*Proc. 1st International Symposium on Architectural Support for Programming Languages and Operating Systems, ACM Sigplan Notices **17**, 4 (Apr. 1982), pp 66-75*

Fast Procedure Calls

Butler W. Lampson

Xerox Research Center
3333 Coyote Hill Rd.
Palo Alto, CA 94304

Abstract

A mechanism for control transfers should handle a variety of applications (e.g., procedure calls and returns, coroutine transfers, exceptions, process switches) in a uniform way. It should also allow an implementation in which the common cases of procedure call and return are extremely fast, preferably as fast as unconditional jumps in the normal case. This paper describes such a mechanism and methods for its efficient implementation.

Key words and phrases: architecture, call, frame, procedure, registers, stack, transfer.

CR categories: 6.33, 6.21

1. Introduction

Well-structured programs typically make a large number of procedure calls; one call or return for every 10 instructions executed is not uncommon [4]. The cost (in space and time) of a procedure call is therefore a critical element in deciding how well a machine supports a programming language. This cost depends on three things:

> the *calling sequence* generated by the compiler;
>
> the *operations* or machine instructions from which the compiler must compose its calling sequence;
>
> the *speed* provided by the implementation of the operations, which determines the speed of a call and return.

This paper considers only compilers which generate a reasonably general-purpose *transfer of control* for each *procedure call* in the source program. It neglects inter-procedural analysis which might rearrange the generated code so drastically that the connection between source procedure calls and object transfers of control is no longer recognizable. Although this kind of analysis may someday play an important role, there has been negligible experience with it to date. Also, we consider only Algol-like languages; these include Pascal, Mesa and Ada, and the same methods will work with only slight modification for Lisp. Many of the ideas can probably be used with Fortran or Cobol also, but we have done no detailed analysis or empirical studies for these cases.

The importance of control transfers has been recognized for a number of years, and recent machine architectures such as the DEC VAX [6] and the Intel iAPX 432 [1] have fairly elaborate operations which are intended to support such transfers. In the current implementations, however, transfers are quite slow [4]. In addition, most such architectures can support only a strictly last-in first-out pattern of transfers, which is unsuitable for coroutines, retained frames, and multiple processes. Under this restriction, each coroutine or process needs a contiguous piece of storage large enough to hold the largest set of frames it will ever have; this makes efficient storage allocation difficult.

We briefly present an abstract scheme for supporting very general control transfers (§ 3), and then describe several possible implementations:

I1) a very straightforward one which models the abstraction in an obvious way (§ 4);

I2) a refinement of I1 which takes much less space, at the expense of some rather tricky encodings and some extra levels of indirection (§ 5);

I3) an optimization which allows instruction fetching to proceed as fast as it does for an unconditional branch (§ 6);

I4) another optimization which reduces the cost of passing parameters and allocating storage for a procedure instance (§ 7).

The main point of the paper is that an extremely general and flexible control transfer mechanism can be supported, and yet simple Pascal-style calls and returns can be executed as fast as in the most specialized mechanism. Indeed, they can be as fast as unconditional jumps at least 95% of the time.

I1 and I2 are realized in the Mesa processor architecture [2], which has been implemented on four machines, including the Alto [8] and the Dorado [9].

2. Levels of abstraction

In describing the design for control transfers and its various possible implementations, we need to distinguish clearly among the several levels of abstraction that are involved. Most abstractly, we have a *model* for control transfers. The source language programmer deals with transfers in terms of this model, and should not be affected by changes at any lower level of abstraction. From his point of view, there is some procedure RUN_S which runs his program.

This procedure is typically implemented by a *compiler* which translates the source program into an object program expressed in some *encoding* (machine instructions plus auxiliary data structures). The encoded program is then executed by an *interpreter*, implemented in hardware, microcode, machine instructions or some combination. Thus $\text{RUN}_S(\text{source}) = \text{RUN}_E(\text{TRANSLATE}_S(\text{source}))$, where the compiler implements TRANSLATE_S and the interpreter implements RUN_E. Changing the interpreter does not affect the encoding or the compiler: it is done whenever a program is moved from one model of a compatible computer family to another. Changing the encoding affects the compiler and the encoded programs, and hence requires recompilation. If done correctly, it does not affect the source programs, and hence is an optimization method which can be used whenever it produces worthwhile cost savings or performance gains.

We have avoided the term *architecture* in this description; although it is often used for what we have called the encoding, its meaning has become sufficiently vague that a narrower word seemed desirable.

3. A control transfer model

Our abstract model for control transfers is described in detail in [3]; this section outlines it briefly. It has two elements:

contexts, the entities among which control is transferred;

XFER, the primitive operation for transferring control.

A context normally corresponds to the activation record or *local frame* of a procedure. It contains

the program counter for that activation;

the arguments and local variables;

references (pointers) to any other environment information, such as static (**own**) data, or activations of lexically enclosing procedures.

The XFER primitive takes a single argument, the *destination* context where execution is to continue. It works in conjunction with two global variables:

returnContext, which holds the context to which control should return; normally, but not always, this is the one executing the XFER;

argumentRecord, which holds the arguments being passed in the transfer.

