Maintaining State in Propagation Solvers

Raphael M. Reischuk[1], Christian Schulte[2], Peter J. Stuckey[3], and Guido Tack[4]

[1] IS&C, Saarland University, Saarbrücken, Germany, reischuk@cs.uni-sb.de
[2] KTH - Royal Institute of Technology, Sweden, cschulte@kth.se
[3] National ICT Australia, Victoria Laboratory, Department of Computer Science and Software Engineering, University of Melbourne, Australia, pjs@cs.mu.oz.au
[4] PS Lab, Saarland University, Saarbrücken, Germany, tack@ps.uni-sb.de

Abstract. Constraint propagation solvers interleave propagation, removing impossible values from variable domains, with search. The solver state is modified during propagation. But search requires the solver to return to a previous state. Hence a propagation solver must determine how to maintain state during propagation and forward and backward search. This paper sets out the possible ways in which a propagation solver can choose to maintain state, and the restrictions that such choices place on the resulting system. Experiments illustrate the result of various choices for the three principle state components of a solver: variables, propagators, and dependencies between them. This paper also provides the first realistic comparison of trailing versus copying for state restoration.

1 Introduction

Constraint propagation solvers interleave propagation (removing impossible values from variable domains) with complete tree search. During propagation and search, these solvers modify their state. Backtracking during search requires the solvers to recover a state that is equivalent to a previous node in the search tree.

The state of a propagation solver, often called the *constraint store*, principally represents three different kinds of information. (1) The information about what *variables* are involved in the problem and what possible values they can take (their domains); (2) the *propagators* implementing the constraints of the problem; and (3) *dependencies* between variables and propagators: which propagators should be (re-)executed on changes to each variable domain.

Each of these kinds of information can change during search. (1) Variables' domain information changes (the key state change made by a propagation solver). New variables may be added by the search (this is uncommon) or by the decomposition of existing propagators. Existing variables may become irrelevant if they are involved in redundant constraints only and are not of interest in the final solution. (2) Propagators can become propagation redundant (they cannot have further effect) and can hence be deleted. New propagators can be added by the search. Propagators can replace themselves with more efficient versions (or even split into multiple propagators) as execution proceeds. (3) Dependencies between variables and propagators can change because a variable becomes irrelevant to a propagator (when the variable is fixed, for example). Dependencies may change as execution proceeds, as in watched literals for Boolean clauses.

There are essentially two ways a propagation solver can store and access state information:

globally: all search nodes access the same global state; and
locally: each search node has a local, independent copy of the state.

Given the way in which state information is stored, there may be restrictions on its use. For global state information we have essentially three options:

static: this state may never be modified during the computation.
backtrack-safe: all changes made to the state are such that the state remains equivalent to all previous states on the path from the root node to the current node. The canonical example for this is watched literals for clausal propagators [12]. Although the literals that are watched are modified as search proceeds forward, the resulting state is correct for all previous states.
trailed: when backtracking to a previous node on the path from the root node, any changes to the state between the current and the previous node are undone. The mechanism for storing the undo information is called a *trail*.

For local state, there is only one option. The solver can make arbitrary changes, as they only affect a single search node. Local state is implemented by combining *copying* and *recomputation*: for some nodes in the search tree, a copy of the state of that node is stored; when backtracking, the closest copy on the path from the root is searched, copied, and the remaining search steps to the target node are redone. Finally a fixpoint is computed for the target node.

Global and local storage approaches differ in how optimistic they are. Global trailed state is *optimistic* in that it assumes that only a small part of the state will need restoration upon backtracking; local (copied) state is *pessimistic* in this sense. Both are pessimistic when it comes to search, as they assume that eventually, the solver will have to backtrack. In this sense, recomputation for local state is optimistic, as failure requires recomputation, which may be more expensive than just untrailing the changes or restarting from a copy.

Due to the different nature of the two approaches, each has advantages over the other. Global state using trailing (and other mechanisms where applicable) offers several advantages over local state. First, less copying is required, since the part that has changed can be substantially smaller than the entire state. Second, backtracking is potentially cheaper than when using recomputation. Finally, it is easier to share information (for example objective function values) between search nodes, because information can be global. On the other hand, local state which is copied and recomputed has the advantages of:

– Simplicity of propagator implementation: each propagator has its own local state information that it can arbitrarily adjust;
– Forward simplification, removing useless parts of the state such as propagation-redundant propagators (trailing systems must trail such removals);
– Tuning of required memory by adjusting recomputation parameters;

– Easy parallel search: each thread can work on a local copy independent of the rest of the computation;
– Straightforward support for best-first search: moving to a different part of the search tree just means moving to a different copy. A solver based on global trailed state can only jump between nodes using recomputation [13].

