CS 33 Discussion 6

May 12, 2017

Agenda

Midterm Review

Program Performance

Caching

Smashing Lab

ILP

Floating Point

Three types:

1. Normalised: Exp is not all 0s and not all 1s
2. Denormalised: Exp is all 0s
3. Special Values: Exp is all 1s

Special values can represent NaN, infinity.

Q2

Suppose we extend the bitwise operations ^, &, | and ~ to operate on floating-point values by applying these bitwise operations to their representations.

For example, since the binary representation of 0.1f is 0x3dcccccd, and ~0x3dcccccd == 0xc2333332, and 0xc2333332 represents -44.799995f, then ~0.1f would yield -44.799995f and ~-44.799995f would yield 0.1f.

Recall that the general rule for floating point operations is that NaNs are infectious, i.e., that if one or both inputs to a floating-point operation is a NaN, then the operation yields a NaN. Which (if any) of the bitwise operations ^, &, | and ~ are infectious on NaNs? Explain.

A2

OR is the only infectious bitwise operator. NaNs are identified by an exponent field that is all 1s and a non-zero fractional field. Thus, we are looking for a bitwise operation that will, with certainty, result in an exponent field of all 1s and a non-zero fractional field.

With |, if either of the operands is a NaN, the resulting value must be a NaN because anything OR'd with a 1 is a 1.

Any of the other operations can or must yield non- NaN values when applied to a NaN. For example: <bitwise ~>NaN will result in an all 0 fractional field, NaN <bitwise ^> 0xFFF...FF has the same effect as <bitwise ~>NaN, NaN <bitwise &> 0 = 0.

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

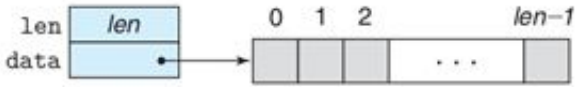


Figure 5.3 **Vector abstract data type.** A vector is represented by header information plus array of designated length.

code/opt/vec.h

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

code/opt/vec.h

data_t can be an int, float, or double

Program Example

```
1   /* Implementation with maximum use of data abstraction */
2   void combine1(vec_ptr v, data_t *dest)
3   {
4       long int i;
5
6       *dest = IDENT;
7       for (i = 0; i < vec_length(v); i++) {
8           data_t val;
9           get_vec_element(v, i, &val);
10          *dest = *dest OP val;
11      }
12  }
```

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 *.

Comparing CPE (cycles per element)

Function	Method	Integer(+)	Integer(*)	Floating point(+)	Floating point (*)
combine1	Abstract unoptimised	22.68	20.02	19.98	20.18
combine1	Abstract -O1	10.12	10.12	10.17	11.14

Eliminating Loop Inefficiencies

```
1   /* Move call to vec_length out of loop */
2   void combine2(vec_ptr v, data_t *dest)
3   {
4       long int i;
5       long int length = vec_length(v);
6
7       *dest = IDENT;
8       for (i = 0; i < length; i++) {
9           data_t val;
10          get_vec_element(v, i, &val);
11          *dest = *dest OP val;
12      }
13  }
```

Figure 5.6 **Improving the efficiency of the loop test.** By moving the call to `vec_length` out of the loop test, we eliminate the need to execute it on every iteration.

Reducing Procedure Calls

code/opt/vec.c

```
1   data_t *get_vec_start(vec_ptr v)
2   {
3       return v->data;
4   }
```

code/opt/vec.c

```
1   /* Direct access to vector data */
2   void combine3(vec_ptr v, data_t *dest)
3   {
4       long int i;
5       long int length = vec_length(v);
6       data_t *data = get_vec_start(v);
7
8       *dest = IDENT;
9       for (i = 0; i < length; i++) {
10          *dest = *dest OP data[i];
11      }
12  }
```

Figure 5.9 **Eliminating function calls within the loop.** The resulting code runs much faster, at some cost in program modularity.

Eliminating Unneeded Memory References

```
1   /* Accumulate result in local variable */
2   void combine4(vec_ptr v, data_t *dest)
3   {
4       long int i;
5       long int length = vec_length(v);
6       data_t *data = get_vec_start(v);
7       data_t acc = IDENT;
8
9       for (i = 0; i < length; i++) {
10          acc = acc OP data[i];
11      }
12      *dest = acc;
13  }
```

Figure 5.10 **Accumulating result in temporary.** Holding the accumulated value in local variable acc (short for "accumulator") eliminates the need to retrieve it from memory and write back the updated value on every loop iteration.

