Inter-Office Memorandum

To	Distribution	Date	November 29, 1977
From	John Wick	Location	Palo Alto
Subject	External links in Mesa	Organization	SDD/SD

XEROX

DRAFT

Filed on: [MAXC]<WICK>CODELINKS.BRAVO

This memo describes a proposed implementation for optionally storing external links in the code segment (rather than in the global frame, as is currently done). It also outlines some other related changes in the structure of global frames, code segments and BCDS, and examines a number of outstanding issues which need to be resolved before implementation can begin.

The basic idea is to reserve a bit in the global frame which indicates whether the external links are stored in the global frame or in the code segment. The current global function call instructions (GFCn and GFCB) would be replaced by instructions which checked this bit on each call. Alternately, a link register could be set up on each change of the global context to point to the base of the linkage vector.

Storing Links Backwards

Although the global frame overhead (now ten words) will be reduced, the fact that external links currently start at a fixed offset from the beginning of the frame gives rise to a much larger *effective* minimum size (since all interesting programs have at least one external link). External links begin at offset 18 from the beginning of the frame, allowing for eight global variables (and ten words of overhead). Thus a module with at least one external link and no globals wastes eight frame locations.

A solution to this problem, suggested by Butler Lampson, is to store the links backwards from the frame (or code) base. When the links are stored in the code, this is the only reasonable format, since it avoids conflict with the entry vector. This requires a relatively simple change in the microcode to locate the desired link. The software which allocates and frees global frames must also be changed, and the format of frame fragments in BCDS may also need revision (see below). But this change could be made without implementing the option for links in the code segment (although there is currently no reason to consider doing so).

External Link Access

Externals referenced by a module are stored in a *linkage vector* (sometimes called a *transfer vector*), which is a contiguous portion of the module's frame. The vector is set up by the loader to contain all of the items imported by the module -- procedures, signals, programs, and frames. Special instructions are implemented for accessing imported procedures; the other items are accessed like normal global variables. The following revisions are planned:

External Procedure Calls

As mentioned above, the current global function call instructions (GFCn and GFCB) will be replaced by a corresponding set of external function calls (EFCn, EFCB) which obtain the link by indexing backwards from either G or C, based on a bit in the global frame. There appears to be no need for instructions which unconditionally reference the frame; the existing set of load and store instructions can be used to handle procedure, signal, program, and frame variables, with calls implemented by SFC.

External Signals and Programs

These new instructions handle references to external procedures only. If external signals (and programs) are also considered part of the linkage vector (as they are now), some method for accessing them is also required. Since the only operations on these interface items require the value on the stack, we propose a load link instruction (LLKB) which behaves like GFCB except that it pushs the desired link on the stack instead of calling it. This should require little additional microstore, since the code for fetching the link must be there anyway. Because the inline code to load a link is long and ugly (isolate the option bit, test it, perhaps lock the code or disable interrupts, index backwards, load the link, perhaps enable interrupts), a KFCB is the only other reasonable alternative.

Imported Frames

Imported frame pointers are the only remaining problem. Clearly they can be handled using LLKB if they are also part of the linkage vector, but it may be better to treat them as variables and allocate them unconditionally in the global frame in order to get a better instruction mix. This would save at most one code byte per procedure access (since DESCBS and FDESCBS both require the imported frame pointer on the stack); the savings for variable accesses would be more. On the other hand, this option would require splitting the linkage vector into several pieces, which means revising the BCD format and complicating NEW and UNNEW (which we might want to do for other reasons anyway; see below).

Number of Links

The design so far limits the number of externals (the sum of procedures, signals, programs, and perhaps imported frames) to 256. This seems reasonable (old data indicates an average of 12.3 external procedures per module), but if not, we could provide a KFCB which, given an offset, would check the option bit and return the link from either the frame or the code segment. This would be terribly inefficient, but it shouldn't happen very often.

