Comparing Trailing and Copying for Constraint Programming

Christian Schulte
Programming Systems Lab, Universität des Saarlandes
Postfach 15 11 50, 66041 Saarbrücken, Germany
schulte@ps.uni-sb.de

Abstract

A central service of a constraint programming system is search. In almost all constraint programming systems search is based on trailing, which is well understood and known to be efficient. This paper compares trailing to copying. Copying offers more expressiveness as required by parallel and concurrent systems. However, little is known how trailing compares to copying as it comes to implementation effort, runtime efficiency, and memory requirements. This paper discusses these issues.

Execution speed of a copying-based system is shown to be competitive with state-of-the-art trailing-based systems. For the first time, a detailed analysis and comparison with respect to memory usage is made. It is shown how recomputation decreases memory requirements which can be prohibitive for large problems with copying alone. The paper introduces an adaptive recomputation strategy that is shown to speedup search while keeping memory consumption low. It is demonstrated that copying with recomputation outperforms trailing on large problems with respect to both space and time.

1 Introduction

A central service in every constraint programming system is search. It demands that previous computation states must possibly be available at a later stage of computation. A system must take precaution by either memorizing states or by means to reconstruct them. States are memorized by *copying*. Techniques for reconstruction are *trailing* and *recomputation*. While recomputation computes everything from scratch, trailing records for each state-changing operation the information necessary to undo its effect.

Most current constraint programming systems are trailing-based. Many of them, for example CHIP [2], cc(FD) [13], Eclipse [3], and clp(FD) [5], are built on top of Prolog, which itself is trailing-based. But also systems that are not built on top of Prolog, like Screamer [12] (Lisp), and ILOG Solver [6] (C++) use trailing.

Copying offers advantages with respect to expressiveness: multiple nodes of a search tree are available simultaneously for further exploration. This is essential

Appears in: Danny De Schreye, editor, *Proceedings of the Sixteenth International Conference on Logic Programming*, Las Cruces, NM, USA, pages 275–289. The MIT Press, November 1999.

for concurrent, parallel, breadth-first, and user-defined search strategies. Implementation can be simpler, since copying is independent of operations and is only concerned with data structures.

On the other hand, copying needs more memory and might be slower since full copies of the computation states are created. Hence, it is not at all clear whether copying is competitive to trailing or not.

This paper shows that copying is competitive and that it offers a viable alternative to trailing for the implementation of constraint programming systems. The following points are discussed:

- The paper clarifies how much more memory copying needs. It is examined for which problems copying is competitive with respect to runtime and memory.
- For large problems with deep search trees the paper confirms that copying needs too much memory. It is shown that in these cases recomputation can decrease memory consumption considerably, even to a fraction of what is needed by trailing.
- It is shown that recomputation can also decrease runtime. The paper introduces adaptive recomputation that creates additional copies during search in order to speed up execution.

The paper uses Mozart [9], an implementation of Oz, as copying-based constraint programming system. The competitiveness of Mozart is stressed by comparing it to several trailing based constraint programming systems.

Plan of the Paper. The next section introduces some basic notions and concepts. Section 3 discusses the main implementation concepts. Section 4 introduces examples and criteria used for empirical evaluation. Section 5 gives an evaluation of copying followed by a comparison to trailing in Section 6. Recomputation is discussed in Section 7 followed by the introduction of adaptive recomputation in the next section. An empirical comparison of several constraint programming systems is given in Section 9.

2 Search for Constraint Programming

A *constraint problem* consists of a collection of *constraints* and a *distribution strategy* (also called labelling or enumeration strategy). The constraint problem defines a *search tree*. Its nodes represent computation states, whereas its arcs represent computation.

In the context of constraint programming, a computation state consists of a constraint store and propagators connected to the constraint store. The constraint store hosts basic (primitive) constraints. In the context of finite domain programming, for example, basic constraints are domain constraints like $x \in D$ where D is a finite domain. Propagators implement more complex constraints (for example, arithmetic

constraints or task-serialization for scheduling). Propagators amplify the store by adding new basic constraints (*constraint propagation*) to the constraint store.