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

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

```
1  /* Unroll loop by 2, 2-way parallelism */
2  void combine6(vec_ptr v, data_t *dest)
3  {
4      long int i;
5      long int length = vec_length(v);
6      long int limit = length-1;
7      data_t *data = get_vec_start(v);
8      data_t acc0 = IDENT;
9      data_t acc1 = IDENT;
10
11     /* Combine 2 elements at a time */
12     for (i = 0; i < limit; i+=2) {
13         acc0 = acc0 OP data[i];
14         acc1 = acc1 OP data[i+1];
15     }
16
17     /* Finish any remaining elements */
18     for (; i < length; i++) {
19         acc0 = acc0 OP data[i];
20     }
21     *dest = acc0 OP acc1;
22 }
```

Q3

The function stricmp(A, B) compares the two strings A and B ignoring case, and returns an int.
If we let a = the lowercased version of A and
b = the lowercased version of B, then stricmp(A, B) returns a negative number if a compares less than b, a positive number if a compares greater than b, and zero otherwise. This function compares strings byte by byte, and assumes only the 52 ASCII letters. Consider the following stricmp implementation.

```
#include <string.h>
#define min(a, b) ((a) < (b) ? (a) : (b))
Char cvtlower (char c){
    if ('A' <= c && c <= 'Z')
        return c - 'A' + 'a';
    return c;
}
```

```
int stricmp (char const *a, char const *b){
  for (size_t i = 0;i < min(strlen(a), strlen(b)); i++)
        if (cvtlower(a[i]) < cvtlower(b[i]))
            return -1;
        else if (cvtlower(a[i]) > cvtlower(b[i]))
            return 1;
    return 0;
}
```

Propose two optimizations of this code, at least one of which is likely to improve performance greatly and the other at least somewhat. Explain why the former is likely to be better than the latter.

A3

Significant Improvement:
Hoist out strlen(a) - This can be done either by doing:

```
int min = min(strlen(a), strlen(b));
for (size_t i = 0; i < min; i++)
```

...or frankly even:

```
int len_a = strlen(a);
int len_b = strlen(b);
for (size_t i = 0; i < min(len_a, len_b); i++)
```

The reason that this provides such a considerable improvement is because strlen will perform n operations where n is the length of the string. Thus, for each iteration of the loop (say there are m iterations), we are performing two strlens. This means the function is doing 2*n*m operations as a result of the strlen. When strlen is hoisted out, the function only does 2*n operations as a result of the strlen.

Minor Improvement:
Reduce the number of calls to cvtlower - This can be done as follows:

```
for (size_t i = 0; i < min(strlen(a), strlen(b)); i++) {
char lower_a = cvtlower(a[i]);
char lower_b = cvtlower(b[i]);
if (lower_a < lower_b))
        return -1;
else if (lower_a > lower_b)
        return 1; }
```

Previously, each loop was performing 4 calls to cvtlower(), which is an O(1) function. This means cvtlower was contributing a total of 4*m operations. Now, it only performs 2 calls per iteration for a total of 2*m operations. Loop unrolling could also have also provided a minor decrease in the number of loop overhead operations.

Q4

Consider the following C function and its translation to x86-64 assembly language. The C function returns a[i] in the typical case where i is in range, and returns 0 otherwise:

```
int subscript (int *a, unsigned i, unsigned int n){
        if (0 <= i && i < n)
                return a[i];
        return 0;
}
```

```
subscript:
        xorl    %eax, %eax
        cmpl    %edx, %esi
        jnb     .L2
        movl    %esi, %esi
        movl    (%rdi,%rsi,4), %eax
.L2:
        ret
```

4a. How can this code be correct? The source has two comparisons, but the assembler has just one.

4b. Why aren't conditional moves helpful for improving this code's performance? Explain.

A4

a. The comparison of 0 <= i is unnecessary and will be optimized out since i is unsigned which means this condition will always be true.
b. The typical case for subscript is that the index is within the bounds. In this code snippet, dynamic branch prediction will generally predict that the branch will not be taken and as a result, it will speculatively perform the instructions for returning a[i]. Assuming a typical access pattern for arrays, this prediction will be successful most of the time, so conditional moves won't improve much.

Amdahl’s Law

- Talks about performance gains that can be obtained by targeted optimizations.
- The main idea is that when one part of the system speeds up, effect on overall system performance depends on how significant the part was and how much it sped up.

Amdahl's Law

- Consider a system in which executing some application requires time T_{old}. Suppose some part of the system requires a fraction α of this time, and that we improve its performance by a factor of k. That is, the component originally required time αT_{old}, and it now requires time $(\alpha T_{old})/k$.
- The overall execution time would thus be

$$T_{new} = (1 - \alpha) * T_{old} + (\alpha T_{old}) / k$$

$$= T_{old} [(1 - \alpha) + \alpha / k]$$

Amdahl’s Law

From this, we can compute the speedup

$$S = T_{old} / T_{new}$$

$$S = 1 / ((1 - \alpha) + \alpha/k)$$

Amdahl’s law simplified - “To significantly speed up the entire system, we must improve the speed of a very large fraction of the overall system.”