The effect of XFER is to suspend execution of the currently running context and begin execution of the destination, which is expected to retrieve the *returnContext* and *argumentRecord* if it is interested.

To call a Mesa (or Pascal, or Algol) procedure, more is needed than a simple transfer of control: a new context must be constructed for the new procedure activation. Abstractly, this is done by providing a *creation* context for the procedure. The code of this context is an infinite loop; on each iteration it creates a new context for the procedure, and forwards control to it:

```
WHILE TRUE DO
  newContext: Context = CreateNewContext [arguments
    to initialize the program counter and other state];
  XFER[newContext]; -- note that returnContext and
    argumentRecord are unchanged--
  END
```

In practice, of course, this is such a common operation that a special case is required, and all our implementations have a special kind of context called a *procedure descriptor*, which consists of a pair (pointer to procedure, pointer to environment). An XFER to such a context results in the actions described by the code above.

When the new procedure gets control, it saves the *returnContext* in one of its local variables called the *returnLink*, and it copies the arguments from the argument record into other local variables. Implementations usually store the argument record in registers if it isn't too large, so that this is efficient. When the procedure returns, its context is normally freed. Abstractly, this is done by transferring to a special context which does the freeing. Again, actual implementations provide a single operation called

RETURN which retrieves the *returnLink*, frees the context, sets *returnContext* to NIL, and then does XFER[*returnLink*].

The essential features of the model are these:

F1) Everything required to resume execution is contained in the context. Hence a single pointer to a context suffices for a return link, and every procedure descriptor includes an environment reference.

F2) Contexts are first-class objects which are allocated and freed explicitly, and not necessarily in last-in first-out order.

F3) Any context may be the argument of any XFER (provided the argument and result types match), so that a choice between procedure call, coroutine transfer or some other discipline is made by the destination context, not the caller.

F4) Arguments and results are handled symmetrically by XFER itself; of course the destination context may treat them differently, e.g., storing arguments in local variables, and using results to continue a computation.

Some languages, including Mesa, have a notion of a *cluster*, *package*, or *interface*, which is a collection of procedures grouped under a common name. An interface called *IO*, for example, might contain procedures *Read*, *Write*, and so forth. A particular procedure in the interface is denoted by a qualified name, e.g., *IO.Read*. Among other things, interfaces simplify the task of linking up a reference to an external procedure such as *IO.Read* (from a client program) with the procedure itself. If the client and the implementation use the same interface definition, they will agree on the position of the *Read* procedure in the interface record. Then the client needs only a pointer to the interface record in order to call any of its procedures. The components of an interface record will be contexts for the various procedures.

4. A simple implementation

The natural implementation of this model represents a context by a pointer to a record whose components are the elements of a local frame:

- the program counter,
- a pointer to each enclosing environment (i.e., to global variables, and to lexically enclosing procedures),
- a return link (another context),
- a component for each argument,
- a component for each declared local variable,
- a component for each temporary variable.

Thus, as required, the context provides all the information needed to continue execution.

The frame is allocated from a heap. Normally there is a single reference to each allocated frame. While the context is in execution, this reference is held in the state variables of the process in which the context is running (and hence in some processor register if the process is actually assigned to a processor). In fact, this is the *only* information needed for the process to execute: it needs other state variables only to control scheduling, timeouts, priority and other things having nothing to do with the sequential execution of the process. When the context has called another one, the single reference to its frame is either in the global *returnContext*, or later in the *returnLink* component of the called context's frame.

Having only one reference to a frame is very convenient, because it ensures that the possessor of the reference can free the frame without having to worry about dangling references, and indeed this is normally what happens on a return. This implementation can readily handle frames which must outlive a return, however. Such frames are called *retained*, and are distinguished by the possible existence of multiple references. Other methods (e.g., garbage collection) are needed to determine when a retained frame can be safely freed. The model and this implementation can easily accommodate both fully general retained frames, and a very efficient but safe method of freeing frames which are used conventionally.

Actually, a context is not simply a local frame pointer, since the common case of a procedure descriptor demands special treatment. Instead, it is a variant record of the form:

```
Context: TYPE=RECORD [
    CASE tag: {frame, proc} OF
        frame =>    [FramePointer];
        proc =>     [code: ProcPointer, env: EnvPointer]
        ENDCASE]
```

The *frame* case is for a return link or any other reference to an already existing context. The *proc* case is for a procedure descriptor; recall that this is an abstract context which constructs the context for a procedure. As we saw in the last section, this abstract context does a highly stylized job, parameterized only by the address of the first instruction for the procedure (the *code* component), and a pointer to the environment for the procedure (the *env* component). It may be implemented by a runtime routine (this is common in Algol and PL/1 implementations), by some combination of instructions in the calling sequence and in the prologue of the procedure, or by microcode.

A procedure return involves a similar abstract context, which likewise may be implemented by a runtime routine, by inline instructions, or by microcode. It frees the current local frame (unless it is retained) after picking up the return context from its *returnLink* component. Then it sets *returnContext* to NIL (an attempt to return from this return would be an error), and does an XFER to the context it obtained from the *returnLink*. This is a pointer to the caller's local frame, so execution resumes in the caller's context, at the instruction pointed to by its *PC* component.

