ON THE TIME REQUIRED FOR RETENTION*

D. M. Berry† L. Chirica† J. B. Johnston§ D. F. Martin† A. Sorkin†

1.1 INTRODUCTION

In block structured languages, upon entry to a block or procedure, storage is allocated for the identifiers declared in the block or procedure. There are two choices as to when to deallocate this storage:

1) upon exit from the block or procedure,
2) when the storage becomes inaccessible.

The former is referred to as the deletion strategy because storage for a block or procedure is deleted (deallocated) automatically upon exit. The latter is referred to as the retention strategy because it is possible for storage for a block or procedure to be retained after exit. These two strategies are not equivalent, as there exist numerous example programs making use of pointer, label, and procedure values, which show the difference between these two strategies. For examples, see Bry71b, CDMPS 73, Fis72, Org73, and Weg71. The difference shows up in the form of a dangling reference in the deletion strategy which does not appear in the retention strategy.

In the deletion strategy, storage for blocks and procedures is allocated and deallocated in a last-in-first-out order. Therefore, storage for blocks and procedures can be managed efficiently on a pushdown stack. However, deletion gives rise to dangling references in the use of pointer, label, and procedure values. This is dealt with in one of two ways which compromises either security or generality:

1) There is no attempt to prevent or detect dangling references, and errors, which are hard for the programmer to detect, can occur. This is the case in PL/I-F in which a pure stack model with no run time check can be used [W1k69].
2) There are restrictions on the use of pointer, label, and procedure values which require the addition of some run time checks to the basic stack model. This is the case with ALGOL 68, in which the run time checks are to prevent dangling references, and with Deletion Parallel Euler, in which the run time checks are to detect dangling references [vWn 69, CDMPS 73].

Retention is both more secure and more general in that it eliminates all possibility of dangling references, and it therefore removes all restrictions on the use of pointer, label, and procedure values. This is the case with GEDANKEN, PAL, Oregano, and Retention Parallel Euler [WE 70, Rey 70, Bry 73, CDMPS 73]. However, retention is considered to be less efficient than deletion, as it requires more sophisticated storage management techniques, including the use of a free list, reference count management, and/or garbage collection.

While retention is clearly nicer, its inefficiency appears to be the major obstacle to its general use. Most programs that are written are single task programs which use integer, real, Boolean, and character string values almost exclusively. They use pointers only for passing parameters by reference, and they use labels and procedures only as constants. Furthermore, these procedures return only integer, real, Boolean, and character string values. These programs do not require retention and run correctly on an implementation using the deletion strategy. We call such programs which run correctly on a deletion implementation, e.g. the stack model, well-stacked (WS). It is only the rare program doing list manipulation or demonstrating a principle of computer science, etc., which requires retention. It seems unfair to penalize the vast majority of programs which are well-stacked with the overhead required for retention.

This paper develops an implementation of generalized (single task)* block structured languages with compile time type determinability, which is capable of handling full retention but which keeps the run time cost of many well-stacked programs near that of the same program on a typical ALGOL 68 style implementation.

This implementation can be regarded as the architectural basis for a block structured machine. Indeed the implementation is described in terms of microcode for an abstract machine.

*This work was supported by the National Science Foundation, Grant Nos. GJ 809 and GJ 28074.

†Computer Science Department, University of California, Los Angeles, Los Angeles, Ca. 90024.

§Computer Science Department, New Mexico State University, Las Cruces, N. M. 88003.

*Although the implementation appears to be easily extendable to handle multi-tasking.

Other similar and independent work with the same goals has been done by Bobrow and Wegbreit [BW73]. They propose an implementation which differs from ours, primarily in two places:

1) The heavy copying overhead of their model occurs with some of the block exits and procedure returns, whereas ours places the heavy copying overhead on the label bindings and gotos.
2) Their model is general enough to handle LISP as well as ALGOL identifier binding and assumes run time type checking. Ours is restricted to languages with ALGOL binding and compile time type checking. These restrictions permit certain optimizations.

This paper assumes familiarity with the following topics:

1) Retention vs. deletion and dangling references [Bry71a, CDMPS73, Weg71]
2) Stack model [RR64]
3) Contour model [Joh71, CDMPS73]
4) ALGOL 68 and its implementation [vWn69, Bry70, BL70, Pck70, Weg71].

1.2 STRATEGY OF THIS PAPER

A generalized block structured language is defined to have only the "interesting" features with the usual block-structured and compile-time-type-checking semantics. First, its concrete syntax is given followed by an abstract syntax representing a partially translated and pruned parse tree, in which it is assumed that all type compatibility and other context conditions have been checked.

Two machines are defined which have identical instruction repertoires. One of these is the Lifetime Stack Machine (LSM) which is a variant of a typical abstract machine used for ALGOL 68 [BL70, Bry70, Weg71]. The machine is a stack machine coupled with run-time lefetime checks on the assignment of pointer, label, and procedure values which prevents potential dangling references. The run time overhead incurred for the lifetime checks appears to be an acceptable cost for preventing dangling references while getting some of the general use of pointer label and procedure values.

The second machine is our proposed implementation of retention, the Partial Reference Count Contour Machine (PRCCM) which is a variant of the contour model as defined by Johnston [Joh71] and as extended by Chirica, Dreisbach, Martin, Peetz, and Sorkin [CDMPS73].

The definition of each machine consists of a description of a machine state and microcode describing the execution of each instruction in the execution of each instruction in the common machine language.

This is followed by a scheme for translating pruned parse trees of programs into machine language programs. This translation effectively defines the semantics of the various language features in each machine.

By knowing how much time is required for each instruction by each machine, and knowing which instructions implement each source language feature, it is possible to obtain comparative time estimates for the two machines, as a function of the features appearing in a program.

It is proved that a large subset of the well-stacked programs (which is to be defined precisely) will run on PRCCM in almost the same time as on the LSM. The main idea is to take the run time overhead required for LSM's lifetime checks (which prevent potential dangling references in the context of deletion) and use this time to do some reference count management in PRCCM*.

SPECIAL NOTE: Due to space limitations, it will not be possible to give the machine instructions, their microcode definitions, the translation of GBSL programs to machine language, and the proofs of the theorems in this version of the paper. For a complete copy of the paper, the interested reader should write to the first co-author.

2. GBSL

GBSL, the generalized block structured language is a powerful but simple language capturing all of the "interesting" features and ignoring those not thought to be relevant to the results.

2.1 FEATURES

GBSL has five types of values (modes), integer, Boolean, pointer, label and procedure values. All of these values may be assigned freely to any variable of the same mode. The basic value returning entities are constants, nils, and identifiers. The operations of the language include binary and unary arithmetic and Boolean operators, assignment, pointer computation (PL/I ADDR), pointer indirection (ALGOL 68 dereferencing), procedure calls, and gotos. The statement grouping facilities include conditionals, blocks, and procedure bodies, the last of which may be assigned to procedure valued variables.

Blocks contain declarations declaring identifiers to be variables of any mode. Statements may optionally be labeled by label constant identifiers which are assumed to be declared in the inner most containing block. Procedure bodies contain declarations of formal parameters of any mode. Actual parameters are passed by value in the ALGOL 60 [Nau63] sense so that formal parameters are assignable variables.

