

Slice-hoisting for Array-size Inference in MATLAB

LCPC 2003

Arun Chauhan and Ken Kennedy

Computer Science, Rice University

History Repeats

“It was our belief that if FORTRAN, during its first months, were to translate any reasonable ‘scientific’ source program into an object program only half as fast as its hand-coded counterpart, then acceptance of our system would be in serious danger... I believe that had we failed to produce efficient programs, the widespread use of languages like FORTRAN would have been seriously delayed.”

–John Backus

Example Compilation

```
function mcc_demo
    x = 1;
    y = x / 10;
    z = x * 20;
    r = y + z;
```

Example Compilation

```
static void Mmcc_demo (void) {  
    ...  
    mxArray * r = NULL;  
    mxArray * z = NULL;  
    mxArray * y = NULL;  
    mxArray * x = NULL;  
    mlfAssign(&x, _mxarray0_); /* x = 1; */  
    mlfAssign(&y, mclMrdivide(mclVv(x, "x"), _mxarray1_)); /* y = x / 10; */  
    mlfAssign(&z, mclMtimes(mclVv(x, "x"), _mxarray2_)); /* z = x * 20; */  
    mlfAssign(&r, mclPlus(mclVv(y, "y"), mclVv(z, "z"))); /* r = y + z; */  
    mxDestroyArray(x);  
    mxDestroyArray(y);  
    mxDestroyArray(z);  
    mxDestroyArray(r);  
    ...  
}
```

Example Compilation

```
static void Mmcc_demo (void) {  
    ...  
    double r;  
    double z;  
    double y;  
    double z;  
    mlfAssign(&x, _mxarray0_); /* x = 1; */  
    mlfAssign(&y, mclMrdivide(mclVv(x, "x"), _mxarray1_)); /* y = x / 10; */  
    mlfAssign(&z, mclMtimes(mclVv(x, "x"), _mxarray2_)); /* z = x * 20; */  
    mlfAssign(&r, mclPlus(mclVv(y, "y"), mclVv(z, "z"))); /* r = y + z; */  
    mxDestroyArray(x);  
    mxDestroyArray(y);  
    mxDestroyArray(z);  
    mxDestroyArray(r);  
    ...  
}
```

Example Compilation

```
static void Mmcc_demo (void) {  
    ...  
    double r;  
    double z;  
    double y;  
    double z;  
    scalarAssign(&x, 1); /* x = 1; */  
    scalarAssign(&y, scalarDivide(x, 10)); /* y = x / 10; */  
    scalarAssign(&z, scalarTimes(x, 20)); /* z = x * 20; */  
    scalarAssign(&r, scalarPlus(y, z)); /* r = y + z; */  
    mxDestroyArray(x);  
    mxDestroyArray(y);  
    mxDestroyArray(z);  
    mxDestroyArray(r);  
    ...  
}
```

Example Compilation

```
static void Mmcc_demo (void) {  
    ...  
    double r;  
    double z;  
    double y;  
    double z;  
    x = 1; /* x = 1; */  
    y = x / 10; /* y = x / 10; */  
    z = x * 20; /* z = x * 20; */  
    r = y + z; /* r = y + z; */  
    /* mxDestroyArray(x); */  
    /* mxDestroyArray(y); */  
    /* mxDestroyArray(z); */  
    /* mxDestroyArray(r); */  
    ...  
}
```

Inferring Types

Inferring Types

- $\text{type} \equiv \langle \tau, \delta, \sigma, \psi \rangle$
 - τ = intrinsic type, e.g., int, real, complex, etc.
 - δ = array dimensionality, 0 for scalars
 - σ = δ -tuple of positive integers
 - ψ = “structure” of an array