Contributions. In this paper we explore the advantages and disadvantages of different state restoration strategies, we show the interdependence of choices for state restoration within a CP system, and we present experiments that measure the performance for various choices in state restoration.

We give an architecture for a fully hybrid system that uses trailing for propagators and recomputation for domains. The paper is the first to identify and analyze the interdependence of important design decisions with respect to different state restoration strategies.

We provide the first comparison of trailing and recomputation based on a production-quality system, Gecode [18]. Both trailing and copying use the same propagation loop, search control, and (as far as possible) propagators and variable data structures, resulting in meaningful results and a fair comparison.

2 Propagation Solvers

This section reviews the basic design of propagation solvers, and which stateful entities they consist of.

The principle components of a constraint programming problem are: a set of variables $\mathcal{V}$ each with a set of possible values, its domain $D(v), v \in \mathcal{V}$ and a set of constraints C on variables $\mathcal{V}$. A constraint c is implemented by a propagator f which in the abstract is a function from domains to domains, e.g. $f(D) = D'$, removing values from the domains of the variables in c that cannot take part in any solution of c that is possible in the domain D.

In practice, attached to each variable $v \in \mathcal{V}$ is information about its current domain $D(v)$. In addition, for each event changing the variable's domain, a *dependency* list of propagators f (to be woken on this event) is also attached. The usual set of events (for integer variables) consists of: *fix*, the variable becomes fixed; *bnd*, the lower or upper bound of the variable changes; and *dom*, some other change to the domain is made. Each propagator f stores at least the variables $vars(c)$ for the constraint c that it implements, as well as internal state relevant to its implementation.

Search. The propagation solver interleaves propagation with search. A *search tree* is constructed and explored incrementally. Each *node* n in the search tree has an attached state S_n.

Search begins from a *root node* ϵ whose state reflects the original problem after the propagation loop has executed (the state of a node always represents a fixpoint of the propagation loop). Given a *current node* n we determine a set of choice constraints $\mathrm{c}_1, \ldots, \mathrm{c}_m$ whose disjunction is a consequence of the current state (for example $v = d \ \vee \ v \neq d$). The choice defines unexplored *child nodes* $n_i = n.\mathrm{c}_i$. To visit node n_i we add propagators representing c_i to the current state S_n and execute the propagation loop to arrive at state S_{n_i}.

If the propagation loop detects failure on the current node n then we choose to visit an unexplored *target node* n' in the search tree. We construct a correct starting state (before propagation) for the target node by copying, recomputing and/or untrailing and then running the propagation loop.

Propagation loop. The propagation loop is the "inner loop" of the propagation solver where it spends the bulk of its computation time. The propagation loop, given a domain D and set of propagators F, computes D' as the greatest fixpoint of all $f \in F$ less than D. The loop starts with an initial queue of propagators $F' \subseteq F$ not guaranteed to be at fixpoint with D. The first propagator from the queue is removed and executed, possibly changing the domains of its variables. These changes create events like *fix*, *bnd* and *dom* which access the dependencies to place more propagators in the queue. Once the propagator has completed, the next propagator from the queue is selected, until the queue is empty. For a complete discussion of the propagation loop, see Schulte and Stuckey [17].

Propagation solver state. There are essentially five kinds of state in a propagation solver: *variable* names; *domain* information for variables (and other variable state); *propagator* names; *internal propagator state*; and *dependencies* between variables and propagators. We separate the names of variables and propagators from their attached state since we may want to use different storage arrangements for each. Note that the propagator queue is always empty at choicepoints, and hence is not part of the state.

Figure 1 shows the access patterns of information in the propagation loop. An arrow $s \rightarrow e$ represents that from data s you can access data e. The importance of this diagram is that it captures interdependencies in choices of storage mechanisms. For an arrow $s \rightarrow e$ if s is local and e is global then this is easily handled since the local object can refer to global names. But if s is global and e is local then we need a way to find which local copy we are referring to, implying a global name *en* for the local e objects and a map $\mathit{map}(\mathit{en}, \mathit{node}) \rightarrow e$. Typically the map is implemented by having the global name be an index in an array or a

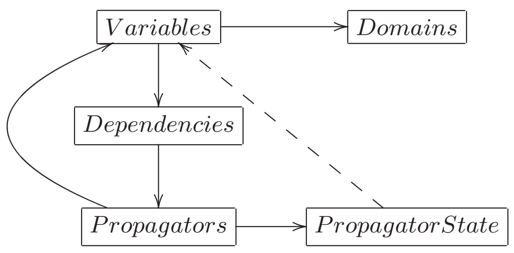

$$\begin{tikzcd}
\boxed{Variables} \arrow[r] \arrow[d] & \boxed{Domains} \\
\boxed{Dependencies} \arrow[d] & \\
\boxed{Propagators} \arrow[r] \arrow[uu, bend left=60] & \boxed{PropagatorState} \arrow[uul, dashed]
\end{tikzcd}$$

Fig. 1. Diagram of access patterns in a propagation solver.

memory offset, and having each node use the same array/memory block. This of course prevents shrinking the array/memory block in the local state as execution proceeds. Also note that many times propagator state does not refer directly to variables but uses the local names in the propagator (hence the dashed arrow).

