Fast, Effective Dynamic Compilation

Craig Chambers, Susan Eggers

Wilson Hsieh

Joel Auslander, Brian Grant, Markus Mock, Matthai Philipose

Scott van Woudenberg

Dynamic Compilation

What is dynamic compilation?

- defer compilation until run-time
- specialize code to **run-time constants**

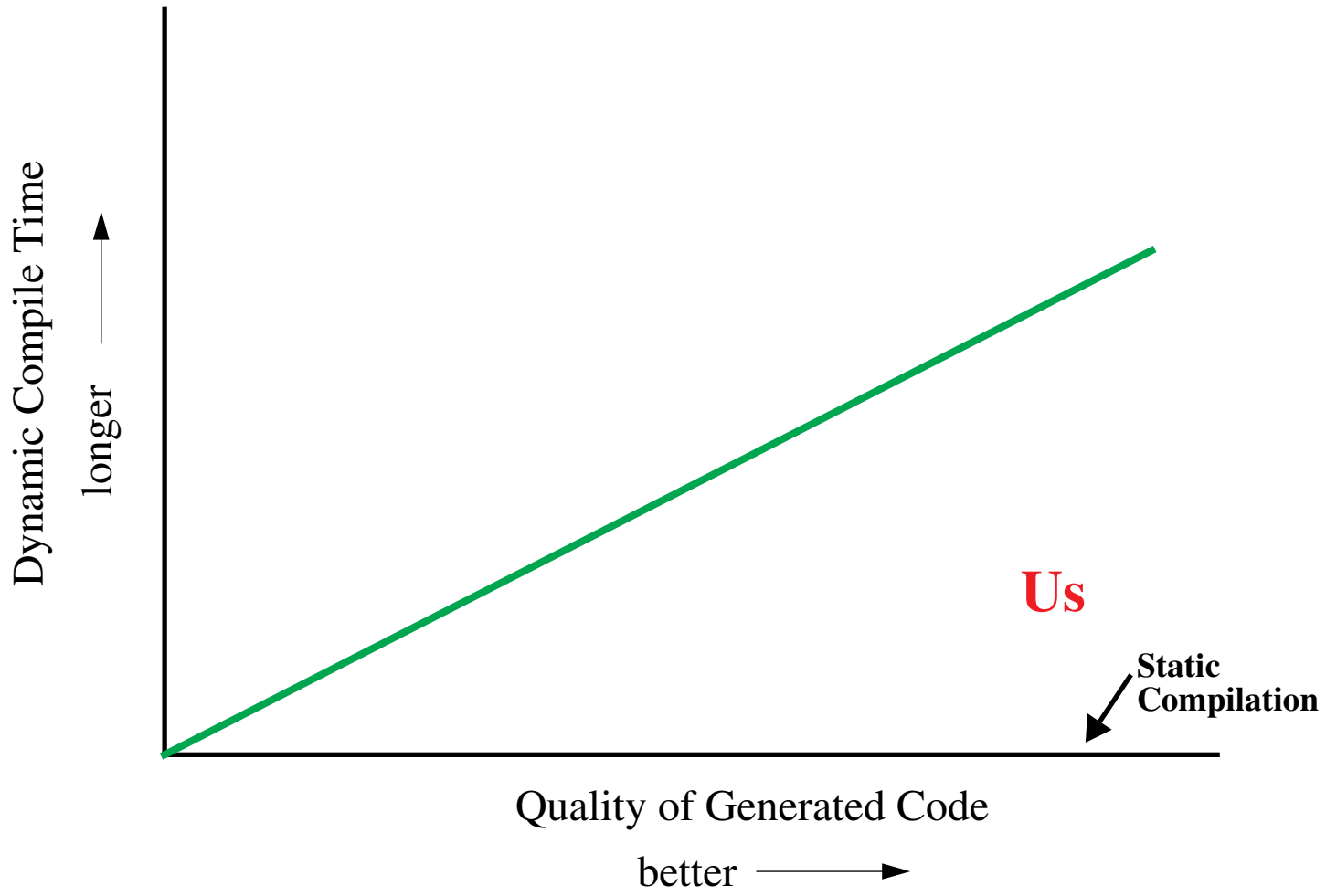
Why do it?

