Design and Implementation Of Annotations for Dynamic Compilation

Markus U. Mock

Department of Computer Science & Engineering
University of Washington

Overview

- What is Dynamic Compilation?
- Approaches to Dynamic Compilation
- Our annotations
- Implementation
- Assessment

Dynamic Compilation

- Delay compilation of (some) code until run time
- Exploit constants that become known at run time (static values)
- Value-specific optimizations
  - Constant propagation
  - Constant folding
  - Constant branch folding & dead code elimination
  - Strength reduction
  - Full loop unrolling $\rightarrow$ induction variable constant

Specialize code for static values

Example Applications

Common property: slowly-changing data structure

Examples:

- Interpreters
- Graphics codes
- Numerical applications
- Systems code (e.g., in SPIN)

Approaches

How to know where to do dynamic compilation?

- Imperative — 'C
  - Programmer writes code that manipulates code fragments
  - Tedious, error-prone, hard to debug
- Fully Automatic
  - Compiler and runtime system decide
  - Difficult — open research problem
- Annotation-Based — Tempo, Fabius
  - Annotations direct compilation
  - Simpler than fully-imperative
  - Debugging support easy

Our Design Goals

Annotations should be:

- Concise
- Declarative

- Expressive
- Fundamental and orthogonal

- Controlling trade-offs
- Safe

Annotations

Two Basic Annotations:

- `makeStatic(x,y);`
- `makeDynamic(x,y);`

`makeStatic` and `makeDynamic` statements within a procedure

Policies:

- Degree of specialization (Specialization Policies)
- Cache of specialized code (Cache Policies)
- Time of specialization (Laziness Policies)

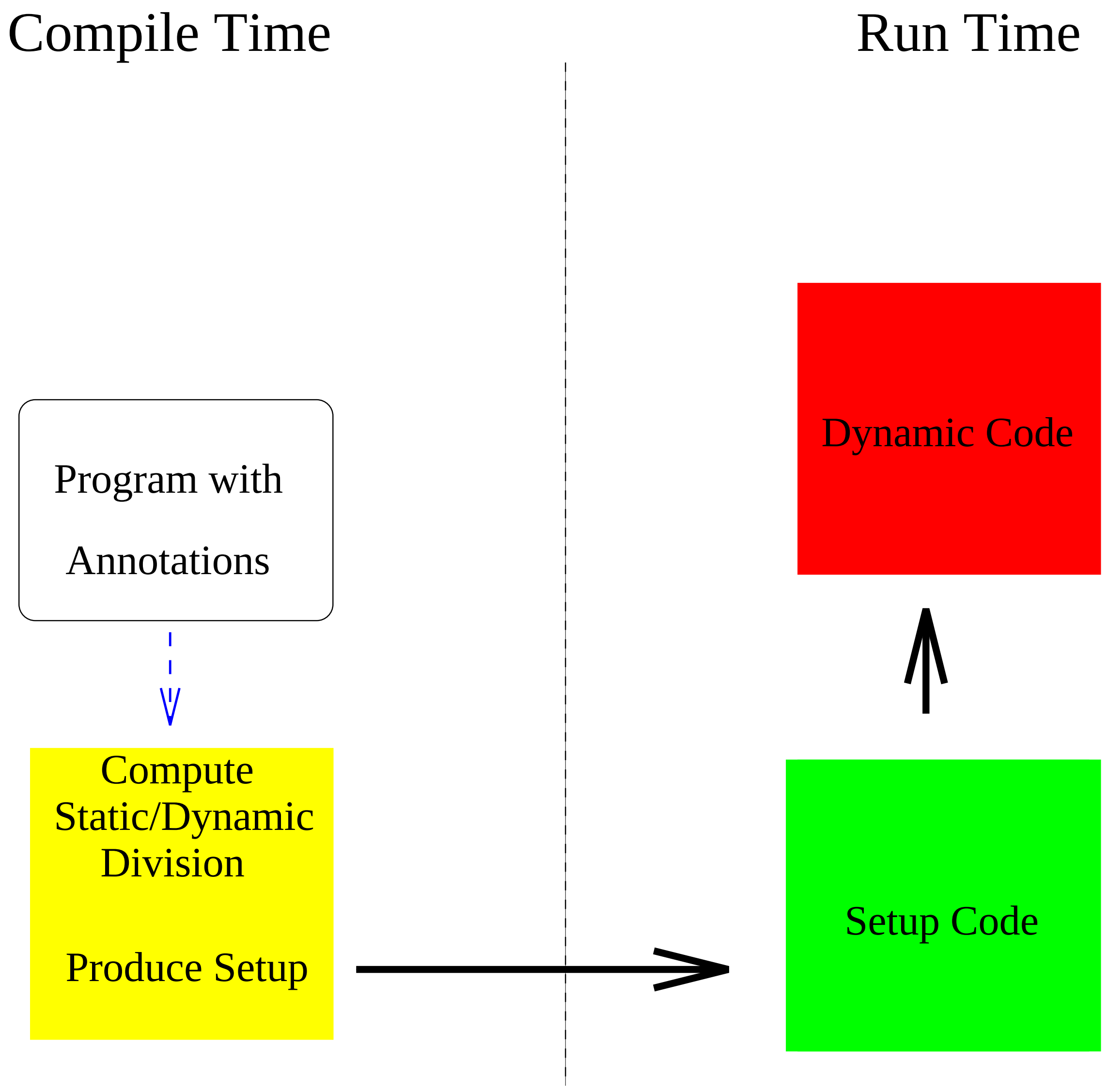
Dynamic Compilation Model

At compile time:

