Massively Parallel Computer Architecture
High-Performance Microprocessor Architecture and Programming

Hiroshi Nakashima
(ACCMS, Kyoto University)

Contents

Overview (1/4)

- **Factor to Determine Performance**
  - First-level Approximation = #basic-ops (≈machine insts)×1/frequency

```
for(x=0;x<nx;x++)
  un[y][x]=u[y][x]+
    (dt/(h*h))*(u[y][x+1]+u[y][x-1]+
                u[y+1][x]+u[y-1][x]-
                4*u[y][x]);
```

FP register / GP (int) register

  - f=u[][]...5, un[][]=f ...1
  - f=f+f ...5, f=f*f ...3, f=f/f ...1
  - g++ ...1, g<g ...1, if()goto ...1
  - → 18/frequency?
  - → supercomputer in KU (18core Xeon) ≈ 7/frequency — why so small?

Overview (2/4)

- **Why "#ops/freq" Overestimates = Parallel Execution of Multiple Insts**
  - in KU's supercomputer
    - f=mem × 2
    - mem=f × 1
    - f=f+f × 1
    - {f=f*f, f=f/f} × 2 (except for f=f/f × 2)
    - integer insts ∋ {g++, g<g, if()goto} × 4

  can be executed in parallel — in reality, we have some other restrictions

→max((f=mem)×5/2, (mem=f)×1/1,
(f=f+f)×5/1, ((f=f*f)×3+(f=f/f)×1)/2,
((g++)×1+(g<g)×1+(if())×1)/4)
= 5? (< 7)

Overview (3/4)

- **Why max(...) Underestimates = Latency of Depending Instruction**
  - in KU's supercomputer (unit=cycle)
    - f=mem ... 5 (+7+23+60?) — 1st / 2nd / 3rd-level cache miss
    - f=f+f ... 3, f=f*f ... 5, f=f/f ... 14(!!) — and we have to wait 5-cycle for next divide
    - {g++, g<g} ... 1
    - if()goto ... 1 (+15?+7+...) — branch misprediction / 1st / ... cache miss

  →avoid division as much as possible!!
  - c.f. other representative ops
    - g=mem ... 4 (+...)
    - g=g+g ... 1, g=g*g ... 3, g=g/g ... 20~26

Overview (4/4)

Loop-invariant code motion is a fundamental optimization but not always (able to be) done by compilers.

When un, dt, h are global, a compiler might unsure that dt and h are invariant due to update of un[][].

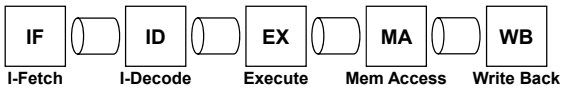
```
dthh=dt/(h*h);
for(t=0;t<tmax;t++){
  for(y=0;y<ny;y++) for(x=0;x<nx;x++)
    un[y][x]=u[y][x]+
      dthh*(u[y][x+1]+u[y][x-1]+
            u[y+1][x]+u[y-1][x]-
            4*u[y][x]);
  tmp=un; un=u; u=tmp;
}
```

lesson-1: Do a simple optimization by yourself.
lesson-2: Don't use global variables in a loop if possible. (e.g., move them into local variables)

- **Performance of Modified Code in KU's Supercomputer ≈ 5~6/frequency**
  - Reasons why >5 are shown later.
  - Many pitfalls to make >> 6
    - → Key issues are to avoid them

Basics of Instruction Pipeline (1/6)

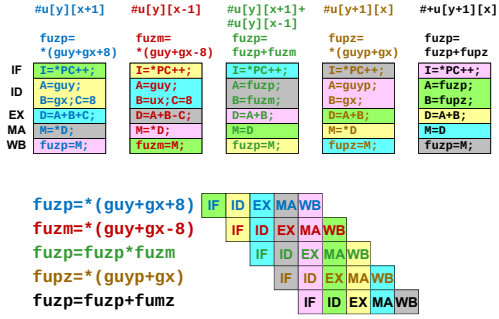
- **Instruction Pipeline**
  - Split instruction execution flow into multiple stages.
  - All stages work in parallel.
- **In Textbooks …**



  - → (up to) 5 (=#stages) instructions are executed in parallel.
- **In Reality …**
  - ≈20 stages (or more) and more complicated (as discussed later)

Basics of Instruction Pipeline (2/6)

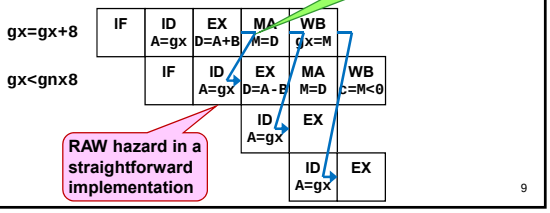
- **Part of DE-solver loop (assume latency=1)**