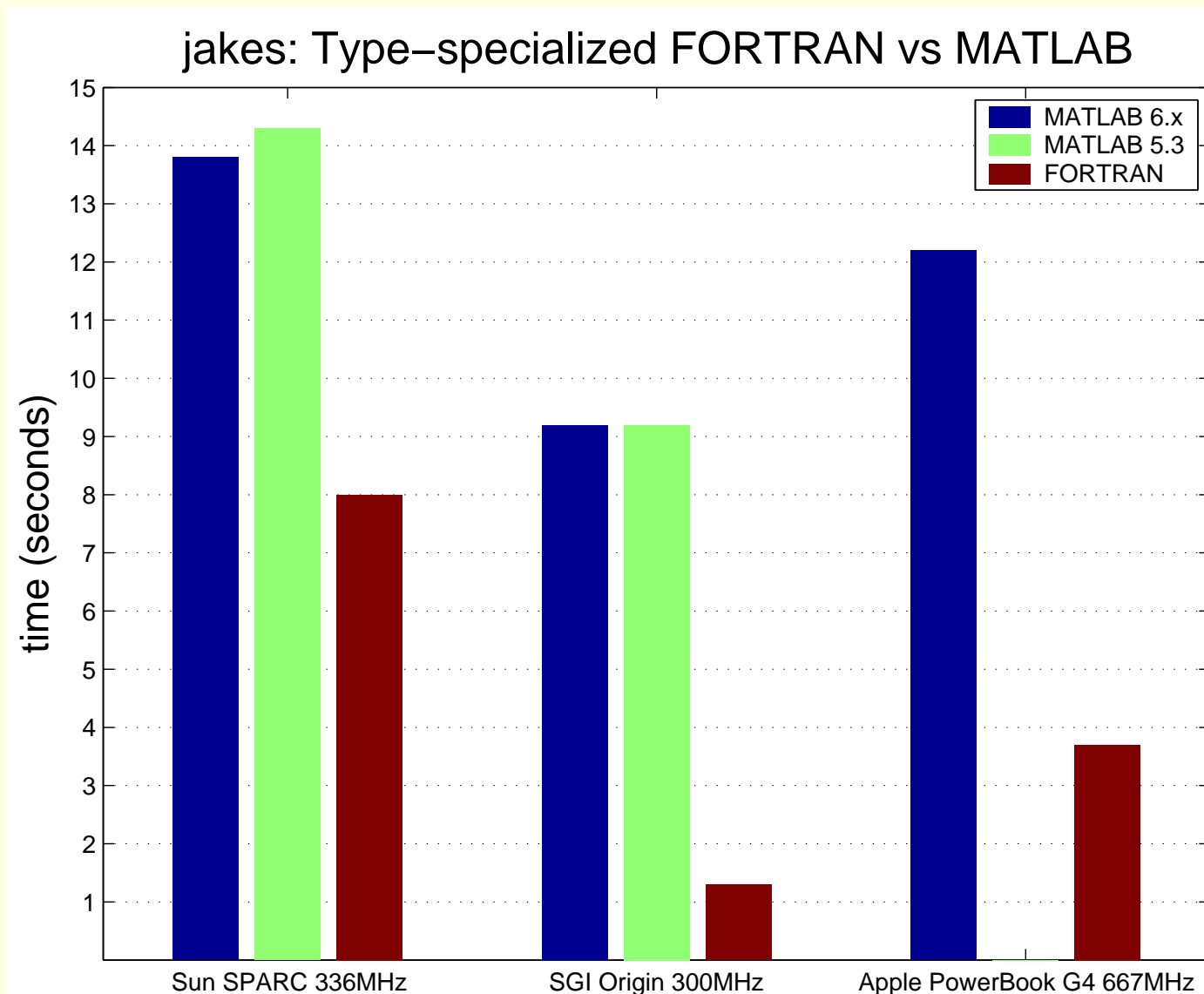
Inferring Types

- $\text{type} \equiv \langle \tau, \delta, \sigma, \psi \rangle$
 - τ = intrinsic type, e.g., int, real, complex, etc.
 - δ = array dimensionality, 0 for scalars
 - σ = δ -tuple of positive integers
 - ψ = “structure” of an array
- type inference in general
 - type = “smallest” set of values that preserves meaning

Inferring Types

- type $\equiv \langle \tau, \delta, \sigma, \psi \rangle$
 - τ = intrinsic type, e.g., int, real, complex, etc.
 - δ = array dimensionality, 0 for scalars
 - σ = δ -tuple of positive integers
 - ψ = “structure” of an array
- type inference in general
 - type = “smallest” set of values that preserves meaning
- type inference for telescoping languages
 - need **all** possible types that preserve meaning