- compute what to specialize, what is static / dynamic
- produce *setup code* that, when run, generates dynamic code

At run time:

- run set-up code, given computed static values (once)
- execute specialized code (many times)


Compile Time
Run Time
Program with
Annotations
Compute
Static/Dynamic
Division
Produce Setup
Dynamic Code
Setup Code

Example 1

```
C Code                 Bytecodes

int add(x,y) {         ADD
   return x+y;         RET
}
```

Dynamically Generated Code

```
valueStack[0] = valueStack[0] + valueStack[1];
return valueStack[0];
```

```
int interpret(byteCodeT byteCodes[], int numArgs, int arglist[]) {

   for (;;) {
      opcode =        byteCodes[pc];
      switch (opcode) {
         case ADD:
            valueStack[vsp-1] = valueStack[vsp-1] + valueStack[vsp];
            vsp--; pc++;
            break;









         case RET:
            return valueStack[vsp];
         ...
      }
   }
}
```

```
makeStatic(byteCodes);
makeStatic(pc);
```

`static`

Example 2

C Code	Bytecodes

```
C Code              Bytecodes

if (x >= 0)         GTE L1
   return 1;        PUSH -1
else                RET
   return -1;       L1:  PUSH 1
                    RET
```

Dynamically Generated Code

```
if (valueStack[0] >= 0) goto L1;
valueStack[0] = -1;
return valueStack[0];
L1:  valueStack[0] = 1;
return valueStack[0];
```

```
case GTE:
   if (valueStack[vsp--] >= 0)
      pc = static byteCodes[pc+1];
   else
      pc +=2;
   break;
```

Example 3

```
C Code                  Bytecodes

for (i=n;i>=0;i--)  L0:  GTE L1
   printf("%d\n",i);
                        RET
                        L1:  WRITE
                        PUSH 1
                        SUB
                        GOTO L0
```

Dynamically Generated Code

```
L0:  if (valueStack[0]>0) goto L1;

return valueStack[0]
L1:  printf("%d\n",valueStack[0]);
valueStack[1] = 1;
valueStack[0] = valueStack[0] - valueStack[1];
goto L0;
```

```
case GOTO:
   pc = static byteCodes[pc+1];
   break;
```

Policies

Specialized code versions per variable:

- *monovariant* — one value
- *polyvariant* — multiple values
- *promoteStatic* — multiple values and *dynamic-to-static promotion*

Cached versions:

- *cache1* — one version
- *cacheAll* — as many as produced

When are dynamic branches specialized:

- *eager* — both at once
- *lazy* — when taken

$$makeStatic(x : \left\{ \begin{array}{l} \text{monovariant} \\ \text{polyvariant} \\ \text{promoteStatic} \end{array} \right\}, \left\{ \begin{array}{l} \text{cache1} \\ \text{cacheAll} \end{array} \right\}, \left\{ \begin{array}{l} \text{eager} \\ \text{lazy} \end{array} \right\})$$

```
...                 /* uses of x */
if (d)              /* eager or lazy?  */
   x = x + 1;
else
   x = x + 2;

M1:                 /* specialize on x or not?  If yes,
                       cache check
                       keep one or all versions?  */
...                 /* uses of x */

M2:  x = read();  /* read from input
y = x + 42;            x still static?
                       if yes, cache check
                       keep one or all versions?  */
...
```

Function Annotations

Function Specialization:

- `specialize`
- `specialize inline`
- `constant`

Advantages:

- No inter-procedural analysis required
- Support separate compilation

Example

```
/* this is in the header file:  */
specialize constant int prod(int n, double v[], double* m[]) on
   ((n,m), (v,m),(n,v,m));


/* this in the C-code file:  */
double sum(int v[], int w[], double* m[]) {
   double s = 0, t;

   makeStatic(i);
   for (i=1; i<MAX; i++)
      s += prod(i,w,m);

   makeStatic(w,m);
   t = prod(-1,v,m);
   s += t;

   return s;
}
```

Implementation

Two IDFAs run in conjunction:

- Propagation of annotations
- Binding Time Analysis

Analyses are formally specified in quals report

Assessment

Our annotations are good because they are

- Concise
- Declarative
- Expressive
- Allow control of trade-offs
- Implementable

Future Work

More automation:

- Alias and derivation analysis
- Profiling support tool

Better optimizations:

- IR language for better optimizations