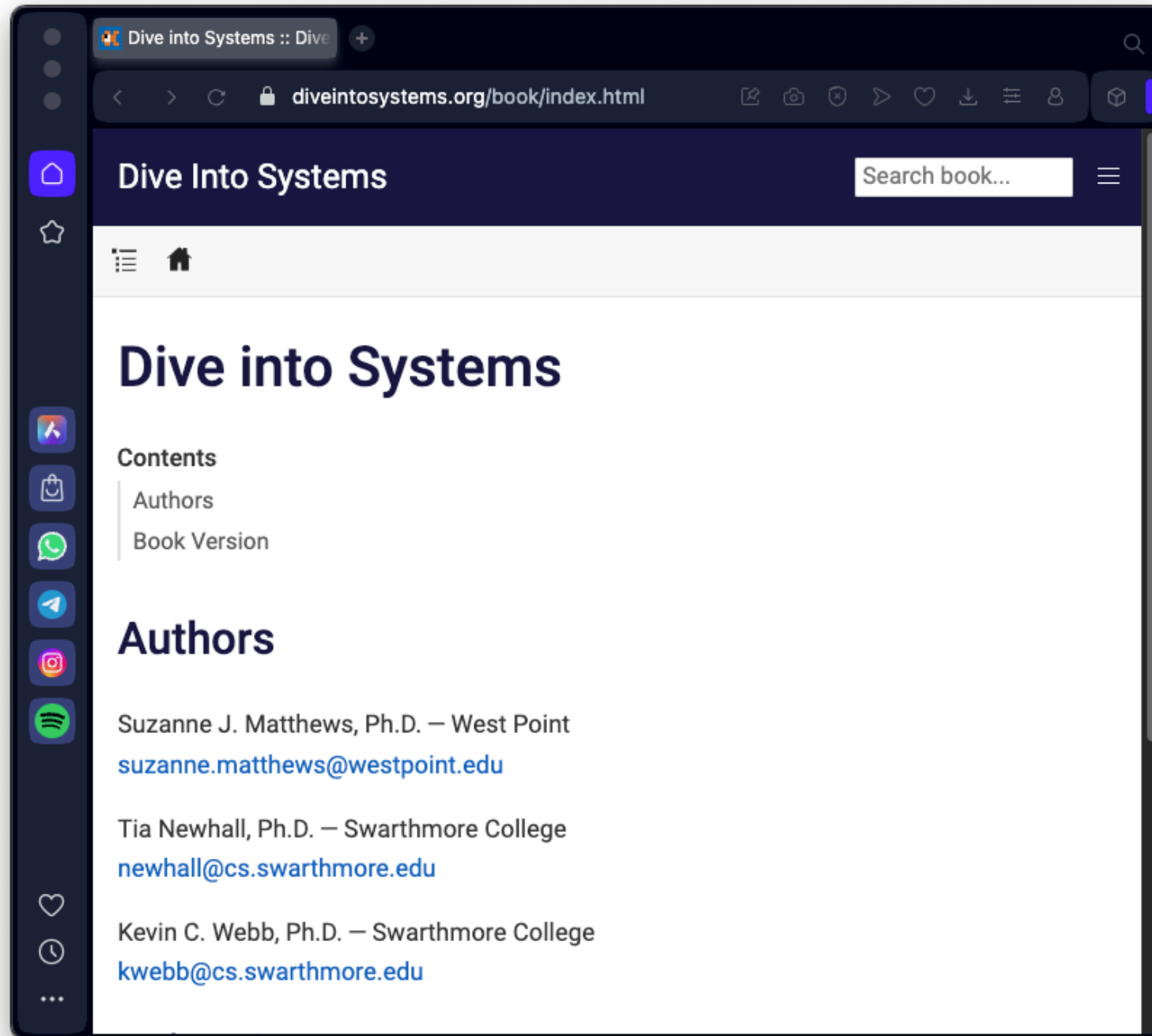


12. Code Optimization

For COMSC 142

Free online textbook



- <https://diveintosystems.org/book/index.html>

Topics

12.1. First Steps

12.2. Other Compiler Optimizations

12.3. Memory Considerations

12. Code Optimization

- gcc optimization flags
 - **-O1**
 - Basic optimizations to reduce code size and execution time
 - While attempting to keep compile time to a minimum.
 - **-O2**
 - Most optimizations that do not involve a space-performance trade-off
 - **-O3**
 - Additional optimizations, such as function inlining
 - May significantly increase compile time

What Compilers Already Do

- **Constant Folding**
 - Constants are evaluated at compile time
 - In the code below
 - Macro expansion replaces
 - **int debug = N-5** with
 - **int debug = 5-5**
 - Constant folding then updates this statement to
 - **int debug = 0**

```
#define N 5  
int debug = N - 5; //constant folding changes this statement to debug = 0;
```

What Compilers Already Do

- **Constant Propagation**
 - Replaces variables with a constant value
 - If the value is known at compile time
 - In the code below, it will change
 - **if (debug)** to
 - **if (0)**

```
int debug = 0;
```

```
    if (debug) {  
        printf("array[%d] is: %d\n", i, array[i]);  
    }
```

