The Semester in Context

Last time
- Dynamic optimizations with DyC

Today
- Dynamo
- DELI
- The big picture

Dynamic Optimization in Dynamo [Bala, et al. 2000]

Modern systems create obstacles for static compilers
- Objects, small procedures, dynamically linked libraries

Idea
- Dynamically optimize native binaries
- Transparent to the programmer
  - No annotations

Mechanism
- Interpret instructions \{ Slow
  - Identify hot traces
- Generate native code for traces
  - Optimize these traces
- Cache these traces for future use \{ Fast
Dynamo Architecture

instructions

Interpret until taken branch

Lookup branch target in cache

miss

Start-of-trace condition?

no

yes

Jump to fragment in cache

Increment trace counter

no

Counter value exceeds hot threshold?

yes

no

Create and optimize new fragment

End-of-trace condition?

yes

no

Emit and link fragment

Fragment Cache

Dynamo Architecture – Interpretation

instructions

Interpret until taken branch

lookup branch target in cache

miss

Start-of-trace condition?

no

yes

Jump to fragment in cache

Increment trace counter

context switch

no

Counter value exceeds hot threshold?

yes

no

Interpret + codegen until taken branch

Create and optimize new fragment

End-of-trace condition?

yes

no

Emit and link fragment

Fragment Cache

May 6, 2015  Dynamic Optimizations
### Dynamo Architecture – Generate and Optimize Traces

1. Interpret until taken branch
2. Lookup branch target in cache
3. If hit, jump to fragment in cache; else, increment trace counter
4. If counter value exceeds hot threshold, interpret + codegen until taken branch and insert optimized code into Fragment Cache; else, start trace condition?
5. If condition is true, create and optimize new fragment; else, end-of-trace condition?

### Dynamo Architecture – Native Execution

1. Interpret until taken branch
2. Lookup branch target in cache
3. If hit, jump to fragment in cache; else, increment trace counter
4. If counter value exceeds hot threshold, interpret + codegen until taken branch and insert optimized code into Fragment Cache; else, start trace condition?
5. If condition is true, create and optimize new fragment; else, end-of-trace condition?
**Fragment Formation**

- **Instructions**
  - Interpret until taken branch
  - Lookup branch target in cache
  - Start-of-trace condition?

- **Trace**
  - Dynamic sequence of instructions
  - May span program boundaries, including procedure calls, indirect branches (unlike superblocks)

- **Fragment**
  - The native code corresponding to a trace

- **Fragment Cache**
  - Interpret + codegen until taken branch
  - End-of-trace condition?

Identifying Traces

- **Instructions**
  - Interpret until taken branch
  - Lookup branch target in cache
  - Start-of-trace condition?

- **Identifying traces**
  - No expensive path or branch profiling
  - Associate a counter with start-of-trace points (e.g., target of backward taken branches)
  - Trace ends with an end-of-trace condition
Trace Optimization

Create straightline code fragments
- Remove unconditional branches on the trace
- Remove redundancies exposed by straightening
  - Redundant loads and assignments
  - Copy propagation, constant propagation, strength reduction, loop invariant code motion, loop unrolling

Fragment Linking

Fragments are linked in the Fragment Cache
- Reduces number of exits from the Fragment Cache
- For example, if we create a new fragment B-D-G-I-J-E:
## Performance Results

### Methodology
- SPEC int 95
- HP C/C++ commercial compiler

### Baselines
- Compared against three baselines
  - Intraprocedural optimizations
  - Interprocedural optimizations
  - Interprocedural optimizations + profiling

### Results
- About 9% better with intraprocedural optimizations
- About 11% better with interprocedural optimizations
- No improvement for interprocedural optimizations + profiling

---

## Related Work

### Trace caches
- Hardware mechanism for caching sequences of decoded instructions

### rePLay [Fahs, et al, Micro 2001]
- Pure hardware solution

### ICOP [Chou and Shen ISCA 2000]
- Similar to Dynamo but uses a dedicated co-processor to optimize the traces
Dynamo vs. JITs

Isn’t Dynamo just like a JIT?
– Both monitor the executing code to see what’s hot
– Both cache, link, and execute native optimized code

How are the two different?
– Source language
  – JIT: Java bytecode (language-dependent)
  – Dynamo: native binary (machine-dependent)
– Scope
  – JIT: works largely on method-by-method basis
  – Dynamo: dynamic traces of instructions (no language barriers)

Another difference
– What if we change the source language for Dynamo? Binary emulation

DELI: Successor to Dynamo [Micro’02]

DELI: Dynamic Execution Layer Interface
– Generalize the idea of Dynamo
  – Observe every instruction in the running program
  – Provide a new interface for inspecting, modifying, caching, and inserting code

Running program

Program observation

Hardware

BLT (binary level translation)
HAM (hardware abstraction module)
DELI Generalizes a Common Theme

**VMWare**
- Full-system emulation
  - eg. Run x86 code on a Sparc
  - Fast, transparent

**Transmeta**
- Emulates x86 code on a VLIW core
  - Flexible: can change underlying hardware while retaining binary compatibility
  - Fast, transparent

Running program

Emulation

Hardware

DELI Generalizes a Common Theme (cont)

**DELI**
- Can operate transparently as an emulator
- Can also allow native and emulated code to run together
  - Facilitates incremental migration across platforms and product generations
    - e.g. Streaming media application:
      - Emulate GUI and application
      - Execute native MPEG decoder for fast execution
    - e.g. Can emulate OS code for a hardware
  - Can integrate into emulated code software patches as native code
Other Uses of DELI

