Code Optimization

- Introduction
  - What is code optimization
  - Processor development
  - Memory development
  - Software design
  - Algorithmic complexity
  - What to optimize
  - How much can we win

1

What is code optimization?

- To design programs so that they can be efficiently executed on a processor
  - use the resources of the processor in an efficient way
- In practice it is impossible to achieve *optimal* performance
  - but we can design computer programs so that they become more (or less) efficient
  - use programming constructs that can be efficiently executed on the processor
- Performance should be a concern in all stages of the development
  - from the choice of solution method to the executable program
  - easiest to improve the performance of a program in the early stages of design (at the highest level of abstraction)

2

Theorethical peak performance

- The maximal number of instructions a processor can execute under ideal conditions
- Example:
  - a processor with different functional units for addition and multiplication
  - can do one addition and one multiplication in a clock cycle
  - cycle time 5 ns = 200 MHz
  - max performance is 400 M operations per second
- Assumptions
  - infinite stream of additions and multiplications
  - operations are independent
  - no other instructions (no branches)
  - data can be accessed immediately without delays

3

Intel processor development

Processor	Year	MHz	Transistors	Addr. space	Cache
8086	1978	8	29 K	1 MB	No
286	1982	12.5	34 K	16 MB	No
386 DX	1985	20	275 K	4 GB	No
486 DX	1989	25	1.2 M	4 GB	8 KB L1
Pentium	1993	60	3.1 M	4 GB	16 KB L1
Pentium Pro P6	1995	200	5.5 M	64 GB	16 KB L1 256 KB / 512 KB L2
Pentium II	1997	266	7 M	64 GB	32 KB L1 256 KB / 512 KB L2
Pentium III	Feb 1999	500	8.2 M	64 GB	32 KB L1 512 KB L2
Pentium III	Oct 1999	700	28 M	64 GB	32KB L1 256 KB L2
Pentium 4 NetBurst	2000	1500	42 M	64 GB	12 K µop 8 KB L1 256 KB L2

4

Processor development

- Moore's law
  - number of transistors on a silicon die doubles every 18 months
  - means also that performance doubles every 18 months
- Number of transistors on a die
  - from 29000 to 42 000 000 = 1448 times more
- Clock rate
  - from 8 MHz to 1500 MHz in 22 years = 187 times faster
- Memory size
  - from 640 KB to 256 MB = 409 times more
- But memory access time has only decreased by 10–20 times

5

Processor development (cont.)

- Microprocessor performance develops much faster than the clock rate
  - the improved performance comes mainly from development in microprocessor architecture
  - not so much from higher clock frequencies
- Much more efficient instruction execution
  - RISC architecture
  - instruction pipelining
  - superscalar instruction execution (instruction level parallelism)
  - out-of-order execution (dynamic instruction execution)
  - speculative instruction execution

6

Memory system development

- Memory size has developed about at the same rate as processor performance
- Memory access time has not developed in the same way
  - memory access is slow compared to instruction execution
- Development in processor architecture to improve memory access time
  - multilevel caches
  - instruction pre-fetching
  - write-combining

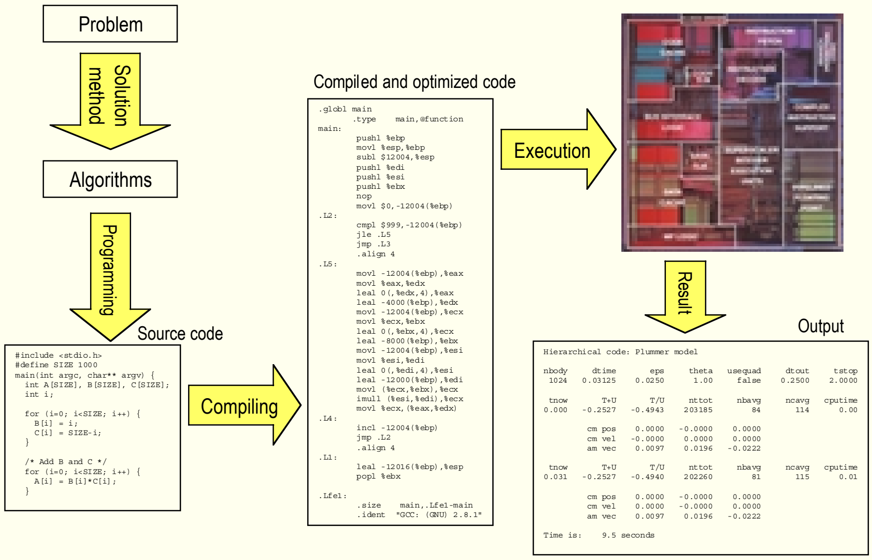
7

Conclusions

- Very fast instruction execution
  - multiple instructions executed each clock cycle
  - instructions do not have to be executed in program order
- Slow memory access
  - processor cycle is normally much faster than the bus cycle
  - only data in registers and cache can be accessed without delay
- Cache memories are small (32 + 32 KB L1, 1 MB L2)
  - for large problems, data will not not fit into cache
- Performance of a program depends strongly on
  - how well the program instructions can use the functional units of the processor
  - how efficiently the processor can access data in memory