- specialized code runs faster
- optimizations at run-time
  - load elimination
  - constant-based opts
  - complete loop unrolling
  - dead code elimination of branches
  - strength reduction

Why is it difficult?

- overhead of the dynamic compile
- time saved by executing specialized code >> time spent optimizing

Dynamic Compiler Frameworks



Our Dynamic Compiler Framework

Our approach:

- high-quality static compiler
  - good job of identifying run-time constants
  - preplans optimizations that will be done at run-time
  - generate machine-code templates with holes

  and
- fast, simple dynamic compiler (the *stitcher*)
  - copies templates
  - fills in holes with run-time values

- general purpose programming languages
- simple annotations

Dynamic Compilation System

Topics

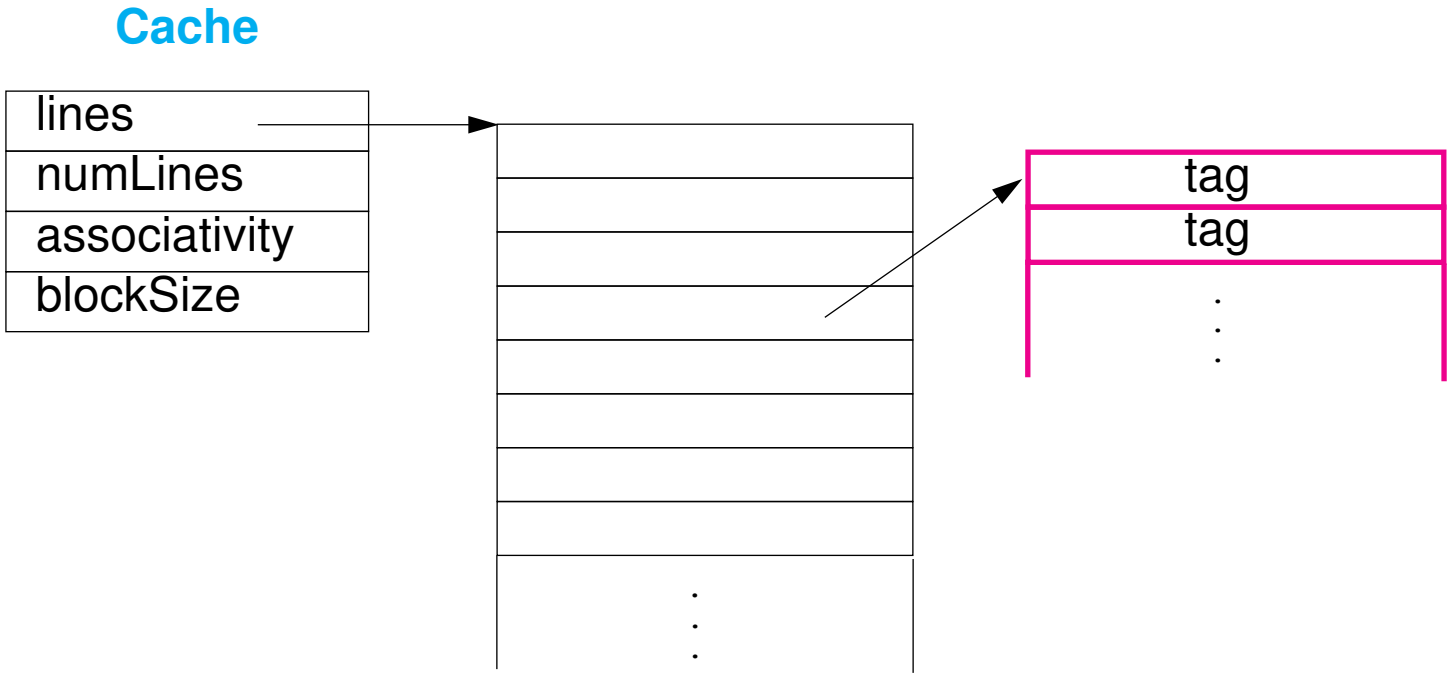
- learn by example: cache lookup routine
- our compiler framework
- results so far
- what's next

