Components for State Restoration in Tree Search

Chiu Wo Choi[1], Martin Henz[1], Ka Boon Ng[2]

[1] School of Computing, National University Of Singapore, Singapore
{choichiu,henz}@comp.nus.edu.sg
[2] Honeywell Singapore Laboratory
kevin.ng@honeywell.com

Abstract. Constraint programming systems provide software architectures for the fruitful interaction of algorithms for constraint propagation, branching and exploration of search trees. Search requires the ability to restore the state of a constraint store. Today's systems use different state restoration policies. Upward restoration undoes changes using a trail, and downward restoration (recomputation) reinstalls information along a downward path in the search tree. In this paper, we present an architecture that isolates the state restoration policy as an orthogonal software component. Applications of the architecture include three novel state restoration policies, called lazy copying, coarse-grained trailing, and batch recomputation, a detailed comparison of these and existing restoration policies with "everything else being equal", and a novel class of engines that uses different restoration policies in different parts of the search tree. The architecture allows the user to optimize time and space consumption of applications by choosing existing or designing new state restoration policies in response to application-specific characteristics.

1 Introduction

Finite domain constraint programming (CP(FD)) systems are software systems designed for solving combinatorial search problems using tree search. The history of constraint programming systems shows an increasing emphasis on software design, reflecting user requirements for flexibility in performance debugging, and application-specific customization of the algorithms involved.

A search tree is generated by branching algorithms, which at each node provide different choices that add new constraints to strengthen the store in each child. Propagation algorithms strengthen the store according to the operational semantics of constraints in the store, and exploration algorithms decide on the order in which search trees are explored.

Logic programming proved to be successful in providing elegant means of defining branching algorithms, reusing the built-in notion of choice points. Constraint programming systems like SICStus Prolog [Int00] and GNU Prolog [DC00] provide libraries for propagation algorithms and allow the programming of exploration algorithms on top of the built-in depth-first search (DFS) by using meta programming. To achieve a more modular architecture, recent systems moved away from the logic programming paradigm. The ILOG Solver library for constraint programming [ILO00] allows us to implement propagation algorithms in C++ and exploration algorithms by using an object that encapsulates the state of search. The language Claire [CJL99] allows for programming exploration algorithms using built-in primitives for state manipulation, and the language Oz provides a built-in data structure called space [Sch97b,Sch00] for implementing exploration algorithms.

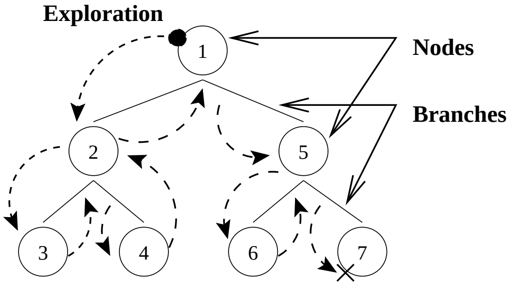
At every node in the search tree, the *state* of variables and constraints is the result of constraint propagation of the constraints that were added along the path from the root to the node. During search, the nodes are visited in the order given by the exploration algorithm. In this paper, we address the question on how the state corresponding to a node is obtained or *restored*. Different systems currently provide different ways of restoring the state corresponding to the target node. All systems/languages except Oz are based on a state restoration policy (SRP) that records changes on the state in a data structure called trail. The trail is employed to restore the state to an ancestor node of the target. Schulte [Sch97b,Sch00] presents a few alternatives based on copying and recomputation of states and evaluates its competitiveness conceptually and experimentally in [Sch99]. The best state restoration policy for a given application depends on the amount of propagation (state change), the exploration and the branching. The goal of this work is to identify software techniques that enable the employment of different SRPs in the same system without compromising the orthogonal development of other components such as propagation, branching and exploration. The architecture allows the user to optimize time and space consumption of applications by choosing existing or designing new SRPs in response to application-specific characteristics. We introduce three novel SRP, namely lazy copying, batch recomputation and coarse-grained trailing, and show that for many applications, the first two considerably improve the time and/or space efficiency over existing SRPs. State restoration is an important aspect of tree search that deserves the attention of users and constraint programming systems designers.

We outline in Section 2 a software architecture for constraint programming systems that will form the base for further discussion. The components are designed and implemented in C++ on the base of the Figaro library for constraint programming [HMN99,CHN00,Ng01]. In Section 3, we describe the two SRPs currently in use, namely trailing and recomputation. At the end of Section 3, we give an overview of the rest of the paper.

