COMP 161 - Lecture Notes - 20 - Evaluating Programs: Efficiency

Spring 2014

In these notes we discuss the use of Valgrind's profiling tools for gathering empirical data about program efficiency.

Computational Resources

Something is efficient when it achieves an outcome with minimal use of resources. If we're to talk about efficient computation, then we need to identify the computational resources we're attempting to minimize while still achieving our goal of program correctness. For this we turn to the machine itself. Computation is a collaboration between the CPU and the Memory system, so the resources we must consider involve the usage of those systems.

1. Computational Work Work carried out by the CPU
2. Memory Allocation Amount of memory used
3. CPU-Memory Communication Cost of moving data from the Memory to the CPU

Memory Efficiency

When we talk about memory system efficiency we're talking about the way in which our program consumes space on the computer. When talking about the memory space of a program we look at two distinct areas: the run-time stack and the *heap*.

Stack Frames and the Run-Time Stack

The run-time stack holds the local data for each active procedure. The set of data local to a specific procedure is called the procedure's stack frame. A procedure becomes active when it is called, and it remains active until the procedure's return statement is reached. When a procedure is called it's frame is added to the stack. Adding things to a stack is called *pushing* to a stack. When a procedure returns, its stack frame is removed from the stack. Removing things from a stack is called *popping* the stack. If a procedure calls another procedure, then the caller remains active until the called procedure, called the callee, is complete. This means that the stack frame of the caller remains on the stack until after the callee is complete and its data is removed. Every single invocation of a procedure gets its

own stack frame. This means recursive procedures do not, in general, share a single stack frame, but instead each recursive invocation gets its own frame. If you call 10 procedures, then 10 frames are produced. It doesn't matter if those are 10 invocations of one procedure or 10 different procedures.

If we examine the behavior of the stack closely, we see that *the frame of the most recently called procedure is always the next frame to get popped from the stack*.[1]. It is this mechanism that allows the computer to keep track of the order in which procedures are called. When a procedure returns, it's frame is removed and the frame for it's caller must be the the next frame on the stack. The computer can then go back to executing the code relative to that frame. The operating system kicks the whole process off by pushing the frame for *main* on to the stack when the program is invoked at the CLI. When *main* returns, the stack and all other program data is cleared.

[1] LIFO: **L**ast **I**n, **F**irst **O**ut

An interesting feature of stack frames is that the frame size for a procedure is fixed and must be determined at compile time[2]. This means that we can analyze and evaluate certain characteristics of a program's stack space usage simply by looking at the code. If you know the frame size per procedure and some information about when and why procedures are called, you can use that to predict things like the min and max stack space used and thereby establish the *best* and *worst* case for stack memory consumption. The average amount of stack space used is usually dependent on program inputs and is therefore tricky to analyze statically.[3]. If we knew or assumed some properties of the data we'll be working with, we might be able to establish some bounds on the average stack size. On the other hand, we might also run our program with actual data and measure the stack size as it runs. This data can then be used to establish some statistical properties of the stack size for our code. Either way, the dynamic behavior of the stack at run-time makes average case analysis difficult.

[2] We say it's a STATIC property of the program if it can be determined at compile time

[3] It is a DYNAMIC property of the program

The Heap

Structures like vector's can grow and shrink while a program is running. They cannot, therefore, be kept on the stack. DYNAMICALLY ALLOCATED data like vector data is stored on the HEAP. When a procedure uses heap data, then it keeps track of that data by stack variables that store the address of the data on the heap, and because addresses are always the same number of bits, the data needed to store addresses is fixed. This is a pretty cool trick. We use fixed sized references to dynamic data so that stack usage is predicable and controllable while still enabling dynamic allocation of memory space

on the heap. Additionally, by restricting access to the heap to stack-based reference variables, we can toss dynamically allocated data in one giant free-for-all space and not worry about one procedure inadvertently accessing another procedure's data. This means we don't need any sense of frames on the heap. It's just raw storage space. The more you study the stack/heap relationship the more you'll see that it's a really wonderful balance of strict control and flexibility. The former is important in writing correct programs where the later is important for simple and efficient programs. Thus, programmers need a little of both.

Analyzing the heap usage of our programs statically is tricky as the space is, by definition, governed by run-time behavior. None the less, we can usually predict the min/best and max/worst of the amount of heap space we'll need by careful analysis of the code and the problem. We then, once again, turn to observations of actual heap allocations on actual data to get a better sense of the average case.

