CS420 – Lecture 2

Marc Snir

Fall 2018

Machine code may have different dependences

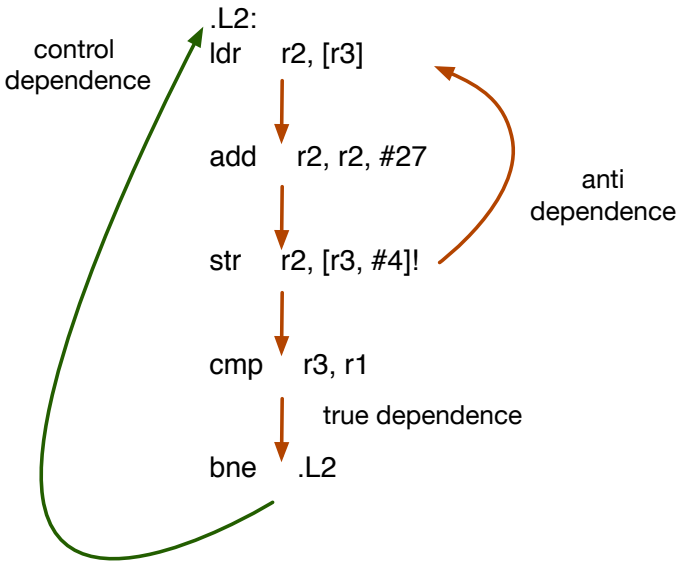
```
#include <stdlib.h>
#define N 25
#define s 27
int main() {
int i;
int a[N];
a[0]=1;
for(i=1; i<N; i++)
a[i]=s+a[i-1];
exit(a[N-1]);
}
```

ARM assembly

```
main:

str     lr, [sp, #-4]!
sub     sp, sp, #108
add     r3, sp, #104
mov     r2, #1
str     r2, [r3, #-100]!
add     r1, sp, #100
.L2:
ldr     r2, [r3]
add     r2, r2, #27
str     r2, [r3, #4]!
cmp     r3, r1
bne     .L2
ldr     r0, [sp, #100]
bl      exit
```

```
.L2:                        //  instruction label
ldr     r2, [r3]            // load register r2 from memory location
                            // at the address in r3
add     r2, r2, #27         // r2 = r2 + 27
str     r2, [r3, #4]!       // increment r3 by 4 and store value of r2
                            // at the resulting address
cmp     r3, r1              // compare r3 and r1; the result goes into
                            // a test bit register
bne     .L2                 // if test bit shows last comparison result
                            // was "not-equal" go to instruction L2;
                            // otherwise, continue with next instruction
```



Main performance bottleneck is dependence on load (load can take 100's of instruction cycles)

Avoid memory accesses

source code

```
temp = a[0];
for (i=1; i<N; i++) {
  temp = s+temp;
  a[i]=temp
}
```

Running time improved by x4. Do not need to wait for store to complete before starting next operation.

Compiler does this transformation on its own (with -O3 optimization)

```
main:
mov     r2, #28
str     lr, [sp, #-4]!
sub     sp, sp, #108
add     r3, sp, #4
add     r1, sp, #100
.L2:
str     r2, [r3, #4]!
cmp     r3, r1
add     r2, r2, #27
bne     .L2
ldr     r0, [sp, #100]
bl      exit
```

Register renaming

```
1. ldr  r1,[#1024]
2. add  r1, r1, #2
3. str  r1, [#148]
4. ldr  r1, [#2090]
5. add  r1, r1, #4
6. str  r1 [#3000]
```

have false (WAR) dependence on register r1
Compiler can solve problem by using another register

```
1. ldr  r1,[#1024]
2. add  r1, r1, #2
3. str  r1, [#148]
4. ldr  r2, [#2090]
5. add  r2, r1, #4
6. str  r2 [[#3000]
```

Problem: Number of registers small (e.g., 16) – compiler runs out of registers for more complicated code.
Solution: Hardware uses "hidden" registers during execution

```
1. ldr  r1,[#1024]
2. add  r1, r1, #2
3. str  r1, [#148]
\\ hardware allocates a new copy
\\ of r1; new copy used subsequent
4. ldr  r1, [#2090]
5. add  r1, r1, #4
6. str  r1 [[#3000]
```

