# 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

- type $\equiv$ $<\tau, \delta, \sigma, \psi>$
  - $\tau$ = intrinsic type, e.g., int, real, complex, etc.
  - $\delta$ = array dimensionality, 0 for scalars
  - $\sigma$ = $\delta$-tuple of positive integers
  - $\psi$ = "structure" of an array

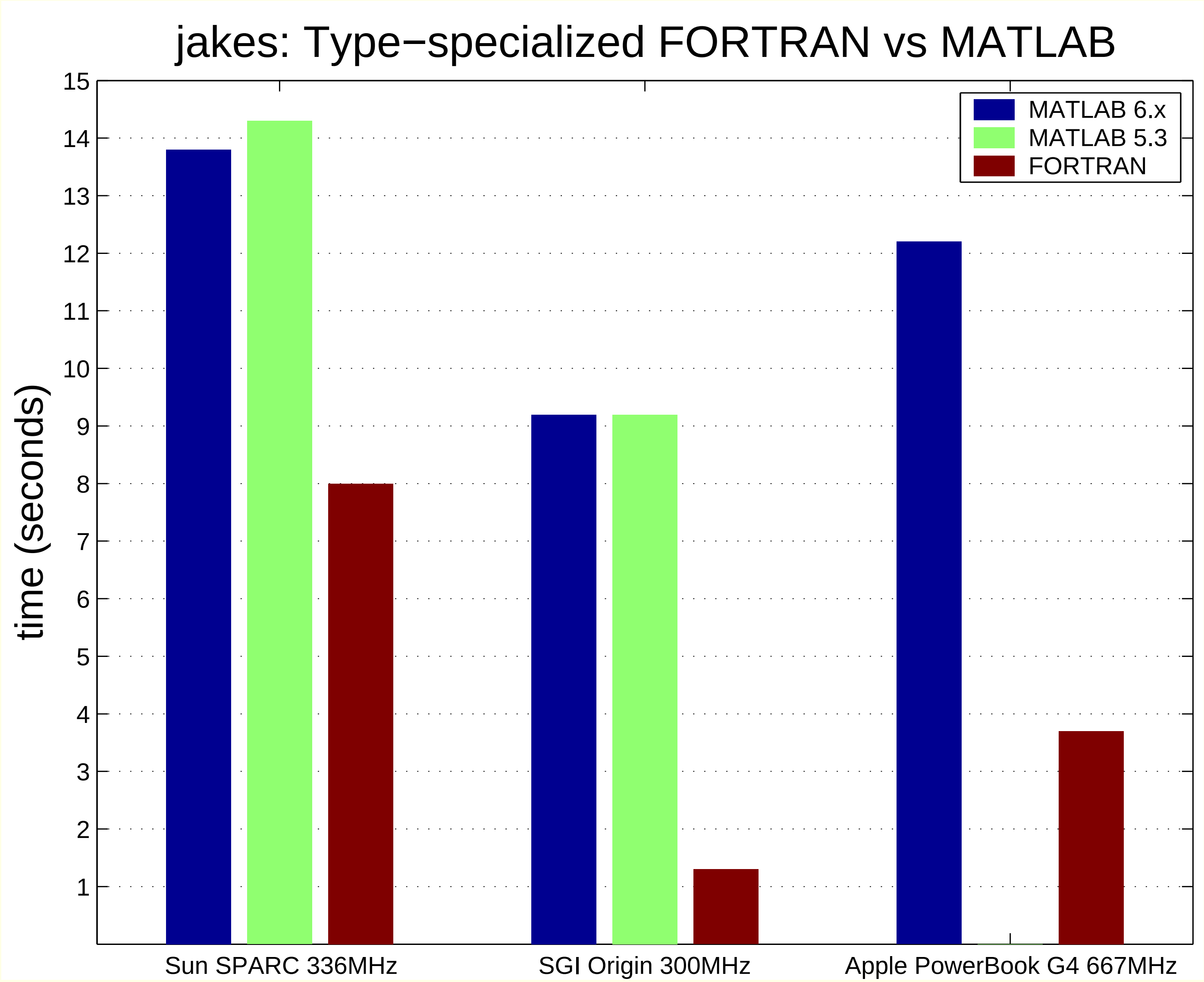
# Inferring Types

- type $\equiv <\tau, \delta, \sigma, \psi>$
  - $\tau$ = intrinsic type, e.g., int, real, complex, etc.
  - $\delta$ = array dimensionality, 0 for scalars
  - $\sigma$ = $\delta$-tuple of positive integers
  - $\psi$ = "structure" of an array
- type inference in general
  - type = "smallest" set of values that preserves meaning