Type-based Specialization



Static Type Inference

(joint work with Cheryl McCosh)

- dimensionality constraints

$$\mathbf{x} = \mathbf{1}$$

$$\mathbf{y} = \mathbf{x} / 10$$

$$\mathbf{z} = \mathbf{x} * 20$$

$$\mathbf{r} = \mathbf{y} + \mathbf{z}$$

Static Type Inference

(joint work with Cheryl McCosh)

- dimensionality constraints

$$\mathbf{x} = \mathbf{1}$$

LHS dims = RHS dims

$$\mathbf{y} = \mathbf{x} / \mathbf{10}$$

(x, y scalar) OR (x, y arrays of same size)

$$\mathbf{z} = \mathbf{x} * \mathbf{20}$$

(x, z scalar) OR (x, z arrays of same size)

$$\mathbf{r} = \mathbf{y} + \mathbf{z}$$

(r, y, z scalar) OR (r, y, z arrays of same size)

Static Type Inference

(joint work with Cheryl McCosh)

- write constraints
 - each operation or function call imposes certain “constraints”
 - incomparable types give rise to multiple valid configurations

Static Type Inference

(joint work with Cheryl McCosh)

- write constraints
 - each operation or function call imposes certain “constraints”
 - incomparable types give rise to multiple valid configurations
- the problem is hard to solve in general
 - efficient solution possible under certain conditions

Static Type Inference

(joint work with Cheryl McCosh)

- write constraints
 - each operation or function call imposes certain “constraints”
 - incomparable types give rise to multiple valid configurations
- the problem is hard to solve in general
 - efficient solution possible under certain conditions
- reducing to the clique problem
 - a constraint defines a level
 - clauses in a constraint are nodes at that level
 - an edge whenever two clauses are “compatible”
 - a clique defines a valid type configuration

Limitations for Array-size Inference

- control join-points may result in too many configs
 - control join-points ignored for array-sizes
- array sizes defined by indexed expressions
 - assignment to `a(i)` can resize `a`
- symbolic expressions may be unknown at compile time
- array sizes changing in a loop not handled

Size May Grow in a Loop

```
function [A, F] = pisar (xt, sin_num)
...
mcos = [];
for n = 1:sin_num
    vcos = [];
    for i = 1:sin_num
        vcos = [vcos cos(n*w_est(i))];
    end
    mcos = [mcos; vcos]
end
...
```

Slice-hoisting: Simple Example

```
A = zeros(1, N);
```

```
y = ...
```

```
A (y ) = ...
```

```
x = ...
```

```
A (x ) = ...
```

Slice-hoisting: Simple Example

```
A = zeros(1, N);  
 $\sigma^A = \langle N \rangle$   
y = ...  
A(y) = ...  
 $\sigma^A = \max(\sigma^A, \langle y \rangle)$   
x = ...  
A(x) = ...  
 $\sigma^A = \max(\sigma^A, \langle x \rangle)$ 
```

Slice-hoisting: Simple Example

```
A1 = zeros(1, N);  
σ1A1 = <N>  
y1 = ...  
A1(y1) = ...  
σ2A1 = max(σ1A1, <y1>)  
x1 = ...  
A1(x1) = ...  
σ3A1 = max(σ2A1, <x1>)
```

Slice-hoisting: Simple Example

```
A1 = zeros(1, N);  
⇒ σ1A1 = <N>  
⇒ y1 = ...  
  A1(y1) = ...  
⇒ σ2A1 = max(σ1A1, <y1>)  
⇒ x1 = ...  
  A1(x1) = ...  
⇒ σ3A1 = max(σ2A1, <x1>)
```

Slice-hoisting: Simple Example

```
⇒  $\sigma_1^{A_1} = \langle N \rangle$   
⇒  $y_1 = \dots$   
⇒  $\sigma_2^{A_1} = \max(\sigma_1^{A_1}, \langle y_1 \rangle)$   
⇒  $x_1 = \dots$   
⇒  $\sigma_3^{A_1} = \max(\sigma_2^{A_1}, \langle x_1 \rangle)$   
  allocate(A1,  $\sigma_3^{A_1}$ );  
  A1 = zeros(1, N);  
  A1(y1) = ...  
  A1(x1) = ...
```