**Code decompression**
- Can decompress code before emulation

**Security**
- Decryption
- Virus detection
- Sandboxing

Installing Untrusted Code

**Software sandboxing**
- Allow sharing within an address space without compromising safety
  - *e.g.*, Install an untrusted module into the kernel
    - System checks that the untrusted module does not overstep its boundaries
      - Checks branch targets
      - Checks addresses of stores
      - Provides safety with respect to the memory system
Revisiting the question: DELI vs. JITs

Both provide virtualization
- Both provide a fast path and a slow path
- Why? Reduces effort and overhead
- We see two systems principles at work:
  - Use an extra level of indirection (virtualization)
  - Optimize the common case (native execution for the fast path)

The Larger Trend

Success disaster
- Mark Weiser’s vision of ubiquitous computing [1988] is coming true
  - Computing is pervasive
  - Computing is non-invasive
  - Computing is woven into the fabric of our lives
- We rely on huge amounts of hardware and software – whose provenance is unknown – to be correct and secure
- We rely on systems whose complexity is overwhelming

Improving software quality
- Many dimensions to consider . . .
Compatibility

**Binary compatibility**
- As the legacy code base increases, compatibility becomes increasingly important
- Compatibility is a form of quality
- Two ways to get compatibility
  - Conform to an existing standard
  - Emulation

Correctness

**Language trend: “managed code”**
- What is managed code?
  - Bounds checks
  - Sandboxing
  - Garbage collection
  - ...

**Safety through language support**
- Typesafe languages
- Use modern languages and use type theory to prove that all references are safe
Security

Limit access to system resources
- A generalization of software sandboxing
- Can some untrusted code inappropriately access the file system?
  - Files can have different access privileges, \textit{e.g.}, read, write, execute
- Can some untrusted code inappropriately access the network?
  - \textit{e.g.}, The code can access my digital camera but not my microphone

Privacy
- Does privileged information leak to the outside world?

How can compilers help?

Program Checking

Check for partial correctness
- Does the program terminate?
- Does the program use locks correctly?
- Does the program allow information to leak to the outside world?
- ...
Program Checking Techniques

Lexical techniques
- Fast, but very superficial (e.g. Lint)

Type systems
- Example: add tainted type qualifier
- Becoming increasingly sophisticated

Model checking and FSMs
- Precise and detailed analysis
- Suffers from state explosion problem

Formal verification
- Requires full formal specification
- Very expensive and not fully automated

Fault Tolerance

Can we protect a program against transient faults?
- An increasingly important issue as feature sizes shrink
- Can insert code that performs redundant computations and checks for correctness
Program Understanding

Does a program do what it’s supposed to do?
- Might it do something that is improper?
- Static analysis is useful here
- Folks at the NSA worry about these kinds of things

Is one program derived from another?
- MOSS: uses static analyses

Does a program contain malware?
- Early work: do simple pattern matching to identify code fragments of known malware
- Use semantic pattern matching (like MOSS)

Protection Against Reverse Engineering

Goal
- Discourage reverse engineering
- More important with the use of bytecodes, which contain considerable information

Solutions
- Physically restrict access to code
- Encrypt code—tends to limit portability because of special hardware needs
- Code obfuscation
- Make it more costly to reverse engineer a program
Program Obfuscation

Name obfuscation
- Scramble the names of identifiers (eg. C Shroud)

Data obfuscation
- Change the way that data is encoded
  - e.g. Replace \( i \) by \( 8i+3 \)
    ```
    int i = 1;                         int i = 11;
    while (i<1000) {                  while (i<8003) {
        i++;                           i += 8;
    }                                 }
    ```
  - e.g. Add indirection to access array elements
  - Change the organization of data
    - e.g. Convert a 2D array into a 1D array

Program Obfuscation (cont)

Control obfuscation
- Disguise control flow
  - Inline procedures
  - Reverse order of loops
  - Insert irrelevant statements (dead code)
  - Dismantle high level constructs
    - e.g. Java has no `goto` statement, but the `bytecode` does
Program Obfuscation (cont)

Control obfuscation
– Opaque Predicates [Collberg, et al.,’98]
  – Predicates whose values are opaque to static analysis
  – Idea
    – Leverage the complexity of alias analysis and shape analysis
    – Insert code that manipulates nodes of a tree or a graph
    – Maintain invariants about specific pointers into the graph
      – eg. p != q
      – Use comparisons of these pointers as opaque predicates
    
    if (p==q)
    // fake code
    else
    // real code

Code Size

Embedded Code
– Runs on embedded hardware with limited memory
  – Code size is an issue

Two solutions
– Code compression
  – Requires decompression
– Code compaction
  – Produce binaries that are small (yet still executable)
  – Standard optimizations on binary code (CSE, constant propagation…)
  – Code re-factoring
    – Find sequences of common code
    – Put these into new procedures
    – Trades off increased execution time for reduced space
**Concepts**

**Dynamic compilation**
- Runtime constants
- Staged compilation
- Native binary optimization
- Identifying and optimizing commonly executed code fragments

**Many dimensions of software quality**
- Compatibility
- Correctness
- Security
- Opaqueness
- . . .

Next Time

**Final exam**
- 9:00am Saturday May 16th GDC 4.304
- You may bring one 8.5” × 11” page of notes (double-sided)

**Project deadline**
- Sunday May 17th, 5:00pm
- Stay tuned to Piazza for details about presentations