V22.0474-001 Software Engineering
Spring 2008

Lecture 21: Code Optimization I

Clark Barrett, New York University

Outline

- Software Performance
- Code Tuning
- Measurement
- Tools: gprof

Sources for today's lecture:

McConnell, Steve. *Code Complete*, Second Edition, ch. 25.

Software Performance

Quality Characteristics and Performance

Programmers have a tendency to place undue importance on the optimality of their code.

> *More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason–including blind stupidity. –W.A. Wulf (CC2, p. 588)*

Software Performance

Quality Characteristics and Performance

Programmers have a tendency to place undue importance on the optimality of their code.

> *More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason–including blind stupidity. –W.A. Wulf (CC2, p. 588)*

> *Premature optimization is the root of all evil. –Donald Knuth (CC2, p. 594)*

Software Performance

Quality Characteristics and Performance

Programmers have a tendency to place undue importance on the optimality of their code.

> *More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason–including blind stupidity. –W.A. Wulf (CC2, p. 588)*

> *Premature optimization is the root of all evil. –Donald Knuth (CC2, p. 594)*

When is raw speed important and when is it not?

Software Performance

Quality Characteristics and Performance

Programmers have a tendency to place undue importance on the optimality of their code.

> *More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason–including blind stupidity. –W.A. Wulf (CC2, p. 588)*
>
> *Premature optimization is the root of all evil. –Donald Knuth (CC2, p. 594)*

When is raw speed important and when is it not?

Other qualities may be more important than speed:

- Correctness
- Clean user interface
- Avoiding downtime
- Delivering product on time

Software Performance

Even if performance is a priority, think about it from several viewpoints before starting low-level code optimization.

Program Requirements

- Performance is stated as a requirement far more often than needed
- Before investing significant time, make sure it's necessary
- Example: System at TRW
  - Requirement of subsecond response time
  - Estimated cost: $100 million
  - Revised analysis indicated 4-second response time was sufficient.
  - Revised estimate: $30 million

Software Performance

Program Design

- Program design can have a significant impact on performance
- Example: data-acquisition program
  - Initial design required evaluation of 13-degree polynomials
  - Revised design used different hardware and many 3rd-order polynomials
  - Unlikely that initial design could have met the performance requirements

Software Performance

Program Design

- Program design can have a significant impact on performance
- Example: data-acquisition program
  - Initial design required evaluation of 13-degree polynomials
  - Revised design used different hardware and many 3rd-order polynomials
  - Unlikely that initial design could have met the performance requirements

Design a performance-oriented architecture, and then set resource goals for individual subsystems

Software Performance

Program Design

- Program design can have a significant impact on performance
- Example: data-acquisition program
  - Initial design required evaluation of 13-degree polynomials
  - Revised design used different hardware and many 3rd-order polynomials
  - Unlikely that initial design could have met the performance requirements

Design a performance-oriented architecture, and then set resource goals for individual subsystems

- Setting individual resource goals makes the system performance predictable
- Making explicit goals improves the likelihood they will be achieved: programmers are goal-oriented
- Make design choices that promote efficiency: e.g. high degree of modifiability makes it easy to swap a less-efficient module with a more-efficient one.

Software Performance

Other performance considerations

- *Class and Routine Design* Choice of data types and algorithms can make a big difference, i.e. quicksort instead of bubble sort; binary search instead of linear search.
- *Operating-System Interactions* Be aware of file I/O, or other interactions with the operating-system. These can be a bottleneck.
- *Code Compilation* A good compiler can often produce faster code than you can. Make sure you are getting the most out of your compiler.
- *Hardware* Sometimes, new hardware is the easiest way to increase performance.
- *Code Tuning* Small-scale local changes to improve efficiency.

Often, dramatic improvements can be made at each level from system design to code tuning, offering a potential speedup of several orders of magnitude.

Code Tuning

Code tuning has the potential for dramatic results, but watch out for common mistakes and pitfalls.

The Pareto Principle

- You can get 80 percent of the result with 20 percent of the effort, or
- 80 percent of the execution time is consumed by 20 percent of the code
- If you optimize *everything*, 80 percent of your effort will be *wasted*
- Moral: first find out where the bottlenecks are and then optimize only the responsible code

Code Tuning

Fallacies and Pitfalls

- *Reducing the lines of code in a high-level language improves the speed or size of the resulting machine code*

Code Tuning

Fallacies and Pitfalls

- *Reducing the lines of code in a high-level language improves the speed or size of the resulting machine code*

  No predictable relationship exists between number of lines of code and ultimate size and speed. Example: loop unrolling.