Cache Lookup Example

```
cacheResult cacheLookup(void *addr, Cache *cache) {

  unsigned blockSize = cache->blockSize;
  unsigned numLines = cache->numLines;
  unsigned tag = (unsigned) addr / (blockSize * numLines);
  unsigned index = ((unsigned) addr / blockSize) % numLines;
  setStructure *setArray = cache->lines[index];
  unsigned associativity = cache->associativity;
  int set;

  for (set = 0; set < associativity; set++) {
    if (setArray[set].tag == tag) return CacheHit;
  }
  return CacheMiss;

}
```

Cache Lookup Example

Address

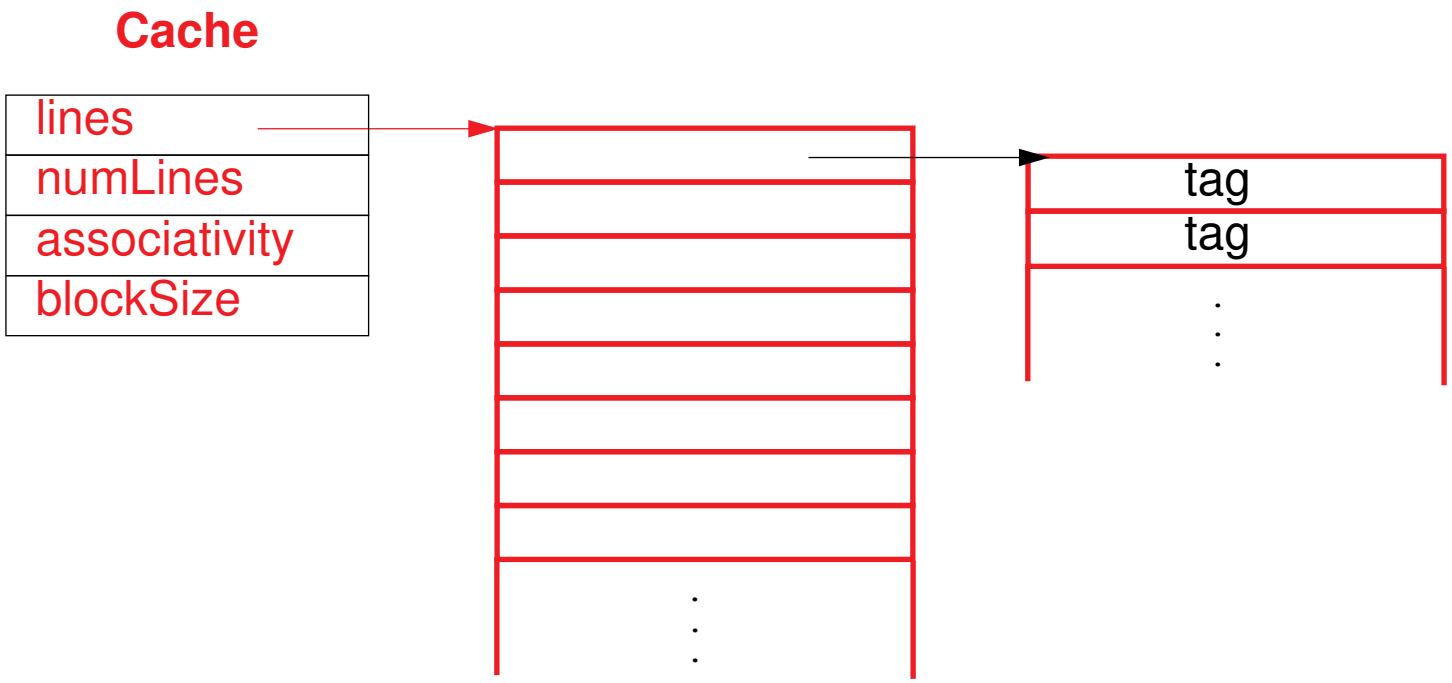


Cache Data Structures

Cache Lookup Example

```
cacheResult cacheLookup(void *addr, Cache *cache) {

  unsigned blockSize = cache->blockSize;
  unsigned numLines = cache->numLines;
  unsigned tag = (unsigned) addr / (blockSize * numLines);
  unsigned index = ((unsigned) addr / blockSize) % numLines;
  setStructure *setArray = cache->lines[index];
  unsigned associativity = cache->associativity;
  int set;

  for (set = 0; set < associativity; set++) {
    if (setArray[set].tag == tag) return CacheHit;
  }
  return CacheMiss;

}
```

