Dynamic Compilation and Systems Software

Craig Chambers
Susan Eggers
Joel Auslander
Matthai Philipose
Markus Mock
Przemyslaw Pardyak

Department of Computer Science & Engineering
University of Washington

Dynamic Compilation

Idea:

- some “things” become known only at run-time
- the compiler could generate better code knowing those “things”

$\Rightarrow$ defer some compilation till run-time

“Things” can be:

- values of (scalar) variables
- (parts of) contents of arrays, data structures
- which path is the common-case path
- ...

Challenges of Dynamic Compilation

Key trade-off:

benefit (higher-quality code) of must outweigh **cost** (run-time compile time)

Challenges:

- how to
  - produce high-quality code at run-time?
  - reduce dynamic compile time?
- how to identify where to apply dynamic compilation?
- how to exploit “non-traditional” optimization opportunities?
  - e.g. in systems code

- how to get it used in practice?

Outline

Overview of our approach

Design space for dynamic compilation

An example application: SPIN event dispatching

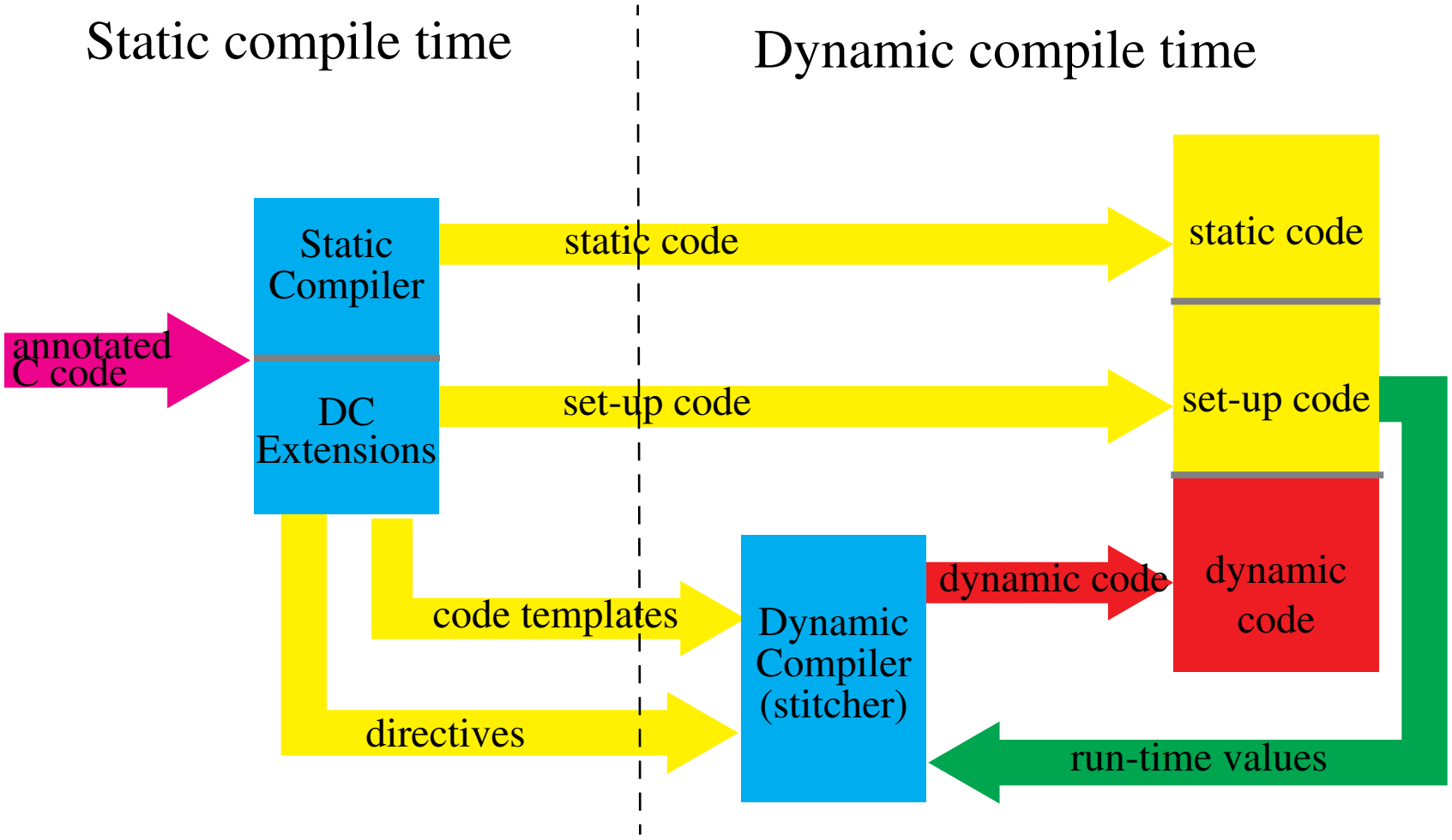
Implementation

- user annotations
- static identification of run-time constants
- optimization of machine code templates
- dynamic code generation

Initial experience

Future directions

Our Framework

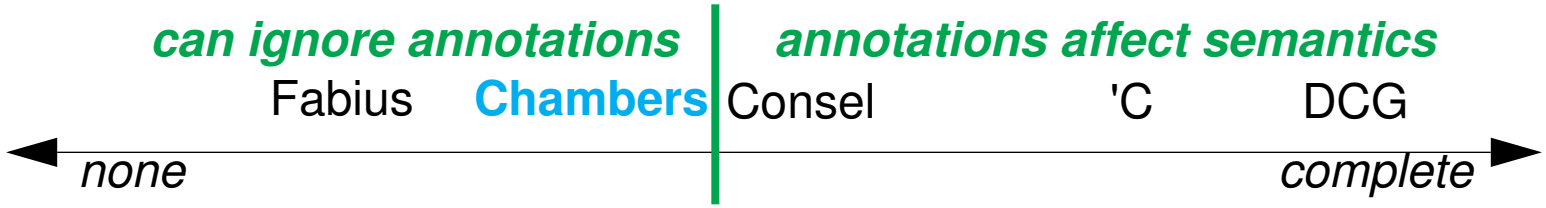


Design Space

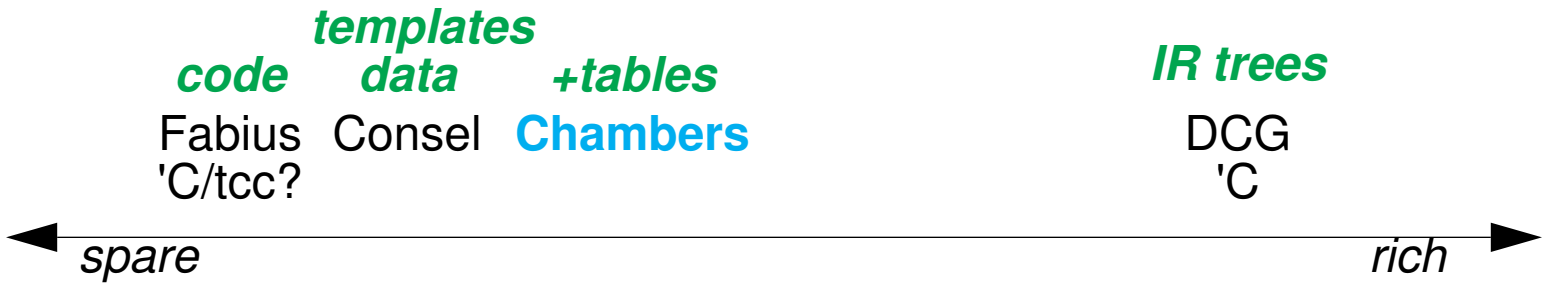
language model

imperative		*functional*
