XEROX Inter-Office Memorandum

To CSL and SSL

Date June 19, 1973

From Butler Lampson

Location Palo Alto

Subject Alto/Bcpl stcrage allocation and function calls

Organization PARC / CSL

1. Overview

This document describes the facilities provided by the standard Alto microcode for storage allocation and function calls. They are intended to support function calls and returns for languages which pass parameters by value and allocate the local environment or frame for a function when it is called, with a corresponding deallocation on the return. Provision is also made for statically allocated frames, for a global storage region which may change during a call or return, and for a coroutine linkage. Finally, function descriptors and return links are interpreted as segmented addresses, so that code segments can be swapped or relocated.

2. Background

This section discusses the major alternatives which were considered in arriving at the present scheme. Hopefully, it will make it easier for the reader to understand why things are done the way they are.

During a transfer of control from one function to another, it is necessary to:

a) obtain the address of the first instruction to be executed in the new function;

b) allocate storage for the new function or release storage fcr the old one;

c) possibly set up pointers to the environment of the new function;

d) pass parameters, possibly including a return link.

Item (a) is complicated by our desire to be able to move code or to keep code out of core, until it is referenced. Both (a) and (b) are closely related to the question of hcw memory is to be

allocated. Item (c) requires decisions about how much environment should be set up automatically.

2.1 Frame allocation

It is well known that two areas which grow and shrink at one end can be conveniently allocated in a single block of memory, by putting one at the top and growing it down, and putting the other at the bottom and growing it up. With more than two areas, life is much more difficult.

Bcpl programs need three kinds of storage: program, stack, and heap. The observation of the preceding paragraph motivates us to combine the stack and the heap, so that there will be only two areas. This arrangement results in a somewhat different kind of storage fragmentation than one gets with a stack, which we now proceed to examine.

A stack has several desirable properties:

S1) Frames are physically adjacent, so that parameters can be deposited by a caller at the end of his frame and magically appear at the beginning of the callee's frame.

S2) Allocating and freeing frames is quick.

S3) Freed space is automatically coalesced into a single block at the top of the stack.

On the other hand:

S4) All the space has to be permanently committed, whether or not it is used.

S5) Every process must have its own stack, because all the good features of the stack depend on the strict last-in first-out discipline.

The heap's advantages are:

H1) Space is committed only when allocated.

H2) Allocating and freeing frames is reasonably quick.

H3) Any number of processes or coroutines can use the same storage zone;

and the disadvantages are:

H4) Frames cannot be related by adjacency, but only by pointers.

H5) Fragmentation may occur.

The last point deserves more attention. It has two aspects:

F1) Let us assume that the heap allocator coalesces adjacent free blocks. If storage is allocated strictly last-in first-out, there will be no fragmentation. If more than one process is involved, however, or if storage which is allocated explicitly has a longer lifetime than the function which allocates it, there will be fragmentation.

F2) In the interest of efficiency, we probably don't want to coalesce every time a frame is freed, but rather to keep a reserve of frames which can be rapidly allocated and freed. This is a fairly small effect, however. If the depth of subroutine calls is 10, say, and we have frame sizes 10, 20, 30, and 40 words in common use, then the maximum amount of space in the reserve is 1050 words, and a more likely amount is 200-400 words. This is pretty conservative.

Adjacency is nice for passing parameters, but we can pass up to three values through the registers more cheaply. If we accept that few functions have more than three parameters, we pay little for giving up adjacency.

The above analysis leads to the conclusicn that allocating frames from a heap is a good deal. Details of the scheme are given in Section 3.

2.2 Program relocation

It would be nice to be able to relocate or swap out both programs and data. On a machine with about the same amount of real and virtual memory, a typeless language like Bcpl cannot get

much value out of data relocation. Program relocation, on the other hand, is quite feasible, since programs are addressed only by labels, functions, and return links. In order to avoid disrupting Bcpl, it is essential to have functions represented in one word. Furthermore, it seems highly desirable to minimize the amount of stuff which has to be permanently resident in core for the mechanism to work.

These considerations lead to a segmentation scheme, in which code is organized into segments, each of which is either not in core, or occupies a contiguous group of words in core. There are 256 segments, each with 128 entry points, so a function reference will fit in 15 bits. A return link, of course, has to be two words.

Each segment has an entry in a resident segment table, which tells whether the segment is in core, and if so, where. A call or return traps if the new segment isn't in core, and otherwise adds the code base from the segment table to the segment-relative program counter. To make this work, a call has to relativize the return link.

Labels are local to a segment, and hence can be stored relative to the code base.

2.3 Global environments

A function always has a local environment or frame. In many cases, it is also convenient to have a global environment G which might be common to all the functions in a segment or group of segments, but which may change during a call or return. This has two advantages:

G1) It allows a group of functions to share a collection of static variables.

