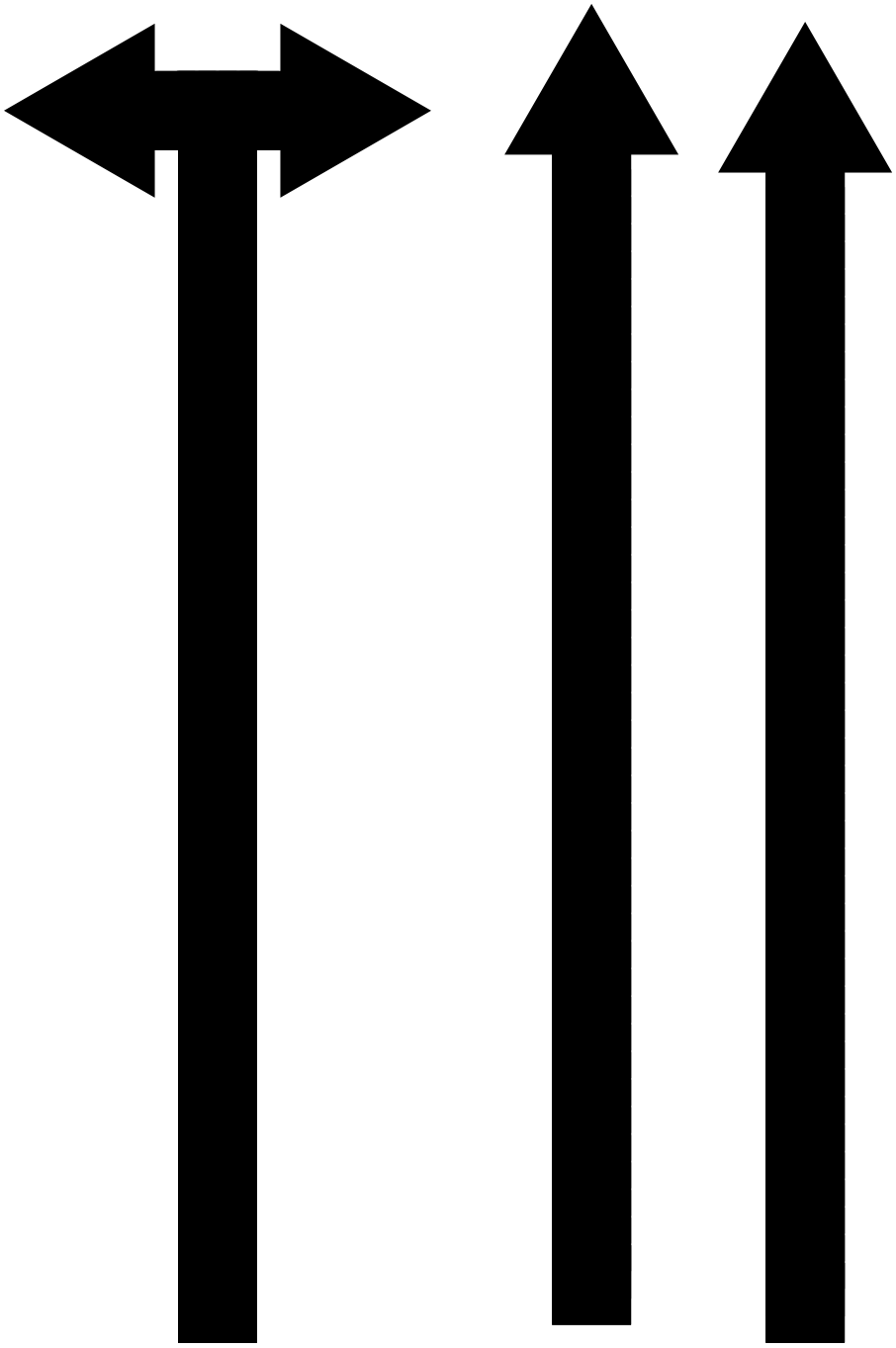

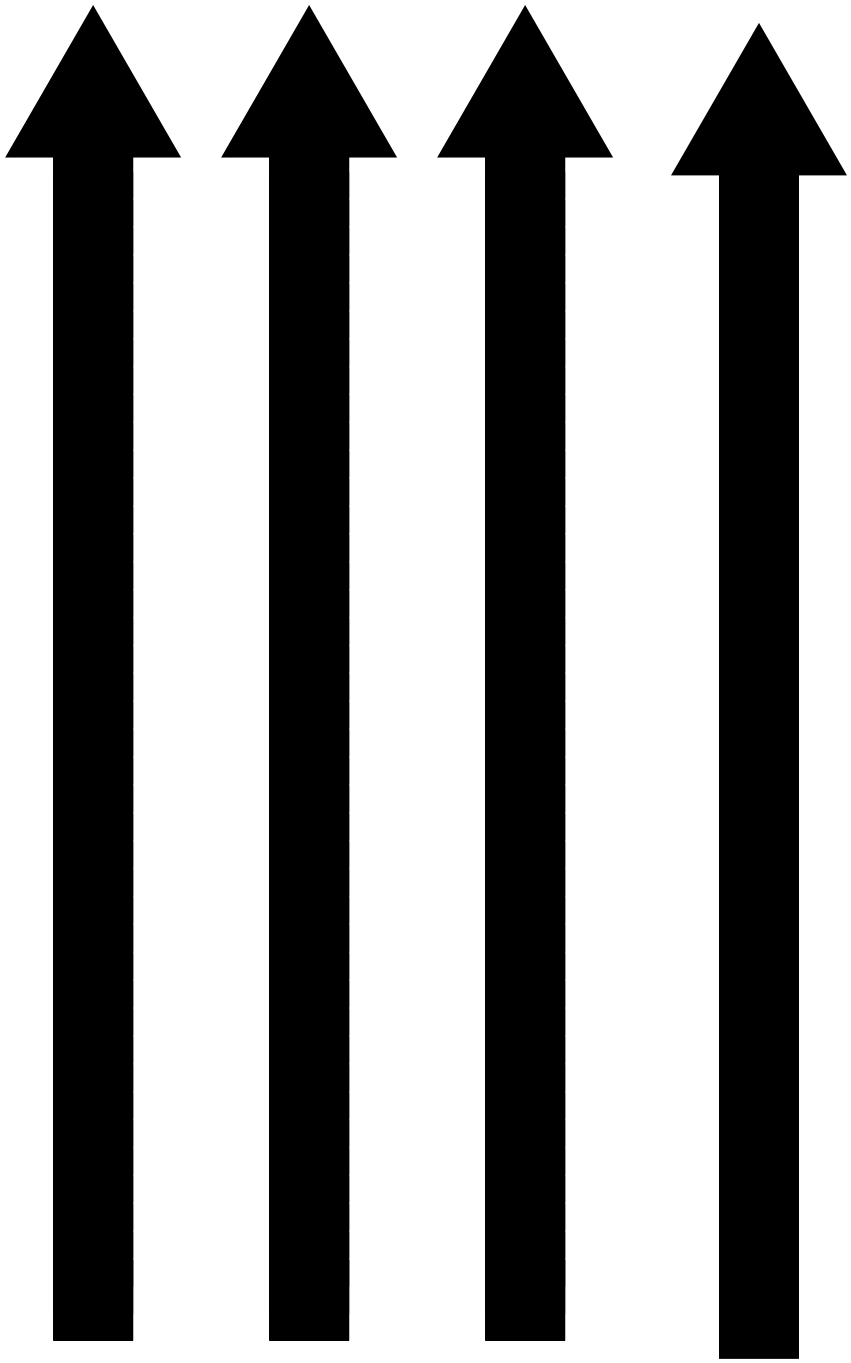

Executable Code

Before Run Time

During Run Time

Dynamic Compiler

- Fills holes with run-time constants
- Performs final optimizations

Static Compiler

- Code is split into template and setup code
- Setup code is compiled into normal machine code
- Template code is compiled into machine code with holes (where run-time constants will later go)

Run Time Constants

```
HOLE_0 = 32
HOLE_1 = 512
HOLE_2 = 16384
HOLE_3 = cacheLines
HOLE_4_0 = 0
HOLE_4_1 = 1
HOLE_4_2 = 2
HOLE_4_3 = 3
```

Dynamically Compiled Code

```
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;
```

Template Code

```
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;

 }

}
```

Setup 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;
}
```

Input: Annotated Source Code

```
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;
 }
}
```

Fast, Effective Dynamic

Joel Auslander, Matthai Philipose, Craig Chambers, Susan J.

ve Dynamic Compilation

ilipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad

Template Code

What is it?

- Template code is optimized machine code with empty immediate fields, called holes, in instructions that use run-time constants

Why does template code lead to fast dynamic compilation?

- Most of the compilation has already been done
- Optimizations based on run-time constants can be done quickly:

optimization	applied to template code
constant inlining	insert run-time constants directly into the holes