Systems. Current constraint programming systems can be roughly categorized as either global state or local state systems, depending on whether their state restoration is predominantly based on trailing or copying with recomputation.

– Constraint solvers based on Prolog such as ECL^{i}PS^e [2] or SICStus [3] traditionally use global state, and state is recovered by trailing. The same is true for several constraint solvers based on object-oriented programming languages, such as ILOG Solver [9], CHOCO [4], or JaCoP [10].
– Gecode [6, 18] is based on local state, using copying and recomputation.
– Minion [7] is a mainly local state system. Variables and propagators are global static. Dependencies can change in a backtrack safe manner. Most domains are stored locally; for Booleans the domain representation is split: half local and half global in a backtrack safe manner; for one kind of integer variables, the domain is kept in a trailed state. Propagator state may be backtrack-safe or local. In addition to these features described in [7, 8], a look at the source code reveals that recent versions of Minion use trailed state for the internal state of some propagators.
– Figaro [5] allows various choices for state representation. Variables and propagators are global, but domains and propagator state can be: local and copied; lazily copied, i.e., they point to a previous copy until modified; or coarsely trailed, i.e., they trail only the first change to a variable or propagator by trailing the complete old value. Dependencies appear to be static.

3 A Hybrid System

This section briefly reviews three techniques for restoring solver state: trailing, copying, and recomputation. It then presents an architecture for a hybrid system that combines the different approaches.

Trailing. Global state that is neither static nor backtrack-safe can be restored upon backtracking using a *trail* of undo information. In its simplest form, before updating information at address a, a pair of a and the current value v at a is pushed on a stack. When backtracking, the solver restores the memory location at address a to v. This is shown in Fig. 2 (a) and (b): when the solver backtracks from the failed (square) node, it undoes the three trail entries, adds the choice constraint c_5, and computes a fixpoint for the dashed target node.

In principle, this technique can be used to store the necessary undo information for arbitrary state, but it can become inefficient when large parts of the state need to be trailed. To overcome these inefficiencies, *multi-value trailing* allows one to trail an address a and a *vector* of values at consecutive memory cells starting at a, and *time stamping* remembers whether an address a has already been trailed for the current search node and hence does not need trailing

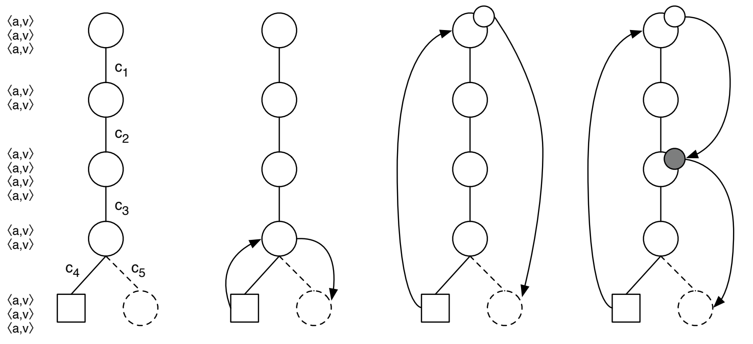


Fig. 2. (a) choices and trail, (b) untrailing, (c) fixed, and (d) adaptive recomputation

again [1]. *Function trailing*, or *semantic trailing*, puts the address of a function and an argument for the function on the trail. Untrailing then simply executes the function, which will cause the undo.

Copying and recomputation. The basis for restoring local state upon backtracking is a *copy* of the state. In the simplest form, a copy is made for each node, this is called *full copying.* To reduce the amount of required memory, solvers based on *recomputation* only store copies at some nodes in the tree. When moving to visit a new target node, the state for the node is computed from the nearest copy above the target node. In Fig. 2 (c), the nearest copy above the target node is at the root (marked as a small circle). When moving from the failed node to the unexplored dashed node, the solver takes the copy, copies it again (it is still needed for later backtracking), adds propagators for the choice constraints c_1, c_2, c_3 and c_5 on the path from the copied node to the target node and computes a fixpoint for the target node. Note this single fixpoint computation is much faster than computing the fixpoint for each intervening node (a technique called *batch recomputation* [5] or *decomposition-based search* [11]).