In this implementation, the processor has registers which hold

> *LF*, a *FramePointer* to the frame for the currently executing context;
>
> *PC*, a *ProcPointer* to the next instruction to be executed (in principle this is a component of the frame, but any reasonable implementation will keep it in a register, and store it into the frame only when control leaves the context);
>
> *returnContext*, which is normally set implicitly by a call (to the current context) or by a return (to NIL);
>
> a stack or some working registers for evaluating expressions, or for passing arguments and results.

Each context must leave the arguments or results on the stack or in the working registers before doing an XFER operation. When control enters a context after an XFER, it expects to find its arguments or results on the stack, and must retrieve them before doing another XFER operation. Since the number of processor registers is finite, an argument or return record can be so large that it will not fit. When this happens, space is allocated from the heap to hold the record, and a pointer is passed in one of the registers. Such *long argument records* are treated like local frames for the purposes of allocation: there is just one reference to each one, and the receiver can therefore free it as soon as he is done with it.

A call to a fixed procedure (whose value is supplied by the compiler or linker) is represented by including the procedure descriptor as a literal in the program. Thus *f* [] in the source results in LOADLITERAL *f*; XFER in the encoding. A call to a procedure in an interface, such as *i.f* [], results in LOADLITERAL *i*; READFIELD *f*; XFER.

5. The Mesa implementation

This section describes the implementation of the control transfer model of § 3 which is used in the current Mesa processors. In describing this and subsequent variations, it is convenient to divide them into three more or less independent parts:

> obtaining the destination program counter;
>
> passing arguments or results;
>
> allocating or freeing a frame.

The Mesa implementation is part of a complete encoding for Mesa object programs which is described in [2]. The main design criterion in this encoding is economy of space. It uses instructions which are one, two or three bytes long; about two-thirds of the instructions compiled for a large sample of source programs occupy a single byte. The encoding uses a stack, rather than registers, for working storage to save address bits, and is heavily optimized for references to local variables stored in the frame of the current context. A somewhat similar design is described by Tanenbaum [7], though its handling of transfers is quite different.

The implementation of transfers is very similar in structure to the one described in the last section. The main differences are a number of optimizations which save space in the encoding. Most of them depend on a single idea: changing a full memory address to an index into a table, and storing the original address in the table entry. Whenever the address is needed, it is retrieved by indirection through the table entry. This scheme has several advantages:

T1) If the full address takes *f* bits, the table index takes *i* bits, and the address is used *n* times, then the space changes from *nf* to *ni*+*f*. The saving varies greatly, depending on the maximum number of objects (which determines *i*) and how often each one is used. For example, if $n=3$, $i=10$ (1024 table entries) and $f=32$, then $96-62=34$ bits are saved, or about one-third.

T2) The memory location of the object can easily be changed, since only the table entry needs to be updated. Of course the location of the table entry cannot be changed, but since it is usually fixed size and small, while most interesting objects are variable size, this is a clear gain.

T3) If there are several objects with a common part and different variable parts (e.g., instances of the same procedure with different environments), several table entries can point to the same common part, and the variable parts can be stored in the table entries themselves.

Of course the scheme also has a cost: the time taken for the level of indirection.

Mesa organizes procedures into groups called *modules*; typically a module implements a single abstraction. The procedures in a module share global variables; the collection of these variables is called a *global frame*. It is possible to have several *instances* of a module, each with its own global variables. The entire module is compiled together; hence compile-time binding of intra-module addresses is possible. The code for all the procedures is collected in a code segment; the base address of this segment is called the *code base*.

The four machines which have realized this design to date have used rather unspecialized microprogrammed processors to implement it. These processors use a modest number (20-40) of registers, and keep nearly all the state in main storage.

5.1 Obtaining the program counter

The Mesa encoding uses four levels of indirection for a typical external procedure call. They are diagrammed in figure 1. The four tables (in the order they are encountered during a call) are:

> A *link vector* LV associated with a module, with a 16 bit entry for each procedure called statically from the module; the entry holds the procedure descriptor. One of these procedures can be addressed within the module by a link vector index.

A *global frame table* GFT with a 16 bit entry for each module instance; the entry holds the address of the global frame for the instance.

A collection of *global frames*. These are not of the same size, but they are limited to a 64k segment of the address space and are quad-aligned; hence 14 bits is enough to address a global frame. In addition to the global variables of the instance, a global frame holds the code base; this is an application of point (3) above.

An *entry vector* EV associated with a module, with a 16 bit entry for each procedure in the module which holds the address of the procedure's first byte (relative to the code base). This first byte gives the size of the procedure's frame (see § 5.3), and the procedure's code starts at the following byte. EV starts at the code base.

