Register Allocation Using Control Trees

Kathleen Knobe[1]
Kenneth Zadeck[2]

Technical Report No. CS-92-13
March 1992

[1] Artificial Intelligence Lab., NE43-630, Massachusetts Institute of Technology, Cambridge, MA 02139
[2] Brown University, Department of Computer Science Box 1910, Providence, RI 02912 USA

Register Allocation Using Control Trees

Kathleen Knobe* Kenneth Zadeck†

Contact Author – Zadeck (e-mail fkz@cs.brown.edu) (phone 401-863-7635)

1 Introduction

Register allocation is like trying to fit blocks in a box. There are two distinct problems: first, the volume of the blocks may be too large for the box, and second, the shapes of the blocks may make the box difficult to pack. No matter how good your packing technique is, you cannot succeed if the volume of blocks is too large. On the other hand, that the volume of blocks is acceptable does not guarantee that the box can be packed.

In the analogy to register allocation, each live range associated with a candidate (object to be allocated) is a block. Other algorithms view blocks as having uniform density. Our blocks have *holes*, regions in which the candidate's live range is reference-free. We prune portions of the program where the number of live candidates is greater than the number of registers (the *register pressure* is too high) by storing the value in memory on entry to the region and reloading it on exit. It is always possible to prune enough that the volume of the candidates does not exceed the volume of the box.

In theory, once the volume problem is solved, packing is easy if one has a chainsaw: when something doesn't fit, cut it in two and try again. In practice, the problem is difficult to solve well because splitting has

*Artificial Intelligence Lab., NE43-630, Massachusetts Institute of Technology, Cambridge, MA 02139. The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-87K-0825 and by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation, IBM Corporation, and AT&T.

†Computer Science Dept., P.O. Box 1910, Brown University, Providence, RI 02912. The research reported in this paper was supported in part by the Office of Naval Research and the Defense Advanced Research Projects Agency under contracts N00014-83-K-0146 and ARPA order 6320 and N00014-91-J-4052, ARPA Order 8225, and by the National Science Foundation under grant CCR-9015988.

a cost: move instructions must be inserted between the split regions. It is important to select not only the proper candidate to split but also the proper location of the split to facilitate packing the remaining blocks and also to minimize the costs of the move instructions.

Both the pruning and the splitting use the *control tree*[1] to take into account the program structure in determining costs. Since the algorithm uses the control tree to guide the pruning and splitting decisions, the algorithm is called *control-tree register allocation* algorithm. This algorithm has the steps shown in Fig. 1.

```
build the control tree;
perform pruning; /*  Sec. 3  */
build interference graph;
perform coloring; /*  Sec. 4  */
```

Figure 1. The control-tree algorithm.

2 Lifetime Information

Lifetime information does not appear in its traditional form in the control-tree register allocation scheme. Instead, each control-tree node contains information pertinent to the candidate within the region of code represented by that node.

Each control tree node n has a number of attributes:

- *Live* variables are the output of standard live-dead analysis.
- $LiveAll[n]$ is the set of candidates that are live throughout n. *LiveAll* at a control-tree node is the intersection of *LiveAll* at all its children. $LiveAll[n] \equiv Live[n]$ if n is a leaf.
- $LivePart[n]$ is the set of candidates that are live at some point within n. *LivePart* at a control-tree node is the union of *LivePart* at all its children. $LivePart[n] \equiv Live[n]$ if n is a leaf.
- $Defined[n]$ is the set of candidates that have a definition within control tree node n. *Defined* at a control-tree node is the union of *Defined* at all its children.

[1] A control tree is a tree that represents the control structure of the program [Sha80]. An abstract syntax tree of the executable parts of a program is rough approximation of the control tree. There are significant differences between the two that will be presented in the full paper.

- *Referenced*[n] is the set of candidates that have a use or definition in any child of n. *Referenced* at a control-tree node is the union of *Referenced* at all its children.

Conceptually, this structure continues down to the level of right and left sides of assignments, which are the minimal lifetime units. In practice, a more compact form is used at the basic block level, as described in the full paper.