Slice-hoisting: Steps

- insert σ statements
- do SSA conversion
- identify the slice involved in computing the σ values
- *hoist* the slice before the first use of the array

Slice-hoisting: Loop

```
A (x ) = ...  
  
for i = 1:N  
    ...  
  
    A = [A f(i)];  
  
end
```

Slice-hoisting: Loop

```
A (x ) = ...  
 $\sigma^A = \langle x \rangle$   
for i = 1:N  
    ...  
  
    A = [A f(i)];  
     $\sigma^A = \sigma^A + \langle 1 \rangle$   
end
```

- add σ statements

Slice-hoisting: Loop

```
A1(x1) = ...  
σ1A1 = <x1>  
for i1 = 1:N  
    ...  
    σ2A1 = φ(σ1A1, σ3A1)  
    A1 = [A1 f(i1)];  
    σ3A1 = σ2A1 + <1>  
end
```

- add σ statements
- do SSA

Slice-hoisting: Loop

```
A1(x1) = ...  
⇒ σ1A1 = <x1>  
⇒ for i1 = 1:N  
    ...  
    ⇒ σ2A1 = φ(σ1A1, σ3A1)  
    A1 = [A1 f(i1)];  
    ⇒ σ3A1 = σ2A1 + <1>  
⇒ end
```

- add σ statements
- do SSA
- identify slice

Slice-hoisting: Loop

```
⇒  $\sigma_1^{A_1} = \langle x_1 \rangle$   
⇒ for  $i_1 = 1:N$   
⇒  $\sigma_2^{A_1} = \phi(\sigma_1^{A_1}, \sigma_3^{A_1})$   
⇒  $\sigma_3^{A_1} = \sigma_2^{A_1} + \langle 1 \rangle$   
⇒ end  
  
allocate( $A_1, \sigma_3^{A_1}$ );  
 $A_1(x_1) = \dots$   
for  $i_1 = 1:N$   
    ...  
     $A_1 = [A_1 \ f(i_1)]$ ;  
end
```

- add σ statements
- do SSA
- identify slice
- hoist the slice

Slice-hoisting: Loop

```
 $\Rightarrow \sigma_3^{A_1} = \langle x_1 \rangle + \langle N \rangle$   
allocate(A1,  $\sigma_3^{A_1}$ );  
A1(x1) = ...  
for i1 = 1:N  
    ...  
    A1 = [A1 f(i1)];  
end
```

- add σ statements
- do SSA
- identify slice
- hoist the slice

Dependencies Can Raise Roadblocks

$A(1) = \dots$

\dots

$x = f(A)$

$A(x) = \dots$

\dots

Dependences Can Raise Roadblocks

$A(1) = \dots$

$\sigma^A = \langle 1 \rangle$

\dots

$x = f(A)$

$A(x) = \dots$

$\sigma^A = \max(\sigma^A, \langle x \rangle)$

\dots

Dependences Can Raise Roadblocks

$$A_1(1) = \dots$$

$$\sigma_1^{A_1} = \langle 1 \rangle$$

...

$$x_1 = f(A_1)$$

$$A_1(x_1) = \dots$$

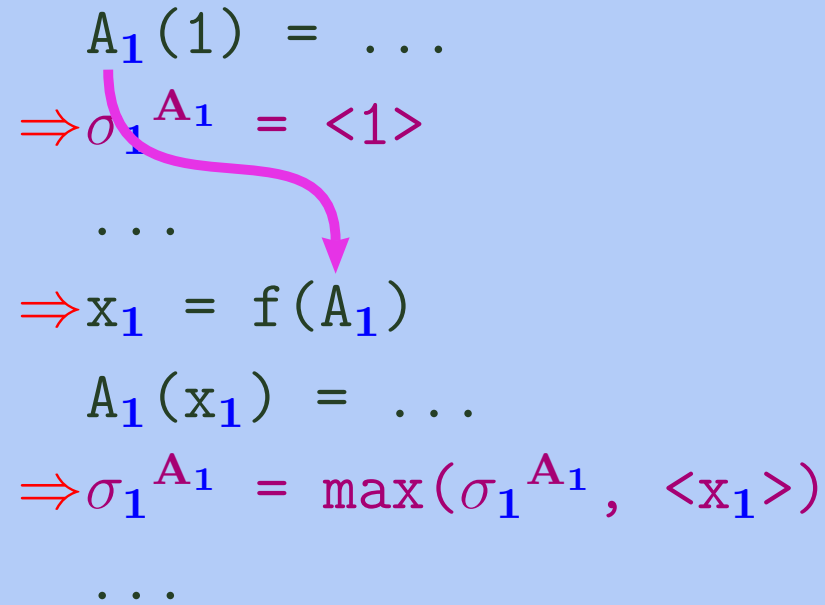
$$\sigma_1^{A_1} = \max(\sigma_1^{A_1}, \langle x_1 \rangle)$$