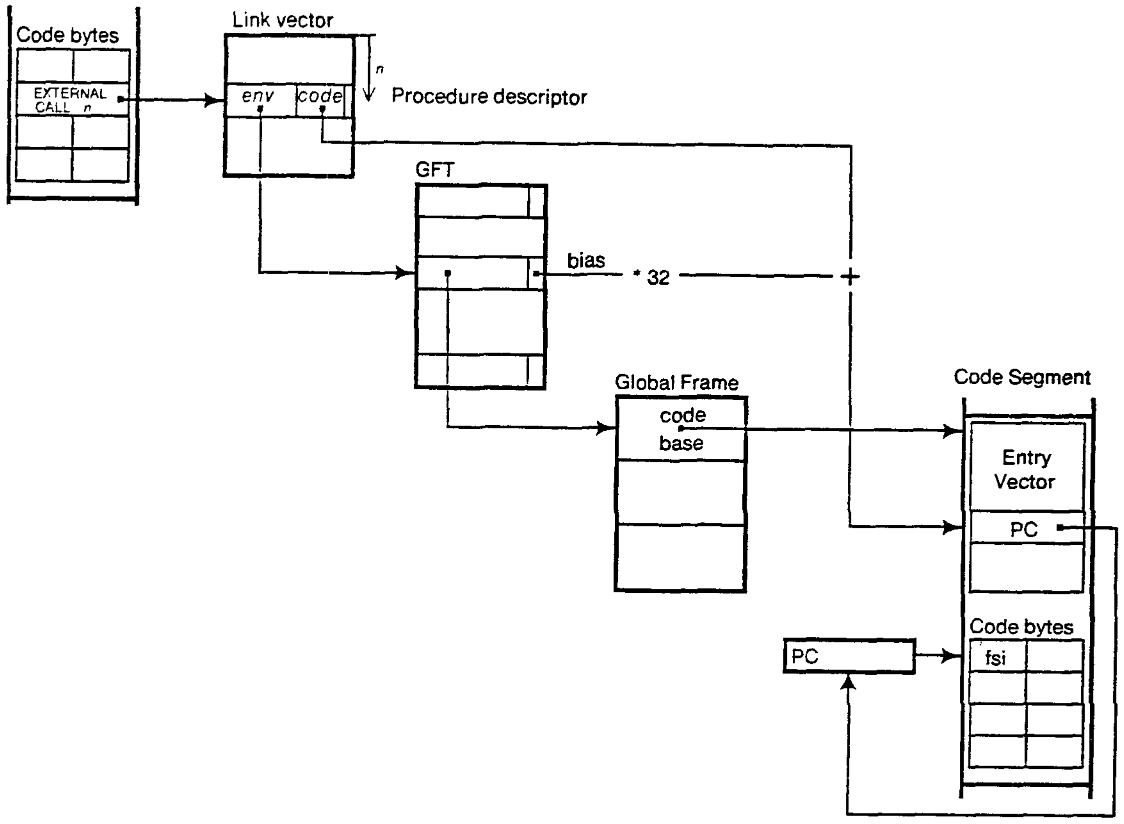

Figure 1: Levels of indirection in a procedure call

A typical procedure call proceeds as follows. First the arguments are pushed onto the stack. Then there is a one or two byte EXTERNALCALL instruction which contains a LV index. There are a number of one-byte opcodes, so that the (statically) most frequently called procedures in a module can be called in a single byte. A single opcode with a one byte address field allows 256 procedures to be called in two bytes. The context is retrieved from LV. Normally it is a procedure descriptor, which is a variant record of the form given in § 4. It is packed into a 16 bit word, with a one bit tag, a ten bit *env* field, and a five bit *code* field. The *env* field is a GFT index, from which the address of the global frame is retrieved. Then the code base is retrieved from the global frame. Finally, the *code* field, which is an EV index, is used to obtain an EV entry which when added to the code base yields the instruction address. The EXTERNALCALL instructions put the current context into *returnContext*, and store it automatically in the *returnLink* component of the newly allocated frame.

Since the *code* field is only five bits, a module can have only 32 entry points with this scheme. The two spare bits in a GFT entry are used to specify a *bias* for the entry point, in multiples of 32. Thus a single module instance may have up to four GFT entries, all pointing to the same global frame, but with different biases, for a total of 128 entries. This rather wasteful scheme is seldom needed, but it provides an important escape hatch for excessively large modules.

Each level of indirection allows a compact encoding of the reference to it:

> LV permits a compact call instruction;
>
> GFT permits a compact *env* field in a procedure descriptor;
>
> the global frame permits multiple instances of a module with a single copy of the code;
>
> EV (together with the encoding of the code base through the *env* field) permits a compact *code* field in a procedure descriptor.

Furthermore, each level provides a (sometimes marginally) useful freedom in relocating the object it points to:

> LV permits external procedure references to be bound without any change to the code, and without any auxiliary data structure which locates all the call instructions.
>
> GFT permits global frames to be moved (unfortunately this is not useful in the current Mesa language, which allows other references to the components of global frames).
>
> The global frame permits the code segment to be moved. This is very important in versions of Mesa without paging, since it allows a simple and efficient implementation of code swapping and relocation.
>
> EV permits a procedure to be moved in the code segment. This allows a procedure to be dynamically replaced by another of a different size, without any loss of efficient packing.

The disadvantage of all this indirection is that it takes a considerable amount of unpacking, and a number of memory references, to get from the EXTERNALCALL instruction to an address which can be used for fetching the next instruction.