Schulte [16] describes two strategies for placing the copies in the tree: *fixed* recomputation places a copy every k nodes (k is called the *recomputation distance*); *adaptive* recomputation places a copy in the middle of the path between the current node and the nearest copy (as sketched in Fig. 2 (d)). Here we reconstruct the state of the middle node from a copy of the root node with propagators for c_1 and c_2 added and a fixpoint computed, then store a copy of this state (the grey small circle), before adding c_3 and c_5 and computing a fixpoint to construct the state for the target node.

Combining trailing and recomputation. Combining trailing and copying with recomputation in a single solver is a nontrivial task. This section presents such a hybrid architecture. The hybrid system has both local state and global state. This results in the following issues.

In order to use the same propagation loop, local and global propagators must be able to reside in the same queues. The queues are realized as local state (to support multi-threaded computation). In Gecode, propagators are elements of a doubly-linked list: either they are in the queue, or they are in a list of idle propagators. Global propagators must therefore be removed from the queues when the memory for the local state is deallocated (for example upon failure).

Variables need dependencies to both local and global propagators. The latter should be global themselves, in order to save the copying cost (which is linear in the number of propagators). Similarly the dependencies for local propagators should be local, too; otherwise we require a secondary *map* lookup from the dependency to the propagator. The currently implemented hybrid system therefore does not support global variables with dependencies to local propagators. In summary, dependencies should always be of the same kind of state (global/local) as the propagators they refer to.

Propagators can refer to both global and local variables. Because of the restriction mentioned above, the implemented hybrid system does not support local propagators with references to global variables.

The crucial interaction between the different kinds of state is on backtracking. When moving to a new target node the solver must untrail to the closest copy *above the common ancestor* of the current and target node (instead of just the closest common ancestor as would be usual in pure trailing systems). This is the nearest place where the solver has a consistent view of the overall state.

When performing recomputation (as in Fig. 2 (c)), a single fixpoint is computed for the target node. It is important to note that this implies that the trail entries for all modifications between the copy and the target are now regenerated, but in a different order than during the original exploration. In particular, the sets of trail entries that corresponded to the individual fixpoints during the original exploration may now be arbitrarily mixed. The trail thus cannot recover any state between the copy and the target. But this is not necessary, as recomputation always starts from a copy anyway. Adaptive recomputation (as in Fig. 2 (d)) works exactly like fixed recomputation, except that two fixpoints are computed, one for the copy in the middle of the path, and one for the target.

We have extended Gecode to support both global and local state for variable domains, propagators and dependencies, all coexisting. For the experiments we restrict ourself to one form of hybrid where propagators can be global or local but domains are always local, and to the fully global system. There are other important hybrids to consider. The extended system also supports a hybrid with fully global state but where we can copy the global state at any node in the search tree. This is an important feature for supporting advanced search strategies such as best-first and parallel search in a global state system [13].

4 Local versus Global State

This section presents and evaluates the possible choices of state representation for variables, propagators, and dependencies.

Experimental platform. The experiments in this paper use three different propagation solvers: one based on fully local state, one based on fully global state, and a hybrid using both global and local state. The local state system is Gecode version 3.0.2. For the global state system, we added a trail to the Gecode kernel and reimplemented Boolean and integer variables and several propagators to use backtrack-safe and trailed global state. The hybrid system is based on local variables, but supports both local and global (backtrack-safe and trailed) propagator state and dependencies.

The models are taken from the standard Gecode distribution and were compiled using the `gcc` compiler, version 4.2.1, on a Pentium 4 machine at 2.8 GHz running Linux. All runtime results are given as the arithmetic mean of 20 runs, with a coefficient of deviation less than 2%. Memory measurements represent the peak amount of the system's *overall allocated* memory.

Benchmarking rationale. All three systems are based on the same core solver. They share the same propagation loop, the same propagator scheduling mechanisms, the same data representation for variable domains and dependencies, the same search engines, and (up to the state representation) the same propagation algorithms. That way, the experiments indeed capture solely the difference in state representation. The starting point for the implementation was a copying solver, and some design decisions still reflect that, so there may be potential for optimization. However, the results show that both the global state and the hybrid solver are competitive with Gecode, and previous benchmarks show that Gecode is one of the fastest solvers available. Furthermore, we will see that we can explain the behavior of all three solvers in terms of properties of the *models*. The results are therefore largely independent of concrete implementation details.

4.1 Variable Domains

The first suite of experiments investigates the performance impact of the choice of state representation for the variable domains. The experiments compare the fully global state system with the hybrid system in which all state is global except local variable domains.