# Inferring Types

- type $\equiv <\tau, \delta, \sigma, \psi>$
  - $\tau$ = intrinsic type, e.g., int, real, complex, etc.
  - $\delta$ = array dimensionality, 0 for scalars
  - $\sigma$ = $\delta$-tuple of positive integers
  - $\psi$ = "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

```
x = 1

y = x / 10

z = x * 20

r = y + z
```

# Static Type Inference
## (joint work with Cheryl McCosh)

- dimensionality constraints

| | |
|---|---|
| **x = 1** | LHS dims = RHS dims |
| **y = x / 10** | (x, y scalar) OR (x, y arrays of same size) |
| **z = x * 20** | (x, z scalar) OR (x, z arrays of same size) |
| **r = y + 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);
σ^A  = <N>
y  = ...
A (y ) = ...
σ^A  = max(σ^A , <y >)
x  = ...
A (x ) = ...
σ^A  = max(σ^A , <x >)
```

# Slice-hoisting: Simple Example

$\mathtt{A}_1 = \mathtt{zeros(1, N);}$

$\sigma_1^{A_1} = \langle \mathtt{N} \rangle$

$\mathtt{y}_1 = \ldots$

$\mathtt{A}_1(\mathtt{y}_1) = \ldots$

$\sigma_2^{A_1} = \mathtt{max}(\sigma_1^{A_1}, \langle \mathtt{y}_1 \rangle)$

$\mathtt{x}_1 = \ldots$

$\mathtt{A}_1(\mathtt{x}_1) = \ldots$

$\sigma_3^{A_1} = \mathtt{max}(\sigma_2^{A_1}, \langle \mathtt{x}_1 \rangle)$

# Slice-hoisting: Simple Example

$A_1$ = zeros(1, N);

$\Rightarrow \sigma_1^{A_1}$ = <N>

$\Rightarrow y_1$ = ...

$A_1(y_1)$ = ...

$\Rightarrow \sigma_2^{A_1}$ = max($\sigma_1^{A_1}$, <$y_1$>)

$\Rightarrow x_1$ = ...

$A_1(x_1)$ = ...

$\Rightarrow \sigma_3^{A_1}$ = max($\sigma_2^{A_1}$, <$x_1$>)

# Slice-hoisting: Simple Example

$\Rightarrow \sigma_{1}^{A_1}$ `= <N>`

$\Rightarrow y_{1}$ `= ...`

$\Rightarrow \sigma_{2}^{A_1}$ `= max(`$\sigma_{1}^{A_1}$`, <`$y_1$`>)`

$\Rightarrow x_{1}$ `= ...`

$\Rightarrow \sigma_{3}^{A_1}$ `= max(`$\sigma_{2}^{A_1}$`, <`$x_1$`>)`

`allocate(`$A_1$`, `$\sigma_{3}^{A_1}$`);`

$A_1$ `= zeros(1, N);`

$A_1(y_1)$ `= ...`

$A_1(x_1)$ `= ...`

# Slice-hoisting: Steps

- insert $\sigma$ statements
- do SSA conversion
- identify the slice involved in computing the $\sigma$ 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 ) = ...
σ^A = <x >
for i = 1:N
    ...

    A = [A f(i )];
    σ^A = σ^A + <1>
end
```

- add $\sigma$ statements

# Slice-hoisting: Loop

```
A_1(x_1) = ...
σ_1^{A_1} = <x_1>
for i_1 = 1:N
    ...
    σ_2^{A_1} = φ(σ_1^{A_1}, σ_3^{A_1})
    A_1 = [A_1 f(i_1)];
    σ_3^{A_1} = σ_2^{A_1} + <1>
end
```

