders of execution which lead to different performance. Very little is known (to the public) about the execution order in the mentioned commercial solvers.

Related Work. It seems that most of the mentioned constraint solvers use a priority queue in which the propagators are buffered for execution. CLP-based languages, such as ECLⁱPS^e or SCISStus use (2 – 13 different) static priority values associated to each constraint respectively its propagator. The propagators with higher priority are executed earlier and thus more often. Little was published about which priority-values are chosen for the constraints [7, 8, 11]. Most of the used priority values seem to be set according to the complexity

of the propagator which they are associated to: expensive computations are delayed until all the cheap things are finished. In ECLⁱPS^e user-defined constraints can be associated to priority values between 1 and 12, but we are not aware of any guidelines on which numbers to choose. Similar mechanisms are available in implementations of CHR [6], where the application of certain rules can be prioritized. However, CHR does not provide any automatic mechanisms to find appropriate values at all. Another way to prioritize was found in the context of propagator learning for GNU Prolog [9]: propagators are prioritized regarding the tightness of their respective constraints. Propagators of tight constraints are executed earlier. In some CP systems also other data structures than priority queues, such as stacks for example, can be used to buffer the propagators (e.g. CHOCO). But again, the decision of which data structure to use is left to the user and guidance on how to make this decision is missing. In early work on CSP Wallace and Freuder [12] investigated several heuristics to choose the best arc during Arc-consistency enforcement with AC-3. Many of their ideas can also be found in work on propagator scheduling. They additionally proposed some heuristics which we plan to apply to propagator based constraint propagation such as learning good priority-values during execution.

Contribution. In this paper we investigate the correlation of execution orders for propagators with the result and performance of constraint solving. In order to ensure correct results we prove that the execution order can be safely manipulated and that idempotent propagators can be omitted from being re-inserted when they are already waiting for execution. Knowing this, we can compare the performance of various execution orders which can be automatically computed during runtime because the result is uniquely determined. The order will be specified by various buffers to store propagators for their execution: FIFO, LIFO, a priority queue and a fair scheduling buffer known from CPU scheduling in Operating Systems. Fairness was not considered in propagator scheduling before. These buffers are varied to become sets, i.e. not store any idempotent propagator twice [10, 8, 11]. Furthermore, we consider two classes of prioritization: cost-oriented [7, 11] and impact-oriented. The first prioritizes computationally cheap propagators, the latter prioritizes propagators which may have a large impact, i.e. which can be expected to prune large portions of the search space. The latter is a new class of execution orders which was not considered before. Finally we compare the use of static priority-values to the dynamic adaptation of the values during runtime.

Organization. In the next Section we describe the theoretical background of constraint propagation with propagators and show that the execution order can be safely varied. In Section 3 we specify more precisely the setting of this investigation. We identify a (necessarily supplied) data structure with which we can manipulate the execution order of propagators and propose and verify several implementations of it. In Section 4 we evaluate these implementations with different benchmark problems and conclude in Section 5.

2 Theoretical Background

A Constraint Satisfaction Problem (V, C, D) is given by a set of variables $V = \{v_1, v_2, \dots, v_n\}$, a set of constraints C and a set of variable domains $D = \{D_1, D_2, \dots, D_n\}$ associated to the variables. Each constraint $c \in C$ is specified by its scope which is a vector of variables (v_i, \dots, v_j) and its semantics, which describes the allowed simultaneous assignments of the variables and is thus a subset of $D_i \times \dots \times D_j$. In this paper we assume all domains to be finite sets of integers.

