



Benchmarking and Profiling

Optimization Rationales

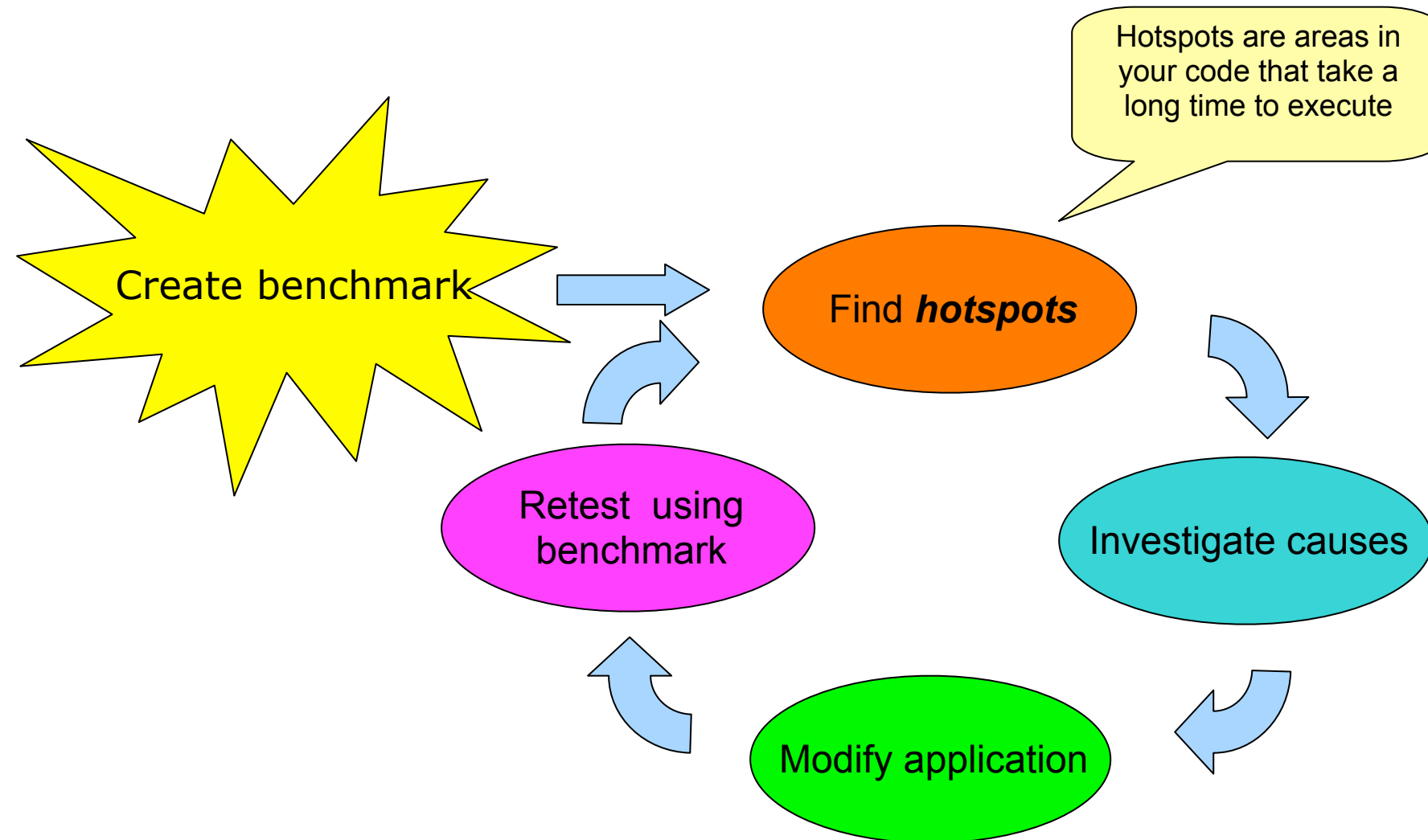
2

What are we optimizing?