Leafs in the search tree can be either failed (constraint propagation attempted to tell a constraint incompatible with the store) or solved (no propagators are left). Inner nodes are called *choices*. For a choice N, the distribution strategy defines how to compute the descendants N_i of the node N. The N_i are also called *alternatives* (of N). For example, in the context of finite domain programming the N_i are computed by telling a basic constraint B_i to N's constraint store that starts further constraint propagation.

The computational service offered by a constraint programming system is to *explore* the search tree of a given constraint problem. In the following we always assume left-most, depth-first exploration. The system must be prepared to follow several alternatives issuing from the same node N. This paper discusses the following three approaches:

Copying. An identical copy of N is created before N is changed.

Trailing. Changes to N are recorded such that they can be undone later.

Recomputation. If needed, N is recomputed from scratch. Discussion of recomputation is postponed to Section 7.

Expressiveness. The main difference as it comes to expressiveness is the number of nodes that are simultaneously available for further exploration. With copying, all nodes that are created as copies are directly ready for further exploration. With trailing, exploration can only continue at a single node at a time.

In principle, trailing does not exclude exploration of multiple nodes. However, they can be explored in an interleaved fashion only and switching between nodes is a costly operation. For this reason all current trailing-based constraint programming systems do not support node-switching.

Having more than a single node available for exploration is essential to search strategies like concurrent, parallel, or breadth-first. The same property is also crucial for user-defined interactive exploration of search trees as implemented by the Oz Explorer [10]. By making nodes of a search tree available as first-class entities (as it is done in Oz [11]), the user can directly profit from the increased expressiveness.

Resource model. Copying essentially differs from trailing with respect to space requirements in that it is *pessimistic*: while trailing records changes exactly, copying makes the safe but pessimistic assumption that everything will change. On the other hand, trailing needs to record information on what changes as well as the original state of what is changed. In the worst case — the entire state is changed — this might require more memory than copying. This discussion makes clear that a meaningful comparison of the space requirements for trailing and copying is only possible by empirical investigations, which are carried out in Section 6.

3 Implementation Issues

This section gives a short discussion of the main implementation concepts and their properties in copying- and trailing-based systems. The most fundamental distinction is that *trailing*-based systems are concerned with *operations* on data structures while *copying*-based systems are concerned with the *data structures* themselves.

Copying. Copying needs for each data structure a routine that creates a copy and also recursively copies contained data structures. A system that features a copying garbage collector already provides almost everything needed to implement copying. For example in the Mozart implementation of Oz, copying and garbage collection share the same routines parametrized by a flag that signals whether garbage collection is performed or whether a node is being copied.

By this all operations on data structures are independent of search with respect to both design and implementation. This makes search in a system an orthogonal issue. Development of the Mozart system has proven this point: it was first conceived and implemented without search and only later search has been added.

Trailing. A trailing-based system uses a trail to store undo information. Prior to performing a state-changing operation, information to reconstruct the state is stored on the trail. In a concrete implementation, the state changing operations considered are updates of memory locations. If a memory update is performed, the location's address and its old content is stored on the trail. To this kind of trail we refer to as single-value trail. Starting exploration from a node puts a mark on the trail. Undoing the trail restores all memory locations up to the previous mark. This is essentially the technology that is used in Warren's Abstract Machine [14, 4].

In the context of trailing-based constraint programming systems two further techniques come into play:

Time-stamping. With finite domains, for example, the domain of a variable can be narrowed multiply. However it is sufficient to trail only the original value, intermediate values need no restauration: each location needs to appear at most once on the trail. Otherwise memory consumption is no longer bounded by the number of changed locations but by the number of state-changing operations performed. To ensure this property, time-stamping is used: as soon as an entity is trailed, the entity is stamped to prevent it from further trailing until the stamp changes again. Note that time-stamping concerns both the operations and the data structures that must contain the time-stamp.

Multiple-value trail. A single-value trail needs $2n$ entries for n changed locations. A multiple value trail uses the optimization that if the contents of $n > 1$ successive locations are changed, $n+2$ entries are added to the trail: one for the first location's address, a second entry for n, and n entries for the locations' values. For a discussion of time-stamps and a multiple value trail in the context of the CHIP system, see [1, 2].