G2) If we redefine the Nova's page Ø addressing to be relative to the global environment, it becomes much more useful, since a group of segments with a common G can safely allocate 256 globals which can be directly addressed, without having to worry about collisions with other segments.

The implementation is to keep the G base in the segment table together with the code base. Because of the Alto's double-word

fetch, this costs very little. If several segments have the same G, the base value is simply duplicated in the segment table.

It is worth noting that this scheme of keeping both global and code base in the segment table allows either to be varied over a group of segments, while the other is kept constant. By varying the global base we get a number of instantiations of the same code, each with different static data. By varying the code base we get a group of code segments sharing the same static data.

2.4 Return links

We want to minimize the amount of information in a return link. The smallest pice of information which can define the program state when a call occurs is the address of the caller's frame. We store the segment number and relative PC of the caller in his frame, and can thus restore the entire state (PC, frame, G) from the address of the frame.

2.5 More on frame allocation

The frame allocation problem has some additional aspects:

choice of frame size;

static rather than dynamic frames;

coroutine linkage.

Since each function in a segment may have a different frame size, we want to specify the frame size in the entry point as well as the relative address of the first instruction. Again, the double-word fetch makes this cheap.

We would like to de-couple the compiler's specification of needed frame size from the storage allocator's decision about how many sizes to provide. This requires an interface convention for specifying frame size, i.e., a set of frame sizes from which the compiler can choose. A reasonable set of frame sizes might be 8, 12, 16, 20, 24, 32, 40, 48, ... Of course, we don't want to build this into the microcode, so we will number the available sizes 1, 2, 3, ... The compiler will then specify the desired size by compiling in the proper number, and the storage allocator

must be able to do something intelligent for each frame size. If the allocator wants to provide fewer frame sizes than the ones defined in the compiler-allocator interface, it needs a way to:

A1) Map a specified frame size into a larger one.

A2) Trap on a specified frame size larger than what the allocator can handle automatically.

It is also desirable to be able to easily switch the storage allocator's data base.

When a function call occurs, we usually want to allocate a frame. Sometimes, however, we want a function to have a permanently allocated frame, either to speed up the call or to make the local variables static. This can be conveniently done with a flag in the frame size word which causes it to be reinterpreted as a frame address.

Coroutine linkage requires a different approach, since a coroutine must be specified by the address of its frame rather than by an entry point. Since the frame contains the saved PC and the segment number, from which the global base and the code base can be obtained, it provides all the information needed to resume execution of the coroutine. To implement this idea we need a flag in the function reference; if the flag is set, the function reference is interpreted as a frame address and a coroutine call (cocall) occurs. This arrangement makes it unnecessary for the compiler to know whether a function call is a cocall or not.

3. The storage allocator

The microcode which implements storage allocation knows how to do two things:

1) Allocate: accepts a bead size BS and either returns a contiguous group of words of this size, called a bead, or traps.

2) Free: accepts a bead and either returns it to free storage or traps if this is too difficult.

The storage allocator works with a collection of free storage, called a zone, which is defined by the address AT of an allocation table and a maximum bead size MAXBS which specifies

the length of the allocation table. The contents of AT!BS tells how to allocate a bead of size BS, unless BS > MAXBS, in which case the allocator traps by calling AT!Ø. The contents of AT!BS is either Ø or the address of a free bead of the proper size to satisfy a request for size BS. A free bead has the form:

word	contents
-2	address of AT!BS
-1	Ø
Ø	address of next bead of size BS, or Ø if there are no more

The -2 and -1 words are not used by the allocator, but are used by the frame deallocator. The allocator traps by calling AT!2 if AT!BS=Ø. Both traps begin with the AC's untouched.

BS must be odd. In other words, only odd entries of AT are used. A bead, on the other hand, starts on an even location. Hence, if AT!BS is odd, it cannot be a bead address and instead is interpreted as the address of another AT entry which is used instead of AT!BS. This mechanism allows a number of bead sizes to be allocated from the same list of free beads. By making the last bead on the free list of size BS point to another AT entry, rather than to Ø, it is possible to automatically allocate a larger size bead if no beads of the proper size are available.

The deallocator has two entry points. Both take an (even) address F.

D1) Free frame: return F to the list at F!(-2) (i.e., F!Ø ← F!(-2)!Ø; F!(-2)!Ø ← F. If F!(-1) ≠ Ø, set F!(-1) ← Ø, and free the frame F!(-1). This permits additional dynamic storage to be freed automatically when a frame is freed.

D2) Free bead: return F to the list at F!Ø (intended for static frames).