Basics of Instruction Pipeline (3/6)

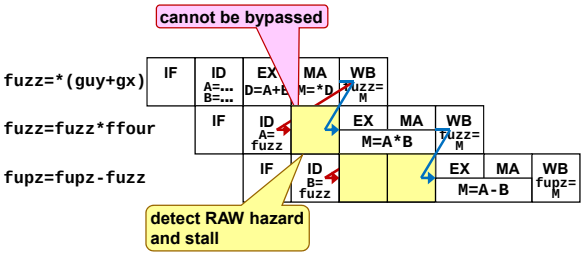
- **Hazard due to Instruction Dependence and Remedy for it**
  - true/flow dependence = output of predecessor is used by successor
  - RAW (read-after-write) hazard = generate/use-timing is reversed.

→bypassing / forwarding



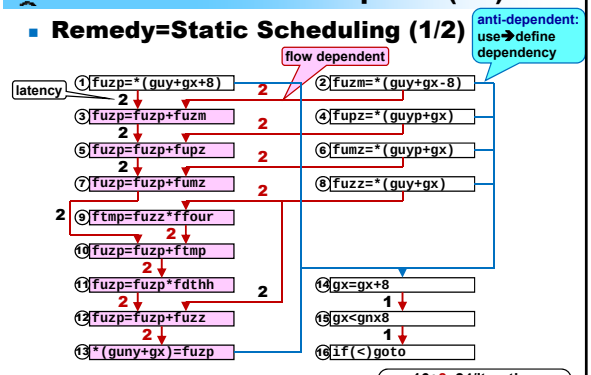
Basics of Instruction Pipeline (4/6)

- **Latency ≥ 2 → not always avoidable → Stall**
  - Even when latency of `f=mem`, `f=f+f`, `f=f*f` is 2 …



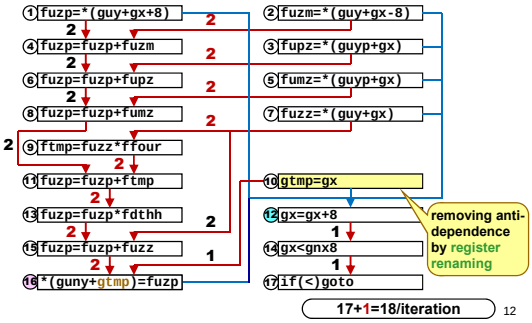
Basics of Instruction Pipeline (5/6)

- **Remedy=Static Scheduling (1/2)**



Basics of Instruction Pipeline (6/6)

- **Remedy=Static Scheduling (2/2)**



ILP (1/13)

- **ILP: Instruction Level Parallelism**
  - Execution order of mutually independent instructions can be modified arbitrarily.
  - ➔They can be executed in parallel
  - ➔Execute multiple instructions in parallel with multiple function units (e.g. ALUs)
  - Variants
    - super scalar: execute ordinary instructions
      - static: in-order execution of compiled code (1990's)
      - dynamic: change order at run time (out-of-order, now)
    - VLIW: one instruction performs multiple operations (e.g. Itanium)
    - SIMD: one instruction operates on multiple data (SSEx: streaming SIMD extension, AVX)

scalar means "not vector (processor)"
very long inst. word
single inst. multiple data

ILP (2/13)

- **KU's Supercomputer**
  - Issue Rate = 7 (#operations which can start at every cycle)
  - #Function Units
    - `{g,f}=mem` × 2, `mem={g,f}` × 1
    - `{f=f*f[+f],f=f+f}` × 1, `{f=f*f[+f],f=f/f}` × 1,
    - `{g++,g<g}` × 2, `{g++,g<g,if()goto}` × 2

    (remark: #FP-ops + #int-ops ≤ 4)

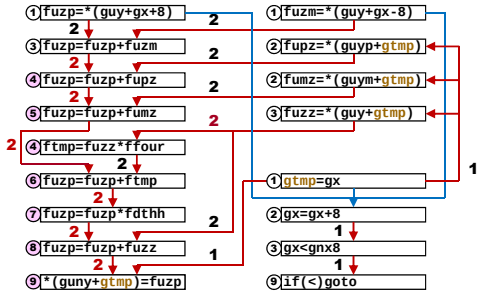
Instructions per Cycle

➔maximum execution throughput (IPC)=7

➔(16inst/7)/iteration = 2.3 cycle/iteration ?