Example	Expl.	Choices	Fail.	Sol.	Depth	Var.	Constr.
Alpha	all	7435	7435	1	50	26	21
100-Queens	one	115	22	1	97	100	14850
100-S-Queens	one	115	22	1	97	100	3
10-Queens	all	6665	5942	724	29	10	135
10-S-Queens	all	6665	5942	724	29	10	3
Magic	one	13	4	1	12	500	501
18-Knights	one	266	12	1	265	7500	11205

Table 1: Characteristics of example programs.

A general but brief discussion of issues related to implementation issues for trailing-based constraint programming systems can be found in [7].

Trailing requires that all operations are search-aware: search is not an orthogonal issue to the rest of the system. Complexity in design and implementation is increased: it is a matter of fact that a larger part of a system is concerned with operations rather than with basic data structure management. A good design that encapsulates update operations will avoid most of the complexity. To take advantage of multiple value trail entries, however, operations require special effort in design and implementation.

Trailing for complicated data structures can become quite complex. Consider as an example adding an element to a dictionary with subsequent reorganization of the dictionary's hash table. Here the simple model that is based on trailing locations might be unsuited, since reorganizing data structures alters a large number of locations. In general, copying offers more freedom of rearranging data structures, for a discussion in the context of finite domain constraints see [8].

The discussion in this section can be summarized as follows. A system that features a copying garbage collector already supports the essential functionality for copying. For a system that does not require a garbage collector trailing might be as easy or possibly easier depending on the number and complexity of the operations.

4 Criteria and Examples

This section introduces constraint problems that serve as examples for the empirical analysis and comparison. The problems are well known, they are chosen to be easily portable to several constraint programming systems (see Section 9).

The main characteristics of the problems are listed in Table 1. Besides of portability and simplicity they cover a broad range with respect to the following criteria.

Problem size. The problems differ in size, that is in the number of variables and constraints, and in the size of constraints (that is the number of variables each constraint is attached to). With copying, the size of the problem is an important parameter: it determines the time needed for copying. It also partly determines the memory requirements (which is also influenced by the search tree depth). Hence, large problem sizes can be problematic with copying.

Amount of propagation. A problem with strong propagation narrows a large number of variables. This presupposes a large number of propagation steps, which usually coincides with state changes of a large number of constraints. The amount of propagation determines how much time and memory trailing requires: the stronger the propagation, the more of the state is changed. The more of the state changes, the better it fits the pessimistic assumption "everything changes" that underlies copying.

Search tree depth. The depth of the search tree determines partly the memory requirements for both trailing and copying. Deep search trees are a bad case for trailing and even more for copying due to its higher memory requirements.

Exploration completeness. How much of the search tree is explored. A high exploration completeness means that utilization of the precaution effort undertaken by copying or trailing is high.

The criteria are not independent. Of course, the amount of propagation determines the depth of the search tree. Also search tree depth and exploration completeness are interdependent: If the search tree is deep, exploration completeness will definitely be low: Due to the exponential number of nodes, the part of the tree that can be explored is relatively small.

All example problems are familiar benchmark problems. Alpha is the well-known cryptoarithmetic puzzle: assign variables A, B, $\ldots$, Z distinct numbers between 1 and 26 such that 25 equations hold. For the popular n-Queens puzzle (place n queens on a $n \times n$ chess board such that no two queens can attack each other) two different implementations are used. The naive implementation (called n-Queens) uses $O(n^2)$ disequality constraints. This is contrasted by a smarter program (which is called n-S-Queens accordingly) that uses three propagators for the same constraints: this leads to much better propagation in relation to the problem size. The two different encodings of the n-Queens puzzle are chosen to analyze the difference between many small propagators and few larger propagators.

The Magic puzzle is to find a magic sequence s of 500 natural numbers, such that $0 \leq x_i \leq 500$ and i occurs in s exactly x_i times. It uses for each element of the sequence an exactly-constraint (each ranging over all variables x_i) on all elements of the sequence. The elements are enumerated in increasing order following a splitting strategy. The goal in 18-Knights is to find a sequence of knight's moves on a 18×18 chessboard such that each field is visited exactly once and that the moves return the knight to the starting field, which is fixed to the lower left field.

