Telescoping MATLAB for DSP Applications

PhD Thesis Defense

Arun Chauhan

Computer Science, Rice University

Two True Stories

Two True Stories

- the world of Digital Signal Processing

Two True Stories

- the world of Digital Signal Processing
  - almost everyone uses MATLAB
  - a large number uses MATLAB exclusively
  - almost everyone hates writing C code
  - prefer coding for an hour and letting it run for 7 days, than the other way round
  - often forced to rewrite programs in C

Two True Stories

- the world of Digital Signal Processing
  - almost everyone uses MATLAB
  - a large number uses MATLAB exclusively
  - almost everyone hates writing C code
  - prefer coding for an hour and letting it run for 7 days, than the other way round
  - often forced to rewrite programs in C
- linear algebra through MATLAB

Two True Stories

- the world of Digital Signal Processing
  - almost everyone uses MATLAB
  - a large number uses MATLAB exclusively
  - almost everyone hates writing C code
  - prefer coding for an hour and letting it run for 7 days, than the other way round
  - often forced to rewrite programs in C
- linear algebra through MATLAB
  - ARPACK—a linear algebra package to solve eigenvalue problems
  - prototyped in MATLAB
  - painfully hand translated to FORTRAN

Lessons

Lessons

- programming is an unavoidable fact of life to conduct research in science and engineering

Lessons

- programming is an unavoidable fact of life to conduct research in science and engineering
- users do not like programming in traditional languages

Lessons

- programming is an unavoidable fact of life to conduct research in science and engineering
- users do not like programming in traditional languages
- users love domain-specific high-level scripting languages
  - MATLAB has over 500,000 worldwide licenses
  - Python, Perl, R, Mathematica

Lessons

- programming is an unavoidable fact of life to conduct research in science and engineering
- users do not like programming in traditional languages
- users love domain-specific high-level scripting languages
  - MATLAB has over 500,000 worldwide licenses
  - Python, Perl, R, Mathematica
- performance problems limit their use

Lessons

- programming is an unavoidable fact of life to conduct research in science and engineering
- users do not like programming in traditional languages
- users love domain-specific high-level scripting languages
  - MATLAB has over 500,000 worldwide licenses
  - Python, Perl, R, Mathematica
- performance problems limit their use
- the productivity connection

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

Pushing the Level Again

Pushing the Level Again

effective compilation

Pushing the Level Again

effective compilation

efficient compilation

Thesis

It is possible to efficiently compile numerical programs written in high-level languages to achieve performance close to that achievable in a lower-level language.

Fundamental Observation

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

```
a = x * y
```

$\Rightarrow$

```
a = MATMULT(x, y)
```

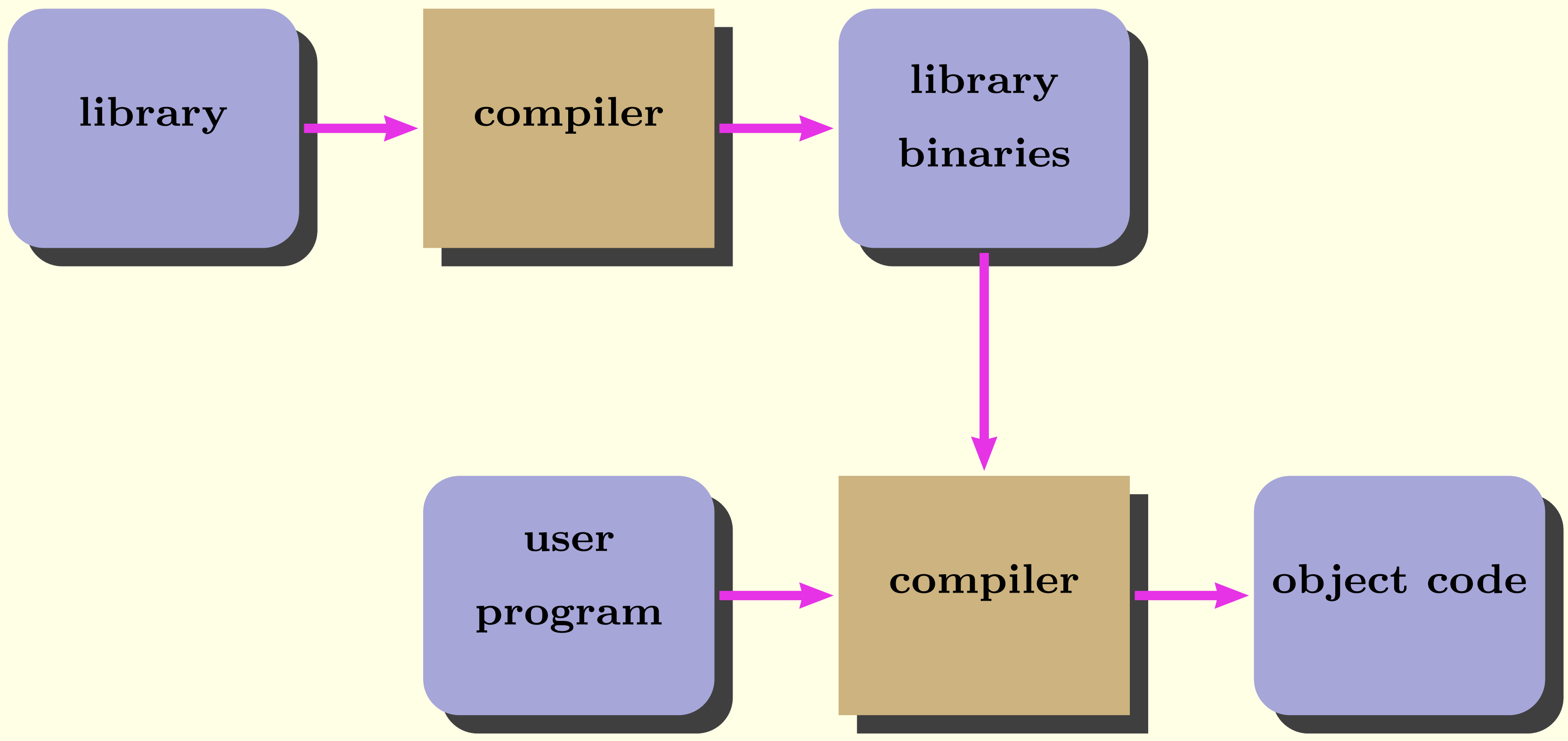
Fundamental Observation

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

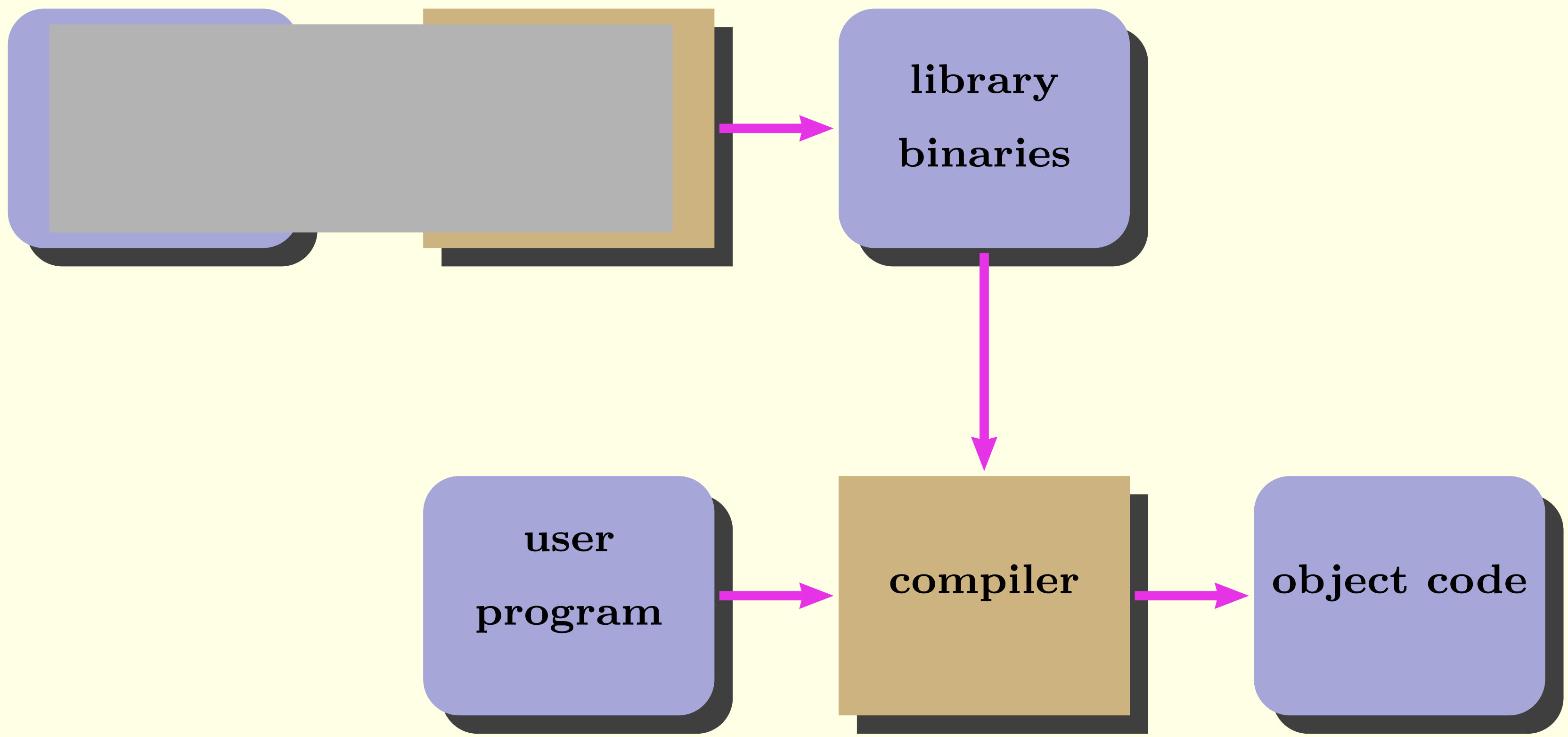
```
a = x * y  ⇒  a = MATMULT(x, y)
```

- libraries **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"

