Dynamic Optimizations

Last time
− Predication and speculation

Today
− Dynamic compilation

Limitations of static analysis
− Programs can have values and invariants that are known at runtime but unknown at compile time. Static compilers cannot exploit such values or invariants
− Many of the motivations for profile-guided optimizations apply here

Basic idea
− Perform translation at runtime when more information is known
− Traditionally, two types of translations are done
  − Runtime code generation (JIT compilers)
  − Partial evaluation (Staged compilation)
Partial Evaluation

Basic idea
- Take a general program and partially evaluate it, producing a specialized program that’s more efficient
  
  \[ f(a,b,c) \rightarrow f'(a,b), \text{ where the result has its third parameter hard-coded into the implementation. } f' \text{ is typically more efficient than } f \]

- Exploit runtime constants, which are variables whose value does not change during program execution, e.g., write-once variables

Exploiting runtime constants
- Perform constant propagation
- Eliminate memory ops
- Remove branches
- Unroll loops

Applications with Runtime Constants

Interpreters: Program being interpreted is runtime constant
Simulators: Subject of simulation (circuit, cache, network) is runtime constant
Graphics renderers: The scene to render is runtime constant
Scientific simulations: Matrices can be runtime constants
Extensible OS kernels: Extensions to the kernel can be runtime constant

Examples
- A cache simulator might take the line size as a parameter
- A partially evaluated simulator might produce a faster simulator for the special case where the line size is 16
Partial Evaluation (cont)

Active research area
- Interesting theoretical results
  - Can partially evaluate an interpreter with respect to a program \(i.e.,\) compile it [1st Futamura projection]
  - Can partially evaluate a partial evaluator with respect to an interpreter \(i.e.,\) generate a compiler [2nd Futamura projection]
  - Can partially evaluate a partial evaluator with respect to a partial evaluator \(i.e.,\) generate a compiler generator [3rd Futamura projection]
- Most PE research focuses on functional languages
- Key issue
  - When do we stop partially evaluating the code when there is iteration or recursion?

Dynamic Compilation with DyC

DyC [Auslander, et al 1996]
- Staged compilation
- Apply ideas of Partial Evaluation
- Perform some of the Partial Evaluation at runtime
  - Can handle more runtime constants than Partial Evaluation
- Reminiscent of link-time register allocation in the sense that the compilation is performed in stages

Tradeoffs
- Must overcome the run-time cost of the dynamic compiler
  - Fast dynamic compilation: low overhead
  - High quality dynamically generated code: high benefit
- Ideal: dynamically translate code once, execute this code many times
- Implication: don’t dynamically translate everything
  - Only perform dynamic translation where it will be profitable
Applying Dynamic Compilation

System goal
- Both fast dynamic compilation and high quality compiled code

How do we know what will be profitable?
- Let user annotations guide the dynamic compilation process

System design
- Dynamic compilation for the C language
- Declarative annotations:
  - Identify pieces of code to dynamically compile: dynamic regions
  - Identify source code variables that will be constant during the execution of dynamic regions

Staged Compilation in DyC

- Make the static compiler do as much work as possible
- Give the dynamic compiler as little work as possible
Dynamically Compiled Code

Static compiler
- Produces machine code templates, in addition to normal mach code
- Templates contain holes that will be filled with runtime const values
- Generates setup code to compute the vals of these runtime consts.

- Together, the template and setup code will replace the original dynamic region

The Dynamic Compiler

The Stitcher
- Follows directives, which are produced by the static compiler, to copy code templates and to fill in holes with appropriate constants

- The resulting code becomes part of the executable code and is hopefully executed many times
The Annotations

cacheResult cacheLookup (void *addr, Cache *cache) {
  dynamicRegion(cache) { /* cache is a runtime constant */
    int blockSize = cache->blockSize;
    int numLines = cache->numLines;
    int tag = addr / (blockSize * numLines);
    int line = (addr / blockSize) % numLines;
    setStructure **setArray = cache->lines[line]->sets;
    int assoc = cache->associativity;
    int set;

    unrolled for (set=0; set<assoc; set++) {
      if (setArray[set]dynamic->tag == tag)
        return CacheHit;
    }
  }
  /* end of dynamic region */
}