A call to a procedure in the same module is handled by a LOCALCALL *n* instruction, where *n* is the EV index rather than the LV index. This kind of call keeps the same environment and code base, and has only one level of indirection; it is just as compact as an EXTERNALCALL instruction.

Procedure returns are encoded by a one-byte RETURN instruction which does *returnContext*: = NIL; XFER[*LF.returnLink*] after freeing the current frame. When a context gets control back after it has done an EXTERNALCALL, it ignores *returnContext*. There are several other instructions which combine an XFER with other operations, to support traps, coroutine linkages, and multiple processes efficiently.

5.2 Passing arguments and results

This is done exactly as in § 4; arguments or results are pushed onto the stack before the XFER with ordinary LOAD instructions. When a procedure is entered after a call, it stores the arguments into local variables with ordinary STORE instructions. After a return, the results are on the stack ready for further computation.

There are two drawbacks of this scheme. One is that although the work to load arguments onto the stack seems to be encoded as compactly as possible and can be executed as fast as possible, the work done to store them is wasteful; it would be much better to have a way of using the arguments in place. The other is that code of the form $f[g[], h[]]$ requires the results of g to be saved before h is called, and then retrieved.

5.3 Allocating the frame

This too is done much as in § 4. However, a specialized heap is used to make the allocation nearly as fast as stack allocation; he same allocator is used for long argument records. A procedure specifies its frame size in its first byte by a *frame size index* into a array of free lists called the *allocation vector* AV. Frame sizes increase from a minimum of about 16 bytes in steps of about 20%; less than 20 steps are needed to cover any size up to several thousand bytes.

An element of AV is the head of a list of free frames of that size; see figure 2. Each frame has an extra word which holds its frame size index, so that the size need not be specified when it is freed. Only three memory references are required to allocate a frame (fetch list head from AV, fetch *next* pointer from first node, store it into list head), and four to free it. If the free list is empty there is a trap to a software allocator which creates more frames of the desired size. Note that the choice of frame sizes is private to the compiler (which asssigns the frame size index values) and the software allocator (which replenishes the free lists), and is not known to the fast heap allocator..

This scheme wastes only 10% of the space in fragmentation, plus space allocated to frames of sizes not currently in demand. These two effects can be balanced: fewer frame sizes means more fragmentation, but more chance to use an existing free frame. The scheme is quite cheap to implement. It requires no special cases to handle the frames of multiple processes or coroutines, retained frames, or argument records, since it does not depend on a last-in first-out discipline.

In doing an XFER to a procedure descriptor, after the frame is allocated the *returnContext* is saved in its *returnLink* component, and the global frame address is saved in its *globalFrame* component. When transferring out of a context, its program counter relative to the code base is saved in the *PC* component. When transferring into a context, the code base is recovered from the global frame and added to the *PC* component to get the next instruction address.

6. Fast instruction fetching

The Mesa implementation just described was designed to minimize the space required to encode control transfers, without compromising the generality of the model, and keeping almost all the state in main storage. We now turn to techniques for executing transfers quickly, in the common case of procedure call and return. These techniques are based on the principle of recognizing common special cases and optimizing them. The fully general mechanism of § 4-5 is preserved, however, so that there is always an orderly *fallback* position if the preconditions for an optimization become too hard to establish.

The goal is to make a call or return as fast as an unconditional jump. Since a machine of even moderate performance is likely to have some kind of instruction fetch unit (IFU), this goal is unlikely to be achieved unless the IFU can follow calls and returns as well as it follows jumps. This observation suggests that a good model for a highly optimized call is the kind of hand-tuned linkage that an assembly language programmer can devise between two routines which are both under his control. In such a linkage, the actual transfer is done by a branch-and-link instruction, which takes the procedure address as a literal operand and leaves the return address in a register. The parameters are passed in registers, and the procedure uses registers for its local storage if possible; ideally things are arranged so that these are disjoint from the registers used by the caller, so that nothing needs to be saved or restored. Of course, this kind of linkage is impractical in serious programming, because it is too hard to maintain all the assumptions on both sides as the program changes. We shall see, however, that it is quite practical for the compiler and the interpreter to provide the same effect for nearly all calls, even without any inter-procedural static analysis.

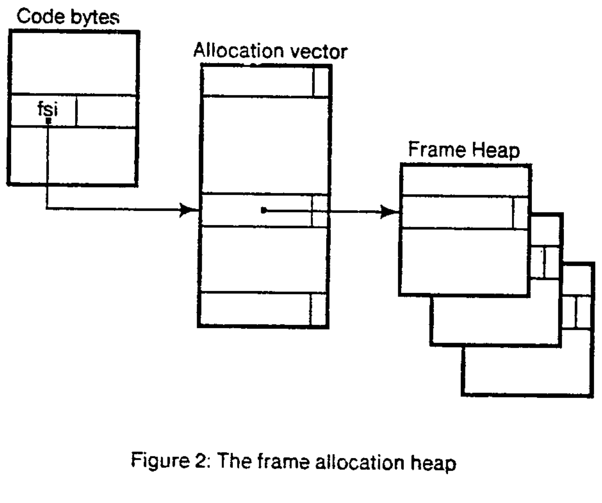

Figure 2: The frame allocation heap

There are two quite different strategies for optimizing transfers (or any other aspect of implementing a programming language). One maintains the same *encoding*, but during execution *dynamically* recognizes special cases and handles them more efficiently. A cache is a standard example of this technique: it maintains the abstraction of a uniform store, but the implementation uses two or more levels of storage, and a perhaps complex scheme for transferring data among the levels and executing read and write operations so that the properties of the simple abstraction are preserved. Dynamic optimizations usually exploit some locality property of the running program.

