Optimising Code

showing that compilers *do* produce good code...

Des Watson

Department of Informatics, University of Sussex

November 2008

1 The Need for Good Code
- Motivation
- Programming languages

2 Optimising Compilers
- Compiler structure
- Approaches to optimisation
- Machine independent optimisation
- Machine dependent optimisation
- Embedded processors
- Real compiler example

3 How Far Can We Go?
- How well are we doing?
- How well can we do?

Goals

What are we trying to optimise? What is high-quality code?

- Execution speed
- Code size
- Data size
- Power consumption
- etc.
- Or any combination?

Why do we bother? And what is “optimal”?

Why bother?

Sometimes optimisation is critical, but often it doesn't matter.

- Speed – real time constraints
- Code size – cost of memory
- Data size – cost of memory
- Power consumption – battery-powered devices

Hence optimisation seems to have a particular relevance to embedded system implementation.

High-level vs low-level languages

One potential approach is to write in assembler, or equivalent.
Then it's up to the skill of the coder to produce good code.
But...

- Software can be complex
- Software engineering issues – reliability too
- It's hard work
- Do we *really* understand the target machine?

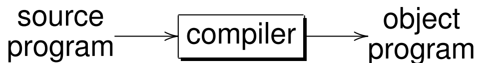
Use a high-level language

So, we should probably stick with high-level languages.

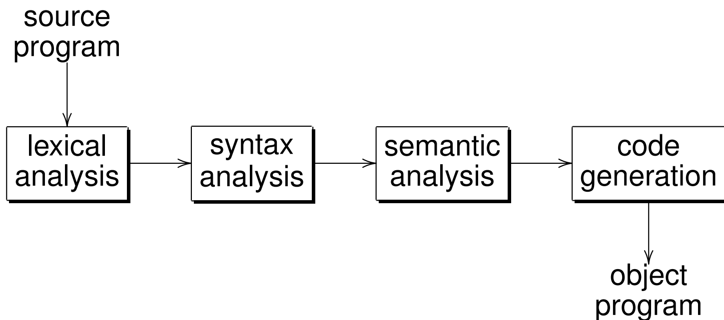
- Would compiled code do the job?
- Are compilers “clever” enough?
- Are some high-level languages better than others for the generation of good code?
- How can we make use of neat hardware tricks?
- What about Java? The Java interpretation issue still gives many organisations sleepless nights.

We need to look at the compilation process and see how compilers can produce good code.

What does a compiler look like?

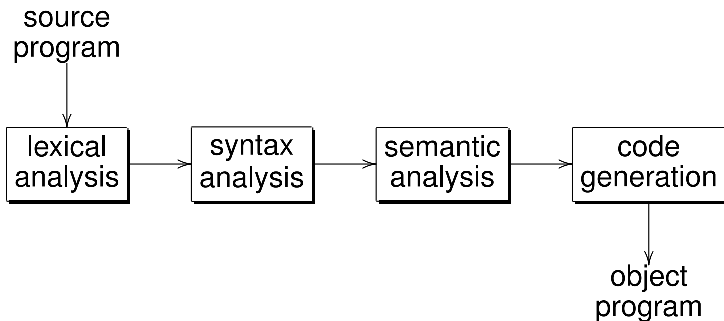


What does a compiler *really* look like?



And where do we optimise?

Look primarily at the interfaces:



- Semantic analysis to code generation – intermediate representation
- Within code generation
- Postpass on generated code

Beware – we have some constraints...

- Code optimisation shouldn't alter the meaning of a program
- It ought not result in an excessive compile-time overhead
- There may be a conflict between space and time optimisation
- We *may* want to make use of the observation that during execution 90% of the time will be spent executing 10% of the code
- We know about many optimisation techniques, but they often interact (unpredictably) with each other
- Aggressive optimisation is *very hard*.

Types of optimisation

Machine independent – applied *before* code generation – loosely concerned with algorithm changes.

Machine dependent – applied *during* or *after* code generation – making use of specific target machine capabilities.

Also, contrast **local** vs **whole-program** optimisation.

Example – common sub-expressions

Largely concerned with removal of redundant code
Data flow analysis as well as control flow analysis.

```
x = a*a / (a*a + b*b);
y = b*b / (b*b + a*a);
```

to:

```
t1 = a*a;
t2 = b*b;
t3 = t1 + t2;
x = t1/t3;
y = t2/t3;
```

This common sub-expression removal is easy to do and can be very effective.