unstructured	*structured*	
Chambers *et al.*	Consel *et al.*	Fabius
DCG?	'C	

user intervention



run-time program representation



Design Space, Part 2

run-time info exploited

scalars	*values +structures*	*arbitrary*
Fabius?	**Chambers** Consel	DCG 'C

limited ◄——————————————————————► *rich*

scope of dynamic compilation

intraprocedural		*interprocedural*
Fabius 'C	**Chambers** DCG	Consel

coping with changes to run-time "constants"

user-specified	*caching*
Consel DCG 'C	**Chambers** Fabius

Example Application: SPIN Event Dispatching

Intermodule communication by raising events

- call procedure ⇒ raise event
- can add handler proc to procedure, with associated guard proc, invoked when procedure invoked and guard returns true

```
raise_event(handlers, args) {
  foreach handler in handlers {
    if (*handler.guard)(args) then (*handler.body)(args)
  }
}
```

Optimizations:

- if unroll `foreach` loop, then can make direct calls to guard and body procs
- then can inline (short) guard procs
- then can factor partially redundant guard subexprs, making decision tree

Problems

Handler list is constructed at run-time

⇒ apply dynamic compilation to `raise_event` proc, treating handlers linked list as run-time constant

Handler list changes at run-time

⇒ flush dynamically-compiled code when "constant" changes, recompile on next event

Need to inline guard procs to get real benefits

⇒ interprocedural scope of dynamic compilation?