The Annotations

cacheResult cacheLookup (void *addr, Cache *cache) {
  dynamicRegion(cache) { /* cache is a runtime constant */
    int blockSize = cache->blockSize;
    int numLines = cache->numLines;
    int tag = addr / (blockSize * numLines);
    int line = (addr / blockSize) % numLines;
    setStructure **setArray = cache->lines[line]->sets;
    int assoc = cache->associativity;
    int set;

    unrolled for (set=0; set<assoc; set++) {
      if (setArray[set]dynamic->tag == tag)
        return CacheHit;
    }
    return CacheMiss;
  } /* end of dynamic region */
}
The Annotations

```c
/*
dynamic
- Any type of data can be considered constant
- In particular, contents of arrays and pointer-based structures are assumed to be runtime constant whenever they are accessed by runtime constant pointers
- To ensure that this assumption is correct, users must insert the dynamic annotation to mark pointer refs that are not constant
*/

if (setArray[set]dynamic->tag == tag)
    return CacheHit;
return CacheMiss;
} /* end of dynamic region */
```

The Annotations

```c
unrolled
- Directs the compiler to completely unroll a loop
- Loop termination must be governed by runtime constants
  - The static compiler can check whether this annotation is legal
  - Complete unrolling is a critical optimization
  - Allows induction variables to become runtime constants

unrolled for (set=0; set<assoc; set++) {
    if (setArray[set]dynamic->tag == tag)
        return CacheHit;
    return CacheMiss;
} /* end of dynamic region */
```
The Annotations

cacheResult cacheLookup (void *addr, Cache *cache) {
dynamicRegion key{cache, foo} {

key
– Allows the creation of multiple versions of a dynamic region, each
using different runtime constants

– Separate code is dynamically generated for each distinct combination
of values of the runtime constants

} /* end of dynamic region */

The Need for Annotations

Annotation errors
– Lead to incorrect dynamic compilation
  – *e.g.*, Incorrect code if a value is not really a runtime constant

Automatic dynamic compilation is difficult
– Which variables are runtime constant over which pieces of code?
– Complicated by aliases, side effects, pointers that can modify memory
– Which loops are profitable to unroll?
– Estimating *profitability* is the difficult part
The Static Compiler

Operates on low-level IR
- CFG + three address code in SSA form

Tasks
- Identifies runtime constants inside of dynamic regions
- Splits each dynamic region subgraph into set-up and template code subgraphs
- Optimizes the control flow for each procedure
- Generates machine code, including templates
  - In most cases, table space for runtime constants can be statically allocated
  - What do we do about unrolled loops?
- Generates stitcher directives

Detecting Runtime Constants

Simple data-flow analysis
- Propagates initial runtime constants through the dynamic region using the following transfer functions

\[-x = y\]  \(x\) is a constant iff \(y\) is a constant

\[-x = y \text{ op } z\]  \(x\) is a constant iff \(y\) and \(z\) are constants and \(\text{op}\) is an idempotent, side-effect free, non-trapping op

\[-x = f(y_1, \ldots, y_n)\]  \(x\) is a constant iff the \(y_i\) are constants and \(f\) is an idempotent, side-effect free, non-trapping function

\[-x = *p\]  \(x\) is a constant iff \(p\) is constant

\[-x = \text{dynamic } *p\]  \(x\) is not constant
Detecting Runtime Constants (cont)

Merging control flow
- If a variable has the same runtime constant reaching definition along all predecessors, it’s considered a constant after the merge.

\[
\begin{align*}
  t_1 &= \text{test} \\
  t_1? \\
  x_1 &= 1 \\
  x_2 &= 2 \\
  x_3 &= \phi(x_1, x_2) \\
\end{align*}
\]

- If test is a runtime constant, then we’ll always take one branch or the other.
- If test is constant, \(\phi\) is idempotent so the result is constant.
- If test is not constant, \(\phi\) is not idempotent, so the result is not constant.