Loop fusion

```
for(i=0;i<N;i++)
    a[i]=a[i]+b[i];
for(i=0;i<N;i++)
    b[i]=b[i]*s;
```

We *fuse* the two loops

```
for(i=0;i<N;i++) {
    a[i]=a[i]+b[i];
    b[i]=b[i]*s;
}
```

Loop fusion

- reduces number of branches;
- usually reduce number of loads (`b[i]` will be loaded only once);
- enable further optimizations

Optimizations

- *Loop unrolling* can be done manually; a good optimizing compiler will do it automatically
- *Register renaming* is done by hardware – no need to do it
- Loop fusion can be done manually; good optimizing compiler will do it automatically
- *Branch prediction* is done by hardware
- *Executing loads earlier* – compiler optimization
- Hardware can also provide *out-of-order* execution: If instruction will surely execute (no branch) and is ready to execute (no dependence) that is can be executed out of order
- Hardware can provide *speculative execution* (e.g., with branch prediction). If speculation turns out wrong, it is undone.

It's the memory, stupid

seq: Program assigns random values to consecutive entries in an array of length 500,000,000. 10 measurements are made

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#define N 500000000
#define s 1103515245
#define t 12345
#define rmax 2147483648
long int a[N];
int main() {
 long  int i,j,k;
 long int m=1;
 time_t time;
 for(i=0;i<N;i++)
  a[i]=i;
 for (k=0;k<10;k++) {
  time = clock();
  for(i=0;i<N;i++) {
   m = (m*s+t)%rmax;
   a[i]=m%N;
  }
  time = clock()-time;
  printf("%lu ",
   1000*time/CLOCKS_PER_SEC);
 }
 printf("/n");
 exit((int)a[N-1]);
}
```

Program variants

Original program (seq)

```
for(i=0;i<N;i++) {
m = (m*s+t)%rmax;
a[i]=m%N;
}
```

Assigns values to random entries in an array of length 500,000,000 *first variant (rand)*]

```
for(i=0;i<N;i++) {
m = (m*s+t)%rmax;
a[m%N]=i;
}
```

second variant (short)