ILP (3/13)

- **Static Scheduling**
  - Latency of `f=mem`, `f=f{+,*}f` = 2



9+6=15/iteration ➔ IPC=17/15=1.13

ILP (3/13)

- **loop unrolling**
  - Expand *k* iterations to have one iteration
  - Improve parallelism in loop body (& reduce loop-control overhead)

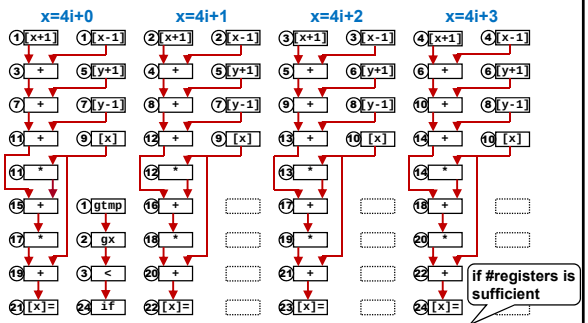
```
for(x=0;x<nx;x++) un[y][x]=u[y][x]+...;
```

4-way unrolling

```
for(x=0;x<(nx/4)*4;x+=4){
  un[y][x+0]=u[y][x+0]+...;
  un[y][x+1]=u[y][x+1]+...;
  un[y][x+2]=u[y][x+2]+...;
  un[y][x+3]=u[y][x+3]+...;
}
for(;x<nx;x++) un[y][x]=u[y][x]+...;
```


ILP (5/13)

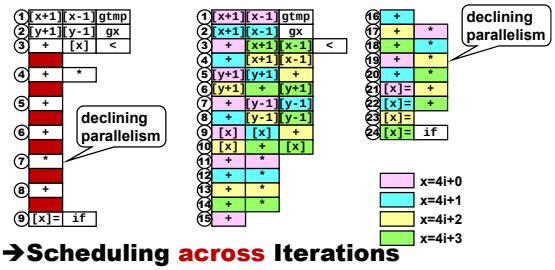
- **Dependence Graph of 4-way Unrolled Loop**



24/(4•iteration) ➔ IPC=56/24=2.33 (68/24=2.83)

ILP (6/13)

- **Limitation of Static Scheduling (incl. Loop Unrolling) = Insufficient Parallelism inside one or *k* Iterations**

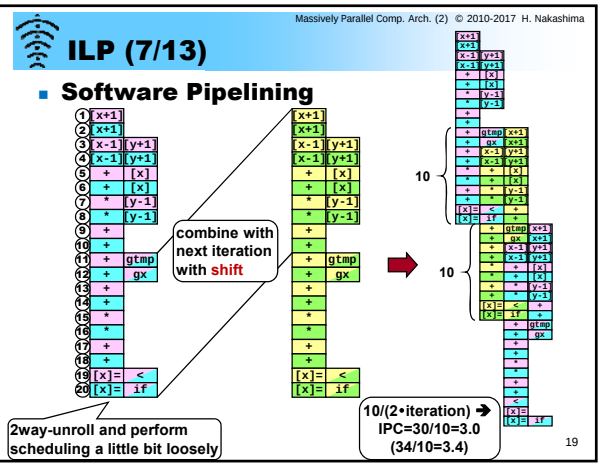


➔**Scheduling across Iterations**

= Software Pipelining

ILP (7/13)

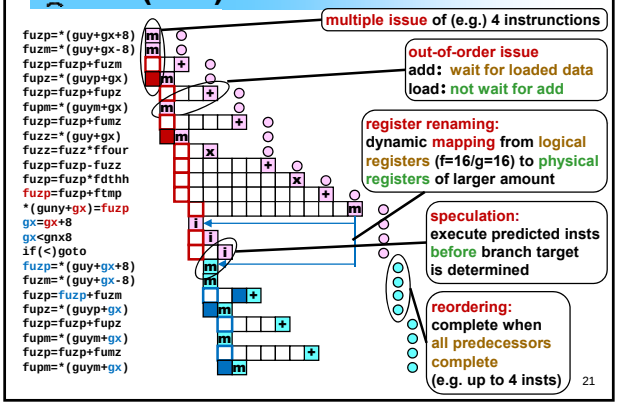
- **Software Pipelining**



ILP (8/13)

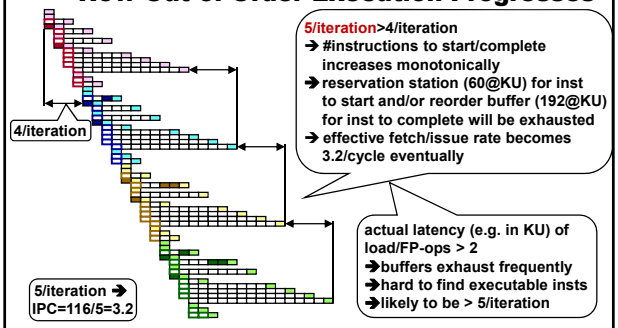
- **Dynamic Scheduling**
  - **= dynamic execution ordering automatically done by microprocessor**
    - **Out-of-order issue** of executable instructions with ready-to-use operands irrespective of their program order
    - Dynamic **register renaming** to remove unnecessary restriction of execution order
    - **Speculative execution** of instructions given by branch predictor
    - **Reordering** for in-order completion to cope with branch misprediction, exception, interrupt, etc.