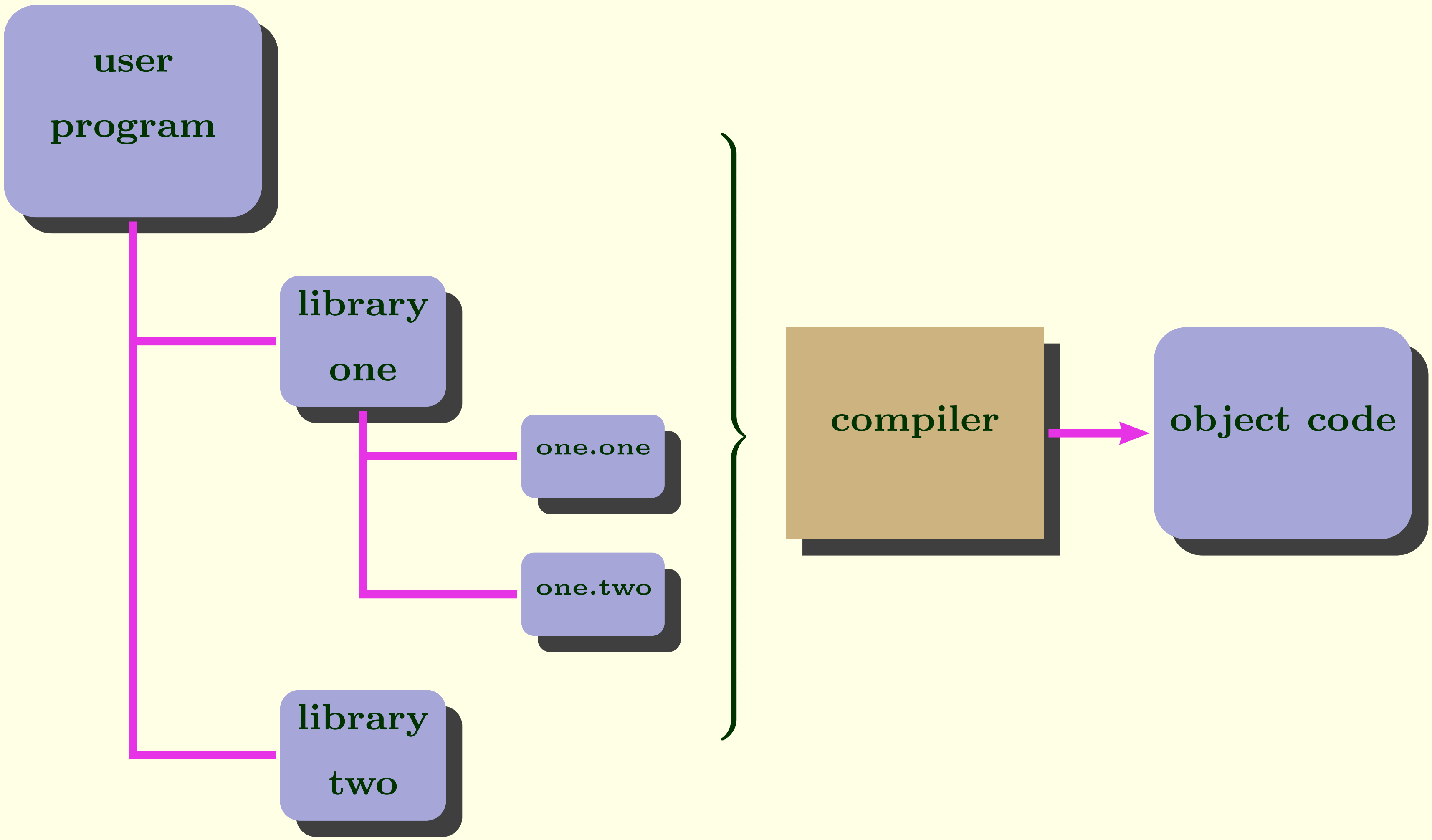
Libraries as Black Boxes



Libraries as Black Boxes



Whole Program Compilation



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

$\boxed{\text{analogous to offline indexing by search engines}}$

Telescoping Languages: Entities

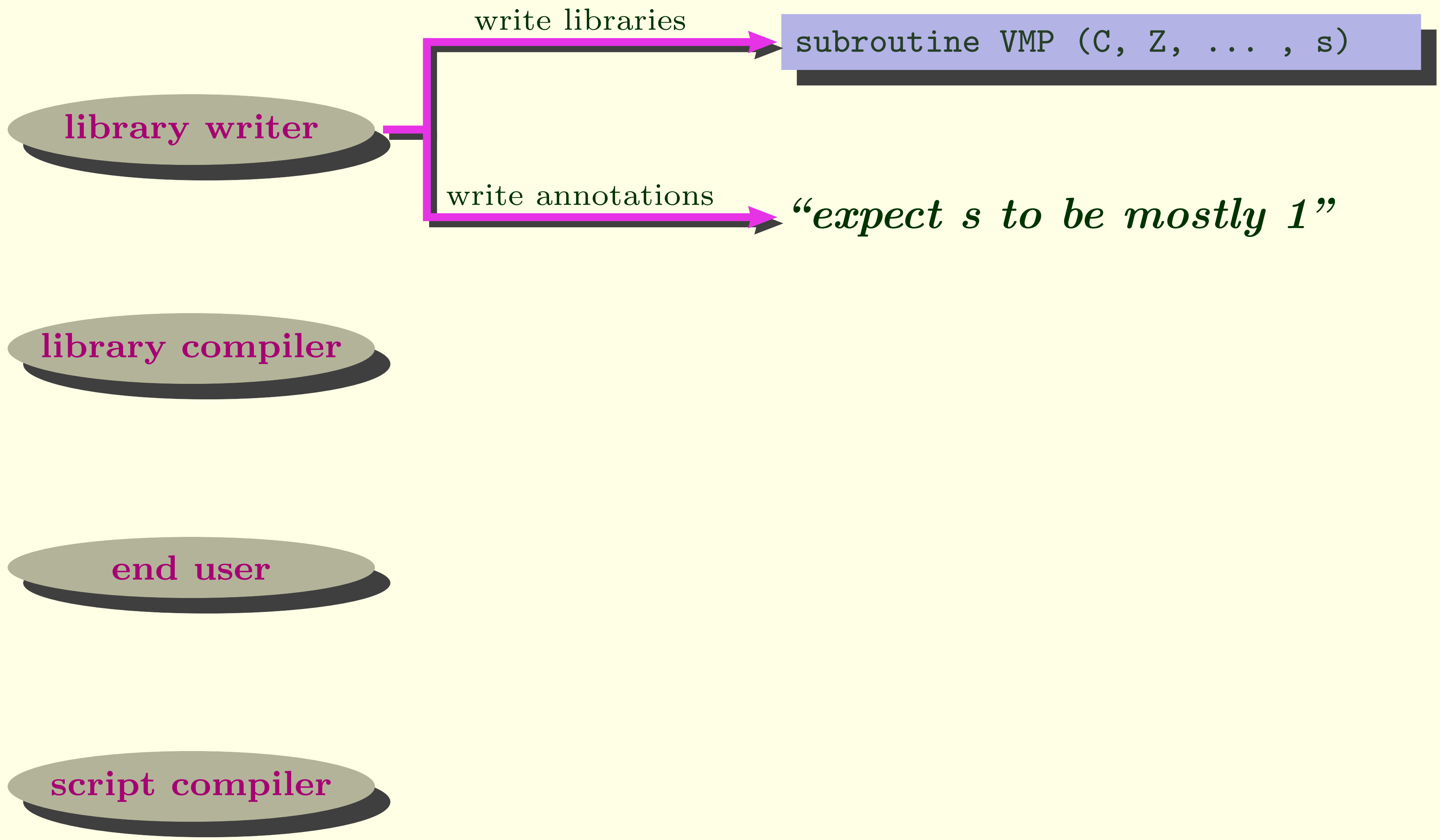
library writer

library compiler

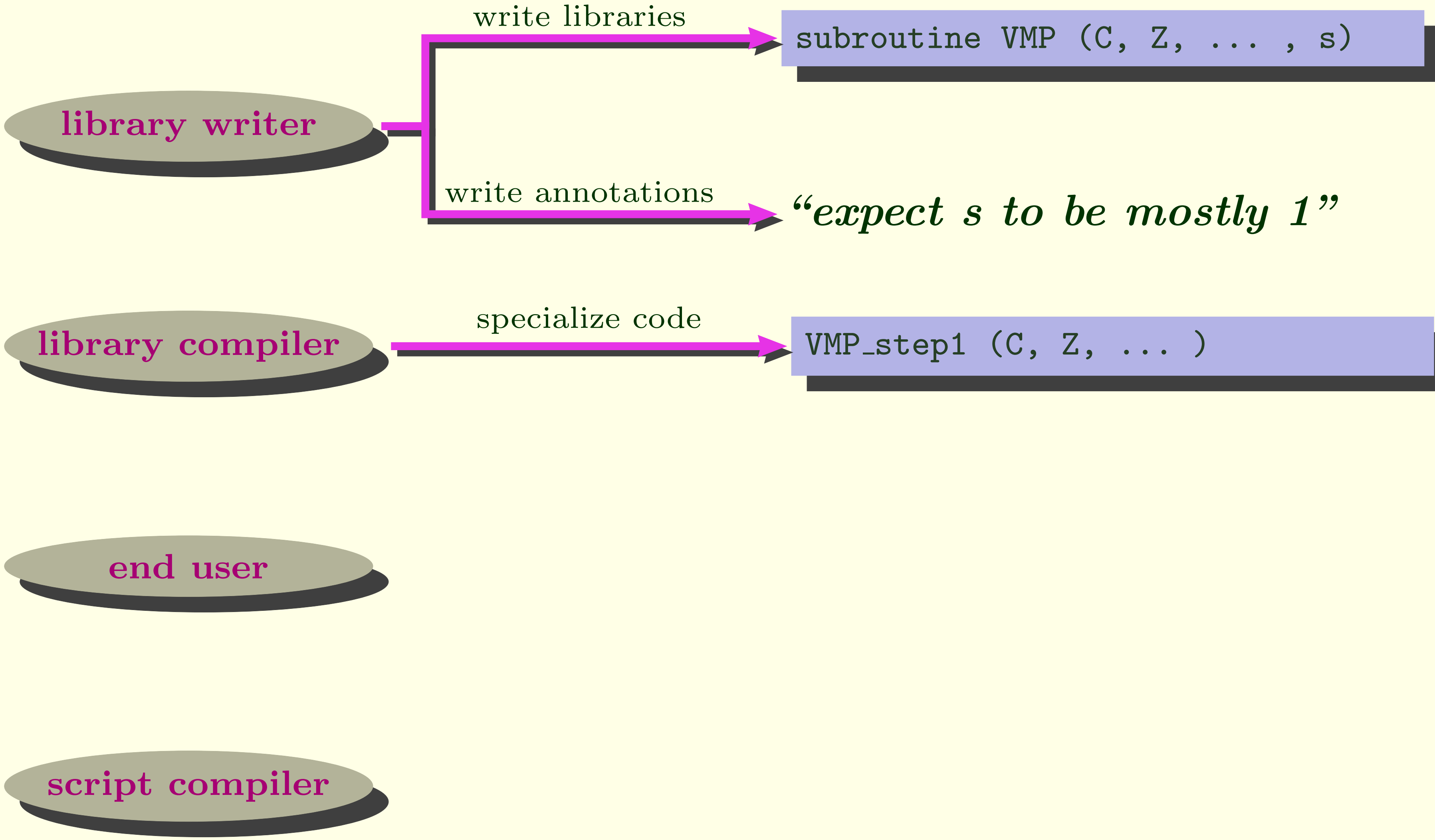
end user

script compiler

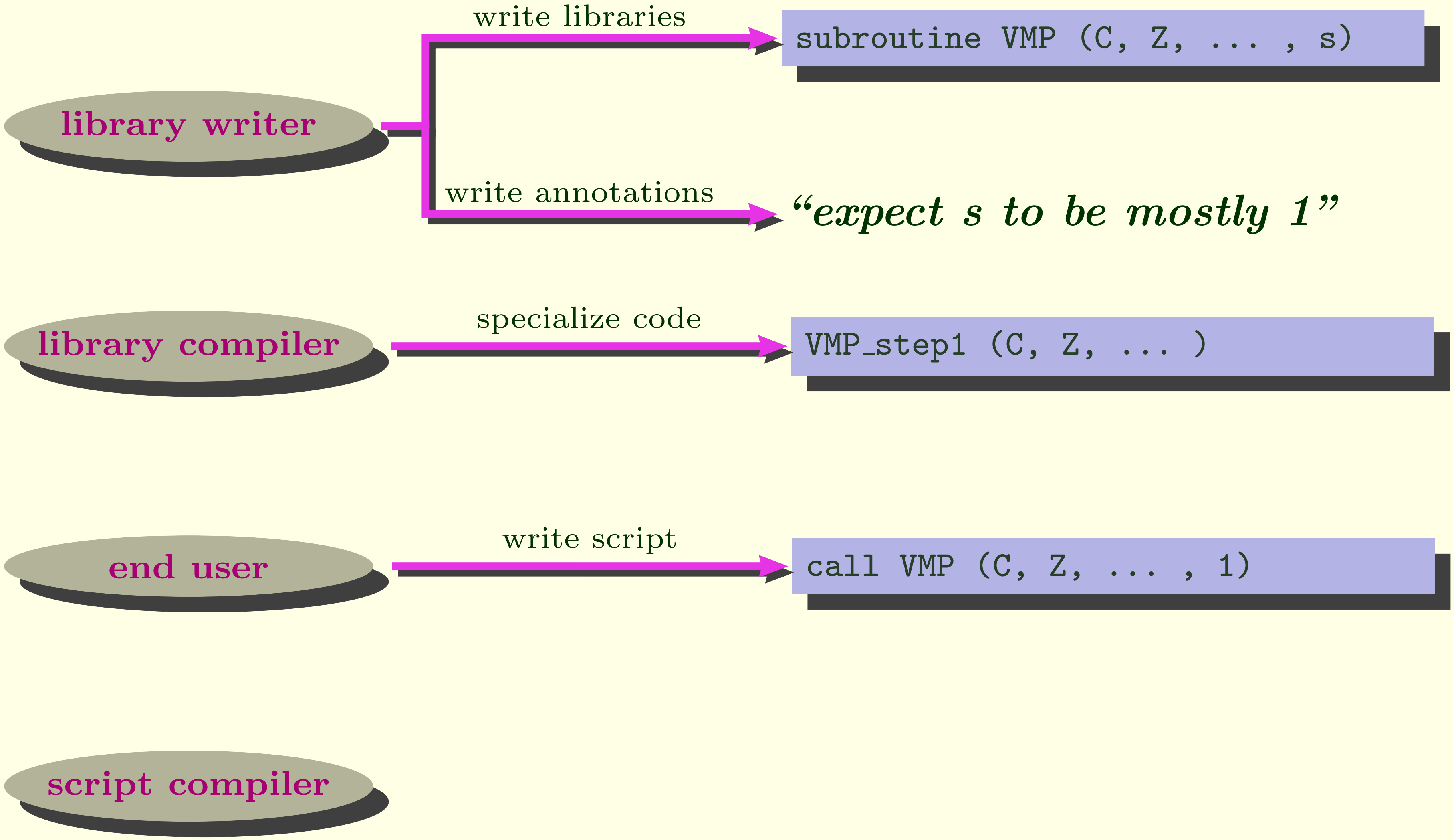
Telescoping Languages: Entities



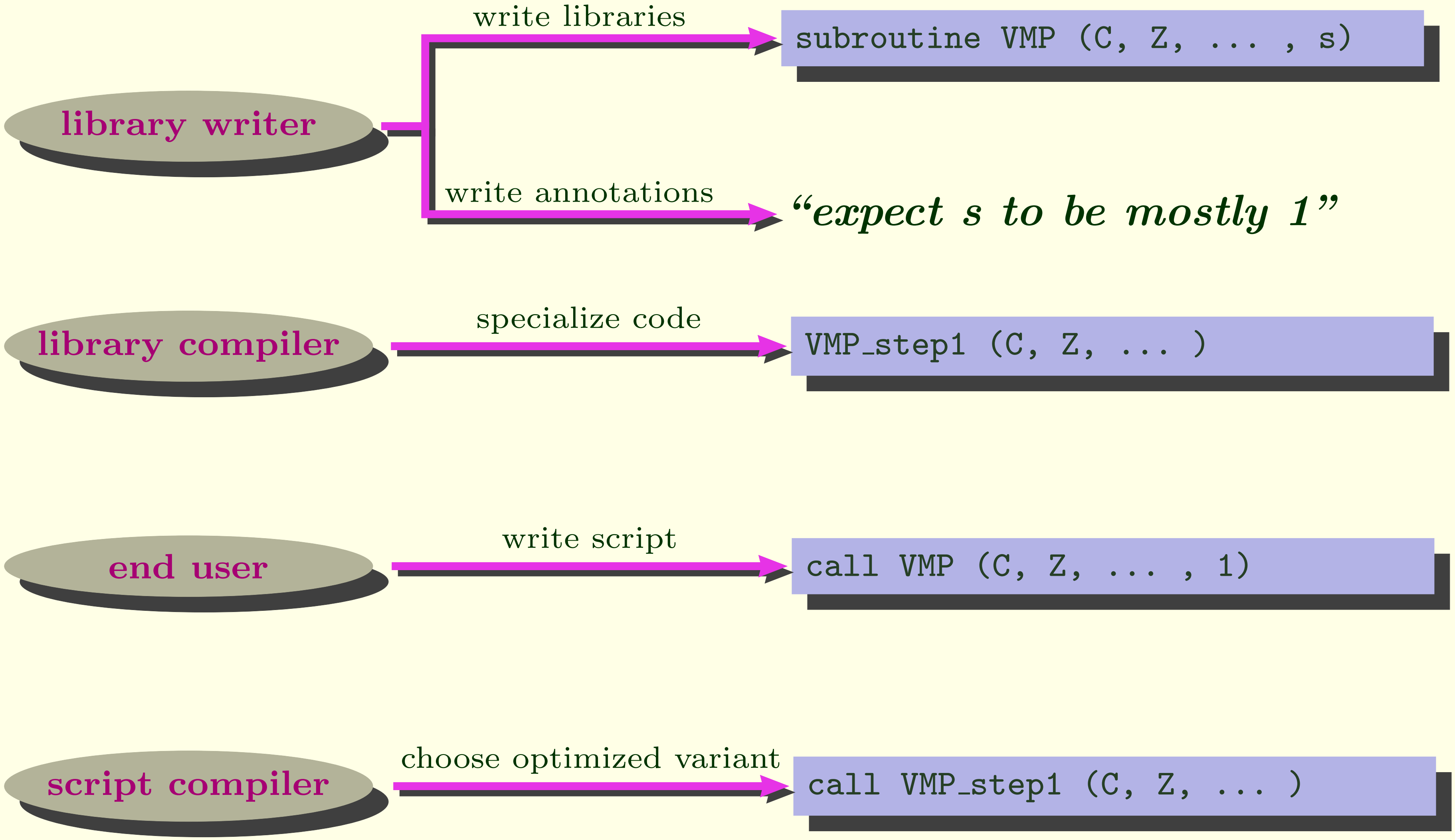
Telescoping Languages: Entities



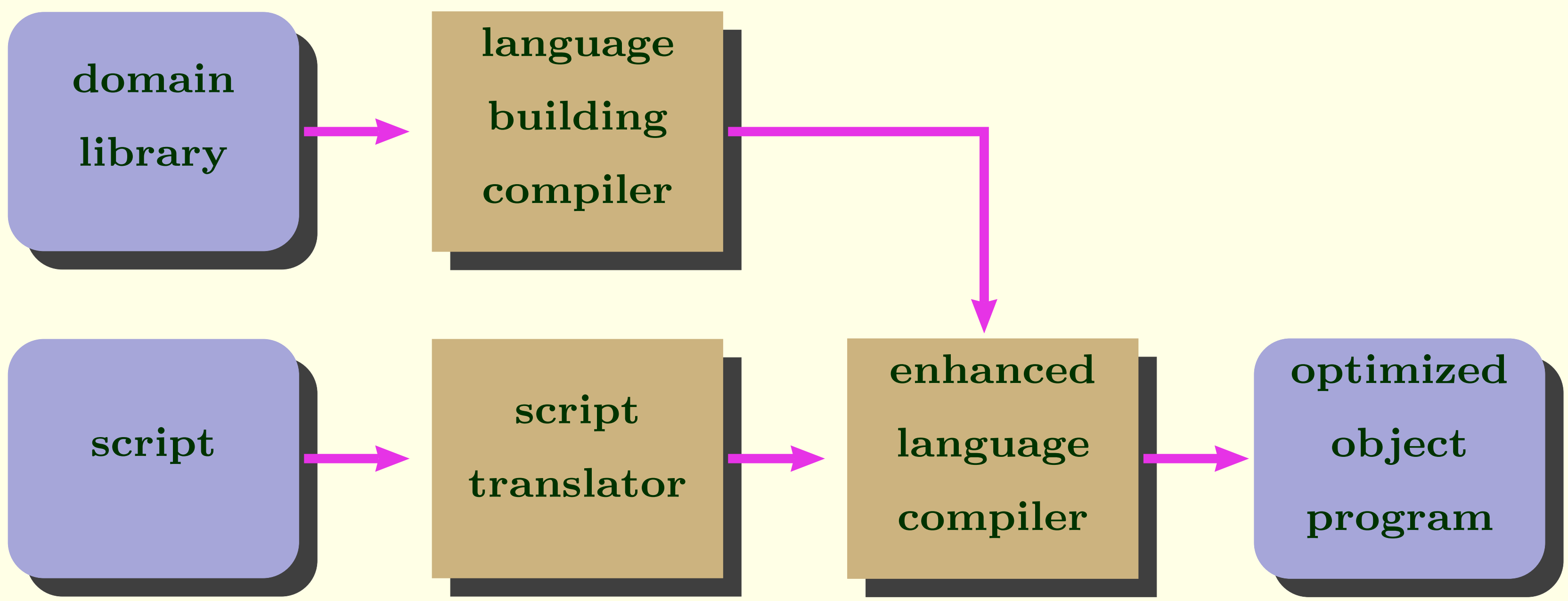
Telescoping Languages: Entities



Telescoping Languages: Entities



Telescoping Languages Approach



Challenges

Challenges

- identifying specialization opportunities
  - which kinds of specializations
  - how many

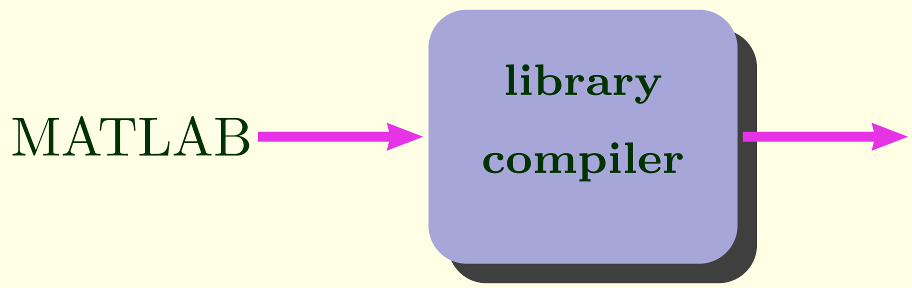
Challenges

- identifying specialization opportunities
  - which kinds of specializations
  - how many
- identifying high pay-off optimizations
  - must be applicable in telescoping languages context
  - should focus on these first

Challenges

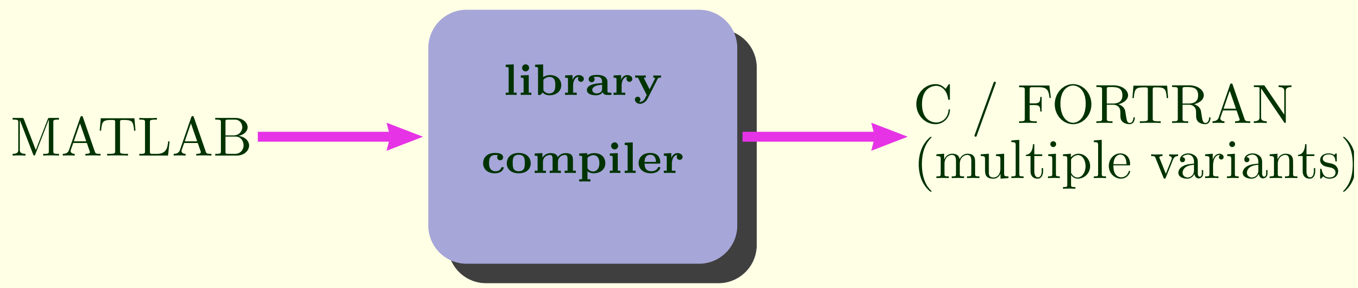
- identifying specialization opportunities
  - which kinds of specializations
  - how many
- identifying high pay-off optimizations
  - must be applicable in telescoping languages context
  - should focus on these first
- enabling the library writer to express these transformations
  - guide the specialization

Developing the Compiler



- compile MATLAB
- emit specialized output code
- implement identified high-payoff optimizations
- implement newly discovered optimizations

Developing the Compiler