2 A Component Design for Search

In CP(FD), the constraint store represents a computational state, hosting finite-domain (FD) variables and constraints. A variable has a domain, which is the set of possible values it can take. A constraint maintains a relation among a set of variables by eliminating values, which are in conflict with the constraint, from variable domains according to the propagation algorithm. Every time a

Fig. 1 Depth-First Tree Search



change is made to a constraint store, a propagation engine performs constraint propagation until it reaches a fix point, in which no constraint can eliminate any more values. In our framework, we represent a constraint store by a data structure called *store* [Ng01].

Usually, constraint propagation alone is insufficient to solve a problem. Therefore, we need tree search to find a solution. A search explores the tree in a top-down fashion. Nodes and branches build up the search tree. It is adequate to view search in terms of these components: branching, node and exploration. Figure 1 provides an illustration of tree search. Circles represent nodes, while lines connecting two nodes represent branches. The numbers inside the nodes give the order of exploration. This particular example shows a DFS. For simplicity, we only consider binary search trees.

Program 1 Declaration of Branching

```
1   class Branching {
2   public:
3       bool done() const;
4       bool fail() const;
5       Branching* choose(store* s,int i) const;
6   };
```

The branching describes the shape of the search tree. Common branching algorithms include a simple labelling procedure (naive enumeration of variables), variable ordering (such as first-fail), and domain splitting. For solving scheduling problems, there is the ranking of tasks on resource (also called resource serialization). In our setting, branching coincides with the notion of a choice point. The class `Branching` shown in Program 1 has a method `choose` (line 5, for conciseness, we refer to C++ member functions as methods) which adds a constraint to the store based on the choice given and returns the branching (choice point) of the child node. `Branching` also defines methods to check whether it is done (line 3) or it has failed (line 4).

A node represents a state in the search tree. The class `Node` shown in Program 2 contains a store, a branching, and pointers to parent and children nodes (line 3-4). The constructor (line 7) takes a store and a branching as arguments. The left and right children nodes are created by calling the method `make_left_child` and `make_right_child` respectively (line 10-11). Each time a child node is created, the branching adds a constraint to the store. To proceed to

Program 2 Declaration of Node

```
1   class Node {
2   protected:
3       store* cs;
4       Branching* branch;
5       Node* parent,left_child,right_child;
6   public:
7       Node(store* s,Branching* b);
8       bool isLeaf() const;
9       bool isFail() const;
10      Node* make_left_child();
11      Node* make_right_child();
12  };
```

the next level of the search tree, constraint propagation must reach a fix point. `Node` also has methods to check if the node is a leaf node (line 8) or a failure node (line 9).

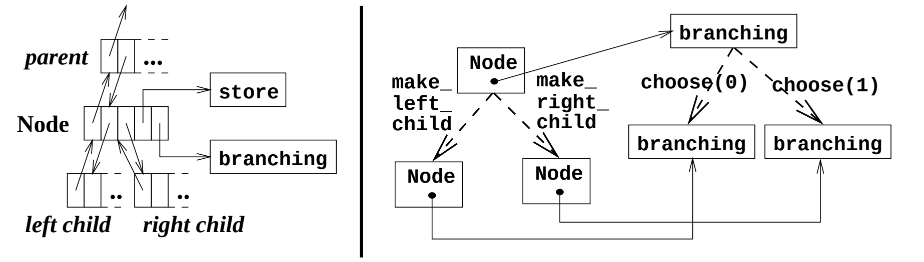
Figure 2 gives a graphical representation of nodes and branchings. The left side shows the design of nodes. A tree is linked bi-directionally, where the parent points to the children and vice versa. The right side shows the relation between nodes and branchings during the creation of children nodes. Solid arrows represent pointers, while labelled, dashed arrows represent the respective method calls. Calling either `make_left_child` or `make_right_child` method creates a child node, which, in turn, invokes the method `choose` of the current node branching that returns a branching for the child node.

The exploration specifies the traversal order of the search tree. DFS is the most common exploration algorithm used in tree search for constraint programming . Program 3 shows the implementation of DFS. Function `DFS` takes a node as an argument and tries to find the first solution using depth-first strategy. It returns the node containing the solution (line 2) or NULL if none is found (line 3). Otherwise, it recursively finds the solution on the left (line 4-5) and right (line 6) subtrees.