Runtime efficiency. Table 1 shows the experimental results. The Knights example has been included for statistics only, the required propagators have not been implemented for the hybrid system. Below the horizontal line are SAT benchmarks. We make the following observations:

- When the average percentage of domains modified in a fixpoint is high ($> 40\%$), copying domains is slightly preferable.
- When the percentage of updated domains is low (and there are many variables) then the optimistic approach of using global trailed domains benefits, and it can be much better than copying.
- The overall results show that trailing versus copying and recomputation of variable domains does not make that much difference except in the extreme cases, illustrated by the Radiotherapy benchmark.

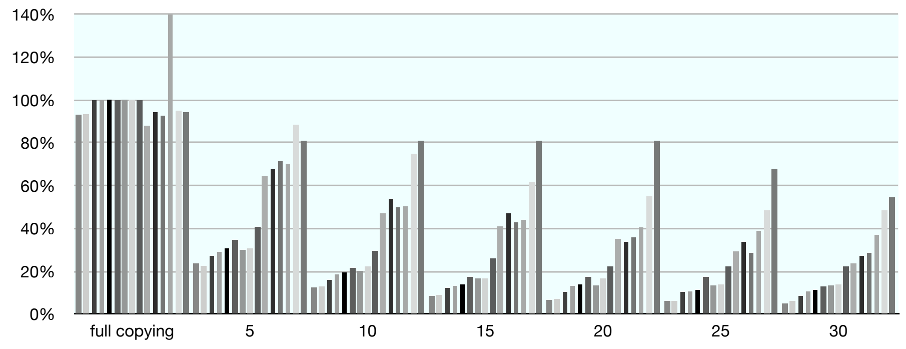
Table 1. Local versus global variable domains. *Var.* is the number of variables in the problem, *Prop.* the number of propagators, *mod %* the average percentage of variables modified per fixpoint.

Benchmark	*Var.*	*Prop.*	*Failures*	*mod %*	*Hybrid* *time (ms)*	*Global* *time %*
Queens (10)	10	3	4992	42.35	31.15	118.46
Queens (100)	100	3	22	32.61	4.40	41.20
Queens (200)	200	3	146 838	3.87	3 033.00	52.51
Golomb Rulers (10)	46	46	173 568	57.27	5 398.50	101.76
Golomb Rulers (bounds, 10)	46	46	30 345	64.10	1 642.00	105.83
Golomb Rulers (bounds, 11)	56	56	689 749	63.05	48 818.00	108.33
Alpha (Z–A)	26	21	10 530 183	42.98	243 556.50	100.00
Alpha (A–Z)	26	21	7 435	30.09	93.15	93.96
Radiotherapy	4 363	2 477	903 982	0.52	360 089.00	4.20
Magic Square (7)	49	19	481 301	19.94	6 075.00	85.37
Sudoku (1)	256	48	9721	8.87	1 007.00	90.07
Sudoku (2)	256	48	14277	8.02	1 281.00	91.76
Failure Stress (500)	2	1 000	1	100.00	3 173.30	171.81
Knights (30)	22188	26880	40	4.25	–	–
Knights (32)	25392	30780	34	4.27	–	–
Knights (34)	28812	34944	20	3.54	–	–
Ramsey (4-4-11)	55	660	211 605	6.65	6 330.00	90.66
Ramsey (4-4-12)	66	990	1 834 459	5.53	56 519.50	85.13
Hanoi (4)	718	4 934	888 424	2.66	95 021.50	58.28
Pigeon Holes (7)	56	204	32 781	10.30	441.00	80.95
Pigeon Holes (8)	72	297	378 344	8.46	6 107.00	69.17
Pigeon Holes (9)	90	415	4 912 515	7.16	88 390.00	77.21
Dubois (20)	60	160	3 145 728	8.05	30 880.00	67.26
Flat (200-1)	600	2 237	167 618	8.45	20 769.00	69.83

Time-stamped trailing. The example Failure Stress exhibits the worst case behavior for trailing. It consists of 1000 propagators for $x < y$ and $y < x$, shrinking the variable domains step by step until detecting failure. Not shown in this table, the memory consumption due to trailing is several orders of magnitude higher than in the local state solver. An implementation therefore must use time-stamped trailing to protect against this pathological behavior, but our experimental system does not use time stamping (to simplify the implementation). However, for all examples we consider for benchmarking here except Failure Stress, each variable is changed at most 2.7 times on average per fixpoint; so the comparisons are fair despite the lack of time-stamped trailing.

