INTRODUCTION

This MTB proposes an optimized call/push/return (CPR) strategy for PL/I and FORTRAN that is compatible with the current scheme. It is based in part on MTB-434, titled "New Call/Push/Return Strategy". The strategy described in MTB-434 proposed incompatible stack frame changes which would affect many (about 90) system programs and would force several users (such as compiler writers) to change their programs. After some discussion, it was felt that the estimated performance gain might not justify the cost of implementation. The strategy described in this MTB does not require any stack frame changes and so has a much lower implementation cost. There is still significant performance improvement, however.

Basically, the proposal is to change the compiled call, entry and return sequences along the lines described in MTB-434 and to optimize the code in the operators somewhat. After the system is recompiled to use this, there should be about 3.6% performance improvement. The gain projected for the strategy in MTB-434 was about 6%. In addition, this MTB describes a way to substantially reduce the CPR overhead for non-quick internal procedures and provides more information about binder optimizations.

PL/I CHANGES

The changes proposed for PL/I external procedures are the same as those proposed in MTB-434 with the exception of those involving stack frame or stack header changes. This means that there will be no double word to copy, the entry pointer will not be replaced, the stack will remain doubly threaded, and the stack end pointer will still be used. A prototype code sequence is attached.

Calls to some internal procedures can be optimized even more. If the compiler knows that a non-quick internal procedure is NOT called through an entry variable, the code can call an intra-segment internal call entry operator similar to that proposed for bound segments. This could speed up recursion in internal procedures.
The changes for internal procedures can be summarized as follows:

- Make most of the changes indicated for external procedures.
- Have a single operator that combines call and entry. Pick up the stack frame size in the calling sequence.
- Freeze the offsets of the call_entry operators in p11_operators so that the transfer vector need not be used. Internal entries are not traced.
- In some cases, the display pointer need not be stored in or retrieved from the argument list.
- Use a different pointer register convention for argument lists so that PRO can continue to point to the operator table.
- Do not load or restore indicators.

BINDER CHANGES

This MTB provides additional implementation information for some of the bound segment optimizations described in MTB-434. The types of changes proposed are the same, although the actual code sequences differ. Quick external procedures (sharing stack frames) is not further discussed here.

The basic bound segment optimization is to bypass the call operator when resolving a link between two components. Instead, an operator (or embedded code) that combines the functions of the call and entry operators is used. In certain cases, the return sequence is also shorter.

The decision about when to optimize is independent of the location of the optimized code. The code is embedded in the object segment if the compiler has allowed room for it and if the bindfile has specified it. Otherwise, special operators are used. Optimization takes place when indicated by relocation bits (defined later in this MTB), subject to certain restrictions. Relocation bits are used for several reasons. They are currently the binder's only means for determining where and how to change/relocate code. They can distinguish between link references that are part of calling sequences and link references used for entry variables, etc. They can indicate reliably the location of return sequences. The definitions of the new relocation bits are given below. Briefly, one pattern means link 15 relocation in a calling sequence. Another pattern means an external return sequence. Entry sequences are already located via definitions.
The binder should always be able to optimize calling sequences that are flagged by the new relocation bits and that can be resolved within the bound segment. It does not matter whether the called entry is retained or not because the original entry sequence remains intact; the optimized code circumvents it. Even calls to components compiled with older versions of the compiler can use the new bound call_entry operators. The binder should always set PR2 to the "real" entrypoint so that stack_frame.entry pointer will be set properly. Since the new (PL/I- and FORTRAN) entry sequences are one word shorter, the compilers should add a pad word to them so that:

1) the bound call_entry operators can always use the same instruction to transfer back to the program, and
2) the symbol block offset, etc. used by stu are the same distance from the entry point.

The new calling sequence should have a word of pad in the form of a NOP instruction to allow room for the binder to insert code to use the new bound call_entry operators. The pad should be added even when the program is compiled with space for embedded entry and return code because the decision about whether to use the operator or embedded entry code depends on the callee. The binder can determine whether a component has space for embedded operator code by a bit in the object map (see Object Map Changes below). An alternative to the pad in the calling sequence would be for the bound-call entry operator to load the stack frame size directly from the entry sequence. While this method would save 1 instruction if the caller and callee were not in the same bound segment, this would cost an extra memory reference if the caller and callee were in the same bound segment. That extra memory reference could be especially expensive on the ADP because of the high expense of loading the cache just to make one memory reference in an 8-word memory block.