Guards and handler procs installed at run-time using dynamic linking

⇒ run-time inlining??

Want to factor redundant guard subexpressions into decision tree

Our Approach

Introduce a run-time representation (e.g. S-exprs) for guard expressions

- compile guards into this representation (by hand, for now)
- include an interpreter for this representation in the SPIN kernel
- “inlining” at run-time is now easy

Write specialized algorithm to factor S-exprs into decision tree

- splitting [Chambers & Ungar, PLDI 90],
  eliminating conditional branches [Mueller & Whalley, PLDI 95]
- by hand, for now

Apply dynamic compilation to interpreter

- convert S-exprs into linear “bytecodes,” to fit into intraprocedural model
- treat bytecode array as a run-time “constant”

Annotated Interpreter

```
struct bytecode {
  enum { CONST, VAR, ADD, LT, BR, ..., DONE } opcode;
  union { VALUE val; int var; int target; ... };
};
raise_event(struct bytecode bytecodes[], args) {
  VALUE stk[N];
  dynamic_region(bytecodes) {
    int pc = 0, sp = 0;
    unrolled for (;;) {
      bytecode i = bytecodes[pc++];
      switch (i.opcode) {
        case CONST: stk[sp++] = i.val; break;
        case VAR: stk[sp++] = args[i.var]; break;
        case ADD: stk[sp-1] = stk[sp] + stk[sp-1]; sp--; break;
        case BR: if (stk[sp--]) pc = i.target; break;
        ...
        case DONE: return;
      }
    }
  }
}
```

Run-Time Constants Identification

Given set of run-time constants at entry to the region,
identify run-time constant expressions in region

Contents of run-time constant arrays & data structures assumed constant,
unless marked otherwise

Perform reachability analysis to handle arbitrary control flow graphs well

Unrolled loops allow induction variables to be run-time constants

- need **multi-way** & **cyclic** loop unrolling, to handle `pc` induction vars w/ gotos

After Run-Time Constants Identification

```
struct bytecode {
  enum { CONST, VAR, ADD, LT, BR, ..., DONE } opcode;
  union { VALUE val; int var; int target; ... };
};
raise_event(struct bytecode bytecodes[], args) {
  VALUE stk[N];
  dynamic_region(bytecodes) {
    int pc = 0, sp = 0;
    unrolled for (;;) {
      bytecode i = bytecodes[pc++];
      switch (i.opcode) {
        case CONST: stk[sp++] = i.val; break;
        case VAR: stk[sp++] = args[i.var]; break;
        case ADD: stk[sp-1] = stk[sp] + stk[sp-1]; sp--; break;
        case BR: if (stk[sp--]) pc = i.target; break;
        ...
        case DONE: return;
      }
    }
  }
}
```

Set-Up Code and Templates

Divide dynamic region into run-time constant operations and control flow
vs. (partially) dynamic operations and control flow

- constant code ⇒ **set-up code**
- dynamic code ⇒ **machine code templates**
- references to constant values in dynamic operations ⇒ **holes**

Optimize set-up code and templates statically

- can customize optimizations to work better assuming unrolled loops

Dynamic Compilation

Static compiler produces **directives** telling dynamic compiler what to do

When dynamic region is first entered, dynamic compiler:

- invokes set-up code to compute constants
- copies templates
- fills in holes with actual constant values
- performs peephole optimizations based on constant value of hole
- selects appropriate successor for run-time constant branches
- unrolls loops as needed