full loop unrolling	stamp out copies of the loop body
strength reduction	replace inefficent instructions with reduced equivalents during constant inlining

Preliminary Results

Benchmark	Run-time Constant Configuration	Asymptotic Speedup	Break-even Point
Reverse-polish stack-based desk calculator	polynomial expression in x & y	1.7	916 interpretations
Scalar-matrix multiply	the scaling factor	1.6	31K multiplies
Sparse matrix-vector multiply	200x200 matrix, 10 elements/row, 5% density	1.8	2650 matrix multiplies
	96x96 matrix, 5 elements/row, 5% density	1.5	1850 matrix multiplies
Event dispatcher in SPIN	7 predicate types, 10 event guards	1.4	722 dispatches
Quicksort record sorter	4 keys	1.2	3050 records

Our Approach

Prepare for Dynamic Compilation Statically

- Programmer annotations identify run-time constants
- Static compiler separates code into *setup code* and *template code*
- Setup and template are compiled to optimized machine code

Fast Dynamic Compilation at Run-Time

- Setup code computes the values of run-time constants
- Dynamic compiler optimizes template code based on these values

Dynamic Compilation

What is it?

- Defering some compilation to run-time

Why do it?

- To optimize code based on *run-time constants*, values that are set early and remain fixed until the program finishes

What's the difficulty?

- Slow dynamic compilation can cost more than it saves
- Determining run-time constants is difficult, especially in C