Compiler Optimization Analyses and Transformations

Prof. Farn Wang, Department of Electrical Engineering

Transformations and Analyses

- Transformations
  - e.g. Loop transformations
  - e.g. Instruction Scheduling
  - e.g. Copy propagation
- Analyses
  - Information gathering to assist transformations

e.g. LLVM has 34 analysis passes and 63 transformation passes

Code Analyses for Optimizations

- Control flow analysis
- Data flow analysis
  - UD chain analysis
  - Live variable analysis
  - Available expressions
- Alias analysis
- Dependence analysis
- Pointer analysis (*point-to analysis*)
- Shape analysis (*check memory leak, safety, circular list, etc*)
- Escape analysis (*is a pt escaped from the current scope?*)

Highlights of Compiler Optimizations

- Traditional code generation/optimizations
  - Basic Blocks and Control Flow Graphs
  - Loops and DAGS
  - Local CSE (Common Sub-expression Elimination), DCE
  - Constant propagation, copy propagation
  - Register Allocation
  - Code Scheduling
- Machine Independent Optimizations
  - Data Flow Analysis
  - Induction variables
  - Global CSE and PRE (Partial Redundancy Elimination)
  - Loop optimizations

- ILP (Instruction Level Parallelism): from RISC to EPIC
  - Hardware that benefit from ILP:
    - Superpipeline
    - Superscalar and VLIW processors
  - Software that expose and enhance ILP
    - Global code motion
    - Speculative code motion
    - Trace Scheduling
    - Software Pipelining
    - Superblocks and Hyperblocks

Chapter 15
Profile Guided Optimizations

Prof. Farn Wang

- Profile Guided and Adaptive Optimizations
  - Profile Gathering: Instrumentation *vs.* PMU (performance-monitoring unit) based
  - Profile consumers:

    e.g. In-lining and Code layout optimizations

    e.g. Code motion and cache prefetching optimizations

    e.g. Value prediction, register promotion
  - JIT and Adaptive optimizations

    e.g. Java HotSpot, Google Chrome (Chrome 55)
  - Compilers implementing PGO

    Intel C++, GCC, Oracle Solaris Studio, MS C++,

    MS Visual Studio Tools, LLVM/Clang

PBO optimized Chrome

- Google Chrome 55 (2016) is built with PBO optimizations
  - New tab page load time: 14.8% faster
  - Page load (time to first paint): 5.9% faster
  - Startup time: 16.8% faster
- Major optimizations:
  - Functions that are used more often will be optimized for speed while functions rarely used will be optimized for code size.
  - Code layout to improve locality

    rarely used functions are moved away

    infrequently used paths are moved away

- TLP (Thread-Level Parallelism) and DLP (data-level parallelism)

  (*MIMD vs. SIMD*)

  - Parallelism *vs.* Locality
  - Loop Transformations
  - Optimization for GPGPU
  - Optimization for HSA Architectures
- MISC related topics
  - DOC: Debugging Optimized Code
  - Finalization in HSA Systems
  - Optimizations for DBT systems
  - Code Obfuscation
  - Machine Learning Guided Compiling Systems

Automatic Parallelization

Automatic Parallelization

- Refers to converting sequential code into multi-threaded or vectorized (or SIMDized) code automatically (i.e. by compilers)
- The goal is to relieve programmers from the tedious and error-prone manual parallelization. This goal remains a grand challenge.
  - **data dependence analysis**
  - **alias analysis**
  - **effective parallelism exploitation granularity**
  - **input data dependence**
- So far programmer's involvements is more successful, although not desirable. (will ML help? If so, in what way?)
  - OpenMP
  - OpenCL

DLP and Parallelizable Loop

```
for (i=0; i< 100; i++) {
        Z[i] = sin(A[i]);
}
```

The 100 assignments can be executed in parallel.

- For shared memory SMP, A[] and Z[] array could be shared
- For distributed memory systems, each array element preferably be allocated in the local memory.