Cache Lookup Example



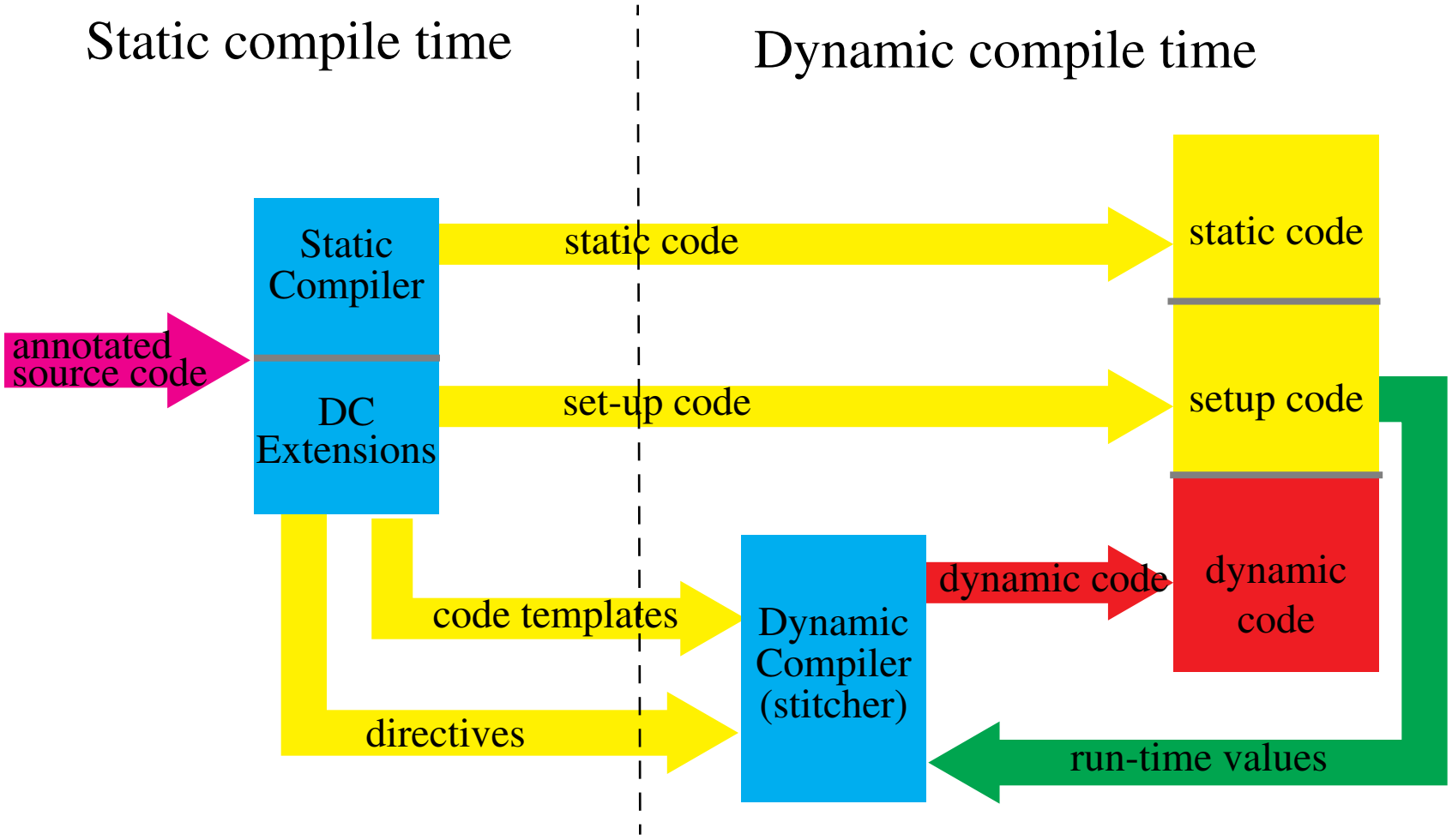
Cache Data Structures

Cache Lookup Example

4-way set associative, 32 byte block cache

```
unsigned tag = (unsigned)addr >> 14;
unsigned index = ((unsigned)addr >> 5) & 511;
setStructure *setArray = cacheLines[index];
if (setArray[0].tag == tag) return CacheHit;
if (setArray[1].tag == tag) return CacheHit;
if (setArray[2].tag == tag) return CacheHit;
if (setArray[3].tag == tag) return CacheHit;
return CacheMiss;
```