Example – constant folding and propagation

Another easy one:

```
x = 3 + 2;
```

transformed to:

```
x = 5;
```

or more interesting:

```
a = 2;
b = a + 1;
x = a + b;
```

transformed to:

```
a = 2;
b = 3;
x = 5;
```

Example – alias analysis

We know about this one:

```
x = a*a / (a*a + b*b);
y = b*b / (b*b + a*a);
```

and introducing p makes little difference.

```
x = a*a / (a*a + b*b);
p = 3;
y = b*b / (b*b + a*a);
```

Example – alias analysis

We know about this one:

```
x = a*a / (a*a + b*b);
y = b*b / (b*b + a*a);
```

and introducing p makes little difference.

```
x = a*a / (a*a + b*b);
p = 3;
y = b*b / (b*b + a*a);
```

But what about:

```
x = a*a / (a*a + b*b);
*p = 3;
y = b*b / (b*b + a*a);
```

Loops

- Concentrate on loops
- Concentrate *really hard* on nested loops
- Take loop invariants out of the loop
- Loop fusion
- Loop unrolling
- Neat tricks with loop rearrangement to maximise use of the cache (machine dependent...)
- etc.

Machine dependent techniques

- Good register allocation
- Instruction selection
- Instruction scheduling
- Fancy addressing modes
- Peephole optimisation – as a postpass
- Extensive use of parallelism
- etc.

Embedded processor characteristics

- Hardware design leans towards specialised or particular applications, sometimes with a wild instruction set (e.g. DSP)
- Usually cheap to manufacture (large volume devices)
- Maybe not particularly powerful (in comparative terms)
- Integrated with support electronics (on chip) to reduce hardware complexity of final system
- ROM, RAM built in
- Potential for high reliability, low power consumption, etc.
- There are lots of them about – it's a huge market.

Parallelism

This is the key to really effective optimisation.
Instruction-level parallelism all the way to distributed systems.
It would be nice to have a compiler that supported the automated implementation of software on multicore processors.

GCC-generated code – source program

```
main()
{
  int k=0;
  int i;
  int a[100];

  for (i=0;i<100;i++) {
    a[i]=0;
    if (a[i]==0) k++;
  }
  printf("%d\n",k);
}
```

GCC-generated code – no optimisation

```
main:
.LFB2:
        pushq   %rbp
.LCFI0:
        movq    %rsp, %rbp
.LCFI1:
        subq    $416, %rsp
.LCFI2:
        movl    $0, -8(%rbp)
        movl    $0, -4(%rbp)
        jmp     .L2
.L3:
        movl    -4(%rbp), %eax
        cltq
        movl    $0, -416(%rbp,%rax,4)
        movl    -4(%rbp), %eax
        cltq
        movl    -416(%rbp,%rax,4), %eax
        testl   %eax, %eax
        jne     .L4
        addl    $1, -8(%rbp)
.L4:
        addl    $1, -4(%rbp)
.L2:
        cmpl    $99, -4(%rbp)
        jle     .L3
        movl    -8(%rbp), %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        leave
        ret
```

GCC-generated code – with optimisation

```
  ...
  for (i=0;i<100;i++) {
    a[i]=0;
    if (a[i]==0) k++;
  }
  printf("%d\n",k);
}

main:
.LFB13:
        movl    $100, %esi
        movl    $.LC0, %edi
        xorl    %eax, %eax
        jmp     printf
```

What’s been done already?

- The pressure was on from the earliest days of compilers

What’s been done already?

- The pressure was on from the earliest days of compilers
- Compiler design has matured

What's been done already?

- The pressure was on from the earliest days of compilers
- Compiler design has matured
- We know which high-level language constructs are hard for the compiler

What's been done already?

- The pressure was on from the earliest days of compilers
- Compiler design has matured
- We know which high-level language constructs are hard for the compiler
- We know quite a lot about the design of hardware to support high-level languages

What's been done already?

- The pressure was on from the earliest days of compilers
- Compiler design has matured
- We know which high-level language constructs are hard for the compiler
- We know quite a lot about the design of hardware to support high-level languages
- We know quite a lot about the design of algorithms to support speed-enhancing hardware

What's been done already?

- The pressure was on from the earliest days of compilers
- Compiler design has matured
- We know which high-level language constructs are hard for the compiler
- We know quite a lot about the design of hardware to support high-level languages
- We know quite a lot about the design of algorithms to support speed-enhancing hardware
- We have access to *many* optimisation techniques

What's been done already?

- The pressure was on from the earliest days of compilers
- Compiler design has matured
- We know which high-level language constructs are hard for the compiler
- We know quite a lot about the design of hardware to support high-level languages
- We know quite a lot about the design of algorithms to support speed-enhancing hardware
- We have access to *many* optimisation techniques
- We can do some pretty neat optimisations

What's left to do?

Thankfully, the list is long!

- High-level optimisation – optimisation "in the large"

What’s left to do?

Thankfully, the list is long!

- High-level optimisation – optimisation "in the large"
- Parallel hardware

What's left to do?

Thankfully, the list is long!

- High-level optimisation – optimisation "in the large"
- Parallel hardware
- Intermediate representations

What's left to do?

Thankfully, the list is long!