ILP (8/13)



ILP (9/13)

- **How Out-of-Order Execution Progresses**



ILP (10/13)

- **Exploiting SIMD (AVX) Mechanism**
  - ymm: 256-bit media register
    - `char` × 32, `short` × 16, `int` × 8, `long long` × 4
    - `float` × 8, `double` × 4

    ➔ perform a particular operation on multiple data
  - 4-way unroll and perform 4 ops by one inst

```
for(x=0;x<nx;x+=4) {
 un[y][x]  =u[y][x]  +
  dthh*(u[y][x+1]+u[y][x-1]+u[y+1][x]  +u[y-1][x]  -4*u[y][x]);
 un[y][x+1]=u[y][x+1]+
  dthh*(u[y][x+2]+u[y][x]  +u[y+1][x+1]+u[y-1][x+1]-4*u[y][x+1]);
 un[y][x+2]=u[y][x+2]+
  dthh*(u[y][x+3]+u[y][x+1]+u[y+1][x+2]+u[y-1][x+2]-4*u[y][x+2]);
 un[y][x+3]=u[y][x+3]+
  dthh*(u[y][x+4]+u[y][x+2]+u[y+1][x+3]+u[y-1][x+3]-4*u[y][x+3]);
}
```

➔ but compilers often fail to SIMD-vectorize

ILP (11/13)

- **Why Compilers Fail to SIMD-vectorize**

```
for(x=0;x<nx;x+=4) {
 un[y][x]  =u[y][x]  +dthh*(...);
 ...
 un[y][x+3]=u[y][x+3]+dthh*(...);
}
```

&un[y][x]==&u[y][x+3] ➔mutually dependent

actually they are independent definitely but compilers are often ignorant of it

this transformation does not preserve semantics if dependent

```
for(x=0;x<nx;x+=4) {
 double t0=u[y][x]  +dthh*(...);
 ...
 double t3=u[y][x+3]+dthh*(...);
 un[y][x]  =t0;
 ...
 un[y][x+3]=t3
}
```

this code means they are independent

SIMD-vectorizable if this transformation is correct

ILP (12/13)

- **How to Make Loop SIMD-Vectorizable**

```
for(t=0;t<tmax;t++){
  double (*restrict uu)[NX]=u,
         (*restrict uun)[NX]=un;
  for(y=0;y<ny;y++) for(x=0;x<nx;x++)
    uun[y][x]=uu[y][x]+
      dthh*(uu[y][x+1]+uu[y][x-1]+
            uu[y+1][x]+uu[y-1][x]-4*uu[y][x]);
  tmp=un; un=u; u=tmp;
}
```

- `*restrict` *p*

may run incorrectly if assurance is incorrect

=*p* or *(*p*+exp) is not be updated by a pointer other than *p*
➔programmer assures `&uun[y][x]!=&uu[y'][x']`
➔SIMD-vectorizable ➔ 1.8 cycle/iteration (> lower bound=1.25)

ILP (13/13)

- **Fine Tuning and Caution**

```
uun[y][x]=uu[y][x]+
  dthh*(uu[y][x+1]+uu[y][x-1]+
        uu[y+1][x]+uu[y-1][x]-4*uu[y][x]);
```

This transformation does not preserve program semantics
←FP operations are not associative

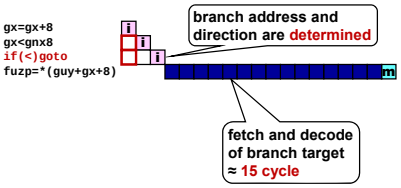
```
uun[y][x]=uu[y][x]+
  dthh*((uu[y][x+1]+uu[y][x-1])+
        (uu[y+1][x]+uu[y-1][x])-4*uu[y][x]);
```

reduce total latency of RHS
➔reduce #-of in-flight instructions and possibility of buffer exhaustion
➔increase chance to find executable instructions

operation result of tuned code may be slightly different from original one

Branch Prediction (1/8)

- **Control Dependency**
  - Target of branch instruction (& its successors) depends on the branch.
  - Control hazard makes pipeline stalled during fetch and decode of branch target which takes place after calculation of branch address and direction.



