Konrad-Zuse-Zentrum für Informationstechnik Berlin

Herbert Melenk Winfried Neun

Implementation of Portable Standard LISP for the SPARC Processor

Technical Report TR 89-6 (July 1989)

Herbert Melenk Winfried Neun

Implementation of Portable Standard LISP for the SPARC Processor

Abstract

The SPARC processor is a RISC (Reduced Instruction Set Computer) microcomputer, built into the SUN4 workstations. Since RISC processors are very well-suited for LISP processing, the implementation of a dialect of LISP (Portable Standard LISP, PSL) boded well for a great speed-up in comparison with other types of microcomputers. A first approach was done at The RAND Corporation in Santa Monica, which was derived from classical processor types like MC68000 or VAX. At the Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB) that initial implementation was redesigned in order to adapt PSL to the specific features of the SPARC processor. The present implementation, in some parts, is very close to the Cray PSL version also done in ZIB. Some timing informations are given in the appendix.

Contents

1. Introduction **1**

2. SPARC Processor Features of Special Interest for LISP Performance **3**

2.1 Register Windowing for Integer Unit 3

2.2 SPARC Logic and Arithmetic Instructions 4

2.3 Delay Slots after Branch or Call Instructions 4

2.4 Memory Access 5

3. PSL Model for the SPARC **6**

3.1 Allocation of PSL Data Structures 6

3.1.1 The LISP Items 6

3.1.2 A Short Discussion on "high or low" Tags 7

3.1.3 Registers 7

3.1.4 Usage of the Register File 8

3.2 SPARC Specific Optimization Pass for the Assembly Phase 9

3.3 Pseudo Registers 12

3.4 Miscellaneous 12

4. Further Developments **13**

Appendix **14**

Acknowledgement **16**

References **17**

1. Introduction

The *Scalable Processor Architecture* (SPARC) processor is a Reduced Instruction Set Computer (RISC) designed by SUN microsystems and was introduced on the market with the SUN4 workstations in 1987 [4]. The development of RISC systems started with the IBM 801 in 1979. One may even look at the Cray-1 or CDC 6x00 series as early RISC machines.

The general characteristics of RISC processors are:

- It uses only few "simple" instructions.
- Almost all instructions last the same number of cpu cycles.
- Limitation of addressing modes to (one or two) simple types.
- Presence of lots of registers as intermediate (fast) memory.
- Same length for all instruction formats.

The high number of registers is said to be a side effect of the reduction of instructions, since now chip surface was free. This feature is of great value for many compilers, since it allows to put local variables in fast storage instead of stack memory. For highly recursive languages like LISP the impact is extraordinary [5].

Because of the reduced number and complexity of the instructions, the processor is able to issue more instructions per time unit than a usual microprocessor with the same clock rate.

On the other hand, the compilation of complex constructs, which can benefit from a more complex instruction set, will use significantly more instructions on a RISC processor, since the complex instruction must be emulated, e.g. the MVCL or BCTR instructions on IBM 370 architecture. The compilation of complex constructs will cost a lot more compilation and optimization efforts. This will make a RISC processor less attractive for many applications.

For implementations of LISP the situation is quite different, since (classic) LISP programs are based mainly on two operation types which have to be implemented fast, namely:

- Operations on lists, which need fast access to the first element of a list (CAR) and the rest of the list (CDR), which are very simple memory addressing modes.
- The function call, since LISP programs normally use recursion in extensive manner and the functions themselves are very small.

These two operations need very primitive instructions only, namely non-indirect load and store from/to memory and a call instruction with an efficient parameter protocol. More complex instructions, if present, could not be used.

An example for that is one of the workhorses of PSL:
The function memq (search an identifier in a list) is defined as:

```
(de memq (x list)
     (cond ((atom list)  nil)
           ((eq (car list) x) list)
           (t (memq x (cdr list)))))
```

The equivalent in RLISP (a dialect of LISP, written ALGOL like):

```
procedure memq (x,list);
    begin;
       if atom(list) then return NIL
        else if car (list) eq x then return list
              else return memq(x,cdr list);
    end;
```

Even the reader not familiar with LISP will see that the execution of that code is dominated by the two memory operations (`Car list`) and (`cdr list`) and the recursive call to memq (atom and eq are in line tests).