Global Frame Structure

Location of Option Bit

To make life easy for the microcode, the option bit (codeLinks: BOOLEAN) should be allocated with the code base, which must be fetched anyway (this argument only makes sense if the linkbase register implementation is used; if the bit is checked on each link access, it doesn't much matter where it is). There are two options: 1) use a low-order bit, thereby requiring the code to be aligned on a quad-word (rather than a double-word) boundary, or 2) use one of the eight leftover high-order bits of the 24-bit code pointer.

The latter alternative isn't very attractive on the Alto, where the second word of the pointer can be used for something else, and it precludes the use of the 32-bit long pointer format on the Dstar. (Although the PrincOps specifies a 24-bit pointer, I see no reason why a 32-bit pointer should be ruled out, as it might prove convenient when packing the code for several modules into a single segment; more on this below.) Since quad-word allocation is already built in for frames, it shouldn't cause any additional problems for code, so the first alternative seems more attractive.

Access to Code Pointer

In any case, the compiler needs easy access to the pointer portion of the codebase field for reading constants from the code segment. It currently obtains it using LGB (reading the address directly from the frame, knowing that the mappedOut bit will be zero). This won't work if there are other bits strewn about in the pointer (besides codeLinks, an XFER trapping bit is under consideration). So the read register instruction needs to be expanded to include C.

Frame Size Index

Even though all of the global frames for a configuration are now allocated compactly in a single data segment, each frame is set up with a "hidden" word at offset minus one containing the frame size index (*fsi*), so that it can be added to the free frame heap if an UNNEW is done on the module. This is somewhat wasteful, given that most modules will never be deleted (exception: once-only initialization code).

With the links stored backwards from the global frame pointer, this practice will become more wasteful, since both the frame pointer and the location of the *fsi* (plus one) must be allocated on quad-word boundaries. We might consider packing the frames as tightly as possible, without reserving space for the *fsi*, and give up the ability to reuse the frame of a deleted module. Alternately, we could restructure the frame into a smaller one (with all the appropriate fields and alignment) and free that. Another possibility is to indicate somehow in the configuration description which modules will be deleted and record this in the BCD.

Note that this applies only to global frames allocated as a result of loading a configuration. Those allocated as a result of copy NEW are assumed to be of shorter lifetime and will come from and be returned to the free frame heap. (Actually, frames for single module configurations are also obtained using ALLOC.) It is probably worthwhile to add a field alloced: BOOLEAN which indicates where the frame came from, so that UNNEW can do the right thing.

Global Frame / Code Segment / Bcd Tradeoffs

A number of additional fields are needed in order to get all these pointers set up properly, both on module instantiation and when the code is swapped (mapped) in and out. I have tried to push these fields into the code or the BCD if possible, to conserve resident space.

Number of Links

The copy NEW and UNNEW operations need to know the number of external links (perhaps rounded up to a multiple of four), so that the actual frame base and frame size can be determined. (If links are in the code, the frame size should be decreased and the code size increased by the number of links.) This can be stored in the second word of the code segment prefix, where it is now (the link base is no longer needed, since it will always be minus one). It seems reasonable that these operations access the code segment, since it is about to be or has recently been swapped in (and NEW needs to get the frame size from the code segment anyway). It doesn't seem reasonable to access the BCD for these operations, since a lot of mechanism is required to locate the proper one.

More on Imported Frames

As mentioned above, it might be desirable to allocate some of the external links (notably imported frames) unconditionally in the frame, to take advantage of the better instruction set. There is one drawback.