- compile MATLAB
- emit specialized output code
- implement identified high-payoff optimizations
- implement newly discovered optimizations

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
(Joint work with Cheryl McCosh)

Inferring Types
(Joint work with Cheryl McCosh)

- 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
(Joint work with Cheryl McCosh)

- type $\equiv <\tau, \delta, \sigma, \psi>$
  - 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

Static Type Inference
(Appears in McCosh's Masters' Thesis)

- dimensionality constraints

x = 1

y = x / 10

z = x * 20

r = y + z

Static Type Inference

(Appears in McCosh's Masters' Thesis)

- dimensionality constraints

x = 1 LHS dims = RHS dims

y = x / 10

z = x * 20

r = y + z

Static Type Inference
(Appears in McCosh's Masters' Thesis)

- dimensionality constraints

x = 1	LHS dims = RHS dims
y = x / 10	(x, y scalar) OR (x, y arrays of same size)
z = x * 20	
r = y + z	

Static Type Inference
(Appears in McCosh's Masters' Thesis)

- 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	

Static Type Inference
(Appears in McCosh's Masters' Thesis)

- 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
(Appears in McCosh's Masters' Thesis)

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

Static Type Inference
(Appears in McCosh's Masters' Thesis)

- 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
(Appears in McCosh's Masters' Thesis)

- 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

- control join-points may result in too many configs

Limitations

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

Size Grows 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

