Superscalar Processors & Thread-Level Parallelism

Instructor:
Alvin R. Lebeck
Some slides from Randy Bryant and Dave O’Hallaron and G. Kedem

Today

- Instruction Level Parallelism
- Parallel Computing Hardware
  - Multicore
  - Multiple separate processors on single chip
- Hyperthreading
  - Replicated instruction execution hardware in each processor
  - Maintaining cache coherence (consistency)
- Thread Level Parallelism
  - Splitting program into independent tasks
    - Example: Parallel summation
    - Some performance artifacts
  - Divide-and conquer parallelism
    - Example: Parallel quicksort
Pipeline Stages

- **Fetch**
  - Select current PC
  - Read instruction
  - Compute incremented PC
- **Decode**
  - Read program registers
- **Execute**
  - Operate ALU
- **Memory**
  - Read or write data memory
- **Write Back**
  - Update register file

The Five Stages of mrmovl

- **Ifetch**: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- **Reg/Dec**: Registers Fetch and Instruction Decode
- **Exec**: Calculate the memory address
- **Mem**: Read the data from the Data Memory
- **WrB**: Write the data back to the register file
Key Ideas Behind Instruction Execution Pipelining

- Overlap execution of instructions
- The `mrmovl` instruction has 5 stages: I-fetch, Reg-Fetch / I-Decode, Execute, Memory-Access, Register Write-Back.

  - Five independent functional units to work on each stage
    - Each functional unit is used only once
  - The 2nd `mrmovl` can start as soon as the 1st finishes its `Ifetch` stage
  - Each `mrmovl` still takes five cycles to complete. latency is still 5 cycles
  - The throughput is much higher; CPI is 1 with ~1/5 cycle time.
  - Instructions start before the previous ones are completed.

---

Single Cycle, Multiple Cycle, vs. Pipeline

**Single Cycle Implementation:**
- Cycle 1: Load
- Cycle 2: Store
- Cycle 3: Waste

**Multiple Cycle Implementation:**
- Cycle 1: Ifetch
- Cycle 2: Reg
- Cycle 3: Exec
- Cycle 4: Mem
- Cycle 5: Wr
- Cycle 6: Ifetch
- Cycle 7: Reg
- Cycle 8: Exec
- Cycle 9: Mem
- Cycle 10: Ifetch

**Pipeline Implementation:**
- Ifetch
- Reg
- Exec
- Mem
- Wr
- R-type
- Ifetch
- Reg
- Exec
- Mem
- Wr
Pipelining Summary

- Most modern processors use pipelining
- Pipelining creates more headaches for exceptions, etc...
- Bypassing/Forwarding for dependent instructions

- Remember the CPU Time Equation
  - Execution Time = Inst/Prog * Cycles/Inst * Seconds/Cycle

- What if instructions are independent?

Superscalar Processors

- Key idea: execute more than one instruction per cycle
- Pipelining exploits parallelism in the “stages” of instruction execution
- Superscalar exploits parallelism of independent instructions
- Example Code:
  - `sub $2, $1, $3`
  - `and $12, $3, $5`
  - `or $13, $6, $2`
  - `add $3, $3, $2`
  - `sw $15, 100($2)`

- Superscalar Execution
  - `sub $2, $1, $3 and $12, $3, $5`
  - `or $13, $6, $2 add $3, $3, $2`
  - `sw $15, 100($2)`
Superscalar Processors

- **Key Challenge:** Finding the independent instructions
  - Instruction level parallelism (ILP)
- **Option 1:** Compiler
  - Static scheduling (Alpha 21064, 21164; UltraSPARC I, II; Pentium)
- **Option 2:** Hardware
  - Dynamic Scheduling (Alpha 21264; PowerPC; Pentium Pro, 3, 4)
  - Out-of-order instruction processing

Instruction Level Parallelism

- **Problems:**
  - Program structure: branch every 4-8 instructions
  - Limited number of registers
- **Static scheduling:** compiler must find and move instructions from other basic blocks
- **Dynamic scheduling:** Hardware creates a big “window” to look for independent instructions
  - Must know branch directions before branch is executed!
  - Determines true dependencies.

