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 ⇒ 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
DyC System - Key Ideas

- Replace repeated computations by their result
  
  \[ z = x \times y \]
DyC System - Key Ideas

- Replace repeated computations by their result
  \[ z = x^y \]
  \[ z = 2^3 \]
DyC System - Key Ideas

- Replace repeated computations by their result

\[ z = x \times y \]
\[ z = 2 \times 3 \]
\[ z = 6 \]
Replace repeated computations by their result

\[ z = x \times y \]
\[ z = 2 \times 3 \]
\[ z = 6 \]
DyC System - Key Ideas

- Replace repeated computations by their result

\[
\begin{align*}
z &= x \times y \\
x &= a[i] \\
z &= 2 \times 3 \\
x &= a[2] \\
z &= 6
\end{align*}
\]
DyC System - Key Ideas

- Replace repeated computations by their result

\[
\begin{align*}
z &= x \cdot y \\
z &= 2 \cdot 3 \\
z &= 6
\end{align*}
\]

\[
\begin{align*}
x &= a[1] \\
x &= a[2] \\
x &= 42
\end{align*}
\]
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

```c
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

```c
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

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

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

```c
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

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

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

```c
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:
  
  \[ z = x \times y \]
  
  print z
Cache & reuse specialized code

- resspecialize when values change
  - lookup before code use:

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

```
print 6
```
DyC System - Code Caching

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

```cpp
z=x*y
print z
if <x,y> != <2,3> goto dyc
print 6
```
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
  - resspecialize when values change
    - lookup before code use
    - cache invalidation when value changes

```python
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

```python
x = a[i]
print x
print 42
```
DyC System - Code Caching

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

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

print 42

a[2] = 21
```
DyC System - Code Caching

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

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

```plaintext
x = a[i]
print x
if !valid goto dyc
print 42
```
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 & cache code
DyC Summary

C Program

Annotated C program

Compiled C program

DyC Compiler

Dynamic Compilers
DyC Summary

C Program

Annotated C program

Compiled C program

Dynamic Compilers

DyC Compiler

Annotated C program
Calpa Overview:

1. Sample input
2. Instrument C program
3. Calpa Annotation Selector
4. DyC Compiler

- C Program
- Value profile
- Compiled C program
- Dynamic Compilers
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
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;
}
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:**

```c
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: {i,size}
    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] {data[],i}  
    t2 = t1->key {data[],i}  

    if t2 == key goto L2;  

    i + i+1  
    goto L0  

L1:  return NULL  

L2:  return t1->fun
Lookup:

\[ i = 0 \] {i, size}

\[ \text{L0: if } i \geq \text{ size goto L1:} \] {i, size}
\[ t1 = &data[i] \] {data[], i}
\[ t2 = t1->key \] {data[], i}
\[ \text{if } t2 == \text{ key goto L2;} \] {data[], i, key}
\[ i = i + 1 \]
\[ \text{goto L0} \]

\[ \text{L1: return NULL} \]

\[ \text{L2: return } t1->\text{fun} \]
Calpa Example - CSV sets

Lookup:

\[
\begin{align*}
    i &= 0 & \{\} \\
    \text{L0: if } i &\geq \text{ size goto L1:} & \{i, \text{size}\} \\
    t1 &= \&data[i] & \{\text{data[]}, i\} \\
    t2 &= t1-& gt;\text{key} & \{\text{data[]}, i\} \\
    \text{if } t2 &= \text{ key goto L2;} & \{\text{data[]}, i, \text{key}\} \\
    i &= i+1 & \{i\} \\
    \text{goto L0} & & \{\} \\
    \text{L1: return NULL} & & \{\} \\
    \text{L2: return } t1-& gt;\text{fun} & \{\text{data[]}, i\} \\
\end{align*}
\]

\[
\{\}, \{i, \text{size}\}, \{\text{data[]}, i\}, \{\text{data[]}, i, \text{key}\}, \{i\}\]
Calpa Example - CSV sets

Lookup:

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

```c
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:
    t1 = &data[i]
    t2 = t1->key
    if t2 == key goto L2;
    i = i+1
    goto L0
L1: return NULL
L2: return t1->fn

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

Costs Benefit

Caching Specialization

140000 120000 100000 80000 60000 40000 20000 0
Calpa Annotation Results

Experimental Questions:
- Annotation quality
- Annotation time
## Calpa Annotation Results

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