Branch Prediction (2/8)

- **Predicting Branch Address**
  - Correspondence of (address of) branch instruction and target address is usually invariant (if taken).
    `0x124C:if(<)goto xloop(=PC-94=0x11E0)`
  - Branch Target Buffer (BTB)
    - A kind of cache to keep (inexact) correspondence of branch instruction address and target address.
    - On the fetch of (branch) instruction, simultaneously lookup BTB to get "maybe target address".
    - ➔ Branch latency = 1 if success.
    - BTB@KU: 4K(??) entry (+ for indirect branches)
  - Return Address Stack (RAS)
    - Stack to predict return address:
      push by call / pop by return
    - RAS@KU: 16(??) entry

Branch Prediction (3/8)

- **Predicting Branch Direction**
  - Directions of a conditional branch (are expected to) have temporal locality
    e.g. loop terminator: #taken=*n*−1 / #not-taken=1
  - Simple prediction mechanism = 1-bit predictor
    - 1-bit array indexed by a segment of branch inst address
    - Record 0 (not-taken) / 1 (taken) for each branch inst
      ➔predict (not) taken if it was (not) taken last time
      ➔sensitive to noise (e.g. miss twice for loop terminator)
  - 2-bit saturate-counter predictor
    - 2-bit array indexed by a segment of branch inst address
    - not-taken ➔ −1 / taken ➔ +1 (min=0 / max=3)
    - 0 or 1 ➔ predict as not-taken ／ 2 or 3 ➔ predict as taken
      ➔one miss does not change its mind
      (e.g. #miss of loop terminator = 1)

Branch Prediction (4/8)

- **1bit vs 2bit**

```
for(y=0;y<4;y++) for(x=0;x<4;j++)
  un[y][x]=u[y][x]+dthh*(...);
```

x	actual	prediction 1bit	prediction 2bit
1	T	N	c=2 ➔ T
2	T	T	c=3 ➔ T
3	T	T	c=3 ➔ T
4	N	T	c=3 ➔ T

T : taken
N : not-taken

accuracy = 50%
accuracy = 75%

Branch Prediction (5/8)

- **Very Dangerous Code**

```
for(i=0;i<N;i++) {
  if(i&1){for odd i}
  else    {for even i}
}
```

accuracy for TNTNTN... = 1/2 at most, 0 at worst

➔
```
for(i=0;i<N;i+=2) {for even i}
for(i=1;i<N;i+=2) {for odd i}
```

or

```
for(i=0;i<N;i+=2) {
  {for even i}
  {for odd i+1}
}
```

recode as them if possible

Branch Prediction (6/8)

- **Using Global history (GH)**
  - 2-bit prediction is too local
    - spatial: only aware of a particular branch
    - temporal: remember almost nothing of actions except for last four
  - GH: remember all directions of last *n* branches (e.g. *n* = 8)
  - Mix GH into index of 2-bit predictor
    - gselect (64K)
      `pred_table[gh][branch_inst_addr&0xFF]`
    - gshare (64K)
      `pred_table[gh^(branch_inst_addr&0xFFFF)];`

in KU, gshare or its variant is used (*n* =32?)

Branch Prediction (7/8)

- **Effect of GH (1/2)**

```
for(t=0;t<tmax;t++) {
  for(y=0;y<4;y++) for(x=0;x<4;j++)
    un[y][x]=u[y][x]+dthh*(...);
  tmp=un; un=u; u=tmp;
}
```

```
for(t):                    T                    T
for(y):    T    T    T    N    T    T    T    N
for(x):TTTN TTTN TTTN TTTN  TTTN TTTN TTTN TTTN
```

y==0					y!=0				
x	GH	for(x)	pred	actual	x	GH	for(x)	pred	actual
1	TTTNNT	TTT...	T	T	1	TTTTNT	TTT...	T	T
2	TTNNTT	TTT...	T	T	2	TTTNTT	TTT...	T	T
3	TNNTTT	TTT...	T	T	3	TTNTTT	TTT...	T	T
4	NNTTTT	NNN...	N	N	4	TNTTTT	NNN...	N	N

Branch Prediction (8/8)

- **Effect of GH (2/2)**

```
for(p=h;p!=NULL;p=p->next){
  if (p->data==x) break;
}
if (p==NULL) ...
```

  - Truth value of p==NULL
    - last branch is for p->data==x (true) ➔ false
    - last branch is for p!=NULL (false) ➔ true
    - ➔ predictable as GH=?????T ➔ false / ?????N ➔ true
- if(i&1){...}else{...} is also predictable but ...

Predicated Instruction (1/2)

- **Conditional Branch & SIMD (1/2)**

```
for(i=0;i<n;i++){
  if (a[i]<0) a[i]=-a[i];  /* a[i]=abs(a[i]);
}
```

definitely correct transformation

```
for(i=0;i<n;i+=4){
  if (a[i+0]<0) a[i+0]=-a[i+0];
  if (a[i+1]<0) a[i+1]=-a[i+1];
  if (a[i+2]<0) a[i+2]=-a[i+2];
  if (a[i+3]<0) a[i+3]=-a[i+3];
}
```

How SIMD-vectorize these branches??