A *name* can be associated with more than one candidate if it has disjoint lifetimes that can be allocated registers independently. Initially, we assume a one-to-one correspondence between names and candidates.[2]. The *headnode* for a candidate C is a control-tree node n that is the lowest node in the control tree representing a program fragment that includes a live range for C, i.e.:

- C is not live on entry to n.
- C is not live on exit from n.
- C is live at some point in n.
- No descendant of n has the above properties.

A *candidate* is a pair consisting of a name and a control-tree node. We use $Name[C]$ and $Node[C]$ to get at the name and headnode of a candidate C. Initially the candidates are the set of names and their headnodes. A name appears exactly once in the initial list of candidates. As the coloring algorithm progresses, a candidate may be split into several candidates that share the same name but have distinct control-tree nodes.

3 Pruning

Our register-allocation technique does no spilling at all. Other register-allocation techniques have a crude approach to spilling: either they spill an entire candidate to memory [CAC+81, Cha82], or they break the candidate into successively smaller pieces, each of which is either allocated or spilled [CH90]. These approaches have the drawback that the choice of which candidate to spill is based not on the cost of spilling

[2] This can be accomplished by using the *right number of names* algorithm [CAC+81]

that candidate, but rather on which candidate is the least important to allocate to a register. The difference is significant. In other algorithms, a priority is associated with each candidate that indicates how important it is to allocate that candidate to a register. Those priorities are essentially the sum of the references times the estimated execution frequency for each reference. The control-tree approach assigns to the *spaces* between the references (which we call *wedges*) a priority that models how much it costs to store and reload the value between the references. We call the process of spilling these wedges *pruning*. While other algorithms perform spilling as part of their coloring process, the control-tree algorithm prunes before coloring. At any point where the register pressure is higher than some preset threshold, portions of candidates are pruned until the register drops below the threshold.

A *wedge* is a subtree of the control tree in which a candidate is live somewhere but is not referenced. The pruning phase finds desirable wedges to allocate to memory in order to lower the register pressure. On every path into the region defined by the pruned wedge where the candidate is both live and dirty, a `store` instruction is inserted. On every path out of the region defined by the wedge where the candidate is live, a `load` instruction is inserted. Because of the way wedges are defined, the candidate will be live on at least one path into and one path out of the region.

In most cases the tops of wedges at control tree node n are the sets of candidates not in $Referenced[n]$ but in $Referenced[n.Parent]$. Sometimes it is desirable to push the wedge tops lower in the control tree. For instance, if the top of the wedge is at an `if-then-else` node, it is desirable to split the wedge into two wedges, one at the top of the `then` part and one at the top of the `else` part. This reduces the cost of `loads` and `stores`, assuming only one child is pruned. On the other hand, it is never desirable to push the top into a loop. The control-tree node types in which it is desirable to push down the wedge tops are called $PushableTypes$, as further described in the full paper.

Pruning is implemented by the code in Figs. 2 and 3. Three data structures are used here:

- $HighPressureCount[n]$ is an array of counters, one for each control tree node n, that indicate how many instructions under n have register pressure above threshold.
- $LiveSize[n, C]$ is 1 if candidate C is live at a control-tree leaf n and 0 if C is not live at leaf n. For interior control-tree nodes, $LiveSize[n, C] \equiv \sum_{m \in n.children} LiveSize[m, C]$.

```
procedure InitPrune(N)
    HighPressureCount[N] ← 0;
    if (N ∈ CT.Leaf) then
        if (|Live[N]| > threshold) then HighPressureCount[N] ← 1; fi;
        foreach (C ∈ Live[N]) do LiveSize[C, N] ← LiveSize[C, N] + 1; od;
        LivePart[N] ← Live[N]; LiveAll[N] ← Live[N];
    else
        foreach (M ∈ N.Children) do call InitPrune(M); od;
        LivePart[N] ← ⋃_{M∈N.Children} LivePart[M];
        LiveAll[N] ← ⋂_{M∈N.Children} LiveAll[M];
        Referenced[N] ← ⋃_{M∈N.Children} Referenced[M];
        HighPressureCount[N] ← ∑_{M∈N.Children} HighPressureCount[M];
        foreach (C ∈ LivePart[N]) do
            LiveSize[C, N] ← ∑_{M∈N.Children} LiveSize[C, M];
        od;
        foreach (C ∈ Referenced[N]) do
            foreach (M ∈ N.Children) do if (C ∉ Referenced[M]) then add C to Wedges[M]; fi; od;
        od;
    fi;
end InitPrune
```

Figure 2. Initialization for pruning code.