What Compilers Already Do

- **Dead Code Elimination**
 - The code outlined below never executes, and is removed

```
int debug = 0;

//sums up all the elements in an array
int doubleSum(int *array, int length){
    int i, total = 0;
    for (i = 0; i < length; i++){
        total += array[i];
        if (0) { //debug replaced by constant propagation by compiler
            printf("array[%d] is: %d\n", i, array[i]);
        }
    }
    return 2 * total;
}
```

What Compilers Already Do

- **Simplifying expressions**
 - **imul** and **idiv** are slow
 - Addition is faster than multiplying by two
 - **return 2 * total;** is changed to
 - **return total + total;**

What Compilers Cannot Always Do: Benefits of Learning Code Optimization

- **Algorithmic Strength Reduction Is Impossible**
 - Compilers can't fix poor choices of data structures and algorithms
 - Like bubble sort instead of quicksort
- **Compiler Optimization Flags Are Not Guaranteed to Make Code "Optimal" (or Consistent)**
 - Higher optimization levels may slow code or cause errors
 - Difficult to debug, because debug flag **-g** is incompatible with optimization flags **-O**

What Compilers Cannot Always Do: Benefits of Learning Code Optimization

- **C/C++** standard does not provide clear guidance for resolving undefined behavior
- Consider running this code with **a = INT_MAX**
- Adding 1 causes integer overflow
- Compiling with no optimizations returns 0
- Compiling with **-O3** returns 1

```
int silly(int a) {  
    return (a + 1) > a;  
}
```

What Compilers Cannot Always Do: Benefits of Learning Code Optimization

- **Pointers Can Prove Problematic**
 - Sometimes two pointers point to the same address
 - "memory aliasing"
- The code below works as expected if the two parameters are different
- But calling **ShiftAdd(&x, &x)** makes them different
- Compilers won't make this optimization

Unoptimized Version

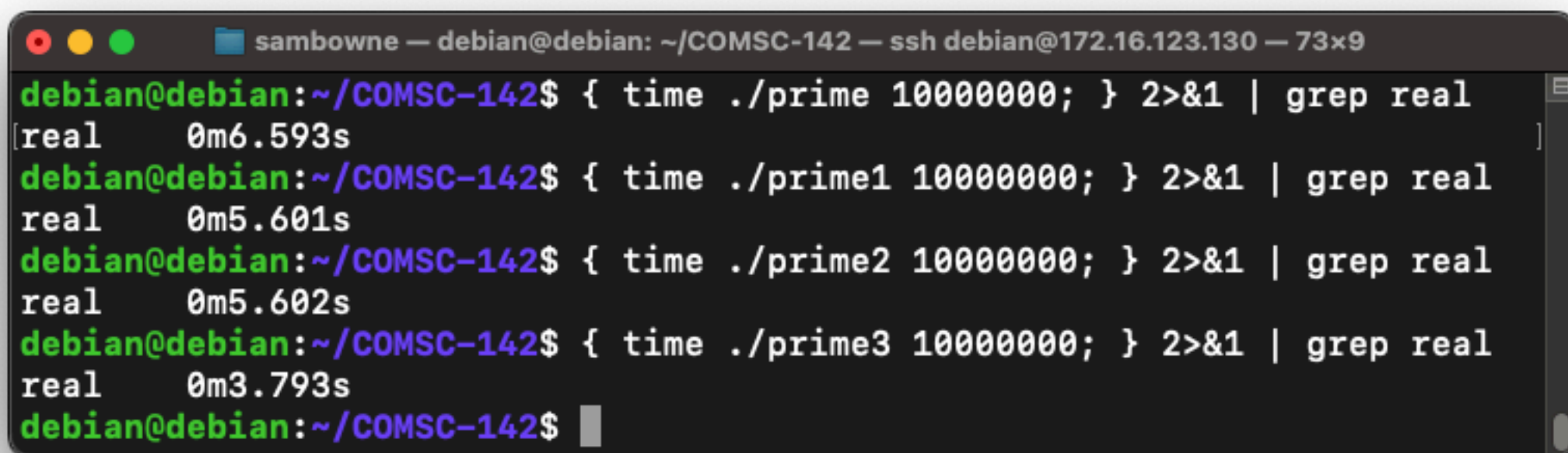
```
void shiftAdd(int *a, int *b){  
    *a = *a * 10; //multiply by 10  
    *a += *b; //add b  
}
```

Optimized Version

```
void shiftAddOpt(int *a, int *b){  
    *a = (*a * 10) + *b;  
}
```

Example

- `wget https://samsclass.info/COMSC-142/proj/prime.c`
- `gcc -o prime prime.c -lm`
- `gcc -O1 -o prime1 prime.c -lm`
- `gcc -O2 -o prime2 prime.c -lm`
- `gcc -O3 -o prime3 prime.c -lm`



```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 73x9
debian@debian:~/COMSC-142$ { time ./prime 10000000; } 2>&1 | grep real
[real    0m6.593s
debian@debian:~/COMSC-142$ { time ./prime1 10000000; } 2>&1 | grep real
real    0m5.601s
debian@debian:~/COMSC-142$ { time ./prime2 10000000; } 2>&1 | grep real
real    0m5.602s
debian@debian:~/COMSC-142$ { time ./prime3 10000000; } 2>&1 | grep real
real    0m3.793s
debian@debian:~/COMSC-142$
```