Although the binder's decision about whether to optimize is independent of the specific compiler, the code to be added requires knowledge of the exact calling and entry sequences. This is a change of direction for the binder. The current philosophy is to depend only on standard object segment features.

Optimizing a return sequence is subject to several more restrictions than the call or entry sequences. An entry may be entered through either the standard sequence or the optimized sequence, but a return cannot be optimized unless the program is known to be called only by another PL/I or FORTRAN program. This means that none of the component's external entries can be retained or used in entry variables. The former is known before relocation begins; the latter cannot be known until all components have been relocated. The binder will keep track of all the potentially optimizable return points as it relocates. Then after all components have been processed, it will know which
components have entries used in entry variables and can change
the return points that are still optimizable. The binder's
definition of "used in entry variables" is an entry referenced
through a link accessed by an instruction with link 15
relocation. In any case, only return points that are flagged by
the new "optimize return" relocation bits can be optimized.

GATE CHANGES

As in MTB-434, this MTB proposes a new gate push operator for
non-hardcore gates and the use of the location transferring to
the setup "subroutine" as the stack frame's entry pointer. This
is not necessary for compatibility but is significantly faster.
Hardcore gates cannot use this operator because they cannot
access the LOT in the standard way.

Fast hardcore gates will be restricted to calling only ALM
programs. (Currently they call only ALM programs anyway.) The
reason for this is that ALM return operators must be used to
return from a lower ring. The PL/I return operator is being
changed to set PR7 only to the base of the stack it is invoked
on, while the ALM return operator will continue to reset PR7 to
the base of the stack being returned to.

ALM CHANGES

Although the code generated for the push pseudo-op will not
change, the .stack frame size builtin variable should still be
added. It is needed to implement the invocation of the gate push
operator via a macro (rather than with a new pseudo-op). Also it
is still a good idea for ALM programs that currently depend on
the push pseudo-op code to use this instead.

UNRESOLVED ISSUE

It is not clear whether it is advisable to do the optimization
where the binder embeds the CPR sequence in the object code.
This saves only 3 instructions and does embed knowledge of stack
frame formats in object code in a way that we normally frown
upon. Also it requires extra work to implement.

NEW CODE SEQUENCES

This section presents the proposed new code sequences to be used
in the operators. The instructions in the operators are
indicated by a vertical line in the left margin. All other
instructions are in the caller's or callee's object segments.
Code in parentheses is not considered to be part of the CPR
mechanism. Argument list preparation is not included. The code sequences have not been completely optimized for pipelined hardware. The PL/I versions are prototypes, since there are several PL/I entry operators. Likewise the bound_call_entry operator is also a prototype, since there must be one for every PL/I external entry operator.
PROPOSED PL/I INTER-SEGMENT CALL SEQUENCE
(Total = 34)

(ldaq arglist_header)
epp2 callee
epp3 arglist
staq pr3|0
tsp1 pr0|call_op
nop

spri1 pr6|stack_frame.return_ptr
epp0 pr3|0
call6 pr2|0

eax7 stack_frame_size
tsp2 pr6|stack_header.new_ent_op,*
(pad)

epp3 pr7|stack_header.stack_end_ptr,*
spri6 pr3|stack_frame.prev_sp
spri0 pr3|stack_frame.arg_ptr
epp1 pr3|0,7
spri1 pr3|stack_frame.next_sp
spri1 pr7|stack_header.stack_end_ptr
epp6 pr3|0
epp2 pr2|-2
spri2 pr6|stack_frame.entry_ptr
spbps2 pr6|text_base_ptr
epaq pr2|0
lprp4 pr7|stack_header.lot_ptr,*au
spri4 pr6|linkage_ptr
stz pr6|stack_frame.operator_return_offset
epp0 operator_table
spri0 pr6|stack_frame.operator_ptr
spri1 pr6|4
ldi 0,d1
tra pr2|4