3 Restoration Policies

The problem of state restoration occurs in systems where a state results from a sequence of complex operations, and where the state corresponding to different

Fig. 2 Tree Node and Relation with Branching



Program 3 Exploration: Depth First Search

```
Node* DFS(Node* node) {
    if (node->isLeaf()) return node;
    if (node->isFail()) return NULL;
    Node* result = DFS(node->make_left_child());
    if (result != NULL) return result;
    return DFS(node->make_right_child());
};
```

(sub)sequences are requested over time. For example, in distributed systems, state restoration is used to recover from failure in a network node [NX95].

In constraint-based tree search, the dominant SRP has been trailing. This policy demands to record the changes done on the state in a data structure, called trail. To go from a node to its parent, the recorded changes are undone. The reason for this dominance lies in the historical fact that constraint programming evolved from logic programming, and trailing is employed in all logic programming systems for state restoration. The combination of the general idea of trailing with constraint-programming specific modifications [AB90], was deemed sufficient for constraint programming.

Schulte [Sch00] shows that other SRPs have appealing advantages. Starting from the idea of copying an entire constraint store, he introduced several SRPs that trade space for time by recomputing the store from a copy made in an ancestor node instead of making a copy at every node [Sch99]. These SRPs have the advantage of not requiring the recording of changes in propagation algorithms, thereby considerably simplifying the design of CP(FD) systems.

In the design presented in Section 2, the place where the SRP is determined is the definition of the methods `make_left_child` and `make_right_child` in the class `Node`. These methods need to create a new node together with its store and branching from the information present in the current node. This indicates that we may be able to arrive at different SRPs by providing different implementations of the `Node` class, without affecting other components such as branching and exploration. The next section shows that it is indeed possible.

By isolating the SRP in a separate component that is orthogonal to the other components, the development of new SRPs may be simplified, which may inspire the development of new SRPs. Indeed, we will present three new SRPs in Sections 5, 6, and 7. By having existing and new SRPs available in one system, we are able to conduct an experimental evaluation of them with "everything else being equal"; we report the results of this evaluation in Section 8. In that section, we also highlight the possibility of dynamically changing the SRP depending on the progress of the search, using a special-case example.

4 Restoration Components

The previous section showed that the `Node` class is the component that decides the SRP. The aim, therefore, is to design different types of nodes for different SRPs, namely, `CopyingNode` for copying and `RecomputationNode` for recomputation. All these nodes inherit from the base class `Node`. Hence, we specify the restoration component of search by passing the correct node type as an argument.

The idea for `CopyingNode` and `RecomputationNode` is presented in [Sch97a] and it allows the Oz Explorer to have copying and recomputation as SRP for DFS exploration. We separate the SRP aspect of nodes from the exploration aspect by implementing SRP-specific extensions of the `Node` base class.

The `Node` base class is similar to the one introduced in Program 2 except that it does not contain a store anymore (i. e. , remove line 3). Rather, the decision on whether to keep a store and on the type of store to keep is to be implemented in the subclasses.

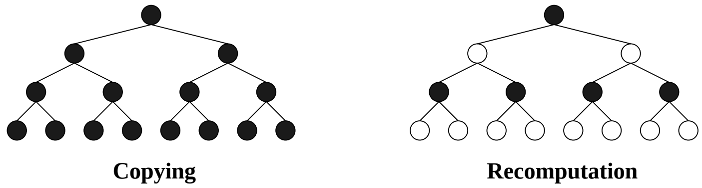
The copying SRP requires each node of the search tree to keep a copy of the store. Hence, the class `CopyingNode` contains an additional attribute to keep the copy. As the store provides a method `clone` for creating a copy of itself, when a `CopyingNode` explores and creates a child node, it keeps a copy of the store and passes the other copy to the child node.

The recomputation SRP keeps stores for only some nodes, and recomputes the stores of other nodes from their ancestors. A parameter, maximum recomputation distance (MRD) of n, means that a copy of a store is kept at every n-th level of the tree. Figure 3 shows the difference between copying and recomputation with MRD of 2. Copies of the stores are kept only in shaded nodes. Copying can be viewed as recomputation with MRD of 1.

For `RecomputationNode`, we introduce four attributes: (1) a pointer to store; (2) an integer counter d to check if we have reached the n-th level of the tree; (3) an integer `choice`, which indicates if the node is the first or the second child of its parent.; (4) and a boolean flag `copy` to indicate the presence of a copy of a store. If d reaches the n-th level limit when creating a child node, a copy of the store is kept and `copy` is set to `true`. During the exploration of a node where recomputation of the store is needed (i. e. , no copy of store is kept), the method `recompute` shown in Program 4 recursively recomputes for the store from the ancestors, by committing each parent's store to the alternative given by `choice` (line 7).