12.1. First Steps

12.1. Code Optimization First Steps: Code Profiling

- **Premature optimization**
 - A programmer attempts to optimize based on "gut feelings" of where performance inefficiencies occur, and not on data
- Measure performance first to identify **hot spots**
 - Areas in the program in which the most instructions occur
- The next slide shows that for **prime.c**, generating primes is the slow step

```
int main(int argc, char ** argv) {
    if (argc != 2) {
        fprintf(stderr, "usage: %s <num>\n", argv[0]);
        printf("where <num> is upper limit of the desired range of primes\n");
        return 1;
    }

    struct timeval tstart, tend;
    int limit = strtol(argv[1], NULL, 10);
    gettimeofday(&tstart, NULL);
    int * array = allocateArray(limit); //array can't be longer than the limit size
    gettimeofday(&tend, NULL);
    printf("Time to allocate: %g\n", getTime(tstart, tend));

    gettimeofday(&tstart, NULL);
    int length = genPrimeSequence(array, limit);
    gettimeofday(&tend, NULL);
    printf("Time to generate primes: %g\n", getTime(tstart, tend));
    printf("%d primes found.\n", length);
    //printf("The first %ld prime numbers are:\n", limit);
    //printArray(array, length);
    return 0;
}
```

```
debian@debian:~/COMSC-142$ ./prime 5000000
```

```
Time to allocate: 2.2e-05
```

```
Time to generate primes: 2.51054
```

```
348513 primes found.
```

```
debian@debian:~/COMSC-142$
```

genPrimeSequence

- Calls **genNextPrime** len times

```
// generates a sequence of primes
int genPrimeSequence(int *array, int limit) {
    int i;
    int len = limit;
    if (len == 0) return 0;
    array[0] = 2; //initialize the first number to 2
    for (i = 1; i < len; i++) {
        array[i] = getNextPrime(array[i-1]); //fill in the array
        if (array[i] > limit) {
            len = i;
            return len;
        }
    }
    return len;
}
```


genNextPrime

- Calls **isPrime** several times

```
// finds the next prime  
int getNextPrime(int prev) {  
    int next = prev + 1;  
    while (!isPrime(next)) { //while the number is not prime  
        next++; //increment and check again  
    }  
    return next;  
}
```