The paper prefers familiar benchmark programs over more realistic problems such as scheduling or resource allocation. The reason is that the programs are also intended for comparing several constraint programming systems. Choosing simple constraints ensures that the amount of constraint propagation is the same with all compared systems.

Example	Time	Copy	GC	CGC	Max
	sec	%	%	%	KB
Alpha	7.80	20.8	3.5	24.3	19
10-Queens	3.49	30.8	3.7	34.5	20
10-S-Queens	2.54	18.4	2.7	21.1	7
100-Queens	2.96	51.3	16.6	67.9	21873
100-S-Queens	0.10	31.7	0.0	31.7	592
Magic	2.61	9.9	11.5	21.5	6091
18-Knights	23.53	36.1	31.5	67.6	121557

Table 2: Runtime and memory performance of example programs.

5 Copying

This section presents and analyses runtime and memory requirements for Mozart, a copying-based implementation of Oz [9]. For more information on hardware and software platforms see Appendix A.

Table 2 displays the performance of the example programs. The fields “Copy” and “GC” give the percentage of runtime that is spent on copying and garbage collection, the field “CGC” displays the sum of both fields. The field “Max” contains the maximal amount of memory used in Kilobytes, that is how much memory must at least be available in order to solve the problem.

The numbers clarify that for all but the large problems 100-Queens and 18-Knights the amount of time spent on copying and garbage collection is around one fourth of the total runtime. In addition, the memory requirements are moderate. This demonstrates that for problems with small and medium size copying does neither cause memory nor runtime problems. It can be expected that for these problems copying is competitive.

On the other hand, the numbers confirm that *copying alone for large problems with deep search trees is unsuited*: up to two third of the runtime is spent on memory management and memory requirements are prohibitive. The considerable time spent on garbage collection is also a consequence of copying: the time used by a copying garbage collector is determined by the amount of used memory.

The two different implementations of *n*-Queens exemplify that copying gets considerably better for problems where a large number of small propagators is replaced by a small number of equivalent global propagators.

6 Trailing

As discussed before, one of the most essential questions in comparing trailing and copying is: how pessimistic is the assumption “everything changes” that underlies copying. An answer seems to presuppose two systems that are identical with the exception of trailing or copying. Implementing two competitive systems is not feasible.

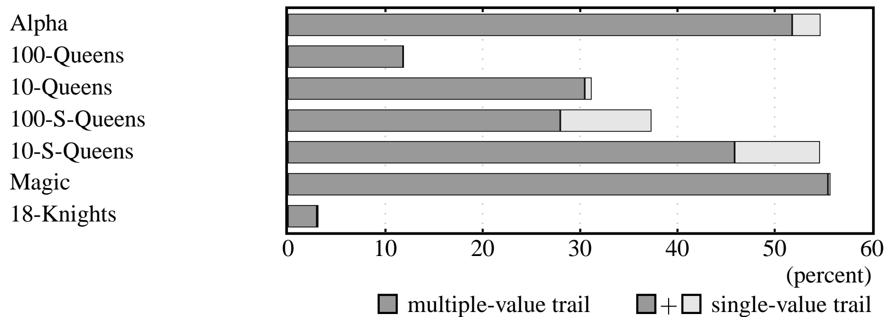

Figure 1: Memory use of trailing versus copying.

Instead, the memory requirements of a trailing implementation are computed from the requirements of a copying implementation as follows. Before constraint propagation in a node N begins, a bitwise copy of the memory area occupied by N is created. After constraint propagation has finished, this memory area is compared to the now changed memory area occupied by N. The altered locations are those that a trailing system must have trailed.

Figure 1 shows the percentage of memory needed by a trailing implementation compared to a copying implementation. The total length of bars depicts the percentage needed by a single-value trail, whereas the dark-colored bar represents the need of a multiple-value trail implementation.

The percentage figures for the multiple-value trail are lower bounds again. Locations that are updated by separate single update operations might happen to be successive even though an implementation cannot take advantage of this fact. It is interesting to note that a multiple-value trail offers some improvement only for 10-S-Queens and 100-S-Queens (around 10%). Otherwise, its impact is quite limited (less than 2%).

The observation that for large problems with weak propagation (100-Queens and 18-Knights) trailing improves by almost up to two orders of magnitude coincides with the observation made with respect to the memory requirements in Section 5. For the other problems the memory requirements are in the same order of magnitude and trailing roughly halves them.

