



Calpa: A Tool for Automating Selective Dynamic Compilation

Markus U. Mock,
Craig Chambers, and Susan J. Eggers

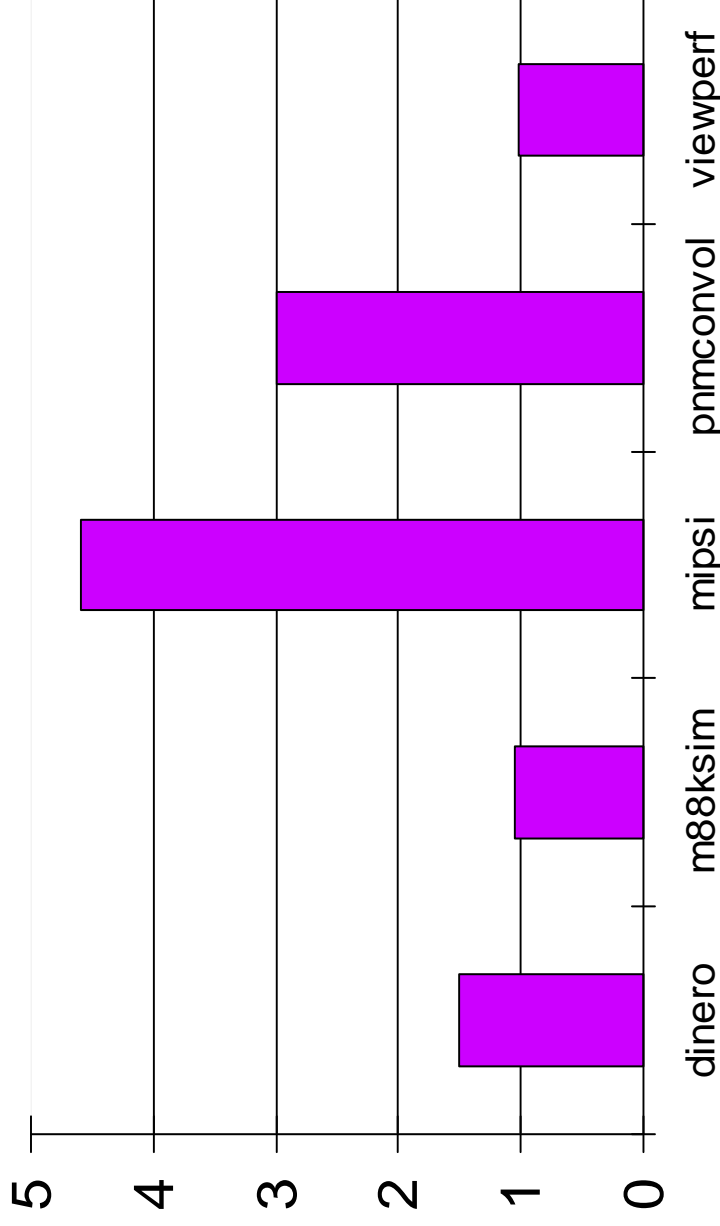
University of Washington
Department of Computer Science and Engineering



Selective Dynamic Compilation

- ◆ **Dynamic**
 - ◆ exploits information available only at run time, e.g. run-time constant variables
 - ◆ run-time compilation cost
- ◆ **Selective**
 - ◆ restrict run-time compilation to profitable program regions and values
 - ◆ other regions are compiled statically (unlike JITs)

DyC Speedups



◆ selectivity & wide range of optimizations \Rightarrow wide applicability with speedups up to 4.6x

DyC's Approach

- ◆ DyC provides an optimization mechanism
 - ◆ programmer annotates *static* variables, regions & selects optimization policies
 - ◆ DyC generates customized dynamic compilers automatically
 - ◆ well-chosen annotations result in speedups
- ◆ Simple annotations
 - ◆ **makestatic(x)**: produce specialized code for x's values

Challenges

- ◆ Speedups depend on
 - ◆ selected regions, variables & policies
 - ◆ architectural details & optimizations
 - ◆ program & input characteristics
- ◆ Manual annotations are hard, requiring
 - ◆ intimate knowledge of the application
 - ◆ predicting the effects of DyC's optimization
- ◆ Practical experience
 - ◆ finding good annotations can take weeks of *human* time