Adaptive recomputation [Sch99] improves recomputation performance by keeping only a copy of the store at a depth equidistant from the depth of an existing copy (or root, if none exists) and the depth of the last-encountered failure. It is straightforward to implement this by introducing another argument to the method `recompute` which counts the length of the recomputation path. The additional copy of the store is made when the counter reaches half the length.

Fig. 3 Copying vs. Recomputation



Program 4 Recomputing Stores in Search Tree

```
Store* RecomputationNode::recompute(int i) {
    Store* rs;
    if (copy)
        rs = cs->clone();
    else
        rs = parent->recompute(choice);
    branch->choose(rs,i);
    return rs;
};
```

During exploration, it is often clear that the store of a node is not needed any longer and can be safely passed to a child. For example in the case of DFS, we passed the store to the second child when the first child's subtree is fully explored. For such cases, nodes provide methods `create_last_right_child` and `create_last_left_child`. When a copy-holding node N is asked for its last child node A, the node N will pass its store to the child node A, which then becomes a copy-holding node. This optimization—described in [Sch00] as Last Alternative Optimization—saves space. It optimizes to do the recomputation step $N \rightarrow A$ only once.

Best solution search (for solving optimization problem) such as branch-and-bound requires dynamic addition of constraint during search to constrain the next solution to be better than the current best solution. The `Node` class has a method

```
State post_constraint(BinaryFunction* BF,store* s);
```

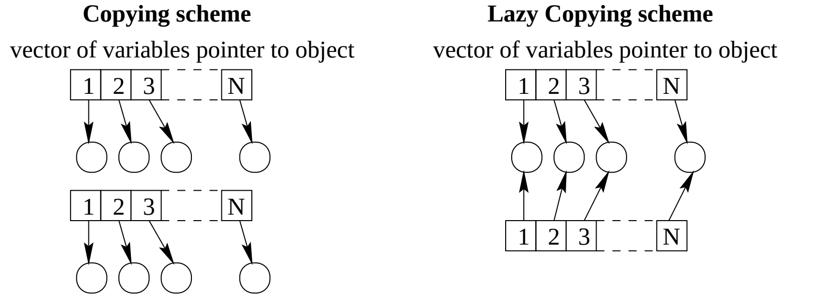
to add this constraint to the store inside a node. This addition is similar to the injection of an computation in an Oz space [Sch97b]. The method takes in a binary function to enforce the order, and the best solution store. It returns `FAIL` if enforcing the order causes failure. However, care should be taken during recomputation, where every node in the tree may not contain a copy of the store. For that, we need to introduce extra attributes to keep the constraints, which will be added as recomputation is performed.

5 Lazy Copying

Lazy copying is essentially a copy-on-write technique, which maintains multiple references to an object. A copy is made only when we write to the object. Some operating systems use this technique for managing processes sharing the same virtual memory [MBKQ96]. In ACE [PGH95], a parallel implementation of Prolog, an incremental copying strategy reduces the amount of information transferred during its share operation. In Or-parallelism, sharing is used to pass work from one or-agent to another, and is similar to the lazy copying strategy.

In other CP(FD) systems, constraints have direct references (pointers) to the variables they use and/or vice versa. In such systems, lazy copying poses the problem that every time an object (say O) is written to become N, every object

Fig. 4 Comparison between Copying and Lazy Copying



that is pointing to O would need to be copied, too, such that each new copy points to N while the old copy continues to point to O. This process needs to be executed recursively, until copies would have been made for the entire connected sub-graph of the constraints and the variables. This problem is avoided through relative addressing [Ng01], where every reference to an object is an address, called ID, in a vector of placeholders.

In our context, constraint and variable objects enjoy relative addressing. We introduce lazy copying stores, which may share individual variable and constraint objects. When we make a copy of the store, the vectors of constraints and variables point to the original objects. Figure 4 shows the differences between copying and lazy copying.

A requirement for lazy copying is that we must keep a reference count to the objects. After we lazily copy a store, the constraints' and variables' reference counts increment by 1. During a write operation, if two or more stores share this object, we create a copy of the object, assign a reference count of 1 to the new object and decrement the reference count by 1 for the old object.