Memory consumption. One advantage of local state is the finer control of memory consumption using recomputation. We can confirm the results presented in [15] for the case where only variable domains are copied or trailed, and the remaining state is global. Table 2 compares the peak memory consumption of the

Table 2. Percentage of peak memory allocated at different recomputation distances, compared to trailing = 100%. Diagram cut off at 140%. Examples (left to right): Queens (200), Queens (100), Flat (200-1), Pigeon Holes (9), Ramsey (4-4-12), Pigeon Holes (7), Pigeon Holes (8), Ramsey (4-4-11), Hanoi (4), Golomb Rulers (bounds, 11), Golomb Rulers (10), Golomb Rulers (bounds, 10), Radiotherapy, Alpha (Z–A), Alpha (A–Z).



global state solver with that of the hybrid solver using different recomputation distances. (The cut off example is Radiotherapy at 158%.) Clearly the hybrid solver is advantageous over the trailing solver in memory usage, with median usage of 20% at recomputation distance 10, and 13% at distance 25.

4.2 Dependencies

Dependencies between variables and propagators are a critical part of the propagation solver, as they are accessed frequently during propagation. The key observation about dependencies was made earlier: dependencies should be the same kind of state (global/local) as the propagators they refer to.

Global dependencies. Static dependencies are sufficient for many small constraints, in particular those that are only propagation redundant when all variables are fixed. Backtrack-safe dependencies are sufficient for watched literals (and beneficial for Boolean clauses [12]). It is difficult to compare backtrack-safe with static or dynamic dependencies, as this leads to different propagator scheduling. Table 3 compares watched literals versus dynamic dependencies (two per clause) that are backtracked, and static dependencies (n for a clause of length n). For some SAT instances, watched literals can reduce the runtime in the global state system drastically (left), while for others, there is hardly any difference (right). For some propagators, however, dynamic dependencies are crucial:

– Canceling dependencies is essential for $\neq$ propagators. Using only static dependencies for the Queens example with binary propagators, the number of propagation steps and the runtime are an order of magnitude higher than with dynamic dependencies, independent of which system is used.

Table 3. Relative performance of backtracked and static dependencies for Boolean clauses compared to using watched literals, in the global state solver.

Benchmark	*Backtracked time %*	*Stateless time %*	*Benchmark*	*Backtracked time %*	*Stateless time %*
Ramsey (4-4-11)	160.57	220.86	Pigeon Holes (7)	91.88	87.54
Ramsey (4-4-12)	193.66	272.75	Pigeon Holes (8)	106.85	103.52
Hanoi (4)	104.23	119.68	Pigeon Holes (9)	100.30	94.05
			Dubois (20)	108.08	103.35
			Flat (200-1)	103.31	105.73

– Modifying dependencies (in a non-backtrack-safe way) is important for reified = propagators. Once the Boolean control variable is fixed to false, the propagator should be woken only for *fix* events. For the Knights examples, the global state system shows a 20% runtime overhead otherwise.

As dependencies are modified much less frequently than domains, the hybrid and global systems use function trailing for dynamic dependencies.

Local dependencies. Local state dependencies are automatically restored upon backtracking. This has the advantage that dynamic dependencies are free, but the disadvantage that watched literals, relying on backtrack-safe state, cannot be implemented directly.

4.3 Propagators and Propagator State

Propagators with internal state are an essential prerequisite for incremental propagation. In the following, the performance impact of the choice of state restoration for propagator state is analyzed.

Runtime efficiency. Table 4 shows the results of comparing the fully local solver with the hybrid solver (differing only in what kind of state the propagators use).

Many simple propagators (e.g. $x \leqslant y$, Boolean clauses) are stateless. The results illustrate that stateless propagators should not be copied. For the SAT examples which are dominated by stateless propagators, the hybrid system improves over the local system. The contrary results for Pigeon Holes are caused by substantially reduced propagation that has occurred (seemingly randomly) due to the use of dynamic dependencies rather than watched literals.

The importance of state in propagators is illustrated by naive `alldifferent`. This propagator has state which eliminates fixed variables from further consideration. If we do not record this state and simply try to remove the values of all fixed variables on each invocation the complexity of the propagator changes. Table 5 shows how a stateless `alldifferent` slows down the hybrid system.

Table 4. Relative runtime of the hybrid and global compared to the local system.