Calpa

- ◆ Tools to automatically produce good DYC annotations
 - ◆ compile-time analyses to identify promising variables & program regions
 - ◆ program profiling to select variables & regions
- ◆ Better or equal to manual annotations, typically in minutes, not weeks

Talk Outline

- ◆ DyC overview
- ◆ Calpa
 - ◆ overview
 - ◆ annotation selector
 - ◆ cost-benefit model
 - ◆ instrumentation tool
 - ◆ example
- ◆ Experimental results
- ◆ Conclusions & future work

DyC System - Key Ideas

- ◆ Replace repeated computations by their result

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

DyC System - Key Ideas

- ◆ Replace repeated computations by their

result

$$z = 2 * 3$$

$$z = 2 * 3$$

DyC System - Key Ideas

- ◆ Replace repeated computations by their

result

$$z = 2 * 3$$

$$z = 2 * 3$$

$$z = 6$$

DyC System - Key Ideas

- ◆ Replace repeated computations by their

result

$$z = 2 * y$$

$$z = 2 * 3$$

$$z = 6$$

$$x = a[i]$$

DyC System - Key Ideas

- ◆ Replace repeated computations by their

result

$$z = 2 * 3$$

$$z = a[1]$$

$$z = 2 * 3$$

$$\mathbf{x} = \mathbf{a}[2]$$

$$\mathbf{z} = \mathbf{6}$$

DyC System - Key Ideas

- ◆ Replace repeated computations by their

result

$$z = 2 * y$$

$$z = a[1]$$

$$z = 2 * 3$$

$$z = a[2]$$

$$z = 6$$

$$z = 42$$

DyC System - Key Ideas

- ◆ Replace repeated computations by their result
 - ◆ instruction with invariant sources
 - ◆ load from invariant data structure

DyC System - Key Ideas

- ◆ Replace repeated computations by their result
 - ◆ instruction with invariant sources
 - ◆ load from invariant data structure
 - ◆ full loop unrolling

```
for (i=0;i<size; i++)  
    sum += a[i];
```

DyC System - Key Ideas

◆ Replace repeated computations by their result

- ◆ instruction with invariant sources
- ◆ load from invariant data structure
- ◆ full loop unrolling

```
for (i=0; i<size; i++)  
    sum += a[i];
```

```
for (i=0; i<3; i++)  
    sum += a[i];
```

DyC System - Key Ideas

- ◆ Replace repeated computations by their result
 - ◆ instruction with invariant sources
 - ◆ load from invariant data structure
 - ◆ full loop unrolling

```
for ( i=0; i<size; i++)  
    sum += a[i];
```

```
for ( i=0; i<3; i++)  
    sum += a[i];
```

```
sum += a[0]  
sum += a[1];  
sum += a[2];
```

DyC System - Key Ideas

- ◆ Replace repeated computations by their result
 - ◆ instruction with invariant sources
 - ◆ load from invariant data structure
 - ◆ full loop unrolling

```
for ( i=0; i<size; i++)  
    sum += a[i];
```

```
for ( i=0; i<3; i++)  
    sum += a[i];
```

```
sum += a[0] 12;  
sum += a[1] 13;  
sum += a[2] 0;
```

DyC System - Code Caching

- ◆ Cache & reuse specialized code

DyC System - Code Caching

- ◆ Cache & reuse specialized code
 - ◆ respecialize when values change

DyC System - Code Caching

- ◆ Cache & reuse specialized code
- ◆ respecialize when values change
 - ◆ lookup before code use:

```
z=x*y  
print z
```

DyC System - Code Caching

- ◆ Cache & reuse specialized code
- ◆ respecialize when values change
 - ◆ lookup before code use:

```
z=2222y  
print z  
  
print 6
```

DyC System - Code Caching

- ◆ Cache & reuse specialized code
- ◆ respecialize when values change
 - ◆ lookup before code use:

```
z=x*x*y  
print z  
if <x,y> != <2,3> goto dyc  
print 6
```

DyC System - Code Caching

- ◆ Cache & reuse specialized code
 - ◆ respecialize when values change
 - ◆ lookup before code use
 - ◆ code cache invalidation when value changes

DyC System - Code Caching

- ◆ Cache & reuse specialized code
 - ◆ respecialize when values change
 - ◆ lookup before code use
 - ◆ cache invalidation when value changes

```
x=a[i]  
print x
```