Example Code: (assume first operand is destination)

```
sub $2, $1, $3
and $12, $3, $2
or $2, $6, $4
add $3, $3, $2
sw $15, 100($2)
```
Exposing Instruction Level Parallelism

- **Branch prediction**
  - Hardware can remember if branch was taken
  - Next time it sees the branch it uses this to predict outcome

- **Register renaming**
  - Indirection! The CS solution to almost everything
  - During decode, map register name to real register location
  - New location allocated when new value is written to reg.

Example Code:
```
sub $2, $1, $3  # writes $2 = $p1
and $12, $3, $2  # reads $p1
or $2, $6, $4  # writes $2 = $p3
add $3, $3, $2  # reads $p3
sw $15, 100($2)  # reads $p3
```

Additional Notes

- All Modern CPUs use pipelines.
- Many CPUs have 8-12 pipeline stages.
- The latest generation processors (Pentium-4, PowerPC G4, SUN/Oracle SPARC) use multiple pipelines to get higher speed (Superscalar design).
- Now, Parallel Architectures...
**Multicore Processor**

- Intel Nehalem Processor
  - E.g., Shark machines
  - Multiple processors operating with coherent view of memory

**Memory Consistency**

- What are the possible values printed?
  - Depends on memory consistency model
  - Abstract model of how hardware handles concurrent accesses
- Sequential consistency
  - Overall effect consistent with each individual thread
  - Otherwise, arbitrary interleaving
**Sequential Consistency Example**

```plaintext
int a = 1;
int b = 100;
```

Thread1:
- `Wa`: a = 2;
- `Rb`: print(b);

Thread2:
- `Wb`: b = 200;
- `Ra`: print(a);

Thread consistency constraints:
- `Wa` —> `Rb` (100, 2)
- `Wb` —> `Ra` (200, 2)
- `Ra` —> `Rb` (2, 200)
- `Wa` —> `Rb` (1, 200)

**Impossible outputs**
- 100, 1 and 1, 100
- Would require reaching both Ra and Rb before Wa and Wb

**Non-Coherent Cache Scenario**

**Write-back caches, without coordination between them**

```
int a = 1;
int b = 100;
```

Thread1:
- `Wa`: a = 2;
- `Rb`: print(b);

Thread2:
- `Wb`: b = 200;
- `Ra`: print(a);

Print 1

Print 100
Snoopy Caches

- Tag each cache block with state
  - Invalid: Cannot use value
  - Shared: Readable copy
  - Exclusive: Writeable copy

Main Memory
- a: 1
- b: 100

Thread 1
- Cache:
  - State: E
  - Value:
    - a: 2

Thread 2
- Cache:
  - State: E
  - Value:
    - b: 200

Thread 1:
- Wa: a = 2;
- Rb: print(b);

Thread 2:
- Wb: b = 200;
- Ra: print(a);

When cache sees request for one of its E-tagged blocks:
- Supply value from cache
- Set tag to S

print 2
print 200
Out-of-Order Processor Structure

- Instruction control dynamically converts program into stream of operations
- Operations mapped onto functional units to execute in parallel

Hyperthreading

- Replicate enough instruction control to process K instruction streams
- K copies of all registers
- Share functional units
Summary: Creating Parallel Machines

- **Multicore**
  - Separate instruction logic and functional units
  - Some shared, some private caches
  - Must implement cache coherency
- **Hyperthreading**
  - Also called “simultaneous multithreading”
  - Separate program state
  - Shared functional units & caches
  - No special control needed for coherency
- **Combining**
  - Core i7: 4 cores, each with 2-way hyperthreading
  - Theoretical speedup of 8X
    - Never achieved in our benchmarks