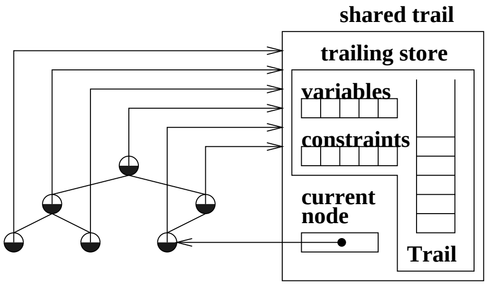
Conceptually, a lazy copying store behaves like a copying store except for its internal implementation, which consists of reference counts. The implementation of `LazyCopyNode` is straightforward. It is just the same as `CopyingNode` by replacing store with a lazy copying store described above.

6 Coarse-grained Trailing

Coarse-grained trailing is an approximation of trailing as implemented in most CP(FD) systems. Instead of trailing updates of memory locations, we trail the complete variable object or constraint object when changes occur. As mentioned in section 5, our architecture provides a relative addressing scheme and allows to make copies of variables and constraints, which make the implementation simple.

Coarse-grained trailing only keeps a single store for the entire exploration. Figure 5 shows its implementation. A half-shaded node represents a trailing node and arrows represent pointers. A trailing node holds a pointer to a common shared trail. The shared trail contains a trailing store and a pointer to the current node where the store is defined. A trailing store is needed because of the strong dependency between the store and the actual trail.

Fig. 5 Coarse-grained Trailing



Program 5 Shared Trail and Trailing Node

```
0   class TrailingNode : public Node {
1   protected:
2       int i,mark; SharedTrail* trail;
3   public: // methods declaration...
4   };
5
6   class SharedTrail {
7   private:
8       TrailingStore* ts; TrailingNode* current;
9   public:
10      SharedTrail(Store* s,TrailingNode* tn);
11      list<TrailingNode*> computePath(TrailingNode* tn);
12      void jump(TrailingNode* tn);
13  };
```

Program 5 shows the declaration of the trailing node and shared trail. The class `TrailingNode` implements the coarse-grained trailing SRP. It contains an integer `mark`, which represents the trail marker for terminating backtracking (line 2). This corresponds to the time stamping technique [AB90]. The integer `i` (line 2) indicates whether the node is the first or second child of its parent. The constructor of the class `SharedTrail` takes a store and a pointer to the root node as argument (line 10). When exploring a node D, which is not pointed to by the current node, the method `jump` (line 12) changes the trailing store from the current node to the node D. First, `jump` computes the path leading to the common ancestor with method `computePath` (line 11), then backtracks to the common ancestor, and finally descends to node D by recomputation.

The implementations of trailing and lazy copying store are closely related, since both create a copy of the changed object before a state modification occurs. Comparing to trailing, the coarse granularity imposes an overhead, which will be significant as constraints become complex (global constraints). If the constraints contain large stateful data structures, trailing may record incremental changes as opposed to copying the whole data structure on the trail as it is done by coarse-grained trailing.

7 Batch Recomputation

Recomputation performs a sequence of constraint additions and fix point computations. At earlier fix point computations, the implicit knowledge of later constraints is not exploited. This means that work is done unnecessarily, since recomputation will never encounter failure. Thus, recomputation can be improved by accumulating the constraints to be added along the path and invoke the propagation engine for computing the fix point only once. Since recomputation constraints are added all at once, we call this technique batch recomputation. Batch recomputation is also applicable to adaptive recomputation, which we call batch adaptive recomputation.

The implementation is straightforward. First, branching should provide a facility to return the constraint that is to be added during recomputation. A facility is needed to accumulate the constraints to the propagation engine. Our propagation queue already served this purpose. Lastly, we invoke the propagation of store explicitly and only once. A condition for the correctness of batch recomputation is the monotonicity of constraints, meaning that different orders of constraint propagation must result in the same fix point.

8 Experiments

This section compares and analyses the runtime and memory requirement of the different SRPs. The setup of the platform is a PC with 400 Mhz Pentium II processor, 256MB main memory and 512MB swap memory, running Linux (RedHat 6.0 Kernel 2.2.17-14). All experiments are conducted using the current development version of the Figaro system [HMN99,CHN00,Ng01], a C++ library for constraint programming.

Each SRP is denoted with the following symbols: CP - Copying, TR - Coarse-grained Trailing, LC - Lazy copying, RE - Recomputation, AR - Adaptive recomputation, BR - Batch Recomputation, BAR - Batch Adaptive Recomputation. To make comparison simple, the MRD for RE, AR, BR and BAR is computed using the formula: $MRD = \lceil depth \div 5 \rceil$ where *depth* is depth of the search tree. All benchmark timings (Time) are the average of 5 runs measured in seconds, and have been taken as wall time. The coefficient of variation is less than 5%. Memory requirements are measured in terms of maximum memory usage (Max) in kilobytes (KB). It refers to the memory used by the C++ runtime system rather than the actual memory usage because C++ allocates memory in chunks.