$$
\begin{aligned}
&\texttt{A}_{\mathbf{1}} \texttt{ = zeros(1, N);}\\
&\sigma_{\mathbf{1}}{}^{\mathbf{A}_1} \texttt{ = <N>}\\
&\texttt{y}_{\mathbf{1}} \texttt{ = ...}\\
&\texttt{A}_{\mathbf{1}}\texttt{(y}_{\mathbf{1}}\texttt{) = ...}\\
&\sigma_{\mathbf{2}}{}^{\mathbf{A}_1} \texttt{ = max(}\sigma_{\mathbf{1}}{}^{\mathbf{A}_1}\texttt{, <y}_{\mathbf{1}}\texttt{>)}\\
&\texttt{x}_{\mathbf{1}} \texttt{ = ...}\\
&\texttt{A}_{\mathbf{1}}\texttt{(x}_{\mathbf{1}}\texttt{) = ...}\\
&\sigma_{\mathbf{3}}{}^{\mathbf{A}_1} \texttt{ = max(}\sigma_{\mathbf{2}}{}^{\mathbf{A}_1}\texttt{, <x}_{\mathbf{1}}\texttt{>)}
\end{aligned}
$$

Slice-hoisting: Simple Example

$A_1 = \texttt{zeros(1, N);}$

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

$\Rightarrow y_1 = \ldots$

$A_1(y_1) = \ldots$

$\Rightarrow \sigma_2^{A_1} = \max(\sigma_1^{A_1}, \langle y_1 \rangle)$

$\Rightarrow x_1 = \ldots$

$A_1(x_1) = \ldots$

$\Rightarrow \sigma_3^{A_1} = \max(\sigma_2^{A_1}, \langle x_1 \rangle)$

Slice-hoisting: Simple Example

$\Rightarrow \sigma_{1}^{A_{1}}\ \texttt{= <N>}$

$\Rightarrow \texttt{y}_{1}\ \texttt{= ...}$

$\Rightarrow \sigma_{2}^{A_{1}}\ \texttt{= max(}\sigma_{1}^{A_{1}}\texttt{, <y}_{1}\texttt{>)}$

$\Rightarrow \texttt{x}_{1}\ \texttt{= ...}$

$\Rightarrow \sigma_{3}^{A_{1}}\ \texttt{= max(}\sigma_{2}^{A_{1}}\texttt{, <x}_{1}\texttt{)}$

$\texttt{allocate(A}_{1}\texttt{, }\sigma_{3}^{A_{1}}\texttt{);}$

$\texttt{A}_{1}\ \texttt{= zeros(1, N);}$

$\texttt{A}_{1}\texttt{(y}_{1}\texttt{) = ...}$

$\texttt{A}_{1}\texttt{(x}_{1}\texttt{) = ...}$

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 ) = ...
σ A  = <x >
for i  = 1:N
   ...