- *Wedges* is an array of lists of candidate names. There is one element of the array for every control-tree node. The list for control-tree node n contains the candidates with wedges that have heads at n. Each list is ordered so that the most desirable candidates to prune are at the front. Three parameters are used to order the list:
  - The number of live, dirty paths entering the control tree node. Candidates with the smallest number of these paths are placed before those with more paths.
  - The number of live paths exiting the control tree node. Candidates with the smallest number of these paths are placed before those with more paths.
  - *LiveSize*. Candidates with greater *LiveSize* should be placed before those with the smaller *LiveSize*.

The choice of which wedges to prune is a function of the cost of each range and the size of the area that is improved. *InitPrune* computes the *HighPressureCount* and *LiveSize* in a bottom-up pass of the control tree, and *Prune* does the actual pruning in a top-down pass of the control tree. At each node in the control tree, *Prune* selects the wedges to prune and calls *PruneWedge* to update the *Live*, *LivePart*, *LiveSize* and *HighPressureCount* structures by a bottom-up walk of the control tree ending at the wedge top.

```
LiveSize[*,*] ← 0;
call InitPrune(CT.Root);
call Prune(CT.Root);

procedure Prune(N)
    if (N.Type ∈ PushableTypes) then foreach (M ∈ N.Children) do add Wedges[N] to Wedges[M]; od;
    else
        Order Wedges[N];
        while (HighPressureCount[N] > 0 and |Wedges[N]| > 0) do
            take W from Wedges[N];
            insert appropriate load and store instructions;
            call PruneWedge(N, W);
        od;
    fi;
    foreach (M ∈ N.children) do if (HighPressureCount[M] > 0) then call Prune(M); fi; od;
end Prune

procedure PruneWedge(N, W)
    delete W from Live[N];
    HighPressureCount[N] ← 0;
    if (N ∈ CT.Leaf) then if (|Live[N]| > threshold) then HighPressureCount[N] ← 0; fi;
    else
        foreach (M ∈ N.children) do
            if (W ∈ LivePart[M]) then
                HighPressureCount[N] ← HighPressureCount[N] + PruneWedge(M, W);
            else HighPressureCount[N] ← HighPressureCount[N] + HighPressureCount[M];
            fi;
        od;
    fi;
    delete W from LivePart[N], LiveAll[N];
    return HighPressureCount[N];
end PruneWedge
```

Figure 3. Pruning.

Assuming that the coloring phase is allowed to split candidates, it is ***sufficient*** to set the threshold at one less than the number of registers available on the machine (see Cytron and Ferrante [CF87]) where by sufficient, we mean that the next phase will always terminate with a coloring. For most machines, this is probably the right place to start. The desirability of further pruning depends on architectural decisions and interactions with the next phase, as discussed further in the full paper.

4 Coloring

The technique for performing the allocation is a variation of Chaitin's graph-coloring algorithm. The main difference occurs when the algorithm blocks, i.e. when there are no unconstrained candidates to remove from the graph: Chaitin's algorithm orders the constrained candidates and pushes them on the stack, and later,

during the assignment phase, any candidates that cannot be assigned a color are spilled and whole process begins again. In the control-tree algorithm, on the other hand, constrained candidates are never pushed onto the stack. When the algorithm is blocked, a heuristic is used to choose a ***parent candidate*** to split into two or more ***child candidates***, since the number of interferences incident on any of the child candidates is not greater (and is typically less) than the number on the parent.

When we split a candidate, we must update the interference graph (see Sec. 4.2). This may cause one or more of the children to become unconstrained, thus unblocking the process. We continue splitting candidates until some candidate becomes unconstrained. Then we continue on pushing candidates onto the stack. Choosing the proper candidate to split is discussed in Sec. 4.3.

Splitting has the unfortunate side effect of raising the number of edges incident on other candidates in the interference graph (some of which may have already been pushed onto the stack), since a candidate that used to interfere with the parent candidate may now interfere with more than one of the child candidates. Rather than removing this newly constrained candidates from the stack by some sort of backtracking, we leave them and continue the process until the graph becomes empty. At that point, register assignment is attempted. If the assignment is successful, we stop, otherwise we perform another round of simplification. Pseudocode is given in Fig. 4.