The set of benchmark problems are: The Alpha crypto-arithmetic puzzle, the Knights tour problem on an 18×18 chess board, the Magic Square puzzle of size 6, a round robin tournament scheduling problem with 7 teams and a resource constraint that requires fair distribution over courts (Larry), aligning for a Photo, a Hamiltonian path problem with 20 nodes, the ABZ6 Job shop scheduling benchmark, the Bridge scheduling benchmark with side constraints, and 100-S-Queens puzzle that uses three distinct (with offset) constraints.

Table 1 Characteristics of Example Programs

example	search	choice	fail	soln	depth	var	constr
Alpha	all/naive	7435	7435	1	50	26	21
Knights	one/naive	266	12	1	265	7500	11205
Magic Square	one/split	46879	46829	1	72	37	15
Larry	one/naive	389	371	1	40	678	1183
Photo	best/naive	23911	23906	6	34	95	53
Hamilton	one/naive	7150	7145	1	66	288	195
ABZ6	best/rank	2409	2395	15	91	102	120
Bridge	best/rank	1268	1261	8	78	44	88
100-S-Queen	one/ff	115	22	1	97	100	3

Table 1 lists the characteristics of the problems. These benchmarks provide the evaluation of the different SRPs based on the following criteria: problem size, amount of propagation, search tree depth, and number of failures. Our comparison of the different SRPs are based on "everything else being equal", meaning all other elements such as store, branching, exploration, etc. are kept unchange except the SRP.

Since different components of a CP(FD) system is dependent on one another, the performance may vary. For instance, the choice of FD representation has a significant effect on the performance. For these experiments, the FD representation is a lists of interval. Some problems may perform differently when a bit vector representation is used. Another remark is that the speed of copying between our system and Mozart is different for the following reasons: different FD representations, amount of data being copied, variable wake up scheme during propagation, and memory management (Mozart uses automatic garbage collection). Therefore, the result does not match exactly with Schulte [Sch99].

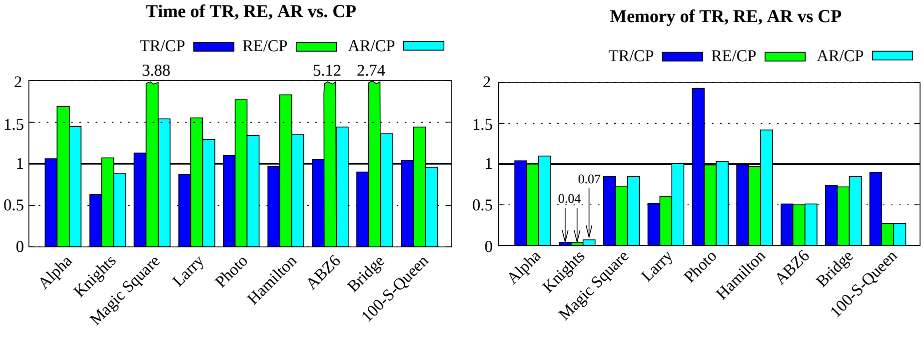
Table 2 gives the runtime and memory performance of copying. While Figure 6 shows the comparison of coarse-grained trailing and recomputation. The numbers are obtained by dividing each SRP's numbers by copying's numbers, below 1 means better performance, while above 1 means worse. This group of comparison confirms the following result of Schulte [Sch99]. Copying suffers from the problem of memory swapping for large problems with deep search trees such as Knights. Recomputation improves copying by trading space for time. Adaptive recomputation minimize the penalty in runtime of recomputation by using more space.

Coarse-grained trailing performs comparatively well to copying and other recomputation schemes. The memory peaks in Photo is probably due to STL library dynamic array memory allocation module which grows the array size by

Table 2 Runtime and Memory Performance of Copying

Example	Time	Max	Example	Time	Max
Alpha	19.200	1956	Hamilton	50.514	2176
Knights	22.086	330352	ABZ6	25.004	4936
Magic Square	160.360	2632	Bridge(10x)	8.582	2888
Larry	5.844	5712	100-S-Queen(10x)	8.444	7816
Photo	35.086	1912			

