A Source-level MATLAB Transformer for DSP Applications

Arun Chauhan and Ken Kennedy
(work done at Rice University)

Arun Chauhan

Indiana University

Background

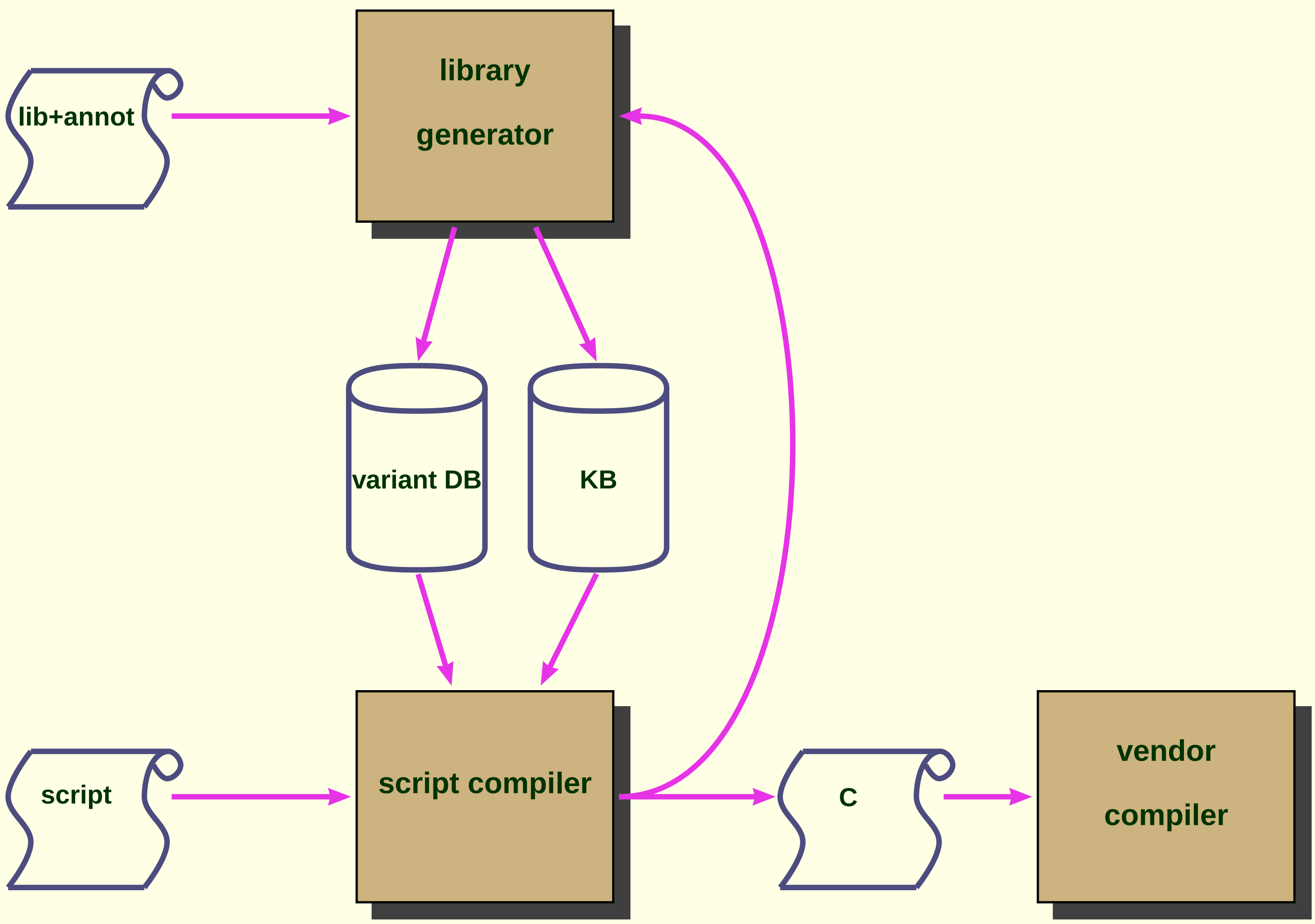
- Problem of programmers' productivity
- Lower productivity affects the progress of science and technology
- Higher-level languages can potentially improve productivity
- Performance problem needs to be solved