What is not captured at all by the comparison's method is that other design decisions for propagators would have been made to take advantage of trailing, as has already been argued in Section 3.

7 Recomputation

Recomputation trades space for time, a node N is reconstructed on demand by redoing computations. The space requirements are obviously low: only the path in the search tree leading to N must be stored (for example, as a list of integers). In particular, the space requirements for recomputation are problem independent.

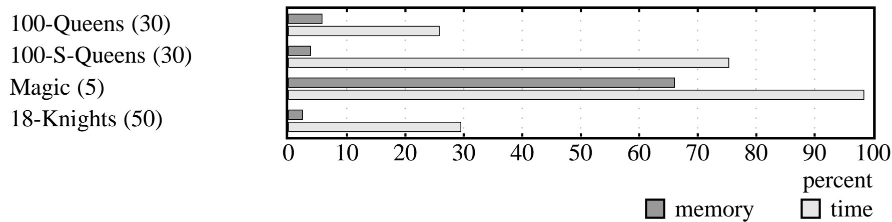

Figure 2: Runtime and memory gain with fixed recomputation.

Basing exploration on recomputation alone is infeasible. Suppose a complete binary search tree of height n, which has 2^n leafs. To recompute a single leaf, n exploration steps are needed. This gives a total of $n2^n$ exploration steps compared to $2^{n+1}-2$ exploration steps without recomputation (that is, the number of arcs). Thus recomputation alone takes approximately $n/2$-times the number of exploration steps both copying and trailing need.

The basic idea of combining recomputation with copying is as follows: copy a node from time to time during exploration. Recomputation then can start from the last copy N on the path to the root. Note that this requires to start from a copy of N rather than from N itself, since N might be needed for further recomputation. The implementation of recomputation is straightforward, see [11] for example.

A simple strategy for recomputation is *fixed recomputation*: limit the number of steps needed to recompute a node by some fixed number n, to which we refer as *MRD (maximal recomputation distance)*. That is, after n exploration steps a copy of the current node is memorized.

An important optimization is as follows: after all but one alternative A of a copied node N have been explored, further recomputation from N always starts with recomputing A. The optimization now is to do the recomputation step $N \rightarrow A$ only once. This optimization corresponds to the `trust_me` instruction in the WAM.

Fixed recomputation with a MRD of n guarantees that the number of stored nodes decreases by a factor of n. Figure 2 displays the improvements obtained by fixed recomputation, where the numbers in parentheses give the employed MRD.

The improvement in memory for 100-Queens and 18-Knights (the two problems for which Section 5 showed that copying entails prohibitive memory requirements) is by two orders of magnitude. 100-S-Queens enjoys the same memory-improvement as 100-Queens, since the search trees are identical. The figures for Magic exhibit that even for problems for which copying is perfectly adequate, memory consumption can be decreased without a runtime penalty.

Our motivation for recomputation was the urge to save memory. However, the numbers in Figure 2 exemplify that recomputation saves both memory and runtime. In particular, the time savings from less copying are larger than the time spent on recomputing.

Fixed recomputation uses less memory than trailing. Figure 3 shows the percentage of memory that fixed recomputation takes in comparison to the memory

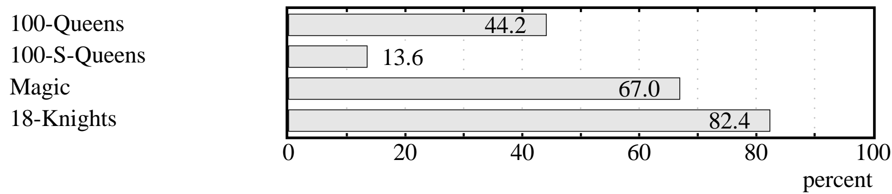

Figure 3: Memory use of fixed recomputation versus trailing.