...

Dependences Can Raise Roadblocks

$$\begin{aligned} A_1(1) &= \dots \\ \Rightarrow \sigma_1^{A_1} &= \langle 1 \rangle \\ &\dots \\ \Rightarrow x_1 &= f(A_1) \\ A_1(x_1) &= \dots \\ \Rightarrow \sigma_1^{A_1} &= \max(\sigma_1^{A_1}, \langle x_1 \rangle) \\ &\dots \end{aligned}$$

Dependencies Can Raise Roadblocks



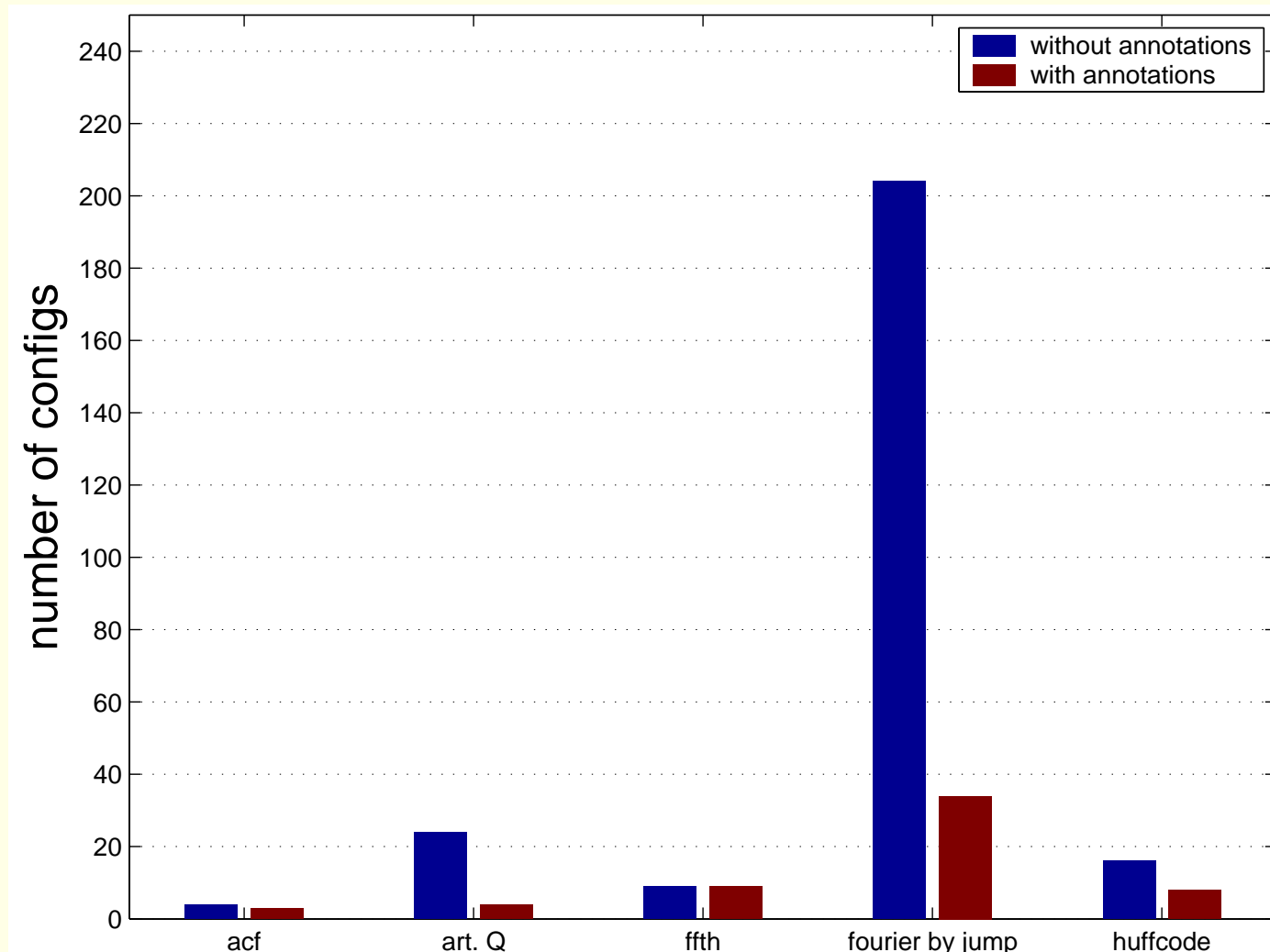
$A_1(1) = \dots$
 $\Rightarrow \sigma_1^{A_1} = \langle 1 \rangle$
 \dots
 $\Rightarrow x_1 = f(A_1)$
 $A_1(x_1) = \dots$
 $\Rightarrow \sigma_1^{A_1} = \max(\sigma_1^{A_1}, \langle x_1 \rangle)$
 \dots