2.2 SYNTAX

The following is a context-free grammar giving the concrete syntax of GBSL. Lower-case hyphenated words are non-terminals, and upper case words, digits, and punctuation except for "→" and "|" are terminals:

```
program→block
block→BEGIN declaration-part statement-part END
declaration-part→ε | declaration-sequence
```

*This way of describing the idea of the proof is due to Henry Bowlden [Bow71].

```
declaration-sequence→declaration ; |
          declaration ; declaration-sequence
declaration→mode identifier
mode→basic mode | pointer-mode | procedure-mode
basic-mode→INT | BOOL | LABEL
pointer-mode→PTR mode
procedure-mode→PROC ( mode-list ) mode
mode-list→mode | mode , mode-list
statement-part→optionally-labeled expression |
        optionally-labeled statement ;
        statement-part
optionally-labeled→identifier : | ε
statement→goto | expression
goto→GOTO expression
expression→binary | unary | assignment |
         address-of | indirection | call |
         conditional | block | procedure-
         body | constant | nil | identifier |
         ( expression )
binary→expression binary-operator expression
unary→unary-operator expression
binary-operator→+ | = |...
unary-operator→-1 ...
assignment→left-part ← expression
left-part→identifier | indirection
address-of→ADDR identifier
indirection→expression IND
call→expression ( expression-list )
expression-list→expression | expression ,
          expression-list
conditional→IF expression THEN expression
         ELSE expression FI
procedure-body→PROC declaration-part
          expression CORP
constant→integer | boolean
integer→0 | 1 | 2 | ...
boolean→TRUE | FALSE
nil→NIL mode
identifiers→... (the usual set of identifiers)
```

The grammar is ambiguous; however, following Scott and Strachey [SS72] we observe that by sufficient use of parentheses it is possible to disambiguate any program.

2.3 CONTEXT CONDITIONS

In any block a procedure a given identifier may appear in at most one declaration of any kind.

The usual block-structured scope rules hold. The scope of a declaration of an identifier is the entire block or procedure in which the declaration occurs minus any internal blocks or procedures in which the same identifier is redeclared. A use of an identifier identifies the declaration of the same identifier in whose scope it lies. Every use of an identifier must identify some declaration.

The declaration of an identifier associates a mode with the identifier. This mode is either INT, BOOL, LABEL, a pointer-mode or a procedure-mode. A pointer-mode specifies the mode of the object pointed to, and a procedure-mode specifies the ordered list of parameter modes and the mode of the returned result.

All expressions return some value whose mode is computable at compile time from the mode of the immediate constituents. Sometimes there are restrictions on the mode of the constituent. The mode computation and restrictions are described in the table below. To the left of the arrow is the expression form with allowable modes in place of the operands and to the right of the arrow is the result mode.

binary +	INT + INT ⇒ INT
=	m = m ⇒ BOOL, for any mode m
unary -	- INT ⇒ INT
assignment	m ← m ⇒ m, for any mode m
address of	ADDR m ⇒ m, for any mode m
indirection	m IND ⇒ n, if m = PTR n for any mode n
call	P(m_1, ..., m_j) ⇒ m, if p = PROC (m_1, ..., m_j)m, for any modes m_1, ..., m_j, m with j≥1
conditional	IF BOOL THEN m ELSE m FI ⇒ m, for any mode m
block	BEGIN ...; m END or BEGIN ...; id: m END ⇒ m for any mode m
procedure body	PROC m_1 id_1; ...; $m_j id_j$; m CORP ⇒ PROC(m_1, ... m_j)m for any modes m_1, ...m_j, m, with j≥1
integer	INT
boolean	BOOL
nil	NIL m ⇒ m for any mode m
identifier	the mode m of the declaration identified by the identifier

A goto is in a class all by itself. Its argument must be an expression of mode LABEL, but the goto itself does not return a value and it has no mode.

To make the mode of all expressions computable at compile time, a goto has been excluded from appearing as an arm of a conditional, as the last statement of a block, and as the expression of a procedure body. A goto can, however, appear as a non-last statement of a block which is, or is in, any of these constructs.

2.4 ABSTRACT SYNTAX

The following abstract syntax [LW69] defines the set of partially pruned parse trees of concrete GBSL programs in which all context conditions are satisfied and in which

1) goto's to labels within the same block are converted to hopto's
2) label constants not referred to or referred to only by hopto's are eliminated
3) the remaining label constants are converted into implicitly declared label variables which are initialized to full-fledged label values on entry to the declaring block.

```
1.  is-program=is-block
2. *is-block=(<s-decl-part:is-decl-list>,<s-st-
    part:is-st-list^<is-expr>>)
3.  is-decl=(<s-id:is-id>,<s-mode:is-mode>
4.  is-mode=is-basic-mode∨is-ptr-mode∨is-proc-mode
5.  is-basic-mode={INT,LABEL,BOOL}
6.  is-ptr-mode=(<s-ptr:is-mode>)
7.  is-proc-mode=(<s-params:is-mode-list>,<s-ret:
    is-mode>)
8.  is-st=is-goto∨is-hopto∨is-expr
9.  is-goto=(<s-goto:is-expr>)
```

```
*(is-p-list^<is-q>)(t)=df is-p-list(t) &
                          is-q(elem(length(t),t)) &
                              is-q⊂is-p
```

```
10. is-hopto=(<s-hopto:is-int>)
11. is-expr=is-binary∨is-unary∨is-assign∨is-addr-
          of∨is-indirection∨is-call∨is-cond∨is-
          block∨is-proc-body∨is-const∨is-nil∨is-
          label-const
12. is-binary=(<s-rd1:is-expr>,<s-rd2;is-expr>,
          <s-op:is-bop>)
13. is-bop={+,=,...}
14. is-unary=(<s-rd:is-expr>,<s-op:is-uop>)
15. is-uop={-,...}
16. is-assign=(<s-lp:is-id∨is-indirection>,
          <s-rp:is-expr>)
17. is-addr-of=(<s-addr:is-id>)
18. is-indirection=(<s-ind:is-expr>)
19. is-call=(<s-proc:is-expr>,<s-params:is-expr-
          list>)
20. is-cond=(<s-cond:is-expr>,<s-then:is-expr>,
          <is-else:is-expr>)
21. is-proc-body=(<s-params:is-decl-list>,
          <s-ret:is-mode>,<s-body:is-expr>)
22. is-const=is-int∨is-bool
23. is-int=...
24. is-bool={TRUE,FALSE}
25. is-nil=(<s-nil:is-mode>)
26. is-label-const=(<s-label:is-int>)
27. is-id=...
```

2.5 ASSUMPTIONS

The partial translation inherent in the abstract syntax includes:

1) Ascertaining that the scope rules have been obeyed,
2) Ascertaining that all expressions are used consistently with their modes,
3) Each goto, g, whose argument is a label constant declared in the block, b, in which g appears is replaced by the hopto

   μ_0(<s-hopto:j>)

   where j is the index of the statement part of b.
4) For each block, b, the following is done: Let ℓ_1, ..., ℓ_n, with $n \geq 0$, be all of the label constants in b which are still referred to (after (3) has been done).
   a) The declaration-list
      <μ_0(<s-id:ℓ_1>,<s-mode:LABEL>)>
      ^ ...
      ^<μ_0(<s-id:ℓ_n>,<s-mode:LABEL>)>
      is concatenated to the declaration part of b.
   b) The statement-list
      <μ_0(<s-lp:ℓ_1>,<s-rp:μ_0(<s-label: j_1+n>)>)>
      ^ ...
      ^<μ_0(<s-lp:ℓ_n>,<s-rp:μ_0(<s-label: j_n+n>)>)>
      where j_i is the index within the statement part of b of the statement labeled ℓ_i, is concatenated to the beginning of the statement part of b.
   c) The argument of all hopto's in the statement part of b is incremented by n.