- ▶ Optimization often degrades maintainability of the code
- ▶ Do not optimize, unless you have to
- ▶ Start by measuring performance
- ▶ Identify costs and optimize only when warranted by measurements
- ▶ Do not guess where the time goes, even experts can be wrong

Process

3



Some Optimization Challenges

4

- ▶ Must build the system first
 - ▶ Must know on what hardware/software environment it will run (different from development environment)
 - ▶ Use release builds (and not dev/debug)
 - for C, -O0 is good for debugging, but -O2 is free optimization for release
 - ▶ Measurements introduce noise (can we trust the tools)
-
- ▶ *Software optimization is an ongoing process that continues throughout the development of a system*

Benchmarks

5

Optimizations must be directed by measurements on benchmarks

Benchmarks should be:

- ▶ **Repeatable**

Must provide consistent measurements in the presence of other tasks in the system, of hardware variability, caching issues, et

- ▶ **Representative**

Must be typical of actual system load

Must exercise typical code paths (beware of slow paths and pathological cases)

- ▶ **Easy to run**

- ▶ **Verifiable**

Must have a checkable answer - to ensure that optimization do not break correctness

- ▶ **Deterministic**

Finding bottlenecks

6

Determine how system resources (memory and processor) are utilized to identify system-level bottlenecks

Measure the execution time for each component

Determine how the various components affect the performance of each other

Identify the most time-consuming code sequences

Determine how the program is executing at the processor level to identify microarchitecture-level performance problems

Tools overview

7

Timing mechanisms

- ▶ Stopwatch: time tool

System load monitors

- ▶ vmstat, iostat...

Software profiler

- ▶ Gprof...

Memory debugger/profiler

- ▶ libmemusage
- ▶ Valgrind...

Timing with `time`

8

time

- ▶ We used `time` in a previous lab
- ▶ The time tool tells how long a program took to execute:

```
$ time sleep 5  
real    0m5.319s  
user    0m0.001s  
sys     0m0.010s
```
- ▶ Divides into three different kinds of timing:
- ▶ real time: Wall-clock time from start to finish. Significantly affected by other processes.
- ▶ user time: Time spent on the CPU. Doesn't represent time spent waiting for I/O or for the operating system to respond to requests.
- ▶ sys time: Time the operating system spent handling requests by this process.

CPU load with top

9

top

- ▶ top gives an overview of processes running on the system
- ▶ Mostly helpful for diagnosing when other processes are taking precious resources.
- ▶ Use it to determine how idle the system is.

CPU load with top

10

Per-process stats

- ▶ PID:pid
- ▶ PR : priority
- ▶ NI : niceness
- ▶ TIME : total (system + user) spent since startup
- ▶ COMMAND : command that started the process
- ▶ USER : username of running process
- ▶ VIRT : virtual image size
- ▶ RES : resident size
- ▶ SHR : shared mem size
- ▶ %CPU : CPU usage
- ▶ %MEM : Memory usage

Memory/CPU load: vmstat

11

vmstat

- ▶ Default output is averages
- ▶ Stats relevant to CPU:
 - in : interrupts
 - cs : context switches
 - us : total CPU in user (including “nice”)
 - sy : total CPU in system (including irq and softirq) – wa : total CPU waiting
 - id : total CPU idle
- ▶ syntax: `vmstat [options] [delay [samples]]`

Profilers

12

Profiler show time elapsed in each function and its descendants

- ▶ number of calls, call-graph (some)

Profilers use instrumentation or sampling

	Sampling	Instrumentation
Overhead	Typically about 1%	High, may be 500% !
System-wide profiling	Yes, profiles all app, drivers, OS functions	Just application and instrumented DLLs
Detect unexpected events	Yes , can detect other programs using OS resources	No
Setup	None	Automatic ins. of data collection stubs required
Data collected	Counters, processor an OS state	Call graph , call times, critical path
Data granularity	Assembly level instr., with src line	Functions, sometimes statements
Detects algorithmic issues	No, Limited to processes , threads	Yes – can see algorithm, call path is expensive