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

- add σ statements

Slice-hoisting: Loop

$$
\begin{array}{l}
\texttt{A}_{\mathbf{1}}\texttt{(x}_{\mathbf{1}}\texttt{) = ...} \\
\sigma_{\mathbf{1}}{}^{\mathbf{A_1}}\ \texttt{= <x}_{\mathbf{1}}\texttt{>} \\
\texttt{for i}_{\mathbf{1}}\texttt{ = 1:N} \\
\quad \texttt{...} \\
\quad \sigma_{\mathbf{2}}{}^{\mathbf{A_1}}\ \texttt{=}\ \phi(\sigma_{\mathbf{1}}{}^{\mathbf{A_1}},\ \sigma_{\mathbf{3}}{}^{\mathbf{A_1}}) \\
\quad \texttt{A}_{\mathbf{1}}\texttt{ = [A}_{\mathbf{1}}\texttt{ f(i}_{\mathbf{1}}\texttt{)];} \\
\quad \sigma_{\mathbf{3}}{}^{\mathbf{A_1}}\ \texttt{=}\ \sigma_{\mathbf{2}}{}^{\mathbf{A_1}}\ \texttt{+ <1>} \\
\texttt{end}
\end{array}
$$

- add σ statements
- do SSA

Slice-hoisting: Loop

$\quad \mathtt{A}_{\mathbf{1}}(\mathtt{x}_{\mathbf{1}}) = \ldots$

$\Rightarrow \sigma_{\mathbf{1}}{}^{\mathbf{A}_1} = <\mathtt{x}_{\mathbf{1}}>$

$\Rightarrow \mathtt{for}\ \mathtt{i}_{\mathbf{1}} = 1{:}\mathtt{N}$

$\qquad \ldots$

$\Rightarrow \qquad \sigma_{\mathbf{2}}{}^{\mathbf{A}_1} = \phi(\sigma_{\mathbf{1}}{}^{\mathbf{A}_1},\ \sigma_{\mathbf{3}}{}^{\mathbf{A}_1})$

$\qquad \mathtt{A}_{\mathbf{1}} = [\mathtt{A}_{\mathbf{1}}\ \mathtt{f}(\mathtt{i}_{\mathbf{1}})];$

$\Rightarrow \qquad \sigma_{\mathbf{3}}{}^{\mathbf{A}_1} = \sigma_{\mathbf{2}}{}^{\mathbf{A}_1} + <1>$