8

Software design



9

Choosing a solution method

- A problem can typically be solved in many different ways
  - we have to choose a correct and efficient solution method
- A solution may include many different stages of computation using different algorithms
  - *Example*: sorting, matrix multiplication, ...
- Each stage in the solution may operate on the same data
  - the data representation should be well suited for all the stages of the computation
  - different stages in the solution may have conflicting reqirements on how data is represented

10

Choosing an algorithm

- A specific problem can typically be solved using a number of different algorithms
- The algorithm has to
  - be correct
  - give the required numerical accuracy
  - be efficient, both with respect to execution time and use of memory
  - be possible to implement within the time frame of the project
- We can use algorithm analysis to estimate the running time and memory requirements of an algorithm
  - tells us how the running time of an algorithm grows when the problem size increases

11

Algorithmic complexity

- Big-Oh notation
  - $T(N) = O(f(n))$ if there are positive constants c and n_0 such that $T(N) \leq c\, f(N)$ when $N \geq n_0$
  - N is the size of the problem to be solved
- Establishes a relative order among the rates of growth of functions
- *Example:* $T(N) = O(N^2)$
  - $T(N)$ is the time to solve a problem of size N
  - for sufficiently large problems, the computation time grows slower than N^2 multiplied with a constant factor c
- Gives an upper bound on the running time

c	Constant
$\log N$	Logarithmic
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

12

Growth rate

- Examples of growth rate for a few typical functions

Function	N=10	N=50	N=100	N=500	N=1000
log N	3.2	5.6	6.6	8.9	9.9
N	10	50	100	500	1000
N log N	32	280	660	4450	9900
N^2	100	2500	10 000	250 000	10^6
N^3	1000	125 000	1 000 000	125 000 000	10^9
2^N	1024	$1.13*10^{15}$	$1.27*10^{30}$	$3.27*10^{150}$	$3.07*10^{301}$

- To compute 10^{15} operations on a 100 MFlop/s processor takes about 130 days
  - to compute 10^{16} operations would take over 3.5 years

13

Constant factors

- Constant factors and low-order terms are ignored in algorithm analysis
  - if the running time depends on the problem size as $2N^2 + 5N$ the complexity of the algorithm is $O(N^2)$
- Lower order terms and constant factors are also important when choosing an algorithm to solve a specific problem
- Example: two algorithms with complexity $O(N)$ and $O(N^2)$
  - the $O(N)$ algorithm has a constant factor c = 1000
  - the $O(N^2)$ algorithm has a constant factor c = 1
- For problems of size smaller than 1000, the $O(N^2)$ algorithm performs better

14

Choice of algorithm

- Largest improvements in efficiency come from a good choice of algorithm
  - make sure that you know the complexity of the algorithm
  - find alternative algorithms to solve the same problem
  - compare the complexity of the alternatives
  - compare the constant factors in the complexity analysis
  - compare the efforts of implementing the algorithms
- Optimizing an inefficient algorithm will only affect the constant factors of the execution time

15

Programming

- Most often we program in high level languages
  - C, C++, Fortran, Java, ...
- Assembly language is only used for special purposes
  - may be used for small, often executed parts of the code (inner loops)
  - may be used to use features of the processor that are not accessible from a high-level language
- Automatically translated into machine code by a compiler
- Compiler optimization
  - the compiler transforms the program into an equivalent but more efficient program

16

Compiler optimization

- The compiler analyzes the code and tries to apply optimizations to improve its performance
  - recognizes code that can be replaced with equivalent, but more efficient code
- Modern compilers are good at low-level optimization
  - register allocation, instruction reordering, dead code removal, ...
- Avoid using inefficient constructs
- Write simple and well-structured code
  - easier for the compiler to analyze and optimize
- Main issues
  - locality of reference
  - instruction level parallelism
  - special-purpose instructions

17

Program execution

- Modern processors are very complex systems
  - superscalar, superpipelined architecture
  - multi-level cache with pre-fetching
  - rotating registers
  - branch prediction
  - out of order execution
- Difficult to understand exactly how instruction are executed by the processor
- Difficult to understand how different alternative program solutions will affect performance
  - programmers have a weak understanding of what happens when a program is executed

18

What to optimize

- Find out where the program spends its time
  - unnecessary effort to optimize code that is seldom executed
- The 90/10 rule
  - a program spends 90% of its time in 10% of the code
  - look for optimizations in this 10% of the code
- Tools to find out where a program spends its time
  - the time command – user and system time
  - measuring with timer functions in the code
  - profilers – gprof and tcov
  - performance counters

19

How much can we improve a program

- Example: matrix multiplication
  - problem size: 1200 x 1200 single-presicion (float)
- Execution times:
  - no optimization: 405 s
    $O(N^3)$ algorithm from school mathematics, no compiler optimization
  - full compiler optimizations: 80 s
    same algorithm, but with all compiler optimization turned on
  - manually optimized library code: 14 s
    cache blocking, loop unrolling, software pipelining
    compiled with all compiler optimization turned on

20