Below we discuss the data structures required to keep the interference information consistent when splitting and how to choose the candidate to be split.

4.1 Initial Interference Graph

Chaitin's algorithm views the interference graph as an undirected graph. For the control-tree algorithm, it is useful to overlay the interference graph onto the control tree. The ***interference graph*** is a graph whose nodes are candidates and whose edges connect the candidates C_1 and C_2 if they interfere. The interference information is implicit in the lifetime information in the control-tree. For example, if C_1 is in $LiveAll[n]$ and C_2 is in $Defined[n]$, then C_1 and C_2 interfere. However, some interferences within control-tree node n may be discovered only when examining descendants of n. For example, if C_1 is in $LivePart[n]$ and C_2 is in $Defined[n]$, then C_1 and C_2 may or may not interfere.

```
procedure Color()
    repeat
        CandidateStack ← SimplifyGraph(InterferenceGraph);
    until (AssignColors(InterferenceGraph, CandidateStack))
end Color
function SimplifyGraph(InterferenceGraph) : stack of candidates
    stack ← ∅;
    while (G ≠ ∅) do
        if (there is a v with degree < N) then choose v with largest degree (among those < N);
        else
            repeat
                find the best candidate, C, to split;
                split C into C1...Ck;
                delete C from InterferenceGraph;
                for i = 1 to k do add Ci to InterferenceGraph; od;
            until (there is a v with degree < N)
            choose v with minimum cost according to Heuristic;
        fi;
        delete v from G;
        push v on stack;
    od;
    return stack;
end SimplifyGraph
function AssignColors(InterferenceGraph, CandidateStack) : boolean
    foreach (for each candidate v popped off stack) do
        if (a color C is available for v) then color(v) ← C;
        else return false;
        fi;
    od;
    return true;
end AssignColors
```

Figure 4. The *Color* and *SimplifyGraph* functions.

The initial interference graph and *Defined* are computed in a bottom-up traversal of the control tree, while *LiveAll* and *LivePart* are computed in the pruning phase. At any control-tree node where a candidate C is in *LiveAll* and candidate D is in *Defined*, we know that C and D interfere. This interference between C and D may be visible at a large number of control-tree nodes. The key to computing the interference graph efficiently is to avoid performing an all-candidates-by-all-candidates cross-product at every node in the control tree. We choose the particular control-tree node at which C makes the transition from being in *LiveAll* to not being in *LiveAll* as the place to add the edge to the interference graph. The set of such candidates is called the *Trans* candidates of the control-tree node. At each control-tree node, the cross-product of *Trans* candidates and those in *Defined* represent the known interferences at that control-tree node. These interferences are added to the interference graph before continuing up the tree. If two candidates interfere, then there is some node in the control tree where one is in *Defined* and the other is in *Trans*. This

technique finds the same interferences as Chaitin's technique but does so faster, since the cross-products are smaller.

```
foreach (N ∈ CT) order Bottom-up do
    if (N ∉ CT.Leaf) then
        Defined[N] ← ⋃_{M∈N.Children} Defined[M];
        Trans ← LiveAll[N] ∧ not LiveAll[N.Parent];
        foreach (y ∈ Defined[N]) do
            foreach (x ∈ Trans) do
                add ((y, headnode[y])(x, headnode[x])) to InterferenceGraph;
                /* Note that at this stage the headnode for each name is unique */
            od;
        od;
    fi;
od;
```

Figure 5. Computation of initial node attributes and initial interferences.

4.2 Updating the Interference Graph

The node in the interference graph representing a candidate C is a name/control-tree-node pair. Each candidate is initially associated with the headnode of that candidate in the control tree. The edges in the interference graph associated with C are partitioned into $k+1$ disjoint classes where k is the number of children of $Node[C]$: partitions 1 through k are ***child partition*** while partition 0 is the ***ancestor partition.*** The basis of the partitioning is the location of $Node[D]$, where D is the candidate on the other end of the interference edge. If $Node[D]$ is located at or below the nth child of $Node[C]$, then the edge (C, D) is placed in child partition n of C. If $Node[D]$ is at or above $Node[C]$, the edge (C, D) is placed in the ancestor partition of C. If edge (C, D) is in a child partition at $Node[C]$ then edge (C, D) is in the ancestor partition at $Node[D]$.

