RICE

COMP 412
FALL 2010

Code Shape, Part II
Addressing Arrays, Aggregates, & Strings

Comp 412

Last Lecture

Code Generation for Expressions

- Simple treewalk produces reasonable code
  - Execute most demanding subtree first
  - Generate function calls inline
  - Can implement treewalk explicitly, with an AG or ad hoc SDT …
- Handle assignment as an operator
  - Insert conversions according to language-specific rules
  - If compile-time checking is impossible, check tags at runtime
  - Talked about reference counting as alternative to GC

Today

- Addressing arrays and aggregates
- Next Time: Booleans & Relationals

How does the compiler handle A[i,j] ?

First, must agree on a storage scheme

Row-major order (most languages)

- Lay out as a sequence of consecutive rows
- Rightmost subscript varies fastest
- A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]

Column-major order (Fortran)

- Lay out as a sequence of columns
- Leftmost subscript varies fastest
- A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]

Indirection vectors (Java)

- Vector of pointers to pointers to … to values
- Takes much more space, trades indirection for arithmetic
- Not amenable to analysis

Laying Out Arrays

The Concept

A

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4

These can have distinct & different cache behavior

Row-major order

A

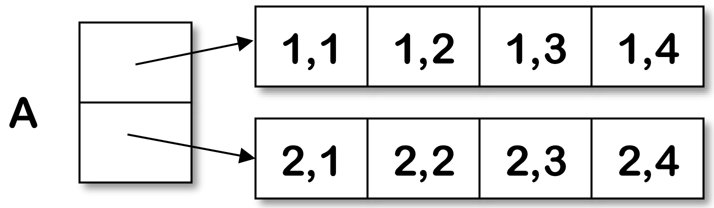
1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4

Column-major order

A

1,1	2,1	1,2	2,2	1,3	2,3	1,4	2,4

Indirection vectors



Computing an Array Address

A[i]

- @A + (i – low) x sizeof(A[1])
- In general: base(A) + (i – low) x sizeof(A[1])

Color Code:
Invariant
Varying

Depending on how A is declared, @A may be
- an offset from the ARP,
- an offset from some global label, or
- an arbitrary address.

The first two are compile time constants.

Computing an Array Address

A[i]

- @A + (i – low) x sizeof(A[1])
- In general: base(A) + (i – low) x sizeof(A[1])

int A[1:10] ⇒ low is 1
Make low 0 for faster access (saves a –)

Almost always a power of 2, known at compile-time ⇒ use a shift for speed

Computing an Array Address

A[i]

- @A + (i – low) x sizeof(A[1])
- In general: base(A) + (i – low) x sizeof(A[1])

What about $A[i_1,i_2]$?

This stuff looks expensive!
Lots of implicit +, -, x ops

Row-major order, two dimensions

$@A + ((i_1 - low_1) \times (high_2 - low_2 + 1) + i_2 - low_2) \times sizeof(A[1])$

Column-major order, two dimensions

$@A + ((i_2 - low_2) \times (high_1 - low_1 + 1) + i_1 - low_1) \times sizeof(A[1])$

Indirection vectors, two dimensions

$*(A[i_1])[i_2]$ — where $A[i_1]$ is, itself, a 1-d array reference

e.g., $@A + (i_1 - low) \times sizeof(A[1])$

Optimizing Address Calculation for A[i,j]

In row-major order

$@A + (i - low_1) \times (high_2 - low_2 + 1) \times w + (j - low_2) \times w$

where $w = sizeof(A[1,1])$

Which can be factored into

$@A + i \times (high_2 - low_2 + 1) \times w + j \times w$

$\quad - (low_1 \times (high_2 - low_2 + 1) \times w) - (low_2 \times w)$

If low_i, $high_i$, and w are known, the last term is a constant

Define $@A_0$ as

$@A - (low_1 \times (high_2 - low_2 + 1) \times w - low_2 \times w$

And len_2 as $(high_2 - low_2 + 1)$

If $@A$ is known, $@A_0$ is a known constant.