Fig. 6 Time and Memory of Coarse-grained Trailing vs. Recomputation

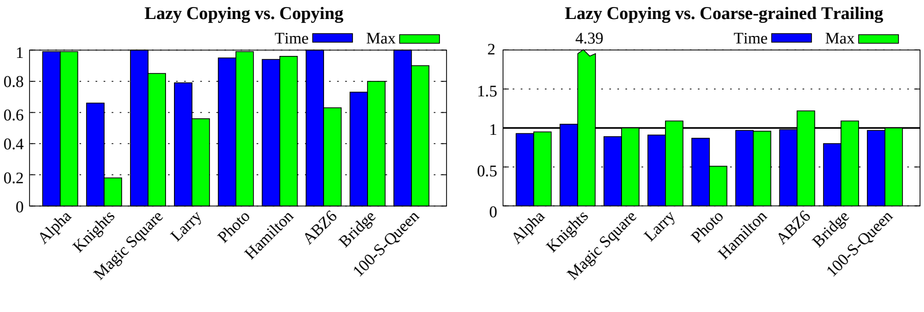


recursive doubling. Coarse-grained trailing provides us with an approximation for comparing the performance of trailing and recomputation.

Lazy copying aims at combining the advantages of both coarse-grained trailing and copying. Figure 7 shows its performance against both SRPs, the numbers are obtained by dividing lazy copying's numbers by copying's and coarse-grained trailing's numbers. Over the benchmark problems, in the worst case, lazy copying performs the same as copying, while for the cases with small amount of propagation, lazy copying can save memory and even time. Unfortunately, lazy copying still performs badly for large problems with deep search trees such as Knights, when compared to coarse-grained trailing. This is due to the extra accounting data we keep for lazy copying. However, lazy copying improves the runtime over coarse-grained trailing for problems like Magic Square, Larry and Bridge where there are many failure nodes. This is because lazy copying can jump directly from one node to another upon backtracking, while coarse-grained trailing has to carry out the extra operation of undoing the changes.

Batch recomputation aims at improving the runtime performance of recomputation. The memory requirement is the same as recomputation. Figure 8 shows the runtime performance of batch recomputation versus recomputation and batch adaptive recomputation versus adaptive recomputation. Batch recomputation improves the runtime of recomputation for all cases. However, batch adaptive recomputation improve only a little over adaptive recomputation ex-

Fig. 7 Performance of Lazy Copying vs. Copying and Coarse-grained Trailing



cept for Larry. This is due to the design of adaptive recomputation which makes a copy in the middle when a failure is encountered, which in turn, reduces the recomputation distance that batch recomputation can take advantage of.

By encapsulating SRP into tree nodes, we are able to come up with a new scheme called *switching* in which we could apply different SRPs in different parts of the search tree. This scheme is useful when different parts of the search tree exhibit distinct characteristics. One special-case example is the problem of finding the first solution for the 510-S-Queen problem. The search tree of this problem have a straight path from the root node to a node of depth 499 where there is a small subtree with some failure nodes and the solution. Therefore, we can have recomputation before depth 499, and coarse-grained trailing after depth 499. The implementation is straightforward, when the exploration reaches depth 499, we will create a coarse-grained trailing node instead of recomputation, because the SRP of a subtree is dictated by its root node. Figure 9 shows the runtime and memory improvement over other SRPs. Its is better than all other SRPs for this problem.

Comparison with other constraint programming systems are needed in order to gauge the effect of the component architecture and the overhead for relative addressing. Initial results are reported in [Ng01].

9 Conclusion

We developed an architecture that allows us to isolate the state restoration policy (SRP) from other components of the system. Its main features are:

Relative addressing: Variable and constraint objects are referred to by IDs, which are mapped to actual pointers through store-specific vectors.

Branching objects: Search trees are defined by branching objects, which are recursive choice points.

Exploration algorithms: Exploration algorithms are defined in terms of a small number of operations on nodes.

SRPs are represented by different extensions of the base class `Node`. Apart from copying, recomputation, we introduced three new SRPs.

Fig. 8 Time of Batch Recomputation vs. Recomputation

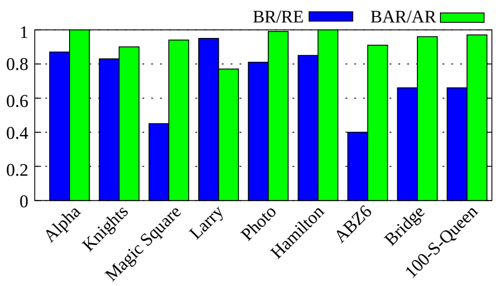
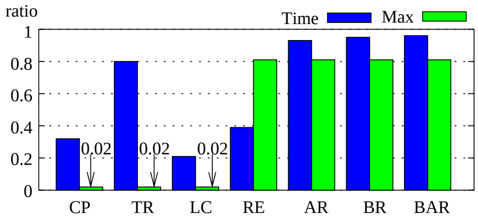


Fig. 9 Time and Memory of Switching vs. other SRPs



Lazy copying uses a copy-on-write technique for variables and constraints and improves over or is equally good as copying on all benchmarks. Lazy copying benefits from relative addressing.