Q5

Suppose your program has three parts that are done in sequence and take 0.5, 0.3, and 0.2 of the time respectively. You can parallelize the first part and speed it up by a factor of 2. Or you can parallelize the second part and speed it up by a factor of 8. Use Amdahl's law to calculate which of these two will give you better performance and why. Suppose you can do both parallelizations: how much will your performance improve compared to the original, or to either parallelization alone? Show your work.

A5

Original:
T0 = 0.5 + 0.3 + 0.2

Parallelize the first part by a factor of 2:
T1 = **0.5/2** + 0.3 + 0.2 = 0.75

Parallelize the second part by a factor of 8:
T2 = 0.5 + **0.3/8** + 0.2 = 0.5 + 0.0375 + 0.2 = 0.7375

Parallelize both parts:
T3 = 0.5/2 + 0.3/8 + 0.2 = 0.4875
T3 < T2 < T1

Q6

Let d = the number 0.1 in C (i.e., the 'double' value 0.1). Let f = the number 0.1f in C (i.e., the 'float' value 0.1f). And let r = the number 0.1 in mathematics (i.e., the real number equal to 1 divided by 10). Recall that the binary representation of r is the repeating sequence 0.0001100110011001100110011... base 2. Sort the values d, f, and r into non-descending order. If two or more of these three values are equal, say so. Assume x86-64 arithmetic with default rounding. Show your work.

A6

The key observation is that r is infinitely repeating while f and d must terminate. Because of this, we will not be able to represent 1/10 precisely with a single or double precision floating point. As a result, f and d may be rounded up (to be greater than 1/10) or rounded down (to be less than 1/10). In order to see which way it rounds, we need to examine the binary.

r = .0001100110011001100...

The greatest power of 2 in this number is 2^-4. An exponent of that range will required normalized representation in both float and double form. Because the fractional contribution of normalized form includes an implicit 1, we have to rewrite the binary to be of this form:

f/d = 2^--4 * (1 + ...)

f/d = 2^--4 * 1.frac

frac = 10011001100110011001... (or sequence of 1001s repeated)

There are 23 fractional bits in a float. This means that the sequence of repeated 1001s will truncate with the third (23 % 4 = 3) digit of 1001 (or 100|1). There are 52 fractional bits in double. This means that the sequence ends in the last digit of the sequence. Thus (the vertical bar represents the cutoff):

float: frac=10011001...1001100|110011001...