Profiling with gprof

13

gprof

- ▶ the GNU profiler
- ▶ instrumentation-based profiling: requires instrumented binaries (compile time)
- ▶ gprof used to parse output file
- ▶ gcov
coverage tool – uses the same instrumentation
- ▶ gprof is instrumentation-based. Other tools exist:
oprofile: system-wide for Linux, sample-based
gperftools: per-process, sample-based

Using *gprof* GNU profiler

14

Compile and link your program with profiling on

- ▶ `gcc -g -pg -c myprog.c utils.c`
- ▶ `gcc -g -pg -o myprog myprog.o utils.o`

Run your program to generate a profile data file

- ▶ Program will run normally (but slower) and will write the profile data into a file called `gmon.out` just before exiting
- ▶ Program should exit using `exit()` function or by returning from `main` (NOT by segfaulting or otherwise crashing)

Run `gprof` to analyze the profile data

- ▶ `gprof a.out`

Using *gprof* GNU profiler

15

```
#include <stdio.h>
long id(long i) {
    long ret = 0;
    for (; i > 0; i--) ret++;
    return ret;
}
long mul(long x, long y) {
    long ret = 0;
    for (; x > 0; x--) ret += y;
    return ret;
}
long main() {
    long x = 0, y = 0;
    x = id(1073741824);
    y = id(1024);
    printf("%ld\n%ld\n", mul(x, y), x * y);
    return 0;
}
```

Understanding Flat Profile

16

The *flat profile* shows the total amount of time your program spent executing each function.

If a function was not compiled for profiling, and didn't run long enough to show up on the program counter histogram, it will be indistinguishable from a function that was never called

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
50.51	2.41	2.41	2	1.21	1.21	id
50.51	4.83	2.41	1	2.41	2.41	mul

How gprof works

17

Instruments program to count calls

Watches the program running, samples the PC every 0.01 sec

- ▶ **Statistical inaccuracy: fast function may take 0 or 1 samples**
- ▶ **Run should be long enough comparing with sampling period**
- ▶ **Combine several gmon.out files into single report**

The output from gprof gives no indication of parts of your program that are limited by I/O or swapping. Samples of the program counter are taken at fixed intervals of run time

number-of-calls figures are derived by counting, not sampling. They are accurate and do not vary if program is deterministic

Profiling with inlining and other optimizations needs care

libmemusage

18

The GNU C library comes with a memory usage profiler

- ▶ Compile however you wish
- ▶ Run program with libmemusage.so:
`env LD_PRELOAD=/lib/libmemusage.so ./a.out`
- ▶ Summary of all mallocs and frees:

Memory usage summary: heap total: 1048576, heap peak: 1024,
stack peak: 96

	total	calls	total memory	failed	calls
malloc		1024	1048576		0
realloc		0	0		0
calloc		0	0		0
free		1024	1048576		

libmemusage for debugging

19

- ▶ Memory leaks occur when you have more mallocs than frees!

Memory usage summary: heap total: 1048576, heap peak: 1048576, stack peak: 96

	total calls	total memory	failed calls
malloc	1024	1048576	0
realloc	0	0	0
calloc	0	0	0
free	0	0	

Valgrind Toolkit

20

Memcheck is memory debugger

- ▶ detects memory-management problems

Cachegrind is a cache profiler

- ▶ performs detailed simulation of the L1, D1 and L2 caches in your CPU

Massif is a heap profiler

- ▶ performs detailed heap profiling by taking regular snapshots of a program's heap

Helgrind is a thread debugger

- ▶ finds data races in multithreaded
- ▶ programs

Memcheck Features

21

When a program is run under Memcheck's supervision, all reads and writes of memory are checked, and calls to malloc/new/free/delete are intercepted

Memcheck can detect:

- ▶ Use of uninitialised memory
- ▶ Reading/writing memory after it has been free'd
- ▶ Reading/writing off the end of malloc'd blocks
- ▶ Reading/writing inappropriate areas on the stack
- ▶ Memory leaks – where pointers to malloc'd blocks are lost forever
- ▶ Passing of uninitialised and/or unaddressible memory to system calls
- ▶ Mismatched use of malloc/new/new [] vs free/delete/delete []
- ▶ Overlapping src and dst pointers in memcpy() and related functions
- ▶ Some misuses of the POSIX pthreads API