Background

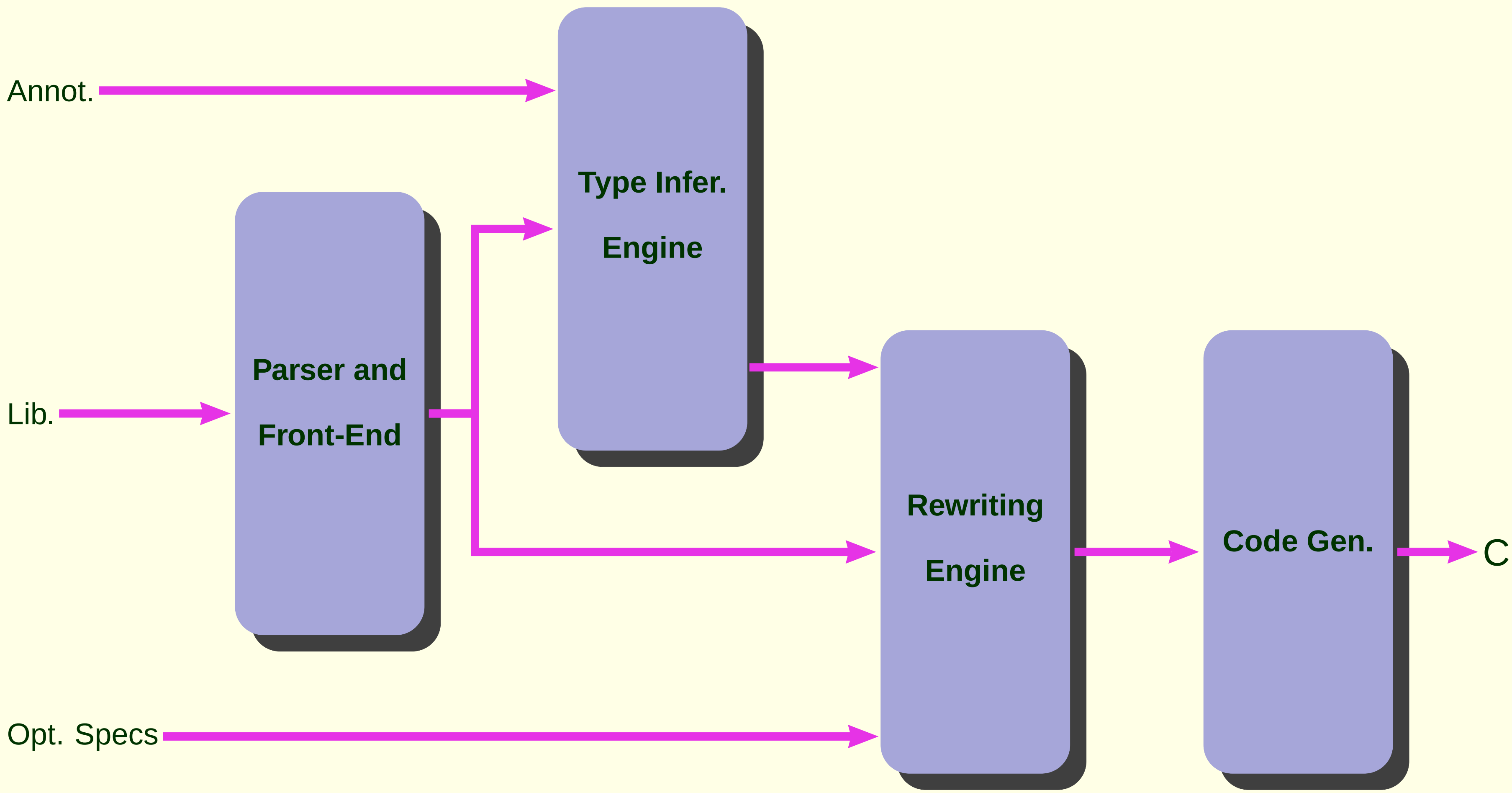
- Problem of programmers' productivity
- Lower productivity affects the progress of science and technology
- Higher-level languages can potentially improve productivity
- Performance problem needs to be solved

Need better compilers!