double: frac = 10011001...1001100 110011001...10011001|10011001

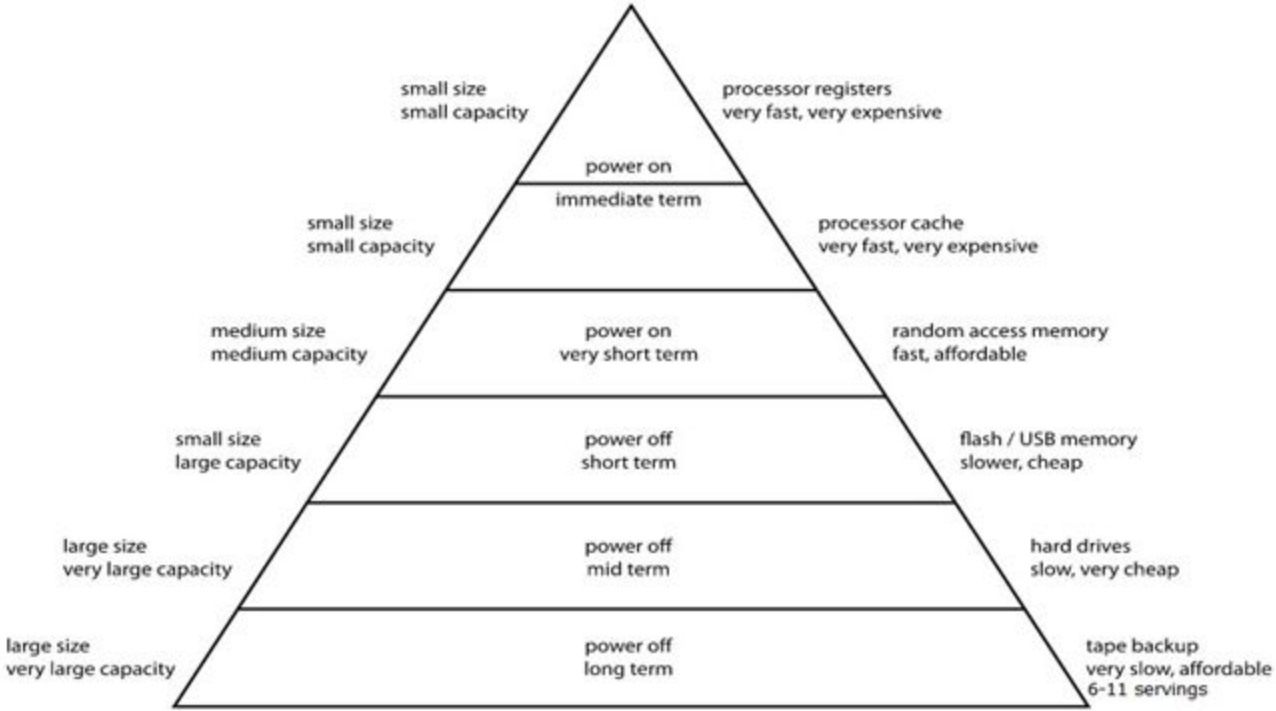
A6

The default rounding mode is "round- to -even", which actually rounds to the closest value and rounds to even only to break a tie (when the actual value is exactly in the middle of two representable numbers). In order to determine if there is a tie, consider the bits to the right of the cutoff. If the bit immediately to the right is 1 and the rest of the bits are 0, then there is a tie. However in this instance, because the bit immediately to the right is 1 AND there are repeated 1s beyond that, both values are actually closer to the rounded -up value. Thus, the actual values will be:

float: frac = 10011001...1001101
double: frac = 10011001...10011001...10011010
real: frac = 10011001...10011001...100110011001...

As a result, both f and d are greater than r. However, f rounds up at a much greater position than the double because the floating point has less precision:
float: 10011001...100110100000…
doub: 10011001...100110011001…
As a result, the ascending order is r,d,f or what is known in the industry as "The Reverse Franklin Delano Roosevelt".

Memory Hierarchy



Courtesy of Wikipedia.org

Memory Hierarchy

- The higher up on the pyramid, the faster the access, but the lower the capacity.
- Because memory is relatively slow, we'd like to spend as much time as possible dealing with the upper levels of the pyramid.
- The very top layer is are the registers.

Memory Hierarchy

- With the RISC-like micro-operations, we have the rule that in order to do anything with data, we must first operate on it in a register.
- The rationale is that if we have a working variable, we'd like to operate on the highest level of the hierarchy as possible to speed things up.
- In a way, this means that registers are like quick temporary copies of the values in memory.

Memory Hierarchy

- When dealing with a value (for example, consider the accumulator), you do all computations on registers.
- Then you commit to memory. That way, you can avoid going to memory altogether.
- However, being at the very tip of the hierarchy means you only have a limited amount of registers.
- What if you wanted to deal with an array?

Memory Hierarchy

- As a result, we introduce a layer of memory that is hidden from assembly/machine instructions.
- This layer operates on the same principle:
  - When you need to operate on data, copy it from DRAM into a faster (but smaller) memory.
  - Once you need it to be committed to actual main memory, copy it back.
- Unlike registers however, it does so implicitly.
- This layer is the cache.

Caching

- Extremely broad overview
- Consider:
  - movl (%ebx), %eax
  - %ebx = 0x10

Caching

- 1.Issue the address to the cache. Does the cache contain the data at address 0x10?
- 2.If the cache has it, then give the data back to the CPU. Else fetch it from RAM

- Implicitly, EVERY memory access goes through the cache.

Caching

- Read Hit:
  - 1.Issue request to cache. The data we're looking for is in the cache.
  - 2. Respond with the data to the CPU

Caching

- Read Miss:
  - 1.Issue request to cache. The data we're looking for is NOT in the cache.
  - 2.Issue request to RAM. As far as you know, the data MUST be in RAM.
  - 3.Respond with data first to cache. Store the data in the cache.
  - 4. Respond with data back to CPU.

Caching

- When we have a cache miss, why do we go through the trouble of pulling the data into the cache?
- Well, we have to have some way of filling the cache.
- Either we populate the cache based on cache misses or we try to pre-populate it based on other information from compilation or running time, which is extremely tricky.

Caching

- First of all, how do we decide what to pull into the cache?
- Let's say, we have movl (%ebx), %eax.
- We need 1 word from memory.
- So we pull in one word from memory into the cache?

Caching

- Caching makes use of a key principle called locality:
  - Temporal Locality: Data that you access is very likely to be accessed again in the near future.
  - Spatial Locality: If you access a piece of data, it's likely that you'll access data that is in a nearby address in the near future.

Caching

- Pulling in a single word would make use of temporal locality, but if we're going to memory, it doesn't make sense to just pull in a value especially since pulling two words from memory is not twice as slow as pulling one word from memory.
- Consider the case where you're iterating through a size 100 array of ints. If we only pull in one word at a time, the cache doesn't help us at all unless we iterate through the array repeatedly.
- We will make 100 requests and we will miss every single time.

Caching

- However, if we pulled in the entire array into the cache when we accessed the first variable, the entire array is set up for us in the
- cache.
- We will have 1 miss and 99 hits.
- As another example, let's say we access a variable on the stack.