- add $\sigma$ statements
- do SSA

# Slice-hoisting: Loop

```
  A₁(x₁) = ...
⇒σ₁^A₁ = <x₁>
⇒for i₁ = 1:N
     ...
⇒    σ₂^A₁ = φ(σ₁^A₁, σ₃^A₁)
     A₁ = [A₁ f(i₁)];
⇒    σ₃^A₁ = σ₂^A₁ + <1>
⇒end
```

- add $\sigma$ statements
- do SSA
- identify slice

# Slice-hoisting: Loop

```
⇒σ_1^{A_1} = <x_1>
⇒for i_1 = 1:N
⇒    σ_2^{A_1} = φ(σ_1^{A_1}, σ_3^{A_1})
⇒    σ_3^{A_1} = σ_2^{A_1} + <1>
⇒end
 allocate(A_1, σ_3^{A_1});
 A_1(x_1) = ...
 for i_1 = 1:N
     ...
     A_1 = [A_1 f(i_1)];
 end
```

- add $\sigma$ statements
- do SSA
- identify slice
- hoist the slice

# Slice-hoisting: Loop

$\Rightarrow \sigma_{\mathbf{3}}{}^{\mathbf{A_1}}$ = <$x_{\mathbf{1}}$> + <N>

allocate($A_{\mathbf{1}}$, $\sigma_{\mathbf{3}}{}^{\mathbf{A_1}}$);

$A_{\mathbf{1}}(x_{\mathbf{1}})$ = ...

for $i_{\mathbf{1}}$ = 1:N

  ...

  $A_{\mathbf{1}}$ = [$A_{\mathbf{1}}$ f($i_{\mathbf{1}}$)];

end

- add $\sigma$ statements
- do SSA
- identify slice
- hoist the slice

# Dependences Can Raise Roadblocks

```
A (1) = ...

...
x  = f(A )
A (x ) = ...

...
```

# Dependences Can Raise Roadblocks

```
A (1) = ...
σ^A = <1>
...
x = f(A )
A (x ) = ...
σ^A = max(σ^A , <x >)
...
```

# Dependences Can Raise Roadblocks

$\mathtt{A}_{\mathbf{1}}\mathtt{(1) = ...}$

$\sigma_{\mathbf{1}}{}^{\mathbf{A}_1}$ `= <1>`

`...`

$\mathtt{x}_{\mathbf{1}}$ `= f(`$\mathtt{A}_{\mathbf{1}}$`)`

$\mathtt{A}_{\mathbf{1}}$`(`$\mathtt{x}_{\mathbf{1}}$`) = ...`

$\sigma_{\mathbf{1}}{}^{\mathbf{A}_1}$ `= max(`$\sigma_{\mathbf{1}}{}^{\mathbf{A}_1}$`, <`$\mathtt{x}_{\mathbf{1}}$`>)`

`...`

# Dependences Can Raise Roadblocks

$$
\begin{array}{l}
\quad \texttt{A}_{\mathbf{1}}\texttt{(1) = ...} \\
\Rightarrow \sigma_{\mathbf{1}}{}^{\mathbf{A_1}} \texttt{ = <1>} \\
\quad \texttt{...} \\
\Rightarrow \texttt{x}_{\mathbf{1}} \texttt{ = f(A}_{\mathbf{1}}\texttt{)} \\
\quad \texttt{A}_{\mathbf{1}}\texttt{(x}_{\mathbf{1}}\texttt{) = ...} \\
\Rightarrow \sigma_{\mathbf{1}}{}^{\mathbf{A_1}} \texttt{ = max(}\sigma_{\mathbf{1}}{}^{\mathbf{A_1}}\texttt{, <x}_{\mathbf{1}}\texttt{>)} \\
\quad \texttt{...}
\end{array}
$$