SECTIONS 3 and 4

The following two sections which define the two machines are given in parallel with the definition of the LSM in the left column and the definition of the PRCCM in the right column. This organization is both to take advantage of similarities and to point out the differences. When the two machines agree completely, the description runs across both columns. When they differ, the description is split into two columns.

—Continue reading at the top of the right-hand column

3. THE LIFETIME STACK MACHINE

The Lifetime Stack Machine (LSM) is an extension of the basic stack model for block structured languages [RR64]. The additional overhead in the LSM is to implement the lifetime checks (scope checks) suggested by the ALGOL 68 Report [vWn69].

4. THE PARTIAL REFERENCE COUNT CONTOUR MACHINE

The Partial Reference Count Contour Machine (PRCCM) is a variant of the basic contour model as defined by Johnston [Joh71] and as modified by Chirica et al [CDMPS73]. The variations include facilities for maintenance of reference counts and partial reclamation of contours whose reference counts indicate that they are inaccessible.

3.1 WORDS

4.1 WORDS

The basic unit of storage in both machines is the word, whose length is left unspecified. Subfields of the word are specified as follows:

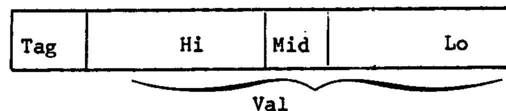

Figure 1

There are four disjoint fields called tag, hi, mid and lo. Their sizes are unspecified. The relative sizes given in the diagram above are intended to be suggestive, as tag and mid will have to hold values in a smaller range than those held by hi and lo. Hi and lo will have to hold addresses of other words. The hi, mid, and lo fields may be selected as a contiguous group by use of the field name val.

Each value is assumed to occupy 1 or 2 words. The word's tag field indicates what type and what part of a value is stored in the word. The tag of a word is maintained but not checked during a computation; that is, an assignment of a value to a word sets the tag of the word to the tag associated with the value regardless of the tag that was there before. There is no need to check tags during the execution of in-

structions because of the assumption of compile-time mode checking. However, the tag is needed by the storage management system and the garbage collector.

3.2 COMPONENTS

4.2 COMPONENTS

The major components of the model are the algorithm and the record of execution. See Figures 2 and 6.

3.2.1 Algorithm

4.2.1 Algorithm

The algorithm consists of a fixed reentrant program composed of polish style instructions to be defined later. The polish style code goes hand-in-hand with the pushdown stack expression evaluation used in the machine.

3.2.2 Record of Execution The record of execution consists of a single pushdown stack and a processor.

3.2.2.1 Pushdown Stack The pushdown stack contains interleaved activation records (AR's) and mini-expression stacks (mes's).

3.2.2.1.1 Activation Record An AR is pushed into the stack upon entry to a block or procedure. The AR is said to be a descendent of the block or procedure, and the block or procedure is said the AR's antecedent. The same AR is popped on exit from the block or procedure.

An AR is organized as shown in Figure 3. The 0th word of an AR, A, is an AR control word with tag ARC. Its components are:

1. In the hi field is the static link, sl, pointing to an AR which is a descendent of the block or procedure nested about A's antecedent. If A's antecedent is a procedure then the sl is a copy of the ep of the called procedure value.
2. In the lo field is the dynamic link, dl, pointing to the AR immediately below A on the stack.
3. In the mid field is the nesting height, nh, giving the static nesting height of A's antecedent. The outermost block is at nesting height 0 and each successive inner block or procedure is of height one greater.

If A's antecedent is a procedure then the word 1 contains a return label with tag RTL whose components are:

1. In the lo field, an instruction pointer, ip, pointing to the instruction to be resumed with upon return from the procedure.
2. In the mid field, a nesting height, nh, which is one more than that of the AR pointed by A's dynamic link.

The remaining words of the AR, i.e., words 1 through n if the antecedent is a block, and words 2 through n in the antecedent is a procedure, contain the subcells for the identifiers declared in the antecedent.

In Figure 2 the stack has three AR's of nesting heights 1, 1, and 0 (from top to bottom). The top AR's antecedent is a procedure because it has a return label. The other two have blocks as antecedents.

3.2.2.1.2 Mes's Above each AR is a mini-expression stack. Each mes serves as an expression evaluation stack for the expressions of the antecedent of the AR just below it. Each mes represents the status of the expression stack at the time the antecedent of the AR just below it. Each mes represents the status of the expression stack at the time the antecedent of the AR just above it was entered or

4.2.2 Record of Execution The record of execution consists of a contour segment, an expression stack, a stack segment, and a processor.

4.2.2.1 Contour Segment The contour segment consists of two doubly linked lists:

1. One is a list of allocated contours.
2. The other is a list of free blocks of memory.

Both lists are ordered by address. The first two words of the segment contain the list heads for these two lists. Both list heads have LH tags and each consists of:

1. in the hi field, a forward link, fl, and
2. in the lo field, a backward link, bl, which points to its first and last elements respectively.

4.2.2.1.1 Contour A contour is allocated upon entry to a block or procedure. The contour is said to be a descendent of the block or procedure and the block or procedure is said to be the antecedent of the contour. The contour is deallocated only after it has become inaccessible.

The general form of a contour is shown in Figure 7. The -1th word of a contour, C, has a tag of LNK and consists of storage management information including:

1. in the hi field, a forward link, fl, and
2. in the lo field, backward link, bl, for linking the contour into the doubly linked list of contours,
3. in the mid field, a length, len, which gives the total length or the contour in words.

Word 0 has a CT tag indicating that it is the base word of a contour. It consists of:

1. in the hi field, a static link, sl, which points to a contour which is a descendent of the block or procedure nested about C's antecedent. Specifically, if C's antecedent is a block, then the sl points to the contour pointed to by the processor's ep just before entry to C's antecedent. If C's antecedent is a procedure, then the sl is a copy of the called procedure value's ep,
2. in the mid field, a nesting height, nh, giving the static nesting height of C's antecedent. The outermost block is at nesting height 0 and each successive inner block or procedure is of height one greater.
3. in the lo field, a reference count, rc, counting the number of ep's and static links pointing to the countour except for those that reside in the countour itself.