DyC System - Code Caching

- ◆ Cache & reuse specialized code
 - ◆ respecialize when values change
 - ◆ lookup before code use
 - ◆ cache invalidation when value changes

```
eval[1]  
print *
```

```
print 42
```

DyC System - Code Caching

- ◆ Cache & reuse specialized code
- ◆ respecialize when values change
 - ◆ lookup before code use
 - ◆ cache invalidation when value changes

```
==a[i]  
print *
```

a[2] = 21

print 42

DyC System - Code Caching

- ◆ Cache & reuse specialized code
- ◆ respecialize when values change
 - ◆ lookup before code use
 - ◆ cache invalidation when value changes

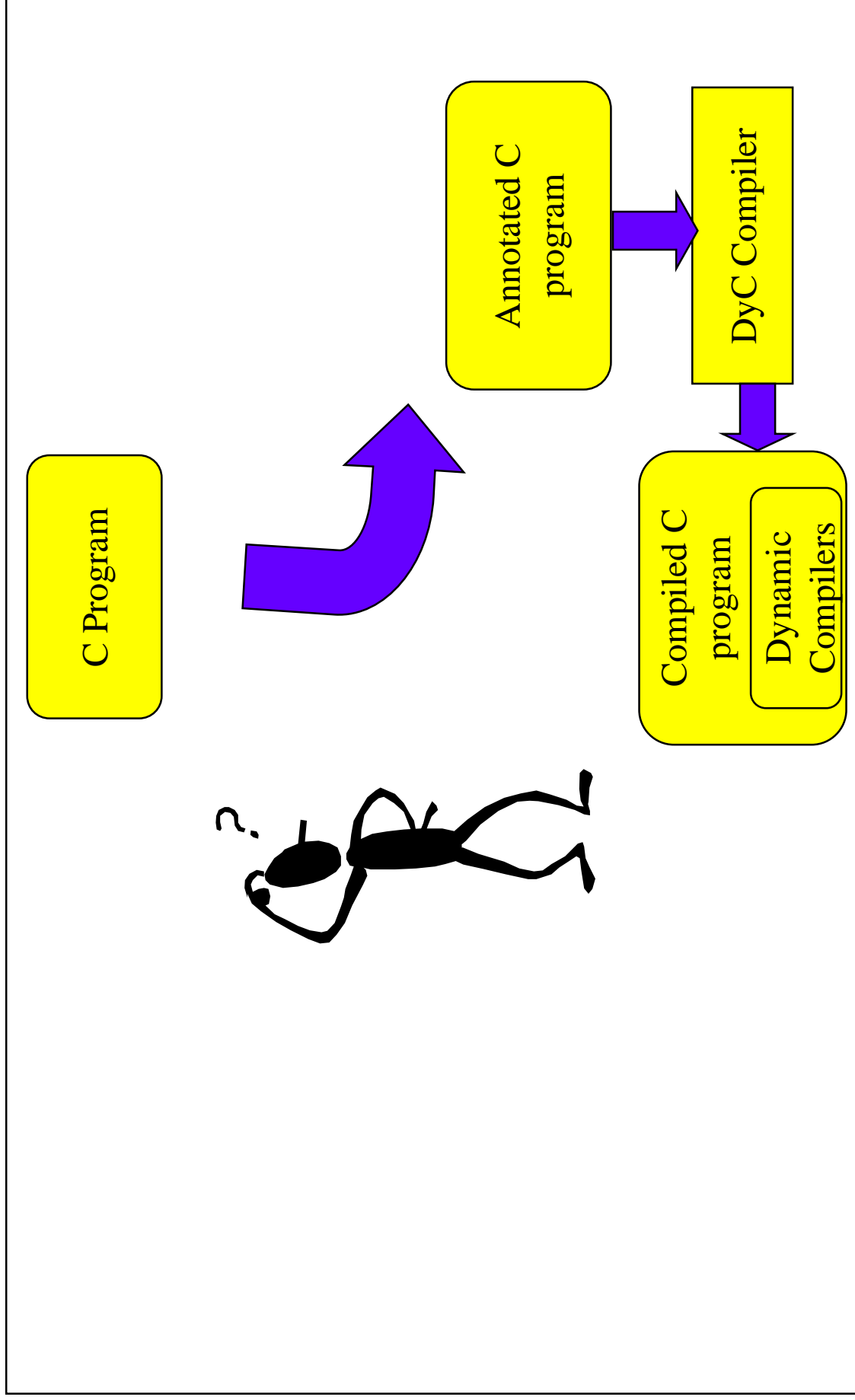
```
==a[i]
print *
if !valid goto dyc
print 42

a[2] = 21
valid= false
```