SIMD-vectorizable of course

Predicated Instruction (1/2)

- **Conditional Branch & SIMD (1/2)**

```
for(i=0;i<n;i++){
  if (a[i]<0) a[i]=-a[i];  /* a[i]=abs(a[i]);
}
```

```
for(i=0;i<n;i+=4){
  int c0=a[i+0]<0;  double t0=-a[i+0];
  int c1=a[i+1]<0;  double t1=-a[i+1];
  int c2=a[i+2]<0;  double t2=-a[i+2];
  int c3=a[i+3]<0;  double t3=-a[i+3];

  a[i+0]= c0 ? t0 : a[i+0];
  a[i+1]= c1 ? t1 : a[i+1];
  a[i+2]= c2 ? t2 : a[i+2];
  a[i+3]= c3 ? t3 : a[i+3];
}
```

- transform latency from branch to arithmetic
- no misprediction because not predicting
➔ efficient (usually)

conditional move

Basics of Cache (1/7)

- **Overview of Cache (≠ Cash)**
  - High-speed memory to fill the access speed gap between CPU and main memory

    CPU : cache : memory = 1 : 1~20 : 50~100
  - Keep data to be likely accessed in near future

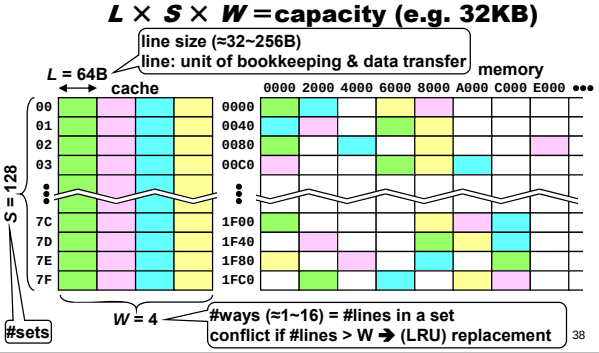
    ➔exploiting/expecting locality
    - temporal: repeated access to data accessed recently
    - spatial: access to data proximate to recently accessed data

    ➔accesses against expectation result in very poor performance
    - e.g. skipping accesses in a huge array

Basics of Cache (2/7)

- **Configuration Parameter:**

  $L \times S \times W$ = capacity (e.g. 32KB)



Basics of Cache (3/7)

- **L > 8B (= `sizeof(double)`)**
  - Exploiting/expecting spatial locality
    - access to `u[x]` ➔ accesses to `u[x+1]`, `u[x+2]`, ...
  - Efficient transfer from/to memory by contiguous access
    - latency = 50~100cycle, bandwidth = 5~10B/cycle

miss rate = $8/L$

```
➔ for(y=0;y<ny;y++) for(x=0;x<nx;x++)      ← better
    un[y][x]=u[y][x]*dthh*(...);
  for(x=0;x<nx;x++) for(y=0;y<ny;y++)      ← worse
    un[y][x]=u[y][x]*dthh*(...);

  for(i=0;i<N;i++){ j=f(i);
    obj[j].x+=obj[j].vx;  obj[j].y+=obj[j].vy;   ← better
  }
  for(i=0;i<N;i++){ j=f(i);
    obj.x[j]+=obj.vx[j];  obj.y[j]+=obj.vy[j];   ← worse
  }
```


Basics of Cache (4/7)

- **$L = 2^l$, $S = 2^s$ ➔ $L \times S = 2^{l+s}$**
  - Data pair conflict with each other if their address difference is $2^{(l+s)}n$.

    ➔Arrays of power-of-two size are dangerous.

```
▪ #define N 1024
  double u[N][N],un[N][N];
  ...
  for(x=1;x<N-1;x++)
    un[y][x]=u[y][x]+dthh*(u[y][x+1]+u[y][x-1]+
                           u[y+1][x]+u[y-1][x]+4*u[y][x]);
▪ struct {double x[N],y[N],z[N],
                 vx[N],vy[N],vz[N];} obj;
  ...
  for(i=0;i<N;i++){
    obj.x[i]+=obj.vx[i];  obj.y[i]+=obj.vy[i];
    obj.z[i]+=obj.vz[i];
  }
```

conflict with each other
$L \times S < 8192$ & $W < 4$
➔ misses always

$L \times S < 8192$ & $W < 6$
➔ misses always

Basics of Cache (5/7)

- **3C of Cache Misses**
  - Compulsory miss
    - data is not in cache on its first access

      ➔inevitable
  - Capacity miss
    - #-of lines accessed after last access > capacity