The code produced by the compilation of this function and its optimization will be shown in Section 3.4.

In Konrad-Zuse-Zentrum für Informationstechnik Berlin, the implementation of Portable Standard LISP (PSL) for Cray processors was started in 1986 [3], [1]. PSL is used mostly as underlying LISP system for the Computer Algebra system REDUCE [2]. The REDUCE implementation on Cray systems based on PSL is one of the world's fastest (measured by REDUCE's Standard Test sequence), and was the first one that ever ran the test sequence in less than 1 second.

Since many characteristics of the RISC processors are common with Cray processors too, the idea for the SPARC processor implementation was to follow the same guidelines.

2. SPARC Processor Features of Special Interest for LISP Performance

In this section we will describe features of the SPARC processor, which are not common to all RISC architectures and which influenced our LISP (compiler) implementation.

2.1 Register Windowing for Integer Unit

The registers of the integer unit, which supports program logic and integer arithmetic, are divided into 8 global registers and a register file. The register file is windowed, i.e. at one moment the application has access to a small fixed sized portion of that register file, the "window". There are machine instructions `SAVE` and `RESTORE` that move the window up and down within that file. Two neighbour windows share 8 registers, which can be used for passing of parameters. If the callee saves, i.e. the function called starts with a `SAVE` instruction and ends with a `RESTORE` instruction, the contents of the registers in the (safe part of) the window is saved across a call. The register file is organized in a circular manner, see e.g. [4]. If the register file is exhausted, a trap is generated and the operating system will save the contents of old windows to the stack and restore them from there if the register file underflows, invisible for the user. At runtime, one can use the register file as if it were almost infinite.

Another view of the same thing is that the top of stack is held in fast memory, i.e. the top of stack is cached.

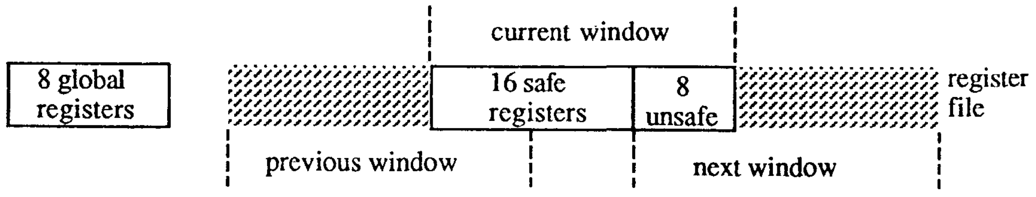


Figure 1: Generalized register layout for the SPARC processor integer unit.

2.2 SPARC Logic and Arithmetic Instructions

On SPARC, most instructions operate on three registers or two registers plus one immediate operand, e.g:

```
                add  r1,r2,r3  resp  add  r1,17,r2
                 or  r1,r2,r3  resp   or  r1,255,r2
```

Many logical operations are provided, including operation which negate the second argument, e.g. `andn  r1,r2,r3` which 'and's the contents of register `r1` and the 1–complement of `r2` to `r3`. There is an arithmetic shift but no circular shift. All shift operations are end off.

The integer arithmetic provides `add` and `subtract`, but not multiply or divide. These have to be emulated in software. This will slow down SPARC performance, if an application uses the integer multiplication or division heavily (e.g. from Computer Algebra).

SPARC architecture includes a tag scheme, which is interpreted by special arithmetic instructions `tadd...` and `tsub...` for add and subtract. This tag scheme uses two low value bits and identifies an integer by two zero bits in the tag field; other types can be defined by the application program. The instructions for 'tagged' add and subtract ensure that the operands and the result are integers in the sense of this tag scheme. The Processor Status Register is changed if the source operands are not integers or if the result overflows, and the application program can handle this case. This way it is possible to handle the generic add and subtract operations for small integers completely in line using two instructions. Unfortunately, the tag scheme of this PSL implementation does not allow to use these tagged integer instructions. The tag scheme is described in Section 3.1.