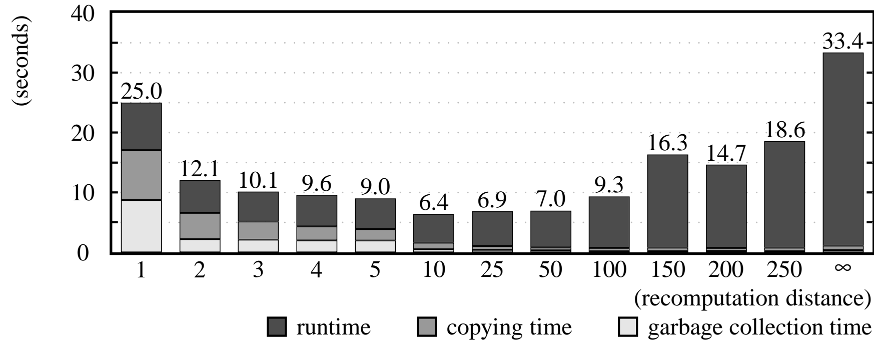

Figure 4: Runtime for 18-Knights with fixed recomputation.

needed by trailing.

Trailing and copying are pessimistic in that they make the assumption that each node needs reconstruction. Recomputation, in contrast, makes the optimistic assumption that no node requires later reconstruction. For search trees that contain few failed nodes, the optimistic assumption fits well. In particular, problems with very deep search trees can profit from the optimistic assumption, since exploration completeness will definitely be low (as argued in Section 4).

Figure 4 relates the runtime to different MRDs for the 18-Knights problem. For a MRD from 1 to 10 the runtime is strictly decreasing because the time spent on copying and garbage collection decreases, while the plain runtime remains constant. With further increase of MRD the runtime increases due to the increasing recomputation overhead.

Figure 4 shows a small peak at a MRD of 150. The search tree for 18-Knights has five failed nodes at a depth of around 260. This means that recomputation has to perform around 110 recomputation steps for each of the nodes. This phenomenon can be observed quite often: slight changes in the MRD (like from 100 to 150 for 18-Knights) results in unexpected runtime behavior. This indicates that for some parts of the search tree the assumption of recomputation is overly optimistic.

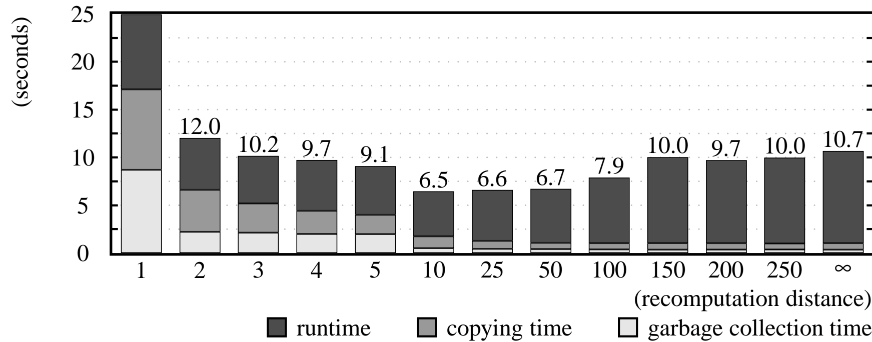

Figure 5: Runtime for 18-Knights with adaptive recomputation.

8 Adaptive Recomputation

In the last section we made the following two observations. Firstly, the optimistic assumption underlying recomputation can save time. Secondly, the fixed choice of a MRD can inhibit this.

If exploration exhibits a failed node it is quite likely that not only a single node is failed but that an entire subtree is failed. It is unlikely that only the last decision made in exploration was wrong. This suggests that as soon as a failed node occurs during exploration, the attitude for further exploration should become more pessimistic.

The following strategy is simple and shows remarkable effect. During recomputation of a node N_2 from a node N_1 an additional copy is created at the middle of the path from N_1 to N_2. To this strategy we refer to as *adaptive recomputation*.

Figure 5 shows the runtime for adaptive recomputation applied to 18-Knights. Not only the peak for a MRD of 150 disappears, also the runtime for large MRD values remains basically constant. Even if copies are created during recomputation only (that is the MRD is ∞) the runtime remains almost unaffected.

This is the real significance of adaptive recomputation: the choice of the recomputation distance is not as important as one would think. Provided that the distance is not too small (that is, no excessive memory consumption), adaptive recomputation adjusts quickly enough to achieve good performance.