Code Tuning

Fallacies and Pitfalls

- *Reducing the lines of code in a high-level language improves the speed or size of the resulting machine code*

  No predictable relationship exists between number of lines of code and ultimate size and speed. Example: loop unrolling.

- *Certain operations are probably faster or smaller than others*

Code Tuning

Fallacies and Pitfalls

- *Reducing the lines of code in a high-level language improves the speed or size of the resulting machine code*

  No predictable relationship exists between number of lines of code and ultimate size and speed. Example: loop unrolling.

- *Certain operations are probably faster or smaller than others*

  Performance varies significantly depending on the language, choice of compiler, version of compiler, version of libraries, operating system, hardware (especially CPU), amount of memory, etc. The only way to reliably determine what is smaller or faster is to *measure*.

Code Tuning

Fallacies and Pitfalls

- *Reducing the lines of code in a high-level language improves the speed or size of the resulting machine code*

  No predictable relationship exists between number of lines of code and ultimate size and speed. Example: loop unrolling.

- *Certain operations are probably faster or smaller than others*

  Performance varies significantly depending on the language, choice of compiler, version of compiler, version of libraries, operating system, hardware (especially CPU), amount of memory, etc. The only way to reliably determine what is smaller or faster is to *measure*.

- *You should optimize as you go*

Code Tuning

Fallacies and Pitfalls

- *Reducing the lines of code in a high-level language improves the speed or size of the resulting machine code*

  No predictable relationship exists between number of lines of code and ultimate size and speed. Example: loop unrolling.

- *Certain operations are probably faster or smaller than others*

  Performance varies significantly depending on the language, choice of compiler, version of compiler, version of libraries, operating system, hardware (especially CPU), amount of memory, etc. The only way to reliably determine what is smaller or faster is to *measure*.

- *You should optimize as you go*

  The primary drawback of this attitude is a lack of perspective. Other qualities are usually more important and harder to fix later. Also, it is very hard to correctly identify bottlenecks during coding.

Code Tuning

Fallacies and Pitfalls

- *Reducing the lines of code in a high-level language improves the speed or size of the resulting machine code*

  No predictable relationship exists between number of lines of code and ultimate size and speed. Example: loop unrolling.

- *Certain operations are probably faster or smaller than others*

  Performance varies significantly depending on the language, choice of compiler, version of compiler, version of libraries, operating system, hardware (especially CPU), amount of memory, etc. The only way to reliably determine what is smaller or faster is to *measure*.

- *You should optimize as you go*

  The primary drawback of this attitude is a lack of perspective. Other qualities are usually more important and harder to fix later. Also, it is very hard to correctly identify bottlenecks during coding.

- *A fast program is just as important as a correct one*

Code Tuning

Fallacies and Pitfalls

- *Reducing the lines of code in a high-level language improves the speed or size of the resulting machine code*

  No predictable relationship exists between number of lines of code and ultimate size and speed. Example: loop unrolling.

- *Certain operations are probably faster or smaller than others*

  Performance varies significantly depending on the language, choice of compiler, version of compiler, version of libraries, operating system, hardware (especially CPU), amount of memory, etc. The only way to reliably determine what is smaller or faster is to *measure*.

- *You should optimize as you go*

  The primary drawback of this attitude is a lack of perspective. Other qualities are usually more important and harder to fix later. Also, it is very hard to correctly identify bottlenecks during coding.

- *A fast program is just as important as a correct one*

  It's easy to write a fast program that does the wrong thing. Without correctness, speed is meaningless.

Code Tuning

When to Tune

Jackson's Rules of Optimization:

- *Rule 1. Don't do it.*
- *Rule 2. (for experts only) Don't do it yet.*

–M. A. Jackson (CC2, p. 596)

- Use a high-quality design (modular and easily modifiable)
- Make sure the program works and does the right thing
- Finally, if performance is lacking, use profiling to determine where the bottlenecks are and fix those
- Example: Graph-drawing Program
  - First version took 45 minutes to draw a graph
  - *We have to start rewriting the whole code base in assembler right now!*
  - After a day of measurement and analysis, a small number of code-tuning changes reduced the drawing time to 30 seconds. A few more changes reduced the time to about 1 second.