  Chances are, we're in a function and it would make sense to pull in that entire function's stack frame into the cache at once.

Caching

- Also, I said that the cache is much smaller than main memory. That's how it can physically afford to be fast.
- In that case, that means we can't fit everything in the cache at once, just as with registers.
- With registers, the compiler can figure it out, but the compiler isn't able to figure out the caching rules.
- When the time comes, we'll have to have some policy to prioritize what to throw out of the cache.

Caching

- How much data is brought into the cache at any given access is a function of the cache itself and the defined “cache block” size, which is the granularity in which cache operations are dealt with.

Caching

- Write Hit:
- When you want to write and you find the data is in the cache, you have a choice to make.
- Do you write just to the cache?
  - This is known as the write-back policy.
- Or do I write to the cache AND also write to memory.
  - This is known as the write-through policy

Caching

- Write Hit (Write-through)
- 1.Issue the write to the cache. The data is in the cache. Write to cache
- 2.Issue the write to memory. Write to memory as well

Caching

- Write Hit (Write-back)
- 1. Issue the write to the cache. The data is in the cache. Write to cache.
- Now you have to remember that the cache is inconsistent with memory. Mark the data in the cache with a “dirty bit”.

Caching

- Write Hit (Write-back)
- When that data must be thrown out of the cache to satisfy another request, write the “dirty” data back into memory to make it consistent.

Write-Hit Policy (Write Policy)

- Benefits of Write-Through?

Write-Hit Policy (Write Policy)

- Benefits of Write-Through?
  - Memory and cache are always synchronized and consistent.
- Downsides?

Write-Hit Policy (Write Policy)

- Benefits of Write-Through?
  - Memory and cache are always synchronized and consistent.
- Downsides?
  - Have to incur that crazy memory penalty every single time you write.
  - Or maybe not? A write buffer can be used in conjunction with a write-through cache. Write to the write buffer after which the processor is allowed to continue while the buffer writes to main memory.

Write-Hit Policy (Write Policy)

- Benefits of Write-Back?
  - Write-through requires 1 write to memory for every store instruction. Write-back effectively collects all of the writes to a particular block and needs to perform one write to memory. This is much better compared to write through if you're frequently writing to the same block.
- Downsides?

Write-Hit Policy (Write Policy)

- Downsides?
  - Write-back is going to be a bit more complicated to manage.

- Realistically however, overall performance is going to be much better with a write-back cache.

Write-Miss Policy (Write Policy)

Write Miss

When you look for data to write in the cache and you find that it's not there, you again have a choice to make.

- Do you first pull the data into the cache, then write?
  This is known as **write-allocate**
- Do you just write the data into memory
  This is known as **no-write-allocate**.

Write-Miss Policy (Write Policy)

- Write miss (no-write-allocate)
- 1. Issue the write to the cache. The data is NOT there.
- 2. Issue the write to RAM. Write to RAM

Write-Miss Policy (Write Policy)

- Write miss (write allocate)
- 1. Issue the write to the cache. The data is NOT there.
- 2. Request the data from RAM (don't write just yet).
- 3. Pull the data from RAM back to cache.
- 4. CPU re-issue write request. Now the write is a hit. Allow write-hit policy to take over