While adaptive recomputation is a good strategy as it comes to runtime, it does not guarantee that memory consumption is decreased. In the worst case, adaptive recomputation does not improve over copying alone.

Figure 6 shows the active heap memory for both fixed and adaptive recomputation applied to 18-Knights. The numbers exhibit that avoidance of peaks in runtime is not paid by peaks in memory (for MRDs between 1 and 5 memory requirements for both fixed and adaptive recomputation are almost identical and thus are left out).

For deep search trees the following technique could help limit the required memory. As soon as exploration has reached a certain depth in the search tree, it is quite unlikely that nodes high above are going to be explored. Thus, copies remaining in the upper parts of the tree could be dropped. This would decrease

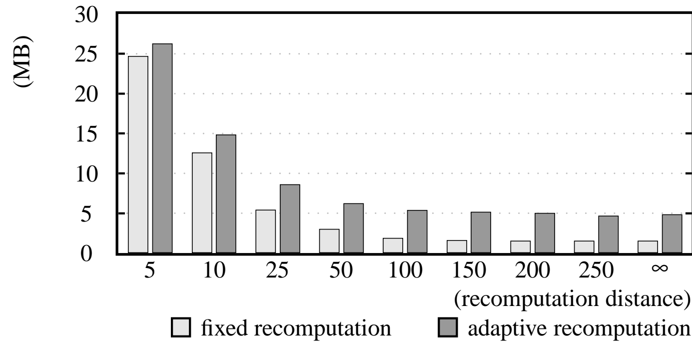

Figure 6: Memory requirements for 18-Knights.

memory consumption and would most likely not affect runtime.

9 Empirical Comparison

This section compares Mozart, a copying-based system, with several trailing-based systems. For more information on the used software and hardware platforms, see Appendix A. The point to compare systems in this paper is to demonstrate that a system that is based on copying can be competitive with trailing based systems.

The runtimes of course do not depend only on the systems' search capabilities, but also on their finite domain implementation. It has been tried to keep the examples' implementations for the different systems as similar as possible. In particular, even if a system provides special propagators for a particular example, the programs do not take advantage of them.

All systems support Alpha, 10-Queens, 100-Queens, and 18-Knights. The propagators that are used for the 10-S-Queens and 100-S-Queens formulation are available in Mozart and Solver only. Eclipse does not support the exactly-constraint that is used in Magic.

Figure 7 shows a relative performance comparison of Mozart with Eclipse, SICStus, and Solver. The figures to the left are without recomputation, the figures to the right use fixed recomputation (the same MRDs as in Figure 2 are used). A number of n below the middle line together with a light gray box means that Mozart performs f-times better. Otherwise, the other system performs f-times better than Mozart.

The figures clearly indicate that a system based on the copying approach is competitive as it comes to runtime. It is worth noting that even for problems that profit from recomputation performance is still competitive without recomputation. In general, this is of course only true if the available memory is sufficient.

The numbers for Mozart with recomputation show that copying together with recomputation for large problems and deep search trees can outperform trailing-based systems.