Dependencies Can Raise Roadblocks

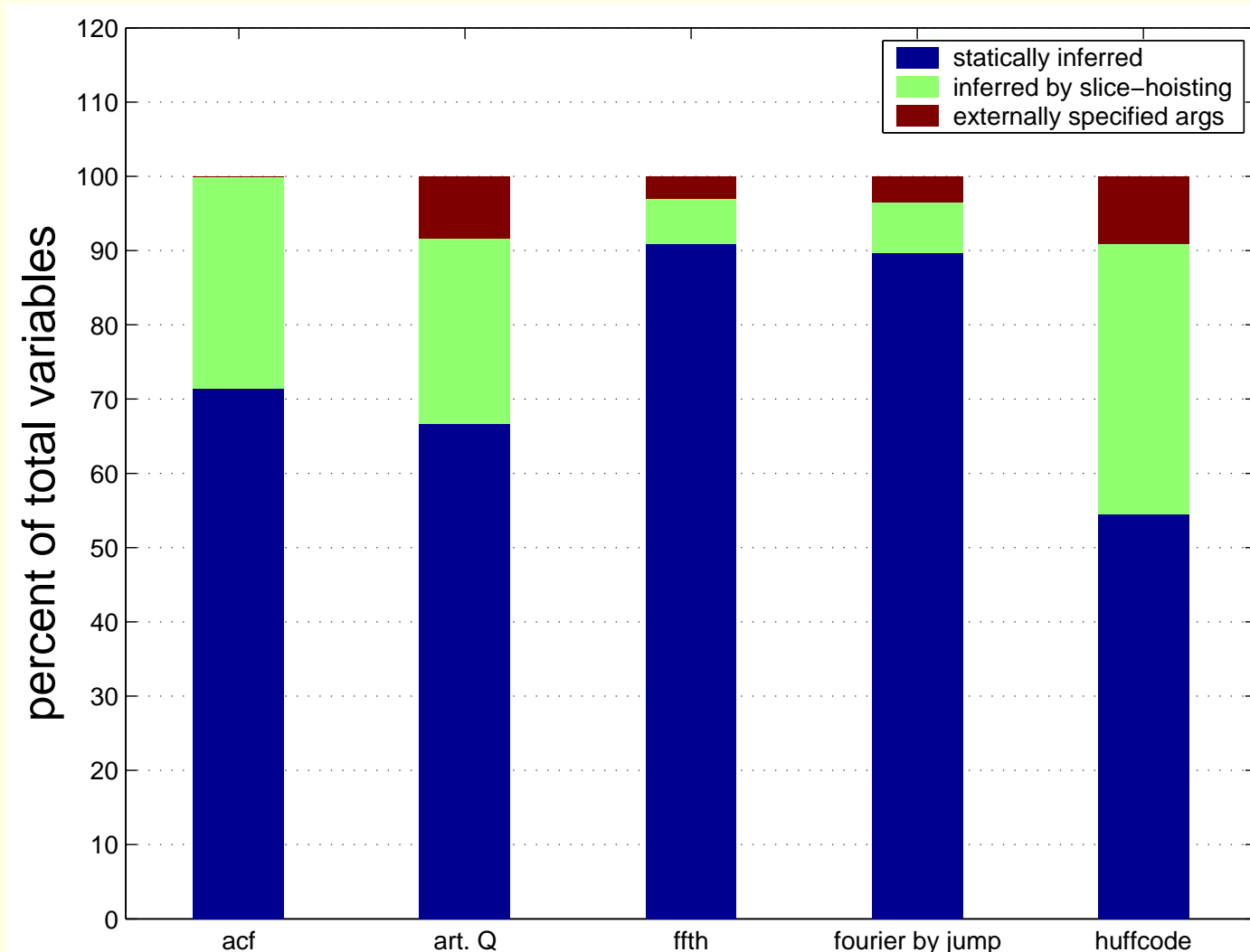
$A_1(1) = \dots$
 $\Rightarrow \sigma_1^{A_1} = \langle 1 \rangle$
 \dots
 $\Rightarrow x_1 = f(A_1)$
 $A_1(x_1) = \dots$
 $\Rightarrow \sigma_1^{A_1} = \max(\sigma_1^{A_1}, \langle x_1 \rangle)$
 \dots

