CS 33 Week 6

Section 1G, Spring 2015
Prof. Eggert (TA: Eric Kim)
v1.0

Announcements

- HW 4: Due 5/8 (**Today!**)
- Lab 3: Due 5/13 (Wednesday)
  - "Smashing" Lab
- **MT 2: 5/19**
  - Pushed back!

Lab 3: "Getting Started"

- Tip: If you're having trouble getting started on Lab 3, check out the guide here:
  - http://www.eric-kim.net/cs33_page/
    - "Getting started with Lab 3"

Overview

- Program Performance/Optimization
- Memory Hierarchy
- Lab 3 ("smashing" lab)

Program Performance

- So far, we have reasoned about code performance at a high-level
  - Example: binary search of a sorted array of length N has O(log N) behavior

In this class: reason at the compiler-level *and* processor-level!

Program Performance

- Compiler-level considerations
  - gcc compiler is very conservative
  - Will not optimize if it compromises program behavior
- Skill: How to write C code to encourage compiler optimizations?

Program Performance

- Processor-level considerations
  - Instruction pipelining
  - Exploiting parallelism
  - Out-of-Order execution (OoO)
- Skill: How to write C code to "encourage" compiler to generate assembly code that fully-utilizes processor?

Program Example

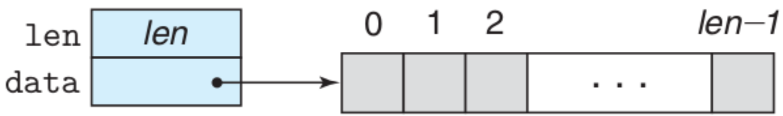


Figure 5.3 **Vector abstract data type.** A vector is represented by header information plus array of designated length.

code/opt/vec.h

```
/* Create abstract data type for vector */
typedef struct {
    long int len;
    data_t *data;
} vec_rec, *vec_ptr;
```

code/opt/vec.h

data_t can be an int, float, or double

Program Example

```
/* Implementation with maximum use of data abstraction */
void combine1(vec_ptr v, data_t *dest)
{
    long int i;

    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

Figure 5.5 **Initial implementation of combining operation.** Using different declarations of identity element *IDENT* and combining operation *OP*, we can measure the routine for different operations.

IDENT is either 0 (add), or 1 (mult).
OP is either + or *.

Loop Unrolling

- Reason 1: Less overhead due to loop bookkeeping (ie "i<n", "i++").
- Reason 2: Exposes structure in code, allowing compiler to perform additional optimizations

Loop Unrolling

```
void combine5(vec_ptr v, data_t *dest) {
  long int i; long int length = vec_length(v);
  long int limit = length-1;
  data_t *data = get_vec_start(v);
  data_t acc = IDENT;
  /* Combine 2 elements at a time */
  for (i = 0; i < limit; i+=2)
    acc = (acc OP data[i]) OP data[i+1];
  /* Finish any remaining elements */
  for (; i < length; i++)
    acc = acc OP data[i];
  *dest = acc;
}
```

Can unroll loop further (ie k > 2)

Loop Unrolling

- Speedup is due to reduced overhead relating to loop maintenance/bookkeeping
- Also: allows *reassociation optimization* (revisit later)

Multiple Accumulators

- Modern processors have fully pipelined add/mult
- Critical bottleneck in combine: we have to write to acc after each mult.
  - Why is this a problem?

    *Constrains each mult to have to occur **sequentially**.*

  - How can we overcome?

    *Remove this constraint! Write to **separate** accumulators.*

Multiple Accumulators

During each iteration of loop body, processor can perform both adds/mults in a **fully-pipelined** manner.

Question: Suppose mult takes 3 clock cycles (ie 3 stages).
How many cycles does it take to perform one iteration of the loop body in
(1) A fully-pipelined mult? **4 cycles**
(2) A non-pipelined mult? **6 cycles**

```
/* Unroll loop by 2, 2-way parallelism */
void combine6(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    long int limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc0 = IDENT;
    data_t acc1 = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc0 = acc0 OP data[i];
        acc1 = acc1 OP data[i+1];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 OP data[i];
    }
    *dest = acc0 OP acc1;
}
```

Pipelining

Not convinced? Suppose we had unrolled combine() **4x**, ie there are <u>four</u> lines in the loop body.

Question: Suppose mult takes 3 clock cycles (ie 3 stages). How many cycles does it take to complete one loop iteration in:

(1) A fully-pipelined functional unit? **6 cycles!**

(2) A non-pipelined functional unit? **12 cycles**

Pipelining

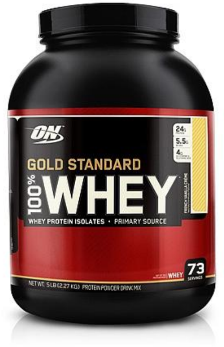
```
1 2 3
a
b a
c b a
d c b
  d c
    d
=> 6 cycles
```

vs.