If the contour's antecedent is a procedure, then word 1 contains a return label which has a tag of RTL and which consists of three components:

1. In the lo field is an ip pointing to the instruction to be resumed with upon return from the procedure.
2. In the hi field is an ep pointing to the contour pointed to by the processor's ep just

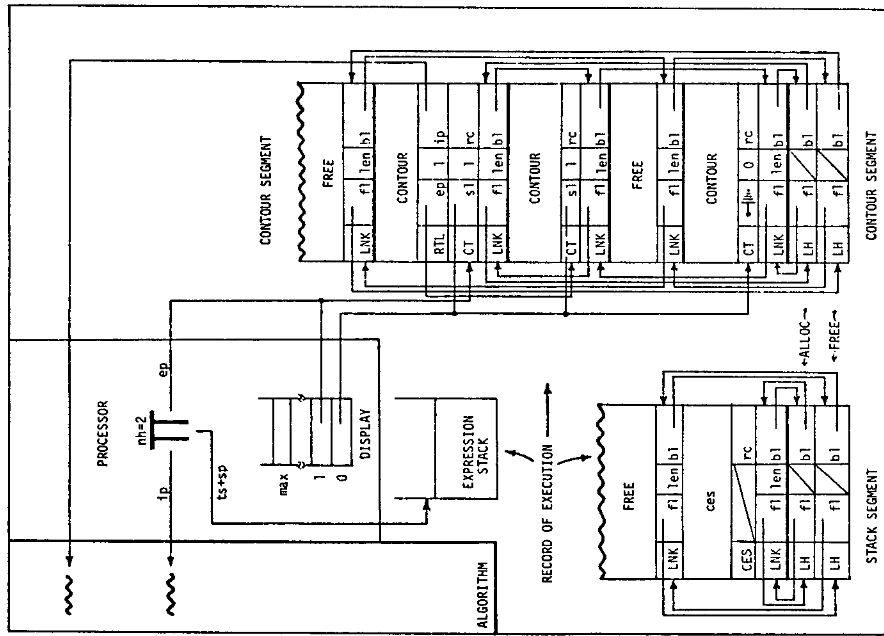

Figure 6

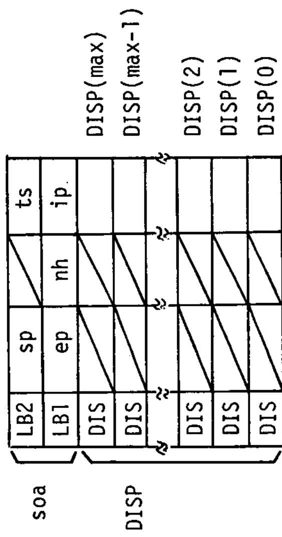

Figure 4

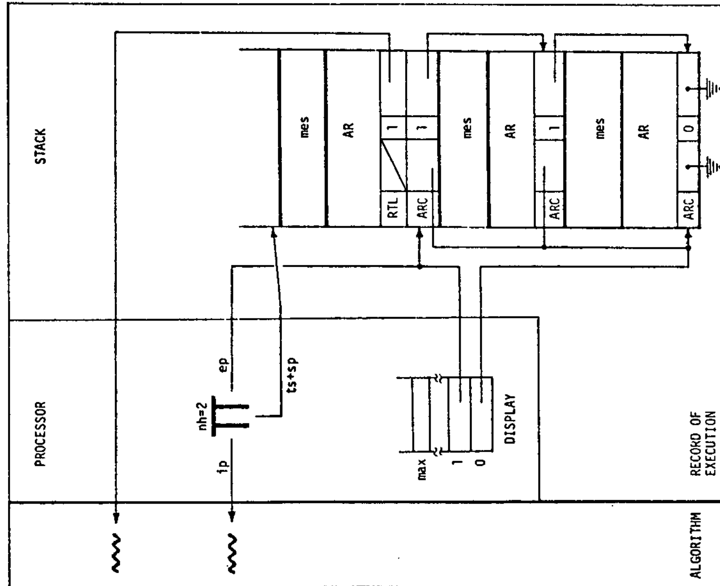

Figure 2

called. If there is no AR above a mes, it is the current one (see Figure 2). Since most block entries and procedure calls occur at statement boundaries, most mes's are empty. Only those mes's just below an AR whose antecedent was entered or called as part of an expression evaluation will be non-empty, e.g., the block and procedure call below:

..;a←1+BEGIN INT a; a←1 END+p(2)+3;...

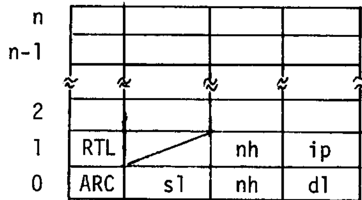

Figure 3

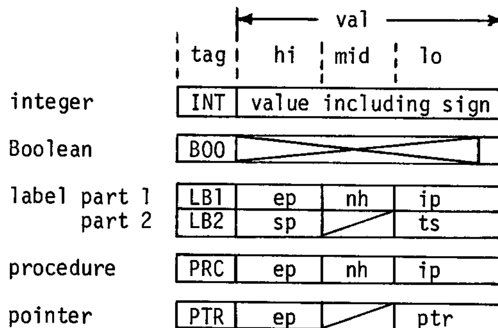

Figure 5

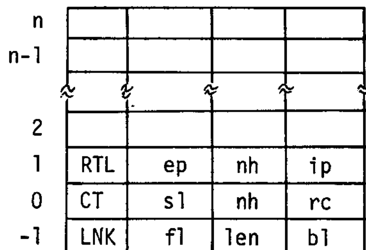

Figure 7

before the call of the antecedent.

3. In the mid field is a nh which is one more than that of the contour pointed to by the ep.

The remainder of the contour, i.e., words 1 through n if the antecedent is a block, and words 2 through n if the antecedent is a procedure, contain the subcells for the identifiers declared in the antecedent.

4.2.2.1.2 Free Block A free block is a block of n contiguous words which are free to be allocated. The first word of a free block has a tag of LNK and consists of three components.

1. In the hi field, a forward link, fl, and
2. In the lo field, a backward link, bl, for linking the free block into the doubly linked list of free blocks a length, len, giving the length of the free block including this word.

4.2.2.2 Expression Stack

The expression stack ES serves as the expression evaluation stack of the entire computation, (i.e., it can be thought of as the mes's merged together).

4.2.2.3 Stack Segment The stack segment consists of two doubly linked lists, both ordered by address:

1. One of copies of the ES (ces's);
2. The other of free blocks of memory.

The first two words of the stack segment are the heads of the two lists which are organized exactly as the heads of the contour segment lists.

4.2.2.3.1 Ces A ces forms the stack part of a label value [CDMPS73]. Rather than storing the stack with the label value, the stack is stored in the stack segment as a ces, and the sp of the label value points to the ces. The ces is a copy of the active portion of the ES at the time the label value is created; that is, if the copied ES has a height of n (ts=n), then the ces would be n+2 words. Word 1 is tagged LNK and is arranged exactly as the first word of a contour; Word 2 is tagged CES and it consists only of a reference count, rc, occupying the lo field. The rc counts the number of sp's pointing to the ces. The remaining words of the ces contain the valid part of the copied ES.

Once initialized, a ces is never modified. When a goto is done using a particular label value, the entire ces of the label is copied into the ES and the ts of the processor is set to the height of the ces.