Granularity of Parallelism

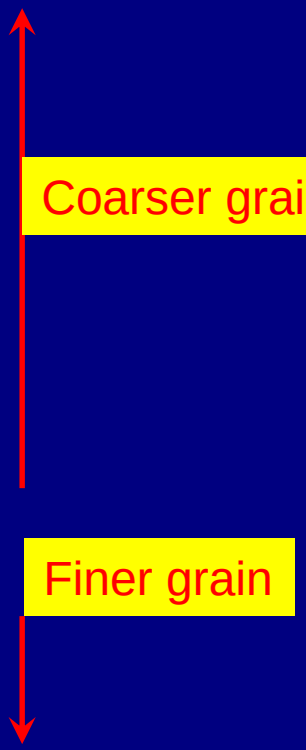
- Coarse-grain parallel processing

  e.g. SETI (Search for Extra-Terrestrial Intelligence) Project. (*This is an experiment that uses home computers connected over the Internet to analyze different portions of radio telescope data in parallel*).

- Most applications require more communication and interaction between processors.
- Fine-grain parallelism may incur too much sync overhead.
- Anti-social Parallelism: a keynote talk by K. Yelick surveys techniques to avoid communications

Granularity of Parallelism

```
For each program module do {
    for each procedure do {
        scanning
        parsing
        code gen
        code opt {
            for each inst do {}
        }}}
```



We prefer to parallelize the outermost loops in a program for a coarser grain parallelism.

Why not? Hard to prove parallelizable

Data-Dependence Problem

- Two statements are data dependent if they both access the same memory location and at least one of them is a write.
- In the presence of data dependence, the order of execution must be preserved.
- Scalar variables can use data flow analysis
- Array variables must exam subscript expressions for cross-iteration dependences.

Data-Dependence Tests

- Data dependence test for multi-dimensional arrays reduces to determining if a system of linear equations has an integer solution that satisfies a set of linear inequality constraints.
- The variables of the linear equations are loop index variables.
- The inequality constraints arise from loop bounds and direct vector relationship
- For single dimensional arrays, there is only one equation to test
- For multi-dimensional arrays, we check if each loop variable is separable (not showing in other subscripts).
- Separable variables allow each subscript to be test independently

GCD Test

- Equations with the **stipulation** that solutions must be integers are known as Diophantine equations.

```
for (i = 1; i< N; i++) {
        Z[2*i] = b[i]…
        C[i] = Z[4*i+1] …
}
```

If a loop carries dependency exists between X(a*i+b) and X(c*i+d) then GCD(a,c) must divide (d-b).

In the above case, GCD(2,4) = 2, and 2 cannot divide 1,

there is no dependency between Z[2*i] and Z[4*i+1].

GCD test is often used to check whether a solution exist.

GCD Test

Another example,

```
for (i = 1; i< N; i++) {
        Z[7*i+3] = Z[3*i+5]+a; }
```

GCD(7,3) = 1 and 1 can divide 5-3=2.

When i=2, Z[17] will be referenced.

This loop is not parallel.