```
1 2 3
a
  a
    a
b
  b
    b
...
=> 12 cycles
```

Reassociation Transformation

- Another way to exploit pipelined acc/mult
  - Assume we are working with integers, not floats
- Recall: Associative Property
  - a*(b*c) = (a*b)*c
- Idea: "Move" parenthesis for huge gains!



Reassociation Transformation

Used to be:

```
acc = (acc OP data[i])
  OP data[i+1];
```

```
/* Change associativity of combining operation */
void combine7(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    long int limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc = acc OP (data[i] OP data[i+1]);
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

Why does this improve things?

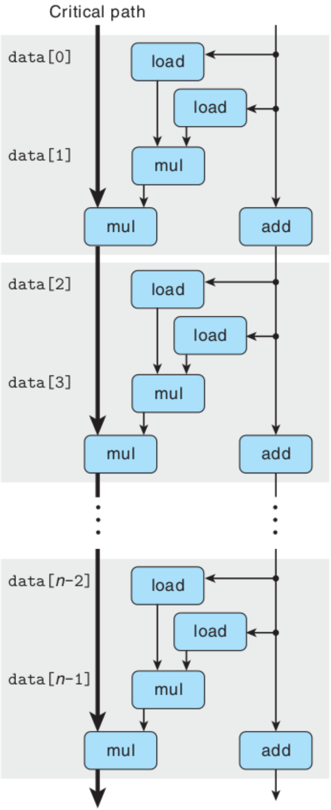


Figure 5.30
Data-flow representation of `combine7` operating on a vector of length n. We have a single critical path, but it contains only $n/2$ operations.

During each iteration, the first multiplication:

`(data[i] OP data[i+1])`

is <u>independent</u> of the second multiplication:

`acc OP (data[i] OP data[i+1])`

In particular: at iteration i, while we are computing:

`acc OP (data[i] OP data[i+1])`

We can also start computing <u>next</u> iteration's mult:

`(data [i+1] OP data[i+2])`

Association: Int vs Float

- Question: Can we play the reassociation trick if we were working with floating point values?

Answer: No!

Question: Give an example where the associative property fails to hold for floating point values.

Things to watch out for

- Keep these things in the back of your mind while coding high-performance software
- Register spilling
- Branch misprediction penalties

Register spilling

- Compiler aims to keep all local variables on registers
- Too many local variables -> Gotta store em' on the stack
  - Penalties: Instead of read/write to registers (instant!), have to read/write to memory (not instant!)

Branch misprediction

- Modern processors employ *speculative execution* to fully utilize CPU for branches
  - Fancy term for: guess which branch to take
- If we guess wrong, then we need to **undo** the instructions we executed!
  - Flush the pipeline, and start again at mispredicted instruction

Some numbers for fun...

"Numbers every programmer should know"

operation ▾	latency (ns) ▾
L1 cache reference	0.5
Branch mispredict	5
L2 cache reference	7
Mutex lock/unlock	25
Main memory reference	100
Compress 1K bytes with Zippy	3,000
Send 2K bytes over 1 Gbps network	20,000
Read 1 MB sequentially from memory	250,000
Round trip within same datacenter	500,000
Disk seek	10,000,000
Read 1 MB sequentially from disk	20,000,000
Send packet CA->Netherlands->CA	150,000,000 ← (1.5e-7 seconds)

Latency numbers every programmer should know

Jeff Dean (http://research.google.com/people/jeff/)

(More interesting numbers…)

Lets multiply all these durations by a **billion**:

Minute:

L1 cache reference 0.5 s One heart beat (0.5 s)

Branch mispredict 5 s Yawn

L2 cache reference 7 s Long yawn

Mutex lock/unlock 25 s Making a coffee

Hour:

Main memory reference 100 s Brushing your teeth

Compress 1K bytes with Zippy 50 min One episode of a TV show (including ad breaks)

Day:

Send 2K bytes over 1 Gbps network 5.5 hr From lunch to end of work day

(More interesting numbers…)

Week

SSD random read 1.7 days A normal weekend

Read 1 MB sequentially from memory 2.9 days A long weekend

Round trip within same datacenter 5.8 days A medium vacation

Read 1 MB sequentially from SSD 11.6 days Waiting for almost 2 weeks for a delivery

Year

Disk seek 16.5 weeks A semester in university

Read 1 MB sequentially from disk 7.8 months Almost producing a new human being

The above 2 together 1 year

Decade

Send packet CA->Netherlands->CA 4.8 years Average time it takes to complete a bachelor's degree

From: http://architects.dzone.com/articles/every-programmer-should-know

Lab 3 ("smashing" lab)

Lab 3 ("smashing lab")

- Reminder: Due **Wednesday**
- Mix of source code reading, gdb inspecting, and exploit generation (the fun part!)
- Start now!