dependence blocks hoisting

Precision of Static Inference



Inference Mechanisms



Advantages

- simple and fast, requiring only basic SSA analysis in its simplest form
- can leverage more advanced analyses, if available
- other optimization phases complement it
- subsumes the inspector-executor style
- works very well within the telescoping languages framework
- most common cases handled without any complicated analysis

Conclusion

- type inference an important enabling step in telescoping languages approach to compiling scripting languages
- static analysis for inferring types is necessary, but inadequate for array-sizes
- slice-hoisting complements the static analysis
 - has several advantages
- study of DSP applications shows excellent improvements in the precision of size-inference

Related Work

- type inference
 - Peng Tu and David Padua
 - Luiz de Rose and David Padua (FALCON)
 - Gheorghe Almási and David Padua (MaJIC)
- inspector-executor
 - Joel Saltz (CHAOS)
- array-sizes for storage management
 - Pramod Joisha and Prithviraj Banerjee
- preallocation
 - Vijay Menon and Keshav Pingali

Bonus Material

Pushing the Level Again

Pushing the Level Again

effective compilation

Pushing the Level Again

effective compilation

efficient compilation

Fundamental Observation

- libraries are the key in optimizing high-level scripting languages

`a = x * y` \Rightarrow `a = MATMULT(x, y)`

Fundamental Observation

- libraries are the key in optimizing high-level scripting languages

$$a = x * y \Rightarrow a = \text{MATMULT}(x, y)$$

- libraries practically **define** high-level languages!
 - a large effort in HPC is towards writing libraries
 - domain-specific libraries make scripting languages useful and popular
 - high-level operations are largely “syntactic sugar”

Telescoping Languages Approach

Telescoping Languages Approach

- pre-compile libraries to minimize end-user compilation time

Telescoping Languages Approach

- pre-compile libraries to minimize end-user compilation time
- annotate libraries to capture specialized knowledge of library writers

Telescoping Languages Approach

- pre-compile libraries to minimize end-user compilation time
- annotate libraries to capture specialized knowledge of library writers
- generate specialized variants based on interesting contexts

Telescoping Languages Approach

- pre-compile libraries to minimize end-user compilation time
- annotate libraries to capture specialized knowledge of library writers
- generate specialized variants based on interesting contexts
- link appropriate versions into the user script

Telescoping Languages Approach

- pre-compile libraries to minimize end-user compilation time
- annotate libraries to capture specialized knowledge of library writers
- generate specialized variants based on interesting contexts
- link appropriate versions into the user script

analogous to offline indexing by search engines