Memcheck Example

22

```
#include <stdio.h>
#include <stdlib.h>
char *f() {
    return malloc(17);
}
#define MM 1000000
int main() {
    int i, j;
    int *p = malloc(sizeof(int) * 10);
    p[12] = 6;
    j = i + 3;
    if (i > 0) printf("i > 0\n");
    f();
    free(p);
    return 0;
}
```

Memcheck Example (Cont.)

23

Compile the program with `-g` flag:

- ▶ `gcc -g test.c -o a.out`

Execute `valgrind` :

- ▶ `valgrind --tool=memcheck --leak-check=yes a.out 2> log`

View log

Memcheck Example (Cont.)

24

Invalid write of size 4

at 0x4005B2: main (test.c:14)

Address 0x4da9070 is 8 bytes after a block of size 40 alloc'd

at 0x4028693: malloc (in memcheck-amd64-linux.so)

by 0x4005A5: main (test.c:13)

Conditional jump or move depends on uninitialised value(s)

at 0x4005C5: main (test.c:17)

Memcheck Example (Cont.)

25

HEAP SUMMARY:

in use at exit: 17 bytes in 1 blocks
total heap usage: 2 allocs, 1 frees, 57 bytes allocated

17 bytes in 1 blocks are definitely lost in loss record 1 of 1
at 0x4028693: malloc (in memcheck-amd64-linux.so)
by 0x400591: f (test.c:6)
by 0x4005DA: main (test.c:19)

LEAK SUMMARY:

definitely lost: 17 bytes in 1 blocks
indirectly lost: 0 bytes in 0 blocks
possibly lost: 0 bytes in 0 blocks
still reachable: 0 bytes in 0 blocks
suppressed: 0 bytes in 0 blocks

Massif tool

26

Massif is a heap profiler - it measures how much heap memory programs use. It can give information about:

- ▶ **Heap blocks**
- ▶ **Heap administration blocks**
- ▶ **Stack sizes**

Help to reduce the amount of memory the program uses

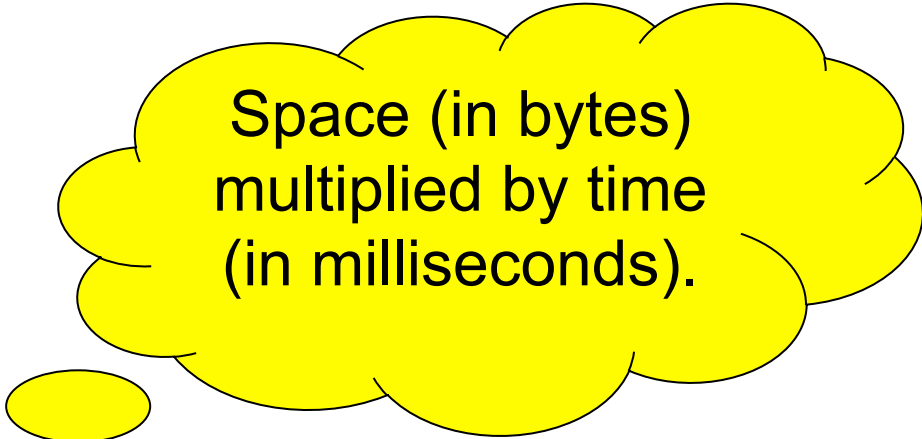
- ▶ **smaller program interact better with caches, avoid paging**