```
#define M 1000
#define N 500000
...
  for (r=0; r<M; r++)
for(i=0;i<N;i++) {
m = (m*s+t)%rmax;
a[m%N]= i;
}
```

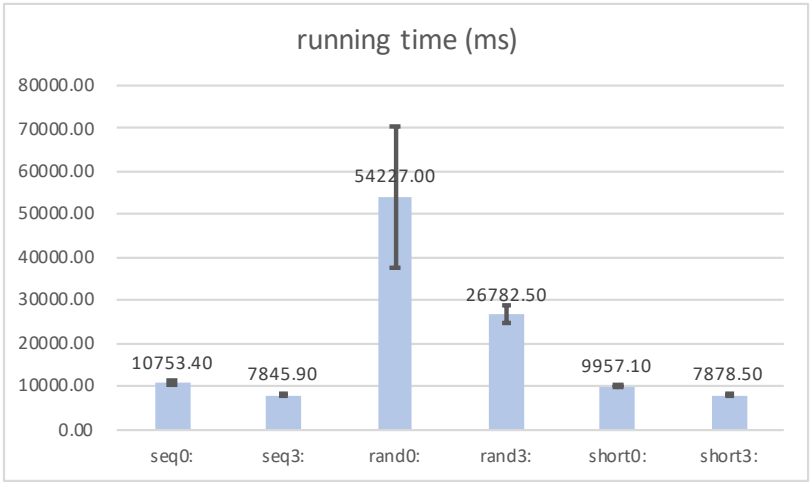
Same number of assignments as the second program but using a shorter array of length 500,000

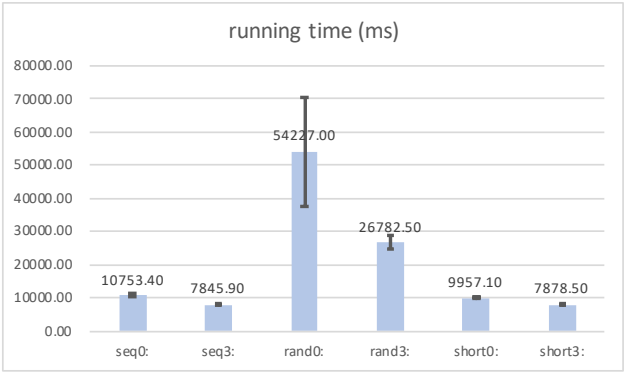
Results from one experiment

- First program compiled with no optimizations 10 runs
- 11353 10452 11317 9946 11538 10623 9772 9813 10609 12111
- Repeated execution always provide same answer but *execution time can vary a lot!*
- Standard deviation: 819
- Confidence interval: 507

Results

0: program compiled with no optimization; 3: program compiled with optimization level 3 (-O3)





- -O3 helps: running time reduced by 19%, 51% and 21%
- Accessessing consecutive locations in memory helps: 3.4x faster
- Having smaller data set helps:3.4x faster
- *Memory access pattern has a major impact on performance!*

Registers and Caches

- ALU can execute (at least) 2–3 operations per nsec (2–3 GHz = 2–3 Gops/s)
- It takes ∼100 nsec to load word from memory

  Problem: If operands are always loaded from memory and stored back, then CPU will run at memory speed (∼200 times slower)

Registers and Caches

- ALU can execute (at least) 2–3 operations per nsec (2–3 GHz = 2–3 Gops/s)
- It takes ∼100 nsec to load word from memory

Problem: If operands are always loaded from memory and stored back, then CPU will run at memory speed (∼200 times slower)

Solutions:
1. Registers – can be accessed at CPU speed (by there are only a few tens of them)

Registers and Caches

- ALU can execute (at least) 2–3 operations per nsec (2–3 GHz = 2–3 Gops/s)
- It takes ∼100 nsec to load word from memory

Problem: If operands are always loaded from memory and stored back, then CPU will run at memory speed (∼200 times slower)

Solutions:
1. Registers – can be accessed at CPU speed (by there are only a few tens of them)