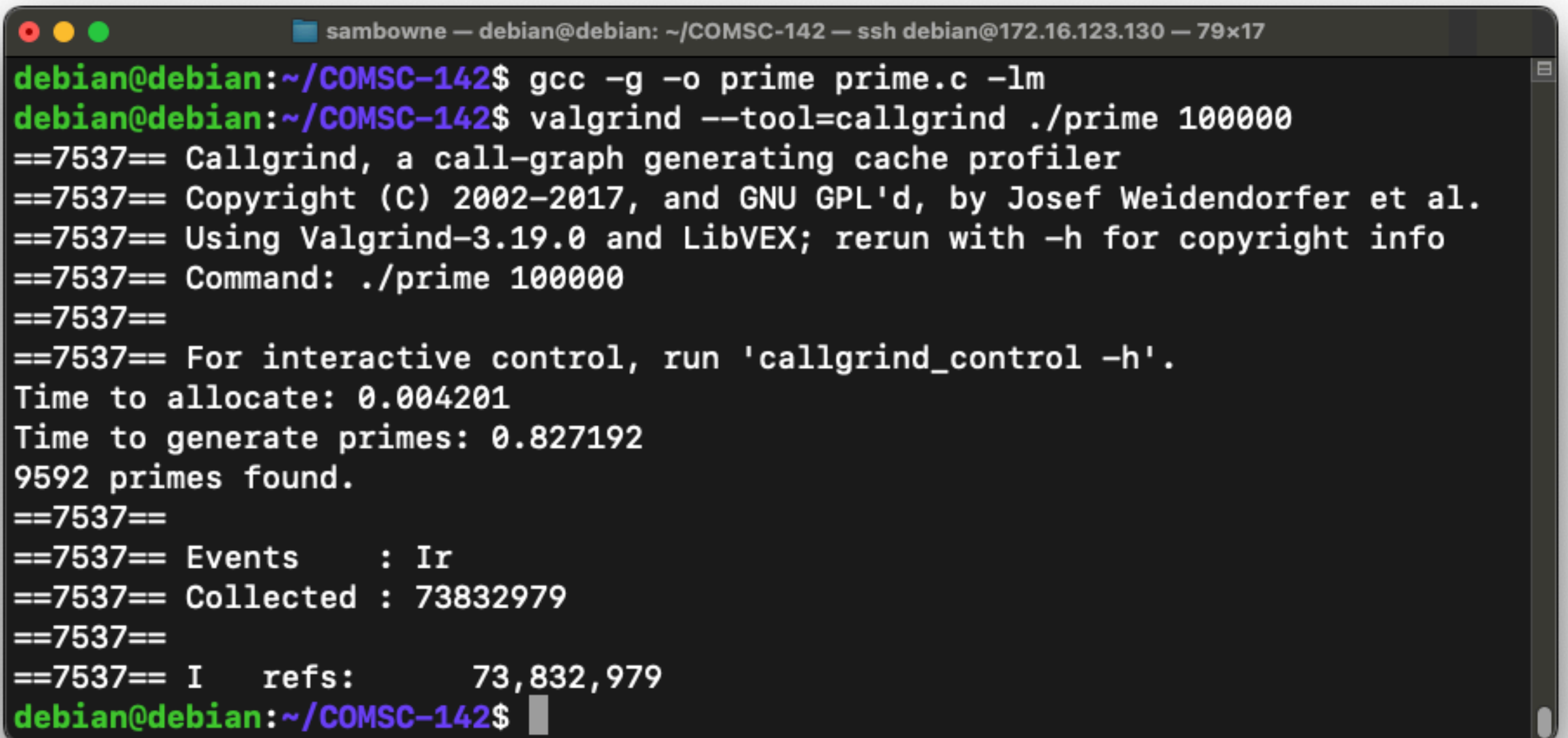
isPrime

- Loop executes many times
- Most likely the **sqrt()** function consumes the most CPU

```
// helper function: checks to see if a number is prime
int isPrime(int x) {
    int i;
    for (i = 2; i < sqrt(x) + 1; i++) { //no prime number is less than 2
        if (x % i == 0) { //if the number is divisible by i
            return 0; //it is not prime
        }
    }
    return 1; //otherwise it is prime
}
```

12.1.1. Using Callgrind to Profile

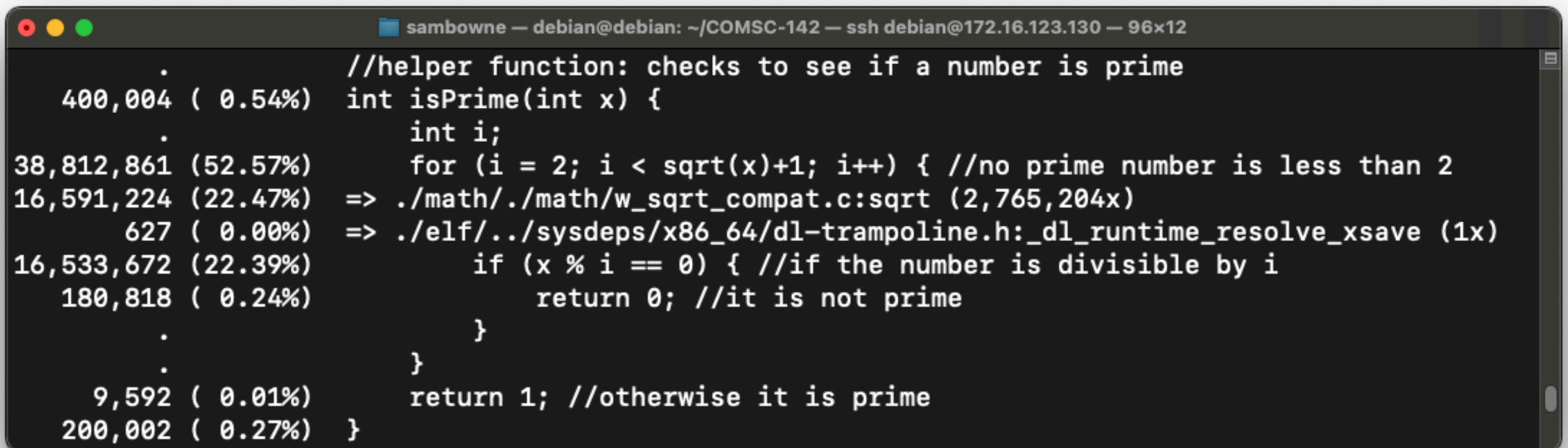
- `gcc -g -o prime prime.c -lm`
- `valgrind --tool=callgrind ./prime 100000`



```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 79x17
debian@debian:~/COMSC-142$ gcc -g -o prime prime.c -lm
debian@debian:~/COMSC-142$ valgrind --tool=callgrind ./prime 100000
==7537== Callgrind, a call-graph generating cache profiler
==7537== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==7537== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==7537== Command: ./prime 100000
==7537==
==7537== For interactive control, run 'callgrind_control -h'.
Time to allocate: 0.004201
Time to generate primes: 0.827192
9592 primes found.
==7537==
==7537== Events      : Ir
==7537== Collected : 73832979
==7537==
==7537== I    refs:      73,832,979
debian@debian:~/COMSC-142$
```

callgrind_annotate

- callgrind_annotate --auto=yes callgrind.out.7537
- Shows that **sqrt()** consumes the most time
- Followed by **x % i**



```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 96x12

. //helper function: checks to see if a number is prime
400,004 ( 0.54%) int isPrime(int x) {
. int i;
38,812,861 (52.57%) for (i = 2; i < sqrt(x)+1; i++) { //no prime number is less than 2
16,591,224 (22.47%) => ./math/./math/w_sqrt_compat.c:sqrt (2,765,204x)
627 ( 0.00%) => ./elf/./sysdeps/x86_64/dl-trampoline.h:_dl_runtime_resolve_xsave (1x)
16,533,672 (22.39%) if (x % i == 0) { //if the number is divisible by i
180,818 ( 0.24%) return 0; //it is not prime
. }
. }
9,592 ( 0.01%) return 1; //otherwise it is prime
200,002 ( 0.27%) }
```

