

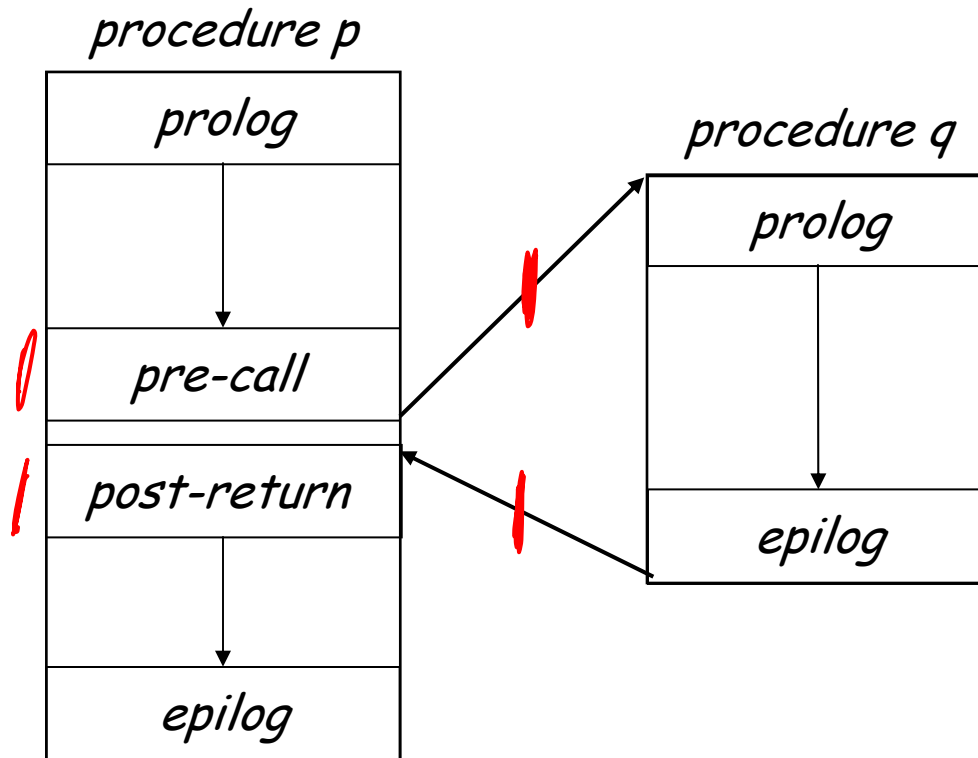
Code Shape IV

Procedure Calls & Dispatch

Copyright 2003, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Procedure Linkages

Standard procedure linkage



Procedure has

- standard **prolog**
- standard **epilog**

Each call involves a

- **pre-call** sequence
- **post-return** sequence

These are completely predictable from the call site \Rightarrow depend on the number & type of the actual parameters

Implementing Procedure Calls

If p calls q ...

- In the code for p , compiler emits pre-call sequence
 - Evaluates each parameter & stores it appropriately ✓
 - Loads the return address from a label (usually not needed) ✓
 - (with access links) sets up q 's access link
 - Branches to the entry of q ✓
- In the code for p , compiler emits post-return sequence
 - Copy return value into appropriate location ✓
 - Free q 's AR, if needed ✓
 - Resume p 's execution ✓

*static
chain*

Invariant parts of pre-call sequence might be moved into the prolog

Implementing Procedure Calls

If p calls q ...


- In the prolog, q must
 - Set up its execution environment
 - (with display) update the display entry for its lexical level
 - Allocate space for its (AR &) local variables & initialize them
 - If q calls other procedures, save the return address
 - Establish addressability for static data area(s)
- In the epilog, q must
 - Store return value (unless "return" statement already did so)
 - (with display) restore the display entry for its lexical level
 - Restore the return address (*if saved*)
 - Begin restoring p 's environment
 - Load return address and branch to it

alloc space
for local vars

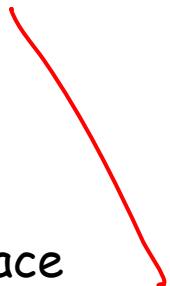


Implementing Procedure Calls

If p calls q , one of them must

- Preserve register values *(caller-saves versus callee saves)*
 - Caller-saves registers stored/restored by p in p 's AR
 - Callee-saves registers stored/restored by q in q 's AR
- Allocate the AR
 - Heap allocation \Rightarrow callee allocates its own AR
 - Stack allocation \Rightarrow caller & callee cooperate to allocate AR 