Splitting candidate C involves replacing C with up to k candidates, one for each child n_i of $Node[C]$ where $C \in LivePart[n_i]$. Splitting a candidate changes the interference graph in several significant ways:

1. Each child candidate of C inherits the interference graph edges from the candidates whose headnodes are at or below that child of $Node[C]$; that is, child candidate i of C inherits interferences in partition i of C.

2. Each child candidate of C does not inherit the candidates whose headnodes are at or below the other

children of $Node[C]$; that is, child candidate i does not inherit interferences in partition j of C when $i \neq j$.

3. Each child candidate of C may inherit some or all of the edges from candidates in the ancestor partition of C.

To redetermine the interferences based on (3), we use a modified version of the algorithm that computed initial interferences; the code is given in Fig. 6. This differs from the batch technique in several ways: we process only the subtree defined by $Node[C]$, and we are interested only in candidates whose names are in the ancestor partition of C. Instead of a bottom-up traversal, the incremental version is top-down, starting at the control-tree node at which the candidate is split. Rather than computing all the interferences with the split candidate in one pass, we compute each possible pair in a separate pass. This allows us to avoid searching subtrees that will not contribute any information.

```
foreach (D ∈ InterferenceGraph[C]) do
    foreach (Child ∈ Node[C].Children) do
        if (CheckInterf(C, D, Child)) then add ((C, i), (D, headnode[D])) to InterferenceGraph; fi;
    od;
od;
function CheckInterf(C, D, Node) : boolean
    if (C ∈ Defined[Node] and D ∈ LiveAll[Node]) then return true; fi;
    if (D ∈ Defined[Node] and C ∈ LiveAll[Node]) then return true; fi;
    if (C ∉ LivePart[Node]) then return false; fi;
    if (D ∉ LivePart[Node]) then return false; fi;
    if (C ∉ Defined[Node] and D ∉ Defined[Node]) then return false; fi;
    foreach (Child ∈ Node.Children) do if (CheckInterf(C, D, Child)) then return true; fi; od;
end CheckInterf
```

Figure 6. Computation of new interferences after a split.

The result of these cross-products is pairs of *names* that interfere. This was exactly what was needed for the initial interferences in which one name was one candidate, but after splitting, several candidates may have the same name. Recall that a candidate is defined as not only a name but a name/headnode pair. Given an interference between a name C and some other name D, we need to find the candidates associated with these two names, that is, we need to find their *headnodes*.

Notice that the bottom-up computation that finds the interference between names C and D stops at a child of $Node[C]$. That child node is the headnode for name C with respect to any interferences found below

it. Conversely, given a name D that interferes with candidate C_i we need to find the candidate, therefore the headnode, associated with D for that interference. The candidate associated with name D at $Node[C_i]$ is exactly the same candidate as is associated with D at $Node[C]$. Since this candidate was in the ancestor partition of C, it will also be in the ancestor partition of C_i.

4.3 Choosing the Candidate to Split

The only question remaining is which candidate to split when the algorithm becomes blocked. The candidates can be partitioned into two sets: (1) those parent candidates that, when split, yield only unconstrained child candidates, and (2) those parent candidates, that when split, yield one or more constrained child candidates. This partitioning is done by making the pessimistic assumption that all of the edges from the ancestors are inherited by all of the children. This is because the control tree allows us to determine cheaply how the interferences from below are partitioned among the children, but not how the interferences in the ancestor partition are partitioned.

Set (1) is the most desirable, and we will choose one from it if it is not empty. The best one from that set is the one with the fewest interferences in its ancestor partition because splitting this candidate does the least damage to the existing stack.

If set (1) is empty, we choose a candidate from set (2). Splitting a candidate from this set (2) may not unblock the algorithm, so our heuristic is to choose the candidate whose children are the least constrained. For each candidate we find the largest number of interferences associated with any of its children; then the candidate to split is the one with the smallest such number.