The other strategy maintains the same *source language*, but changes the encoding to reflect special cases which are recognized *statically* (usually by the compiler). This static method cannot deal with rapid changes in the program, since it requires recompilation or relinking to handle them. It usually exploits some fixed relationships among parts of the program, and it does not depend on any locality properties to do this. A common name for this process is *early binding*.

In dealing with procedure calls, it is natural to use static optimization to improve the process of obtaining the new instruction address, giving up both code compactness and the freedom to rebind dynamically in various places in favor of a direct link from the caller to the called procedure at address *p*. Thus the idea is to have a DIRECTCALL *p* instruction; at *p* is stored the global frame address *GF* and the frame size *fsi*, immediately followed by the first instruction. Thus the IFU can treat a DIRECTCALL just like an unconditional jump, except that it converts *GF* and *fsi* into instructions of the form SETGLOBALFRAME *GF* and ALLOCATEFRAME *fsi*; there is no reason to waste space in storing the opcodes for these instructions.

The disadvantages of this scheme are the advantages of the current Mesa scheme:

D1) The call instruction is larger: four bytes instead of one, for a 24-bit program address space; by having four DIRECTCALL instructions, we can extend this to 26 bits, and so forth. Of course, two bytes of LV entry are saved, so the space is only 30% more if the procedure is called only once from the module.

D2) Multiple instances of *p*'s module are not possible, since the global environment information is bound into the code. But multiple instances, though useful, are not the norm, and it is reasonable to optimize for the single instance case.

D3) Linking to *p* requires fixing up addresses throughout the code, as is traditional in conventional linkers. This is especially inconvenient if the linkage has to be changed, but this happens relatively seldom.

D1 is a simple time-space tradeoff. Further early binding may be able to put the called procedure close to the caller, so that a shorter, PC-relative address can be used. With 16 such SHORTDIRECTCALL opcodes, a three byte instruction can address one megabyte around the instruction. If this succeeds, the space is the same as in the current scheme for a single call of *p* from a module, and 50% more (6 bytes instead of 4) for two calls.

D2 is dealt with by falling back to the scheme of § 5 when multiple instances are required. Depending on the details of the encoding, this may require relinking or even recompiling callers. Once the fallback has been accomplished, however, all the flexibility of the model is again available. Thus the DIRECTCALL linkage should be regarded as an early binding, which is appropriate when the nature of the procedure is well known; in a large programming system, most procedures are "in the system" rather than the object of current development, and hence are well known for this purpose. If there is uncertainty about the procedure, it is best to stay with the more costly but flexible scheme. Note that with either linkage the program behaves identically (except for space and speed), so changing between them only changes the balance among space, speed of execution, and speed of changing the linkage. Thus D3 too is a performance tradeoff.

Returns cannot be handled statically, since in general static analysis does not reveal enough information about the caller(s). However, the IFU can keep a small stack of return information: frame pointer, global frame pointer *GF* and *PC*. As long as calls and returns follow a LIFO discipline this allows returns to be handled as fast as calls. When something unusual happens (e.g., any XFER other than a simple call or return, or running out of space in the return stack), fall back to the general scheme by flushing the return stack: the frame pointer *LF* goes into the *returnLink* component of the next higher frame, and the *PC* goes into the *PC* component of *LF*. The global frame pointer can be discarded, since it can be recovered from the local frame. Then the rule for a return is simple: if the return stack is empty, proceed as in § 5. Otherwise start fetching instructions from the *PC* value on the return stack, and restore the frame and global frame registers from those values.

7. Fast local variables and parameters

This idea of maintaining a limited stack in registers can also be applied to local frames and argument passing.

7.1 Frames in registers

Following [4], we suppose that the processor has a small number of register banks (say 4-8) of some modest fixed size (say 16 words). Each of these banks can hold the first

16 words of some local frame. When a new frame is needed, it is allocated in the usual way, but a register bank (assuming one is free) is also allocated to shadow its first few words. References to the shadowed words are made directly to the register bank; to simplify this, the encoding is designed so that such references are made with distinct instructions (the consequences of addressing these words with ordinary memory-reference instructions are discussed later). When the frame is freed, the shadowing register bank is also marked free, and can then be used to shadow a newly created frame; its contents are unimportant, and never need to be saved in storage. The return stack discussed in § 6 keeps track of the bank associated with each local frame, and the IFU passes this information along to the processor as the operand of a XFER.