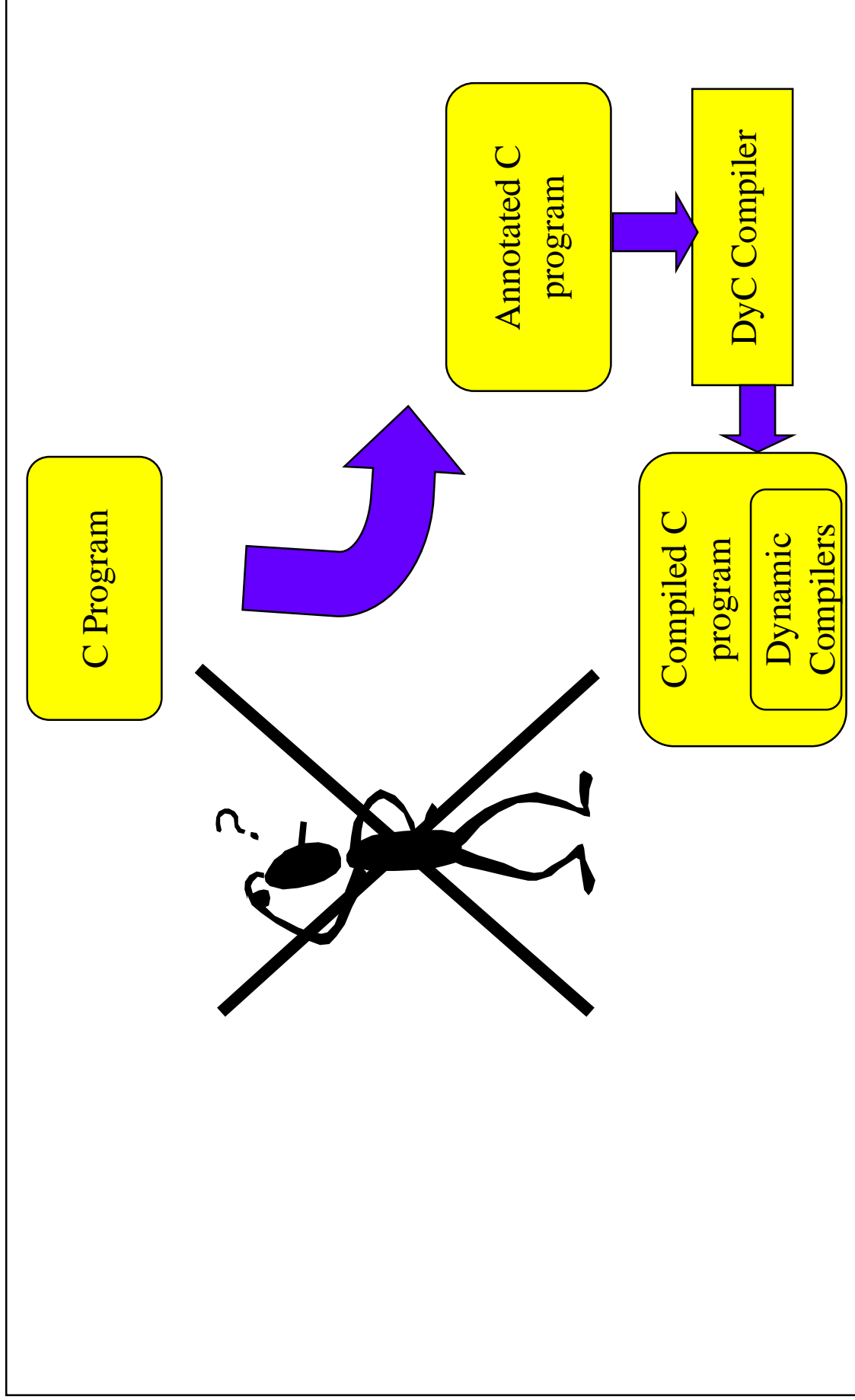
DyC Summary

- ◆ DyC provides a mechanism
 - ◆ specialize code for specific values
 - ◆ cache specialized code for reuse
 - ◆ key lookup-based
 - ◆ invalidation-based
- ◆ Mechanism is driven by user annotations
- ◆ Annotations control where, what and how to specialize & and cache code