4.2.2.3.2 Free Block The free blocks of the stack segment are arranged exactly as the free blocks of the contour segment.

3.2.2.2 Processor

4.2.2.4 Processor

The processor consists of a site of activity (label) and a display (see Figure 4).

3.2.2.2.1 Site of Activity

4.2.2.4.1 Site of Activity

The site of active (soa) occupies the last two words of the processor which are tagged LB1 and LB2. The word tagged LB1 consists of

1. in the lo field, an instruction pointer, ip, pointing to the next instruction to be executed
2. in the hi field, an environment pointer, ep, pointing to the topmost AR on the stack

2. In the hi field, an environment pointer, ep, pointing to a contour which is a descendent of the currently executed block or procedure.

3. in the mid field, a nesting height, nh, which is one more than that of the AR or contour pointed by the ep.

The word tagged LB2 consists of

4. in the hi field, a stack pointer, sp, pointing to the base of the stack

The word tagged LB2 consists of

4. In the hi field, a stack pointer, sp, pointing to the base of the ES.

5. in the lo field, a top of stack index, ts, which gives the displacement, in words, of the first free word on top of the mes on top of the topmost AR, i.e., the current height of the stack measured in words, i.e., ts+sp points to the first free word on top of the topmost mes (see Figure 2).

5. In the lo field, a top of stack index, ts, which gives the displacement of the first free word on top of the ES (see Figure 6).

3.2.2.2.2 Display

4.2.2.4.2 Display

The processor's accessing environment is the ordered list of ARs or contours, $A_n, \ldots, A_0$, with $n \geq 0$, such that the processor's ep points to A_n, and for each i such that $n \geq i \geq 1$, the static link of A_i points to A_i-1. Necesarily the nesting height of each A_i is i, and the nesting height of the processor is n+1.

The display is a vector of pointers to the ARs or contours of the processor's environment such that DISP [i] points to A_i. Each element of the display is tagged DIS and the pointer occupies the lo field. The display is of some convenient size, max, which is thereby the maximum nesting height allowed for any program. The display can be constructed at any time from the processor's ep and nh according to the following axioms:

1) DISP[nh-1] = ep
2) for i from nh-2 by -1 to 0, DISP[i] = the static link of the AR pointed to by DISP[i+1].

In practice, the display is reconstructed, possibly only partially, only when the environment of the processor changes.

3.3 VALUE REPRESENTATIONS

4.3 VALUE REPRESENTATIONS

The individual subcells of the AR's, mes's, contours, ES, and ces's contain values which may be one of the following types: integer, boolean, pointer, label, or procedure. Each value occupies one or two words. Figure 5 shows all of the value formats.

3.3.1 Integer Value

4.3.1 Integer Value

An integer value has an INT tag and occupies the entire val field. It is assumed that the representable values are the signed integers in some large convenient range.

3.3.2 Boolean Value

4.3.2 Boolean Value

A boolean value has a tag of BOO and is treated as though it occupies the entire val field (even though only one bit is really needed). The range of representable values is TRUE and FALSE.

3.3.3 Pointer Value

4.3.3 Pointer Value

A pointer value, which is tagged by PTR consists of two parts:

1. in the lo field is the pointer proper, ptr, which points directly to the pointed-to subcell;
2. in the hi field is the environment pointer ep, which points to the AR or contour containing the pointed-to subcell.

3.3.4 Label Value A label value is a copy of an soa of the processor and consists of two words tagged LB1 and LB2. The word tagged LB1 has

1. in the lo field, an ip pointing to the first instruction of the labeled statement,
2. in the hi field, an ep
3. in the mid field, an nh

and the word tagged LB2 has

4. in the hi field, an sp

(2–4: all copies of the same components of the processor at the time the label value is created)

5. in the lo field, a ts which is one less than the ts of the processor at the time the label value is created.

4.3.4 Label Value A label value is a partial copy of an soa of the processor and consists of two words tagged LB1 and LB2. The word tagged LB1 has

1. in the lo field, an ip pointing to the first instruction of the labeled statement,
2. in the hi field, an ep
3. in the mid field, an nh

(2–3: both copies of the same component of processor at the time the label value is created)

and the word tagged LB2 has

4. in the hi field an sp pointing to a ces which is a copy of the active portion of the ES minus the top word at the time the label value is created,
5. in the lo field, a ts, which is one less than the ts of the processor at the time the label value is created.

3.3.5 Procedure Value

4.3.5 Procedure Value

A procedure value which is tagged PRC, has but three parts:

1. in the lo field, an ip pointing to the entry point of the procedure,
2. in the hi field, an ep computed as follows: Let nhp be the nesting height of the innermost block or procedure surrounding the procedure body which contains the identified declaration of a non-local of the body. If nhp is defined, then the ep is a copy of DISP [nhp] of the processor at the time the procedure value is created. If nhp is not defined then the ep is NIL.
3. In the mid field, an nh which equals nhp+1 if nhp (described in (2) above) is defined and is zero otherwise.

The ep of the procedure value will be recognized as the necessary environment pointer of the procedure [Bry 70].

3.4 IDENTIFIER ACCESSING

4.4 IDENTIFIER ACCESSING

In the LSM and the PRCCM, identifier accessing is done by the traditional (i,j) pair method. At compile time each identifier is converted into an (i,j) pair, where i is the nesting height of the block or pro cedure containing the declaration of the identifier, and j is the relative displacement of the subcell for the identifier within the AR or contour for the declaring block or procedure.

Assume that the declaration part of the block or procedure, bp, declaring the identifier id_ℓ is

$$<\mu_0(<\text{s-id}:id_1>,<\text{s-mode}:m_1>)> \\ \wedge.....\wedge<\mu_0(<\text{s-id}:id_n>,<\text{s-mode}:m_n>)>$$

where $1 \le \ell \le n$, the id_i's are identifiers and the m_i's are modes.

Associated with each mode, m, is a size S(m) as follows

$$S(m) = \begin{cases} 2 \text{ if } m = \text{LABEL} \\ 1 \text{ otherwise} \end{cases} .$$

Then the j for id_ℓ is computed by

$$j = 1 + p + \sum_{k=1}^{\ell-1} S(m_k)$$

where

$$p = \begin{cases} 1 \text{ if bp is a procedure body} \\ 0 \text{ if bp is a block.} \end{cases}$$

3.5 LIFETIME CHECK

Each assignment of a pointer, label or procedure value is accompanied by a lifetime check which ascertains that the ep of the assigned value points to an AR no higher on the stack than the AR containing the subcell to which the value is being assigned. This insures that the AR pointed to by the ep of the assigned value will be on the stack, and therefore, the assigned value will be valid, at least as long as the AR to which the value is being assigned.

Upon exit from a block or procedure, if a pointer, label, or procedure value is returned, a lifetime check must be performed. This check ascertains that the ep of the value being returned points to an AR no higher on the stack than the AR pointed to by the dynamic link of the popped AR. This insures that the ep of the returned value does not point to the AR being popped from the stack.

The time overhead for the lifetime checks is as follows:

1. Each assignment of a pointer, label, or procedure value requires a constant time for the lifetime check.
2. Each exit of a block or procedure in which a pointer, label, or procedure value is being returned requires a constant time for the lifetime check.