2. Load data earlier than needed for it to have time to arrive into register before it is used

Registers and Caches

- ALU can execute (at least) 2–3 operations per nsec (2–3 GHz = 2–3 Gops/s)
- It takes ∼100 nsec to load word from memory

Problem: If operands are always loaded from memory and stored back, then CPU will run at memory speed (∼200 times slower)

Solutions:
1. Registers – can be accessed at CPU speed (by there are only a few tens of them)
2. Load data earlier than needed for it to have time to arrive into register before it is used
3. Caches

Registers and Caches

- ALU can execute (at least) 2–3 operations per nsec (2–3 GHz = 2–3 Gops/s)
- It takes ∼100 nsec to load word from memory

Problem: If operands are always loaded from memory and stored back, then CPU will run at memory speed (∼200 times slower)

Solutions:

1. Registers – can be accessed at CPU speed (by there are only a few tens of them)
2. Load data earlier than needed for it to have time to arrive into register before it is used
3. Caches
4. Cache prefetch – CPU guesses what will be loaded in the near future and bring it into the cache ahead of time

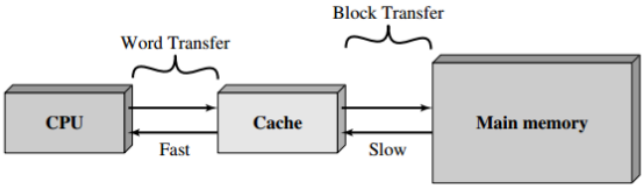
Cache

- Can build large DRAM memory (off chip), but access time is high and bandwdith is low.
- Can build fast SRAM memory (on chip), but capacity is small and power consumption is high
- **Idea:** Use both to provide what looks like memory with capacity of DRAM and speed of SRAM
  - Keep frequently accessed data in cache

DIMM

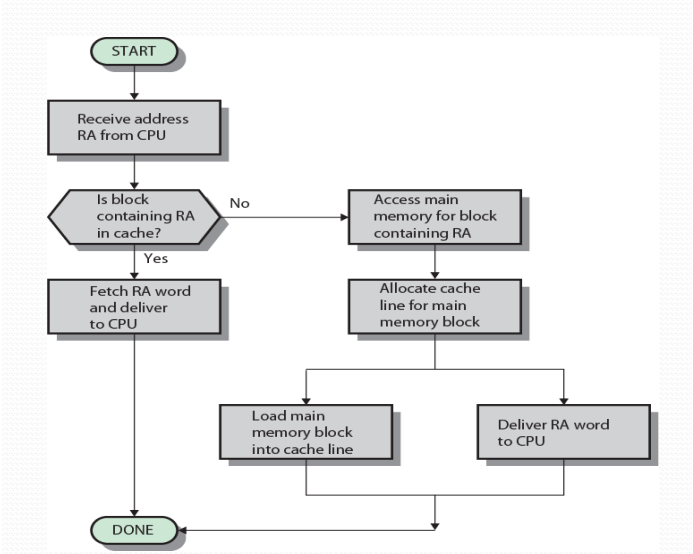
Computer Board

Cache



- Main memory is slow. In order to get higher bandwidth, data is moved from memory to cache in large blocks (typically 64 bytes) – *cache lines*.
- Data is moved from cache to registers in single words (8 bytes).
- A load first looks for the loaded word in cache; if it is not there, it is brought from memory to cache

Flow chart for a LOAD