(random code)
.
.
(end of code)
call6 pr0{return_op}

spri6 pr7|stack_header.stack_end_ptr
epp6 pr6|stack_frame.prev_sp,*
epp0 pr6|stack_frame.operator_ptr,*
ldi pr6|stack_frame.return_ptr+1
rtcd pr6|stack_frame.return_ptr
INTRA-SEGMENT CALL SEQUENCE
(WITH OPERATORS)
(TOTAL = 26)

(epp4  pr6|linkage_ptr,*)
(ldaq  arglist_header)
epp2  callee
(epp3  arglist
staq  pr3!0
eax7  stack_frame_size
tsp1  pr0|bound_call_entry_op

tra  bound_call_entry
spri1  pr6|stack_frame.return_ptr
epbp7  pr6!0
epp1  pr7|stack_header.stack_end_ptr,*
spri6  pr1|stack_frame.prev_sp
spri3  pr1|stack_frame.arg_ptr
epp5  pr1!0,7
spri5  pr1|stack_frame.next_sp
spri5  pr7|stack_header.stack_end_ptr
epp6  pr1!0
spri2  pr6|stack_frame.entry_ptr
spbp2  pr6|text_base_ptr
spri4  pr6|linkage_ptr
stz  pr6|stack_frame.operator_return_offset
spri0  pr6|stack_frame.operator_ptr
spri5  pr6!4
tra  pr2!x

(random code)

(end of code)
call6  pr0|return_op_no_ind
spri6  pr7|stack_header.stack_end_ptr
epp6  pr6|stack_frame.prev_sp,*
rtcd  pr6|stack_frame.return_ptr
INTRA SEGMENT CALL SEQUENCE
(NO OPERATORS)
(Total = 23)

(epp4  pr6|linkage_ptr,*
(ldapq  arglist_header)
epp3  arglist
staq  pr3;0
stcd  pr6|stack_frame.return_ptr
call6  callee
nop

eax7  stack_frame_size
epp2  *-N
epp1  pr7|stack_header.stack_end_ptr,*
spri6  pr1|stack_frame.prev_sp
spri3  pr1|stack_frame.arg_ptr
epp5  pr1;0,7
spri5  pr1|stack_frame.next_sp
spri5  pr7|stack_header.stack_end_ptr
epp6  pr1;0
spri2  pr6|stack_frame.entry_ptr
spbp2  pr6|text_base_ptr
spri4  pr6|linkage_ptr
stz  pr6|stack_frame.operator_return_offset
spri0  pr6|stack_frame.operator_ptr
spri5  pr6;4
(random code)
.
.
.(end of code)
epbp7  pr6;0
spri6  pr7|stack_header.stack_end_ptr
epp6  pr6|stack_frame.prev_sp,*
rtcd  pr6|stack_frame.return_ptr
CURRENT PL/I INTERNAL CALL SEQUENCE
(total = 48)

(fld arglist_head, du)
epp2 callee
eax1 arglist
tsx0 pr0!call_int_this

tra call_int_this
ora 8, dl
epbp7 pr6; 0
staq pr7; 0, 1
stx0 pr6; stack_frame.return_ptr + 1
epp0 pr7; 0, 1
spr6 pr0; 2, au
tra pr2; 0
eax7 stack_frame_size
epp2 pr7; stack_header.pl1_operators_ptr,*
tsp2 pr2; int_entry_op

tra int_entry
epaq pr2; 0
lprp4 pr7; stack_header.lot_ptr,*au
epp3 pr7; stack_header.stack_end_ptr,*
spr6 pr3; stack_frame.prev_sp
spr10 pr3; stack_frame.arg_ptr
epp1 pr3; 0, 7
spr11 pr3; stack_frame.next_sp
spr11 pr7; stack_header.stack_end_ptr
epp6 pr3; 0
lda pr0; 0
epp3 pr0; 2, au*
spr13 pr6; display_ptr
epp2 pr2; 3
tra save_link
spr14 pr6; linkage_ptr
spr12 pr6; stack_frame.entry_ptr
spbp2 pr6; text_base_ptr
spbp2 pr6; stack_frame.return_ptr
stz pr6; stack_frame.operator_ret_ptr
epp0 operator_table
spr10 pr6; stack_frame.operator_ptr
spr11 pr6; 4
ld1 0, dl
tra pr2; 5
(random code)
.
(end of code)