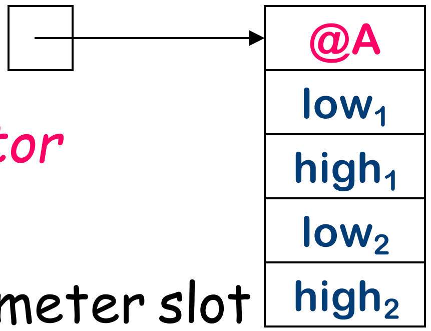
Then, the address expression becomes

$@A_0 + (i \times len_2 + j) \times w$

Compile-time constants

Array References

What about arrays as actual parameters?



Whole arrays, as call-by-reference parameters

- Need dimension information → 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

Array References

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 is 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

Array References

What about variable-sized arrays?

Local arrays dimensioned by actual parameters

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

→ Different access costs for textually similar references

This presents a lot of opportunity 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

Array Address Calculations

Array address calculations are a major source of overhead

- Scientific applications make extensive use of arrays and array-like structures
  - Computational linear algebra, both dense & sparse
- Non-scientific applications use arrays, too
  - Representations of other data structures
    - → *Hash tables, adjacency matrices, tables, structures, …*

Array calculations tend iterate over arrays

- Loops execute more often than code outside loops
- Array address calculations inside loops make a huge difference in efficiency of many compiled applications

Reducing array address overhead has been a major focus of optimization since the 1950s.

Example: Array Address Calculations in a Loop

A, B are declared as conformable floating-point arrays

```
DO J = 1, N
    A[I,J] = A[I,J] + B[I,J]
END DO
```

Naïve: Perform the address calculation twice

```
DO J = 1, N
    R1 = @A_0 + (J x len_1 + I ) x sizeof(A[1,1])
    R2 = @B_0 + (J x len_1 + I ) x sizeof(A[1,1])
    MEM(R1) = MEM(R1) + MEM(R2)
END DO
```

Code generated by a translator will almost certainly work this way. (treewalk code generator)

Imagine a 5 point stencil:

```
A[I,J] =  0.2 * (A[I-1,J] + A[I,J] + A[I+1,J]
                 + A[I,J-1] + A[I,J+1] )
```

Inefficiency is an artifact of local translation

Example: Array Address Calculations in a Loop

```
DO J = 1, N
    A[I,J] = A[I,J] + B[I,J]
END DO
```

More sophisticated: Move common calculations out of loop

```
R1 = I x sizeof(A[1,1])
c = len_1 x sizeof(A[1,1])     ! Compile-time constant
R2 = @A_0 + R1
R3 = @B_0 + R1
DO J = 1, N
    a = J x c
    R4 = R2 + a
    R5 = R3 + a
    MEM(R4) = MEM(R4) + MEM(R5)
END DO
```

Loop-invariant code motion

Example: Array Address Calculations in a Loop

```
DO J = 1, N
    A[I,J] = A[I,J] + B[I,J]
END DO
```

Very sophisticated: Convert multiply to add

```
R1 = I x sizeof(A[1,1])
c = len₁ x sizeof(A[1,1])     ! Compile-time constant
R2 = @A₀ + R1 ;  R3 = @B₀ + R1
DO J = 1, N
    R2 = R2 + c
    R3 = R3 + c
    MEM(R2) = MEM(R2) + MEM(R3)
END DO
```

J is now bookkeeping

A good compiler would rewrite the end-of-loop test to operate on R2 or R3

(Linear function test replacement)

Structures and Records

Structures and records have two complications

Each declared structure has a set of fields

- Size and offset
- Compute base + offset for field
- Use size to choose load width and register width

Structures and records can have dimensions

- Arrays of structures
- Fields that are arrays or arrays of structures
- Use array address calculation techniques, as needed

Structures and records require compile-time support in the form of a table that maps field names to <*offset,size*> tuples.