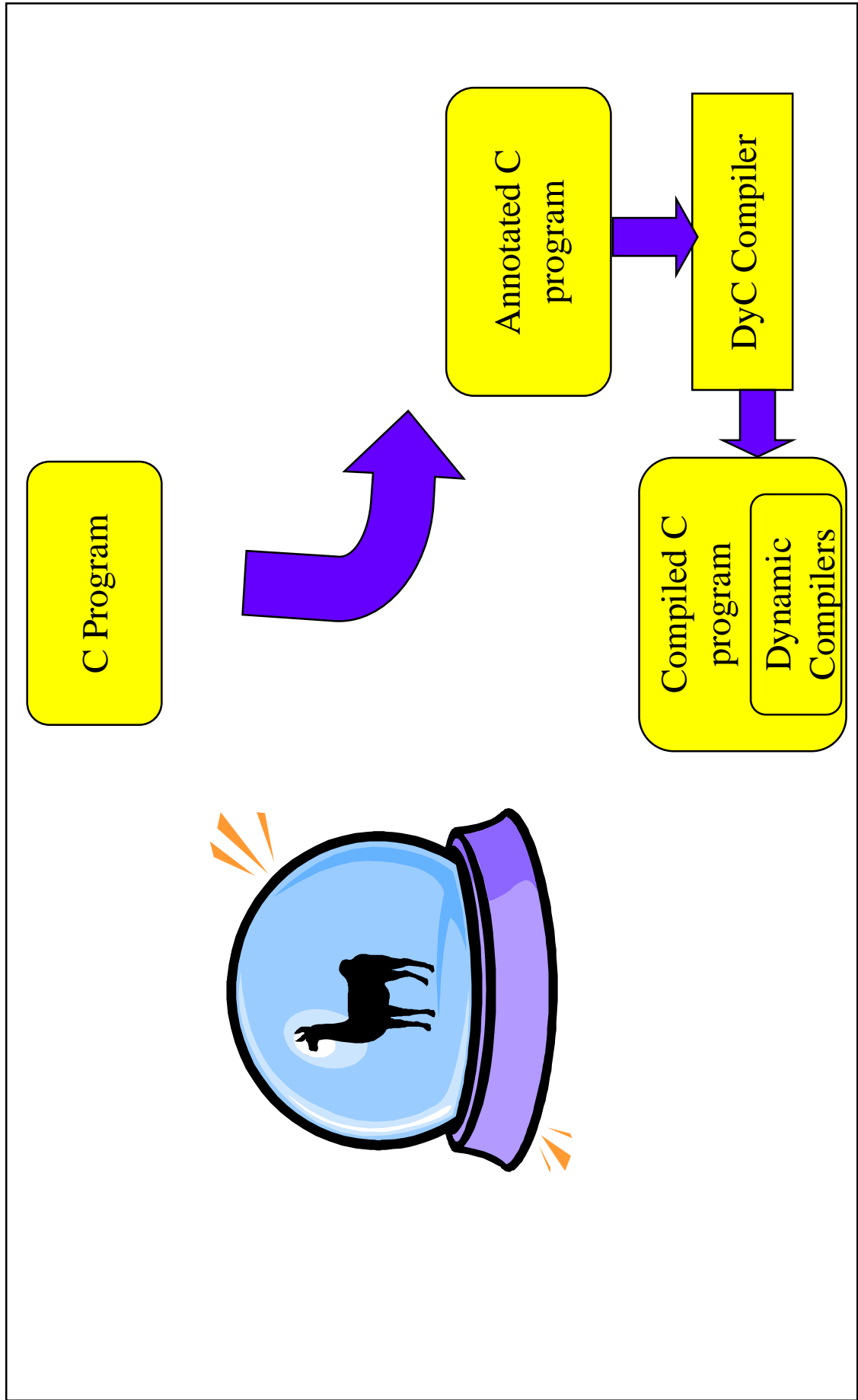
DyC Summary



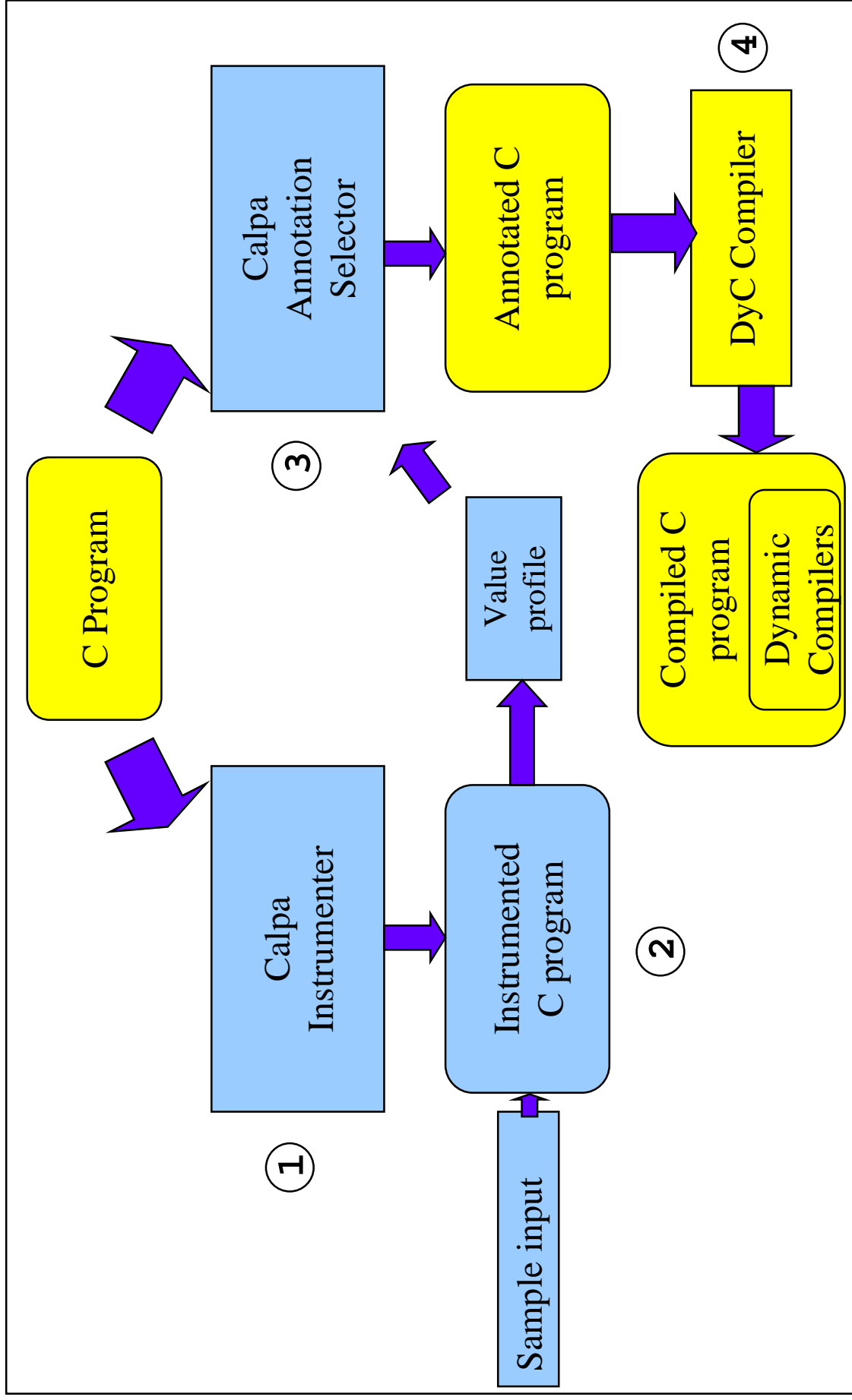
DyC Summary



Calpa



Calpa Overview:



Calpa's Annotation Selector

- ◆ Selects best annotation candidates:
 - ◆ compute initial *Candidate Static Variables*
 - ◆ derived from program's computations
 - ◆ combine sets
 - ◆ enlarges specialization benefit
 - ◆ evaluate choices with cost-benefit model
 - ◆ retains best choice
 - ◆ terminate combinations when
 - ◆ possibilities exhausted or improvement diminishes

Calpa's Cost Model

- ◆ Calpa models three kinds of dynamic compilation costs:
 - ◆ Specialization cost
 - ◆ paid once for particular set of values
 - ◆ Dispatching cost
 - ◆ paid periodically for each key lookup
 - ◆ Invalidation check cost
 - ◆ paid periodically for variables & data structures with that caching policy