2.3 Delay Slots after Branch or Call Instructions

Because of pipelined instruction issue, the instruction after a branch instruction is already ready to issue and will be executed if not prevented. The slot behind a control transfer instruction is called *delay slot*. For example, in the instruction sequence

```
        ConditionalBranch  label
        nextinst 1
        ...
label   nextinst n
```

the instruction nextinst1 will be issued regardless whether the Conditionalbranch is taken or not. A "naive" compiler has to put in no-op instructions into the delay slots. There are alternative conditional jump instructions with ",a" added to the original name, which annul the delay slot instruction if the jump is not taken. For example, in the instruction sequence

```
             ConditionalBranch,a  label
             nextinst 1
             ...
      label  nextinst n
```

nextinst1 is evaluated only if the branch to label is taken. However, in this form at least one cycle is lost.

2.4 Memory Access

The instructions for data transfer between memory and registers have two formats:

```
ld  [ r1 + r2 ] ,       r3
ld  [ r1 + simm13] ,    r3 r1  may be g0(= 0)
```

Here simm13 is a sign extended 13 bit immediate value, i.e. when addressing memory, the offset is limited to 4k bytes. To load or store constant memory locations requires that a (temporary) register must be filled with the constant address. Such a constant memory access takes at least two instructions. Sacrificing one global register (as recommended by the Architecture Manual [8]), allows a memory area of 8k bytes to be addressed by one instruction. This can be used for fast access to often used data. As these variables are often referenced, this area will most probably reside in memory cache.

Example. Loading from 123450 (assume %g7 as constant pointer = 123400)

```
         normal code:                          improved:

sethi    %hi(123450),r10              ld    [%g7 + 50],r10
ld       [r10 + %lo(123450)],r10
```

%hi and %lo are assembler macros, which separate the high and the low part of an address.

3. PSL Model for the SPARC

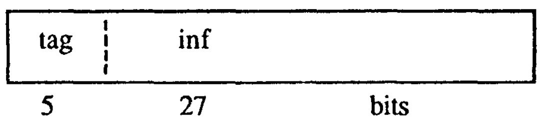
In this section, the actual mapping of PSL structures to the components of the SPARC processor is decribed. It can be expected that this mapping needs further refinement, e.g. if the ratio between memory access time and cycle time will change for newer SPARC systems.

3.1 Allocation of PSL Data Structures

The mapping of PSL structures to the system hardware must define the LISP item, the locations for PSL's 'real registers', the temporary register and the frame (local variables of a function).

3.1.1 The LISP Items

The basis data structure of LISP is the item. In the actual SPARC PSL implementation, the item consists of a *tag part* and an *inf part.* The tag part consists of 5 bits in the most significant part of a word and the inf part of 27 bits. The tag field defines the interpretation of an item. The tag field size allows 31 different data types immediately discriminated by tag inspection. The inf part contains either a direct operand, e.g. a small number, or an address. The size of the inf field restricts the address space for LISP data to 128 M bytes.



Examples:

tag = 0 , inf = nn	represents a positive small integer
tag = 4 , inf = addr	points to a string starting in addr
tag = 9 , inf = addr	points to a pair of items in addr
tag =31 , inf = nn	represents a negative small integer

The importance of the logical and shift instructions for LISP implementations follows from the need to have fast access to tag and inf part. The fast construction of items may also be important, dependent on the application.

3.1.2 A Short Discussion on "high or low" Tags

With "high tags", we do not mean high tags in the PSL sense, but the fact that tags are allocated in the high value bits of a word. Other LISP implementations for the SPARC use low tags instead of PSL's high tags. "Low tags" means that the tag area within an item is in the lowvalue bits. Advantages on the SPARC are obviously the ability to use the tagged integer instructions and the easy tag stripping operation (which is simply a right shift, if tag stripping is needed at all) and some technical fascinating things that `NIL` and T may be direct operands in operations and so no register with the value `nil` is needed.

What held us back was the great overhead involved by a more complicated access to the tag information. An implementation with low tags implies that the tag information must be splitted into 3 low tag bits in the items least significant bits and additional information elsewhere. The tag information is used for such processes like garbage collection and arithmetic. The fast access to tags is of greatest importance for fast discrimination of LISP items and for the 'bread and butter' predicates like pairp (consp), atom, idp and so on.

Nevertheless, the technical interesting features of low tags will keep the process of discussion on that topic open for a while.