Insert loads from run-time table for big constants

Superscalar instruction issue is problematic

After Dynamic Compilation

source guards:

```
if (x > 1) goto L1;
handler1(args);
L1:
if (y + 2 < z) goto L2;
handler2(args);
L2:
```

bytecodes:

```
VAR(x)
CONST(1)
GT
BR(L1)
CALL(handler1)
L1:
VAR(y)
CONST(2)
ADD
VAR(z)
LT
BR(L2)
CALL(handler2)
L2:
```

generated code:

```
stk[0] = args[x]
stk[1] = 1
stk[0] = stk[0] > stk[1]
if (stk[0]) goto L1;
handler1(args);
L1:
stk[0] = args[y]
stk[1] = 2
stk[0] = stk[0] + stk[1]
stk[1] = args[z]
stk[0] = stk[0] < stk[1]
if (stk[0]) goto L2;
handler2(args)
L2:
```

Initial Experience

application	asymptotic speed-up (application)	break-even point
event dispatching	2.1 (0.28)	~1500 events
calculator	1.5	
sparse vector-matrix multiply	2.1	670 multiplies
record sorting	1.2	900 records
`dinero` (cache simulator)	1.2	

Dynamic compile instructions per generated instruction ≈ 350 or more

- separating set-up code, run-time constants table, directives, and stitcher is **expensive**

Future Directions & Issues

Interprocedural analysis, dynamic procedure specialization, dynamic inlining

Augment dynamic program representation, to support selected dynamic opts

- “register actions” for eliminating constant stack references
- general register allocation, instruction scheduling, post-inlining opts, ...

Assisting/automating annotations

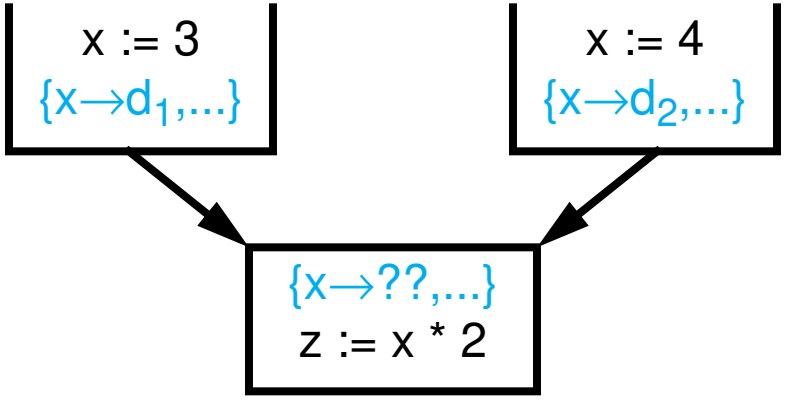
- dynamic profile information
- static analysis of downstream benefits of converting something to a constant

More general kinds of dynamic quasi-invariants

Applications???

Run-Time Constant Merges

What to do at control flow merges?



Depends on whether merge is **constant** or **dynamic**

- constant: only one predecessor will be executed at run-time

If a var is constant along all predecessors, but with different defs, then:
var is constant afterwards iff merge is constant

Identify constant merges using branch reachability analysis,
in parallel with run-time constants identification pass

Unrolled Loops

Since each iteration of a completely-unrolled loop will get its own code,
treat loop head as a constant merge

- plans out what code will look like after run-time loop unrolling

Allows **induction variables** assigned different but fixed values on each iteration
to be treated as run-time constants

- many run-time constants derived from loop induction vars

For byte-coded interpreter programs, need **multi-way & cyclic loop unrolling**

- assignment to `pc` induction var implements forward & backward jumps
  $\Rightarrow$ unrolled loop needs to have forward & backward jumps, too

Programmer annotates unrolled loops,
since practicality hard to assess automatically

Optimizing Set-Up Code and Templates

Once split apart, optimize set-up code and templates separately,
with full arsenal of optimizations

- must adjust some optimizations to cope with hole operands
- can modify some optimizations to exploit which loops will be fully unrolled

Then generate code for each

Also generate **directives** (simple bytecoded language)

- tells dynamic compiler how to turn template code into executable code,
given table of run-time constants computed by set-up code

Dynamic Compilation

Dynamic compiler executes directives:

copies templates,
fills in holes with actual constant values,
performs peephole optimizations based on constant value of hole,
selects appropriate branch for run-time constant branch predicate,
unrolls loops as needed

What if run-time constant value too big for hole immediate field?

- can insert code to compute constant into scratch register, or
- can allocate a table of big constants for dynamic region,
  and insert load from table into scratch register
- floating point & pointer r.t. constants $\Rightarrow$
  insert table load in templates, for better scheduling

Superscalar instruction issue is problematic