12.1.2. Loop-Invariant Code Motion

- Moves static computations that occur inside a loop
 - to outside the loop without affecting the loop's behavior
- May cause **side effects**
 - if the function call inside the loop changes in a way the compiler can't detect
 - Such as some other global variable changing

Improved Code

```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 74x13

//helper function: checks to see if a number is prime
int isPrime(int x) {
    int i;
    int max = sqrt(x)+1;
    for (i = 2; i < max; i++) { //no prime number is less than 2
        if (x % i == 0) { //if the number is divisible by i
            return 0; //it is not prime
        }
    }
}
```

```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.1...

debian@debian:~/COMSC-142$ ./prime 5000000
Time to allocate: 2.2e-05
Time to generate primes: 2.41924
348513 primes found.
debian@debian:~/COMSC-142$ ./prime2 5000000
Time to allocate: 2.6e-05
Time to generate primes: 1.44763
348513 primes found.
debian@debian:~/COMSC-142$
```

fsqrt

- Optimization flags allow the compiler to replace **sqrt** library function calls with the **fsqrt** assembly code instruction
- This is called **inlining**

Kahoot!

Ch 12a

12.2. Other Compiler Optimizations

12.2. Other Compiler Optimizations: Loop Unrolling and Function Inlining

- It's usually best to let the compiler perform the optimization
 - When the optimized code is more difficult to read and understand
 - Confusing developers is usually a bigger problem than a small performance cost

12.2.1. Function Inlining

- Replaces calls to a function with the body of the function
- Avoids the overhead of a function call
- Makes it easier for the compiler to identify other potential improvements, like
 - constant propagation
 - constant folding
 - dead code elimination

```
// allocation of array  
int *a = allocateArray(lim);
```

```
// allocation of array (in-lined)  
int *a = malloc(lim * sizeof(int));
```

12.2.1. Function Inlining

- The **-finline-functions** flag suggests to GCC that functions should be inlined
- This optimization is turned on at level 3
- Programmers should generally avoid inlining functions manually
- Inlining functions carries a high risk of significantly reducing the readability of code, increasing the likelihood of errors, and making it harder to update and maintain functions

12.2.2. Loop Unrolling

- Reduce the number of iterations of a loop
 - By increasing the work performed in each iteration

```
// no prime number is less than 2  
for (i = 2; i < max; i++) {  
    // if the number is divisible by i  
    if (x % i == 0) {  
        return 0; // it's not prime  
    }  
}
```

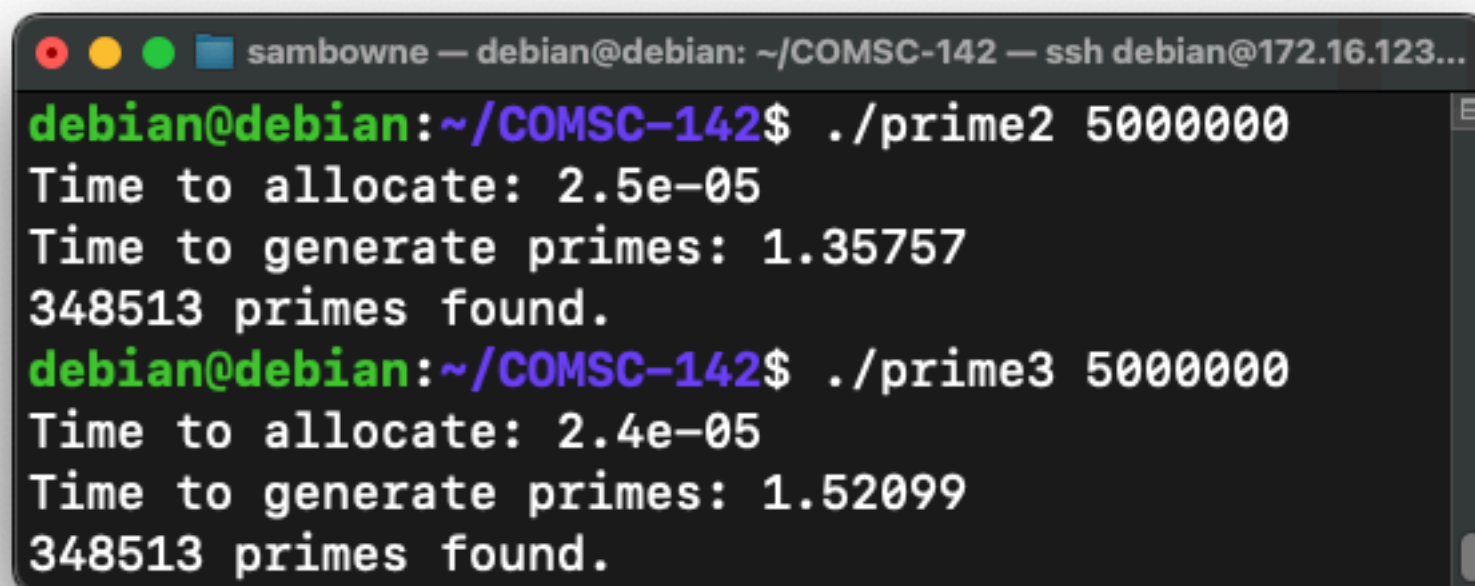
```
// no prime number is less than 2  
for (i = 2; i < max; i+=2) {  
    // if the number is divisible by i or i+1  
    if ( (x % i == 0) || (x % (i+1) == 0) ) {  
        return 0; // it's not prime  
    }  
}
```