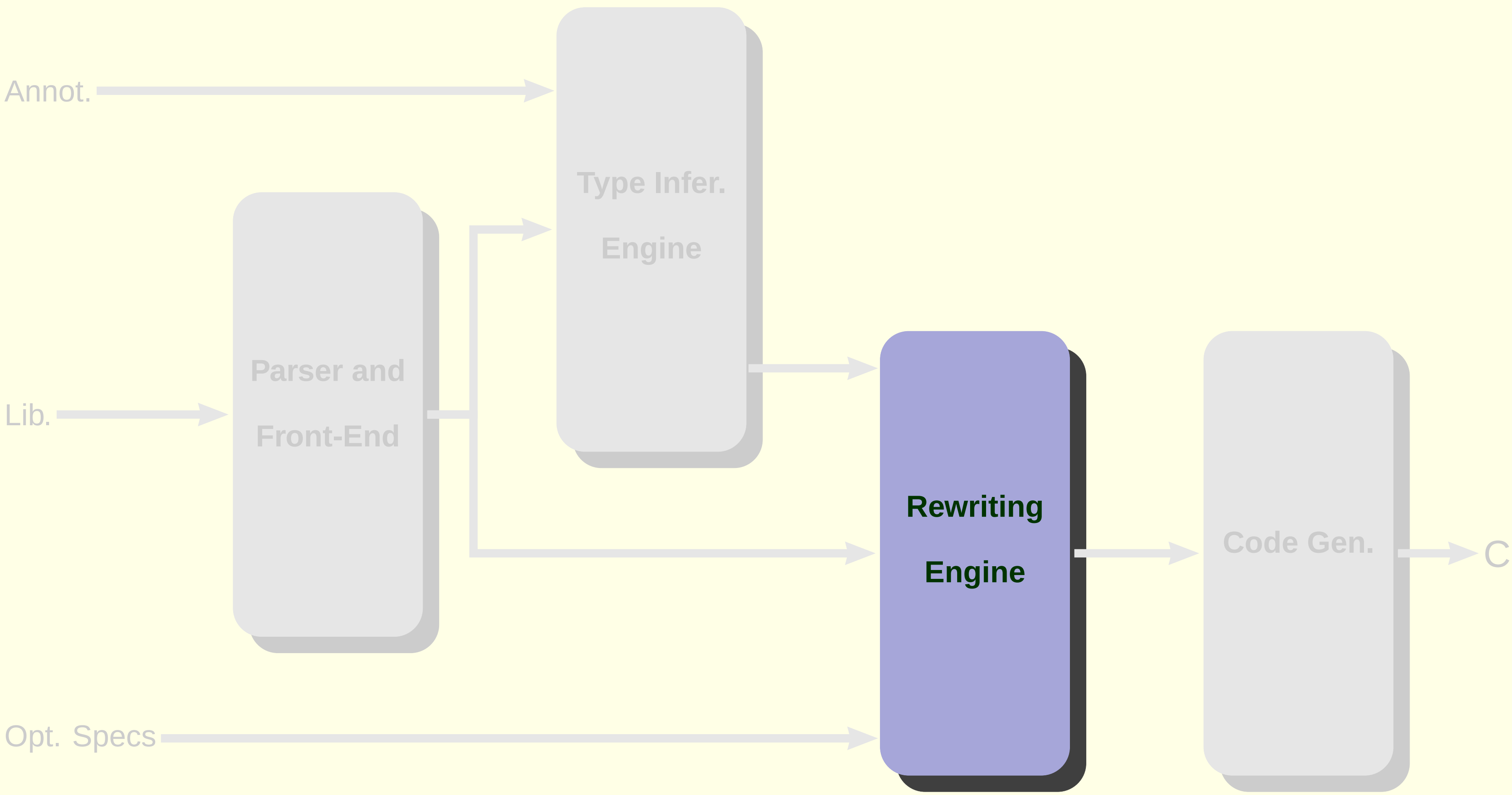
Telescoping Languages



Compiler Components



Compiler Components



Relevant Transformations

"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*

Study of DSP Applications

- MATLAB applications from the ECE department
  - real applications being used in the DSP and image processing group
- Looked for high-level transformations
- Discovered
  - two novel procedure-level transformations
  - relevance of several well known transformation techniques