Summation Example

- **Sum numbers 0, ..., N-1**
  - Should add up to (N-1)*N/2
- **Partition into K ranges**
  - ⌊N/K⌋ values each
  - Accumulate leftover values serially
- **Method #1: All threads update single global variable**
  - 1A: No synchronization
  - 1B: Synchronize with pthread semaphore
  - 1C: Synchronize with pthread mutex
    - “Binary” semaphore. Only values 0 & 1
Accumulating in Single Global Variable: Declarations

typedef unsigned long data_t;
/* Single accumulator */
volatile data_t global_sum;

/* Mutex & semaphore for global sum */
sem_t semaphore;
pthread_mutex_t mutex;

/* Number of elements summed by each thread */
size_t nelems_per_thread;

/* Keep track of thread IDs */
pthread_t tid[MAXTHREADS];
/* Identify each thread */
int myid[MAXTHREADS];

Accumulating in Single Global Variable: Operation

nelems_per_thread = nelems / nthreads;
/* Set global value */
global_sum = 0;

/* Create threads and wait for them to finish */
for (i = 0; i < nthreads; i++) {
    myid[i] = i;
    Pthread_create(&tid[i], NULL, thread_fun, &myid[i]);
}
for (i = 0; i < nthreads; i++)
    Pthread_join(tid[i], NULL);

result = global_sum;
/* Add leftover elements */
for (e = nthreads * nelems_per_thread; e < nelems; e++)
    result += e;
Thread Function: No Synchronization

```c
void *sum_race(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;
    for (i = start; i < end; i++) {
        global_sum += i;
    }
    return NULL;
}
```

Unsynchronized Performance

- \( N = 2^{30} \)
- Best speedup = 2.86X
- Gets wrong answer when > 1 thread!
Thread Function: Semaphore / Mutex

Semaphore

```c
void *sum_sem(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

    for (i = start; i < end; i++) {
        sem_wait(&semaphore);
        global_sum += i;
        sem_post(&semaphore);
    }
    return NULL;
}
```

Mutex

```c
pthread_mutex_lock(&mutex);
global_sum += i;
pthread_mutex_unlock(&mutex);
```

Semaphore / Mutex Performance

- Terrible Performance
  - 2.5 seconds ➔ ~10 minutes
- Mutex 3X faster than semaphore
- Clearly, neither is successful
Separate Accumulation

- Method #2: Each thread accumulates into separate variable
  - 2A: Accumulate in contiguous array elements
  - 2B: Accumulate in spaced-apart array elements
  - 2C: Accumulate in registers

```c
/* Partial sum computed by each thread */
data_t psum[MAXTHREADS*MAXSPACING];
/* Spacing between accumulators */
size_t spacing = 1;
```

Separate Accumulation: Operation

```c
nelems_per_thread = nelems / nthreads;

/* Create threads and wait for them to finish */
for (i = 0; i < nthreads; i++) {
    myid[i] = i;
    psum[i*spacing] = 0;
    Pthread_create(&tid[i], NULL, thread_fun, &myid[i]);
}
for (i = 0; i < nthreads; i++)
    Pthread_join(tid[i], NULL);

result = 0;
/* Add up the partial sums computed by each thread */
for (i = 0; i < nthreads; i++)
    result += psum[i*spacing];
/* Add leftover elements */
for (i = nthreads * nelems_per_thread; i < nelems; i++)
    result += data[i];
```
Thread Function: Memory Accumulation

```c
void *sum_global(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

    size_t index = myid*spacing;
    psum[index] = 0;
    for (i = start; i < end; i++) {
        psum[index] += i;
    }
    return NULL;
}
```

Memory Accumulation Performance

- Clear threading advantage
  - Adjacent speedup: 5 X
  - Spaced-apart speedup: 13.3 X (Only observed speedup > 8)
- Why does spacing the accumulators apart matter?
False Sharing

- Coherency maintained on cache blocks
- To update psum[i], thread i must have exclusive access
  - Threads sharing common cache block will keep fighting each other for access to block