tra pr0;return_op

tra return_mac
epbp7 pr6;0
spri6 pr7;stack_header.stack_end_ptr
epp6 pr6;stack_frame.prev_sp,*
epbp7 pr6;0
epp0 pr6;stack_frame.operator_ptr,*
ldi pr6;stack_frame.return_ptr+1
rtcd pr6;stack_frame.return_ptr
NEW PL/I INTERNAL CALL SEQUENCE
(Total = 27)

(ldaq arglist_header)
epp2 callee
epp3 arglist
staq pr3;0
eax7 new_stack_frame_size
tsp1 pr0;int_call_entry_this

spri1 pr6;stack_frame.return_ptr
epp4 pr6;linkage_ptr,*
epbp7 pr6;0
epp1 pr7;stack_header.stack_end_ptr,*
spri6 pr1;stack_frame.prev_sp
spri6 pr1;display_ptr
spri4 pr1;linkage_ptr
spri3 pr1;stack_frame.arg_ptr
epp3 pr1;0,7
spri3 pr1;stack_frame.next_sp
spri3 pr7;stack_header.stack_end_ptr
epp6 pr3;0
spri2 pr6;stack_frame.entry_ptr
spbp2 pr6;text_base_ptr
stz pr6;stack_frame.operator_return_offset
spri0 pr6;stack_frame.operator_ptr
spri3 pr6;4
tra pr2;x

(random code)
.
.
(end of code)
call6 pr0;return_op_no_ind

spri6 pr7;stack_header.stack_end_ptr
epp6 pr6;stack_frame.prev_sp,*
rtcd pr6;stack_frame.return_ptr
NEW GATE PUSH SEQUENCE
(Total = 18)

ldx7 .stack_frame_size,du
epp2 pr7;stack_header.pl1_operators_ptr,*
tsp2 pr2;gate_push_op
tra gate_push
epp4 pr7;stack_header.stack_end_ptr,*
spr13 pr4;stack_frame.entry_ptr
spr16 pr4;stack_frame.prev_sp
spr10 pr4;stack_frame.arg_ptr
epp6 pr4;0
epaq pr3;0
lprp4 pr7;stack_header.lot_ptr,*au
spr14 pr6;stack_frame.lp_ptr
epp5 pr6;0,7
spr15 pr7;stack_header.stack_end_ptr
spr15 pr6;stack_frame.next_sp
eax7 1
stx7 pr6;stack_frame.translator_id
tra pr2;0

...
NEW RELOCATION BITS

Define the following new relocation type:

"11011"b - optimize

where

optimize indicates an instruction or code sequence that can be changed to be made more efficient. The specific changes depend on the compiler(s) involved and may be subject to restrictions. The five bits of relocation code are immediately followed by a fixed length 3-bit field that specifies the type of code to be optimized. Currently only "001"b - link 15 relocation at the beginning of a calling sequence, and "010"b - external return sequence, are defined.
BINDFILE CHANGES

Define the following master keyword:

Embed_Entry_Return

Whenever optimization is indicated by relocation codes, embed in the bound segment code that is normally in the call, entry, and return operators. This can occur only when the compiler has allowed space for the code. WARNING: use of this keyword causes the bound segment to contain code that is dependent on system conventions which are subject to change.

Define the following normal keyword:

no_embed_entry_return

Do not embed operator code in this component even if the compiler has allowed room for it.
OBJECT MAP CHANGES

Define the following object map flags:

has_cpr_pad

The object segment has space for system-dependent call/push/return code to be inserted.

embeds_cpr_code

The object segment contains code normally found in the call, entry, or return operators. This code may stop working if the system's call/push/return conventions change.

These additions require only compatible changes to the object map and object_info_structures.