3.6 MACHINE INSTRUCTIONS

Section omitted.

3.7 DEFINITIONS

Definition 1: A program $p \in$ GBSL is said to be lifetime well-stacked (LWS) if it can be run on LSM without incurring a lifetime error.

It may be observed that

$$\{p|p \text{ is LWS}\} \subsetneq \{p|p \text{ is WS}\}.$$

The inclusion is obvious since the LSM is a dele-

4.5 REFERENCE COUNTS AND STORAGE MANAGEMENT

Each copying and/or erasing of a reference countable value is accompanied by an update of the referred-to-cell. A reference countable value is either

1. a pointer value
2. a label value
3. a return label
4. a procedure value
5. a site of activity of the processor, or
6. a static link of a contour.

A referred-to cell is the contour pointed to by

1. the ep of a pointer value,
2. the ep of a label value,
3. the ep of a return label,
4. the ep of a procedure value,
5. the ep of a site of activity, or
6. the static link of a contour,

or the ces pointed to by

2. the sp of a label value.

In particular, if a reference countable value is copied, pushed, or created, then the reference count of each referred-to cell is incremented by one unless the value resides in the referred-to cell.

If a reference countable value is erased, overwritten, or popped, then the reference count of each referred-to cell is decremented by one unless the value resides in the referred-to cell. If the value is a label, then if the reference count of the ces pointed to by its sp has gone to zero, it is necessary to ripple. Rippling consists in decrementing the reference counts of all cells referred to by any reference countable value in the ces. If any of the values in the ces are labels, then there is a recursion of rippling. Thus erasing a label value can result in an arbitrary amount of reference count decrementation.

The net result of this reference count maintenance scheme is that at any time, the reference count of each contour counts the number of ep's and static links that point to it, except for those

tion machine. The inequality is demonstrated by the program

```
1  BEGIN PTR INT p;
2      BEGIN INT b;
3         p←ADDR b
4      END;
5      p←NIL INT
6  END
```

The assignment in line 3 causes a lifetime error; however, the program is well-stacked because it never uses the dangling pointer.

SPECIAL NOTE:
To save space, Sections 5, 6, and 7 are given from here on in the left-hand column of the page.

5. TRANSLATION OF GBSL PROGRAMS INTO MACHINE LANGUAGE PROGRAMS

Section omitted.

6. EXAMPLE PROGRAM

Section omitted.

7. RESULTS

In this abbreviated version of Section 7 the theorems are stated and only an outline of their proofs are given. The complete proofs depend on the microcode for the instructions which has not been given.

The first theorem states that PRCCM, when executing a lifetime will stacked program that does not generate isolated knots, behaves essentially like a stack implementation.

Theorem 1: Let $p \in$GBSL be LWS. Suppose that p does not generate isolated knots.
Then,
Let S_i be a snapshot in PRCCM (p). Suppose that $C_1, \ldots, C_n$ is the complete address ordered list of contours in the contour segment of S_i. Then, the following may be said: (See Figure 8)

1. $C_1, \ldots, C_n$ are adjacent to each other and are at the bottom of the contour segment
2. The free list consists of exactly one element F sitting above and adjacent to C_n, and comprises all of the segment not included in $C_1, \ldots, C_n$ and the list heads.
3. The processor's ep points to C_n.
4. The list $C_n, \ldots, C_1$ gives the order that the contours will be exited if the normal block exit and procedure return chain were followed.
5. The list $C_1, \ldots, C_n$ gives the order in which the contours were allocated.

Proof: By induction on the length of the computation.

The initial snapshot clearly satisfies the conditions since no contours have been allocated and the free list consists of a single free block containing the entire segment except for the list heads.

Assume that conditions 1-5 hold at snapshot S_i. Show that they hold in snapshot S_{i+1}. There are two major kinds of instructions to consider:1) the en-

that reside in the contour itself. Also at any time, the reference count of each ces counts the number of sp's that point to it (no sp can ever reside in the ces that it points to).

The normal method of storage management consists in the following:

1. Both the free list and the allocated list of the contours and stack segments are ordered by address.
2. Allocation of a contour or a ces is done on a first bit basis from the beginning of the appropriate free list, i.e., the first free block in the list large enough to hold the new contour or ces is taken. Any left over space in the block is left on the free list. The allocated block is linked to the appropriate allocated list by searching for its place from the end of the list.
3. Reclamation of inaccessible contours and stacks is done ultimately by garbage collection. This must be because the reference count of an inaccessible cell is not always zero. This situation happens if there is an isolated knot of cells. An isolated knot of cells is a set of cells (not including the processor) pointing only to each other and pointed to by no other cells outside the set. Note that because pointers pointing to the cells in which they reside are not counted in reference counts, knots consisting of one cell do not bother the implementation.

The reference counts of contours and ces's are used merely to help reclaim inaccesible contours and ces's sooner than they would be by the garbage collector, thereby postponing and hopefully obviating the need for garbage collection.

It has already been seen how erasure of a label value may lead to deallocation of the ces that its sp points to and to deallocation of other ces's pointed to by sp's in the first deallocated ces, etc.

As for contours, there are only three times at which it is possible for a contour to be reclaimed by virtue of a zero reference count: block exit, procedure exit, and goto. At each of these times there is an attempt to reclaim all those contours that would normally be deallocated in a deletion model. If the appropriate contours do not have a reference count of zero at this time, they are left in the allocated list to be picked up later by the garbage collector.

Upon exit from a block or procedure, the exited contour's reference count is decremented by 1, reflecting the change in the processor's ep. If at this point, the reference count is not zero, nothing further is done. If, however, the reference count of the exited contour is zero, it can be returned to the free list. Before this is done, it is necessary to decrement the reference counts of all cells referred to by any pointers, labels, and procedure values and any return label in the contour, and to initiate rippling for the sp's of any lobel values in the contour. It is possible to generate code which goes directly to the cells containing reference countable values, since the modes of the identifier declared in the exited block or procedure are known at compile time. Therefore, if

vironment changing instructions, i.e., block entry, block exit, procedure call, procedure return, and goto, 2) all other instructions....

Corollary 2: Let p∈GBSL be LWS.
Suppose p does not generate any isolated knots.
Then,
1. Allocation of a contour (finding a free block big enough and linking the contour in the allocated list) takes a constant amount of time.
2. Freeing a contour (removing it from the allocated list and finding its place in the free list with possible merging) takes a constant amount of time.
3. If allocation fails, it is becuase there is not enough room at all in the segment and is not due to failure to have a big enough free block due to fragmentation.
4. Garbage collection will never occur (i.e., if allocation fails there is no space to reclaim).

Proof:
These all follow striaght forwardly from the proof of Theorem 1.

Theorem 3: Let p∈GBSL be LWS and suppose that p does not generate isolated knots.
Suppose p does not have labels (after compiling away hoptos), pointers, and procedures).
Then,*
1. $T(LSM(p)) \le Q + h\times(be)$
2. $T(PRCCM(p)) \le Q + h'\times(be)$
3. $|T(LSM(p)) - T(PRCCM(p))| = \Theta(be)$

Where: be is the number of block entries that occur in the computation.
h and h' are constant per block entry.
Q is time for everything else which is identical in both models.