- High-level optimisation – optimisation "in the large"
- Parallel hardware
- Intermediate representations
- Better flow analysis

What's left to do?

Thankfully, the list is long!

- High-level optimisation – optimisation "in the large"
- Parallel hardware
- Intermediate representations
- Better flow analysis
- Automated generation of code generators/optimisers

What's left to do?

Thankfully, the list is long!

- High-level optimisation – optimisation "in the large"
- Parallel hardware
- Intermediate representations
- Better flow analysis
- Automated generation of code generators/optimisers
- Proving correctness

What's left to do?

Thankfully, the list is long!

- High-level optimisation – optimisation "in the large"
- Parallel hardware
- Intermediate representations
- Better flow analysis
- Automated generation of code generators/optimisers
- Proving correctness
- Developing new processor designs, particularly for compact or power-conscious systems

What's left to do?

Thankfully, the list is long!

- High-level optimisation – optimisation "in the large"
- Parallel hardware
- Intermediate representations
- Better flow analysis
- Automated generation of code generators/optimisers
- Proving correctness
- Developing new processor designs, particularly for compact or power-conscious systems
- Relaxing the requirement for fast compilation (NP-complete problems)

What's left to do?

Thankfully, the list is long!

- High-level optimisation – optimisation "in the large"
- Parallel hardware
- Intermediate representations
- Better flow analysis
- Automated generation of code generators/optimisers
- Proving correctness
- Developing new processor designs, particularly for compact or power-conscious systems
- Relaxing the requirement for fast compilation (NP-complete problems)
- Better understanding of the interactions between optimisations
- etc.

How much further is there to go?

Thinking *quantitatively* is helpful.

- Compare execution times of programs with and without optimisation
- How much better will compilers be in 5 years?
- How much faster will processors be in 5 years?
- What happens if we have plenty of time to do the compilation?

Searching for optimised code

An interesting question – can we ever produce code that is truly *optimal*?

Searching for optimised code

An interesting question – can we ever produce code that is truly *optimal*?
Answer – yes, sometimes, by exhaustive search. This is the *superoptimizer*.
For example:

```
int signum(int x) {
  if (x>0) return 1;
  else if (x<0) return -1;
  else return 0;
}
```

The superoptimizer

For the 68000 target, average compilers can do it in 9 instructions.

The superoptimizer

For the 68000 target, average compilers can do it in 9 instructions.
8 instructions are easy for hand-written code.

The superoptimizer

For the 68000 target, average compilers can do it in 9 instructions.
8 instructions are easy for hand-written code.
Clever programmers can do it in 6 instructions.

The superoptimizer

For the 68000 target, average compilers can do it in 9 instructions.
8 instructions are easy for hand-written code.
Clever programmers can do it in 6 instructions.
Superoptimizer:

```
add.l     d0,d0
subx.l    d1,d1
negx.l    d0
addx.l    d1,d1
```

x input in d0, signum placed in d1.
Superoptimizer: A look at the smallest program, Henry Massalin, ACM Sigplan Notices, 22:10, October 1987.

Summary

- We need code optimisation – particularly for embedded systems
- Compilers are already pretty good at optimisation
- But there's a lot left to do, particularly with parallel systems

What's the problem?

We need to be able to generate compact code for our embedded processor from (say) C source code:

- Say 32 or 64 bit – very small processors (e.g. 4-bit) have interesting and special problems
- Ignore (for now) fancy hardware characteristics
- Ignore (for now) multicore architectures
- Largely ignore (for now) Java implementations – they have interesting and special problems too
- But what are we optimising? "Space" means lots of different things
- Does speed matter?

An easy way forward

All we need is an *optimising* compiler – space optimisation is usually available in production compilers.

- Use existing (conventional) compiler technology
- Usual optimisations – good flow analysis, removal of redundant code, good register allocation, etc.
- Careful instruction selection, aggressive peephole optimisation, interprocedural, intermodule optimisation, etc.
- Don't inline, don't loop unroll
- Job done – maybe not??

Practicalities

Suppose the code generated by our favourite compiler, with all the right options just isn't good enough.
Then:

- Write a better compiler (not recommended for quick results)
- Find out ways in which the code generated by our existing compiler can be improved.
- Then start hacking.
  - Hack compiler
  - Write some sort of prepass for the source code
  - Write some sort of postpass for the generated code

A specific task – prepass processing

Development of a space-optimising compiler for a particular embedded processor.
Translate:

```
if (x >= 1) { ... }
```

to:

```
if (x > 0) { ... }
```

etc.

Postpass optimisation – process the generated code

- Loosely based on peephole optimisation
- Find common code sequences – common code elimination at the source level is *hard*
- Use of “subroutines”
- Overhead of subroutine calls? Use of registers in subroutine calls?
- What about code sequences identical apart from registers used?
- Use of “unallocated” instructions
- Leading to interpretation??