Lab 3: "Getting Started"

- Tip: If you're having trouble getting started on Lab 3, check out the guide here:
  - http://www.eric-kim.net/cs33_page/
    - "Getting started with Lab 3"

Additional Resources

- For a review of stack smashing, canaries, etc., check out my Week 4 discussion notes:
  - http://eric-kim.net/cs33_page/
    - Starts on slide 30, "Bounds Checking"

Tips on Lab

- How does '-fstack-protector-strong' work?
- How does the Address Sanitizer (-fsanitize=address) work?
    - https://code.google.com/p/address-sanitizer/

Recall: Stack Randomization

- Many exploits rely on knowing addresses of local variables/buffers on the stack
    - Easiest: absolute memory addrs, ie `0xffffc29c`

Defense: <u>Randomize</u> stack addresses!

Don't grow stack at some fixed memory location (say, `0xffffff00`). Instead, add random offsets for each program execution (`0xffffff00 + rand()`).

(An instance of ASLR: Address Space Layout Randomization)

Stack Randomization

- ASLR isn't fool proof

<u>**Scenario:**</u> Suppose attacker needs to guess the stack address of a local buffer (ie to point return address back to the buffer).
How can attacker do this on a system with stack randomization?

Answer: NOP-sleds!

Stack Randomization

- Question: Do seasnet.ucla.edu machines employ stack randomization?
  - How to check?

Related Question: Suppose a machine does employ stack randomization. On this machine, you run gdb on a program.
Within gdb, will stack addresses still be random?

Writing Config Files

- In the lab, at one point you'll need to handcraft a configuration file
  - src/thttpd-no -p 50000 -D -C mybadness.txt

Contents of: mybadness.txt
0000000000000
0000000000000
…
0000000

Suppose the program reads this text file into a "char line[LEN]" array.
Question: What bytes get written to the array?

Answer: `0x30 0x30 0x30 0x30 … 0x30 0x30`
Text files are ASCII-encoded:
"0" -> 0x30   "1" -> 0x31, "a" -> 61, etc.

Writing Config Files

- How to write arbitrary bytes to the array?
- want to write this:
  - `0xffffabcd`
- not this:
  - "0x66 0x66 0x66 0x66 0x61 0x62 0x63 0x64"
    - `0x6666666661626364`

Writing Config Files

- Simple way: Write a C program to write your config file
  - stdio.h contains basic file read/write library
  - http://www.cplusplus.com/reference/cstdio/fputc/

stdio.h: Basic C I/O

Instead of writing ASCII values 'a'-'z', can directly write raw bytes:

```
fputc(0xff, pFile);
fputc(0xff, pFile);
fputc(0xab, pFile);
fputc(0xcd, pFile);
```

(Little-endian vs big-endian - something to worry about for exploit?)

```
/* fputc example: alphabet writer */
#include <stdio.h>

int main ()
{
  FILE * pFile;
  char c;

  pFile = fopen ("alphabet.txt","w");
  if (pFile!=NULL) {

    for (c = 'A' ; c <= 'Z' ; c++)
      fputc ( c , pFile );

    fclose (pFile);
  }
  return 0;
}
```

Useful functions: fopen(), fclose(), fputc()

Q8: Generating .s files

- Odd: running the commands from spec results in assembly being saved to .o files, rather than .s files

Workaround: Just rename.o files to .s files (see Piazza Question @290)

```
$ make clean
$ make CFLAGS='-m32 -S -O2 -fno-inline -fstack-protector-strong'
$ mv src/thttpd.o src/thttpd-sp.s
# repeat for thttpd-as.s, thttpd-no.s
```

Classic Stack Smash Attack

- Necessary Ingredients
  - Find a vulnerable buffer to overflow
  - Handcraft exploit code (ie x86 op codes)
  - Write exploit code to buffer
    - Overwrite saved return address on stack to the *stack* address of your exploit code
- Obstacles:
  - NX bit, stack randomization

Idea: Utilize library!

- Trick victim to execute arbitrary code
  - Hard! Lots of defenses.
- Trick victim to execute already-available code
  - Easier! Might be enough to achieve exploit.
  - Return-oriented programming (ROP) is built on top of this principle

<unistd.h>

- POSIX operating system API
  - defined by *all* variants of UNIX, including MacOSX, GNU/Linux, etc.
- Defines interface to talk to operating system
  - File read/write, device handling, system calls, etc

http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/unistd.h.html

<unistd.h>

```
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
```

http://pubs.opengroup.org/onlinepubs/7908799/xsh/execl.html

Allows C programs to ask the OS to run programs.

Note: The shell (ie terminal) is also a program!

Can ask the shell to run programs for us!

Example: execl

```
#include <unistd.h>
int main() {
    execl("/bin/sh", "/bin/sh", "-c", "ls", NULL);
}
```

```
$ gcc -o try_execl try_execl.c
$ ./try_execl
try_execl try_execl.c
```