5 Discussion

We present two independent heuristics to improve register allocation, one for pruning and another to do splitting, while coloring. The distinction between these two problems is one of the interesting aspects of this technique. Pruning can be incorporated into any register-allocation scheme. Our splitting heuristics enhance Chaitin's already effective coloring scheme. Separating the two problems allows different metrics to be used for each. By using the control tree in both cases, we can make the best decision on the basis of the

overall structure of the program.

Pruning lowers the register pressure throughout a program by allocating to memory regions where a candidate is live but not used. This is less expensive than spilling, which allocates to memory arbitrary regions where the candidate is live. In any well-engineered compiler, the design of the optimization phase is constrained by the amount of register pressure it is allowed to create. Optimizations like loop unrolling, code motion, and inlining significantly increase the register pressure, often by creating large regions where the candidate is live but not referenced. While the current generation of architectures has increased the number of registers available, it has also increased the need for more aggressive optimization. Pruning makes the optimization phase less constrained by the register pressure limitation.

Splitting has been difficult to incorporate into graph coloring because updating the interference graph as the candidates are split was thought to be costly. Chaitin's algorithm does no splitting while Chow's algorithm uses a weaker notion of interference. While we have provided some heuristics to guide the splitting decisions, the key contribution of the splitting technique is the use of the control tree to incrementally update the interference graph efficiently.

6 Related Work

Chaitin's algorithm [CAC+81, Cha82] is the basis for our coloring phase. Experience with this technique has shown that it colors most programs very well, but tends to fail catastrophically by making poor spilling decisions when the register pressure exceeds the number of available registers. Pruning attacks this problem directly.

Chow's algorithm [CH84, CH90] uses splitting as a fallback when coloring fails. Because this algorithm does not distinguish between stacking and assignment as Chaitin's algorithm and therefore ours do, Chow's colorings are generally inferior when the pressure is moderate. On the other hand, because Chow's technique can split when blocked, it achieves better allocations than Chaitin's technique when the register pressure is very high or the live ranges very tangled. By incorporating splitting into Chaitin's algorithm, we achieve the advantages of both techniques.

Callahan and Koblens [CK91] also incorporate hierarchical information into their register-allocation

technique. The algorithm first splits every candidate at every control-tree node where it is live. Then Chaitin's coloring technique is performed independently on each control-tree node with one significant change: pseudo-colors rather than real colors are assigned to each candidate. This algorithm has much the same effect as ours: since different costs are associated with the different pieces of the candidates, their local spilling has the same effect as our pruning. Furthermore, their splitting is much more aggressive than ours, since they assume that everything has been split while we split only on demand.

The real difficulty of the Callahan and Koblens algorithm is how to put the program back together after such aggressive splitting. Since each control-tree node has been colored independently, it is necessary to *match up* the colorings as well as possible. Whereever the matching fails, `move` instructions must be inserted, and the number of these inserted instructions may be very large since this is not modeled in the coloring phase. On the other hand, this may not be a problem for them: the dataflow machine for which they are generating code may be able to perform a very large number of `move` instructions in the time necessary to do a single memory reference.

References

[CAC+81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.

[CF87] R. Cytron and J. Ferrante. What's in a name? *Proc. 1987 IEEE International Conf. on Parallel Processing*, pages 19–27, August 1987.

[CH84] F. Chow and J. Hennessy. Register allocation by priority-based coloring. *Proc. SIGPLAN'84 Symp. on Compiler Construction*, pages 222–232, June 1984. Published as SIGPLAN Notices Vol. 19, No. 6.

[CH90] F. Chow and J. Hennessy. The priority-based coloring approach to register allocation. *ACM Trans. on Programming Languages and Systems*, 12(4), October 1990.

[Cha82] G. J. Chaitin. Register allocation and spilling via graph coloring. *Proc. SIGPLAN'82 Symp. on Compiler Construction*, pages 98–105, June 1982. Published as SIGPLAN Notices Vol. 17, No. 6.

[CK91] D. Callahan and B. Koblens. Register allocation via hierarchical graph coloring. *Proc. SIGPLAN'91 Symp. on Compiler Construction*, June 1991. Published as *SIGPLAN Notices* Vol. 26, No. 6.

[Sha80] M. Sharir. Structural analysis: a new approach to flow analysis in optimizing compilers. *Computer Languages*, 5:141–153, 1980.