DSP Applications

- `jakes_mp1`: fast fading signals using the Jakes model
- `codesdhd`: Viterbi decoder
- `newcodesig`: simulates the transmitter and the channel of a wireless system
- `ser_test_fad`: value iteration algorithm for finite horizon and variable power to minimize outage
- `sML_chan_est`: implements a block in a SimuLink system
- `acf`: computes auto-correlation of a signal
- `artificial_queue`: simulates a queue
- `ffth`: computes an FFT on a `real` vector
- `fourier_by_jump`: implements Fourier analysis by the method of jumps
- `huffcode`: computes Huffman codewords based on their lengths

High-payoff Transformations

- Procedure strength reduction
- Procedure vectorization
- Loop vectorization
- Library identities
- Common subexpression elimination
- Beating and dragging along
- Constant propagation

XML-based Language

- Enables library writers to express transformations of interest
- Can specify type-based specializations
- Powerful enough to specify library indentities
- Serves as a driver for the source-level optimization phase

Procedure Strength Reduction

```
for i = 1:N
    ...
    f (a, b, i);
    ...
end
```

Procedure Strength Reduction

```
for i = 1:N
    ...
    f (a, b, i);
    ...
end
```

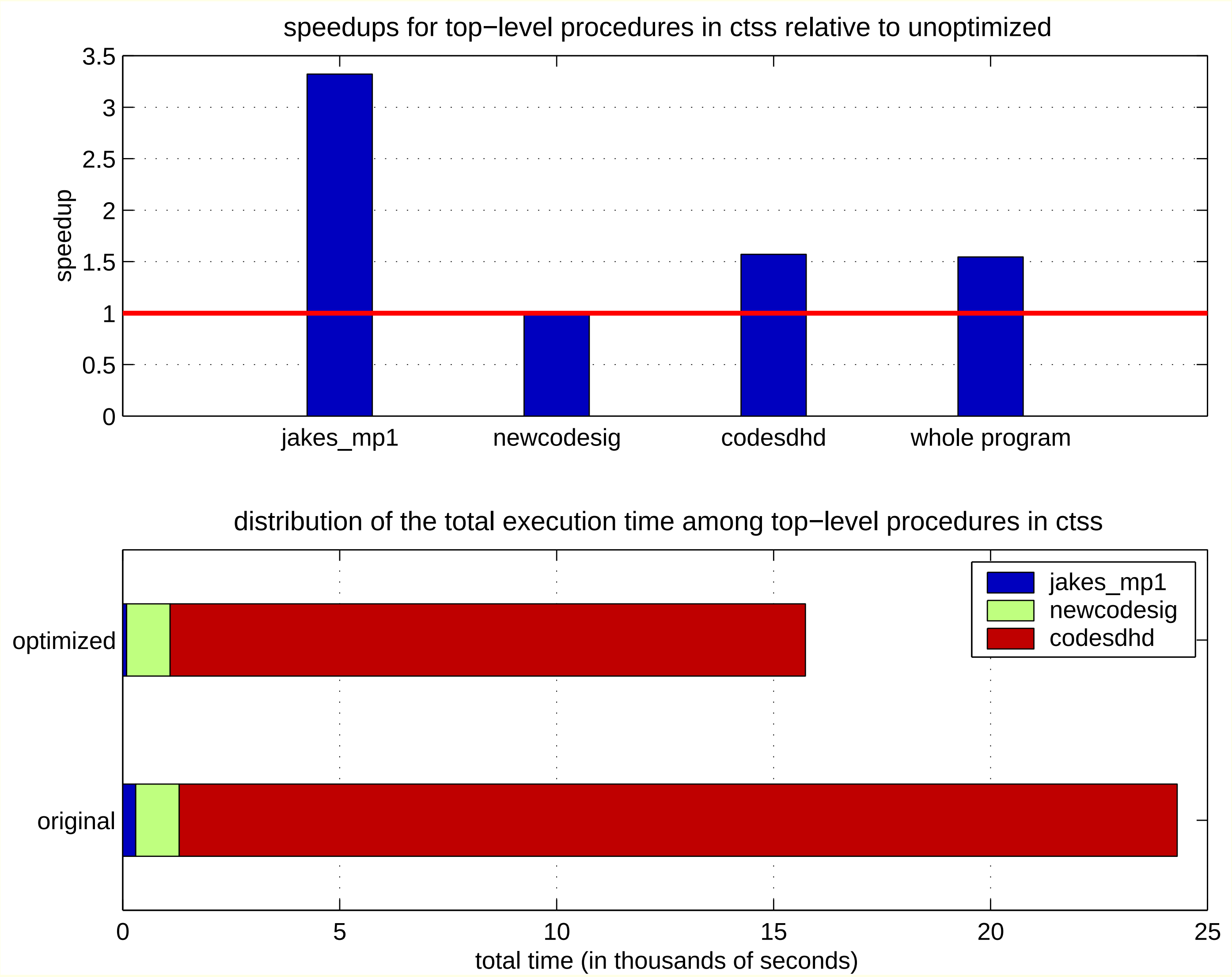
→

```
f_init (a, b);
for i = 1:N
    ...
    f_iter (i);
    ...
end
```