Space tradeoff

- Pre-call & post-return occur on every call
- Prolog & epilog occur once per procedure
- More calls than procedures
 - Moving operations into prolog/epilog saves space 

Implementing Procedure Calls

If p calls q , one of them must

- Preserve register values (caller-saves versus callee saves)

If space is an issue

- Moving code to prolog & epilog saves space
- As register sets grow, save/restore code does, too
 - Each saved register costs 2 operations
 - Can use a library routine to save/restore
 - ◆ Pass it a mask to determine actions & pointer to space
 - ◆ Hardware support for save/restore or storeM/loadM

{ 32, 64,
128, 256

Can decouple who saves from what is saved

Implementing Procedure Calls

If p calls q , one of them must

- Preserve register values (caller-saves versus callee saves)

If space is an issue

- All saves in prolog, all restores in epilog
 - Caller provides a bit mask for caller-saves registers
 - Callee provides a bit mask for callee-saves registers
 - Store all of them in same AR (*either caller or callee*)
 - Efficient use of time and code space
 - May waste some register save space in the AR
- Caller-save & callee-save assign responsibility not work

Implementing Procedure Calls

Evaluating parameters

- Call by reference \Rightarrow evaluate parameter to an lvalue
- Call by value \Rightarrow evaluate parameter to an rvalue & store it

Aggregates, arrays, & strings are usually c-b-r

- Language definition issues
- Alternative is copying them at each procedure call (\$\$)
 - \rightarrow Small structures can be passed in registers (*in & out*)
 - \rightarrow Can pass large c-b-v objects c-b-r and copy on modification

AIX does this for C

Implementing Procedure Calls

Evaluating parameters

- Call by reference \Rightarrow evaluate parameter to an lvalue
- Call by value \Rightarrow evaluate parameter to an rvalue & store it

Procedure-valued parameters

- Must pass starting address of procedure
- With access links, need the lexical level as well
 - \rightarrow Procedure value is a tuple $\langle \textit{level}, \textit{address} \rangle$
 - ♦ May also need shared data areas (*file-level scopes*)
 - ♦ In-file & out-of-file calls have (*slightly*) different costs
 - \rightarrow This lets the caller set up the appropriate access link

Implementing Procedure Calls

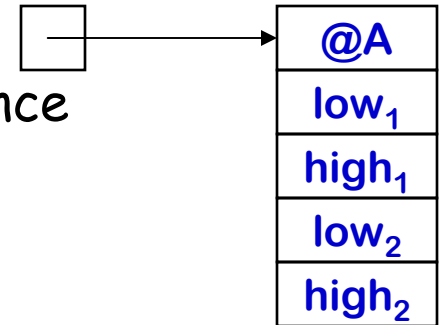
What about arrays as actual parameters?

Whole arrays, as call-by-reference parameters

- Callee needs dimension information \Rightarrow build a *dope vector*
- Store the values in the calling sequence
- Pass the address of the dope vector in the parameter slot
- Generate complete address polynomial at each reference

Some improvement is possible

- Save len_i and low_i rather than low_i and $high_i$
- Pre-compute the fixed terms in prologue sequence



What about call-by-value?

- Most c-b-v languages pass arrays by reference
- This is a language design issue

Implementing Procedure Calls

What about $A[12]$ as an actual parameter?

If corresponding parameter is a scalar, it's easy

- Pass the address or value, as needed
- Must know about both formal & actual parameter
- Language definition must force this interpretation

What if corresponding parameter is an array?

- Must know about both formal & actual parameter
- Meaning must be well-defined and understood
- Cross-procedural checking of conformability

⇒ Again, we're treading on language design issues

Fortran 77
lets amazing
things happen
in this case...

An Aside That Doesn't Fit Well Anywhere ...

What about code for access to variable-sized arrays?

Local arrays dimensioned by actual parameters

- Same set of problems as parameter arrays
- Requires dope vectors (or equivalent)
 - Place dope vector at fixed offset in activation record

⇒ Different access costs for textually similar references

This presents lots of opportunities for a good optimizer

- Common subexpressions in the address polynomial
- Contents of dope vector are fixed during each activation
- Should be able to recover much of the lost ground

⇒ Handle them like parameter arrays

Implementing Procedure Calls

What about a string-valued argument?