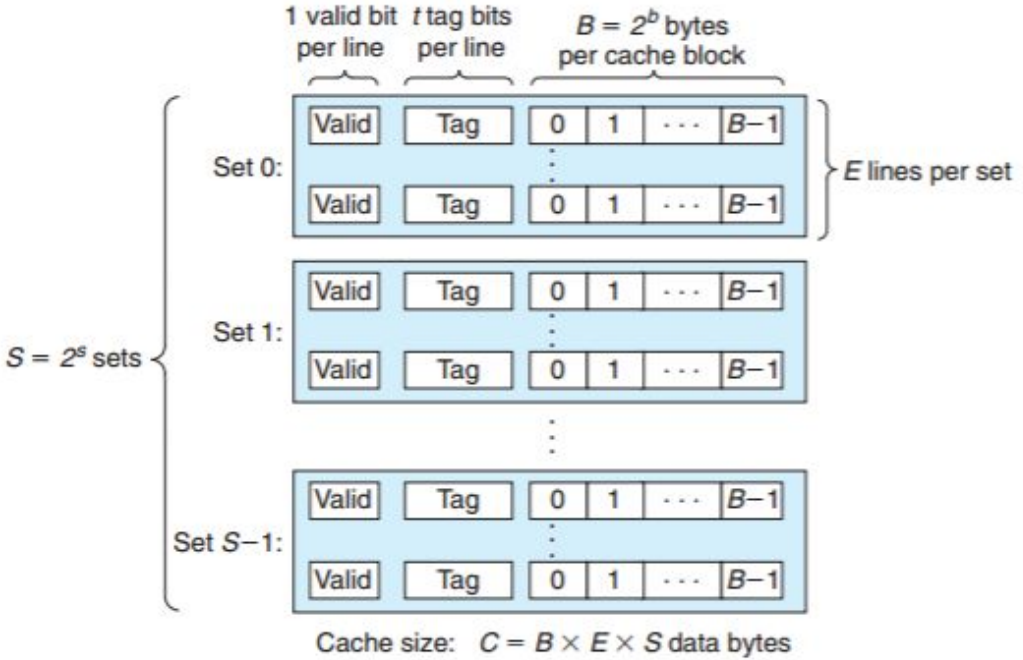
Cache

- Definitions
- Cache blocks:
  - A chunk of bytes in memory that are adjacent.
  - Ex: If block size of 2^5 bytes, then each block is 32 bytes long.
  - Block 0: MEM[0] to MEM[31]
  - Block 1: MEM[32] to MEM[63]
  - Etc...
  - Note: MEM[15] belongs to block 0.

Cache

- Definitions
- Each cache consists of an “array” of “sets”.
- Each set consists of one or more cache blocks.
- Each block has a corresponding tag, valid, and dirty (sometimes) bit.
- A tag is like an id that can be used to identify a cache block.
- A valid bit indicates whether the value in the cache is a valid block that corresponds to data.

Figure 6.27
General organization of cache (S, E, B, m). (a) A cache is an array of sets. Each set contains one or more lines. Each line contains a valid bit, some tag bits, and a block of data. (b) The cache organization induces a partition of the m address bits into t tag bits, s set index bits, and b block offset bits.



Q7

We have a special kind of SRAM cache called Cache Z. Cache Z uses a write-back approach, but omits the dirty bit. Instead, whenever it needs to know whether a cache line is dirty, it loads the corresponding data from RAM and compares it to the data in the cache line: if they compare equal, Cache Z acts as if the dirty bit were zero, and if they compare non equal, Cache Z acts as if the dirty bit were nonzero. Compare the pros and cons of Cache Z to an ordinary write-back write-allocate cache. Propose an improvement to Cache Z's performance that does not involve adding a dirty bit.

A7

According to Cache Z's policy, each time a block must be evicted, you need to go to memory to read the corresponding block. Then, you compare the value of the block in the cache to the value that was fetched from memory. If the values are different, you write the updated block to memory. This means that for each eviction, you MUST read from memory and you might have to write back to memory.

Pros of Cache Z:
-Unlike in a traditional write -back cache, you **do not need to store the dirty bit.** You save a single bit per block

Cons of Cache Z:
-Each eviction costs at least a memory access and **requires 2 memory accesses** if the block is dirty. A traditional write back cache will only require 1 memory access if the block is dirty.

Improvement:
-When evicting, simply write the block in cache back to memory without doing an additional check. Consider a case where there are n evictions and n/2 of those evictions need to be written back to memory. In Cache Z, this will require 1.5*n memory accesses. With this improvement, n memory accesses are necessary, which means that it will always perform at least as well as Cache Z which will require between n and 2n memory accesses.

Q8

Suppose you have an x86-64 machine with the following characteristics:

2 sockets
12 CPU cores per socket
L1 instruction cache: 32 KiB per core
L1 data cache: 32 KiB per core
L2 cache: 256 KiB per core
L3 cache: 30 MiB per socket
64 GiB DRAM per socket

L1 and L2 caches are private to each core. L3 cache is shared by all cores in a socket. All caches are writeback.

8a. The single instruction cache at L1 is fast but very small: why won't performance suffer greatly if your program's kernel doesn't fit into 32 KiB?

8b. Consider the following program, and assume its x86-64 code fits entirely within the L1 instruction cache, and assume that the source and destination do not overlap.

```
#define N <<you pick the constant>>
void transpose(int dst[N][N], int src[N][N]) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            dst[j][i] = src[i][j];
}
```

Suppose this function is often executed in your multithreaded application on the specified machine. What values of N do you recommend for good performance, and why? Look for local sweet spots for N. State any further assumptions you're making.

A8

The architecture includes a unified L2 and L3 cache. As a result, if the working set of instructions doesn’t fit into L1, we can always use the L2/L3 caches. Since the L2 and L3 caches are still quite fast, performance won't be greatly impacted.

The best N will be one that will allow both the src and dst arrays to fit entirely in one of the caches (ie L1, L2, or L3). In general, we want to find an N that satisfies the equation: 4 bytes/int * N^2 ints/array * 2 arrays <= sizeof(cache)
For example, if we want to fit src and dst in the L1 cache: 4 * N^2 * 2 = 32 KiB
2^3 * N^2 = 2^15
N^2 = 2^12
N = 2^6 = 64
So, any N<64 will result in the src and dst arrays fitting entirely in the L1 cache.

