Fast, Effective Dynamic Compilation

Joel Auslander
Matthai Philipose
Craig Chambers
Susan Eggers
Brian Bershad

Department of Computer Science & Engineering
University of Washington

Overview of Dynamic Compilation

Defer compilation of **dynamic regions** of programs till run-time,
when **run-time constants** have been computed

Perform constant-related optimizations at run-time:

- constant propagation & folding, strength reduction
- eliminate branches with constant predicates, and eliminate unneeded path
- fully unroll loops

Some applications:

- interpreters
- simulators
- extension and query mechanisms for OS & DB systems
- graphics renderers

Key trade-off: quality of generated code vs. cost of dynamic compilation

Outline

Running example: cache lookup function in cache simulator

- our language and user-interaction models
- our compiler structure
- relationship to related work

Key binding time analysis algorithms

Results for first-generation system

Future directions

The Example

```
LookupResult cacheLookup(unsigned addr, Cache *cache) {
  unsigned blockSize = cache->blockSize;
  unsigned numSets   = cache->numSets;
  unsigned associativity = cache->associativity;
  unsigned setSize   = blockSize * associativity;

  unsigned tag   = addr / (setSize * numSets);
  unsigned index = (addr / setSize) % numSets;
  BlockInfo blocks[] = cache->sets[index];

  int i;
  for (i = 0; i < associativity; i++) {
    if (blocks[i].tag == tag) return CacheHit;
  }

  return CacheMiss;
}
```

Run-Time Constants

```
LookupResult cacheLookup(unsigned addr, Cache *cache) {
  unsigned blockSize = cache->blockSize;
  unsigned numSets   = cache->numSets;
  unsigned associativity = cache->associativity;
  unsigned setSize   = blockSize * associativity;

  unsigned tag   = addr / (setSize * numSets);
  unsigned index = (addr / setSize) % numSets;
  BlockInfo blocks[] = cache->sets[index];

  int i;
  for (i = 0; i < associativity; i++) {
    if (blocks[i].tag == tag) return CacheHit;
  }

  return CacheMiss;
}
```

Exploiting Run-Time Constants

"Hoist" run-time constant calculations out of main code, executed at most once

Cache constants in:

- run-time table data structure, inserting table references in main code
  [Data Specialization]
  - can rely on static compilation alone
  - can't fully unroll loops

- immediates of dynamically-generated main code
  [Fabius, Tempo]

- either, depending on size of constant
  [us]

Identifying Profitable Run-Time Constants

Ideal: automatically identify run-time constants and profitable dynamic compilation

Challenges:

- sophisticated whole-program alias & side-effect analysis
- dynamic profile information to identify frequency of dynamic regions

User **hints** [Fabius]

- functional language, so no side-effect analysis needed

User **directives** [Tempo, us]

- support general-purpose imperative languages, but safety depends on user
- compiler identifies derived run-time constants, does optimizations

User **manipulation** of code [‘C]

- arbitrary optimizations, but requires user effort

Annotated Example

```
LookupResult cacheLookup(unsigned addr, Cache *cache) {
  dynamicRegion(cache) {
    unsigned blockSize = cache->blockSize;
    unsigned numSets   = cache->numSets;
    unsigned associativity = cache->associativity;
    unsigned setSize   = blockSize * associativity;

    unsigned tag   = addr / (setSize * numSets);
    unsigned index = (addr / setSize) % numSets;
    BlockInfo blocks[] = cache->sets[index];

    int i;
    unrolled for (i = 0; i < associativity; i++) {
      if (blocks[i].tag == tag) return CacheHit;
    }

    return CacheMiss;
  }
}
```

Overview of Our System

Two main goals:

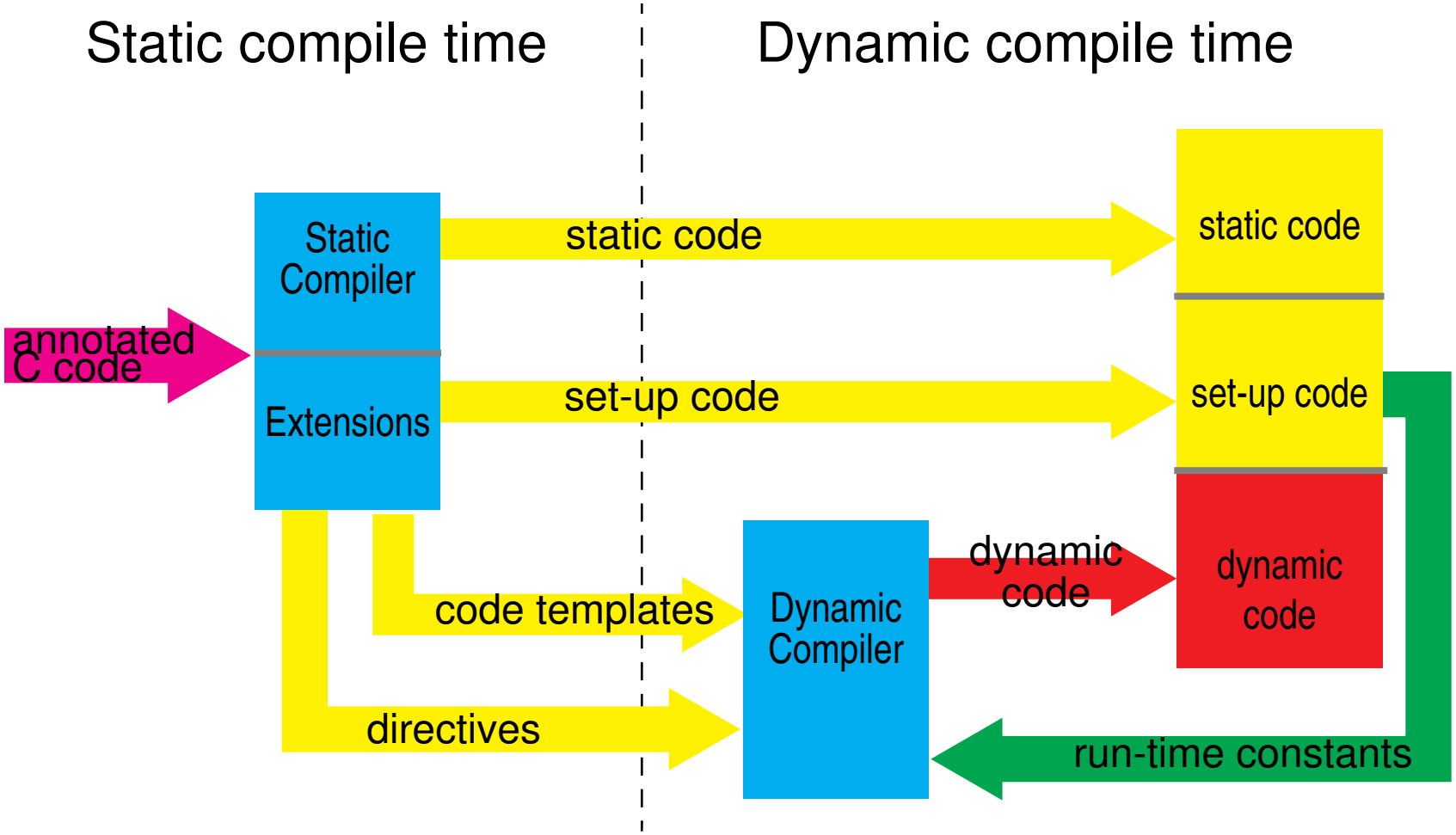
High-quality generated code

- integrate optimizations within optimizing compiler (Multiflow)

Fast dynamic compilation

- plan effect of dynamic compilation statically
- produce machine code templates, with holes for run-time constant values
- dynamic compiler just evaluates run-time constant calculations,
  copies templates, and fills in holes

Compiler Framework



Set-Up Code and Templates for Example

Set-up code:

```
blockSize          = cache->blockSize;
Consts.numSets     = cache->numSets;
associativity      = cache->associativity;
Consts.setSize     = blockSize * associativity;
Consts.ss_ns       = setSize * numSets;
Consts.cacheSets = cache->sets
int i;
for (i = 0; i < associativity; i++) {
  Consts.loop[i].i = i;
}
```

Template 1:

```
unsigned tag   = addr / Consts.ss_ns;
unsigned index = (addr / Consts.setSize) % Consts.numSets;
BlockInfo blocks[] = Consts.cacheSets[index];
```

Template 2:

```
if (blocks[Consts.loop[i].i].tag == tag) return CacheHit;
```

Dynamically-Generated Code for Example

For 4-way set-associative cache with 512 sets and 32-byte blocks:

```
unsigned tag   = addr >> 14;
unsigned index = (addr >> 5) & 511;
BlockInfo blocks[] = 0x4523450[index];
if (blocks[0].tag == tag) return CacheHit;
if (blocks[1].tag == tag) return CacheHit;
if (blocks[2].tag == tag) return CacheHit;
if (blocks[3].tag == tag) return CacheHit;
return CacheMiss;
```