Code Tuning

Common Sources of Inefficiency

- *Input/Output operations*
  - Accessing memory is *much* faster than accessing a file (1000 times faster in one test)
  - Local files are typically faster than files accessed over a network
  - Think twice about relying on file I/O in a speed-critical part of a program
- *Paging*
  - Paying attention to locality can make a big difference.
  - Example: Iterating over a large 2-dimensional array; up to 1000 times difference depending on the order of the indices.
- *System Calls*
  - Calls to system routines often involve a context switch.
  - Consider these alternatives:
    * Avoid making system calls
    * Write your own services
    * Work with the system vendor to make the call faster

Code Tuning

Common Sources of Inefficiency

- *Interpreted Languages*
  - Java tends to be slightly slower than C and C++
  - Interpreted languages like PHP and Python are up to 100 times slower
- *Errors*
  - Leaving debugging code in place
  - Memory leaks
  - Polling nonexistent devices until they time out
  - Bad database design (i.e. linear search instead of binary search)
- *Expensive operations*
  - Some function calls
  - Division
  - Transcendental functions (i.e. square root)

Measurement

You can't reliably identify performance bottlenecks without measuring them

You can't assess if you've fixed a performance problem unless you measure again

Experience is often not only unhelpful, but often misleading. With a new machine, language, or compiler, the results may be very different.

Example: *Which code is faster?*

```
sum = 0;
for (row = 0; row < rowcount; row++) {
  for (column = 0; column < columnCount; column++) {
    sum += matrix[row][column];
  }
}
```

or

```
sum = 0;
elementPointer = matrix;
lastElementPointer = matrix[rowCount-1][columnCount-1]+1;
while (elementPointer < lastElementPointer) {
  sum += *elementPointer++;
}
```

Measurement

Answer: It depends

- In a typical setting, there is no difference: the compiler makes this transformation automatically
- In another setting, performance improved more than 50 percent
- In another setting, performance degraded about 10 percent

Measurement

Be Precise

- Use a profiling tool (more about this later)
- In Unix, you can use the `time` command
- For more fine-grained measurement, use system routines that measure elapsed time according to the system clock
- Try to factor out measurement overhead and program-startup overhead

Iterate

- A single technique will rarely result in a dramatic improvement
- Many attempts to speed things up will actually slow things down
- With repeated effort and measurement, you can usually identify multiple techniques which together are very effective

Summary of Code Tuning Approach

1. Use a modular and easily modifiable design.
2. If performance is poor, then
   (a) Save a working version of the code.
   (b) Measure the system to find hot spots.
   (c) Determine whether weak performance comes from inadequate design, data types, or algorithms and whether code tuning is appropriate. If code tuning is not appropriate, go back to step 1.
   (d) Tune the bottleneck identified in (c).
   (e) Measure each improvement *one at a time*.
   (f) If a potential improvement doesn't improve the code, revert to the saved copy from step (a). (Typically, more than half your attempts will produce no effect or actually degrade performance.)
3. Repeat from step 2.

Using gprof to measure performance

`gprof` works with `g++` to figure out where the time is spent when you run your program.

To use gprof, you must do the following:

1. Compile source files with `-pg`
2. Run the executable (it will run slower than usual, but the relative performance of each part of the program will be the same)
3. Type `gprof` *`executable`*. If you want to redirect the output to a file, add `>` *`outputfile`*.

Using the advanced Makefile from the sample project (get the latest version), you can do step one automatically by typing `make GPROF=1`.

We will look at an example in `sample_project/src/generic/test/test2.cpp`.

Using gcov to test code coverage

`gcov` works with `g++` to figure out which lines of your program have been executed.

To use gcov, you must do the following:

1. Compile source files with `-fprofile-arcs -ftest-coverage`
2. Run the executable
3. For each source file *`source`*`.cpp` whose coverage you want to examine, type `gcov -o` *`objectDir source`*`.cpp`
4. Output from gcov is in the file *`source`*`.cpp.gcov`

Using the advanced Makefile from the sample project (get the latest version), you can do step one automatically by typing `make GCOV=1`.

To additionally get information on branches, use `gcov -b -o` *`objectDir source`*`.cpp`.

Using `gcov` as part of your testing strategy is a good idea.

Using a makefile to package a source distribution

The makefile for the sample project can package a source distribution of your project.

Here's what you need to do to use it:

1. Make sure you have all of your public header files in `src/Makefile`.
2. Make sure each module's `Makefile` includes any private header files.
3. Type `make dist`
4. You now have a file with all the code in it which you can copy to another location or computer: `dist.tar.gz`.
5. To unpackage the code, type `tar zxvf dist.tar.gz`.