Detect leaks that aren't detected by traditional leak-checkers, such as Memcheck

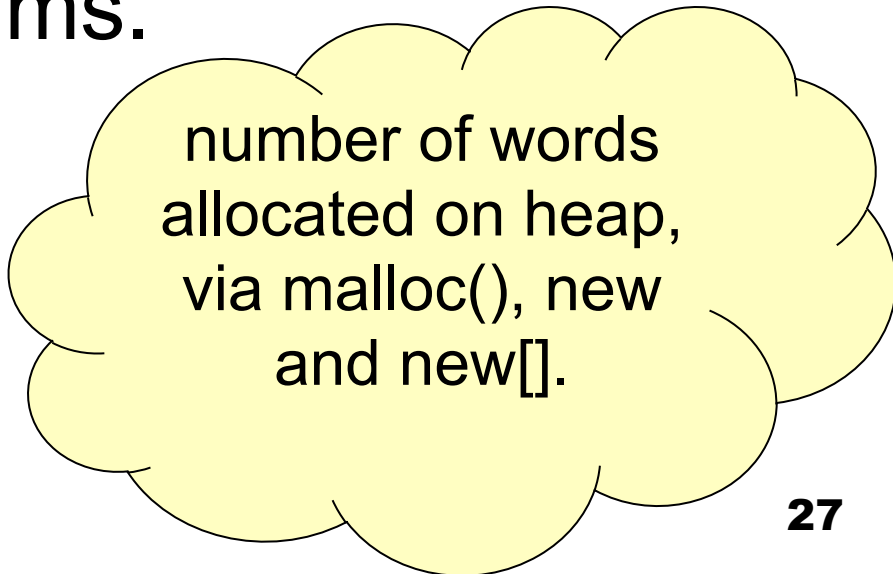
- ▶ **That's because the memory isn't ever actually lost - a pointer remains to it - but it's not in use anymore**

Executing Massif

- Run `valgrind --tool=massif prog`
 - Produces data file
- Use `ms_print` to handle data
 - Produces following:
 - Summary
 - Graph Picture
 - Report
- Summary will look like this:
 - Total spacetime: 2,258,106 ms.
 - Heap: 24.0%
 - Heap admin: 2.2%
 - Stack (s): 73.7%