$\Rightarrow \mathtt{end}$

- add σ statements
- do SSA
- identify slice

Slice-hoisting: Loop

$\Rightarrow \sigma_1^{A_1} = <x_1>$

$\Rightarrow$ `for` $i_1 = 1{:}N$

$\Rightarrow \quad \sigma_2^{A_1} = \phi(\sigma_1^{A_1}, \sigma_3^{A_1})$

$\Rightarrow \quad \sigma_3^{A_1} = \sigma_2^{A_1} + <1>$

$\Rightarrow$ `end`

`allocate`$(A_1, \sigma_3^{A_1});$

$A_1(x_1) = \ldots$

`for` $i_1 = 1{:}N$

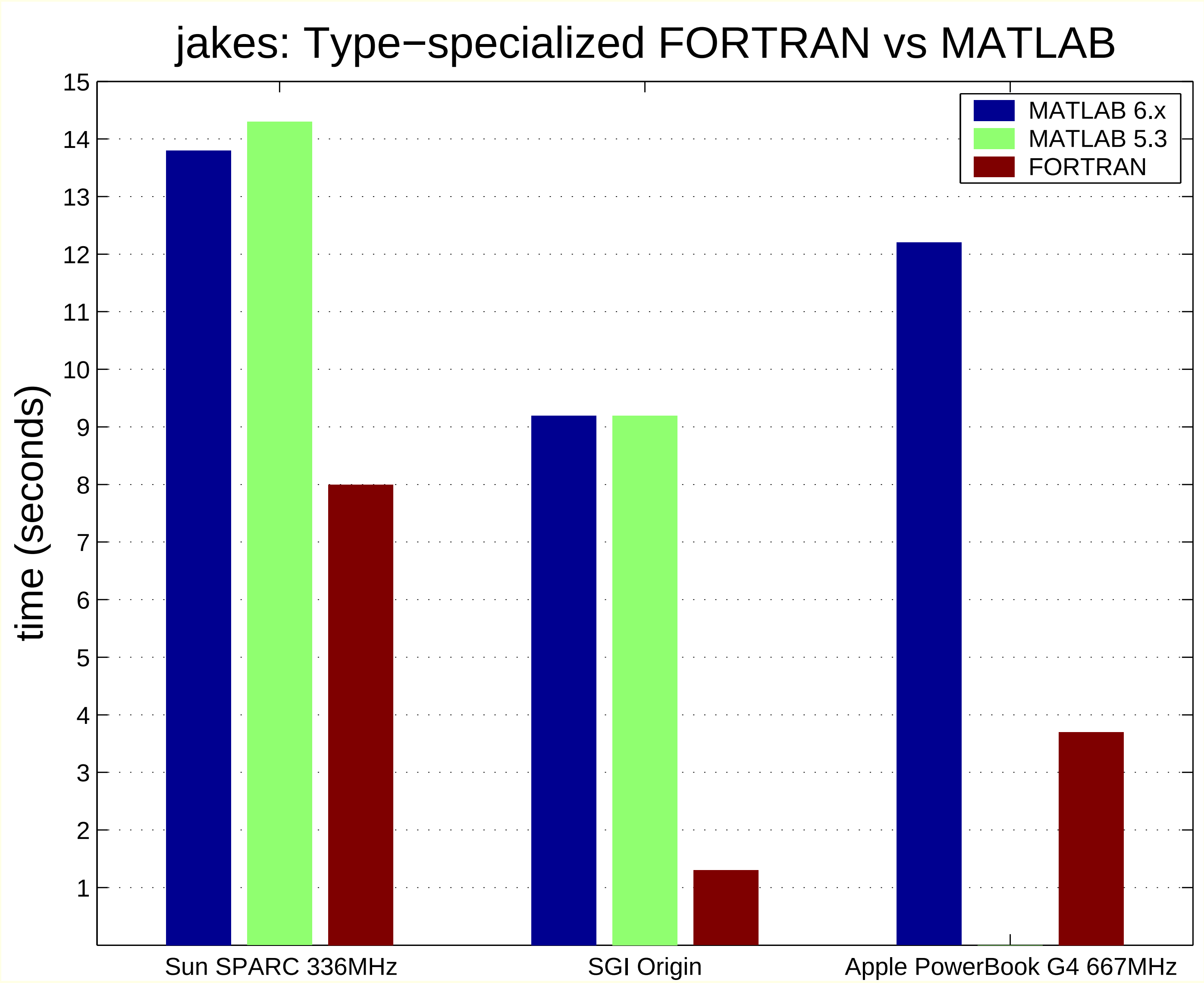
$\quad \ldots$

$\quad A_1 = [A_1\ f(i_1)];$

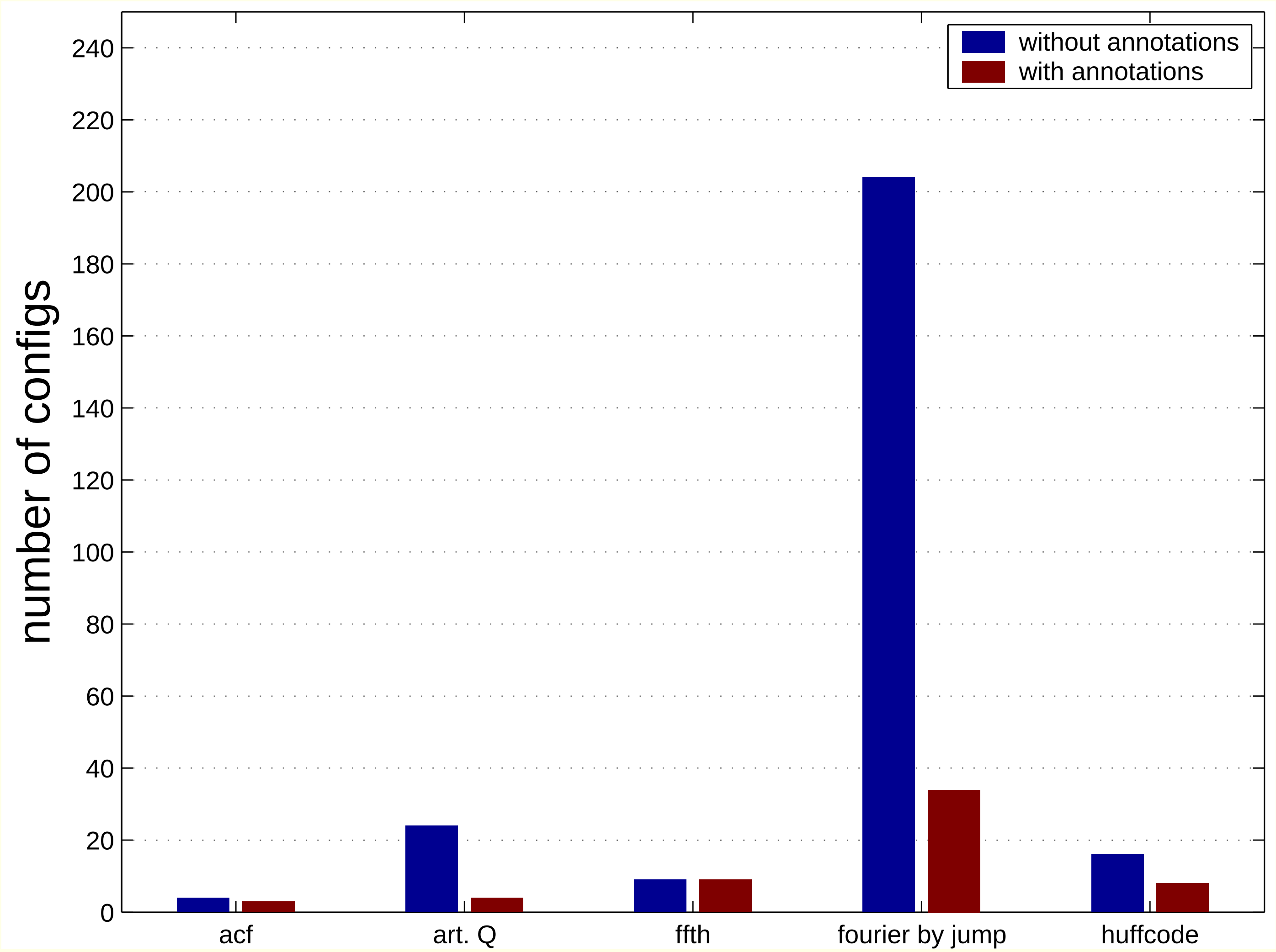
`end`

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

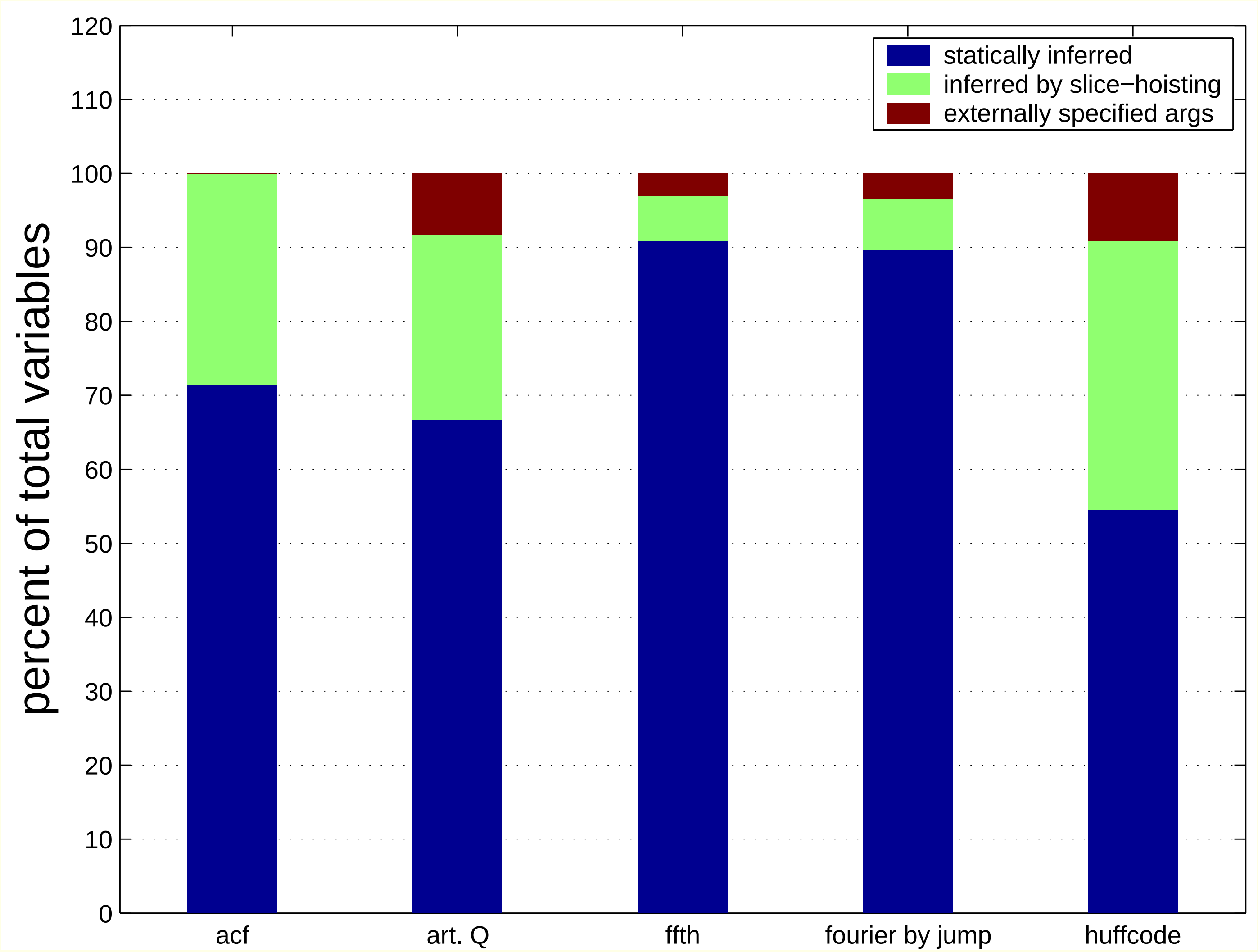
Type-based Specialization



Precision of Static Inference



Inference Mechanisms



Relevant Optimizations

"It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts."

–Sir Arthur Conon Doyle in a *A Scandal in Bohemia*

Identifying and Discovering

- study of DSP applications
  - real life code from the ECE department
- identified high-payoff well-known optimization techniques
- discovered two novel optimizations
  - procedure strength reduction
  - procedure vectorization

High-payoff Optimizations

- vectorization
  - 33 times speedup in one case!
- common subexpression elimination
- beating and dragging along
- constant propagation

High-payoff Optimizations

- vectorization
  - 33 times speedup in one case!
- common subexpression elimination
- beating and dragging along
- constant propagation
- library identities
  - single call replaces a sequence
- value of library annotations

Procedure Strength Reduction

Procedure Strength Reduction

```
for i = 1:N
   ...
   f (c1, c2, i, c3);
   ...
end
```

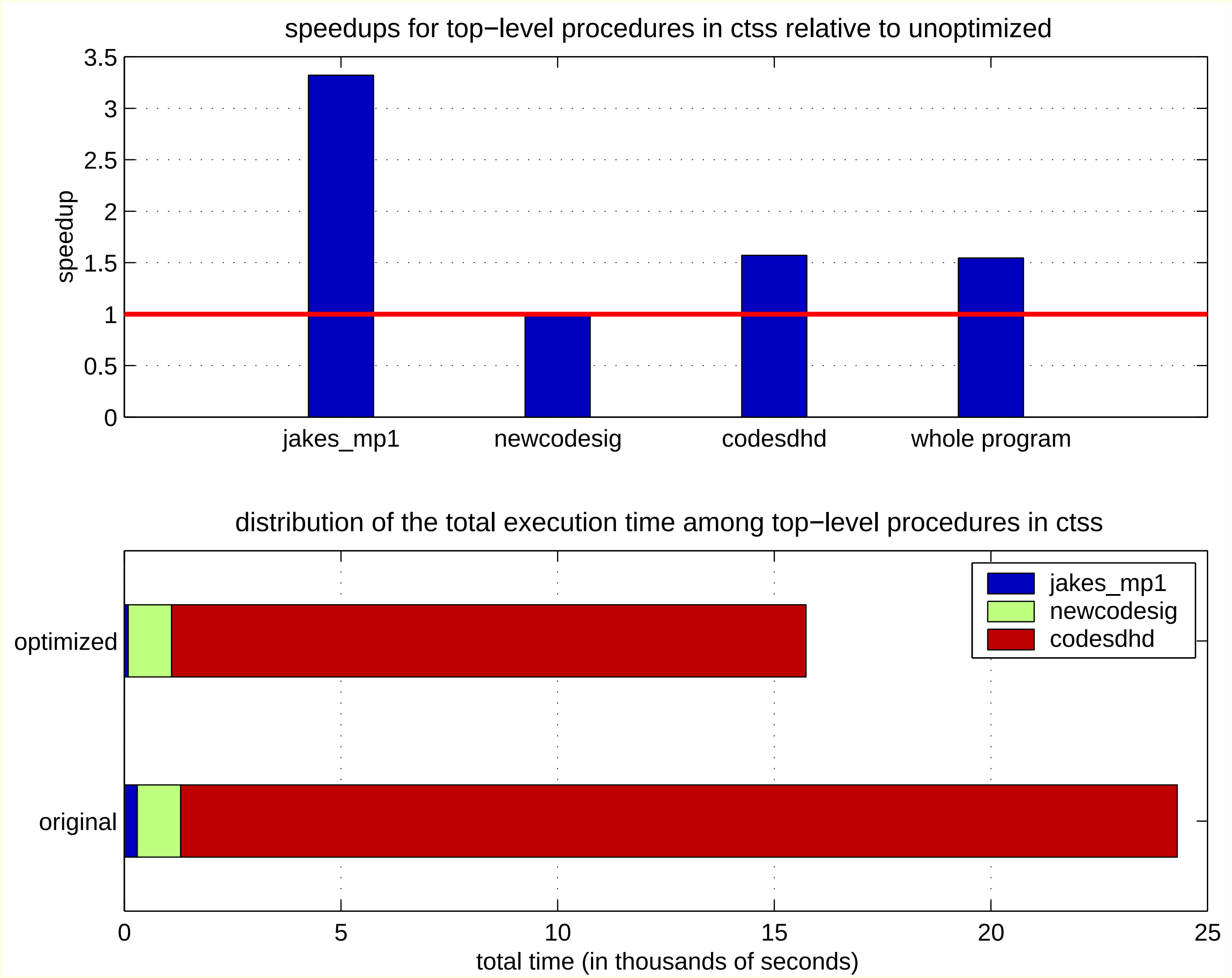
Procedure Strength Reduction

```
for i = 1:N
    ...
    f (c1, c2, i, c3);
    ...
end
```

