Code optimization techniques

Francesco Versaci & Alberto Bertoldo

Advanced Computing Group
Dept. of Information Engineering, University of Padova, Italy
`cyberto@dei.unipd.it`

May 19, 2009

The Four Commandments

1. The *Pareto* principle

80% of the effects comes from 20% of the causes

- 80% of the performance is affected by 20% of the code
- 80% of the final performance is obtained by 20% of the efforts
- Do you **really** need/want to spend 4x the time to get 25% speedup?

2. The *Clever Man* principle

Exploit compiler's optimization flags

3. The Occam's razor

Entia non sunt multiplicanda praeter necessitatem[a]

[a]Entities should not be multiplied beyond necessity

4. The *Weel Reinvention* principle

Reuse already optimized code as much as possible

Development tools

Choosing compilers

Obey the *Clever Man* principle

GNU Compiler Collection[a]

[a] `http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html`

-O*n*	enable machine-independent optimizations (e.g. -O3)
-f*x*	manually enable/disable machine-independent optimizations
-m*x*	automatic machine-dependent optimizations

-mtune=cpu-type	restructuring without using a specific ISA
-march=cpu-type	use specific ISA (e.g. SSE2)

Intel Compilers[a]

[a] `http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284264.htm`

-O*n*	enable machine-independent optimizations (e.g. -O3)
-x*t*	automatic machine-dependent optimizations
-parallel	automatic parallelization for multithreaded architectures

Using libraries

Obey the *Weel Reinvention* principle

Math libraries

- BLAS & LAPACK, FFT, math operations, random number generation
- Vendor libraries (IBM ESSL, Intel MKL, AMD CML)
- Generated libraries (ATLAS, Spiral, FFTW, Goto)

Other libraries

- Image and signal processing, string processing, compression, cryptography, etc.
- Vendor libraries (Intel IPP, AMD PL)
- Generated libraries (ATLAS, Spiral, FFTW, Goto)

Template Libraries

- Containers, iterators, algorithms, and functors
- C++ Standard Library
- Boost C++

Other compiler techniques

- Use compiler-specific directives to ease compiler's job (pragmas)
- Use machine-specific instructions
- Use intrinsics

C/C++ optimizations

Floating-point operations

- Use the f or F suffix (for example, 3.14f) to specify a constant value of type float
- Use function prototypes for all functions that accept arguments of type float
- Extract common subexpressions, especially costly ones (e.g. divisions)
- Addition are quicker than multiplications
- Use array notation instead of pointer notation when working with arrays

Conditional instructions

- Make the fall-through path more probable in `if` statements
- Put high-probability case before `switch` statements
- Exploit short circuiting
- Exploit bitwise operators instead of logic operators
- Avoid postincrement and postdecrement in conditions
- Remember that compound conditions are translated into a series of conditional branches
- Use tables indexed on conditions instead of switch or if statements
- Exploit else-clause removal

Function calls
General optimizations

- Avoid function pointers
- Declare a nonmember function as static
- Use the *const* specifier for member functions
- Avoid virtual functions and virtual inheritance
- Fully prototype all functions
- Don’t define a return value if not used

Function calls
Arguments

- Using global variables may prevent local optimizations
- Constant arguments improve optimizations
- Put frequently used parameters in the leftmost positions
- Pass or return a pointer to structure/union/class, or pass it by reference
- Pass non-aggregate types (i.e. `int` and `short`) by value rather than passing by reference
- If you bind functions, use the same order for parameters

Inline functions or macros

Best candidates

- Small size
- Frequently called
- Few callers
- One or more compile-time constant parameters used in `if`, `switch`, `for`
- Perform only a load/store or simple comparisons/operations

Drawbacks

- Increased code size
- Poor I-cache efficiency

Memory management

Dynamic memory

- Minimize the use of malloc
- Use efficient memory managers
- Enforce memory alignment
- Check for memory leaks

Structures

- Declare the largest members first
- Place variables near each other if they are frequently used together
- Adjust structure sizes to power of two
- Sort and pad C and C++ structures to achieve natural alignment

Memory management
Variables

- Prefer local automatic variables
- Declare local variables in the inner most scope
- Sort local variables in decreasing order
- Prefer static global variables to external variables
- Group external variables into structures or arrays
- If a global variable is needed, copy its value to a local variable and use it
- Avoid taking the address of a variable with `&` operator
- Use constants instead of variables, especially as loop bounds
- Use register-sized integers
- Use the smallest floating-point precision

Expressions

- Reduce store-to-load dependencies
- Assign common subexpressions to local automatic variables
- Avoid both explicit and implicit (pay attention!) integer/floating-point conversion
- Code the integer and floating-point arithmetic in separate computations
- Transform division by the same denominator in multiplications
- Avoid exception handling
- Prefer initialization over assignment
- Minimize both implicit and explicit type casting
- Pay attention using volatile and register

64-bit mode

- Use 64-bit mode only if you need to access large data
- Avoid performing mixed 32- and 64-bit operations
- Use `long` for variables which will be frequently accessed

Loops

Compilers already do most loop transformations

Recall the Occam's razor: **your** "clever" optimization may prevent **compiler's** (much more clever) optimizations

- Take constant expressions out of the loop
- Count down instead of up
- Minimize stride
- Consider store VS recompute tradeoffs
- Keep array index expressions as simple as possible
- Keep index type compatible to pointer type
- Simplify loop expressions as much as possible
- If loop expressions ARE complicated, try manual transformations[1]

[1] `http://en.wikipedia.org/wiki/Loop_optimization`

Input/Output

- Use binary streams instead of text streams
- Use the low-level I/O functions, such as `open` and `close`
- Design your own buffering for the low-level functions
- Access multiple of 4K, which is the size of a page