In either case, the deallocator does nothing if F!Ø (or F!-2) = Ø, and it traps by calling AT!4 if F!Ø (or F!-2) is even. This allows software to free beads which have BS < MAXBS, or which require special treatment for some other reason. The trap leaves F in AC3.

It would be prudent to provide the trap routines with static frames, since an infinite loop of traps may ctherwise result.

An allocation table has the form:

word Ø: trap routine for FS > MAXFS

word 1: free list for FS = Ø

word 2: trap routine for empty free list

word 3: free list for FS = 1

word 4: trap routine for deallocating if F!Ø (or FS!(-2)) is even

word 5: free list for FS = 2

word 6: unused

word 7: free list for FS = 3

word 8: unused

...

There are three instructions which invoke the storage allocator. All are parameterless:

ALLOCATE: takes BS in AC3, returns address of bead (word Ø) in AC3

FREEF: takes F in AC3, leaves ACs unchanged. Frees a frame.

FREEB: takes F in AC3, leaves ACs unchanged. Frees a bead.

4. Code segmentation

Any reference to a code segment which is not local to the segment must be a segmented address. Such an address may take two forms:

S1) Entry reference: one word, with the format:

bits 0-6: entry number EN (allows 128 entries/segment)

bits 7-14: segment number SN (allows 256 segments)
bit 15: Ø

S2) Instruction reference: two words, with the format:

word Ø: program counter PC, relative to code base of segment

word 1: segment number SN

The segment number SN is an index into a segment table which starts at a fixed location SEGTAB. An entry in this table is called a segment descriptor SD. It occupies two words and has the format:

word Ø: global base for the segment

word 1: code base for this segment (must be even). If the word is odd, the segment is said to be not present, and any attempt to transfer into this segment will cause a trap to a fixed location NOTPRESENT.

Note that it is possible to have several segments with the same code base (different incarnations of the code), or with the same global base (sharing the globals), or both (which doesn't seem too useful).

At the beginning of each code segment is an entry table, whose double-word elements are entry points, with the format:

word Ø: frame size FS

even = static frame. FS is the frame address.

odd = dynamic frame. FS is the index into the allocation table.

word 1: address of first instruction, AFI, relative to the code base.

Note that a static frame should have F!(-2) = Ø so it won't be freed during a return.

There are no instructions which directly invoke any of this machinery.

5. Call and return

The operand of a call instruction is a word called a function reference FCN. There is also a second operand, the number of arguments NA, which is obtained in a clever manner described below. Once the instruction has its operands, it proceeds as follows:

C1) Relativize the PC and store it into SAVEDPC, which is F!Ø. Add 1 if NA = many.

C2) If FCN is even, this is a coroutine call (cocall) and FCN is the address of a frame. Obtain NEWPC by fetching the instruction reference (SAVEDPC, SN) from (FCN!Ø, FCN!1) and evaluating it. This also sets up GB and CB. Then go to step C5.

C3) If FCN is odd, it is a segmented address. Obtain the SN, GB, and CB for it. Fetch the entry point (FS, AFI) and compute NEWPC ← CB + AFI.

C4) If FS is even, the function has a static frame: NEWF ←FS. Otherwise, allocate a frame of the size specified by FS, put its address in NEWF, and put SN in NEWF!1.

C5) Store F in NEWF!Z. Set F ← NEWF.

C6) Store the number of arguments specified by NA (3 if NA = many): F!3 ← ACØ, F!4 ← AC1, F!5 ← AC3. Set ACØ ← NA.

C7) Send control to NEWPC if NA = many, NEWPC + 1 otherwise.

A return has no arguments. It proceeds as follows:

R1) NEWF ← F!2.

R2) Free frame F.

R3) F ← NEWF. Fetch the instruction reference from F as in step C2, and set up GB and CB.

R4) Transfer control to NEWPC.

Observe that return does no automatic storing of arguments.

There are two groups of 5 call instructions:

CALLPØ, CALLP1, CALLP2, CALLP3, CALLPn

CALLFØ, CALLF1, CALLF2, CALLF3, CALLFn

They are ten of the address-only new instructions. The CALLPs use PC!a!Ø as the FCN operand, and the CALLFs use F!a, where a is the address field. The NA operand is obtained from the opcode. Note that the first instruction of the function is skipped unless NA = many (CALLxn). This instruction should be a call to a routine which copies the extra arguments. The saved PC is incremented by 1 when NA = many to skip over a word which will tell this routine how many arguments there are and where to find them.

There is a single parameterless return instruction.

The format of a frame is:

name	word	contents
	-2	pointer to list header to return frame to, or Ø
	-1	pointer to list of dependent beads, or Ø
SAVEDPC	Ø	PC, relative to code base, when this frame has done a call
SN	1	owner's segment number
OLDF	2	caller's frame
	3	first argument
	...	