Constraint Propagation is generally formalized by the Chaotic Iteration (CI) of propagators³ [4]. Given a CSP (V, C, D) a set of propagators $prop(c)$ is deduced from the semantics of any constraint $c \in C$. Each constraint can use more than one propagator to implement its semantics (by sequentially executing all of them) [10, 11]. Each propagator p uses input variables $in(p)$ from which it may infer reductions of the domains of its output-variables $out(p)$ of which it can change the domain. Propagators are monotonic functions (i.e. $D \sqsubset D' \Leftrightarrow p(D) \sqsubset p(D')$) in the Cartesian product of the powersets of all variable domains $\mathcal{P}(D_1) \times \dots \times \mathcal{P}(D_n)$. Furthermore, propagators must ensure that for each variable the domain can only be restricted but not extended: given a propagator p with $p((D_1, \dots, D_n)) = (D'_1, \dots, D'_n)$, then $D'_i \subseteq D_i$ must hold. Given this, the Chaotic Iteration over all propagators induced by C (i.e. $\bigcup_{c \in C} prop(c)$) will reach a uniquely determined fixed point [1]. This is a vector of variable domains for which no further restrictions can be inferred with the used propagators. The starting point for this approximation are the initial domains D .

However, Chaotic Iteration does not define how the fixed point is computed. In theory, each propagator is executed infinitely often to reach the fixed point. When we actually want to solve a CSP we naturally have to find ways to reach this fixed point without infinite computations. A round-robin algorithm will have poor performance, as most propagators that are executed will not be able to infer any further domain reductions. Thus, a more sophisticated algorithm is used in most solvers today. It is described in detail for `clp(FD)` (i.e. the predecessor of GNU Prolog) in [3]. A more formal definition of this concept can be found in [10, 11]. The idea is to store for each variable v all propagators p with $v \in in(p)$ and relate them to domain events on v after which they have to be triggered. With this, only those propagators are re-executed that can possibly infer further domain reductions.

Example 1. A propagator of a constraint $v < w$ only needs to be re-executed upon changes on the largest value of D_w and the smallest value of D_v , all other changes of D_v or D_w may not lead to further domain reductions and can thus be omitted.

³ Many different names for these procedures can be found in the literature. When we restrict ourselves to finite integer domains, they are all practically the same: propagators [11], domain reduction functions [1], closure-operators [5], constraint frames [3] etc.

The generally desired result of Constraint Propagation in a CSP (V, C, D) is generally considered the fixed point of the Chaotic Iteration of the propagators which are induced by C . To start, each propagator will have to be executed at least once. After that, new domain reductions may possibly be inferred (only) from the newly reduced domains. Thus some propagators will have to be executed again in order to approximate the desired fixed point and so on. We define Event-based Propagation to use a simplified version of this execution model. Without restricting generality, we consider only one sort of events, namely *any* change of the domain of one variable. For any other kind of event, as they are listed in [3] or [11] for example, we could define a further iteration rule in the following definition.

Definition 1. *In A CSP (V, C, D) , Event-based Propagation (EBP) is defined iteratively by*

1. *For each $c \in C$ the propagators $\text{prop}(c)$ are executed once*
2. *If D_v is reduced, then all propagators $p \in \text{prop}(c)$ with $v \in \text{in}(p)$ and $c \in C$ are executed.*

We can show that EBP leads to the desired result. This is the fixed point of the Chaotic Iteration of the propagators that are implied by the posted constraints.

Theorem 1. *The Event-Based Propagation of a CSP $\mathcal{A}(C, V, D)$ will lead to the same result as the Chaotic Iteration of $\bigcup_{c \in C} \text{prop}(c)$ starting at D .*

Proof. Soundness : Since the same propagators are used, there cannot exist any values that are pruned in EBP, but not in the Chaotic Iteration.

Completeness : If EBP was not complete, then there would have to be a value that is pruned with CI, but not with EBP. This would imply that a propagator which has the potential to prune that value is not executed in EBP. As every propagator is defined to be executed at least once, this propagator would have to be not re-executed after a domain reduction. However, this contradicts the definition of EBP and can thus not have happened.

This theorem implies that the order in which the propagators are executed is irrelevant. The only crucial requirement for the correctness is that each propagator is re-executed whenever a respective event has occurred.

