CS415 Compilers

Procedure abstractions

part 4

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Office hours: Thursday, May 9, 11:30 - 12:30pm

• Review session on Monday, May 13, 12:30 - 2:00pm, CoRE 305

• Project 3 has been posted and is due Wednesday, May 8
  Need an extension?

• Homework #8 has been posted and is due today, Monday, May 6
  Need an extension?

• Final exam: May 14, noon - 3:00pm, SEC-118 (our current room)
  CONFLICTS?
  If you have a conflict, please send me the details of your conflict:

  class, email of instructor, time of scheduled exam

  NEED TO KNOW BY TODAY, MAY 6.
Activation Record Basics

- **parameters**
- **register save area**
- **return value**
- **return address**
- **addressability**
- **caller’s ARP**
- **local variables**

- Space for parameters to the current routine
- Saved register contents
- If function, space for return value
- Address to resume caller
- Help with non-local access
- To restore caller’s AR on a return (control link)
- Space for local values & variables (including spills)

One AR for each invocation of a procedure
Where do activation records live?

- If lifetime of AR matches lifetime of invocation, **AND**
- If code normally executes a “return”
  \[ \Rightarrow \text{Keep ARs on a stack} \]

- If a procedure can outlive its caller, **OR**
- If it can return an object that can reference its execution state
  \[ \Rightarrow \text{ARs must be kept in the heap} \]

- If a procedure makes no calls
  \[ \Rightarrow \text{AR can be allocated statically} \]

Efficiency prefers static, stack, then heap
Establishing Addressability

Must create base addresses
• Global & static variables
  → Construct a label by mangling names (i.e., &_fee)
• Local variables
  → Convert to static data coordinate and use ARP + offset
• Local variables of other procedures
  → Convert to static coordinates (level, offset)
  → Find appropriate ARP
  → Use that ARP + offset

Must find the right AR
Need links to nameable ARs
Using **access links** (static links)

- Each AR has a pointer to most recent AR of immediate lexical ancestor (mylevel - 1)

- Lexical ancestor need not be the caller

- Reference to <p,16> runs up access link chain to p

- Cost of access is proportional to lexical distance

Some setup cost on each call
Establishing Addressability

Using access links

Assume
- Current lexical level is 2
- Access link is at ARP - 4

Maintaining access link
- Calling level \( k + 1 \) (\( k \) is current level)
  → Use current ARP as link in new AR
- Calling level \( j < k \)
  → Find ARP for \( j - 1 \)
  → Use that ARP as link in new AR

<table>
<thead>
<tr>
<th>( SC )</th>
<th>Generated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;2,8&gt;)</td>
<td>loadAl ( r_0, 8 ) (\Rightarrow r_2 )</td>
</tr>
</tbody>
</table>
| \(<1,12>\) | loadAl \( r_0, -4 \) \(\Rightarrow r_1 \)  
             | loadAl \( r_1, 12 \) \(\Rightarrow r_2 \) |
| \(<0,16>\) | loadAl \( r_0, -4 \) \(\Rightarrow r_1 \)  
             | loadAl \( r_1, -4 \) \(\Rightarrow r_1 \)  
             | loadAl \( r_1, 16 \) \(\Rightarrow r_2 \) |

Access & maintenance cost varies with level
All accesses are relative to ARP \((r_0)\)
Using a display
- Global array of pointer to nameable ARs
- Needed ARP is an array access away

Reference to \(p,16\) looks up \(p\)'s ARP in display & adds 16
- Cost of access is constant \((\text{ARP + offset})\)

Some setup cost on each call

<table>
<thead>
<tr>
<th>Display</th>
<th>parameters</th>
<th>register</th>
<th>save area</th>
<th>return value</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>level 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>level 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>level 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>level 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ARP</th>
<th>parameters</th>
<th>register</th>
<th>save area</th>
<th>return value</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>local variables</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parameters</th>
<th>register</th>
<th>save area</th>
<th>return value</th>
<th>return address</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved ptr.</td>
<td>caller's ARP</td>
<td>local variables</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>caller's ARP</th>
<th>local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Establishing Addressability