False Sharing Performance

- Best spaced-apart performance 2.8 X better than best adjacent
- Demonstrates cache block size = 64
  - 8-byte values
  - No benefit increasing spacing beyond 8
Thread Function: Register Accumulation

```c
void *sum_local(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;
    size_t index = myid * spacing;
    data_t sum = 0;
    for (i = start; i < end; i++) {
        sum += i;
    }
    psum[index] = sum; return NULL;
}
```

Register Accumulation Performance

- Clear threading advantage
  - Speedup = 7.5 X
- 2X better than fastest memory accumulation
A More Interesting Example

- Sort set of N random numbers
- Multiple possible algorithms
  - Use parallel version of quicksort
- Sequential quicksort of set of values X
  - Choose “pivot” p from X
  - Rearrange X into
    - L: Values ≤ p
    - R: Values ≥ p
  - Recursively sort L to get L’
  - Recursively sort R to get R’
  - Return L’ : p : R’

Sequential Quicksort Visualized

```
X

p

L    p    R

p2

L2  p2  R2

: :

L’
```
Sequential Quicksort Visualized

void qsort_serial(data_t *base, size_t nele) {
    if (nele <= 1)
        return;
    if (nele == 2) {
        if (base[0] > base[1])
            swap(base, base+1);
        return;
    }
    /* Partition returns index of pivot */
    size_t m = partition(base, nele);
    if (m > 1)
        qsort_serial(base, m);
    if (nele-1 > m+1)
        qsort_serial(base+m+1, nele-m-1);
}

- Sort nele elements starting at base
  - Recursively sort L or R if has more than one element
Parallel Quicksort

- Parallel quicksort of set of values $X$
  - If $N \leq N_{\text{thresh}}$, do sequential quicksort
  - Else
    - Choose “pivot” $p$ from $X$
    - Rearrange $X$ into
      - $L$: Values $\leq p$
      - $R$: Values $\geq p$
    - Recursively spawn separate threads
      - Sort $L$ to get $L'$
      - Sort $R$ to get $R'$
    - Return $L': p: R'$

- Degree of parallelism
  - Top-level partition: none
  - Second-level partition: $2X$
  - ...

Parallel Quicksort Visualized

```
X

p

L p R

L2 p2 R2 p L3 p3 R3

L' p R'
```
Parallel Quicksort Data Structures

/* Structure that defines sorting task */
typedef struct {
    data_t *base;
    size_t nele;
    pthread_t tid;
} sort_task_t;

volatile int ntasks = 0;
volatile int ctasks = 0;
sort_task_t **tasks = NULL;
sem_t tmutex;

- Data associated with each sorting task
  - base:  Array start
  - nele:  Number of elements
  - tid:  Thread ID

- Generate list of tasks
  - Must protect by mutex

Parallel Quicksort Initialization

static void init_task(size_t nele) {
    ctasks = 64;
    tasks = (sort_task_t **) Alloc(ctasks, sizeof(sort_task_t *));
    ntasks = 0;
    Sem_init(&tmutex, 0, 1);
    nele_max_serial = nele / serial_fraction;
}

- Task queue dynamically allocated
- Set Nthresh = N/F:
  - N  Total number of elements
  - F  Serial fraction
    - Fraction of total size at which shift to sequential quicksort
Parallel Quicksort: Accessing Task Queue

```c
static sort_task_t *new_task(data_t *base, size_t nele) {
    P(&tmutex);
    if (ntasks == ctasks) {
        ctasks *= 2;
        tasks = (sort_task_t **) Realloc(tasks, ctasks * sizeof(sort_task_t *));
    }
    int idx = ntasks++;
    sort_task_t *t = (sort_task_t *) Malloc(sizeof(sort_task_t));
    tasks[idx] = t;
    V(&tmutex);
    t->base = base;
    t->nele = nele;
    t->tid = (pthread_t) 0;
    return t;
}
```