- Call by reference \Rightarrow pass a pointer to the start of the string
 - \rightarrow Works with either length/contents or null-terminated string
- Call by value \Rightarrow copy the string & pass it
 - \rightarrow Can store it in caller's AR or callee's AR
 - \rightarrow Callee's AR works well with stack-allocated ARs
 - \rightarrow Can pass by reference & have callee copy it if necessary ...

Pointer acts as a "descriptor" for the string, stored in the appropriate location (register or slot in the AR)

Implementing Procedure Calls

What about a structure-valued parameter?

- Again, pass a descriptor
- Call by reference \Rightarrow descriptor (pointer) refers to original
- Call by value \Rightarrow create copy & pass its descriptor
 - \rightarrow Can allocate it in either caller's AR or callee's AR
 - \rightarrow Callee's AR works well with stack-allocated ARs
 - \rightarrow Can pass by reference & have callee copy it if necessary ...

If it is actually an array of structures, then use a dope vector

If it is an element of an array of structures, then ...

What About Calls in an OOL (Dispatch)?

In an OOL, most calls are indirect calls

- Compiled code does not contain address of callee
 - Finds it by indirection through class' method table
 - Required to make subclass calls find right methods
 - Code compiled in class C cannot know of subclass methods that override methods in C and C 's superclasses
- In the general case, need dynamic dispatch
 - Map method name to a search key
 - Perform a run-time search through hierarchy
 - ◆ Start with object's class, search for 1st occurrence of key
 - ◆ This can be expensive
 - Use a method cache to speed search
 - ◆ Cache holds $\langle \textit{key, class, method pointer} \rangle$

How big?

Bigger \Rightarrow more hits &
longer search

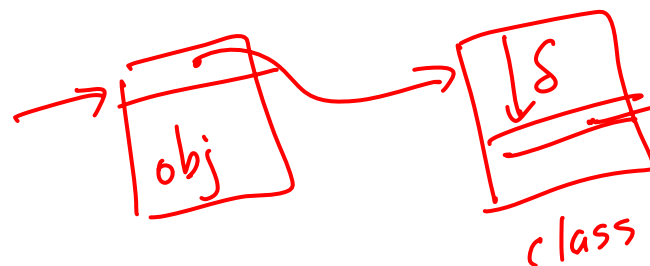
Smaller \Rightarrow fewer hits,
faster search

What About Calls in an OOL (Dispatch)?

Improvements are possible in special cases

- If class has no subclasses, can generate direct call ✓
 - Class structure must be static or class must be **FINAL**
- If class structure is static
 - Can generate complete method table for each class
 - Single indirection through class pointer (*1 or 2 operations*)
 - Keeps overhead at a low level
- If class structure changes infrequently
 - Build complete method tables at run time
 - Initialization & any time class structure changes
- If running program can create new classes, ...
 - Well, not all things can be done quickly

Java



*S known at compile time
method code for
method m*

What About Calls in an OOL (Dispatch)?

Unusual issues in OOL call

- Need to pass receiver's object record as (1st) parameter
 - Becomes self or this
- Typical OOL has lexical scoping in method
 - Limited to block-style scoping ⇒ no need for access links
 - Can overlay successive blocks in same method *(reuse)*
- Method needs access to its class
 - Object record has static pointer to superclass, and so on ...
 - Class pointers don't need updating like access-links
- Method is a full-fledged procedure
 - It still needs an AR ...
 - Can often stack allocate them *(HotSpot does ...)*

What About setjmp() and longjmp() ?

Unix system calls to implement abnormal returns

- Setjmp() stores a descriptor for use with longjmp()
- Invoking longjmp(*d*) causes execution to continue at the point after the setjmp() call that created *d*

How can we implement setjmp() & longjmp() ?

- Setjmp() must store ARP and return address in descriptor
 - What about values of registers and variables?
 - If they are to be preserved, must compute a closure
- Longjmp() must restore environment at setjmp()
 - Restore ARP & discard ARs creates since setjmp()
 - ◆ Cheap with stack-allocated ARs, might cost more with heap