Coarse-grained trailing is a form of trailing that copies the state of variables and constraints, as opposed to incremental changes, onto the trail.

Batch recomputation modifies recomputation by installing all constraints to be added to the ancestor at once and improves over Schulte's recomputation for all benchmarks.

The presented architecture allows the user to optimize time and space consumption of applications by choosing existing or designing new SRPs in response to application-specific characteristics. We highlighted the flexibility of the architecture using an example of a problem-specific SRP, where the state restoration policy is switched dynamically during search. The SRP components are designed and implemented in C++ on the base of the Figaro library for constraint programming [HMN99,CHN00,Ng01], and evaluated on a set of benchmarks ranging from puzzles to realistic scheduling and timetabling problems. State restoration is an important aspect of tree search that deserves the attention of users and constraint programming systems designers.

Acknowledgements

We thank Tobias Müller and Christian Schulte for valuable feedback on this paper, Ong Kar Loon for continuous discussions and collaboration on the Figaro library, and Edgar Tan for comments.

References

[AB90] 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.

[CHN00] Tee Yong Chew, Martin Henz, and Ka Boon Ng. A toolkit for constraint-based inference engines. In Enrico Pontelli and Vítor Santos Costa, editors, *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000*, Lecture Notes in Computer Science 1753, pages 185–199, Boston, MA, 2000. Springer-Verlag, Berlin.

[CJL99] Yves Caseau, François-Xavier Josset, and François Laburthe. CLAIRE: Combining sets, search and rules to better express algorithms. In Danny De Schreye, editor, *Proceedings of the International Conference on Logic Programming*, pages 245–259, Las Cruces, New Mexico, USA, 1999. The MIT Press, Cambridge, MA.

[DC00] Daniel Diaz and Philippe Codognet. The GNU prolog systems and its implementation. In *ACM Symposium on Applied Computing*, Como, Italy, 2000. Documentation and system available at `http://www.gnu.org/software/prolog`.

[HMN99] Martin Henz, Tobias Müller, and Ka Boon Ng. Figaro: Yet another constraint programming library. In *Proceedings of the Workshop on Parallelism and Implementation Technology for Constraint Logic Programming*, Las Cruces, New Mexico, USA, 1999. held in conjunction with ICLP'99.

[ILO00] ILOG Inc., Mountain View, CA 94043, USA, `http://www.ilog.com`. *ILOG Solver 5.0, Reference Manual*, 2000.

[Int00] Intelligent Systems Laboratory. *SICStus Prolog User's Manual*. SICS Research Report, Swedish Institute of Computer Science, URL `http://www.sics.se/isl/sicstus.html`, 2000.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, Reading, MA, 1996.

[Ng01] Ka Boon Kevin Ng. *A Generic Software Framework For Finite Domain Constraint Programming*. Master's thesis, School of Computing, National University of Singapore, 2001.

[NX95] R. H. B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, (6):165–169, 1995.

[PGH95] Enrico Pontelli, Gopal Gupta, and Manuel Hermenegildo. &ACE: A high performance parallel prolog system. In *9th International Parallel Processing Symposium*, pages 564–571. IEEE Press, 1995.

[Sch97a] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press, Cambridge, MA.

[Sch97b] Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Principles and Practice of Constraint Programming—CP97, Proceedings of the Third International Conference*, Lecture Notes in Computer Science 1330, pages 519–533, Schloss Hagenberg, Linz, Austria, October/November 1997. Springer-Verlag, Berlin.

[Sch99] Christian Schulte. Comparing trailing and copying for constraint programming. In Danny De Schreye, editor, *Proceedings of the International Conference on Logic Programming*, pages 275–289, Las Cruces, New Mexico, August 1999. The MIT Press, Cambridge, MA.

[Sch00] Christian Schulte. *Programming Constraint Services*. Doctoral dissertation, Universitt des Saarlandes, Naturwissenschaftlich-Technische Fakultt I, Fachrichtung Informatik, Saarbrcken, Germany, 2000. To appear in Lecture Notes in Artificial Intelligence, Springer-Verlag.