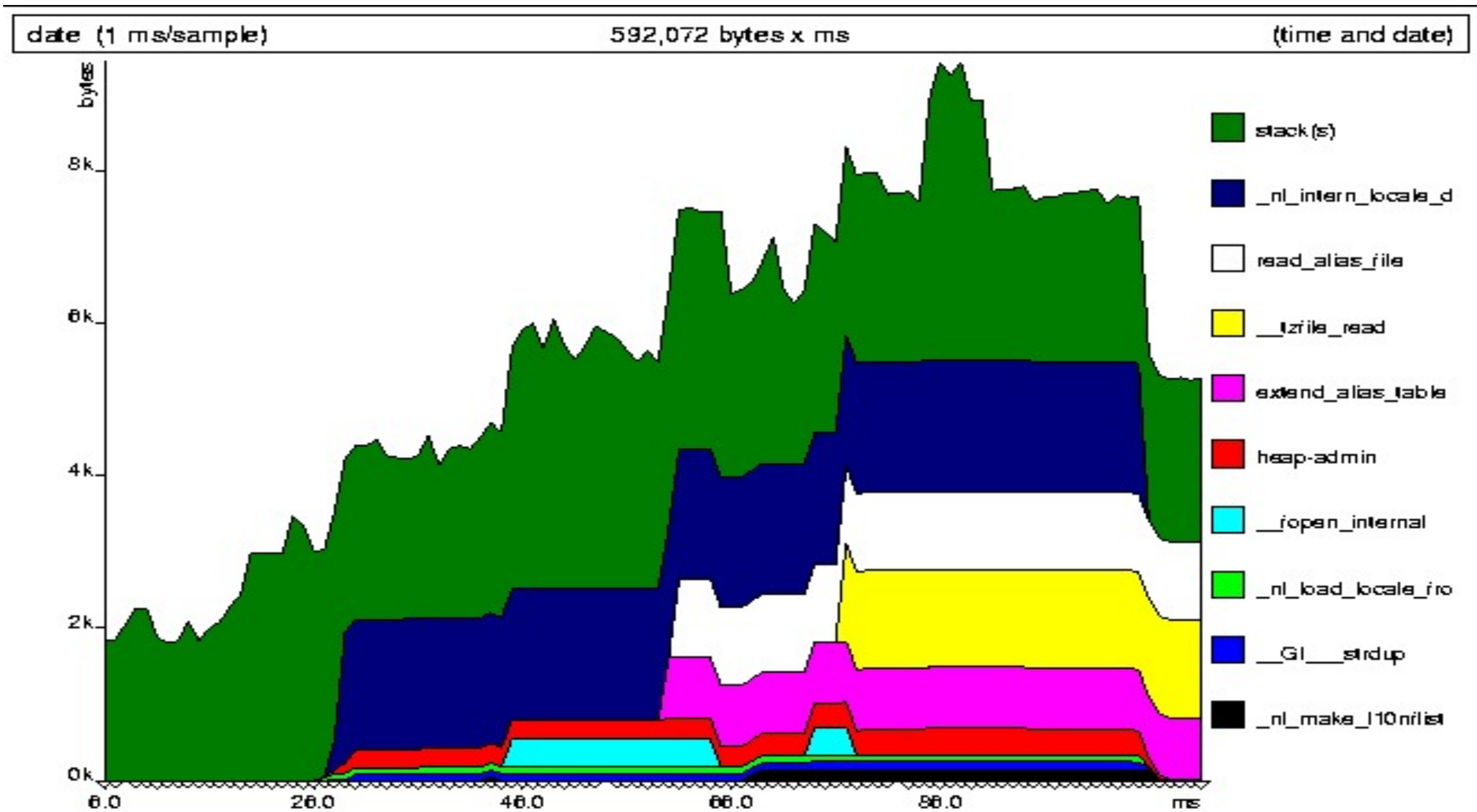


Space (in bytes)
multiplied by time
(in milliseconds).



number of words
allocated on heap,
via `malloc()`, `new`
and `new[]`.

Spacetime Graphs



Spacetime Graph (Cont.)

29

Each band represents single line of source code

It's the height of a band that's important

Triangles on the x-axis show each point at which a memory census was taken

- ▶ Not necessarily evenly spread; Massif only takes a census when memory is allocated or de-allocated
- ▶ The time on the x-axis is wall-clock time
 - not ideal because can get different graphs for different executions of the same program, due to random OS delays

Text/HTML Report example

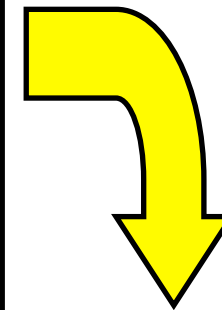
30

Contains a lot of extra information about heap allocations that you don't see in the graph.

Heap allocation functions accounted for 50.8% of measured spacetime

Called from:

- 22.1%: 0x401767D0: _nl_intern_locale_data (in /lib/i686/libc-2.3.2.so)
- 8.6%: 0x4017C393: read_alias_file (in /lib/i686/libc-2.3.2.so)
- *(several entries omitted)*
- and 6 other insignificant places



Shows places in the program where most memory was allocated

Context accounted for 22.1% of measured spacetime

0x401767D0: _nl_intern_locale_data (in /lib/i686/libc-2.3.2.so)

Called from:

- 22.1%: 0x40176F95: _nl_load_locale_from_archive (in /lib/i686/libc-2.3.2.so)

Summary

31

Overall view of system performance:

- ▶ top, vmstat

Course view of process timing:

- ▶ time

Fine view of program timing:

- ▶ gprof, gcov
- ▶ oprofile
- ▶ gperftools

Coarse memory profiling:

- ▶ libmemusage

Memory debugging:

- ▶ valgrind (memcheck, massif)
- ▶ memscrambler, mudflap

Summary

32

Overall view of system performance:

- ▶ top, vmstat

Course view of process timing:

- ▶ time

Fine view of program timing:

- ▶ gprof, gcov
- ▶ oprofile
- ▶ gperftools

Coarse memory profiling:

- ▶ libmemusage

Memory debugging:

- ▶ valgrind (memcheck, massif)
- ▶ memscrambler, mudflap