Static Compiler Phases

Given control flow graph for procedure containing dynamic region(s):

1. Identify derived run-time constants,
   given initial run-time constants and `unrolled` annotation

2. Divide dynamic region into control flow subgraphs for
   run-time constant calculations (set-up code) and
   run-time variable calculations (templates) with special operands (holes)

3. Compile and optimize whole procedure

4. Produce machine code,
   plus directives to dynamic compiler to copy templates and fill holes

Identifying Run-Time Constants

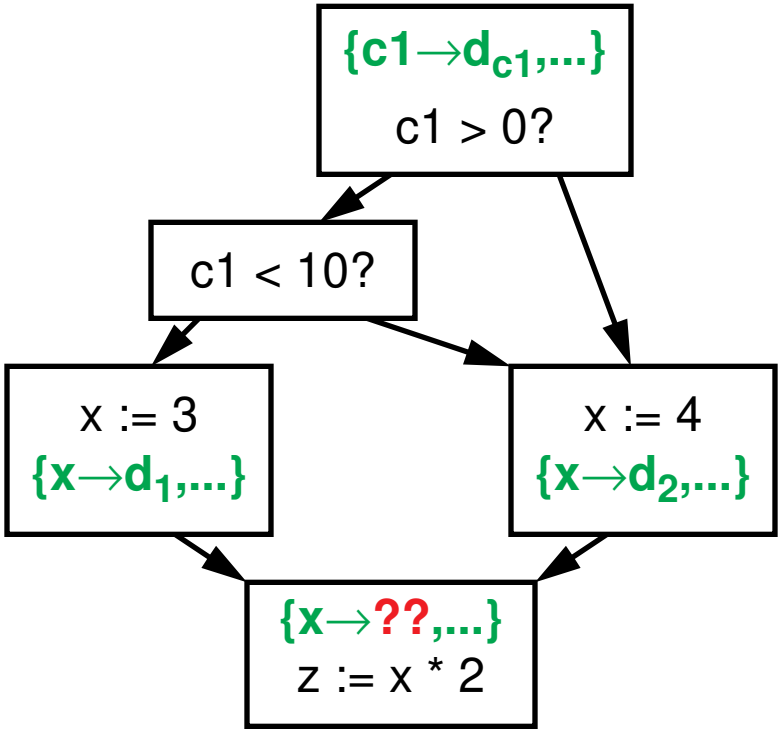
Identifying derived run-time constants is an abstract form of constant propagation, like binding time analysis in partial evaluators

- if operands are run-time constant & operator has no side-effects
  $\Rightarrow$ result is run-time constant

- if pointer is run-time constant $\Rightarrow$ result of load through pointer is constant, unless annotated otherwise

- if loop is unrolled, then induction variables can be run-time constant

Run-Time Constant Merges



If merge's predecessors are mutually-exclusive, deduce more run-time constants

Identify constant merges using **constant branch reachability analysis**

- handles arbitrary control flow graphs well
- run in parallel with identifying run-time constants

First Generation Results

On DEC Alpha 21064:

Benchmark	**Asymptotic Speedup**
Reverse-polish stack-based desk calculator	1.7
Scalar-matrix multiply	1.6
Sparse matrix-vector multiply	1.8
Event dispatcher in SPIN	1.4
Quicksort record sorter	1.2

VLIW-like dual-issue restrictions on Alpha problematic for patching templates

Dynamic compilation overhead: 150 or more instructions/generated instruction

Future Work

Faster dynamic compilation: merge set-up, directives, and stitcher

Run-time constants that change

Interprocedural dynamic regions, dynamic procedure specialization,
dynamic inlining

Augment dynamic program representation to support selected dynamic opts

- "register actions" for eliminating constant stack references:
  event dispatcher speed-up: **1.4** $\rightarrow$ **4.1**

Assisting/automating annotations

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

Summary

High-quality dynamically-compiled code:

- applying intraprocedural optimizations to procedures containing dynamically-generated code
- deriving run-time constants well in presence of arbitrary control flow
- future: expanded scope of analysis & additional run-time optimizations

Fast dynamic compilation:

- planning most dynamic compilation work statically
- future: even faster by integrating set-up with stitching

Support general language model with side-effects, pointers, casts, etc.

- rely on (sparse) programmer annotations
- future: work towards producing annotations automatically

Credits

Other team members:

Brian Grant
Markus Mock
Scott van Woudenberg
Wilson Hsieh
Przemyslaw Pardyak

Support:

ARPA
ONR
NSF

DEC
IBM
Sun