- Dynamically expand by doubling queue length
  - Generate task structure dynamically (consumed when reap thread)
- Must protect all accesses to queue & ntasks by mutex

Parallel Quicksort: Top-Level Function

```c
void tqsort(data_t *base, size_t nele) {
    int i;
    init_task(nele);
    tqsort_helper(base, nele);
    for (i = 0; i < get_ntasks(); i++) {
        P(&tmutex);
        sort_task_t *t = tasks[i];
        V(&tmutex);
        Pthread_join(t->tid, NULL);
        free((void *) t);
    }
}
```

- Actual sorting done by tqsort_helper
- Must reap all of the spawned threads
  - All accesses to task queue & ntasks guarded by mutex
Parallel Quicksort: Recursive function

```c
void tqsort_helper(data_t *base, size_t nele) {
    if (nele <= nele_max_serial) {
        /* Use sequential sort */
        qsort_serial(base, nele);
        return;
    }
    sort_task_t *t = new_task(base, nele);
    Pthread_create(&t->tid, NULL, sort_thread, (void *) t);
}
```

- If below Nthresh, call sequential quicksort
- Otherwise create sorting task

Parallel Quicksort: Sorting Task Function

```c
static void *sort_thread(void *vargp) {
    sort_task_t *t = (sort_task_t *) vargp;
    data_t *base = t->base;
    size_t nele = t->nele;
    size_t m = partition(base, nele);
    if (m > 1)
        tqsort_helper(base, m);
    if (nele-1 > m+1)
        tqsort_helper(base+m+1, nele-m-1);
    return NULL;
}
```

- Same idea as sequential quicksort
- Sort 2^37 (134,217,728) random values
- Best speedup = 6.84X

- Good performance over wide range of fraction values
  - F too small: Not enough parallelism
  - F too large: Thread overhead + run out of thread memory
Implementation Subtleties

- Task set data structure
  - Array of structs
    ```c
    sort_task_t *tasks;
    ```
    - new_task returns pointer or integer index
  - Array of pointers to structs
    ```c
    sort_task_t **tasks;
    ```
    - new_task dynamically allocates struct and returns pointer

- Reaping threads
  - Can we be sure the program won’t terminate prematurely?

Amdahl’s Law

- Overall problem
  - $T$  Total time required
  - $f$  Fraction of total that can be sped up ($0 \leq f \leq 1$)
  - $s$  Speedup factor

- Resulting Performance
  - $T_p = \frac{fT}{s} + (1-f)T$
    - Portion which can be sped up runs $k$ times faster
    - Portion which cannot be sped up stays the same
  - Maximum possible speedup
    - $k = \infty$
    - $T_{\infty} = (1-f)T$
Amdahl’s Law Example

- **Overall problem**
  - $T = 10$  Total time required
  - $f = 0.9$  Fraction of total which can be sped up
  - $s = 9$  Speedup factor

- **Resulting Performance**
  - $T_p = 0.9 \times 10/9 + 0.1 \times 10 = 1.0 + 1.0 = 2.0$
  - Maximum possible speedup
    - $T_m = 0.1 \times 10.0 = 1.0$

Amdahl’s Law & Parallel Quicksort

- **Sequential bottleneck**
  - Top-level partition: No speedup
  - Second level: $\leq 2X$ speedup
  - $k^{th}$ level: $\leq 2^{k-1}X$ speedup

- **Implications**
  - Good performance for small-scale parallelism
  - Would need to parallelize partitioning step to get large-scale parallelism
    - Parallel Sorting by Regular Sampling
      - H. Shi & J. Schaeffer, J. Parallel & Distributed Computing, 1992
Lessons Learned

- **Must have strategy**
  - Partition into $K$ independent parts
  - Divide-and-conquer
- **Inner loops must be synchronization free**
  - Synchronization operations very expensive
- **Watch out for hardware artifacts**
  - Sharing and false sharing of global data
- **You can do it!**
  - Achieving modest levels of parallelism is not difficult