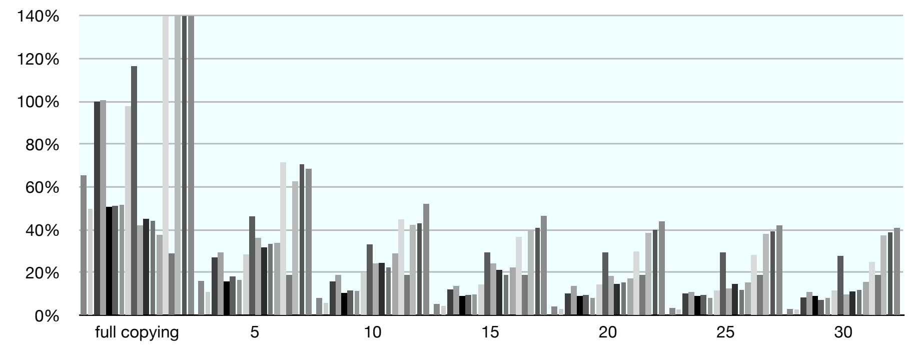
Benchmark	*Local*	*Hybrid*	*Global*
	time (ms)	*time %*	*time %*
Queens (10)	36.90	83.33	100.00
Queens (100)	1.50	293.67	120.67
Queens (200)	3 244.00	93.45	49.10
Golomb Rulers (10)	5 925.10	90.98	92.72
Golomb Rulers (bounds, 10)	1 657.75	110.59	104.83
Golomb Rulers (bounds, 11)	51 325.00	105.37	103.04
Alpha (Z–A)	275 562.50	87.68	88.39
Alpha (A–Z)	109.40	86.47	80.00
Radiotherapy	409 870.00	88.25	3.69
Magic Square (7)	6 593.00	92.45	78.66
Sudoku (1)	1 390.25	63.15	65.24
Sudoku (2)	1 743.50	93.36	67.42
Failure Stress (500)	3 173.30	100.00	171.81
Ramsey (4-4-11)	11 486.50	55.11	48.58
Ramsey (4-4-12)	138 305.50	40.87	36.21
Hanoi (4)	112 305.00	84.61	54.19
Pigeon Holes (7)	272.00	162.13	119.85
Pigeon Holes (8)	3 239.50	188.52	145.30
Pigeon Holes (9)	45 846.50	192.80	152.89
Dubois (20)	36 695.00	84.15	60.73
Flat (200-1)	25 205.00	82.40	61.42

Table 5. Relative runtime using a stateless `alldifferent` propagator.

Benchmark	*time %*	*Benchmark*	*time %*
Queens (10)	126.83	Golomb Rulers (10)	103.16
Queens (100)	509.93	Alpha (Z–A)	116.80
Queens (200)	1 129.65	Alpha (A–Z)	134.16

Memory consumption. The memory consumption at different recomputation distances of the local solver compared to the global solver is shown in Tab. 6. Some examples require significantly more memory in the local solver. (The cut off examples are Radiotherapy at 183% and Knights (30, 32, 34) at more than 300%.) This occurs because all propagators including their complete state must be copied, penalizing examples with many propagators. This is especially true for the added examples Knights (30, 32, 34), which use large numbers of reified propagators. On the other hand, some examples require significantly less memory in the local solver. This is due to the automatic garbage collection made possible

Table 6. Percentage of peak memory allocated at different recomputation distances, compared to trailing. The examples are the same as in Tab. 2, plus three instances of Knights (30,32,34) added on the right.



by copying: propagation redundant propagators and fixed auxiliary variables can be removed. A trailing solver could partly achieve the same effect by deleting the trail after computing the fixpoint of the root node. In summary, recomputation saves memory of the same order of magnitude as in the hybrid system, while sometimes starting from significantly higher amounts at full copying.

Recomputing propagator state. Some propagator state can be reconstructed from other state information. Its only purpose is to make the propagation more incremental. A well-known example of this is the variable-value graph used in domain consistent `alldifferent` [14]. It can be reconstructed from the domains of the variables. Both local and global propagators can use recomputed state. The advantage is that the state does not need to be trailed or copied. Each time the search moves to a non-child target node, this propagator state must be recomputed. This may seem expensive but the number of failures is much smaller than the number of executions of a propagator. Gecode use this approach for the variable-value graph for `alldifferent`, while JaCoP [10] (a trailing solver) also uses the approach for several propagators.

4.4 Summary

Trailing and copying with recomputation have been the dominant restoration techniques. Although there are several papers attempting comparative studies of the two techniques, this is the first time that a realistic comparison based on a production-quality solver is presented. Let us therefore summarize the results.

– Each of the solvers (local, hybrid, and global) is the best on some examples.
– While the trailing solver is generally faster than a copying solver, in almost all cases the difference between them is less than a factor of two.

– The trailing solver is more robust in terms of runtime than either the copying or hybrid solvers for problems with very weak propagation. Robustness for problems with strong propagation can be achieved by time-stamped trailing.
– Copying with recomputation is more robust than either trailing or full copying in terms of memory for any problem.