XML Example: Procedure Strength Reduction

```
<specialization>
  <match>
    <forLoopStmt index="i">
      <lower>L</lower> <upper>U</upper> <step>S</step>
      <body>
        <anyStmt label="1" minCount="0" maxCount="unlimited"/>
        <!-- simple statement f(a, b, i) -->
        <anyStmt label="2" minCount="0" maxCount="unlimited"/>
      </body>
    </forLoopStmt>
  </match>
  <substitute>
    <!-- simple statement f_init(a, b) -->
    <forLoopStmt index="i">
      <lower>L</lower> <upper>U</upper> <step>S</step>
      <body>
        <putStmt label="1"/>
        <!-- simple statement f_iter(i) -->
        <putStmt label="2"/>
      </body>
    </forLoopStmt>
  </substitute>
</specialization>
```

Speedup by PSR



Procedure Vectorization

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

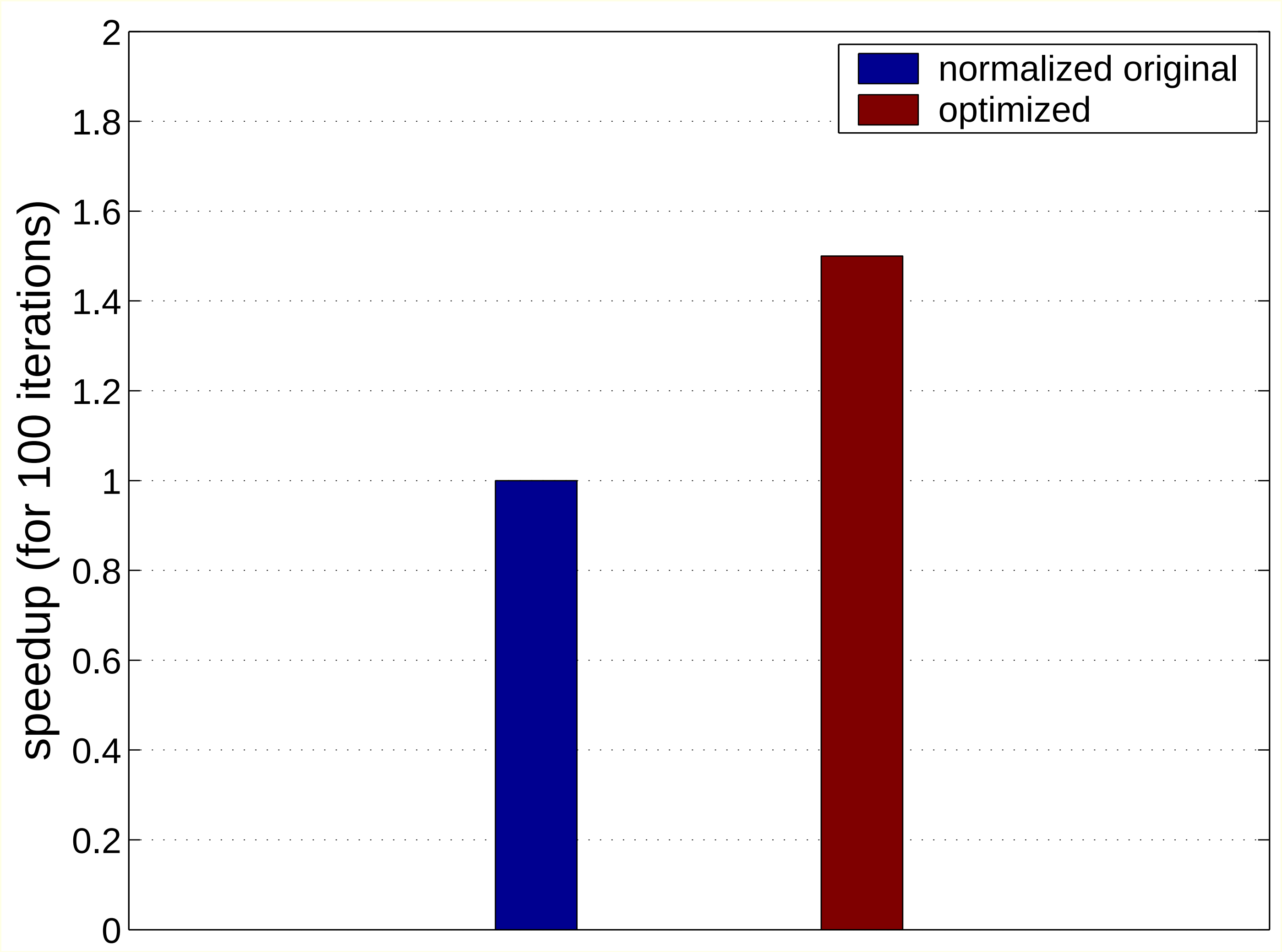
Procedure Vectorization

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

→

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

Applying to `jakes`



Algorithm

input:

rewriting rule, $\mathcal{R} = <C, P, S>$

abstract syntax tree, T

output:

transformed syntax tree, $T^{'}$

uses:

`search_pattern`

`replace_pattern`

`replace_occurrences`

```
procedure rewrite
  return if the context C not verified
  L = list of the top-level statements in T
  pattern_handle = search_pattern(P, T)
  if found
    if S is a substitute then
      if replacing P by S does not violate any dependencies
        T' = replace_pattern(T, pattern_handle, S)
      else
        T' = T;
      endif
    else
      T' = replace_occurrences(T, pattern_handle, S)
    endif
  endif
  // now repeat the process for each statement recursively
  for each compound statement, M, in L
    H = abstract syntax tree for M
    H' = rewrite(R, H)
    T' = T with H replaced by H'
    T = T'
  endfor
  return T'
```