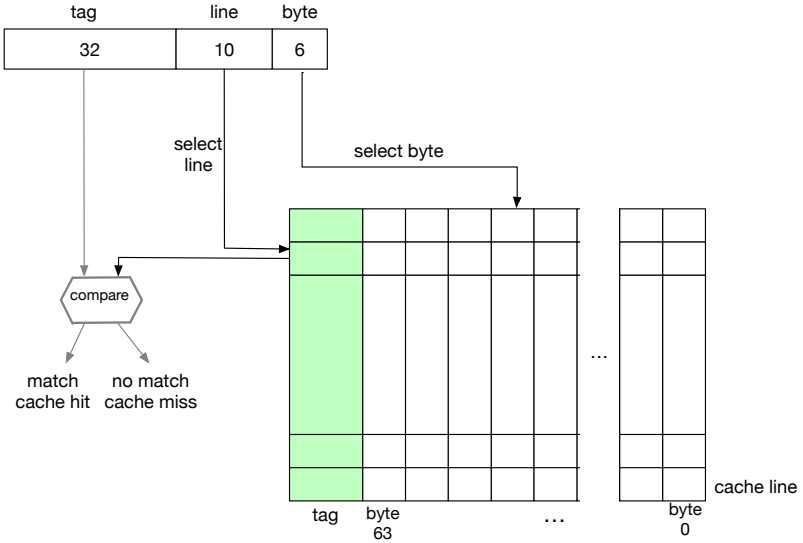
- How do we search the cache?
- What do we do if the cache is full?

Cache organization

Typical numbers

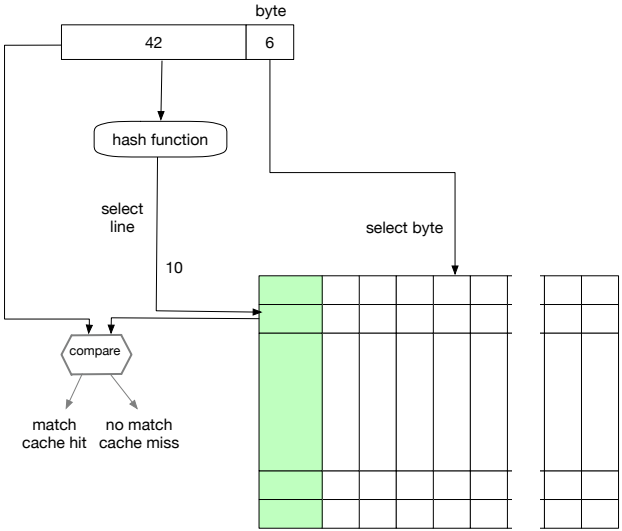
- Cache line = 64 bytes = 8 words (size of block transfered from memory)
- Cache size = 64 KB = 1024 cache lines
- Physical memory address is 48 bits – memory is byte addressable.
  - Typically, physical address is shorter than virtual address (i.e., value of pointer – 64 bits). We will not discuss virtual to physical address. translation.

Cache direct mapping



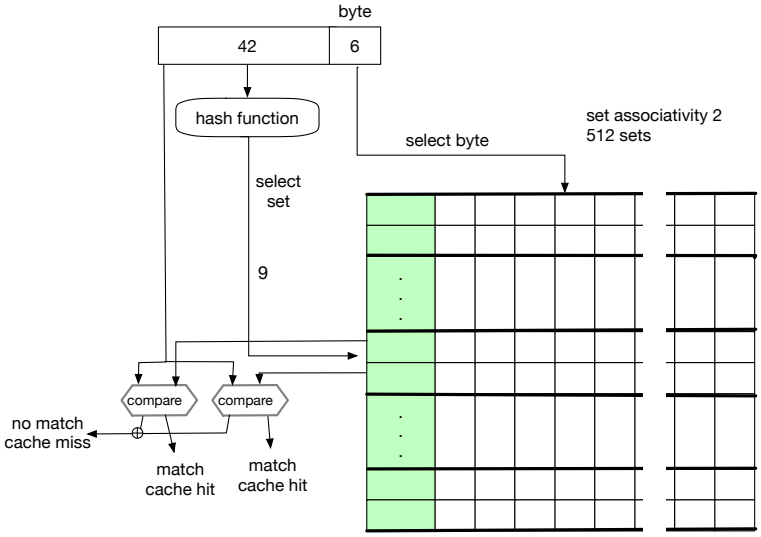
- Logic is simple
- A byte can be in only one location in cache; location is determined by last 16 bits of its address. Two lines that are $k \times 2^16$ apart in memory *conflict* in cache.
- When a line is loaded, previous line in that location is evicted

Direct mapping improved



- Logic is slightly more complicated
- A memory line can be in only one location in cache; location is determined by all 42 bits of its address
- Regular access patterns (e.g., strided) are not a problem
- But can still have conflict misses

Set-associative cache

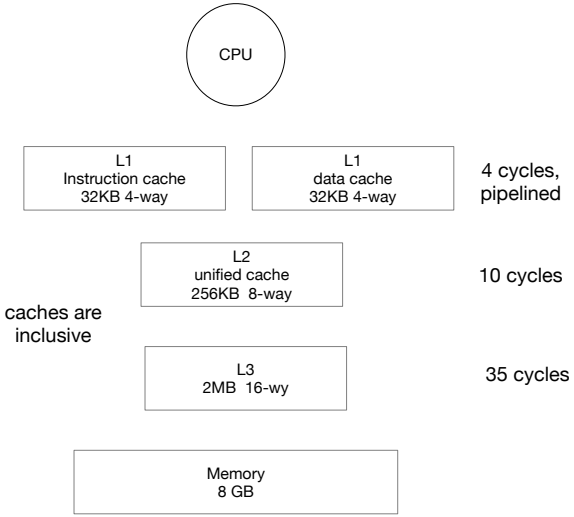


- Logic is more complicated
- A line can be in only one set, but in any line in this set; set is determined by all 42 bits of its address
- Conflict misses are less likely
- Have a choice which line to evict. Usually evict "older" line.

Ideal cache: Fully associative cache

- The entire cache is one associativity set: A cache line can be stored anywhere in the cache.
- Least Recently Used (LRU): Always evict the line that was not accessed for the longest time.
- Not practical: Would need to compare all tags
- Practical caches are an approximation of a fully associative, LRU cache
- One can analyze code behavior assuming "ideal" caches

Can have multiple cache levels



Typical numbers

A more complex processor