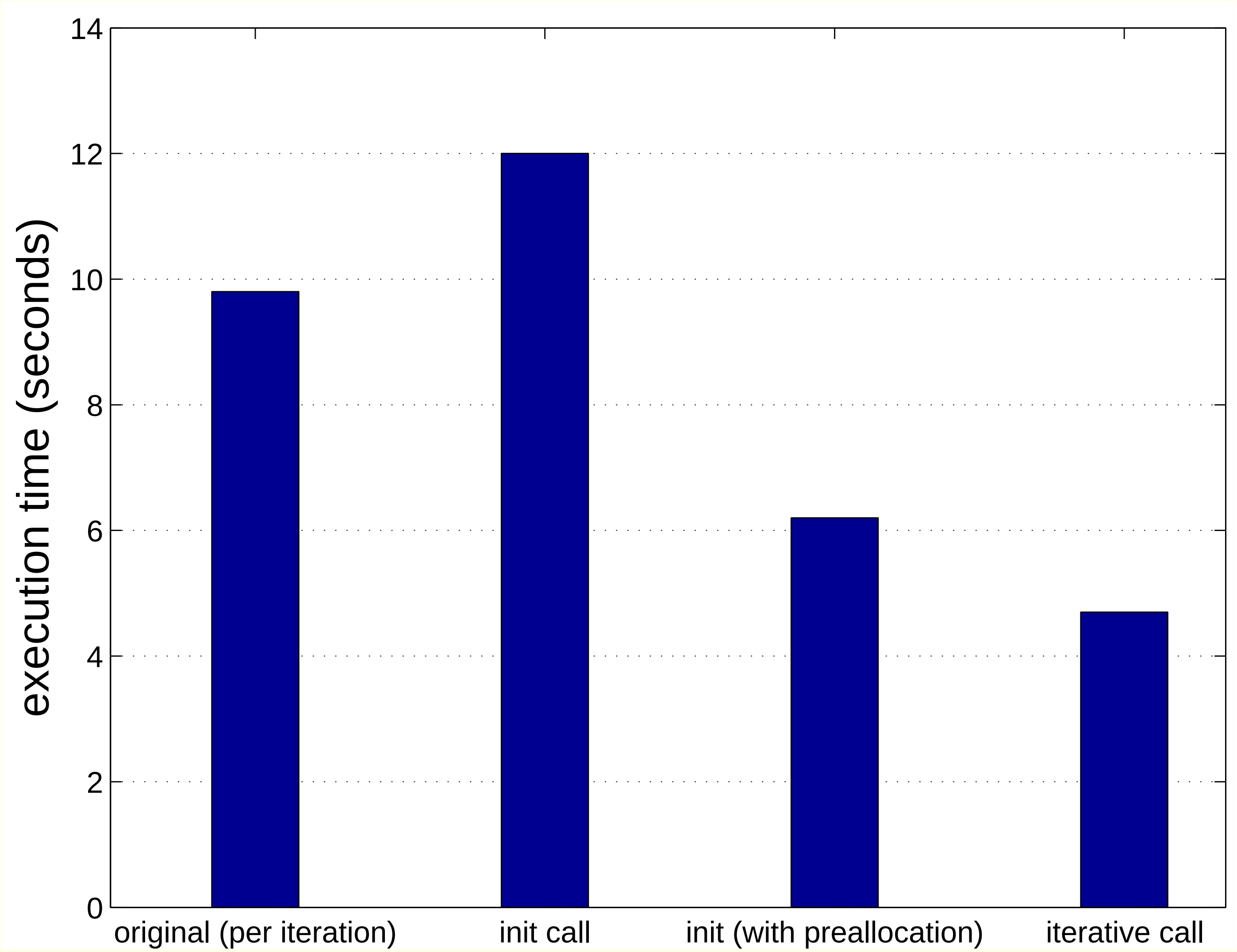
→

```
f_init (c1, c2, c3);
for i = 1:N
    ...
    f_iter (i);
    ...
end
```

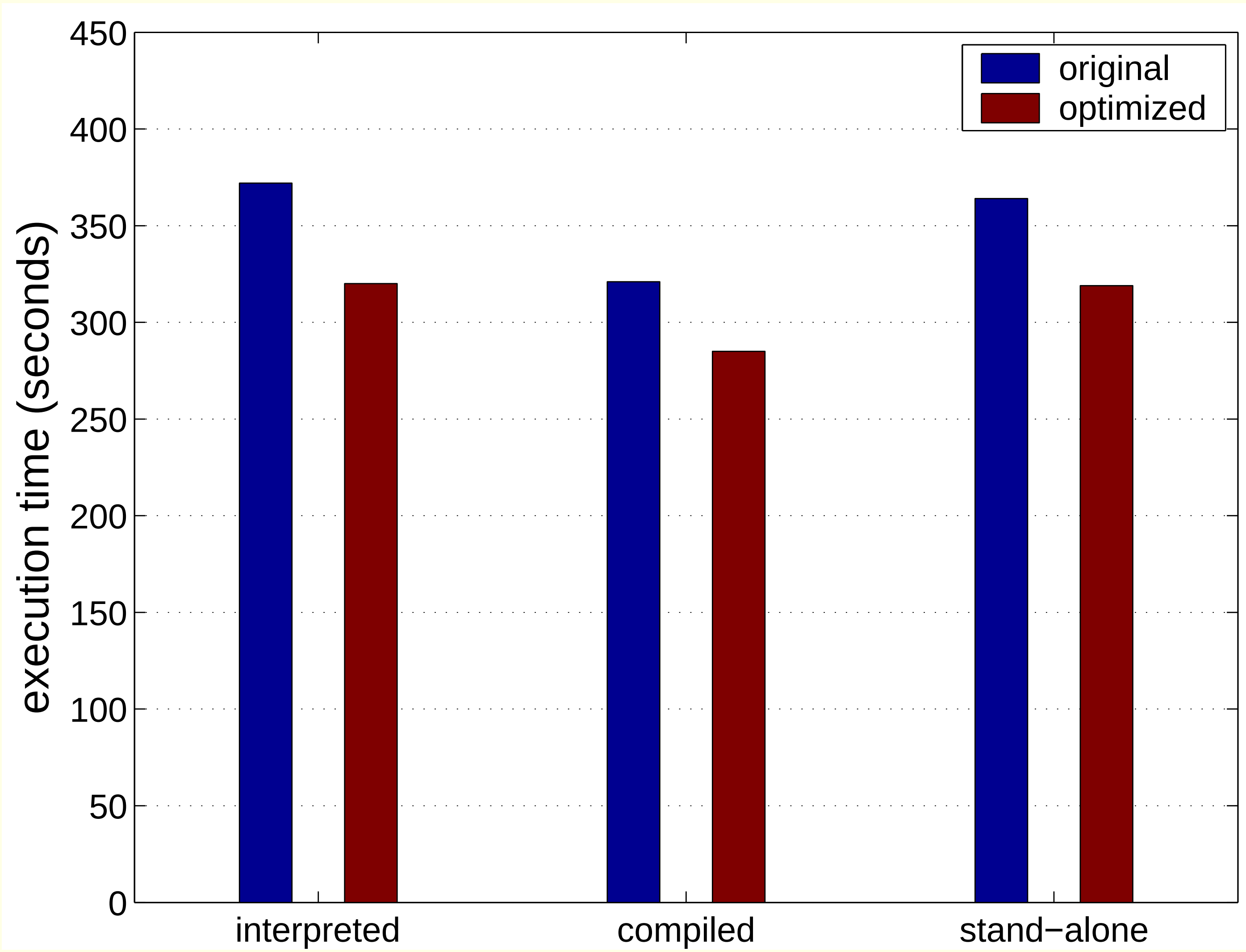
Applying to ctss



Applying to `sML_chan_est`



Effect of `mcc` Compilation



More on Strength Reduction

- procedure strength reduction somewhat different from operator strength reduction
  - could be similar if the `iter` component provided
- automatic differentiation is a more powerful approach that matches procedure strength reduction
  - more work needed to utilize automatic diff. for optimizing loops