It is currently possible to find all of the external links in a module by looking at only the code (the link base and number of links are stored there, and the links are in one contiguous chunk of the frame). This is used in two situations:

> When doing a copy NEW, the links are examined for bindings to the source (old) frame, and updated to point to the destination (new) frame. Presumably we could do away with this (rather obscure) feature, since the compiler goes to considerable length to convert bindings to self into local accesses.
>
> (Aside: when doing a regular new, the location of the links is determined by the frame fragment(s) in the BCD. Currently there is only one, but multiple fragments will be necessary to implement *Global Frame Templates* (see the Mesa task list)).
>
> Although it is not implemented in the current system, it should be possible, when doing an UNNEW (or more generally, unloading a configuration), to find all the bound references to the dying module(s) and unbind them. This is not just to catch calls on the deleted module. The loader will not overwrite bindings that appear to be valid, so if some other implementer is to be loaded later, the bindings to the deleted implementer must be undone.

In any case, if we allow any of the external links to float around in the frame, it will take an unreasonable amount of space to describe their locations in the code segment. It is probably best to provide separate operations that unbind as well as UNNEW by going through all the BCDs; and to leave the semantics of UNNEW as they are now: no unbinding.

We now have only to design a reasonably storage efficient scheme for describing multiple frame fragments in the BCD. All my attempts with sequences of sequences running backwards and forwards from various frame bases look terrible. Help!

Locating the Code

Define the *code origin* to be the beginning of the segment containing the code (which is currently always a page boundary), and the *code prefix* to be the location of the code prefix and entry vector (the contents of the C register); the difference between these is the *code offset*, which is now always zero. If links are stored in the code, the offset is just the number of links, rounded up to a multiple of two (or four or perhaps eight). Note that when swapping in the code, we have a problem: we need to know the offset in order to find the code prefix, which contains the number of links, from which we can compute the offset . . . We could of course store the number of links at the beginning of the code as well as with the prefix, but there is a better way (I think).

We can get around this problem by initially storing the offset in the BCD, and transfering it to (the least significant word of) the codebase field in the frame when the module is instantiated. Thereafter, when the code is swapped in, we add the segment origin to codebase to obtain the code prefix address; when it is swapped out, subtract the origin from codebase to recover the offset. If the Dstar allows 32-bit pointer format for the codebase, the offset can be left undisturbed, and only the page pointer needs to be modified.

The advantage of this scheme is that code packing -- i.e. packing the code for several modules into a single segment, swapped as a unit -- is obtained for free; we have only to set up the offsets to skip the code for preceeding modules (and take into account code links where space is allocated for them). If the BCD is further modified to contain the number of code words (it currently contains pages), we can tell whether space for code links has been reserved or not, but a field codeLinks: BOOLEAN in the module table entry would be more convenient.

More Unresolved Issues

The most difficult unresolved problem is how and when the option for code links should be specified. Current thinking is that some combination of the binder and the loader (and don't forget Bootmesa) should implement this option. The reasons are as follows:

> All module instances which share the code links must be bound the same (this is thought to be the common case). Since the number of module instances is usually a binding time (and sometimes a runtime) decision, the code compiled should be the same regardless of which option is used.
>
> If links are stored in the code, space must be allocated for them in the file as well as in memory, but it shouldn't be allocated unconditionally. Since the binder already has facilities for shuffling the code around and rebuilding the BCD, and it knows all about external links, it is the logical choice for making code links optional. It should mark the code as well as the BCD in some way, so that the loader (and NEW) will know that code links are possible.
>
> The loader can arrange to put the links in either place as an option, assuming space for the links has already been provided in the file by the binder. Code links will slow down the loading operation considerably, as each code segment will have to be swapped in and rewritten. It is also not clear whether the option should apply to configurations or individual modules.

An alternate proposal is to modify the binder to allow binding to image files, thereby nailing down all the *gfis* in the configuration. This would improve loading speed considerably even in the frame link case, and eliminate rewriting code segments that contain links, since all the links can be determined at binding time. The cost is a considerable loss of generality in the combinations of configurations that could be loaded together.

Distribution:
Charnley
Geschke
Lampson
Lynch
Mitchell

Redell
Satterthwaite
Simonyi
Thacker
Mesa Group
Mesa Users