3.1.3 Registers

PSL uses up to 15 'logical registers' for parameter passing between caller and callee. These PSL registers do not need to be real hardware registers, but it is recommendable to have the Registers 1–5 as real registers. Since SPARC has so many real registers, PSL's registers 1–5 are allocated as real registers, namely in the global registers $g1 \ldots g5$, because the PSL registers must be identical for caller and callee. Thus the passing of parameters is different from the usual calling sequence of the SPARC operating system, which passes parameter through the overlapped register windows (see Section 2.1). The PSL registers 6–15 are allocated in memory. They are used when a function has more than 5 parameters, which is a rare case in LISP. The 'logical registers' Reg 6 ... Reg 15 are located such that they can be loaded with one instruction (see 3 below).

There are some special register allocations for global registers:

- Register $g0$ is a NULL register by hardware.
- Register $g6$ is Reg `NIL`, i.e. it contains the most often needed value .
- Register $g7$ points into a special area where often used values are kept, e.g. Reg 6 ... Reg 15, for memory management and the pointers to PSL's main structures, as described in Section 3.4.

3.1.4 Usage of the Register File

As explained in Section 2.1, the view of a function to the register file is limited to a window of 24 Registers (14 safe + 6 unsafe + linkage information and stackpointers).

It is a natural approach to put local variables (up to 12) into the safe registers. The first safe register will keep the length of the frame. Since the windows are pushed down to memory in multiples of 16 words, the information about the length of a frame is essential for functions that have to inspect the stack. Such a function, e.g. the garbage collector, has to discriminate between 'living' items which belong to the dumped frame and superfluous (random) information that is not used as frame in a register. Since LISP relies on the interpretation of tagged items, a 'dirty' item picked randomly may produce a fatal error. Therefore, the stack has to be inspected with extreme care. Moreover, in some cases, standard PSL does not preset all frame items, this way it was possible to pick random items from the frame. For the Sparc PSL implementation a pass on the generated code was added, which inserts instruction that preset the undefined frame locations. When the window is dumped to stack, a frame with up to 12 elements looks like:

lng	Frame elements	inf mask	f.p	retaddr
x	1 2 3 4 5 6 7 8 9 10 11 12	x	x	x

f.p. = old stack pointer, retaddr = return address

inf mask = special mask for (inf operation

A frame with more than 12 elements is a rare case in LISP. The frame items past the twelfth are allocated conventionally on the stack.

For comparison, the frame structure for Cray PSL: The top frame is held in a block of T–registers, which can be swapped into memory by a Blockstore/Blockload instruction by the user program. The usage of T–registers as fast intermediate memory is of great impact on the performance, especially for the Cray, since the ratio between memory access and register access is about 15 versus approx. 2 on the SPARC.

lng	retaddr	Frame elements
x	x	1 2 3 4 ... ... 33 34

The temporary registers are allocated in the 6 unsafe registers, (in special T–Registers on a Cray). The T–register on the Cray cannot be used as operands for logical or arithmetic operations, so we have to put the frame items into a 'real' register first.

Since the frame registers are not specialized, they can be used for arithmetic and logical operations directly.

This implies that the often used constructs:

```
(prog (x cnt list))          % generated cmacros
. . . .
(setq x (car list))          (*More (car (frame1)) (frame 3))
(setq cnt (+ cnt 1))         (*WPLUS2 (frame 2))  (wconst 1))
. . . .
(setq list (cdr list))       (*More (cdr (frame 3)) (frame 3))
```

may work directly with the frame registers as operands without using an additional register.

3.2 SPARC Specific Optimization Pass for the Assembly Phase

The special behaviour of the SPARC instructions, especially the delay slots in conditional jumps, caused the need for an additional optimization pass for the PSL compiler. Conditional jumps are very often used in LISP code. The delay slots of the conditional jumps must be filled with no-op instructions by the compiler first. These no-op instructions will slow down the performance, if they cannot be filled with useful instructions. The fill of the delay slot must be the very last phase before the instructions are generated by the PSL compiler.

For example, the LISP code

(hugo NIL) % call hugo with one parameter = NIL

will first be generated as

```
(mov (reg NIL) (reg 1))
(call hugo)
(nop)                         % delay slot
```

and is then rearranged as:

```
(call hugo)
(mov (reg NIL) (reg 1))   % delay slot
```

since the `(mov (reg NIL) (reg 1))` can be issued in the delay slot of the call instruction. This optimization cannot be done on cmacro level already, since the cmacro *link does not have any insight into the surrounding cmacros. This is a typical context sensitive peephole transformation.