Q9

Section 5.9.2 of the book shows how to apply the associative law to improve performance while unrolling a loop. Can we use a similar idea to improve performance by applying the distributive law A*(B + C) == A*B + A*C? Why or why not?

```
1   /* Change associativity of combining operation */
2   void combine7(vec_ptr v, data_t *dest)
3   {
4       long int i;
5       long int length = vec_length(v);
6       long int limit = length-1;
7       data_t *data = get_vec_start(v);
8       data_t acc = IDENT;
9
10      /* Combine 2 elements at a time */
11      for (i = 0; i < limit; i+=2) {
12          acc = acc OP (data[i] OP data[i+1]);
13      }
14
15      /* Finish any remaining elements */
16      for (; i < length; i++) {
17          acc = acc OP data[i];
18      }
19      *dest = acc;
20  }
```

Figure 5.26 **Unrolling loop by 2 and then reassociating the combining operation.** This approach also increases the number of operations that can be performed in parallel.

A9

There are two cases to consider. The first case is when A is the accumulator.
(1) a=a*(b[i]+c[i])
(2) a=a*b[i]+a*c[i]

For (1), the bottleneck would be the multiplication operation, since we can perform the addition of iteration i + 1 before waiting for the multiplication of iteration i to finish. Thus, the critical path would consist only of the multiplication.

For (2), the bottleneck involves the multiplication operations (both of which can be done in parallel) and the addition operation. In this case, the two multiplications and the addition must be done sequentially; we cannot do the multiplication of iteration i + 1 while doing the addition of iteration i because the multiplication of iteration i + 1 relies on the updated value of “a” from the addition of iteration i. Thus, the critical path would consist of a multiplication operation and an addition operation.

A9

Since (2) is not as efficient as (1), the distributive law doesn't improve performance.

However, if you assume that the accumulator is b, we would have:

(1) b=a[i]*(b+c[i])
(2) b=a[i]*b+a[i]*c[i]

Since the bottleneck is the same for both cases, applying the distributive law doesn't matter. In (1), the critical path involves an addition and then a multiplication. The addition of iteration i + 1 cannot be done in parallel with the multiplication of iteration i because the addition uses "b" which relies on the result produced by the multiplication of the previous iteration. This is also true of (2).

Smashing Lab

This assignment investigates an old-fashioned way of breaking into systems executing x86 machine code, along with a couple of machine-level defenses against this attack. It's chosen not to give you a toolkit to break into other sites – the method is well-known and ought to be commonly defended against nowadays – but instead, to give a general idea of how an attacker can break into a system by exploiting behavior that is undefined at the C level but defined at the machine level, and what we can do about it at the machine level.

ADDITIONAL USEFUL LINK : http://acme.com/software/thttpd/thttpd_man.html

Smashing Lab

Modifying sthttpd-2.27.0

```
--- sthttpd-2.27.0/src/thttpd.c    2014-10-02 15:02:36.000000000 -0700
+++ sthttpd-2.27.0-delta/src/thttpd.c   2015-04-30 19:15:24.820042000 -0700
@@ -999,7 +999,7 @@ static void
 read_config( char* filename )
     {
     FILE* fp;
-    char line[10000];
+    char line[100];
     char* cp;
     char* cp2;
     char* name;
@@ -1012,7 +1012,7 @@ read_config( char* filename )
     exit( 1 );
     }
-    while ( fgets( line, sizeof(line), fp ) != (char*) 0 )
+    while ( fgets( line, 1000, fp ) != (char*) 0 )
     {
     /* Trim comments. */
     if ( ( cp = strchr( line, '#' ) ) != (char*) 0 )
```

Smashing Lab

First,

```
make clean
```

1. (SP) for strong stack protection:

```
make CFLAGS = '-g3 -O2 -fno-inline -fstack-protector-strong'
mv src/thttpd src/thttpd-sp
```

2. (AS) for address sanitization:

```
make CFLAGS = '-g3 -O2 -fno-inline -fsanitize=address'
mv src/thttpd src/thttpd-as
```

3. (NO) for neither:

```
make CFLAGS = '-g3 -O2 -fno-inline -fno-stack-protector -zexecstack'
mv src/thttpd src/thttpd-no
```

Smashing Lab

Now, we have 3 executables: `thttpd-sp, thttpd-as, thttpd-no`

Task 1 - Find a way to **crash** `thttpd-**` and **log** the backtrace.

Task 2 - Generate the assembly language code for thttpd.c and run it

Task 3 - Compare the implementations of `handle_read` in the 3 variants.

Task 4 - Describe how `-fstack-protector-strong` and `-fsanitize=address` prevent buffer-overrun exploits in `handle_read`

Task 5 - Build an exploit for the bug in variant NO that relies on the attacker tricking the victim into invoking thttpd with a particular value for the -C option, that causes the victim web server that is invoked via GDB with -C to remove the file target.txt in the working directory of the web server. Or, if such an exploit is impossible, explain why not, and investigate simple ways to alter the compiler flags (or, in the worst case, to insert plausible bugs into the source code) to make the exploit possible.