Using a display

<table>
<thead>
<tr>
<th>SC</th>
<th>Generated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;2,8 &gt;</td>
<td>loadAl r₀, 8 ⇒ r₂</td>
</tr>
<tr>
<td>&lt;1,12 &gt;</td>
<td>loadl _disp ⇒ r₁</td>
</tr>
<tr>
<td></td>
<td>loadAl r₁, 4 ⇒ r₁</td>
</tr>
<tr>
<td></td>
<td>loadAl r₁, 12 ⇒ r₂</td>
</tr>
<tr>
<td>&lt;0,16 &gt;</td>
<td>loadl _disp ⇒ r₁</td>
</tr>
<tr>
<td></td>
<td>loadAl r₁, 16 ⇒ r₂</td>
</tr>
</tbody>
</table>

Assume
- Current lexical level is 2
- Display is at label _disp

Maintaining access link
- On entry to level j
  → Save level j entry into AR (Saved Ptr field)
  → Store ARP in level j slot
- On exit from level j
  → Restore level j entry

Desired AR is at _disp + 4 x level

Access & maintenance costs are fixed
Address of display may consume a register
Access links versus Display

- Each adds some overhead to each call
- Access links costs vary with level of reference
  - Overhead only incurred on references & calls
- Display costs are fixed for all references
  - References & calls must load display address
  - Typically, this requires a register

Your mileage will vary

- Depends on ratio of non-local accesses to calls
- Extra register can make a difference in overall speed

For either scheme to work, the compiler must insert code into each procedure call & return
How do procedure calls actually work?

- At compile time, callee may not be available for inspection
  - Different calls may be in different compilation units
  - Compiler may not know system code from user code
  - All calls must use the same protocol

Compiler must use a standard sequence of operations

- Enforces control & data abstractions
- Divides responsibility between caller & callee

Usually a system-wide agreement *(for interoperability)*
Procedure Linkages

Standard procedure linkage

```plaintext
procedure p

prolog

pre-call

post-return

epilog

procedure q

prolog

epilog
```

Procedure has
- standard prolog
- standard epilog

Each call involves a
- pre-call sequence
- post-return sequence

These are completely predictable from the call site ⇒ depend on the number & type of the actual parameters
Procedure Linkages

Pre-call Sequence
• Sets up callee’s basic AR
• Helps preserve its own environment

The Details
• Allocate space for the callee’s AR
  → except space for local variables
• Evaluates each parameter & stores value or address
• Saves return address, caller’s ARP (control link) into callee’s AR
• If access links are used
  → Find appropriate lexical ancestor & copy into callee’s AR
• Save any caller-save registers
  → Save into space in caller’s AR
• Jump to address of callee’s prolog code
Procedure Linkages

**Post-return Sequence**
- Finish restoring caller’s environment
- Place any value back where it belongs

**The Details**
- Copy return value from callee’s AR, if necessary
- Free the callee’s AR
- Restore any caller-save registers
- Copy back call-by-value/result parameters
- Continue execution after the call
Procedure Linkages

**Prolog Code**
- Finish setting up callee’s environment
- Preserve parts of caller’s environment that will be disturbed

**The Details**
- Preserve any callee-save registers
- If display is being used
  - Save display entry for current lexical level
  - Store current ARP into display for current lexical level
- Allocate space for local data
  - Easiest scenario is to extend the AR
- Handle any local variable initializations

With heap allocated AR, may need to use a separate heap object for local variables
### Epilog Code

- Wind up the business of the callee
- Start restoring the caller’s environment

### The Details

- Store return value?
  - Some implementations do this on the return statement
  - Others have return assign it & epilog store it into caller’s AR
- Restore callee-save registers
- Free space for local data, if necessary *(on the heap)*
- Load return address from AR
- Restore caller’s ARP
- Jump to the return address

If ARs are stack allocated, this may not be necessary. (Caller can reset stacktop to its pre-call value.)
Things to do and next class

Work on the project!

Q&A