Corollary 1. *EBP is correct with any execution order of the propagators.*

3 Propagator Scheduling

During propagation, the propagators are executed and re-executed when they are triggered due to a domain reduction. Since many constraints can be triggered by one reduction we need a buffer to store all the propagators which remain to

be executed. The basic version of such a buffer is a queue, i.e. a FIFO data structure. However, there seems to be a lot of potential to speed up constraint propagation by using more sophisticated buffers [10]. The desired effect of this is to make the execution of certain propagators obsolete, because their effect has already been achieved.

Example 2. Consider the CSP $A < B, B < C$. A propagator for each constraint must be executed at least once such that the initial buffer will be $\langle A < B, B < C \rangle$. After one execution we might obtain $\langle B < C, B < C \rangle$ which will lead to an obsolete execution of $B < C$.

The question we want to answer in this paper is: how can we implement the buffer in order to speed up EBP by omitting useless executions of propagators? Schulte and Stuckey have already described three classes of optimizations in [11]: EBP as described above; prevent the re-insertion of idempotent propagators; execute preferably the propagators with low computational complexity. In this paper we concentrate on EBP as this is most widely used and seems to be most efficient. We investigate a wider set of possible buffers and consider also impact-oriented prioritization and thus qualitatively extend the state-of-the-art.

The characterizing property of a buffer is its implementation of the methods *push* and *pop* which add respectively delete values from the buffer. We consider four basic data structures for the buffer:

FIFO a standard queue, *pop* will always return the item which was *pushed* first.

LIFO a standard stack, *pop* will always return the item which was *pushed* last.

Order- \prec a static priority queue, *pop* will always return the item which has the highest priority wrt. \prec

Sched- \prec a fair priority queue, *pop* will usually return the item with the largest wrt. \prec , this may differ when low-priority items are waiting too long.

Order- \prec will always return the highest priority item it stores. When two items have the same priority, we schedule them in FIFO manner. To depend the scheduling only on priorities can lead to “starvation” of items with low priority. Starvation can be prevented by a fair scheduling algorithm as used in many areas of computer science (e.g. CPU scheduling) today. For propagator scheduling we can only use non-preemptive algorithms to prevent Reader-Writer-Problems on the variable domains. The efficiency-results known from CPU scheduling for example cannot be expected to hold in propagator scheduling, because the number of future jobs depends on the prioritization⁴. We use a simple form of an “aging” algorithm to implement **Sched- \prec** . Aging prevents starvation by considering a combination of the actual priority and the time a task is waiting for execution. We implement this by setting the priority in Sched- \prec to the quotient of the priority in Order- \prec over the solver lifetime when the propagator is added to the buffer.

⁴ “shortest job first” will not necessarily be the best for our purposes although we know the costs of the jobs, i.e. the propagator’s computational complexity.

In order to improve the basic buffers, we considered methods to prevent the multiple storage of idempotent propagators [10, 8, 11]. This means, that any propagator will only be stored, if it is not already in the buffer. As we thus potentially execute less propagators than EBP, we need to show that Propagation remains correct.

Theorem 2. *EBP will remain correct, if idempotent propagators are not multiply stored in the propagator queue.*

Proof. Because of Corollary 1 we do not restrict generality when we make the following assumption: whenever two identical propagators are stored in the buffer, then they will be executed directly after each other. Since we assume propagators to be idempotent, the execution of the second will not have any effect and can thus be omitted.

Knowing this, we can safely prevent the storage of idempotent propagators which are already in the buffer. For this we propose a variant for all of the above described basic data types:

X-set will not *push* an item, if it is already stored in buffer **X**.