Measuring Memory Usage

Memory usage is measured in Bytes(B)[4] and in some cases bits (b). One byte is equivalent to 8 bits. Valgrind provides us with a memory usage profiler called **Massif**[5] that can be used to monitor the heap and stack usage of our program. Massif monitors and measures the size of the stack and heap over time. However, we have three ways to count time with massif: milliseconds, bytes allocated/deallocated, and instructions executed. The later is the default unit of measurement and generally a good one use. For small/short programs with a small number of instructions executed, measuring by Bytes is often a better alternative. Measuring time with milliseconds sometimes nice, but execution time is subject not just to teh code we wrote but the computer we're running it on. So, time based analysis forces us to consider the effects of the machine as well as the code.

[4] or MB, KB, GB

[5] `http://valgrind.org/docs/manual/ms-manual.html`

To use massif you must compile all your program source code with the *-g* option. This causes the compiler to tag the code in such a way that the profiler can attribute execution behavior to C++ code. Remember, what's running is based on compiler generated assembly which is a translation of your C++, so *-g* let's the profiler attribute profiled data to C++ rather than assembly.

Next you have to run your program under massif. This means that Valgrind will simulate the execution of the program and measure the memory usage with massif as it does so. You need to tell valgrind to use massif as well as the options for massif that you want it to use. The basic options we'll consider are time units and which parts of the memory system to measure. The command looks like this:

```
valgrind --tool=massif --time-unit={i,B,ms} --heap={yes,no} --stacks={yes,no} <program> <program input and
```

When you have specific choices, they're listed in curly braces. When you have to give names or items that are not from a fixed set of options, like the name of your program, then you'll see a description of the item in $< >$. The result of running massif is a file called *massif.out.<pid>* where *pid* is the process id for your program. This number is generally unique for each process and will be unique every time you run massif. There are options for choosing the name of the output file, or if you want to rename it you can use the CLI commands.

The massif output is not meant to be read by humans. Instead, they provide a program called *ms_print* which graphs the results as well as produces several tables worth of data. Using this program is simple, just pass it the massif.out file. Be aware that the results are dumped to the standard output, so you should probably be ready to redirect them to a file.

```
ms_print massif.out.<pid>
```

1. Compile all source code with *-g* option.[6]

   ```
   g++ <source-file> -c -Wall -g
   ```

2. Run the program through massif as follows:

   ```
   valgrind --tool=massif --time-unit={i,B,ms} --heap={yes,no} --stacks={yes,no} <program> <program input a
   ```

3. Use *ms_print* to get graph of the data (and possibly redirect them to a file)

   ```
   ms_print massif.out.<pid>
   ```

You can run the program multiple times to gather data for different time units or to do separate heap and stack data collection.

[6] Consider using special Makefile rules for this.

CPU Efficiency

The CPU executes instructions, so CPU efficient programs execute fewer instructions. Because fewer instructions represents a small computational work load, it often leads to faster execution times. Execution time, however, is dependent on more than just instructions executed[7], so we tend to avoid time as strict measure of computational work load. Operations in C++ often have a clear correspondence to instructions, that is a single C++ operation typically corresponds to several instructions, but they are not equivalent to the instructions counted by programs like Valgrind.

[7] see your Architecture and Organization class for more details

To evaluate the efficiency of a program or procedure we can simple tally up the total number of instructions. A line-by-line count allows us to get a more fine grained picture of what's going on. If we count C++ operations, we can get a rough idea of total instructions, so analyzing the computational work load of our program and establishing a min and max value is possible simply by looking at the code. Average case analysis is still difficult to do statically as it is most often determined by specific features of program inputs. So, we will once again turn to dynamic, run-time profilers to paint us a picture of what to expect for average inputs.

Counting Instructions

Valgrid's *callgrind*[8] tool will gather data about instructions executed by each line of code across the total execution of the program. In addition to associating a cost with each procedure, callgrind will attempt to map out the call-graph of our program. This graph, or map, links procedures by caller/callee relationships. Recursion posses a bit of a challenge to programs like callgrind as it creates cycles in this graph. These cycles also throw off instruction counts a bit, but more on that later.

[8] `http://valgrind.org/docs/manual/cl-manual.html`

Running callgrind is more or less the same as massif. We'll ignore call grind options for now and just get the standard data.

```
valgrind --tool=callgrind  <program> <program options>
```

The resultant file is called *callgrind.out.<pid>* where *pid* is, once again, the process id assigned to the program execution that Valgrind profiled.