12.2.2. Loop Unrolling

- **Branch predictors** attempt to guess which way a branch will go in advance
 - So that **speculative execution** can be performed
- This makes each loop iteration more costly
 - The processor may incorrectly guess whether the loop is ending

12.2.2. Loop Unrolling

- Manual loop unrolling led to a minimal improvement in your textbook
 - And no improvement at all on my system
- Stick to compiler flags, don't manually unroll loops



```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123...
debian@debian:~/COMSC-142$ ./prime2 5000000
Time to allocate: 2.5e-05
Time to generate primes: 1.35757
348513 primes found.
debian@debian:~/COMSC-142$ ./prime3 5000000
Time to allocate: 2.4e-05
Time to generate primes: 1.52099
348513 primes found.
```

12.3. Memory Considerations

matrixVector.c

- wget <https://samsclass.info/COMSC-142/proj/matrixVector.c>

```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 80x19

//declare, allocate and fill input and output matrices
gettimeofday(&tstart, NULL);
int ** matrix = malloc(rows*sizeof(int *));
int ** result = malloc(rows*sizeof(int *));

//allocate matrices
for (i = 0; i < rows; i++){
    matrix[i] = allocateArray(cols);
    result[i] = allocateArray(cols);
}

//fill matrices
for (i = 0; i < rows; i++){
    fillArrayRandom(matrix[i], cols);
    fillArrayZeros(result[i], cols);
}
gettimeofday(&tend, NULL);
printf("Time to allocate and fill matrices: %g\n", getTime(tstart, tend));
```

matrixVector.c

```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 80x14

//allocate and fill vector
gettimeofday(&tstart, NULL);
int * vector = allocateArray(cols);
fillArrayRandom(vector, cols);
gettimeofday(&tend, NULL);
printf("Time to allocate vector: %g\n", getTime(tstart, tend));

//perform matrix-vector multiplication
gettimeofday(&tstart, NULL);
matrixVectorMultiply(matrix, vector, result, rows, cols);
gettimeofday(&tend, NULL);
printf("Time to matrix-vector multiply: %g\n", getTime(tstart, tend));
```

```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 57x5

debian@debian:~/COMSC-142$ ./matrixVector 10000 10000
Time to allocate and fill matrices: 2.04818
Time to allocate vector: 0.000149
Time to matrix-vector multiply: 2.46814
debian@debian:~/COMSC-142$
```

12.3.1. Loop Interchange

- Switches the order of inner and outer loops in nested loops in order to maximize cache locality
- GCC finds this difficult to do
 - Does not perform it by default
- It's up to programmers to do it manually

```
//cycles through every matrix column  
//in inner-most loop (inefficient)  
for (j = 0; j < col; j++){  
    for (i = 0; i < row; i++){  
        res[i][j] = m[i][j] * v[j];  
    }  
}
```

```
//cycles through every row of matrix  
//in inner-most loop  
for (i = 0; i < row; i++){  
    for (j = 0; j < col; j++){  
        res[i][j] = m[i][j] * v[j];  
    }  
}
```

Version	Program	Unoptimized	-01
Original	matrixVector	2.01	2.05
With Loop Interchange	matrixVector2	0.27	0.08

12.3.2. Some Other Compiler Optimizations for Improving Locality: Fission and Fusion

- Matrix filling is the next limitation

```
$ gcc -o matrixVector2 matrixVector2.c
$ ./matrixVector2 10000 10000
Time to allocate and fill matrices: 1.29203
Time to allocate vector: 0.000107
Time to matrix-vector multiply: 0.271369
```

```
//fill matrices
for (i = 0; i < rows; i++){
    fillArrayRandom(matrix[i], cols);
    fillArrayZeros(result[i], cols);
}
```

Loop Fission

- Break loops apart
- Allows multicore processors to assign different cores for each loop

```
//fill matrices  
for (i = 0; i < rows; i++){  
    fillArrayRandom(matrix[i], cols);  
    fillArrayZeros(result[i], cols);  
}
```