Representing and Manipulating Strings

Character strings differ from scalars, arrays, & structures

- Fundamental unit is a character
  - Typical sizes are one or two bytes
  - Target ISA may (or may not) support character-size operations
- Set of supported operations on strings is limited
  - Assignment, length, concatenation, translation (?)
- Efficient string operations are complex on most RISC ISAs
  - Ties into representation, linkage convention, & source language

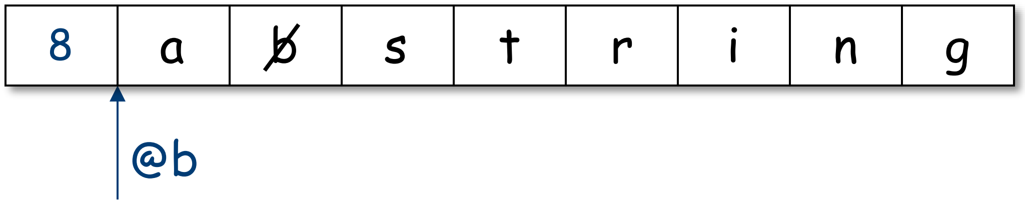
Subword data

§ 7.6 in EaC

Representing and Manipulating Strings

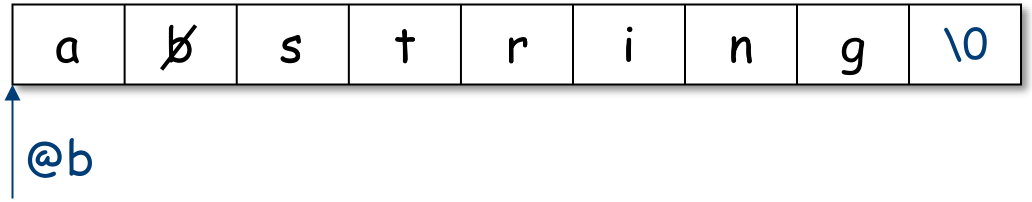
Two common representations

- Explicit length field



Length field may take more space than terminator

- Null termination



- Language design issue
  - Fixed-length versus varying-length strings *(1 or 2 length fields)*

String representation is a great case study in the way that one design decision (C, Unix) can have a long term impact on computing (security, buffer overflow)

Representing and Manipulating Strings

Each representation as advantages and disadvantages

Operation	Explicit Length	Null Termination
Assignment	Straightforward	Straightforward
Checked Assignment	Checking is easy	Must count length[1]
Length	O(1)	O(n)
Concatenation	Must copy data	Length + copy data

Unfortunately, null termination is almost considered normal

- Hangover from design of C
- Embedded in OS and API designs

[1] Checked assignment requires both a current length for the string and an allocated length for the buffer.

Manipulating Strings

Single character assignment

- With character operations
  - Compute address of rhs, load character
  - Compute address of lhs, store character
- With only word operations *(>1 char per word)*
  - Compute address of word containing rhs & load it
  - Move character to destination position within word
  - Compute address of word containing lhs & load it
  - Mask out current character & mask in new character
  - Store lhs word back into place

Manipulating Strings

Multiple character assignment

Two strategies

1. Wrap a loop around the single character code, or
2. Work up to a word-aligned case, repeat whole word moves, and handle any partial-word end case

With character operations

1. Easy to generate; inefficient use of resources
2. Harder to generate; better use of resources

Requires explicit code to check for buffer overflow (⇒ length)

With only word operations

1. Lots of complication to generate; inefficient at runtime, too
2. Fold complications into end case; reasonable efficiency

Source & destination aligned differently
⇒ much harder cases for word operations

Manipulating Strings

Concatenation

- String concatenation is a length computation followed by a pair of whole-string assignments
  - Touches every character
- Exposes representation issues
  - Is string a descriptor that points to text?
  - Is string a buffer that holds the text?
  - Consider a □ b || c
    - →Compute b || c and assign descriptor to a?
    - →Compute b || c into a temporary & copy it into a?
    - →Compute b || c directly into a?
- What about a call to fee(b || c) ?

Manipulating Strings

Length Computation

- Representation determines cost
  - Explicit length turns length(b) into a memory reference
  - Null termination turns length(b) into a loop of memory references and arithmetic operations
- Length computation arises in other contexts
  - Whole-string or substring assignment
  - Checked assignment (buffer overflow)
  - Concatenation
  - Evaluating call-by-value actual parameter or concatenation as an actual parameter

And we didn't consider a list of characters, …