As will continue to be the pattern, we use a separate program, *callgrind_annotate* this time, to get a human readable report of the callgrind data file. Instruction count data can be reported as an *inclusive* cost, where the cost of the procedure includes the cost of any procedure it calls, or *exclusive*, where the cost of a procedure is only that of the local instructions and not the instructions executed by called procedures. Recursion can and will inflate the inclusive costs for a recursive procedure. This makes our style of using a top-level procedure to initiate the recursion particularly helpful as the inclusive cost of that procedure should reflect the total cost of recursive process.

Callgrind collects data about all of the code executed. This includes library code as well as the code you yourself wrote. It presents procedures executed in order of most to least instructions executed. This can make analysis based on the sorted list difficult. To make analysis of your code easier, you can have callgrind_annotate add instruction counts and callgraph data to your code, i.e. it will annotate

source code with profile data. So, the big sorted table of data gives you a sense of the big picture and your annotated code lets you see what's going on in the code you wrote.[9]

[9] There are ways of telling callgrind to ignore parts of the program. You're welcome to explore them on your own.

```
callgrind_annotate --inclusive={yes,no} <callgrindfile> <source to annotate>
```

Once again, the results of the data processor are printed to the standard out, so you might need to redirect them to a file for reference later.

All together:

1. Compile source code with the *-g* option.
2. Run your code under callgrind to produce profiling data.

   ```
   valgrind --tool=callgrind  <program> <program options>
   ```

3. Use callgrind_annotate and profiling data to annotate your source and get overall instruction counts.

   ```
   callgrind_annotate --inclusive={yes,no} <callgrindfile> <source to annotate>
   ```

Where CPU meets Memory

Not all instructions are created equal. Instructions that work on data that is stored within the CPU memory space, called the CACHE, are very fast. Other instructions require data stored in the off-CPU memory system. The CPU will only work on data in the cache, so this off-CPU memory must be moved in to the cache. When the data we need resides in the CPU cache, we call it a CACHE HIT. If, however, that data is not in the Cache then there has been a CACHE MISS and the data must be pulled in to the cache from RAM, or worse yet the hard drive.. This process is, by computer standards, slow and costly. The exact details of this process are covered in your Architecture and Organization class.

Cache misses are a way of measuring the COMMUNICATION cost of our program[10]. Communications efficient programs aim to minimize cache misses. Communication ends up being a major bottleneck for overall performance so a lot of time and energy often goes in to reducing cache misses and thereby increasing the CACHE HIT RATE. This typically a system dependent process as different CPUs have different caches. We're not really going to get into cache related issues in this class but we will at least learn to measure cache performance.

[10] Parallel programs use other events to measure communication. We're only looking a sequential programs

Valgrind's *cachegrind* provides a detailed break down of instruction behavior with respect to the cache. It counts instructions that read/write from the cache and from memory and tells you how

many of those instructions missed the cache. Cachegrind simulates two caches, one for storing instructions[11] and one for program data. Each cache can miss at two levels. Level 1 corresponds to misses within the CPU cache system. They are less costly as they don't necessarily mean we had to fetch data from the RAM or from the HDD. The lower level misses represents cache misses that require fetching data from off-CPU memory and are therefore the very costly cache misses corresponding to CPU to Memory communication. The two values are related and so we need to take both into account as we evaluate the communication cost of our code. Programs only read data from the instruction portion of memory so we only get data about Instruction reads and misses. For the data cache we get information about read and write instructions. This results in nine data points: Instruction Reads, instruction read misses at level 1 (L1), instruction read misses at the lower level (LL), data reads, data read misses at L1, data read misses at LL, data writes, data write misses at L1, and data write misses at LL.

[11] yes your code exists in memory

To run of cachegrind, we again compile our code with the *-g* option. We can then run the program through Valgrind:

```
valgrind --tool=cachegrind <program> <program option>
```

This spits out a data filed named *cachegrind.out.<pid>*. Once we're ready to look at the data we can use *cg_annotate* to annotate our source code with the counts.

```
cg_annotate <cachegrind data file> <source files to annotate>
```

If we want to aggregate data from multiple cachegrind data files then we can use *cg_merge*.

```
cg_merge <cachegrind file list> -o <outputfile name>
```

We can then use cg_annotate to get a report on the aggregate data.

Profiling vs Testing

We use unit tests to assess the correctness of our code. Some unit testing frameworks let you write tests that fail if some code takes too long, but they're generally not used to profile code and measure efficiency. Profiling is a step we take *after* our tests pass and we're reasonably sure our code functions correctly. If tests are passing and your code doesn't work as intended, then understanding its efficiency isn't something you should be focused on.