If an overflow occurs (i.e., a new frame is created, and there are no register banks available), then the contents of the oldest bank is written out into the frame. If an XFER is done to a frame which doesn't have a shadowing bank, a free bank is assigned and loaded from the frame. As with a cache (of which this is a highly specialized form), the effectiveness of the scheme depends on the rarity of underflow and overflow. Fragmentary Mesa statistics indicate that with 4 banks it happens on less that 5% of XFERs; and [4] reports that with 4-8 banks the rate is less than 1%. Intuitively, this means that long runs of calls nearly uninterrupted by returns, or vice versa, are quite rare. Measurements are needed on a larger set of programs to confirm the effectiveness of this scheme, and of the elaborations described below.

As usual, when life gets complicated because of a process switch, trap or whatever, we fall back to the general scheme: all the banks are flushed into storage. It may be worthwhile to keep track of which registers have been written, to avoid the cost of dumping registers which have never been written.

To gain maximum advantage from this scheme, the register banks should be large enough that nearly all references to local variables will fit. Mesa statistics suggest that 95% of all frames allocated are smaller that 80 bytes, and this sets a conservative upper bound on the size of a register bank. With 8 banks of 80 bytes, there would be about 5000 bits of registers, which does not seem unreasonable.

The next step is to speed up frame allocation. Since nearly all local frames are fairly small, a reasonable strategy is to make the smallest frame size the 80 bytes just cited; hopefully this would handle 95% of all frame allocations. Now the processor can keep a stack of free frames of this size, and allocation will be extremely fast; furthermore, it can be done in parallel with the rest of na XFER operation. When the stack underflows, or if a larger local frame is needed, we fall back to the general scheme as usual. If the general scheme is five times more costly and it is used 5% of the time, the effective speed of frame allocation is .8 times the fast speed.

One drawback of this approach is that extremely deep recursion, or the presence of a very large number of processes, might result in thousands of 80 byte frames in which only 20 bytes are used. If this is a real problem, an alternative strategy is to defer allocating the frame until a register bank must be flushed out. This means that 95% of the time there will be no allocation at all. Unfortunately, it also means that a local variable may have no assigned memory address; the consequences of this are discussed in § 7.4 below.

7.2 Argument passing

In addition to local variables, the Mesa encoding also makes use of an evaluation stack for expression evaluation and argument passing. It is natural to use the register banks of § 7.1 to hold this stack also. As Patterson points out [4], this has the nice property that after the arguments have been loaded on the stack, the bank holding the stack can be renamed to be the shadower for the local frame of the called procedure. As a consequence, the arguments will automatically appear as the first few local variables, without any actual data movement. Thus on a call the pattern is:

(top of return stack).*Lbank*: = current *Lbank*
current *Lbank*: = *stack*
stack: = newly assigned bank

On a return, the stack should remain as it is, and the current frame should be freed:

free current *Lbank*
current *Lbank*: = (top of return stack).*Lbank*

Thus the banks are *not* used in last-in first-out order. Figure 3 illustrates.

This scheme provides essentially free passing of arguments and results; the only cost is the instructions to load them on the stack, and this seems unavoidable since the desired values must be specified somehow. Of course addresses can be loaded instead of values, but this isn't much cheaper, and it makes every reference to the value of an argument more expensive. Schemes which assemble either addresses or values in memory will certainly be more expensive.

7.3 Why not just a cache?

Why is this scheme better than simply keeping the frame in main storage, and taking advantage of a cache to make accesses faster? There are several reasons.

- A register bank is faster than a cache, both because it is smaller, and because the addressing mechanism is much simpler. Designers of high-performance processors have typically found that it is possible to

Bank 4	--	--	--	--	S	S	L = FD	--
Bank 3	--	S	S	L = FB	FB	L = FB	FB	L = FB
Bank 2	S	L = FA	--	S	L = FC	--	S	S
Bank 1	L = FX	FX	L	FX	FX	FX	FX	FX
Return stack	--	1, PCX --	--	1, PCX --	3, PCB 1, PCX --	1, PCX --	3, PCB 1, PCX --	1, PCX --
Lbank	1	2	1	3	2	3	4	3
Sbank	2	3	3	2	4	4	2	2
Begin in X	call A	return	call B	call C	return	call D	return	▪ ▪ ▪ ▪

Figure 3: Assignment of register banks for stacks and frames

read one register and write another in a single cycle, while two cycles are needed for a cache access. It is not too hard to build a cache which can accept a reference every cycle, but the latency is still two cycles. Also, since there are not too many registers it is feasible to duplicate or triplicate them, so that several registers can be read out simultaneously.

- Because they are faster, registers have more bandwidth, especially if they are duplicated. But more important, storing frequently accessed locals in registers frees up cache bandwidth for more random references. Half or more of all data memory references may be to local variables [4]. Removing this burden from the cache effectively doubles its bandwidth.
- The locality of references to local variables is so much better than the locality of data references in general that another level of memory hierarchy which exploits this locality is highly worthwhile. Since the compiler can control quite well how these references are made, the hardware supporting this level can be much simpler than a general-purpose cache, which must fully support the simple read-write properties of storage in general.
- As a corollary of the last point, the simple ways in which local variables are addressed (nearly always by a constant displacement in an instruction) makes the addressing logic for a register bank both simple and fast. The scheme we have describe addresses the registers from one of two base registers concatenated with a four or five bit displacement. No comparators, associative lookup or other complication is needed.
- The free argument passing of § 7.2 won't work if local variables are in a cache (unless the stack is kept there too, which is clearly not a good idea, because it demands *much* more bandwidth).