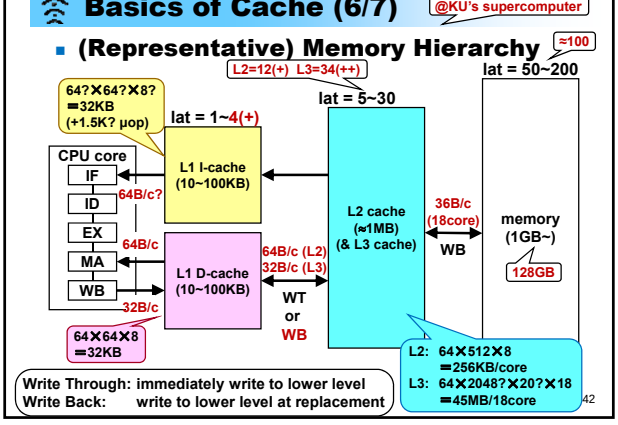
      ←poor temporal (and/or spatial) locality
  - Conflict miss
    - #-of competing lines accessed after last access > #ways

      ←e.g. discontiguous accesses (esp. with 2^n stride)

Basics of Cache (6/7)

- **(Representative) Memory Hierarchy**

@KU's supercomputer

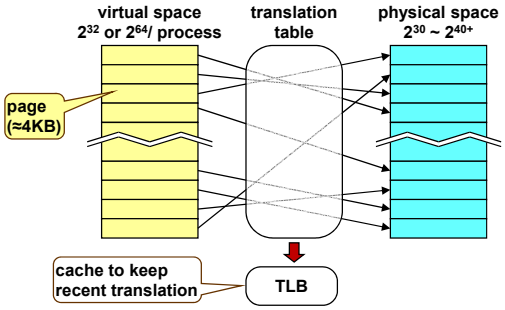


Basics of Cache (7/7)

- **Prefetch = Load a line to cache if the line is expected to be accessed in near future**
  - Hardware Prefetch
    = prefetch based on detection of particular access pattern
    e.g. in KU's supercomputer
    - contiguous access to L1D➔prefetch next line
    - last k addresses of a load = b + s × i (0 ≤ i < k)
      ➔prefetch line including b + s × k to L1D
    - a 64B-line in L2 is loaded to L1
      ➔prefetch next line to L2 if absent
    - watch 32(?) contiguous access streams
      ➔if L1 require a line in a stream, prefetch next line to L2/L3
  - Software Prefetch
    - prefetch a line to L1 by a dedicated instruction
    - compiler inserts prefetch instructions estimating access patterns and effectiveness

TLB (1/2)

- **Virtual/Physical Address Translation**



TLB (2/2)

- **Translation Look-aside Buffer**
  - Cache of PA translated from VA
    - key = virtual page number (high-order bits of VA)
    - data = physical page number (high-order bits of PA)
  - Relatively small ≈ 10~1000 entries
    - @KU: L1-ITLB = 32 × 4way = 128 (+α)
      L1-DTLB = 16 × 4way = 64 (+α+β)
      L2-TLB = 256 × 6way = 1536 (+α)
  - miss ➔ page table lookup ➔ large overhead
    ➔many accesses with long stride ➔ miserable

```
for(x=0;x<1000;x++) for(y=0;y<1000;y++)
  un[y][x]=u[y][x]+dthh*(...);
```

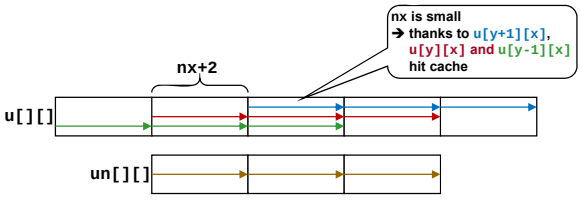
2000 pages are accessed in inner loop ➔ 2 misses / iteration

- **L2/L3 cache is accessed by PA**
  ➔contiguous in VA ≠ contiguous in L2/L3

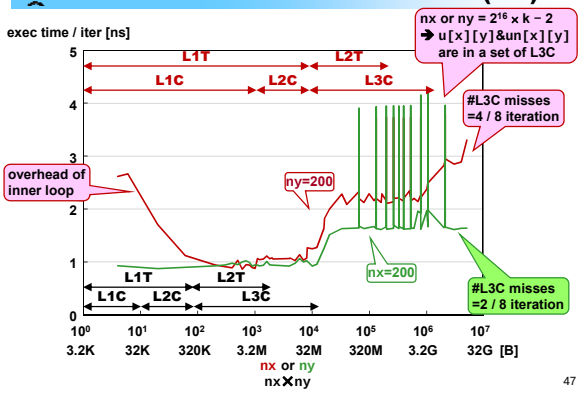
Performance with Cache & TLB (1/7)