STACK HEADER CHANGES

The stack header will be grown to add pointers to four PL/I external entry operators. The new pointers can be used in testing without any system changes. However, by the time the proposed operators are installed, the pointers in the stack header must be initialized when the stack is created, and all programs that know about the size of the stack header should be recompiled.
/* BEGIN INCLUDE FILE ... stack_header.incl.pl1 .. 3/72 Bill Silver */
/* modified 7/76 by M. Weaver for *system links and more system use of areas */
/* modified 3/77 by M. Weaver to add rnt_ptr */
/* modified 3/80 by M. Weaver to add new entry op ptrs */
dcl sb ptr;
/* the main pointer to the stack header */

dcl 1 stack_header based (sb) aligned,

2 pad1 (4) fixed bin,
2 old_lot_ptr ptr,
2 combined_stat_ptr ptr,
2 clr_ptr ptr,
2 max_lot_size fixed bin(17) unal,
2 main_proc_invoked fixed bin(11) unal,
2 run_unit_depth fixed bin(5) unal,
2 cur_lot_size fixed bin(17) unal,
2 system_free_ptr ptr,
2 user_free_ptr ptr,
2 null_ptr ptr,
2 stack_begin_ptr ptr,
2 stack_end_ptr ptr,
2 lot_ptr ptr,
2 signal_ptr ptr,
2 bar_mode_sp ptr,
2 pl1_operators_ptr ptr,
2 call_op_ptr ptr,
2 push_op_ptr ptr,
2 return_op_ptr ptr,
2 return_no_pop_op_ptr ptr,
2 entry_op_ptr ptr,
2 trans_op_tv_ptr ptr,
2 isot_ptr ptr,
2 sct_ptr ptr,
2 unwinder_ptr ptr,
/* (0) also used as arg list by outward_call_handler */
/* (4) pointer to the lot for current ring (obsolete) */
/* (6) pointer to area containing separate static */
/* (8, 10) pointer to area containing linkage sections */
/* (10, 12) DU number of words allowed in lot */
/* (10, 12) DL nonzero if main procedure invoked in ru */
/* (10, 12) DL number of active run units stacked */
/* (11, 13) DU number of words (entries) in lot */
/* (12, 14) pointer to system storage area */
/* (14, 16) pointer to user storage area */
/* (16, 20) */
/* (18, 22) pointer to first stack frame on the stack */
/* (20, 24) pointer to end of last stack frame on the s */
/* (22, 26) pointer to the lot for the current ring */
/* (24, 30) pointer to signal procedure for current rin */
/* (26, 32) value of sp before entering bar mode */
/* (28, 34) pointer to pl1_operators $operator table */
/* (30, 36) pointer to standard call operator */
/* (32, 40) pointer to standard push operator */
/* (34, 42) pointer to standard return operator */
/* (36, 44) pointer to standard return / no pop operator */
/* (38, 46) pointer to standard entry operator */
/* (40, 50) pointer to translator operator ptrs */
/* (42, 52) pointer to ISOT */
/* (44, 54) pointer to System Condition Table */
/* (46, 56) pointer to unwinder for current ring */
2 sys_link_info_ptr ptr,
2 rnt_ptr ptr,
2 ect_ptr ptr,
2 assign_linkage_ptr ptr,
2 ext_entry_op_ptr ptr,
2 ext_entry_desc_op_ptr ptr,
2 ss_ext_entry_op_ptr ptr,
2 ss_ext_entry_desc_op_ptr ptr,
2 pad2 (26) bit (36) aligned;
/* (48, 60) pointer to *system link name table */
/* (50, 62) pointer to Reference Name Table */
/* (52, 64) pointer to event channel table */
/* (54, 66) pointer to storage for (obsolete) hcs_$as

The following offset refers to a table within the pl1 operator table. */
dcl tv_offset fixed bin init(361) internal static; /* (551) octal */

The following constants are offsets within this transfer vector table. */
dcl (call_offset fixed bin init(271),
push_offset fixed bin init(272),
return_offset fixed bin init(273),
return_no_pop_offset fixed bin init(274),
entry_offset fixed bin init(275)) internal static;

The following declaration is an overlay of the whole stack header. Procedures which
move the whole stack header should use this overlay.
/*
dcl stack_header_overlay (size(stack_header)) fixed bin based (sb);

END INCLUDE FILE ... stack_header.incl.pl1 */