Compiler Optimization

Jordan Bradshaw

Outline

- Overview
- Goals and Considerations
  - Scope
  - Language and Machine
- Techniques
  - General
  - Data Flow
  - Loop
  - Code Generation
- Problems
- Future Work

References

- Watt, David, and Deryck Brown. *Programming language processors in Java*. Pearson Education, 2000. 346-352. Print.
- "Compiler Optimization." *Wikipedia*. Wikimedia Foundation, 25 04 2010. Web. 25 Apr 2010. <http://en.wikipedia.org/wiki/Compiler_optimization>

Compiler Optimization

Compiler optimization is the process of generating executable code tuned to a specific goal. It is *not* generating 'optimal code'.

- Goals:
  - Speed
  - Memory Usage
  - Power Efficiency
- Difficulties
  - Multiple ways to solve each problem
  - Hardware architectures vary

Overview

- Compilers try to improve code by:
  - Reducing code
  - Reducing branches
  - Improving locality
  - Improving parallel execution (pipelining)
- How this is done depends on:
  - The language being optimized
  - The target machine
  - The goal to optimize toward

Optimization Considerations

Goals of Optimization

- Speed:
  - Most obvious goal
  - Try to reduce run time of code
- Memory Usage:
  - Also common (esp. in embedded systems)
  - Try to reduce code size, cache misses, etc.
- Power:
  - More common recently
  - Useful for embedded systems
- Other:
  - Debugging?

Scopes of Optimization

- Peephole Optimization – Replace sequences of generated instructions with simpler series
- Local Optimization – Perform optimizations within a code block / function
- Global Optimization – Perform optimizations on entire program
- Loop Optimization – Perform optimizations to improve loop performance

Language and Machine

Optimizations can be performed in a language or machine dependent or independent manner.

- Language
  - Many languages share features
  - Language constructs (such as pointers) can make this hard
- Machine
  - General optimizations work on many architectures
  - Knowledge of the machine gives better benefits.

Machine Considerations

- Target machines can vary *a lot:*
  - Registers
  - Pipeline structure
  - Execution units
  - Available instructions
  - Cache
- A good compiler will try to take advantage of as many of these as possible
- Or, it may try to balance these for a set of architectures (i.e. AMD and Intel)

Optimization Techniques

Optimization (General)

- Some optimizations can be done on code found anywhere in the program
- Constant folding – Perform as much arithmetic as possible

3/2 => 1.5 (4/3) * 10 + 0.2f => 13.53
((4/3) * i) / 2 => 0.66 * i

- Subexpression elimination

(2.4 * i) + (2.4 * j) => 2.4 * (i + j)

Optimization (Peep Hole)

- Examine a series of instructions and try to reduce it to a simpler set
- Alternatively, replace individual instructions with more suitable ones
  - Shifting can be more efficient than multiplication or division
  - The XOR trick
- Doesn't depend much on the global information

Optimization (Program Flow)

- Jumps and calls should be avoided
- Functions can be inlined to avoid a call
- if/else
  - If compiler can guess result, generate code to minimize jumps
  - Use lazy conditionals
  - Try simple conditions first

```
if (variable || callFunction())

not

if (callFunction() || variable)
```

Optimization (Program Flow)

- Dead code can be removed

```
if (true)
        doThis()
else
        doThat();
```

```
if (true)
        doThis()
```

- Invariant code can be factored out

```
if (variable)
{
        var1 = 0;
        doThis();
}
else
{
        var1 = 0;
        doThat();
}
```

```
var1 = 0;
if (variable)
{
        doThis();
}
else
{
        doThat();
}
```

Optimization (Loops)

- Loops take up most of the program's time
- Should get most of the attention
- A *lot* of ways to optimize loops
  - Induction analysis

```
for (i = 0; i < 10; i++)
{
        doSomething(); j++;
}
```

```
j += i;
```

  - Invariant analysis – Values that are the same each loop can be factored out

Optimization (Loops)

- Loop fission – Break loop up to improve locality
- Loop fusion – Combine loops operating over same range

```
for (I = 0; I < 10; i++)
{
        doSomething();
}

for (I = 0; I < 10; i++)
{
        doSomethingElse();
}
```

```
for (I = 0; I < 10; i++)
{
        doSomething();
        doSomethingElse();
}
```

Optimization (Loops)

- Loop interchange
  - Swap nested loops
  - Can improve memory locality

```
for (i = 0; i < 10; i++)
{
	for (j = 0; j < 10; j++)
	{
		// Notice indices are backwards
		array[j][i] = i * j;
	}
}
```

Optimization (Loops)

- Loop unrolling
  - Decrease loop overhead
  - Increase code size
  - Helps to know loop range

```
for (i = 0; i < 10; i++)
{
	doSomething();
}
```

```
for (i = 0; i < 5; i++)
{
	doSomething();
	doSomething();
}
```

```
for (i = 0; i < j; i++)
{
	// How many times to unroll this loop?
	doSomething();
}
```

Optimization (Loops)

- Loop splitting:
  - Break different cases of a loop up
  - Different from fission

```
for (i = 0; i < 10; i++)
{
        if (i ==0)
                doSomething();
        else
                doSomethingElse();
}

doSomething();
for (i = 1; i < 10; i++)
{
        doSomethingElse();
}
```

Optimization (Loops)

- Other optimizations:
  - Pipelining: execute code over multiple iterations to improve pipelining
  - Parallelization: execute iterations on multiple processors / execution units
  - Inversion: convert while to do-while, may reduce jumps
  - Reversal: execute code in reverse order, may improve dependencies

Optimization (Code Gen)

- Instructions should be executed in an order that minimizes stalls
- Use instructions that do more if possible (vector math, madd)
- Allocate as much memory to registers as possible (difficult!)
- Factor out redundant code where possible

Problems with Optimization

Optimization Problems

- Compilers must balance goals from before
- Compilers must balance performance across multiple architectures
- Compilers don't understand what you're programming
  - Optimization can't find a better algorithm to solve your problem
- No optimization is actually optimal
- Optimization can be slow

Sample Data

Sample Data

- Program continuously multiplies and divides a number by 2, for 10 seconds

Optimization	Iterations	Program Size
None	145941324	27KB
Size	188573683	26KB
Speed	201574135	25KB

Future Work

Future Work

- Optimization can *always* get better
  - The original Fortran compiler writers had to work hard to win over assembler coders
  - Programmers expect no less than perfection from the compiler
- New hardware / software demands new techniques be developed
  - Multi core optimization
  - More registers / memory models
  - Hot Spot optimization

Questions?