What remains to specify are the possible orders \prec we use for the **Order-** \prec and **Sched-** \prec buffers. We propose the following:

- comp** the complexity of the propagator, as also described in [7, 11]
- bound** a lower bound of the expected pruning achieved by the propagator (e.g. the portion of bounds-consistent values of its variables)
- tight** an estimation of the tightness of the constraint

The latter two orders were to our knowledge not (knowingly⁵) applied to propagator scheduling before. With considering the expected pruning (**bound**) and the estimated tightness (**tight**) we tackle the topic of the predicted impact of a propagator. A tight constraint can be expected to have a larger impact than a loose constraint, i.e. it can be expected to prune more values and thus make the execution of other propagators obsolete. According to the fail-first-principle [2] propagators of constraints with a high expected impact should thus be executed first. They have the highest potential to prune the domains and will usually be able to prune the most.

Finally all these prioritization-strategies can be applied statically and dynamically. In the static version the respective values will be computed once, when the constraint is first inserted. In the dynamic case, the priority is computed whenever the propagator is (re-)inserted in the buffer.

X-Y-dyn will compute the priority **Y** dynamically before every insertion to the buffer **X**.

⁵ Arnaud Lallout denied that this is done by their algorithm after the presentation of [9] at the FLAIRS'04 conference.

4 Empirical Evaluation

All our methods target the reduction of the number of executed propagators during constraint solving. However, they all yield extra computational costs which have in our implementations constant or linear worst case complexity. Thus an empirical evaluation is necessary to show the usefulness of the proposed methods. We think that it does not make much sense to count the executed propagators for evaluation purposes, since the propagators differ a lot in complexity. A more fine-grained metric, which counts separately for propagators of different complexity classes would make more sense but the results would be very hard to compare. Instead we use good old runtime to check whether the extra effort used for smart propagator scheduling pays. All experiments were run with the firstcs solver [7] on a 1.8GHz Linux PC. In the following we use the abbreviations described in the previous section to specify the used algorithm. For example Order-set-comp-dyn will thus stand for experiments with an complexity-ordered dynamic queue buffer where multiple insertions of idempotent propagators are prevented.

It can be seen in [11] that (in contrast to the counted propagation steps) no single method can be expected to be always the best. Thus it is essential to consider various benchmark problems, which use varying combinations of used constraints:

queens The famous 27-queens problem implemented with primitive constraints only

queens-a 27-queens implemented with three AllDifferent constraints

golomb The golomb ruler problem with 10 marks and the optimal length of 55 implemented with primitive constraints only

golomb-a The golomb ruler with global constraints

L10x5,L15x5,L10x10.1,L10x10.3,L10x10.5,FT10x10 Job-Shop-Scheduling

Problems from the OR library which can be found at

<http://www.brunel.ac.uk/depts/ma/research/jeb/info.html>.

These problems are implemented with global scheduling constraints and simple in-equations.

First we compare the basic buffer types. Figure 1 shows the respective results. They can be summarized as follows:

- The effort to use the **set**-variant of **LIFO** almost always pays significantly. Thus we used this variant for all further tests.
- **FIFO** is always better than **LIFO**.
- **Order** is better than non-prioritized buffers whenever some diversity in the priority of the propagators exist. This applies to all investigated problems but **Order** profits only when global constraints are used.
- The extra effort to ensure fairness (**Sched** vs. **Order**) during propagation in priority queues does not pay. This may, however, result from a poor aging algorithm. We plan to find better parameters for this in future work.

Next we evaluate the various priority-computations. The results are shown in Figure 2. It can be seen that:

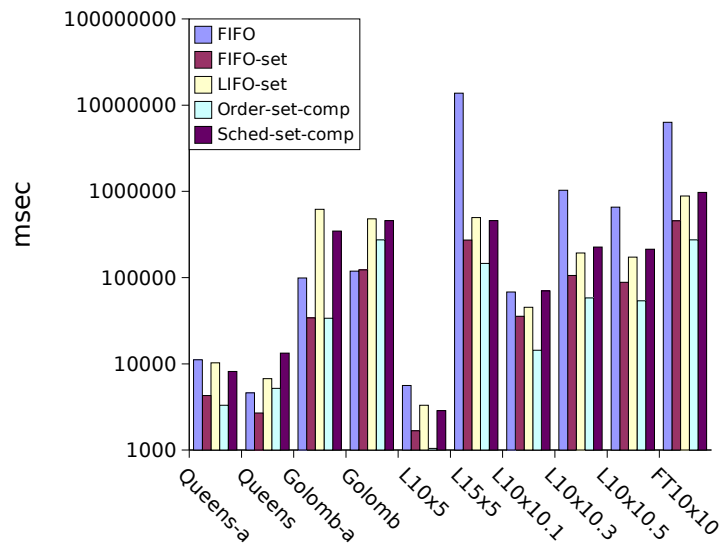


Fig. 1. Comparison basic buffer types.

- The estimation of the tightness seems to be a good measure to express the expected impact of a constraint. The performance of **bound** and **tight** are very similar in almost all tests.
- Whenever global constraints are involved, the complexity based prioritization **comp1** is better and otherwise the impact based **tight/bound** are. The poor performance of impact-based methods with global constraints may result from bad estimations of the priority values. The impact of arithmetic constraints can be predicted much more precisely.

Finally we check whether the evaluation of the priority before every insertion of a propagator into the buffer pays. Thus we compare the **dyn** versions of the buffers to the standard case where the priority is only computed once upon the construction of the constraint. The results are presented in Figure 3, they can be summarized as follows:

- It seems that only in problems without global constraints and in combination with the **comp**-priority the dynamic versions perform better than the static versions.
- The relatively complex computation of the expected impact seems to be far too costly to be computed dynamically. The combination **bound-dyn** yields poor results in all experiments.

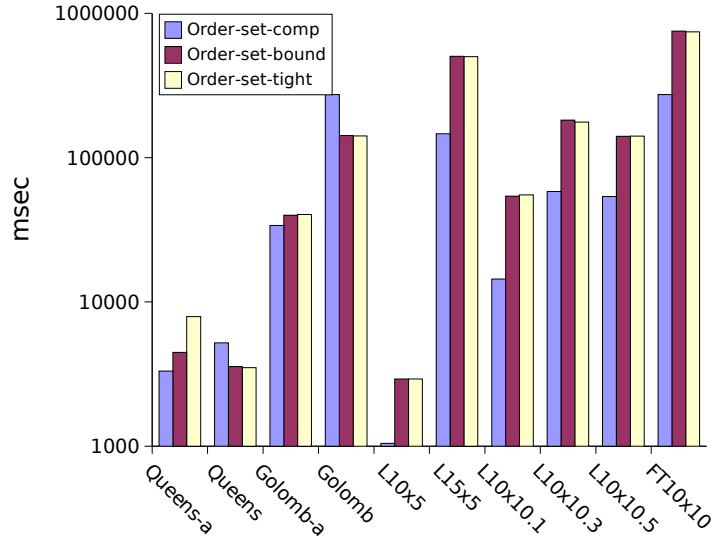


Fig. 2. Comparison of priority orders in **Order** buffer.

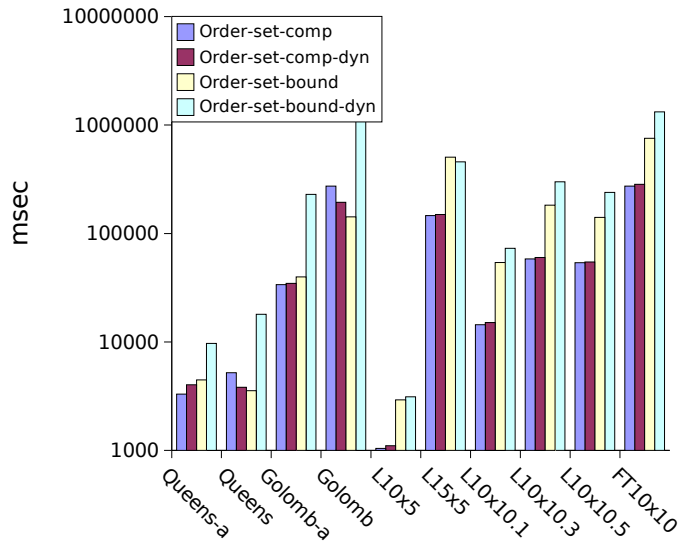


Fig. 3. Comparison of dynamic and static prioritization.