Another case: the conditional code

```
        (when (> a b)  ..... )
        (hugo nil nil)
```

is assembled and optimized as:

	original		improved
	`(bg,a  label)`		`(bg,a newlabel)`
	`(nop)`		`(mov (reg nil) (reg 1))`
	`.....`		`.....`
`label`	`(mov  (reg  nil) (reg 1))`	`label`	`(mov (reg  nil) (reg 1))`
	`(mov (reg 1) (reg 2))`	`newlabel`	`(mov (reg 1) (reg 2))`
	`(call hugo)`		`(call hugo)`
	`.....`		`.....`

(`bg,a` is branch on greater and annul the delay slot instruction, if jump is not taken).

The insertion of a new label allows to copy the instruction at target 'label' to the delay slot, in case that the instruction is movable, i.e. is not a control transfer itself. It may turn out that the instruction at label is unreachable because an unconditional jump is situated just before the address label. In order to save code space, this instruction is deleted in this case.

As example, we look at the code of the function `memq` from the introduction in original and optimized form.

```
(de memq (u v)
  % EQ version of Member
  (cond ((not (pairp v)) nil)
        ((eq u (car v)) v)
        (t (memq u (cdr v)))))
```

This function is naively compiled into the following (19) instructions:

```
             L0003:
  9930A01B    srl       r2,x'1B,r12
  80A32009    subcc     r12,9,r0              pair test
  22800005    be,a      L0004
  01000000    nop
  82100006    or        r0,r6,r1              return
  81C3E008    jmpl      r15+8,r0
  01000000    nop
```

```
          L0004:
9628801D    andn      r2,r29,r11            (car operation
D802E000    ld        [r11+0],r12
80A0400C    subcc     r1,r12,r0             (eq test
32800005    bne,a     L0005
01000000    nop
82100002    or        r0,r2,r1              return
81C3E008    jmpl      r15+8,r0
01000000    nop
          L0005:
9228801D    andn      r2,r29,r9             (cdr operation
C4026004    ld        [r9+4],r2
10BFFFEF    ba        L0003                 recursive call
01000000    nop
```

The code is optimized on instruction level to the following (15) instructions:

```
9930A01B    srl       r2,x'1B,r12
          L0001:
80A32009    subcc     r12,9,r0
22800004    be,a      L0002
9228801D    andn      r2,r29,r9
81C3E008    jmp      r15+8,r0
82100006    or        r0,r6,r1
          L0002:
D8026000    ld        [r9+0],r12
80A0400C    subcc     r1,r12,r0
32800004    bne,a     L0003
9428801D    andn      r2,r29,r10
81C3E008    jmpl      r15+8,r0
82100002    or        r0,r2,r1
          L0003:
C402A004    ld        [r10+4],r2
10BFFFF4    ba        L0001
9930A01B    srl       r2,x'1B,r12
```

To demonstate the improvement, we run a little test series:

Testcase1 :	100000 times	`(memq 17 '(17 20 21))`	`% find immediately case`
2 :	100000 times	`(memq 22 '(17 20 21))`	`% not found case`
	original	optimized	
Testcase1 :	70 ms	70 ms	
2 :	340 ms	255 ms	

As one may expect, the difference in the "immediate found" case is below the measurement tolerance, but the "not found" case is significantly faster (about 25 % less cpu time)

3.3 Pseudo Registers

The Pseudo Registers are an implementation corresponding to the hint in Architecture Manual [8]. The global register *g7* is used as a pointer into a pseudo register area, whose elements are special heavy used value cells e.g. SYMFNC, SYMVAL etc. Those variables are held in *B* and *T* registers (or Local Memory) in the Cray implementations to guarantee fast access. On the SPARC architecture these cells are accessible with one instruction and because the memory is cached on SPARC such a pseudo register, it is most probably contained in cache.

3.4 Miscellaneous

While porting PSL to the Cray, we learned a lot about what one can optimize in PSL with a great profit rendered, and we also developed tools for analysing behaviour and performance of PSL. We, therefore, tried to adapt useful features from the Cray implementation to the SPARC version:

- The functions cons, ncons and xcons (the basic constructors for LISP) are compiled in line.
- The functions equal, eval, list1 .. list5 and parts of the garbage collector have been (partially) handcoded.
- Speed-up for get operations for REDUCE's most often used properties.
- Redesign of the dispatch for generic arithmetic.
- Disassembler for SPARC code.
- Overall cpu-time analyse using the UNIX profil feature.
- Tools for determination of the number of calls and the time consumption of a LISP function.