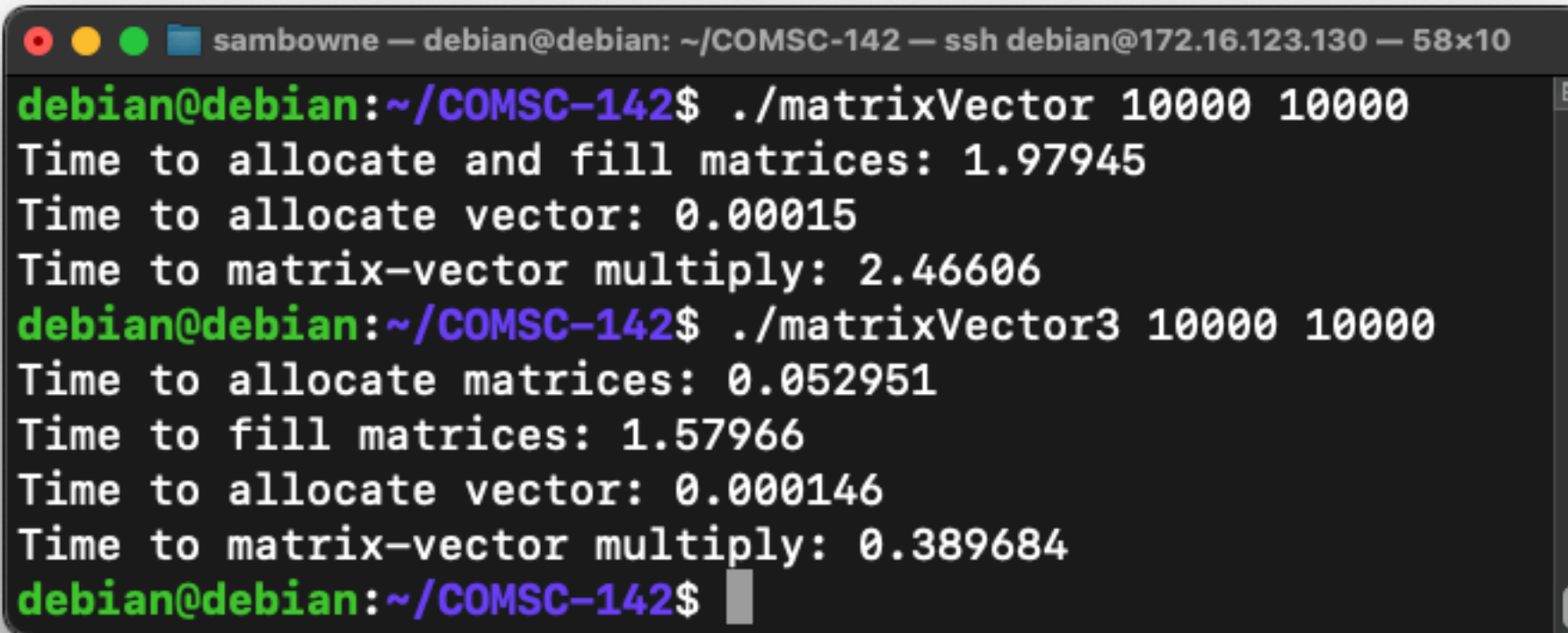
With loop fission

```
for (i = 0; i < rows; i++) {  
    fillArrayRandom(matrix[i], cols);  
}  
  
for (i = 0; i < rows; i++) {  
    fillArrayZeros(result[i], cols);  
}
```

Loop Fission

- FillArrayZeroes is not needed
- Modest improvement

```
for (i = 0; i < rows; i++) {  
    matrix[i] = allocateArray(cols);  
    result[i] = allocateArray(cols);  
}  
  
for (i = 0; i < rows; i++) {  
    fillArrayRandom(matrix[i], cols);  
    //fillArrayZeros(result[i], cols);  
    //no longer needed  
}
```