# Dependences Can Raise Roadblocks

$A_1(1) = \ldots$

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

$\ldots$

$\Rightarrow x_1 = f(A_1)$

$A_1(x_1) = \ldots$

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

$\ldots$

# Dependences Can Raise Roadblocks

$A_1(1) = \ldots$

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

$\ldots$

$\Rightarrow x_1 = f(A_1)$

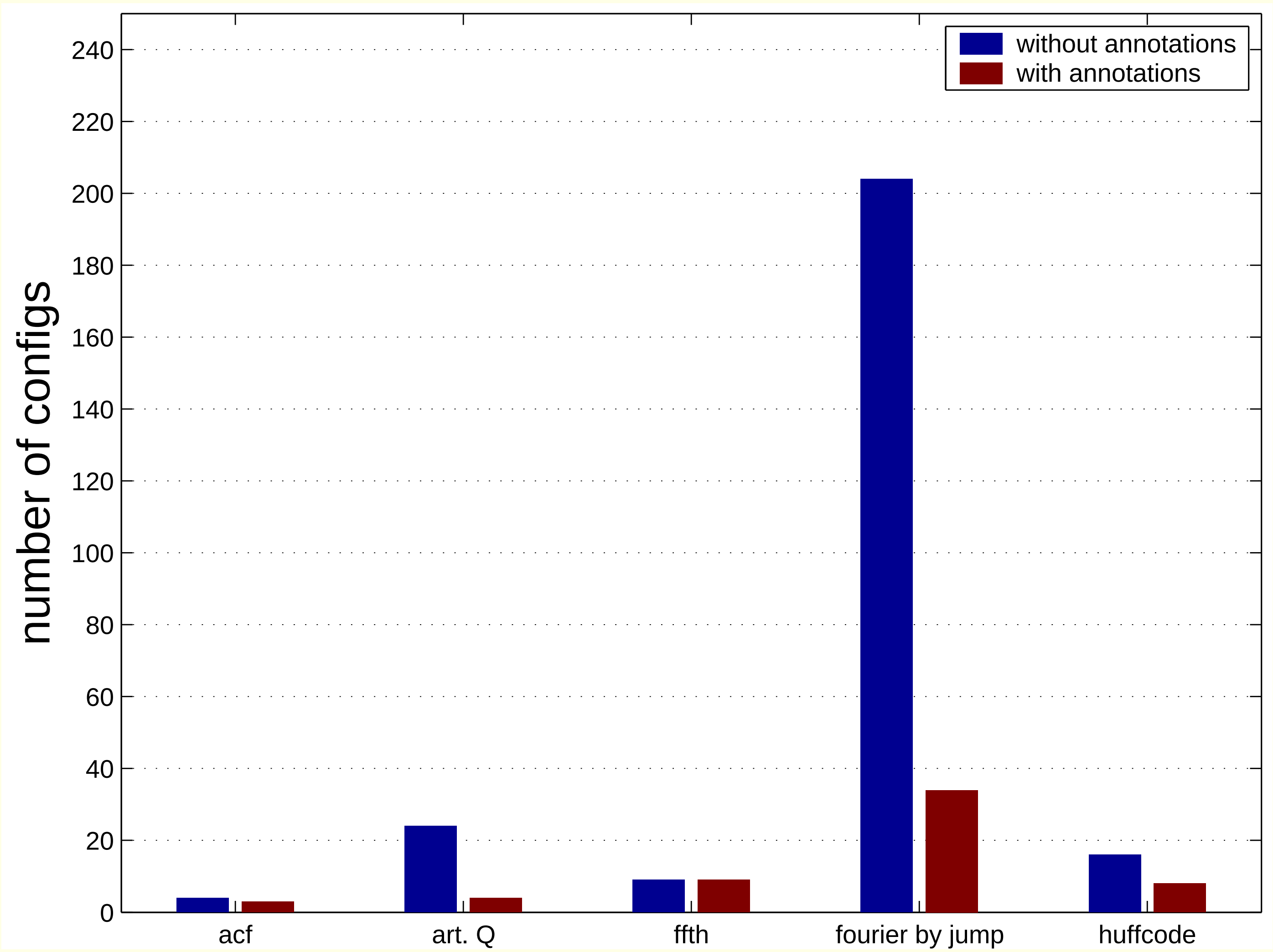
$A_1(x_1) = \ldots$

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

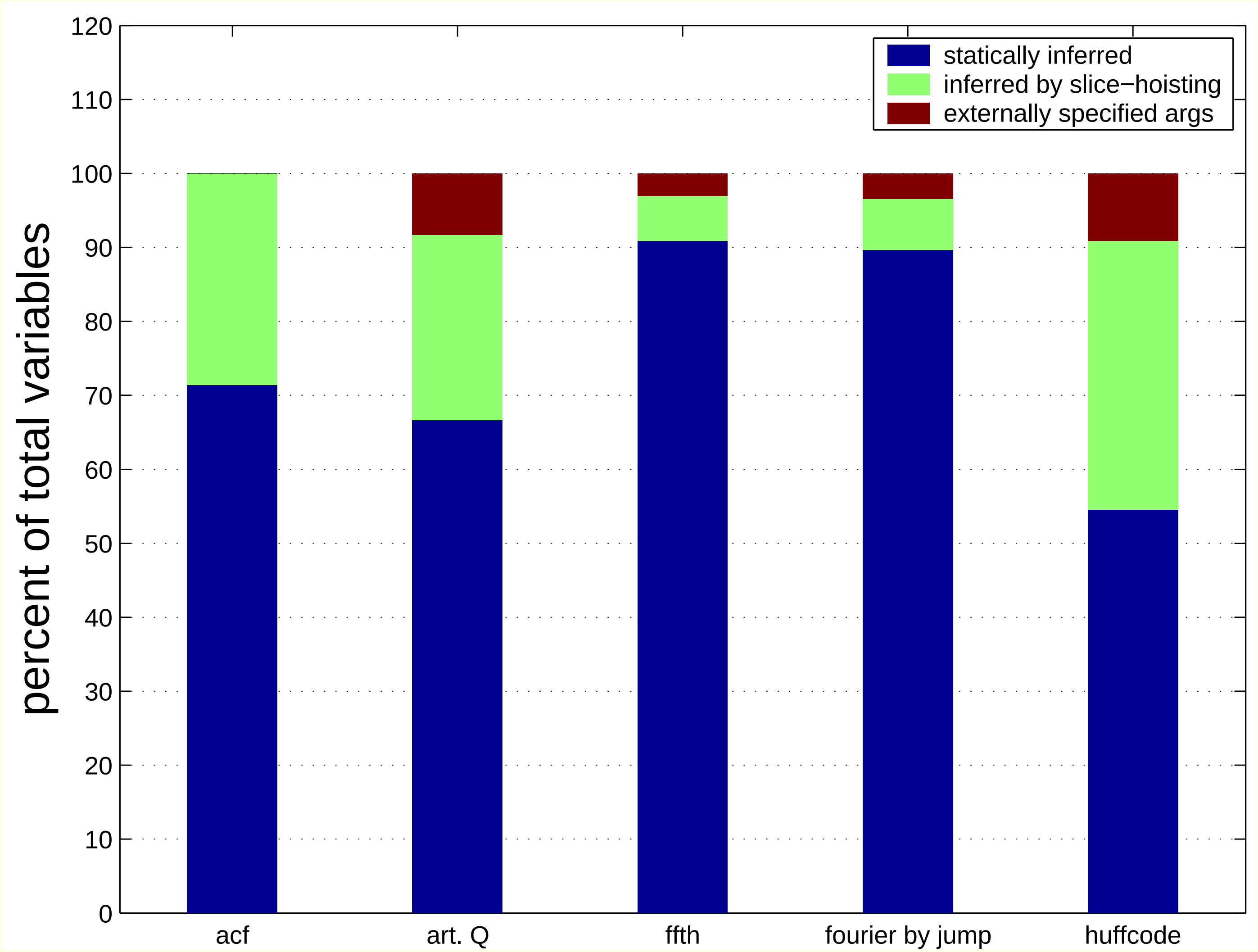
$\ldots$

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  ⇒  a = 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