Dynamic Compiler Framework



Dynamic Compiler Framework

Annotations

```
cacheResult cacheLookup(void *addr, Cache *cache) {

 dynamicRegion(cache) {
  unsigned blockSize = cache->blockSize;
  unsigned numLines = cache->numLines;
  unsigned tag = (unsigned) addr / (blockSize * numLines);
  unsigned line = ((unsigned) addr / blockSize) % numLines;
  setStructure *setArray = cache->lines[line];
  unsigned associativity = cache->associativity;
  int set;

  unrolled for (set = 0; set < associativity; set++) {
   if (setArray[set].tag == tag) return CacheHit;
  }
  return CacheMiss;
 }
}
```

Dynamic Compiler Framework

Templates

- **computations** that depend in part on dynamic variables
- **markers** to guide the stitcher
  - machine-code with **holes** for run-time constant values
  - delimiters for the dynamic regions
  - aides for loop unrolling

Dynamic Compiler Framework

Templates

```
cacheResult cacheLookup(void *addr, Cache *cache) {

 dynamicRegion {

  unsigned tag = (unsigned) addr / HOLE_2;
  unsigned line = ((unsigned) addr / HOLE_0) % HOLE_1;
  setStructure *setArray = HOLE_3[line];

  unrolled for (set) {
    if (setArray[HOLE_4set].tag == tag) return CacheHit;
  }
  return CacheMiss;

 }

}
```

Dynamic Compiler Framework

Set-up Code

- computes run-time constants that can be derived from the annotations
- part of the mechanism for performing optimizations at run-time

Dynamic Compiler Framework

Set-up Code

```
HOLE_0 = cache->blockSize;
HOLE_1 = cache->numLines;
HOLE_2 = HOLE_0 * HOLE_1;
HOLE_3 = cache->lines;
associativity = cache->associativity;
for (set = 0; set < associativity; set++) {
 HOLE_4set = set;
}
```

Dynamic Compiler Framework

Stitcher Directives

- instruct the stitcher how to produce executable code from:
  - machine-code templates
  - set-up code results
- the rest of the mechanism for performing optimizations at run-time
- examples
  - patch `HOLE_0`
  - unroll another loop iteration
  - eliminate a branch and the path not taken
  - adjust branch offsets

Dynamic Compiler Framework

What the static compiler does

- identifies run-time constants
  - run-time constants identification
  - reachability analysis
- extracts set-up code and templates
- optimizes them
- generates machine code and stitcher directives

What the dynamic compiler (stitcher) does

- executes set-up code **once**
- applies the directives **once**
- optimized templates executed repeatedly