Proof:
The conditions restrict the code that is generated to a subset of all the instructions. The result follows by inspection of this subset. In particular, there will be no scanning of contours about to be freed for reference countable value; This is because it can be determined at compile time that there are no reference countable values in any contour.

Note that (3) says essentially that $h\times(be)$ and $h'\times(be)$ balance each other off. The actual constants h and h' depend on the specific machine implementations.

Theorem 4: Let p∈GBSL be LWS and suppose that p does not generate isolated knots.
Suppose p does not have any labels (after compiling away hopto's and pointers.
Suppose that all computations of procedure values are accompanied by assignments or calls.
Then,

$$T(LSM(p)) \le Q + h\times(be) + i\times(proca) + j\times(procc)$$
$$T(PRCCM(p)) \le Q + (h'+m\times i')\times(be) + i'\times(proca) + (j'm\times i')\times(procc)$$

Where: be is the number of block entries in the computation.
proca is the number of procedure assignments in the computation.
procc is the number of procedure calls in

*T(LSM(p)) means "Time for LSM to execute p".

there are no reference countable cells in the contour there is no scanning overhead at exit time. When the exited contour is finally freed, the free list of the contour segment is searched from the beginning to find the proper place for the contour in the address ordered free list. If the freed contour is physically adjacent to any of its neighbors on the list, it and its neighbors are merged into one free block.

Upon a goto the processor's ep is reset to a copy of the ep of the label used in the goto. In general the new ep bears no relation whatsoever to the old ep of the processor before the goto. However, if the program is well-stacked, the ep of the label points to a contour somewhere on the chain of normal returns (i.e., the dynamic chain) from the contour pointed to by the old ep. Therefore, in the hopes that the program is well-stacked, there is an attempt to deallocate all of the contours on this chain:
1. Let C be the contour pointed to by the old ep.
2. The reference count of C is decremented by 1 reflecting the resetting of the processor's ep or the deallocation of the previous contour on the chain.
3. If C's reference count is not zero, the looping stops and control continues at the labeled statement.
4. If C's reference count is zero, then the contour must be freed. Before it can be freed, each word of the contour must be scanned for the presence of a reference countable value. The reference counts of all cells referred to by any of these reference countable values must be decremented and the rippling of sp's must be initiated if necessary. (Because it is not possible at compile time, to tell which contours will be on the return chain, it is not possible for the code to go directly to the cells containing reference countable values; instead some interpretive scheme, such as scanning must be used.)
5. C is freed by searching the free list from the beginning to find its place and merging C with its neighbors if possible.
6. If C has a return label, the next contour is that pointed to by the return label's ep. If not, the next contour is that pointed to by the static link of C. If there is no next contour (the return label ep or the static link is NIL) the looping stops, and control continues at the labeled statement.
7. If there is a next contour, it is taken as C and control branches back to step 2.

The overhead required for reference count maintenance and storage management is as follows:
1. Each pushing or popping of a pointer or procedure value requires a constant time for reference count incrementation or decrementation.
2. Each assignment of a pointer or procedure value requires a constant time for decrementing the reference count of the cell referred to by the overwritten value and for incrementing the reference count of the cell referred to by the assigned value.
3. Each pushing of a label value requires a constant time for incrementing the reference counts of the contour and the ces referred to by the label value.

the computation.
h and h' are constant per block entry.
i and i' are constant per procedure assignment.
j and j' are constant per procedure call.
m is the maximum number of procedures declared in any block or procedure of p.
Q is the time for all the rest which is the same in both models.

Corollary 5: In Theorem 4, h'+mi' can be replaced by H' which is constant per block entry per program and j'+mi' can be replaced by J' which is constant per procedure call per program. Consequently for a given p$\in$GBSL satisfying the conditions of Theorem 4,
$|T(LSM(p))-T(PRCCM(p))| = \Theta(be + proca + procc)$.

Theorem 6: Let p$\in$GBSL be LWS and suppose that p does not generate isolated knots.
Suppose p does not have any labels (after compiling away hopto's).
Suppose that all computations of pointer and procedure values are accompanied by assignments, indirections or calls.
Then,

$T(LSM(p)) \le Q + h\times(be) + i\times(proca) + j\times(procc) + i\times(ptra)$

$T(PRCCM(p)) \le Q + (h'+m\times i')\times(be) + i'\times(proca) + (j'm\times i')\times(procc) + i'\times(ptra) + k'\times(ptri)$

Where: be, proca, procc are as in Theorem 5.
ptra is the number of pointer assignments in the computation.
ptri is the number of pointer indirections in the computation.
h and h' are constant per block entry.
i and i' are constant per pointer or procedure assignment.
j and j' are constant per procedure call.
k' is constant per pointer indirection.
m is the maximum number of procedures and pointers declared in any block or procedure of p.
Q is the time for all the rest which is identical in both models.

Note here that k', the time per pointer indirection in PRCCM, cannot be balanced off to anything in LSM, but k' is constant.

Corollary 7 to Theorem 6 is as Corollary 5 is to Theorem 4.
For a given p$\in$GBSL satisfying the conditions of Theorem 6,
$|T(LSM(p)) - T(PRCCM(p))| - k'\times(ptri) = \Theta(be + proca + procc + ptra)$

Observation on labels:
If labels and gotos are added, then

1. In LSM, the added time is constant per label assignment and per goto.
2. But in PRCCM, the added time per label assignment and goto is not constant. The time does not even have a compile time computable bound per program. The essential reason is that erasing of the entire expression stack occurs in these operations. In order to maintain stack-like behavior as per Theorem 11.1, a potentially unbounded rippling reference count decrementation may be required.

Hopto's and branches that occur during execution of control structures like conditionals, loops, while-do's etc. do not incur this cost since no environ-

4. Each popping of a label value requires a constant time for decrementing the reference counts of the contour and ces referred to by the label value, plus a potentially unbounded time if the ces's reference count goes to zero. The potentially unbounded time is both to decrement the reference counts of the cells referred to by the arbitrary number of reference countable values in the ces and to ripple if any of the reference countable values is a label.
5. Each assignment of a label value requires a potentially unbounded time for the reference count decrementation and possible rippling associated with overwriting of a label value (as in popping) and a constant time for incrementing the reference counts of the contour and ces referred to by the assigned label value.
6. Each indirection requires a constant time for decrementing the reference count of the cell referred to by the overwritten pointer value.
7. Each block or procedure entry requires a constant time for setting the reference count of the new contour and a potentially unbounded time for searching for a free block large enough.
8. Each block or procedure exit requires a constant time for decrementing the reference count of the exited contour, and if its count goes to zero, a potentially unbounded time proportional to the number of pointers and procedures in the contour to decrement the reference counts of the cells referred to by them. (For a given program with a maximum number of pointers and procedures in a block or procedure this time can be bounded.), a potentially unbounded time to decrement the reference counts of the cells referred to by the label values in the contours and to ripple if necessary, and a potentially unbounded time to search for the freed contour's place in the free list.
9. Each goto requires potentially unbounded time to decrement the reference counts of cells referred to by the arbitrary number of reference countable values in the ES, potentially unbounded time to ripple the sp's of the labels in the ES, potentially unbounded time to increment the reference counts of the cells referred to by the arbitrary number of reference countable values in the copied ces and potentially unbounded time to chase through the arbitrary deep chain of returns, spending a potentially unbounded time at each contour in the chain.