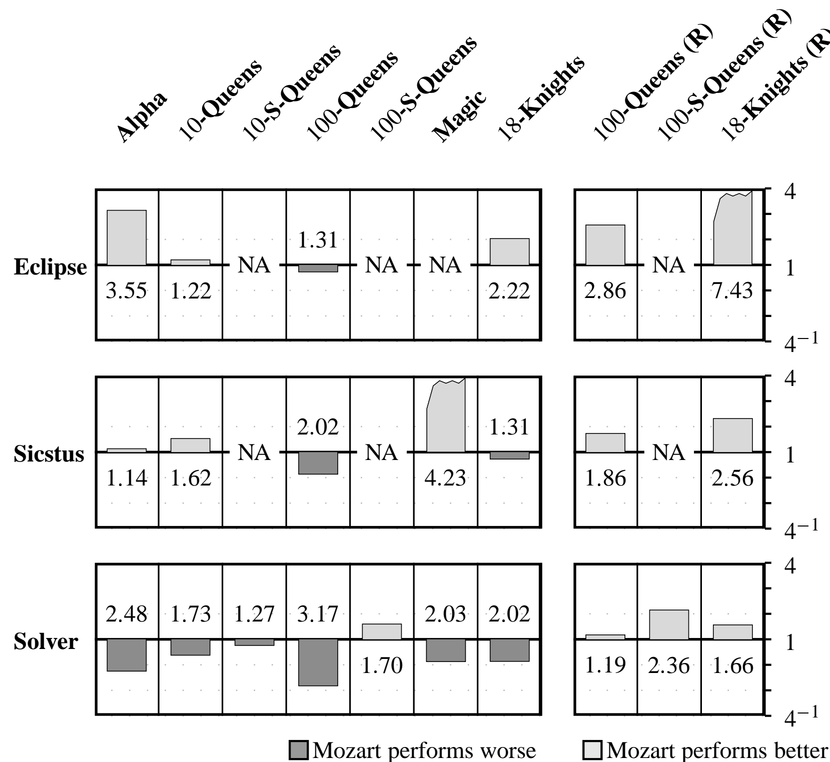

Figure 7: Empirical runtime comparison.

A Used Hardware and Software Platforms

All programs have been run on a single processor Sun ULTRASparc-1 170 with 300 Megabytes of main memory and using SunOS 5.5 as operating system. All runtimes have been taken as wall time (that is, absolute clock time), where the machine was unloaded: difference between wall and actual process time is less than 5%. All numbers presented are the arithmetic mean of 25 runs, where the coefficient of variation is less than 5% for all benchmarks and systems but 100-S-Queens for Solver, where the deviation was less than 15%.

The following systems were used: Mozart 1.1.0, Eclipse 3.7.1, SICStus Prolog 3.7.1, and ILOG Solver 4.400.

Acknowledgements. I am grateful to Thorsten Brunklaus, Katrin Erk, Leif Kornstaedt, Tobias Müller, Andreas Rossberg, and the anonymous referees for providing comments that helped to improve the paper.

References

[1] Abderrahamane Aggoun and Nicolas Beldiceanu. Time Stamps Techniques for the Trailed Data in Constraint Logic Programming Systems. In *Actes du Séminaire 1990–Programmation en Logique*, pages 487–509, Tregastel, France, May 1990. CNET.

[2] Abderrahamane Aggoun and Nicolas Beldiceanu. Overview of the CHIP Compiler System. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 421–437. The MIT Press, Cambridge, MA, USA, 1993.

[3] Abderrahamane Aggoun, David Chan, Pierre Dufresne, Eamon Falvey, Hugh Grant, Alexander Herold, Geoffrey Macartney, Micha Meier, David Miller, Shyam Mudambi, Bruno Perez, Emmanuel Van Rossum, Joachim Schimpf, Periklis Andreas Tsahageas, and Dominique Henry de Villeneuve. ECL^{i}PS^e 3.5. User manual, European Computer Industry Research Centre (ECRC), Munich, Germany, December 1995.

[4] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. Logic Programming Series. The MIT Press, Cambridge, MA, USA, 1991.

[5] Philippe Codognet and Daniel Diaz. Compiling constraints in `clp(FD)`. *The Journal of Logic Programming*, 27(3):185–226, June 1996.

[6] ILOG. ILOG Solver: Reference manual, May 1997. Version 4.0.

[7] Joxan Jaffar and Michael M. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–582, May 1994. Special Issue: Ten Years of Logic Programming.

[8] Tobias Müller and Jörg Würtz. Extending a concurrent constraint language by propagators. In Jan Małuszyński, editor, *Proceedings of the International Logic Programming Symposium*, pages 149–163, Long Island, NY, USA, 1997. The MIT Press.

[9] Mozart Consortium. The Mozart programming system, 1999. Available from `www.mozart-oz.org`.

[10] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press.

[11] Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 519–533, Schloß Hagenberg, Linz, Austria, October 1997. Springer-Verlag.

[12] Jeffrey Mark Siskind and David Allen McAllester. Screamer: A portable efficient implementation of nondeterministic Common Lisp. Technical Report IRCS-93-03, University of Pennsylvania, Institute for Research in Cognitive Science, 1993.

[13] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(FD). *The Journal of Logic Programming*, 37(1–3):139–164, October 1998.

[14] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Artificial Intelligence Center, Menlo Park, CA, USA, October 1983.