Dynamic Compiler Framework

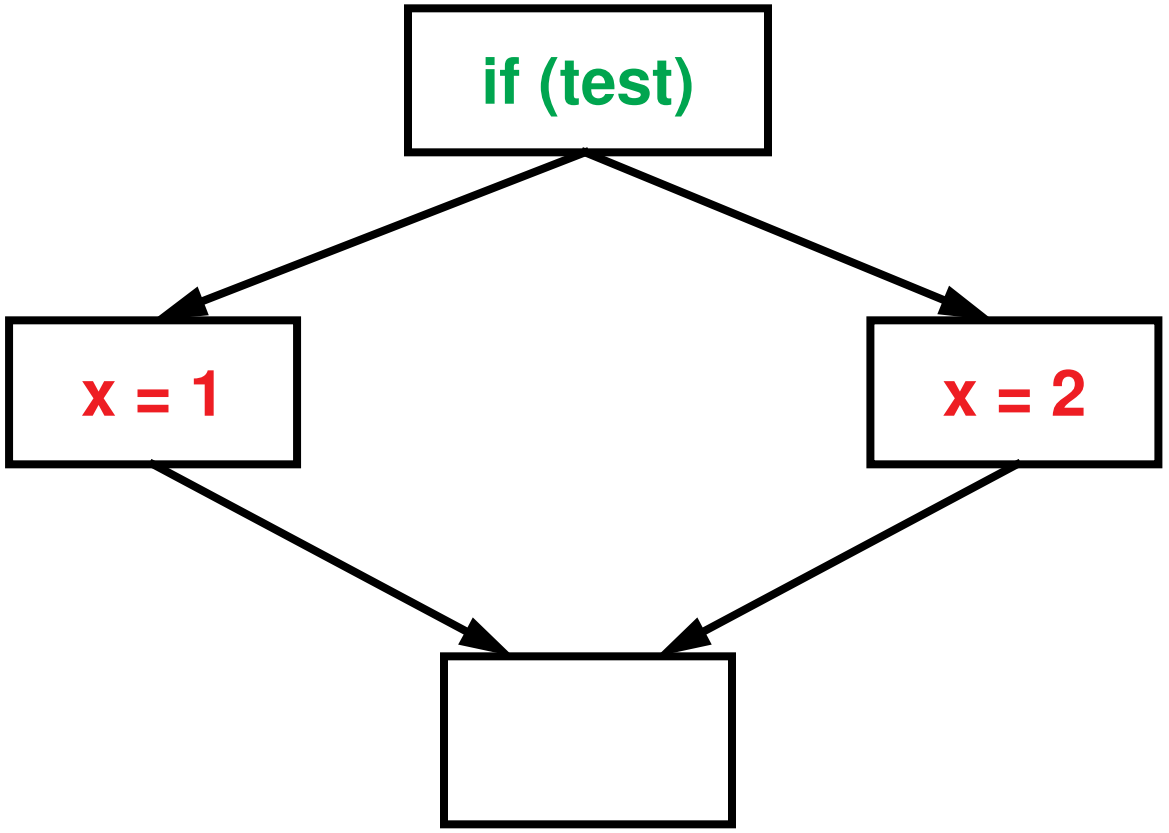
Run-time constants identification

- computes variables that are constant at each program point
- start with annotated set: add to it
- rich set of data structures (arrays, pointer-based structures)