The key observation to be made is that if the program happens to be lifetime well stacked (cf. Section 3.7) and no isolated knots of more than one cell are formed to foul up the reference counts, then all contours will have a zero reference count upon exit, and will thus be returned to the free list immediately. By exit in this context, we mean exit by way of a goto as well as by block or procedure exit.

Given:

1. The ordering of the free and allocated lists by address,
2. The first bit allocation scheme, and
3. The merging of freed contours to their adjacent neighbors on the free list,

the observation implies that for lifetime well-

ment change occurs in these cases.
Moral: Retention can be made fairly cheap for structured programs [Dij72].

8. CONCLUSIONS

This paper has presented two implementations of generalized block structured languages, one which implements the deletion strategy with lifetime checks and the other which implements the retention strategy.

The two implementations have been expressed as machines executing from the same code so that their times for given program features may be compared.

The results show that for lifetime well-stacked programs which generate no isolated knots and which are structured in the sense of structured programming, retention can be made to run in approximately the same order of magnitude of time as the Lifetime Stack Machines.

It is the authors' belief that most programs meet these conditions (or should). Programs not meeting the first two conditions either need retention or are doing some list manipulation in which retention might be useful. Presumably the users of these programs would be willing to pay for the full cost of retention. Programs not meeting the third condition (i.e., which have gotos) bear the brunt of the expense. However this is a desirable deterrent to using something which should be used but sparingly. (One co-author dissents from this last opinion.)

9. BIBLIOGRAPHY

Note: DSIPL (pronounced "disciple") is Proceedings of ACM Symposium on Data Structures in Programming Languages, SIGPLAN Notices, February 1971.

Bry70 Berry, D. M.,"Necessary env.", SIGPLAN Notices (September 1970).

Bry71a Berry, D. M., "Introduction to Oregano," DSIPL (1971).

Bry71b Berry, D. M., "Block Structure: Retention vs. Deletion," SIGACT Proceedings, (1971).

BW72 Bobrow, D. B. and Wegbreit, B. A Model and Stack Implementation of Multiple Environments, Harvard University Center for Research in Computing Technology, Report 7-72, (1972).

Bow71 Bowlden, H., Private Communication, September 1971.

BL70 Branquart, P. and Lewi, J., "A Scheme of Storage Allocation and Garbage Collection for ALGOL 68," in Pck (1970).

CDMSP73 Chirica, L. M., Dreisbach, T. A. Martin, D. F., Peetz, J. G., Sorkin, A., Two PARALLEL EULER Run Time Models: The Dangling Reference Imposter Environment,

stacked programs not generating isolated knots of more than one cell, the contour segment will look as shown in Figure 8 at all times.

The allocated contours will be adjacent to each other and will all be in the "bottom" of the contour segment in the order of their activation. The processor's ep will point to the "top most" contour. In addition, the free list will consist of exactly one block which sits immediately on top of the "top most" allocated contour; this free block comprises the entire unallocated portion of the contour segment.

Therefore, upon entry to a block or procedure, allocation of a contour will always find sufficient space in the first element of the free list (or else the segment itself is exhausted). The new contour will be carved out of this free block, thus preserving the situation. Similarly, upon exit, the top most contour's reference count will be zero. The contour will be returned to the free list and will be merged with the free block, thus again preserving the situation.

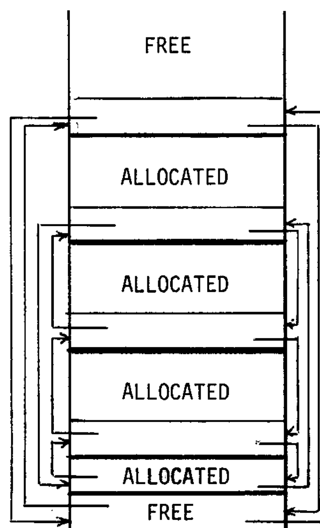

Figure 8

Note that under these conditions, because the free list is searched from the beginning on both allocation and deallocation, and the allocated list is searched from the end to put a newly allocated contour in its place, allocation and deallocation of contours will take a constant amount of time.

4.6 MACHINE INSTRUCTIONS

Section omitted.

4.7 DEFINITIONS

Definition 2: A cell, C, in the record of execution (i.e., a contour, the ES, or a ces) is said to be accessible if and only if one of the following holds:

1. C is pointed to by the ep or sp of the processor.
2. C is pointed to by an ep, sp, sl, or dl stored in some cell D which is accessible.

It is not necessary to take display elements and pointers-proper into account in determining

and Label Problems, This Proceedings, (1973).

Dij72 Dijkstra, E. W., "Notes on Structured Programming," in Dahl, Dijkstra, and Hoare, Structured Programming, Academic Press: London, (1972).

Fis72 Fischer M. J. "Lambda Calculus Schemata," Proceedings of ACM Conference on Proving Assertions about Programs, SIGPLAN Notices 7:1, January 1972.

Joh71 Johnston, J. B., "The Contour Model of Block Structured Processes," DSIPL, (1971).

LW69 Lucas, P. and Walk, K, "On the Formal Description of PL/I," Annual Review of Automatic Programming, 6:3, (1969).

Nau62 Naur, P., "Revised Report on the Algorithmic Language ALGOL 60," CACM, 6:1, (January 1963).

Org73 Organick, E. I., Computer Systems Organization, Academic Press: New York, (1973).

Pck70 Peck J. E. L. (Ed.), ALGOL 68 Implementation, North Holland: Amsterdam (1970).

RR64 Randal, B. and Russell, L. J., ALGOL 60 Implementation, Academic Press: New York, (1964).

Rey70 Reynolds, J. C., "GEDANKEN - A simple Typeless Language Based on the Principle of Completeness and the Reference Concept," CACM 13:5 (May 1970).

SS72 Scott, D., and Strachey, C., "Toward a Mathematical Semantics for Computer Languages," Proceedings of the Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, (April 1971).

vWn69 van Wijngaarden, A., et al., "Report on the Algorithmic Language ALGOL 68," Num. Math. 14, 79-218, (1969).

Wlk69 Walk, K, et al., Formal Definition of PL/I, ULD Version III IBM Vienna, (1969).

Weg71 Wegner, P., "Data Structure Models for Programming Languages," DSIPL, (1971).

WE70 Wozencraft, J. M. and Evans, A., Notes on Programming Linguistics, E.E. Dept., MIT, (1970).

accessibility because cells pointed to by these pointers are necessarily pointed at by other pointers which are taken into account.

Definition 3: A program pϵGBSL is said not to generate isolated knots if at no time during its execution does a set of more than one cell pointing only to one another become inaccessible.

The concept of generation of isolated knots is important because reference count maintenance ceases to identify all inaccessible cells if there are isolated knots. Our handling of reference counts does not count pointers pointing to the cell in which they reside so knots of one cell do not bother the implementation.