5 Conclusion

We have shown that the execution order of propagators in event-based propagation can be varied while retaining correct propagation results. Furthermore we showed that the re-insertion of idempotent propagators may be safely omitted when they are already buffered for execution. These theoretical results allow to safely implement arbitrary buffers for propagator scheduling.

We implemented such buffers which adapt the execution order of propagators dynamically during runtime in a standard constraint solver. Doing this, we extended the state of the art with two new scheduling techniques: fair-scheduling and impact-oriented prioritization. Ensuring fairness during propagator scheduling does not show any positive effect in our experiments so far. Despite the low computational complexity of our algorithm we could never achieve a runtime improvement. Impact-oriented ordering seems to depend a lot on the used notion of “impact”. Whenever good estimations of the impact are available, the impact-oriented scheduling is better than cost-oriented propagator scheduling. Preventing the multiple storage of propagators seems to be a good way to speed up constraint solving in all our experiments. Computing the priority values dynamically as opposed to just once upon the creation of a constraint does not pay in general. The computation of priority values of propagators should preferably only be executed once.

6 Future Work

In future work we plan to make many more and larger scale experiments and to evaluate more types of prioritization. These include combinations of the heuristics presented here, such as a quotient of cost over impact or the application of different methods in differing contexts (staged propagators [10, 11] or constraint specific metrics such as cost-oriented for global and impact-oriented for primitive constraints). Furthermore we plan to follow up research on the impact of fairness during propagator scheduling. We think results from CPU scheduling can be exploited much more in order to find near to optimal orderings of propagators. Finally we plan to apply learning techniques during scheduling. Propagators that have had a large impact in earlier executions may want to be preferably executed (or not, because their potential is already exhausted). We hope that with all the results gained in these experiments we will be able to find a generally efficient propagator scheduling heuristic which can be chosen as a standard for constraint solvers.

References

1. Krzysztof R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1998.
2. C.J. Beck, P. Prosser, and R.J. Wallace. Trying again to fail-first. In *Proc. ERCIM/CologNet workshop*, 2004.

3. Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming*, 1996.
4. P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. *SIGPLAN Notices*, 12(8):1–12, 1977. ACM Symposium on Artificial Intelligence and Programming Languages.
5. F. Fages, J. Fowler, and T. Sola. Experiments in reactive constraint logic programming. *Journal of Logic Programming*, 37:1-3:185–212, Oct-Dec 1998.
6. Thom Frühwirth. Constraint handling rules. *Constraint Programming: Basics and Trends, LNCS 910*, 1995.
7. M. Hoche, H. Müller, H. Schlenker, and A. Wolf. firstcs – a pure java constraint programming engine. In *Proc. 2nd International Workshop on Multiparadigm Constraint Programming Languages MultiCPL'03*, 2003.
8. Matthias Hoche. Analyse und entwicklung unterschiedlicher scheduling-verfahren zur laufzeit-optimierung der propagation eines java-basierten constraint-lösers. Master's thesis, TU Berlin, January 2004.
9. A. Legtchenko, A. Lallouet, and A. Ed-Dbali. Intermediate consistencies by delaying expensive propagators. In *Proc. FLAIRS 04*, 2004.
10. Georg Ringwelski. *Asynchrone Constraintlösen*. PhD thesis, Technical University Berlin, 2003.
11. C. Schulte and P. Stuckey. Speeding up constraint propagation. In *Proc. CP04*, September 2004.
12. R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proc. 9th Canadian Conf. on AI*, pages 163–169, 1992.