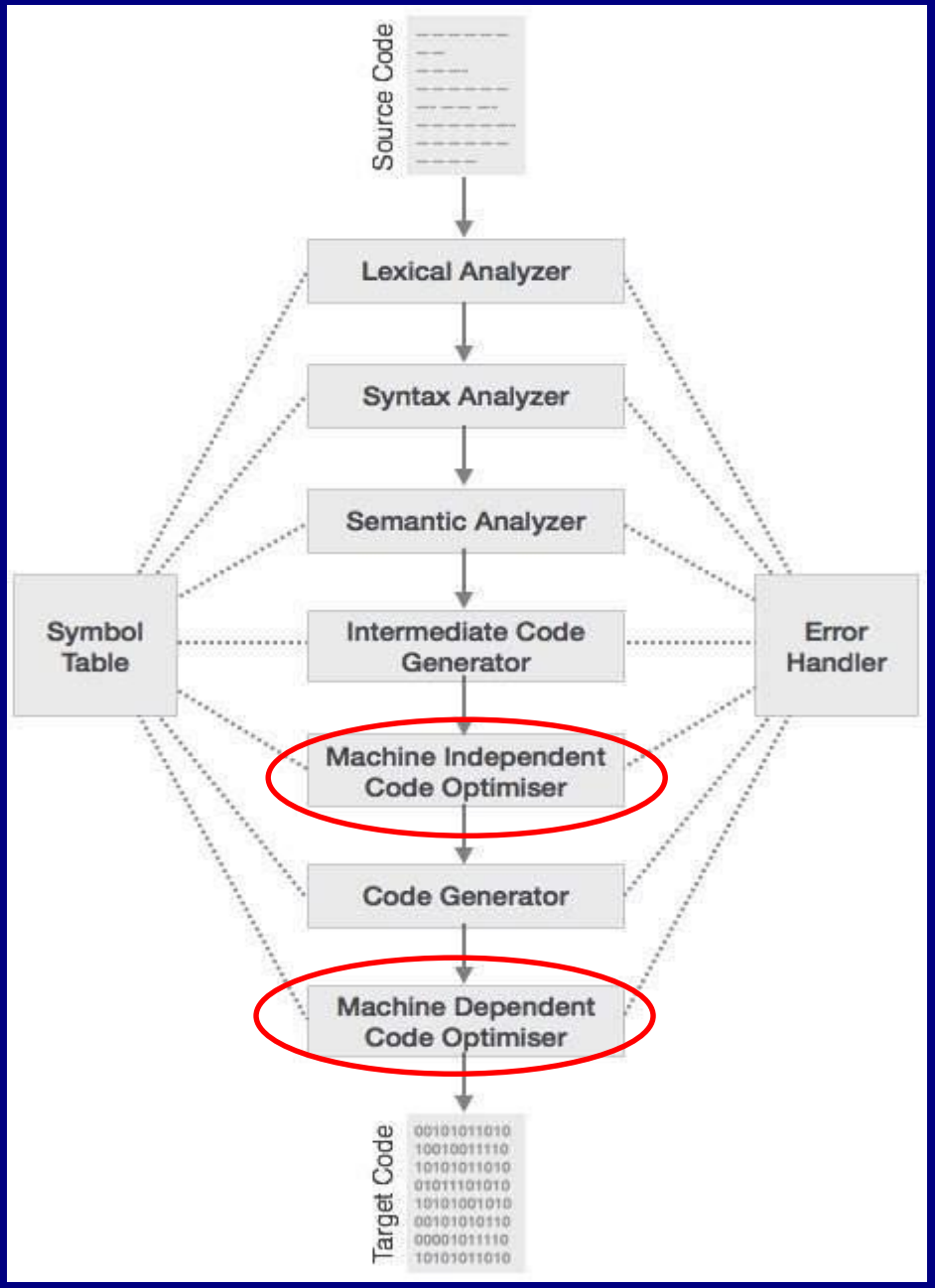
Basic Steps for testing DP

1) Apply GCD test (Greatest Common Divisor) test, which checks if there is an integer solution to the inequalities, using the theory of Diophantine equations. If there are no solutions, there are no data dependence.

2) Using a set of heuristics to handle the large number of typical inequalities.

3) When heuristics do not work, integer linear programming may be used.

Machine Independent Optimizations


Source Code
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Symbol Table
Intermediate Code Generator
Error Handler
Machine Independent Code Optimiser
Code Generator
Machine Dependent Code Optimiser
Target Code

Machine Independent Optimizations

- Code Motion

  Reduce frequency with which computation performed if it will always produce same result. Especially moving code out of loop.

```
for (i = m; i < n; i++)
for (j = m; j < i; j++)
        a[n*i + j] = b[j];
```

n*i is loop invariant

- Is this code motion always safe? How about n/i?

  how about a[i]/a[k], can we move 1/a[k]?
- Is this code motion always profitable?

Machine Independent Optimizations

- Strength Reduction

  16xA → A << 4

  A/16 → A >> 4 ? Is this correct?

  Ax19 → A << 4 + A << 1 + A ? Is this faster ?

- Common Sub-Expression Elimination

  (a + b) ….. =(a + b) → t1 = (a + b), ….. = t1

  What if target machine has very few registers?

- Making use of registers

  use pseudo registers

Machine Independent Optimizations

- Redundancy Elimination
- CSE/DCE
- LICM
- In-lining
- Strength reduction
- Jump elimination/Branch folding
- Loop fusion/loop distribution/loop unrolling
- …. Many machine independent optimizations are now machine dependent. *RISC,VLIW , Multi-core and GPGPU raised the need for more advanced compiler optimizations.*

Data Cache Prefetching

Introduction

- Gap between processor and memory is increasing

  This is traditional "move data to computation" thinking.

  However, some recent researches argue for "move computation to data" or "near data computing"
- One way is to initiate prefetches ahead of the actual request so that data can be brought to some storages closer to the processor.
- Prefetches can be inserted by hardware or by software, and software prefetches can be performed statically or dynamically.

Pros and Cons of HW vs SW Prefetching?