4. Further Developments

One can think of some further things to improve for SPARC:

- The discussion on high tags vs. low tags is still open. It may turn out that low tags are of importance to achieve a next speed-up.
- Based on the standard PSL version an improved bignum package is under development in ZIB. It will be usable on 32 bit systems like SUN3 or SUN4 and will speed-up the bignum computation.
- Not all optimizations that are in effect on the Cray (i.e. the lap optimization passes) are ported to the SPARC yet.

Appendix

Actual Timings (May 1989)

The results achieved with the current SPARC PSL are compared with another LISP implementation (Allegro CL), which were published by Franz Inc. [7] and the Cray PSL version. It is noteworthy that the Cray version reaches its peak performance when bignums (infinte precision integers) are involved, which is not true in any of the following tests except for the Groebner Test Suite from REDUCE Netlib. PSL Timings were taken on SUN4-260 in ZIB and for Cray X-MP at Cray Research Inc. in Mendota Heights. Franz Inc. ran the tests on a SUN4-260.

Some tests from the Computer Algebra system REDUCE:

REDUCE standard test sequence:

SPARC PSL	Cray X-MP PSL
5.3 sec	1.0 sec

Excalc package test:

SPARC PSL	Cray X-MP PSL
40 sec	10 sec

Groebner package test:

SPARC PSL	Cray X-MP PSL
20 sec	3.5 sec

Gabriel's benchmarks [6]:

Benchmark	SUN4 (PSL)	CRAY X-MP(PSL)	SUN4 (Allegro CL)
boyer	2567	708	3316
browse	5525	1066	4100
ctak	782	213	450
dderiv	918	289	1050
deriv	765	259	667
destruc	374	100	366
div2-iter	306	73	250
div2-recur	1428	105	1316
fft	3757	2421	800
fprint	731	58	283
fread	391	121	867
puzzle	1462	680	1400
stak	527	110	817
tak	885	330	750
takl	799	196	583
takr	119	65	133
tprint	153	40	400
trav-init	1581	692	1066
trav-run	6290	3043	5400
triang	21641	9769	21416

All Times in Milliseconds

Computer Descriptions:

SUN4/260, 32 MB Main Memory, 50 MB Heap Space
Portable Standard LISP Version 3.4.

CRAY X-MP/416 UNICOS 5.0, Dedicated Mode, 1 CPU,
Portable Standard LISP Version 3.4, 8MW PSL Heapsize.

SUN4/260, 16 MB Main Memory, 45 MB Swap Space, Allegro Common LISP 3.0
Data Source: Franz Inc., Publication

Acknowledgement

We like to thank Anthony C. Hearn, Tsuyoshi Yamamoto, Chris Burdorf (The RAND Corporation, Santa Monica) and Julian Padget (University of Bath) for starting the migration of PSL to the SPARC and for making their results available for us. Denton Olson (CRAY Research Inc.) ran the tests for the Cray-XMP.

References

[1] J. W. Anderson, W. F. Galway, R. R. Kessler, H. Melenk, W. Neun: *Implementing and Optimizing LISP for the Cray.* IEEE Software (July 1987)

[2] A. C. Hearn: *REDUCE User's Manual, Version* 3.3. The RAND Corporation (1987).

[3] M. L. Griss & A. C. Hearn: *A Portable LISP Compiler.* Software Practice and Experience, Vol. 11 (1981)

[4] R. B. Garner: *Scalable RISC Architecture in SunTechnology.* Sun Microsystems Inc., Vol. 1, No. 3 (1988).

[5] S. S. Muchnick: *Optimizing SPARC Compilers in SunTechnology.* Sun Microsystems Inc., Vol. 1, No. 3 (1988).

[6] R. P. Gabriel: *Performance and Evaluation of LISP systems.* MIT press (1985).

[7] Franz Inc.: *Common LISP Benchmarks on a SUN*4/260. Franz Inc., Berkeley, California.

[8] Sun Microsystems Inc.: *The SPARC Architecture Manual.* SUN Inc., Mountain View (1987).