Optimizations

Integrated optimizations
- For best quality code, optimizations should be performed across dynamic region boundaries, e.g., global CSE, global register allocation.
- Optimizations can be performed both before and after the dynamic region has been split into setup and template codes.

Restrictions on optimizing split code
- Instructions with holes cannot be moved outside of their dynamic region.
- Holes cannot be treated as legal values outside of the dynamic region. (e.g., Copy propagation cannot propagate values of holes outside of dynamic regions)
- Holes are typically viewed as constants throughout the dynamic region, but induction variables become constant for only a given iteration of an unrolled loop.
The Stitcher

Performs directive-driven tasks
- Patches holes in templates
- Unrolls loops
- Patches pc-relative instructions (such as relative branches)

Performs simple peephole optimizations
- Strength reduction of multiplies, unsigned division, modulus

The End Result

Final dynamically generated code from our example
- Assuming the following configuration:
  - 512 lines, 32 byte blocks, 4-way set associative
  - cacheLines is an address loaded from the runtime constants table

```c
void cacheLookup(void *addr, Cache *cache) {
    int gat = addr >> 14;
    int line = (addr >> 5) & 511;
    setStructure **setArray = cache->lines[line]->sets;
    if (setArray[0]->tag == tag) goto L1;
    if (setArray[1]->tag == tag) goto L1;
    if (setArray[2]->tag == tag) goto L1;
    if (setArray[3]->tag == tag) goto L1;
    return CacheMiss;
L1:   return CacheHit;
}```
The Original Code without Annotations

```c
void cacheResult cacheLookup (void *addr, Cache *cache) {
    int blockSize = cache->blockSize;
    int numLines = cache->numLines;
    int tag = addr / (blockSize * numLines);
    int line = (addr / blockSize) % numLines;
    setStructure **setArray = cache->lines[line]->sets;
    int assoc = cache->associativity;
    int set;

    for (set = 0; set < assoc; set++) {
        if (setArray[set]->tag == tag)
            return CacheHit;
    }
    return CacheMiss;
}
```

Performance Results

Two measures of performance

- **Asymptotic improvement**: speedup if overhead were 0
- **Break even point**: the fewest number of iterations at which the dynamic compilation system is profitable

<table>
<thead>
<tr>
<th>benchmark</th>
<th>asymptotic speedup of dynamic regions</th>
<th>breakeven point</th>
</tr>
</thead>
<tbody>
<tr>
<td>calculator</td>
<td>1.7</td>
<td>916 interpretations</td>
</tr>
<tr>
<td>matrix multiply</td>
<td>1.6</td>
<td>31,392 scalar *'s</td>
</tr>
<tr>
<td>sparse mat multiply</td>
<td>1.8</td>
<td>2645 matrix *'s</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>1858 matrix *'s</td>
</tr>
<tr>
<td>event dispatcher</td>
<td>1.4</td>
<td>722 event dispatches</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.2</td>
<td>3050 records</td>
</tr>
<tr>
<td></td>
<td>1.2</td>
<td>4760 records</td>
</tr>
</tbody>
</table>
**Evaluation**

**Today’s discussion**
- Simple caching scheme
  - Setup once, reuse thereafter
- More sophisticated schemes are possible
  - Can cache multiple versions of code
  - Can provide eager, or speculative, specialization
  - Can allow different dynamic regions for different variables

**Recent progress on DyC**
  - More complexity is needed
  - Extremely difficult to annotate the applications
- Automated insertion of annotations [Mock, et al 2000]
  - Use profiling to obtain value and frequency information

**Lessons**

**Is dynamic compilation worthwhile?**
- For optimization, need to be careful because of dynamic compilation costs
- Important for Java (Just in Time compilers)

**Dynamo: HP Labs** [Bala, et al 2000]
- Dynamically translate binaries (no annotations)
- Only modest performance improvements
- But many other interesting uses (DELI system)
  - Emulation of novel architectures
  - Software sandboxing
  - Software verification
Next Time

Reading
- [Padua & Wolfe 86]

Lecture
- Parallelism and locality