- **Memory Access Pattern**

```
for(y=0;y<ny;y++) for(x=0;x<nx;x++)
  un[y][x]=u[y][x]+
    dthh*(u[y][x+1]+u[y][x-1]+
          u[y+1][x]+u[y-1][x]-4*u[y][x]);
```



Performance with Cache & TLB (2/7)

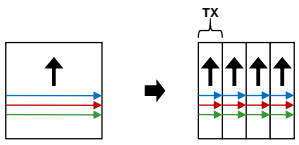


Performance with Cache & TLB (3/7)

- **Optimization for Large nx : x-dim Tiling**

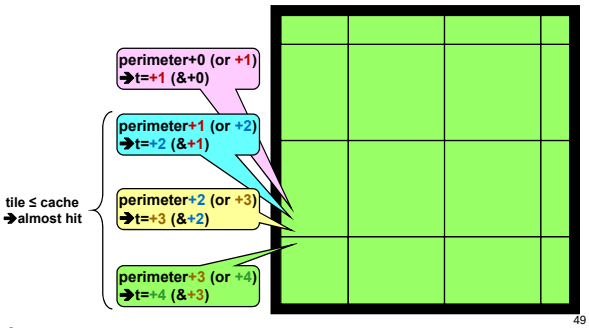
```
for(xx=0;xx<nx;xx+=TX) {
  int nxx=(xx+TX<nx)?xx+TX:nx
  for(y=0;y<ny;y++) for(x=xx;x<nxx;x++) {
    uun[y][x]=uu[y][x]+
      dthh*(uu[y][x+1]+uu[y][x-1]+
            uu[y+1][x]+uu[y-1][x]-4*uu[y][x]);
}
```

- large nx: #misses / 8-iteration = 4 ➔ 2

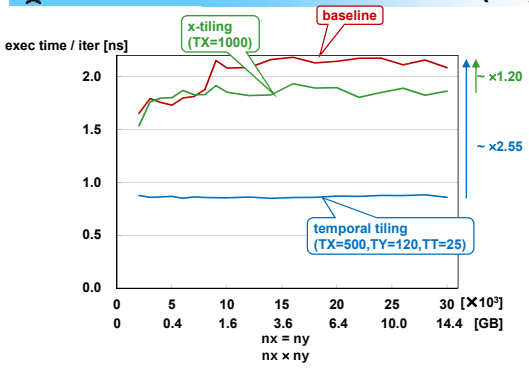


Performance with Cache & TLB (4/7)

- **Super Optimization = Temporal Tiling**



Performance with Cache & TLB (5/7)



Performance with Cache & TLB (6/7)

- **Contiguous vs Discontiguous Accesses**

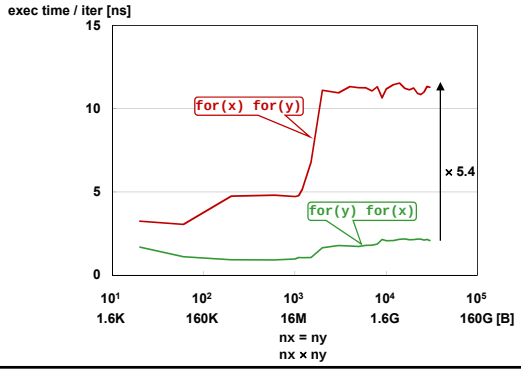
```
for(y=0;y<ny;y++) for(x=0;x<nx;x++)
  uun[y][x]=uu[y][x]+
    dthh*(uu[y][x+1]+uu[y][x-1]+
          uu[y+1][x]+uu[y-1][x]-4*uu[y][x])
```

  - #cache-misses = 2~4 / 8-iteration
    #TLB-misses = 2~4 / 256K-iteration

```
for(x=0;x<nx;x++) for(y=0;y<ny;y++)
  uun[y][x]=uu[y][x]+
    dthh*(uu[y][x+1]+uu[y][x-1]+
          uu[y+1][x]+uu[y-1][x]-4*uu[y][x])
```

  - #cache-misses = 2.25 / 1-iteration
    #TLB-misses = 2 / 1-iteration

Performance with Cache & TLB (7/7)



Conclusion: Key Issues of HPC

- **Pipeline / ILP / out-of-order / SIMD**
  - Relying on compiler/hardware is almost safe.
  - Remember `restrict` in C programming
- **Branch Prediction / Speculation**
  - Loops with many iterations are usually OK.
  - Some `if-else` in loops are really dangerous.
- **Cache / TLB**
  - Dominant performance factor.
  - Don't let down your stupid hardware.
  - Small care can improve performance significantly.
    - As much contiguous as possible.
    - Avoid access conflict if possible.