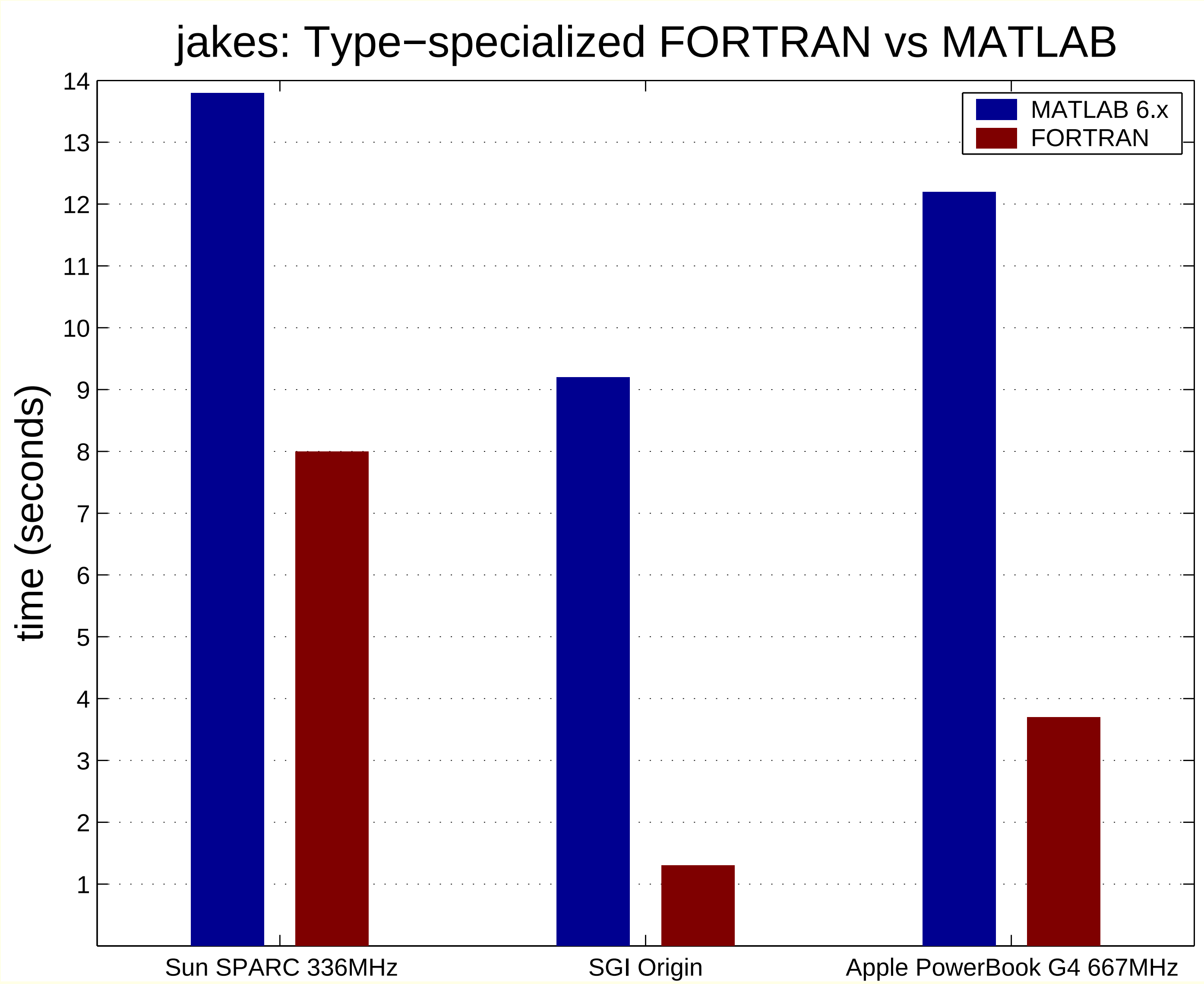
Other Program Transformations

- Type-based specialization
  - order of magnitude performance improvements
- `while`-`for` conversion
- Copy propagation
- Loop-invariant code motion