5 Related Work

There are a couple of papers that describe or compare different techniques for state management in constraint solvers. Schulte [15] introduced the copying and recomputation state maintenance model, and defined fixed and adaptive recomputation. The paper compared the Mozart local state solver versus a number of global state systems and showed that it was comparable or better than each of them in runtime. It also illustrated (but only by simulation of trailing) how the memory requirements of a local state system can be less than global (trailing) state systems when it is using fixed or adaptive recomputation. Here we reinforce the results on memory, but also show that trailing is more robust in runtime than copying and recomputation.

Choi et al. [5] compared different state maintenance approaches in the Figaro system. They compare copying versus lazy copying and coarse-grained trailing. In the system variables and propagators are global, while dependencies appear to be static. Lazy copying is a kind of copy-on-write technique, where there is a level of indirection added to each variable and propagator. Coarse grained trailing works similarly. There is one global *map* which is timestamped. Whenever an object is to be changed, the old value pointed to by the map is copied to the trail and the timestamp updated, and the object can be modified. Their results showed that coarse-grained trailing was faster than full copying which was itself faster than the recomputation approaches, and each of trailing and the recomputations usually gave significant savings in memory; while lazy copying improved on copying and coarse-grained trailing in terms of execution and memory. The results, while interesting, are for a research prototype solver that is several orders of magnitude slower than state of the art solvers.

6 Conclusion

In this paper we set out the possible ways in which a propagation solver can choose to maintain state, and the restrictions that such choices place on the resulting system. We describe how to combine global and local state maintenance in a single solver, and have extended Gecode to support both kinds of state. This allows us to give the first realistic comparison of trailing versus copying solvers, using a state of the art solver. Our results show that while the global state solver is in general faster than the copying and recomputation solver, and avoids some worst case behavior, it uses substantially more memory. As parallelism becomes more important, driven by multi-core CPUs, we foresee the importance of hybrid trailing and copying solvers to support this.

Acknowledgements. We thank Thibaut Feydy and Sebastian Brand for many interesting discussions related to this work. Raphael Reischuk and Guido Tack were supported by the Saarbrücken Graduate School of Computer Science which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

References

[1] A. Aggoun and N. Beldiceanu. Time Stamps Techniques for the Trailed Data in Constraint Logic Programming Systems. In *Actes du Séminaire 1990 de programmation en Logique*, pages 487–509. CNET, Lannion, France, May 1990.

[2] K. R. Apt and M. Wallace. *Constraint Logic Programming Using ECLiPSe*. Cambridge University Press, 2006.

[3] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. H. Hartel, and H. Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *LNCS*, pages 191–206, Southampton, UK, Sept. 1997. Springer.

[4] CHOCO, 2009. http://choco-solver.net.

[5] C. W. Choi, M. Henz, and K. B. Ng. Components for state restoration in tree search. In T. Walsh, editor, *CP*, volume 2239 of *LNCS*. Springer, 2001.

[6] Gecode: generic constraint development environment, 2009. www.gecode.org.

[7] I. P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, editors, *European Conference on Artificial Intelligence*, pages 98–102. IOS Press, 2006.

[8] I. P. Gent, C. Jefferson, and I. Miguel. Watched literals for constraint propagation in Minion. In F. Benhamou, editor, *CP*, volume 4204 of *LNCS*, pages 182–197. Springer, 2006.

[9] ILOG Solver, part of ILOG CP, 2009. http://www.ilog.com/products/cp.

[10] JaCoP, 2009. http://jacop.osolpro.com/.

[11] L. Michel and P. Van Hentenryck. A decomposition-based implementation of search strategies. *ACM Trans. Comput. Logic*, 5(2):351–383, 2004.

[12] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 530–535, New York, NY, USA, 2001. ACM Press.

[13] L. Perron. Search procedures and parallelism in constraint programming. In J. Jaffar, editor, *CP*, volume 1713 of *LNCS*, pages 346–360. Springer, 1999.

[14] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI '94: Proceedings of the Twelfth National Conference on Artificial intelligence (vol. 1)*, pages 362–367, Menlo Park, CA, USA, 1994.

[15] C. Schulte. Comparing trailing and copying for constraint programming. In D. D. Schreye, editor, *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 275–289, Las Cruces, NM, USA, 1999. MIT Press.

[16] C. Schulte. *Programming Constraint Services*, volume 2302 of *LNCS (LNAI)*. Springer, 2002.

[17] C. Schulte and P. J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, Dec. 2008.

[18] C. Schulte, G. Tack, and M. Z. Lagerkvist. Modeling with Gecode, 2009. www.gecode.org/doc-latest/modeling.pdf.