Profiling is either carried out in context, i.e. we profile the main application itself, or it's done by special programs designed for profiling. These programs are setup to run the code we want to profile

in such a way as to make gathering efficiency measurements easier. This means isolating key procedures and potentially repeating the execution of the code we're trying to profile in such a way that a more robust and information rich profiling data set can result from a single execution of the program. In short, programs written for profiling often attempt to execute code under controlled laboratory-like conditions.[12]

[12] That is how you take accurate measurements in science, right?

If we want to profile procedures that act on vectors, then what kind of vectors should we profile with? It largely depends on your goal. What do you want to know about the performance of your program? If you have a specific application in mind and you know what kind of vectors you expect to get in your target application, then you should profile with those. If instead you're interested in average performance, you should profile with a wide array of vectors. Typically, we avoid small sized vectors[13] because measuring small things is hard. Larger vectors should take more work to process and thereby give us a good sense of whats happening. It's also generally the case that critical differences in implementations really stand out when working with large data sets. Notice that this is a little bit counter to how we tend to develop test cases. When writing test cases we think of small to moderate examples that expose key problem details. When coming up with data sets for profiling we often think of large data sets that represent the problem as a whole. It also makes a lot of sense to use random data sets when we're not concerned with performance under specific conditions as we're not measuring the expected outcome but the quality of effort to reach that outcome.

[13] unless that's what we see in our application

In the end, profiling is typically an act of scientific experimentation. If your goal is to compare implementations of a procedure, then run the different implementations on the exact same data so that the code is the only variable. If you're measuring average performance of a single implementation, then run it on lots of different data sets, not just a single data set. If you're interested in performance under a specific condition, then generate that exact condition. In the end, profiling is always goal driven. You might be gathering data simply to explore that data and look for interesting patterns and phenomenon, but if that's the case then you need to articulate what part of the program's performance your exploring and why. Once you're done exploring you're likely to need to generate more data to test a hypothesis about that phenomenon or even that absence of phenomenon. The point here is simple: gather data for a specific reason other than gathering data. Once you know that reason, you'll know what kind of data to gather and can figure out how best to gather it.

Recap

We've seen how to use Valgrind profiling tools to measure a program's use of three computational resources: Computational Work, Memory Space, and CPU to Memory Communication. Efficient programs minimize the use of these three resources. Massif allows us to examine the memory usage of our program by measuring the stack and heap allocations. Callgrind allows us to examine the computational work load of our program by measuring the number of instructions attributed to each line of C++. Finally, Cachegrind let's us examine the communication cost of our program by measuring the the number of memory access instructions and how often those instructions miss the cache.

Now let's consider a few ways to put this data to work for us.

1. Comparing Implementations and contrasting their efficiency
   (a) Compare the inclusive instruction count and cache miss counts of different implementation of a procedure to see which is more work and communication efficient respectively.
   (b) Use heap/stack peak allocations to compare memory usage of different procedure implementations
2. Guided Optimization
   (a) "Make the common case fast." Identify most frequently invoked procedures and lines of code and attempt to reduce the cost per execution.
   (b) Evaluate the effect of changes to code by looking at the numbers from before and after the changes
   (c) Evaluate the effectiveness of compiler optimizations by looking at the numbers with and without optimizations

Accurate measurement in science is rarely perfect and computer science is no exception; Valgrind's measurements do not always tell the whole store. That doesn't mean we should not measure. When making decisions and judgments about program efficiency we must use some form of objective data or we're not being scientifically rigorous. Profiler data is relatively easy to obtain and there are few reasons to not use it to shed light on the underlying workings of our code.

What can't our profiler data tell us? Well for starters, it can't tell us about the code we didn't profile. We really know nothing concrete about how our program will perform on data that we didn't profile. We could make statistical inferences, but these will not account for situations we did not measure. We'd always have to worry

about the outliers that are not like the situations seen by our profiler data. What can we do then to answer questions like: "Is there an implementation I haven't tried that would do better?" or "Is my implementation the best it can be for my given strategy?" If we're optimizing code, driving down the counts, then how do we know when to stop? How do we know how low we can go?

When science needs or wants to make predictions, it turns to theory. So, to understand why we're seeing the numbers we're seeing, make predictions about how they can and cannot change, and put them in context with the unobserved computational universe, we need to turn to the tools of theoretical computer science. As is the case for all theoretical branches of science, these tools rely on mathematics. With mathematics we can rigorously define the conditions in which a program is executed and mathematical prove what can and cannot happen under those conditions. We can then check our predictions and assumptions against the observed reality of our profiler data and assess the validity of our theoretical models.