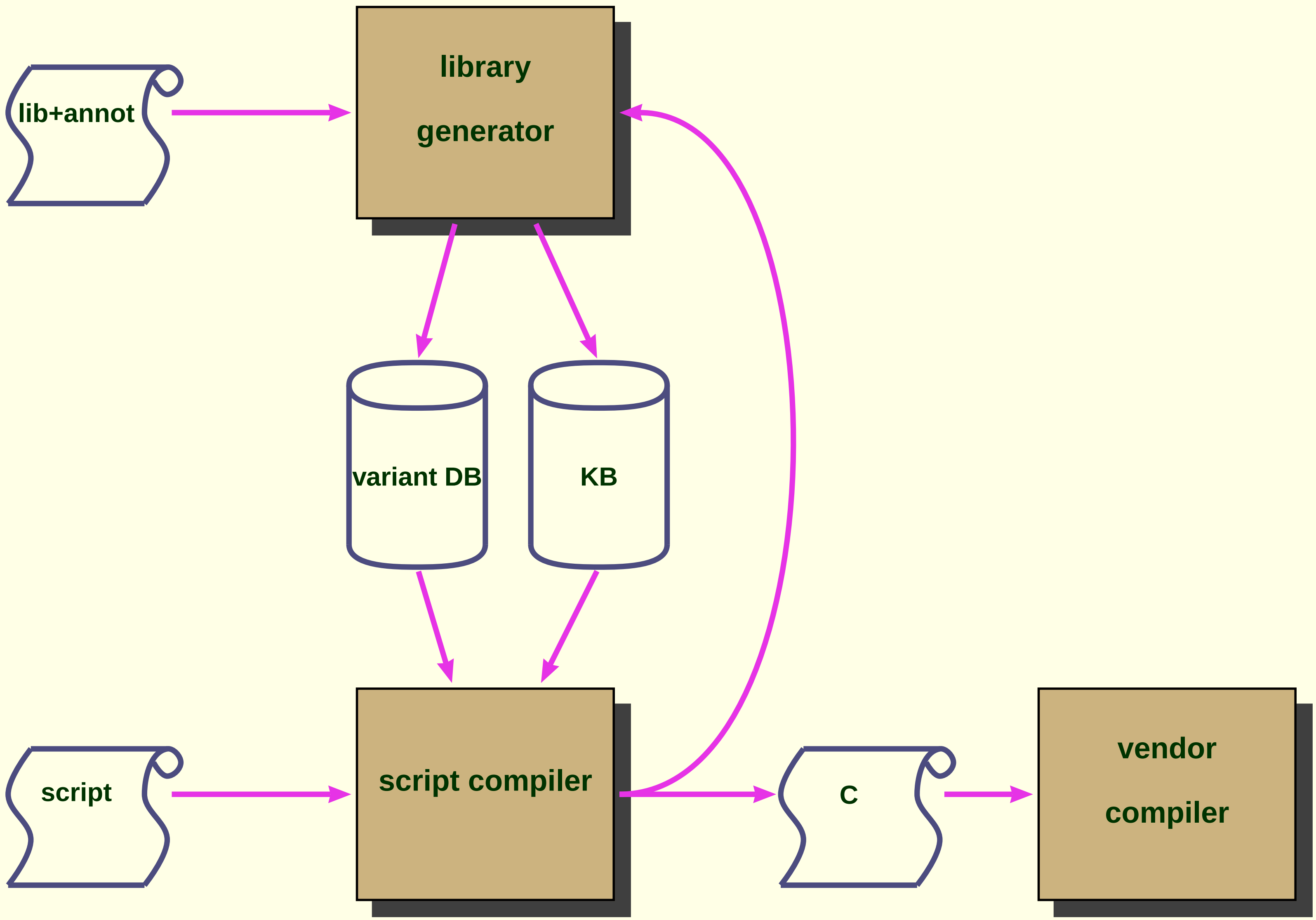
Example: Type-based Specialization

```
<specialization>
  <context>
    <type var="x" dims="0"/>
    <type var="y" dims="0"/>
  </context>
  <match>
    <simpleStmt>
      <function> generic_ADD </function>
      <input> <var>x</var> <var>y</var> </input>
      <output> <var>z</var> </output>
    </simpleStmt>
  </match>
  <substitute>
    <simpleStmt>
      <function> scalar_ADD </function>
      <input> <var>x</var> <var>y</var> </input>
      <output> <var>z</var> </output>
    </simpleStmt>
  </substitute>
</specialization>
```

Speedup by Type Specialization



The MATLAB Compilation System



Conclusion

- MATLAB an important language for DSP researchers
- Performance bottlenecks impede wider application
- Source-level MATLAB transformations can payoff handsomely

`http://www.cs.indiana.edu/~achauhan/`