Calpa's Benefit Model

- ◆ Main benefit:
 - ◆ static instructions are executed only once (at specialization time)
- ◆ Compute benefit by
 - ◆ compute static instructions from CSV set
 - ◆ ignore instructions not on the critical execution path
 - ◆ multiply cycles saved by execution frequency

Calpa's Instrumenter

- ◆ Provides data for the cost-benefit model:
 - ◆ basic block execution frequency
 - ◆ values of variables / data structures
 - ◆ tracks data accessed through pointers
 - ◆ alias analysis relates run-time addresses to source variables & data structures
 - ◆ frequency of changes

Calpa Example

- ◆ Determine static variables for each instruction
- ◆ Combine sets to larger sets making multiple instructions static
- ◆ Use cost-benefit model to evaluate a CSV

Calpa Example

```
void* lookup(data_t data[], int size, int key)
    for (int i=0; i<size; i++)
        if (data[i].key == key) return data[i].fun;
    return NULL;
}
```

Calpa Example

```
Lookup:
    i=0
L0:   if i >= size goto L1:
      t1 = &data[i]
      t2 = t1->key
      if t2 == key goto L2;
      i = i+1
      goto L0
L1:   return NULL
L2:   return t1->fun
```

Calpa Example - CSV sets

```
Lookup:
    i=0
L0:   if i >= size goto L1:
      t1 = &data[i]
      t2 = t1->key
      if t2 == key goto L2;
      i = i+1
      goto L0
L1:   return NULL
L2:   return t1->fun
    }
```

Calpa Example - CSV sets

```
Lookup:
    i=0
L0:    if i >= size goto L1:
        t1 = &data[i]
        t2 = t1->key
        if t2 == key goto L2;
        i = i+1
        goto L0
L1:    return NULL
L2:    return t1->fun
    }
```

Calpa Example - CSV sets

```
Lookup:
    i=0
L0:   if i >= size goto L1:
      t1 = &data[i]
      t2 = t1->key
      if t2 == key goto L2;
      i = i+1
      goto L0
L1:   return NULL
L2:   return t1->fun
      {}
      {i,size}
      {data[],i}
      {data[],i}
```

Calpa Example - CSV sets

```
Lookup:
    i=0
L0:   if i >= size goto L1:
      t1 = &data[i]
      t2 = t1->key
      if t2 == key goto L2;
      i = i+1
      goto L0
L1:   return NULL
L2:   return t1->fun
      {}
      {i, size}
      {data[], i}
      {data[], i}
      {data[], i, key}
```

Calpa Example - CSV sets

```
Lookup:
    i=0
L0:   if i >= size goto L1:
      t1 = &data[i]
      t2 = t1->key
      if t2 == key goto L2;
      i = i+1
      goto L0
L1:   return NULL
L2:   return t1->fun

      {}
      {i,size}
      {data[],i}
      {data[],i}
      {data[],i,key}
      {i}
      {}
      {}
      {data[],i}

{{{},{i,size},{data[],i},{data[],i,key},{i}}
```

Calpa Example - CSV sets

```
Lookup:
    i=0
L0:   if i >= size goto L1:
      t1 = &data[i]
      t2 = t1->key
      if t2 == key goto L2;
      i = i+1
      goto L0
L1:   return NULL
L2:   return t1->fun
      {data[],i,size}
```

Calpa Example - CSV sets

```
Lookup:
    i=0
L0:   if i >= size goto L1:
      t1 = &data[i]
      t2 = t1->key
      if t2 == key goto L2;
      i = i+1
      goto L0
L1:   return NULL
L2:   return t1->fun
      {data[],i,size}
```

Calpa Example - Cost

Lookup:

```
if 85 == key goto L2.0
...
if 66 == key goto L2.19
L1:   return NULL
L2.0: return fun1
L2.1: return fun2
...
L2.19: return fun20
```