Instruction Level Parallelism

- The optimizations presented are all serial optimizations that can marginally improve execution time.
- Marginal improvement can still be significant depending on how much time is spent on the section that was improved.

Instruction Level Parallelism

- However, anything a traditional program does is limited by the fact that a single program or a single thread is expected to the entire work of a program.
- If we have to iterate over a length 100 array and assign a value to each element, there's not a whole lot a single thread can to do get around this.
- But what if we have two processors dedicated to the same task?

Instruction Level Parallelism

- If instead, we have two processors each working on half of the data, we can finish the array assignment in half of the time, which provides a speedup of 2, which was already faster than the best loop unrolling case.
- Plus, the performance improvement provided by splitting the task among independent agents doesn't degrade nearly as quickly. If we had 100 processors, we could complete the task in the time it takes a processor to execute a single instruction.

Instruction Level Parallelism

- It's this observation that has driven the industry towards parallelism rather than simply increasing clock speed when it comes to improving performance.
- Why can't we just keep increasing the clock speed?

Instruction Level Parallelism

- Power Wall:
  - As the clock rate increases, the power needed to operate everything increases. In addition to requiring a lot of power, much of it is leaked/dissipated as heat.
- Memory Wall:
  - The processor speeds have increased significantly since the 80's.
  - Also recall the RAM access time has remained essentially the same.

Instruction Level Parallelism

- Memory Wall:
  - The bottleneck is still in RAM access and it is the thing we'd need to reduce in order to get a significant improvement.

Instruction Level Parallelism

- In order to execute a single simple micro operation, it goes through several steps or “stages” to complete. In the common academic example, there are 5 stages.
- This is known as the pipeline.

Instruction Level Parallelism

- Each stage corresponds to a physical component in the processor.
- IF (Instruction Fetch): Takes as input the address of the instruction (%rip). Then it finds the actual instruction in memory.
- ID (Instruction Decode/Register Read): Given the instruction bytes, get the appropriate values from memory.

Instruction Level Parallelism

- EX (Execution): Execute any arithmetic operation that the instruction needs to do.
- MEM (Memory): Read/write from memory if necessary.
- WB (Write back): Write to register if necessary.

Instruction Level Parallelism

- Assume that each block takes a single cycle to execute. A single instruction would take 5 cycles to complete.
- The latency of each instruction is 5 cycles.
- Our throughput of this system is 1 instruction per every 5 cycles.

Instruction Level Parallelism

- Consider the execution of: add %eax, %ebx
- Step 1. The instruction address is used to fetch the instruction
- Step 2. The values from %eax and %ebx are loaded
- Step 3. %eax is added to %ebx.
- Step 4. The add instruction doesn't use
- memory Step 5. Save the value back into %ebx

Instruction Level Parallelism

- At any given point in time, we are only using one of five available components.
- This is an opportunity to execute multiple instructions at the same time.

Instruction Level Parallelism

- Say we have the following instruction sequence:
  - 1. add %eax, %ebx
  - 2. sub %ecx, %edx
  - 3. xor %esi, %edi
- Say the execution of this snippet begins at time 0.
- At time 0: the add would be in the IF stage.
- At time 1: the add would be in the ID stage and the sub would be in the IF stage.
- At time 2: the add would be in the EX stage…

Instruction Level Parallelism

- Based on this, how long does each instruction take to complete?
  - It still takes five cycles for any given instruction to complete. The latency has not changed.
- However, how many instructions are completed per cycle?
  - 1 instruction is completed per 1 cycle rather than 5 cycles. The throughput has increased from 1/5 instructions per cycle to 1 instruction per cycle. THAT is a huge gain.

Instruction Level Parallelism

- This example of ILP can simultaneously execute 5 instructions at any given time.
- …or is this perhaps a little too idealized?
- What are some cases where we wouldn't be able to achieve this optimal case?

Instruction Level Parallelism

- Data Hazard: Without pipelining, when each instruction begins, it can safely assume that the previous instruction has completed. With pipelining, that assumption is no longer true.
- Consider:
- add %eax, %ebx
- add %ebx, %ecx
- The first add will not have saved the new value to %ebx by the time the second add needs to read from it.

Instruction Level Parallelism

- Control Hazard:
- When a conditional operation enters the pipeline, it's expected that another operation will enter the pipeline in the next cycle.
- But what instruction should that be if the first operation was conditional?
- We guess using branch prediction, but if we guess wrong, we wasted time doing unnecessary work.

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

References

- Slides modified from DJ Kim, UT Wang and Shikhar Malhotra
- http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html

Thank You