Procedure Vectorization

Procedure Vectorization

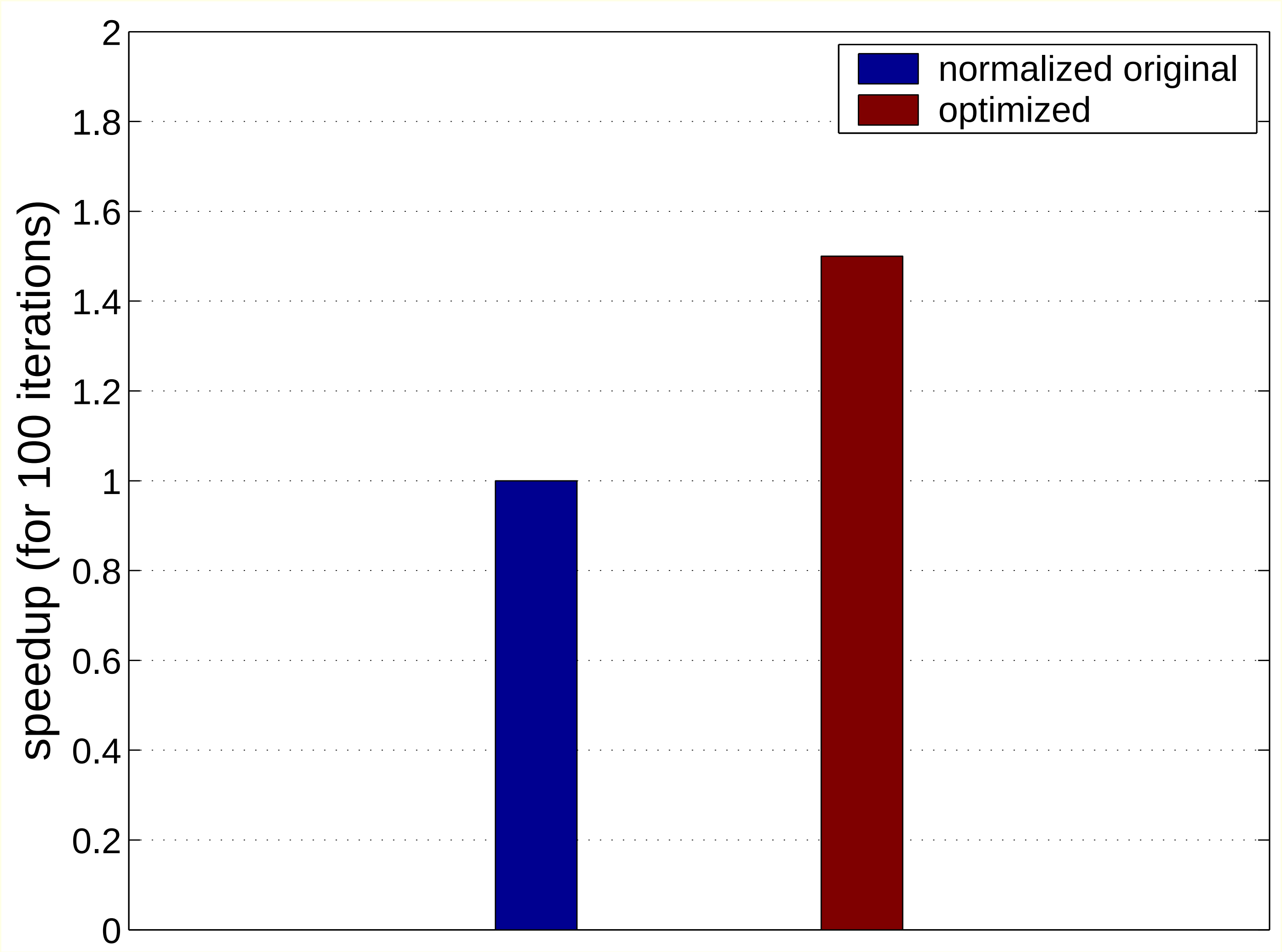
```
for i = 1:N
    f (c₁, c₂, i, A[i]);
end
...
function f (a₁, a₂, a₃, a₄)
    <body of f>
```

Procedure Vectorization

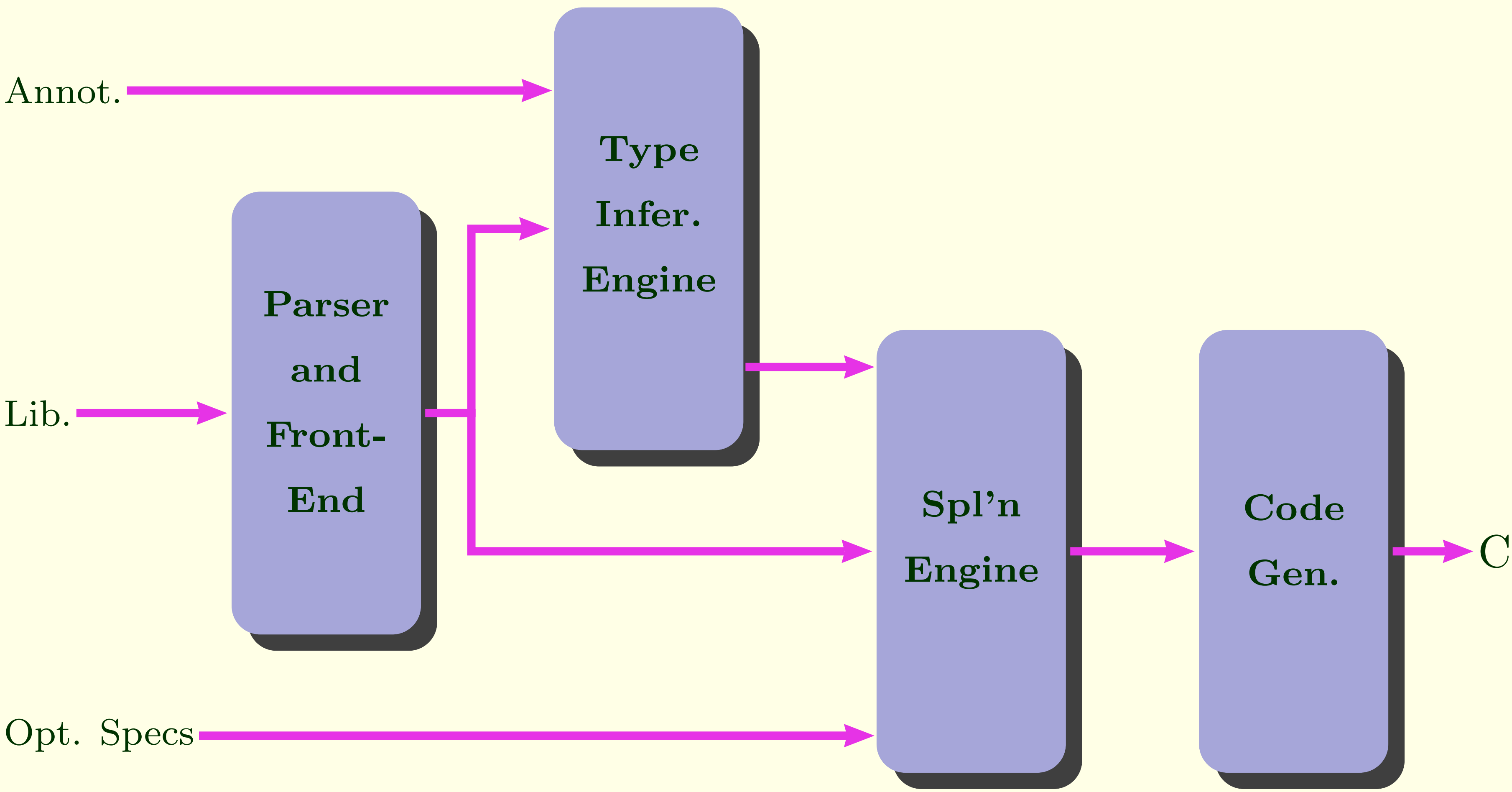
```
for i = 1:N
    f (c1, c2, i, A[i]);
end
...
function f (a1, a2, a3, a4)
    <body of f>
```

```
f_vect (c1, c2, [1:N], A)
...
function f_vect (a1, a2, a3, a4)
    for i = 1:N
        <body of f>
    end
```

Applying to jakes



Overall Architecture



Contributions

Contributions

- validation of the telescoping languages strategy
  - the library compiler component

Contributions

- validation of the telescoping languages strategy
  - the library compiler component
- type-based specialization
  - $\mathcal{NP}$-completeness of type-inference for straight line code
  - a new way to infer types
  - slice-hoisting as a new approach to do dynamic size-inference

Contributions

- validation of the telescoping languages strategy
  - the library compiler component
- type-based specialization
  - $\mathcal{NP}$-completeness of type-inference for straight line code
  - a new way to infer types
  - slice-hoisting as a new approach to do dynamic size-inference
- identification of relevant optimizations

Contributions

- validation of the telescoping languages strategy
  - the library compiler component
- type-based specialization
  - $\mathcal{NP}$-completeness of type-inference for straight line code
  - a new way to infer types
  - slice-hoisting as a new approach to do dynamic size-inference
- identification of relevant optimizations
- discovery of two new optimizations
  - procedure strength reduction and procedure vectorization

Contributions

- validation of the telescoping languages strategy
  - the library compiler component
- type-based specialization
  - $\mathcal{NP}$-completeness of type-inference for straight line code
  - a new way to infer types
  - slice-hoisting as a new approach to do dynamic size-inference
- identification of relevant optimizations
- discovery of two new optimizations
  - procedure strength reduction and procedure vectorization
- infrastructure development
  - a novel compiler architecture

Contributions: Publications

- Arun Chauhan and Ken Kennedy. Procedure strength reduction and procedure vectorization: Optimization strategies for telescoping languages. In *Proceedings of ACM-SIGARCH International Conference on Supercomputing*, June 2001. Also available as Reducing and vectorizing procedures for telescoping languages. *International Journal of Parallel Programming*, 30(4):289–313, August 2002.

- Arun Chauhan, Cheryl McCosh, Ken Kennedy, and Richard Hanson. Automatic type-driven library generation for telescoping languages. *To appear in the Proceedings of SC: High Performance Networking and Computing Conference, 2003.*

- Arun Chauhan and Ken Kennedy. Slice-hoisting for dynamic size-inference in MATLAB. *In writing.*

- Cheryl McCosh, Arun Chauhan, and Ken Kennedy. Computing type jump-functions for MATLAB libraries. *In writing.*

Future Directions

- high-level reasoning
- time-bound compilation and AI techniques
- dynamic compilation and the grid
- automatic parallelization
- automatic differentiation
- other domains