Profile
Info:
size = 20

Specialization Cost: 41 * 100 cycles

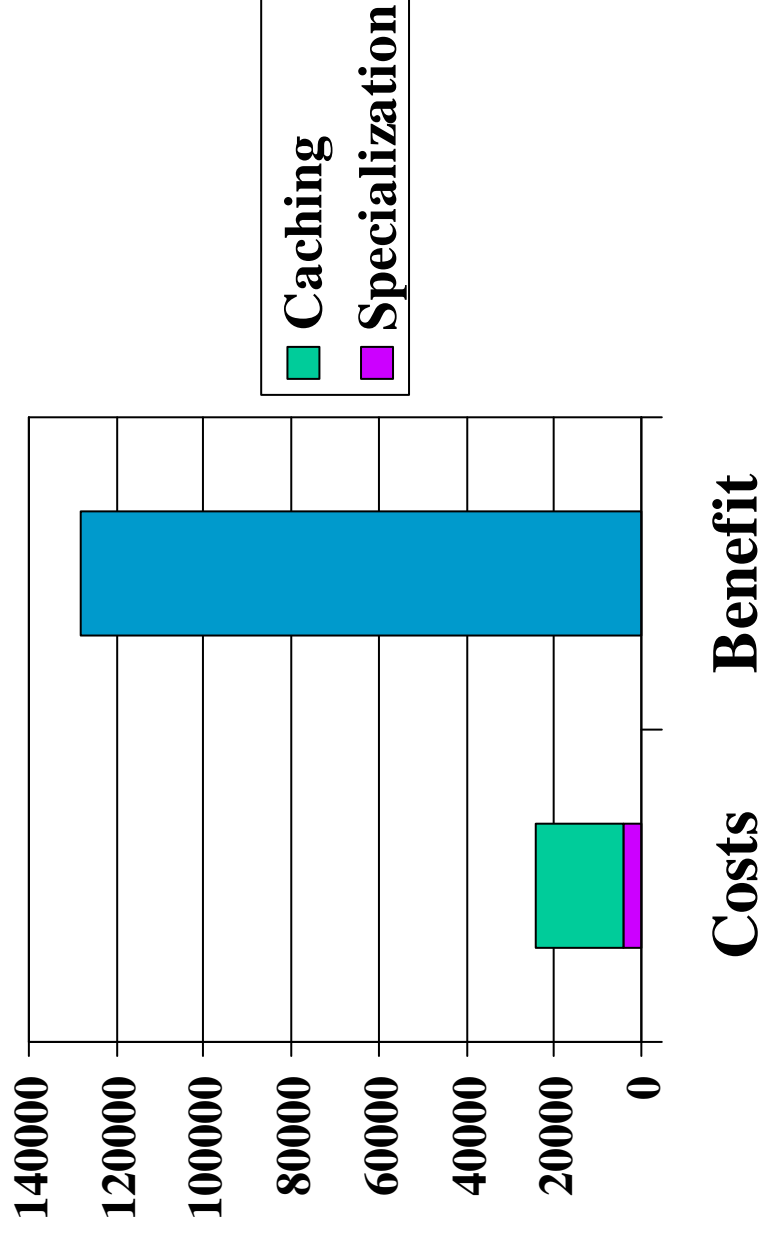
Caching Cost: 10 cycles per call

Calpa Example - Benefit

```
Lookup:
  i=0
L0:  if i >= size goto L1:      11 * 1 cycles
     t1 = &data[i]           10 * 1 cycles
     t2 = t1->key             10 * 2 cycles
     if t2 == key goto L2;    0
     i = i+1                  10 * 1 cycle
     goto L0                  10 * 1 cycle
L1:  return NULL              0
L2:  return t1->fn            1 * 2 cycles
```

Total: 2,000 * 64 cycles = 128,000 cycles

Calpa Example



Calpa Annotation Results

- ◆ Experimental Questions:
 - ◆ Annotation quality
 - ◆ Annotation time

Calpa Annotation Results

Program	Size (lines)	Instrumentation Time	Profiling Time	Annotation Time	Speedup
binary	111	0.2s	1.9s	6s	1.8
dotproduct	136	0.1s	0.3s	2s	5.7
query	226	0.4s	7.8s	15s	1.4
romberg	134	0.3s	0.4s	26s	1.3
pnmconvol	333	1.2s	17.1min.	75s	3.0
dinero	2,397	4.6s	13.8min.	27min.	1.5
m88ksim	11,549	10.7min.	3.5hours	8.0hours	1.05

Conclusion & Future Work

- ◆ Calpa produces good annotations in manageable time
 - ◆ minutes or hours of *machine time* not weeks of *human time*
- ◆ Explore design space
 - ◆ varying cutoff parameters etc.
- ◆ Study sensitivity of results to different profiling inputs