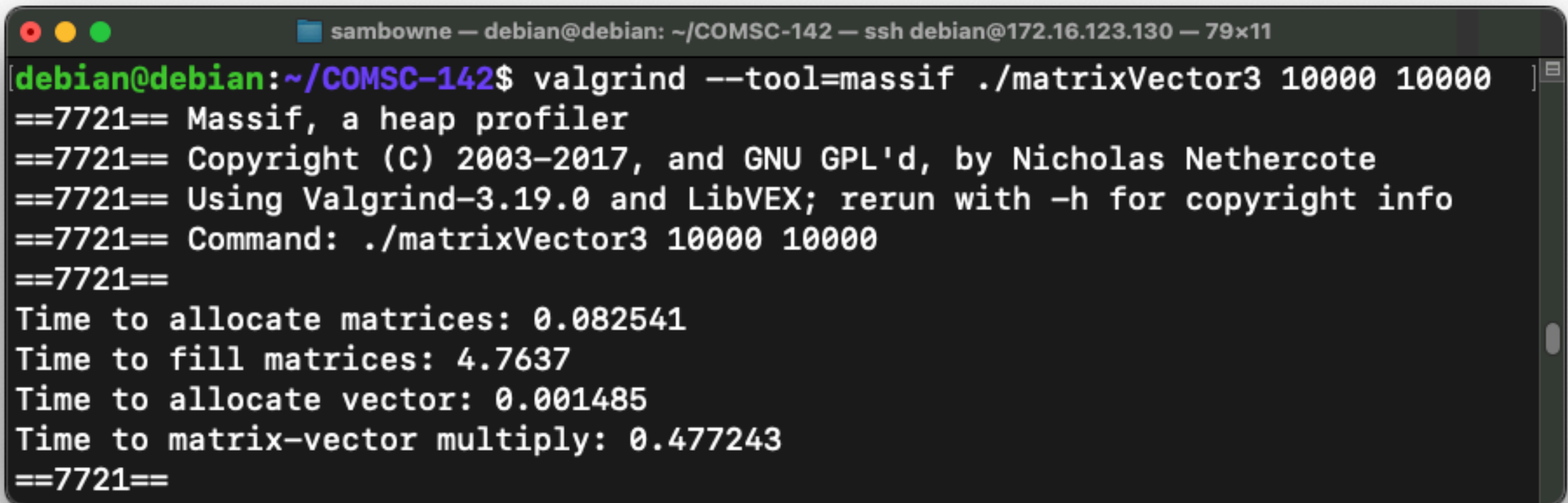


A terminal window titled 'sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 58x10'. The window shows the execution of two programs, 'matrixVector' and 'matrixVector3', with their respective timing outputs.

```
debian@debian:~/COMSC-142$ ./matrixVector 10000 10000  
Time to allocate and fill matrices: 1.97945  
Time to allocate vector: 0.00015  
Time to matrix-vector multiply: 2.46606  
debian@debian:~/COMSC-142$ ./matrixVector3 10000 10000  
Time to allocate matrices: 0.052951  
Time to fill matrices: 1.57966  
Time to allocate vector: 0.000146  
Time to matrix-vector multiply: 0.389684  
debian@debian:~/COMSC-142$
```


Massif

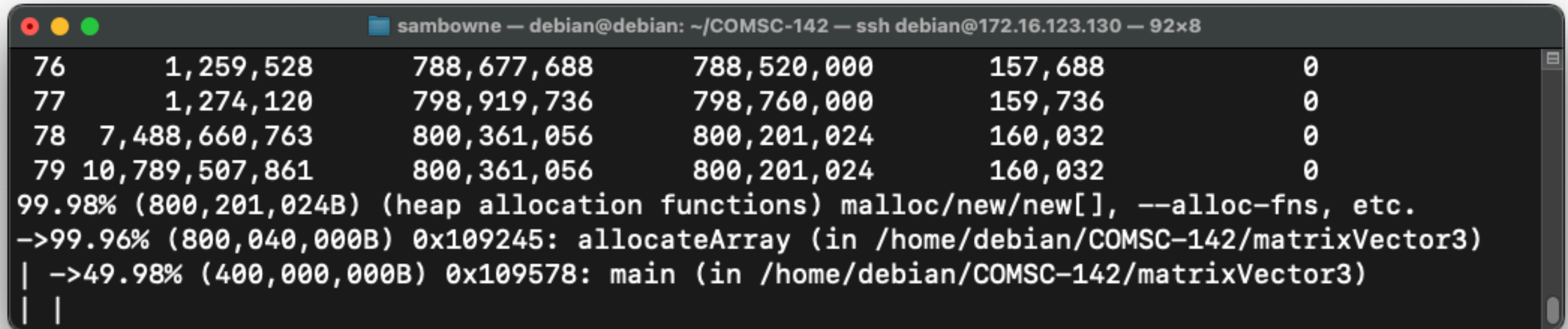
- Profiles how memory usage fluctuates

A terminal window with a dark background and light text. The window title bar shows 'sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 79x11'. The prompt is 'debian@debian:~/COMSC-142\$'. The command entered is 'valgrind --tool=massif ./matrixVector3 10000 10000'. The output shows Massif version 7721, copyright information, and timing statistics for memory allocation and matrix-vector multiplication.

```
sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 79x11
[debian@debian:~/COMSC-142$ valgrind --tool=massif ./matrixVector3 10000 10000 ]
==7721== Massif, a heap profiler
==7721== Copyright (C) 2003-2017, and GNU GPL'd, by Nicholas Nethercote
==7721== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==7721== Command: ./matrixVector3 10000 10000
==7721==
Time to allocate matrices: 0.082541
Time to fill matrices: 4.7637
Time to allocate vector: 0.001485
Time to matrix-vector multiply: 0.477243
==7721==
```

Massif

- Shows that 99.96 % of memory was used by **allocateArray**



The screenshot shows a terminal window with the title bar "sambowne — debian@debian: ~/COMSC-142 — ssh debian@172.16.123.130 — 92x8". The terminal displays the output of a Massif memory analysis. It shows a table of memory allocation data for lines 76 through 79, followed by a summary of heap allocation functions and a breakdown of memory usage by function.

Line	Size (B)	Address	Start (B)	End (B)	Count
76	1,259,528	788,677,688	788,520,000	157,688	0
77	1,274,120	798,919,736	798,760,000	159,736	0
78	7,488,660,763	800,361,056	800,201,024	160,032	0
79	10,789,507,861	800,361,056	800,201,024	160,032	0

99.98% (800,201,024B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->99.96% (800,040,000B) 0x109245: allocateArray (in /home/debian/COMSC-142/matrixVector3)
| ->49.98% (400,000,000B) 0x109578: main (in /home/debian/COMSC-142/matrixVector3)
| |

12.4. Key Takeaways and Summary

- Choose Good Data Structures and Algorithms
- Use Standard Library Functions Whenever Possible
- Optimize Based on Data and Not on Feelings
- Split Complex Code into Multiple Functions
- Prioritize Code Readability
- Pay Attention to Memory Use
- Compilers Are Constantly Improving

Kahoot!

Ch 12b