7.4 Pointers to locals

It may be necessary to supply storage addresses which can be dereferenced to yield local variables. This can happen in a language like BCPL or C which explicitly allows the programmer to construct pointers to local variables. It can also happen in Pascal when a local variable is passed as a **var** parameter, or when a nested procedure does up-level addressing. There are two tiresome consequences:

C1) The variable must *have* an address; this rules out the trick of deferring allocation of a frame in storage until it is needed for dumping the register bank.

C2) When the address is used, the data must come from (or go to) the register, rather than to storage. Alternatively, the register bank must be flushed when any register in it is addressed. This is an instance of the multiple copy problem.

The simplest solution is avoidance: outlaw pointers to local variables or the local frame. The (doubtless incomplete) list of ways in which such pointers can arise suggests, however, that this may be unacceptable. We therefore consider how to deal with these problems.

C1 can be handled in two ways. One possibility is to give up the deferred allocation trick entirely, since it pays off only when there is deep recursion or a very large number of processes, each with several frames. Alternatively, if there is a special operation for generating a pointer to a local variable, this operation can do the allocation. In the normal case no such pointers will exist, and no allocation will be done.

C2 can be avoided in most languages by flagging local frames to which pointers can exist; this can be done statically by the compiler, or dynamically as suggested in the previous paragraph. A flagged frame is flushed to storage whenever control leaves its context; of course it must be reloaded whenever control returns. Now the frame can be correctly referenced by ordinary storage instructions, except when control is in its context. This is good enough for Pascal; it fails if the context, or some other process executing concurrently, can get hold of pointers to its own locals as source-language values. Either of these cases is highly undesirable for other reasons, however, and can reasonably be outlawed.

If C2 is not avoided, it must be detected. Patterson [4] suggests that by confining frames to a fixed *frame region* of the address space, we can be sure for most storage references that C2 has not arisen; another possibility is to mark pages containing frames. An address in the frame region, however, must be compared with the address assigned to each of the register banks. If there is a match, then the register bank can be flushed, after which the storage reference can proceed normally. Alternatively, the reference can be diverted to read or write the proper register. The mechanism needed for this is similar to that required on machines like the PDP-10 whose registers are also part of the address space. Depending on the implementation of the storage system, diversion may be easy or quite difficult. If it is easy, it is certainly the best way to deal with C2; even if the register reference is handled quite clumsily, so that it takes much longer than an ordinary storage reference, such references are not common, and hence the cost will be small.

8. Conclusion

We have seen that a very general model for control transfers can be implemented with a wide variety of tradeoffs among three factors:

Simplicity of the implementation (both compiler and interpreter); § 4 maximizes simplicity.

Space taken up by the program (both instructions and procedure descriptors); § 5 minimizes space.

Speed of execution; § 6-7 maximizes speed.

Clearly many intermediate positions are possible. If a moderate amount of implementation complexity can be tolerated, an encoding which allows both the generality of § 5 and the early binding of § 6 is attractive: the programming environment can automatically convert between the two representations when appropriate.

It is worth remembering that the value of an optimization depends on the statistics of the programs being executed. Our empirical data is very preliminary, and it is always possible that changes in the important applications, or in programming style, will cause the conclusions to change. Past experience suggests, however, that major changes in the source programs seldom affect the value of previously valid optimizations by more than a few percent.

The scheme of § 5 has been implemented in the Mesa programming system for several years. We intend to try the ideas of § 6-7 in the near future.

References

1. Colley, S. *et. al.* The object-based architecture of the Intel 432. *Proc. COMPCON,* Feb. 1981.
2. Johnsson, R. and Wick, J.D. An overview of the Mesa processor architecture, ACM *Symp. Architectural Support for Prog. Lang. and Operating Sys.*, Palo Alto, Mar. 1982.
3. Lampson, B.W. *et. al.* On the transfer of control between contexts. *Lecture Notes in Computer Science* 19, Springer, 1974.
4. Patterson. D. A. and Sequin, C. H. RISC I: A reduced instruction set VLSI computer. *8th Symp. Computer Architecture,* Minneapolis, May 1981.
5. Sites, R.L. How to use 1000 registers. *Caltech Conference on* VLSI, Jan. 1979.
6. Strecker, W.D. VAX – 11/780: A virtual address extension to the DEC PDP – 11 family. *Proc. NCC,* Jun. 1978, 967-980.
7. Tanenbaum, A. Implications of structured programming for machine architecture. *Comm.* ACM 21, 3, Mar. 1978, 237-246.
8. Thacker, C.P. *et. al.* Alto: A personal computer. In *Computer Structures: Readings and Examples.* 2nd ed., Siewiorek, Bell and Newell, eds., McGraw-Hill, New York, 1981.
9. Lampson. B.W. and Pier, K.A. A processor for a high performance personal computer. *Proc. 7th Int. Symp. Computer Architecture,* La Baule, France, May 1980. Also in Technical Report CSL-81-1, Xerox Research Center, Palo Alto, CA, Jan. 1981.