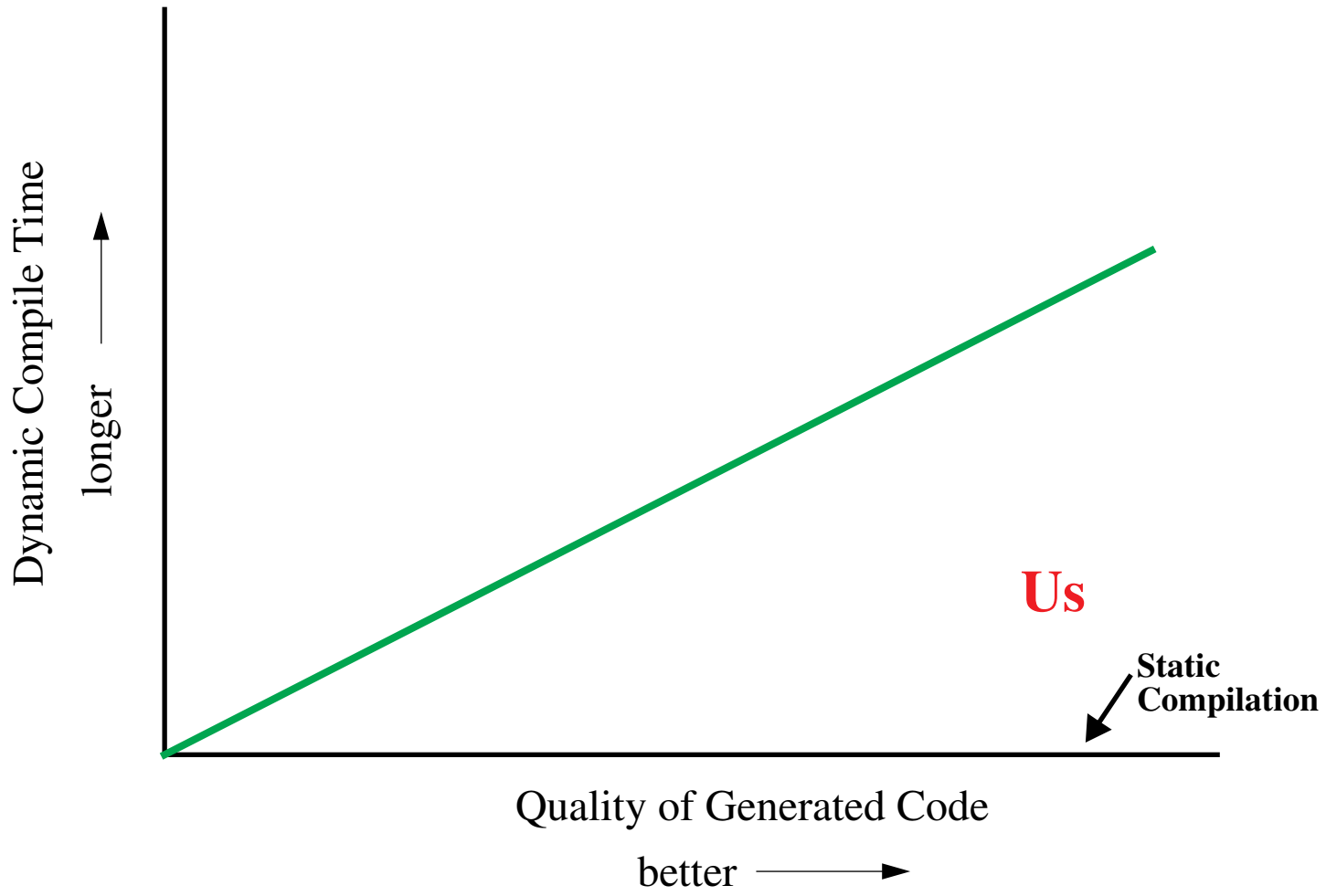
Reachability analysis

- identifies run-time constants that lie on paths after a run-time constant branch
- rich set of control structures (`if`, `switch`, unstructured control flow)

Dynamic Compiler Framework



Dynamic Compiler Frameworks



Dynamic Compiler Framework

Our approach

- **high-quality dynamically-compiled code**
- **low dynamic compilation cost**

How we'll do this

- do as much work as possible statically & produce optimized templates
  - do a good job of identifying run-time constants
    (run-time constants and reachability analyses)
  - machine-code templates contain only code that depends on run-time constants and variables
    (good planning & separate set-up code)
  - optimize machine-code templates with their surrounding code
    (separate set-up code)
  - use a high quality static compiler
    (Multiflow compiler)

- leave only fast optimizations for run-time
  - completes machine-code template construction
  - patch holes, copy templates
  - peephole optimizations

Preliminary Results

Application	Dynamic Region	Run-time Constant	Speedup	DC overhead
Reverse-polish stack-based desk calculator	the function interpreter loop	polynomial expression in x & y	1.5	.2%
Scalar-matrix multiply	multiplication loop	the scaling factor	1.7	.7%
XLisp				
POV (ray tracer)				
sort on records				
robot designer				
sparse matrices				
SPIN dispatcher				

Future Directions

Explore and expand applications suite

- numerical & scientific code
- systems code
- parallel codes
- ...

Extend basic framework

- handle run-time constants that can change
- handle interprocedural dynamic regions
- support inlining
- support unrolling loops over trees, dags, and graphs
- more, as motivated by applications

More Future Directions

Investigate alternative divisions of labor between static & dynamic compilers

Basic trade-off: as the dynamic compiler does more optimizations

==> better quality code

==> slower dynamic compilation

Enrich machine-code representation of programs

to support desired run-time optimizations

Ideas:

- static compiler generates specialized intermediate representation designed to support selected dynamic optimizations quickly
- for different regions, make different tradeoffs, based on benefits & costs of dynamic optimizations for each region

More Future Directions

Explore automatic dynamic compilation

Key ingredients:

- dynamic profile information
- static analysis of potential benefits of dynamic compilation
- alias & side-effect analyses
- effect of underlying architecture on dynamic compilation