Metrics for Prefetching Effectiveness

- Coverage

  the fraction of original cache misses that are prefetched

- Accuracy

  the fraction of prefetches that are useful

- Timeliness

  whether the prefetch can fully hide the cache miss latency.

Metrics for Prefetching Effectiveness

- Coverage

  the fraction of original cache misses that are prefecthed. A high coverage could eliminate most of the misses.

- Accuracy

  the fraction of prefetches that are useful. Accurate prefetches avoid increasing memory traffic that consume precious memory bandwidth.

- Timeliness

  whether the prefetch can fully hide the cache miss latency. Timely prefetches are more desirable.

Challenges

- Coverage

  Aggressive prefetching may issue many prefetches, and achieve high coverage, but may not be accurate

- Accuracy

  Conservative prefetching may be more accurate, but yield a low coverage

- Timeliness

  If a prefetch is initiated too early, it may pollute the cache, or even be replaced before being referenced.

  If initiated too late, it may not hide the full miss latency

Software Data Prefetching

- Architectural Support
  - ISA support: A prefetch instruction

Can we use a regular load for cache prefetch ?

  - Micro-architectural support

Software Data Prefetching

- Architectural Support
  - ISA support: A prefetch instruction
    - Non-binding
    - No side effects (no traps, no page faults)
    - Speculative in nature, with no impact on correctness
    - Like a hint – can be ignored
    - Prefetch for read/write
    - Block prefetch?
    - Prefetch to which cache hierarchy?

Software Data Prefetching

- Architectural Support
  - Micro-architectural support
    - Lockup-free (non-blocking) caches, with MSHR (Miss Status and Handling Register) to support MLP
    - Useless prefetch detection
    - Profiling support (miss instruction address, miss address, miss latency, … etc)
    - Prefetch buffer (optional)

Practical Data Prefetching

- Locality Analysis
  - Hard since iteration space and the size of active data space are usually unknown
  - Conditional statements complicate the analysis
  - Often leave to users to enable or disable prefetch
  - When enabled, prefetches inserted for every array
  - Profile guided prefetch has sometimes been used
  - Aggressive prefetching is default more often
  - Unrolling is often performed to minimize prefetches
  - Prefetching for the first few iterations is not done
- Prefetch Scheduling
  - This has been adopted

Software Data Prefetching

- Stride based
  - Unit stride
  - Non-unit strides
- Indirect (Indexed) Array
  - A[B[i]] often used for Sparse matrix
  - Both regular prefetch and data prefetching are used
  - Prefetch distance may need to be longer
  - (both B[] and A[] may miss)
- MLP: Memory Level Parallelism, what matters is how many prefetches triggered misses can be overlapped.

Software Data Prefetching

Pointer Prefetching

Dynamically managed data structures such as Linked list Data Structures (**LDS**), trees, graphs exhibit poor locality. Since accesses are often serialized, initiating early cache prefetching is difficult.

Two techniques have been used:

Natural pointers: for LDS

Jump pointers: created specifically for cache prefetching

Natural Pointer Prefetching

```
Struct node {data, next}
*ptr, *list_head;
ptr = list_head;
while (ptr) {
        prefetch(ptr->next);
        ....
        ptr=ptr->next;
}
```

```
Struct node {data, left, right}
*ptr;
void recurse(ptr) {
        prefetch(ptr->left);
        prefetch(ptr->right);
        ....
        recurse(ptr->left);
        recurse(ptr->right);
}
```

Greedy Prefetch

Natural Pointer Prefetching

Data Linearization

- If the LDS structures are laid out linearly in memory, then stride-based prefetching could be used.
- How to achieve linearized layout?
  - at creation time

    more desirable

    must maintain separate memory pool for different data structures.
  - re-layout

    requires data copy at runtime, incurs more overhead.

Jump Pointer Prefetching

Jump pointer approach insert special pointers into the LDS, called jump pointers, for the sole purpose of data prefetching.

```
Struct node {data, next, jump}
*ptr, *list_head, *prefetch_array[PD], *history[PD]
for (i=0; i<PD; i++)                    // prologue loop
        prefetch(prefetch_array[i]);

ptr = list_head;
while (ptr->next) {                     // steady state loop
        prefetch(ptr->jump);
        ….
        ptr=ptr->next;
}
Prefetch_array and history are filled in separate loops
```

Jump Pointer Prefetching